Descriptor Matching Project Report

Fluz Internship Assignment

Mohamed Anas Aaffoute
May 11, 2025

Contents

1 Introduction

This project involved mapping raw transaction **descriptors** to a reference **merchant list** using a data engineering pipeline. The descriptors are the text strings appearing on transaction records (often messy or containing extra information), and the goal was to identify which known merchant each descriptor refers to.

The report is structured to reflect my learning process. I begin with understanding the data and setting up the environment, then describe the database schema design and ingestion of data. I detail the descriptor cleaning and fuzzy matching logic (including how I leveraged the *RapidFuzz* library for string matching), and highlight how I utilized AI (ChatGPT) for guidance. I also discuss testing procedures and various challenges I encountered (such as database constraints and false matches) along with the fixes implemented. Finally, I present the results of the matching process (how many descriptors were successfully matched).

2 Data Understanding

At the outset, I analyzed the two primary datasets provided:

- **Descriptors list:** This dataset contained *674* transaction descriptors, each with a unique identifier. A descriptor is a raw text string from a transaction record (for example, `"05/10 WAL-MART #5826"` or `"UBER EATS*MC DONALD'S"`). These strings often include additional information like dates (e.g., `05/10`), store numbers (e.g., `#5826`), or combined names (e.g., an aggregator `UBER EATS` and an actual merchant `MC DONALD'S` separated by an asterisk).

- **Merchant list:** This dataset contained *117* unique merchant names (each with an associated merchant ID). These are the canonical names of merchants (e.g., `WALMART`, `UBER EATS`, `STARBUCKS`, etc.) that we attempt to match the descriptors to. The merchant ID is a unique identifier (provided in the data) for each merchant, which would be used as a foreign key if linking the records.

Understanding the data was crucial. I observed that many descriptors were not an exact match to merchant names due to the extra characters or slight naming differences. For example, a descriptor like `"WAL-MART #5826 05/10"` clearly refers to `"Walmart"` but has a different spelling and includes a date and store number. Another example is `"UBER EATS*MC DONALD'S"` which includes an intermediary (Uber Eats) and an actual restaurant name. According to the project notes, if an intermediary (aggregator) is present, the descriptor should map to that intermediary (so `"UBER EATS*MC DONALD'S"` should map to `"UBER EATS"` rather than McDonald's). These observations meant that a direct one-to-one string comparison would not be sufficient; a combination of cleaning and fuzzy matching would be needed to correctly map descriptors to merchants.

I also anticipated that some descriptors might not correspond to any merchant in the provided list (for instance, a truly unknown merchant or a niche one not listed). Those would remain unmatched and need to be reported for further analysis. Overall, the data understanding phase made it clear that the task required careful text processing and a robust matching strategy.

3 Environment Setup

Before diving into coding the solution, I set up my environment and tools:

- **Programming language and libraries:** I chose Python for data processing libraries. Key libraries used include:
  - `pandas` for data manipulation (reading/writing CSVs, DataFrames for descriptors and merchants).
  - `SQLAlchemy` (with MySQL) for database connectivity and operations.
  - `rapidfuzz` for efficient fuzzy string matching (an optimized alternative to the older FuzzyWuzzy library).
  - Python's built-in `re` (regex) module for descriptor cleaning.
- **Database:** A MySQL database was used to store the data. Using a database ensured that the data could be queried and updated easily, and it reflects a real-world scenario where transaction records and master lists reside in databases.
- **Configuration:** To avoid hardcoding database credentials in the code, I created a separate `config.py` file containing the database connection details (host, port, username, password, database name). The main script imports this config. Listing 1 shows how the configuration is used to build a database connection.

I ensured the above libraries were installed (ChatGPT was helpful in recommending `rapidfuzz` for speed). With this setup, I was ready to proceed with the pipeline implementation.

Listing 1: Loading database credentials from config and connecting to MySQL

```
# config.py (credentials file)
from dotenv import load_dotenv
import os

load_dotenv()

DB_USER = os.getenv("DB_USER")
DB_PASS = os.getenv("DB_PASS")
DB_HOST = os.getenv("DB_HOST", "127.0.0.1")
DB_PORT = os.getenv("DB_PORT", "3306")
DB_NAME = os.getenv("DB_NAME")
```

4 Schema and Ingestion

With the environment ready, I designed the database schema and ingested the data using a script (`db_ingest.py`):

- **Merchant List table:** A table `merchant_list` with columns *merchant_id* (primary key) and *merchant_name*. This holds the 117 known merchants. The provided data already included a unique ID for each merchant, which I used directly. Each merchant name is stored in uppercase for consistency.
- **Descriptors table:** A table `descriptors` with columns *descriptor_id* (primary key) and *descriptor* (the raw descriptor text). The descriptor text is stored as a TEXT field since lengths vary and some descriptors are fairly long. Additional columns (to store matching results) were planned to be added later.

Using the ingestion script `db_ingest.py`, I read the source Excel/CSV file: the merchant list and descriptors were loaded into Pandas DataFrames and then inserted into the database. Initially, I did not include the merchant identifier in the descriptors table because those values were unknown until the matching was performed. In other words, I did *not* set a foreign key on `descriptors.merchant_id` at creation time, to avoid complications. All descriptors were inserted with just their raw text and an ID.

One challenge here was handling the schema evolution. After the initial load, the descriptors table needed extra columns for storing the cleaned descriptor, the matched merchant name, and the matched merchant ID. I addressed this by altering the table schema within the main matching script itself, just before performing matches. This approach ensured that if I re-ran the script, it would add the columns only if they didn't already exist (making the process idempotent). Listing 2 illustrates this step: the script attempts to add each new column and catches the error if the column is already present, ignoring that specific error. This way, re-running the pipeline would not fail due to trying to recreate existing columns.

Listing 2: Idempotent addition of result columns to the descriptors table

```
with engine.begin() as conn:
    for col_sql in (
        "ADD␣COLUMN␣cleaned_descriptor␣TEXT␣NULL␣AFTER␣descriptor",
        "ADD␣COLUMN␣merchant_name␣VARCHAR(255)␣NULL",
        "ADD␣COLUMN␣merchant_id␣VARCHAR(36)␣NULL"
    ):
        try:
            conn.execute(text(f"ALTER␣TABLE␣descriptors␣{col_sql};"
                ))
        except Exception as e:
            # Ignore "column already exists" errors (MySQL error
                code 1060)
            if "1060" not in str(e):
                raise
```

It's worth noting that the provided descriptor dataset had some columns (like a placeholder *merchant_ name*, city, state) that were not used in the matching process. My ingestion logic either omitted those or they remained null, because the purpose was to determine the merchant via our algorithm. The design choice to add *merchant_ name* and *merchant_ id* columns after matching (rather than as a foreign key constraint from the start) proved useful. It allowed unmatched descriptors to have a NULL merchant reference without violating referential integrity, and deferred the linkage until we were confident in the matches.

By the end of ingestion, the database contained all descriptors and merchants. I verified the counts (674 rows in `descriptors`, 117 rows in `merchant_list`) to ensure everything was loaded correctly.

5 Cleaning and Matching

The core of the project was cleaning the raw descriptor strings and then matching them to the merchant list using a combination of exact matching and fuzzy matching. I approached this in a step-by-step manner, refining the method as I encountered new challenges. The final process can be broken into two main parts: descriptor cleaning (standardizing the text) and the matching algorithm (including fuzzy logic).

5.1 Descriptor Cleaning

Raw descriptors often contain irrelevant or inconsistent information that can hinder matching. I developed a cleaning function to normalize descriptors by removing or standardizing these elements:

- **Remove transaction-specific codes:** Many descriptors included date stamps or store/order numbers (e.g., `"05/10"` or `"ORDER #12345"` or numeric IDs like `"00028341"`). These are not part of the merchant name, so they need to be stripped out.
- **Handle aggregators and separators:** If a descriptor contains an asterisk (`*`), it usually separates an intermediary from the merchant (for example, `"PAYPAL *ABC STORE"` or `"UBER EATS*MC DONALD'S"`). In such cases, per the instructions, the part before the `*` (the intermediary) is the one we want to match. I therefore split the string at the asterisk and only kept the first portion.
- **Remove punctuation and unify case:** Punctuation like hyphens or apostrophes can cause mismatches (e.g., `"Wal-Mart"` vs `"Walmart"`). I removed all non-alphanumeric characters and trimmed extra whitespace. I also converted the descriptor to uppercase to ignore case differences.

The cleaning function uses regular expressions to find patterns and replace them with spaces. Listing 3 shows the implementation of this function. Each

regex targets a specific kind of noise: dates, long numeric tokens, and "ORDER #" phrases. After applying all regex substitutions and removing punctuation, the result is uppercased and extra spaces are collapsed. For example, `"05/10 Wal-Mart #5826"` becomes `"WALMART"` after cleaning, and `"Uber Eats*McDonald's 00028341"` becomes `"UBER EATS"`.

Listing 3: Cleaning a raw descriptor string to its core merchant-related text

```
import re

# Precompiled regex patterns for cleaning
_date_rx = re.compile(r"\b\d{2}/\d{2}\b")         # matches dates
    like 05/10
_store  = re.compile(r"\b\d{3,}\b")               # matches long
    numeric strings (store IDs)
_order  = re.compile(r"ORDER\s*#?\s*\d+", re.IGNORECASE)  # matches
     "ORDER #12345"
_punct  = re.compile(r"[^\w\s]")                  # matches any
    punctuation character

def clean_descriptor(raw: str) -> str:
    # 1. Keep text before '*' (if present) to handle cases like "
        AGGREGATOR*MERCHANT"
    core = raw.split("*", 1)[0]
    # 2. Remove dates and numeric codes
    core = _date_rx.sub("␣", core)
    core = _store.sub("␣", core)
    core = _order.sub("␣", core)
    # 3. Remove punctuation
    core = _punct.sub("␣", core)
    # 4. Normalize whitespace and convert to uppercase
    return "␣".join(core.upper().split())
```

After this cleaning step, each descriptor is reduced to a simplified form that should closely resemble a merchant name. However, I noticed that some cleaned descriptors still didn't perfectly match the merchant list entries due to slight naming differences or abbreviations. For example, `"WAL-MART"` would clean to `"WAL MART"` (space instead of hyphen) versus the merchant list having `"WALMART"` (no space), or an entry like `"DICKS SPORTINGGOODS"` vs `"DICK'S SPORTING GOODS"`. To handle these, I introduced an **alias mapping**.

The alias mapping (`aliases.csv`) is a simple two-column file I created, listing common descriptor patterns and their corresponding canonical merchant name. Before attempting fuzzy matching, I apply a substitution: if a cleaned descriptor *starts with* one of the known patterns, I replace that portion with the canonical name. This handles cases like:

- `"WAL MART"` → `"WALMART"` (alias for `"WAL-MART"` or `"WM"` variants).
- `"WM SUPERCENTER"` → `"WALMART"` (some descriptors started with "WM SUPERCENTER" which actually refers to Walmart).
- `"H MART"` (with a space) → `"H MART"` (canonical version without the hyphen, ensuring consistency for the Korean grocery chain vs. any confusion with "Walmart").

- `"DD "` → `"DOORDASH"`, `"GH "` → `"GRUBHUB"` (short codes for food delivery services).
- `"ATT*BILL PAYMENT"` → `"ATT"` (AT&T billing descriptors).

These aliases were derived from patterns I observed in the data and known common abbreviations. I loaded the aliases into a dictionary and then applied a function over the cleaned descriptor field to substitute any prefixes that matched. This step greatly improved matching for those tricky cases that normal cleaning didn't fully normalize.

5.2 Fuzzy Matching Strategy

With descriptors cleaned and normalized as much as possible, the next step was to match them against the list of merchant names. I implemented a multi-step matching strategy:

1. **Exact or substring match:** First, check if the cleaned descriptor contains any merchant name as a substring. If yes, it's an immediate match. For example, if the cleaned descriptor is `"STARBUCKS COFFEE"`, it clearly contains `"STARBUCKS"` which is a merchant name. This step catches obvious cases with no need for fuzzy scoring.

2. **Candidate pool filtering:** If no direct match is found, prepare for fuzzy matching by narrowing down the candidate merchants. Instead of comparing the descriptor against all 117 merchants (which is feasible but can produce odd matches), I filter the merchant list to those that share at least one word (token) with the cleaned descriptor. For instance, if the descriptor tokens are {`"CITY", "MARKET"`}, I will consider merchant names that have either "CITY" or "MARKET" in them. This avoids comparing, say, `"MARKET"` with an unrelated merchant like `"STARBUCKS"`. If the filter yields no candidates (no shared token), I fall back to using the full merchant list just in case.

3. **Fuzzy scoring:** Using the `rapidfuzz` library, I compute similarity scores between the cleaned descriptor and each candidate merchant name. I employed two different scoring methods:

   - `token_set_ratio`: This method compares the sets of tokens in each string, effectively ignoring word order and duplicates. It's good for cases where the descriptor might have extra words not in the merchant name or vice versa.
   - `partial_ratio`: This method looks for the best partial match of the shorter string in the longer string, which is useful if the descriptor contains the full merchant name plus extra terms.

   I take the top match from each of the two scoring methods. Often they agree on the same best match; if they differ, I compare their scores and

choose the higher one as the tentative best match for that descriptor. I also note the score of the second-best match (from the `token_set_ratio` results) for the next step.

4. **Match acceptance criteria:** Not every high-scoring fuzzy match is necessarily correct (some different names can coincidentally score high). I defined criteria to decide if the best match is reliable enough to accept:

   - **High score threshold:** If the best similarity score is ≥ 85, I consider it a strong match and accept it.
   - **Token presence with moderate score:** If the actual merchant name appears as a substring in the cleaned descriptor (for example, the descriptor `"JOE Starbucks"` contains "STARBUCKS") and the score is at least 75, I accept it. The rationale is that the name appearing directly, even if there is other text, is a good indicator.
   - **Score gap criterion:** If the best score is ≥ 70 and also at least 15 points higher than the second-best score, I accept it. This means the top match was significantly better than the rest, indicating a clear winner.

   If none of these conditions are met, I do *not* assign a merchant to the descriptor (marking it as unmatched). These rules were determined through trial and error, aiming to balance sensitivity (catching as many true matches as possible) and precision (avoiding false matches).

I implemented the above logic in the main script, looping through each cleaned descriptor and determining its best match. Listing 4 shows a simplified snippet of the matching algorithm in code form.

Listing 4: Fuzzy matching descriptors to merchant names with acceptance criteria

```
from rapidfuzz import fuzz, process

matches = []
no_matches = []

for _, row in df.iterrows():
    desc_id = row["descriptor_id"]
    raw_text = row["descriptor"]
    cleaned = row["cleaned"]
    # Step 1: Exact or substring match
    exact = next((name for name in name_list if name in cleaned),
        None)
    if exact:
        matches.append((desc_id, cleaned, exact, name_to_id[exact])
            )
        continue

    # Step 2: Limit candidate pool by shared token(s)
    desc_tokens = set(cleaned.split())
```

```
pool = [m for m in name_list if desc_tokens & set(m.split())]
    or name_list

# Step 3: Compute fuzzy similarity scores (token_set_ratio and
    partial_ratio)
best_match, best_score = process.extractOne(cleaned, pool,
    scorer=fuzz.token_set_ratio)[:2]
alt_match, alt_score   = process.extractOne(cleaned, pool,
    scorer=fuzz.partial_ratio)[:2]
if alt_score > best_score:
    best_match, best_score = alt_match, alt_score

# Determine second best score for gap calculation
results = process.extract(cleaned, pool, scorer=fuzz.
    token_set_ratio, limit=2)
second_score = results[1][1] if len(results) > 1 else 0
score_gap = best_score - second_score

# Step 4: Acceptance criteria for match
if best_score >= 85 \
   or (best_match in cleaned and best_score >= 75) \
   or (best_score >= 70 and score_gap >= 15):
    matches.append((desc_id, cleaned, best_match, name_to_id.
        get(best_match)))
else:
    no_matches.append((desc_id, raw_text, cleaned, best_match,
        best_score))
```

In the code above, `name_list` is the list of all merchant names (converted to uppercase) and `name_to_id` is a dictionary mapping merchant name to merchant ID for quick lookup. I populate those by reading the `merchant_list` table at the start of the script. Each descriptor goes through the exact match check, then the filtered fuzzy matching. Depending on whether it meets the acceptance criteria, I append it to either the `matches` list (with the chosen merchant) or the `no_matches` list.

After processing all descriptors, the script performs a batch update on the database. For each confirmed match in `matches`, it updates the corresponding row in the `descriptors` table, setting the *cleaned_ descriptor*, *merchant_ name*, and *merchant_ id* fields. I used parameterized SQL (via SQLAlchemy's text queries) to safely execute these updates. Any descriptor that remained unmatched simply retains NULL in the merchant fields.

Additionally, I configured the script to output the list of unmatched descriptors to a CSV file (`unmatched.csv`) along with their cleaned text, the best guess merchant, and the score. This was extremely useful for debugging and for potentially extending the merchant list later. The script also prints a summary line to the console, e.g., `"Matched 629 / 674 rows (45 unmatched)"`, to give a quick performance statistic.

6 Use of AI (ChatGPT) in the Process

Throughout the project, I used ChatGPT as a supportive tool to enhance my problem-solving. Here are some key areas where AI assistance was valuable:

- **Choosing tools and libraries:** Initially, I was considering using Python's `fuzzywuzzy` library for string matching. After discussing with ChatGPT the need for efficient fuzzy matching, it suggested `rapidfuzz` as a modern, faster alternative. This advice guided me to adopt `rapidfuzz`, which turned out to be very effective.

- **Structuring the matching logic:** I consulted ChatGPT on how to combine multiple fuzzy matching metrics. It helped me outline the approach of using both token-based and partial matching, and even suggested considering the difference between the top two scores to judge match confidence. These insights directly influenced the design of my acceptance criteria (the gap of 15 points rule came from understanding how to avoid ambiguous matches).

- **Regex and data cleaning:** Crafting the right regular expressions for cleaning was iterative. I used ChatGPT to refine a regex for matching date formats and order numbers. For example, my initial attempt at catching `"ORDER #12345"` patterns missed some cases, and ChatGPT helped correct the pattern (`ORDER\s*#?\s*\d+` with case-insensitivity). This saved time and ensured my cleaning function was robust.

- **Debugging and troubleshooting:** When I encountered issues, like an SQLAlchemy error while altering the table or unexpected matches, I described the problem to ChatGPT. It often pointed out possible causes. For instance, I had an issue with duplicate column additions (which I later solved by catching the error). ChatGPT explained the MySQL error code 1060 to me, confirming that it meant a duplicate column, which gave me confidence to handle it as a non-critical exception. In another case, I was worried about why a certain descriptor wasn't matching; by walking through the logic with ChatGPT, I realized I hadn't uppercased the merchant list names initially, causing case mismatches. Fixing that resolved the issue.

- **Improving alias handling:** I shared a few tricky descriptors with ChatGPT (like `"WM SUPERCENTER"` and `"ATT*BILL PAYMENT"`). It agreed that a dictionary of known patterns would help and even gave examples of alias mappings. This validated my plan to use the `aliases.csv` approach and provided a few ideas for entries to include.

Using AI in this way was like having a helpful colleague to brainstorm with. It didn't solve the problem outright, but it provided guidance, suggestions, and explanations that accelerated my learning and development process. Importantly, I always tested and verified any suggestion in my specific context — AI

might suggest something plausible in general, but I learned to adapt it to the particulars of my data and environment.

7 Testing

I tested the pipeline in stages, which helped catch issues early and ensure each component worked correctly:

- **Database checks:** After running the ingestion, I ran simple queries (using SQL or pandas `read_sql`) to confirm that the expected number of records were in the tables and that fields looked correct. For example, I checked that `descriptors` had 674 entries and spot-verified a few entries to see that the text was loaded properly (no truncation or encoding issues).
- **Cleaning function tests:** I wrote a few ad-hoc tests for the `clean_descriptor` function by printing its output for sample inputs. For instance, I tested strings like `"05/10 Walmart #1234"` and `"PayPal *Home Depot 000111"` to ensure the output was `"WALMART"` and `"PAYPAL"` respectively. These manual tests confirmed that the regex patterns were working as intended (dates and numbers removed, etc.). Whenever I noticed an anomaly (e.g., if `"H-Mart"` came out as `"H MART"` with a space, which might still match fine but I wanted consistency), I adjusted the cleaning or alias rules accordingly.
- **Step-by-step matching review:** For a few descriptors, I walked through the matching logic manually. One example was `"POS CITY FRESH MARKET 123"`; I checked how my code tokenized it ({`"POS","CITY","FRESH","MARKET"`}), filtered candidates (likely merchants with "City" or "Market"), and then what fuzz matching would yield. This particular case helped me realize I needed an alias for `"POS CITY FRESH MARKET"` to `"CITY MARKET"`, as the best match was "CITY MARKET" but with a slightly lower score until I added that alias.
- **Reviewing matches and mismatches:** After running the full matching process, I took the output and analyzed both the matched and unmatched results:
  - I scanned through a sample of matched descriptors (now with assigned merchant names in the database) to see if they made sense. For instance, seeing `"STARBUCKS"` assigned to `"STARBUCKS"` is trivial, but I wanted to check borderline cases like `"KAUAI COFFEE"` matched to `"KAUAI COFFEE COMPANY"` (which was correct, and I had an alias to ensure consistency on naming).
  - The `unmatched.csv` was extremely helpful. It listed descriptors that weren't matched along with the best guess the algorithm had and the score. I opened this CSV in a spreadsheet and sorted by score

to see if any high-scoring ones were mistakenly left unmatched. One example: an early run left `"ADIDASONLINESTORE"` unmatched with a best guess of `"ADIDAS"` at around 70

- I also checked for any false positives in the matched list by looking at cases where the score was just at the threshold. If something looked suspicious (for example, if a descriptor for a local cafe somehow matched to `"STARBUCKS"` with 70

- **Multiple run idempotency:** I ran the main matching script multiple times to ensure that it could handle reruns gracefully. Thanks to the schema alteration guarding (Listing 2), running it again would not duplicate columns. Also, the matching logic itself simply recomputed and updated the same fields, so it was effectively idempotent. This gave me confidence that if I adjusted some logic (like adding a new alias or tweaking thresholds) I could rerun the script on the same data without issues.

By iteratively testing and reviewing outputs, I was able to improve the accuracy of the matching step by step. The testing phase often blurred with the development phase — each test taught me something that led to a code improvement, which I would then test again. This trial-and-error loop, while time-consuming, was a great learning experience in understanding how small changes in logic affected the overall results.

8 Challenges and Fixes

During the development of this project, I encountered several challenges. Below I summarize the main issues and how I addressed each one:

- **Foreign key constraint concerns:** I initially considered enforcing a foreign key relationship between the descriptors and merchant tables (so that each matched descriptor must reference a valid merchant ID). However, since most descriptors wouldn't have a merchant ID until after the matching process, this was problematic (inserting rows with a null or temporary ID would violate a strict foreign key). *Fix:* I designed the schema to exclude the foreign key at first, and later added the *merchant_id* column in descriptors after computing matches. I allowed this column to be NULL for unmatched cases. This way, referential integrity can be enforced only on the matched subset, or a foreign key constraint could be added post-matching if desired.

- **Column name mismatches:** The ingestion data had slightly different column naming conventions than I wanted in the database (for example, "merchant name" vs "merchant_name"). Also, the descriptors dataset included a "merchant_name" column that was supposed to be filled by the matching process, but was empty initially. *Fix:* In the ingestion step, I explicitly specified column names to match my schema. I named the

merchant table columns as `merchant_id` and `merchant_name`. For the descriptors, I only imported the ID and descriptor text. Later, when adding the result columns via ALTER TABLE (Listing 2), I handled the potential duplication: if a column like `merchant_name` already existed (perhaps due to the initial import of an empty column), the script caught the error and skipped adding it. This ensured consistency between the code expectations and the actual database schema.

- **Noisy data in descriptors:** As noted earlier, raw descriptors came with dates, numbers, and other noise that would confound matching. Initially, without cleaning, the fuzzy matcher was scoring some completely wrong pairs highly simply because of these extraneous tokens. *Fix:* I implemented the comprehensive cleaning function. Removing irrelevant substrings dramatically improved the quality of matches. For example, before cleaning, a descriptor with a date might falsely match a merchant that happened to have numbers in its name or some substring alignment; after cleaning, such spurious matches disappeared because those tokens were gone.

- **Aggregator vs. merchant confusion:** Some descriptors included an intermediary (like `"UBER EATS"` or `"PAYPAL"`). In early tests, my matcher might pick up the latter part of the descriptor and match the wrong merchant. For instance, `"UBER EATS*MC DONALD'S"` was initially yielding a high score for "MCDONALD'S" (which is in the merchant list) rather than "UBER EATS". This was contrary to the requirement of mapping to the intermediary. *Fix:* I modified the cleaning step to split at `*` and only keep the first part. After this change, "UBER EATS*MC DONALD'S" cleaned to "UBER EATS", which then correctly matched the Uber Eats entry. This was a critical fix to adhere to the expected logic for such cases.

- **Merchant name variations and abbreviations:** Direct string matching was impeded by variations like "Wal-Mart" vs "Walmart", or abbreviations like "WM" for Walmart, "DD" for DoorDash, etc. The fuzzy matching alone could handle some of these, but not always reliably (e.g., "WM Supercenter" might fuzz-match to "Woodman's Market" incorrectly because of the "W" and "M" letters). *Fix:* The introduction of the alias dictionary resolved most of these issues. By standardizing these variants to a canonical form before matching, I essentially taught the algorithm those equivalences. This dramatically improved matches for those patterns. For example, once "WAL-MART" and "WM" were aliased to "WALMART", descriptors containing those cleaned up to "WALMART" exactly, yielding immediate matches.

- **Fuzzy matching false positives:** One significant challenge was ensuring that the fuzzy matcher didn't choose the wrong merchant when descriptors were somewhat ambiguous. For instance, consider a descriptor like `"H MART"` (a Korean grocery chain) versus `"WALMART"`: without precautions, a

fuzzy match might see "H MART" and "WALMART" as similar (because "MART" is common) and mistakenly match them if the score was just above a naive threshold. *Fix:* I took several measures to reduce false positives:

  - Requiring a shared token between descriptor and merchant name (the token overlap filter) means completely unrelated names won't even be compared.
  - The acceptance criteria were tuned to be conservative. For example, in the "H MART" vs "WALMART" scenario, the best score might be around 80. Initially I had a lower threshold that would accept that, but I raised the strict threshold to 85, and also added the condition that if the merchant name isn't actually present in the descriptor, a higher bar is set. This prevented that incorrect match.
  - I also relied on manual review of the `unmatched.csv` and matched results to catch any oddities and then adjusted the logic. Each adjustment (like the gap criterion or token presence rule) was tested again to ensure it improved accuracy.

  After these fixes, the rate of false positives dropped significantly. Practically all accepted matches in the final run were correct upon manual inspection.

- **Iteration and continuous improvement:** This was more of a process challenge. Each time I fixed one issue, I had to be mindful that I might introduce another or miss something else. *Fix:* I adopted an iterative approach: make one change at a time, rerun the matching, and inspect the outcome. For example, after adding a bunch of aliases, I checked that none of the aliases unintentionally misclassified a descriptor (I had to ensure my alias patterns were specific enough not to override legitimate text). By incrementally building up the solution and constantly testing, I gradually converged to a stable and effective pipeline.

These challenges taught me a lot about data quality issues and the importance of flexibility in design. Being able to adapt the schema (like adding columns later) and the logic (tweaking thresholds) was crucial. I also learned the value of defensive programming (e.g., idempotent operations and error handling) to make the pipeline robust.

9 Final Results and Conclusion

After implementing the cleaning and matching pipeline and refining it through testing, I achieved a high match rate on the descriptor dataset. In the final run, the script matched about **629 out of 674 descriptors** to a merchant in the list. This corresponds to roughly 93.3% of descriptors successfully identified. The remaining **45 descriptors (approximately 6.7%)** were left unmatched because

they did not meet the acceptance criteria or likely because their corresponding merchants were not present in our merchant list.

The unmatched descriptors were written to `unmatched.csv` for further analysis. On examining these, many appeared to be niche merchants or variants that we did not have in the merchant list or alias file. For instance, a descriptor might refer to a small local business that wasn't in the provided merchant list. These could be addressed in the future by expanding the merchant database or improving the matching logic to flag potential new merchants.

Overall, the outcome is positive: the majority of transaction descriptors can now be automatically mapped to known merchants, which would be very useful in a real-world setting (for example, in banking or expense categorization applications).

In conclusion, this project was an enlightening exercise in data cleaning, fuzzy matching, and pipeline development. By incrementally building the solution and learning from each challenge, I developed a system that achieves a high accuracy in matching transaction descriptors to merchants. The involvement of AI (ChatGPT) throughout the process not only sped up development but also served as a learning aid, helping me adopt best practices and approaches that I might not have arrived at as quickly on my own. The final result meets the project requirements, and the experience has equipped me with valuable insights for tackling similar data engineering problems in the future.