etl_script

March 31, 2025

import lib

[1]:
```
import warnings
import pandas as pd
import sqlite3

warnings.filterwarnings('ignore')
```

1 Extract

[2]:
```
# File paths
CUSTOMERS_CSV = "data/sources/customers.csv"
TRANSACTIONS_CSV = "data/sources/transactions.csv"
PRODUCTS_CSV = "data/sources/products.csv"
DB_NAME = "retail_data.db"

# Extract
customers = pd.read_csv(CUSTOMERS_CSV)
transactions = pd.read_csv(TRANSACTIONS_CSV)
products = pd.read_csv(PRODUCTS_CSV)
```

2 Transform

For each data source, the cleaned data frame will set the name `proc_*`

2.1 Customers

[3]:
```
customers["email"].tail()
```

[3]:
```
995              @example.com
996    anneparks@example.org
997       cjones@example.net
998    coreyhill@example.org
999         remember@example
Name: email, dtype: object
```

reference **gmail**'s mail naming convention

A valid email is a set of characters in format `username @ domain` whereas:

- a valid `username` which is:
  - start character must be a letter `(a-z)` or digit `(0-9)`
  - other character must be a letter `(a-z)`, digit `(0-9)` or dot `(.)`
- a valid `domain` which is:
  - start with group of characters only letter `(a-z)` or digit `(0-9)`
  - next character is dot `(.)`
  - end with group of characters only letter `(a-z)` or digit `(0-9)`

row index 995 and 999 at above dataframe has invalid email address

Processing

[4]:
```
# vaild email pattern
pattern = r"^[a-zA-Z0-9][a-zA-Z0-9.]+@[a-zA-Z0-9]+\.([a-zA-Z0-9]+)$"
valid_emails = customers["email"].str.contains(pattern)
# customers with invalid email
invalid_email_customers = customers[~valid_emails]
# check invalid email
invalid_email_customers["email"].head(5)
```

[4]:
```
3         invalid_email@
14    email.example.com
17      machine@example
21    email.example.com
26    agreement@example
Name: email, dtype: object
```

Processed Customer

[5]:
```
# customers with valid email
proc_customers = customers[valid_emails]
```

2.2 Product

[6]:
```
products["category"].unique()
```

[6]:
```
array(['fashion', 'electronics', 'Electronics', 'home', 'Home', 'Fashion'],
      dtype=object)
```

For this data set, we only need to lowercase to make them consistent.

Processed Product

[7]:
```
proc_products = products.copy()
proc_products["category"] = products["category"].apply(lambda x: x.lower())
# check
proc_products["category"].unique()
```

[7]:
```
array(['fashion', 'electronics', 'home'], dtype=object)
```

2.3 Transaction

2.3.1 Deduplicate

[8]:
```
transactions["transaction_id"].duplicated().sum()
```

[8]: 1000

[9]:
```
transactions.duplicated().sum()
```

[9]: 1000

- If we got duplication on whole row contents -> keep only one unique row
- if duplication on `transaction_id` -> source data has problems

In our situation, `drop_duplicates()` is enough

[10]:
```
dedup_transactions = transactions.drop_duplicates()
```

2.3.2 handle invalid transaction dates

[11]:
```
# count how many date are in the format DD/MM/YYYY
transactions_dd_mm_yyyy =␣
↪dedup_transactions[dedup_transactions["transaction_date"].str.
↪contains(r"^\d{2}/\d{2}/\d{4}$")]
print("num transaction in format dd/mm/yyyy is: ", transactions_dd_mm_yyyy.
↪count().max())
```

```
num transaction in format dd/mm/yyyy is:  159872
```

[12]:
```
# count how many date are in the format YYYY-MM-DD
transactions_yyyy_mm_dd =␣
↪dedup_transactions[dedup_transactions["transaction_date"].str.
↪contains(r"^\d{4}-\d{2}-\d{2}$")]
print("num transaction in format yyyy-mm-dd is: ", transactions_yyyy_mm_dd.
↪count().max())
```

```
num transaction in format yyyy-mm-dd is:  32040
```

[13]:
```
# locate the other date format that is not in the two above format
invalid_transactions =␣
↪dedup_transactions[~dedup_transactions["transaction_date"].str.
↪contains(r"(^\d{2}/\d{2}/\d{4}$)|(^\d{4}-\d{2}-\d{2}$)")]
print("invalid_transactions count: ", invalid_transactions.count().max())
```

```
invalid_transactions count:  8088
```

[14]:
```
invalid_transactions['transaction_date']
```

[14]:
```
8              enough
20            million
```

```
23            involve
27             single
29              first
            …
199931        parent
199944    production
199963          most
199980       medical
199992          such
Name: transaction_date, Length: 8088, dtype: object
```

Most of them are text, we can skip it

We dont know for the date that in slash-break format is in `dd/mm/yyyy` or `mm/dd/yyyy`, try using pandas convert datetime function helper, if no `ValueError` exception is raised, our assumption is correct

[15]:
```python
_ = pd.to_datetime(transactions_dd_mm_yyyy["transaction_date"], format="%d/%m/
↪%Y")
```

No error! So the slash-break datetime format is `dd/mm/yyyy` format.

Move forward to have a view on `yyyy-mm-dd`(or `yyyy-dd-mm`) format

[16]:
```python
transactions_yyyy_mm_dd["transaction_date"].head()
```

[16]:
```
3     2023-02-30
4     2025-01-40
5     2023-02-30
13    2024-13-45
21    2024-13-45
Name: transaction_date, dtype: object
```

[17]:
```python
# try to convert the date to datetime, if it fails, return these rows to na␣
↪then compare with original num rows
# try %Y-%m-%d
converted_transaction_yyyy_mm_dd = pd.
↪to_datetime(transactions_yyyy_mm_dd["transaction_date"], format="%Y-%m-%d",␣
↪errors="coerce")
print(converted_transaction_yyyy_mm_dd.isna().count() ==␣
↪transactions_yyyy_mm_dd["transaction_date"].count())
# try %Y-%d-%m
converted_transaction_yyyy_dd_mm = pd.
↪to_datetime(transactions_yyyy_mm_dd["transaction_date"], format="%Y-%d-%m",␣
↪errors="coerce")
print(converted_transaction_yyyy_dd_mm.isna().count() ==␣
↪transactions_yyyy_mm_dd["transaction_date"].count())
```

```
True
True
```

So we know that all of these `%Y-%m-%d` transaction date is invalid format

Final processed transactions:

[18]:
```
# add these invalid transaction date to invalid_transactions df
invalid_transactions = pd.concat([invalid_transactions,␣
 ↪transactions_yyyy_mm_dd])
# valid transactions
proc_transactions = dedup_transactions[~dedup_transactions["transaction_id"].
 ↪isin(invalid_transactions["transaction_id"])]
proc_transactions["transaction_date"] = pd.
 ↪to_datetime(proc_transactions["transaction_date"], format="%d/%m/%Y")  #␣
 ↪convert to datetime
```

3 Load

After processing data, From 3 original dataframe, we got `proc_*` dataframe which is cleaned data.

[19]:
```
# Load the cleaned data into a SQLite database with three tables: customers,␣
 ↪transactions, and products.
conn = sqlite3.connect(DB_NAME)
# write to db
proc_customers.to_sql("customers", conn, if_exists="replace", index=False)
proc_transactions.to_sql("transactions", conn, if_exists="replace", index=False)
proc_products.to_sql("products", conn, if_exists="replace", index=False)
```

[19]: 50000

[20]:
```
pd.read_sql(
  sql="""
  SELECT sql
  FROM sqlite_master
  WHERE type = 'table' AND name = 'transactions'
  """, con=conn).values[0][0]
```

[20]:
```
'CREATE TABLE "transactions" (\n"transaction_id" TEXT,\n  "customer_id" TEXT,\n
"transaction_date" TIMESTAMP,\n  "amount" REAL\n)'
```

4 Data Aggregation

> Write a SQL query to calculate the total transaction amount per customer and save the results in a new table called `customer_revenue`.

[21]:
```
customer_revenue = pd.read_sql(
  sql="""
  SELECT customer_id, sum(amount) as total_transaction_amount
  FROM transactions
  GROUP BY customer_id
```

```
  """, con=conn)

customer_revenue.to_sql("customer_revenue", conn, if_exists="replace",␣
↪index=False)
```

[21]: 1000

[22]:
```
pd.read_sql(
  sql="select * from customer_revenue", con=conn).head()
```

[22]:

	customer_id	total_transaction_amount
0	004454d5-8b28-4675-ac13-ffef982bc471	72340.74
1	009d8fce-6cef-4fb3-b7c3-59ce09c1e58a	73812.20
2	0131b090-2ca1-48f5-bf15-35b3f1923bdd	86753.93
3	014c1663-714f-4ea4-9d3a-9d477224044e	76594.32
4	01bd6b97-54eb-4413-85d8-c78fadf1e6f6	75030.63