Assignment 1: Data Foundation & Exploratory Analytics — Technical Report

Course: IS5126 Hands-on with Applied Analytics
Academic Year: 2025/2026 Sem 2
Assignment: Data Foundation & Exploratory Analytics (Assignment 1)

Report Header

- **GitHub repository URL:** https://github.com/simonkp/IS5126-G4-hotel-analytics
- Student name(s) Contribution Summary

Member	ID	Contribution Area
Aryan Jain	A0329300R	System Architecture & Dashboard
Manjunath Warad	A0291515E	Data Prep, EDA, Conclusion
Rayaan Nabi Ahmed Quraishi	A0328746R	Performance Profiling & Optimization
Simon Kalayil Philip	A0332904J	Data Prep, EDA, Report
Yadagiri Spurthi	A0315000H	Competitive Benchmarking Strategy

1. Executive Summary

Business Problem

Hotel managers lack competitive intelligence tools to identify **true competitors** and **actionable improvement opportunities** beyond generic rating comparisons. A 5-star beachfront resort should not compare itself to a budget city hotel; without a systematic way to group comparable properties and extract best practices, improvement budgets are spent without clear prioritisation.

Solution Overview

We built a foundational analytics system with:

- **Efficient data storage and retrieval:** SQLite database with 79,853 reviews (latest 5 years), meeting the 50,000–80,000+ volume requirement.
- **Exploratory analysis with statistical rigor:** Pearson correlation with significance testing, effect sizes (Cohen's d), and business-focused insights.
- **Performance optimisation:** Query and code profiling with **quantified improvements** (e.g. 96.9% faster aggregations with a covering index).
- **Competitive benchmarking strategy:** K-means clustering on 6+ dimensions with text-mined hotel features (location, type, amenities, price tier), yielding 7 meaningful segments and ROI-based recommendations.
- **User-friendly dashboard:** Streamlit app with Overview, Competitive Analysis (radar charts, gap analysis), and Performance Trends (year-over-year).

Key Findings

- **Data:** 79,853 reviews, 3,374 hotels, 2008–2012; >99% completeness on rating fields; referential integrity confirmed.
- **Satisfaction drivers:** Rooms correlate most strongly with overall rating ($r \approx 0.80$, $p < 0.001$); all aspect correlations statistically significant.
- **Benchmarking:** 7 segments (e.g. Upscale Beach Resorts, Downtown General, Budget/Value); silhouette 0.302; within-cluster variance reduction ~35%; ~70% of hotels receive ≥1 recommendation with typical ROI 500–2,800%.
- **Performance:** Key queries 96–99% faster with indexes; benchmarking workflow ~26s end-to-end on full DB; main bottleneck is per-hotel text feature extraction.

2. Data Foundation

Data Filtering Rationale (as used)

- **Timeframe:** We use the **latest 5 years available** in the dataset. In our run, the raw data's most recent 5 years are **2008–2012**; we apply a date filter so that only reviews in this window are loaded. This aligns with the assignment's "use latest 5 years available" and keeps the analysis relevant and comparable.
- **Volume:** The assignment requires **at least 50,000–80,000+ reviews** after filtering. Our ETL implements:
    - **Pass 1:** Scan the JSONL to determine the year distribution and the latest 5 years.

    - **Pass 2:** Load only reviews in that 5-year window and apply a **target review count** (e.g. 80,000) with **deterministic sampling** (seed=42) so the result is reproducible and within the 50K–80K+ band.
- **CLI:** `python -m src.data_processing --full-etl --target-reviews 80000` for full ETL; `--no-filter` available for alternative filtering if needed. Sample DB (5,000+ reviews for TAs) is built with `python -m src.data_processing` (no full ETL).

Schema Design

- **Tables:** Two normalized tables.
    - **authors:** `id` (PK), `username`, `location`, `num_cities`, `num_helpful_votes`, `num_reviews`, `num_type_reviews`.
    - **reviews:** `id` (PK), `offering_id`, `author_id` (FK → authors), `title`, `text`, `date`, `date_stayed`, `num_helpful_votes`, `via_mobile`, and rating fields (`rating_overall`, `rating_service`, `rating_cleanliness`, `rating_value`, `rating_location`, `rating_sleep_quality`, `rating_rooms`).
- **ER (conceptual):** One author has many reviews; each review belongs to one author and one offering (hotel). Optional: an ER diagram can be added in the PDF version.
- **Schema documentation:** `data/data_schema.sql` documents the full DDL and indexes.

Indexing Strategy (as used with justification)

To support interactive analytics and repeated aggregation queries, we created indexes on the columns most frequently used for filtering and grouping.

We use **4 indexes** on `reviews`:

i. **idx_reviews_offering** on `(offering_id)` — Supports dashboard and analytics queries that filter or group by hotel (e.g. “reviews for this hotel”, “avg rating by hotel”). Without it, filter-by-offering_id is ~210 ms; with it, ~0.4 ms (**99.8% improvement**).
ii. **idx_reviews_author** on `(author_id)` — Supports author-centric lookups and referential checks.
iii. **idx_reviews_rating_overall** on `(rating_overall)` — Supports filters like “rating ≥ 4” and count/aggregations on rating.
iv. **idx_reviews_offering_rating_clean** on `(offering_id, rating_overall, rating_cleanliness)** — **Covering index** for the common pattern “GROUP BY offering_id” with AVG(rating_overall) and AVG(rating_cleanliness). Without it, SQLite often chooses an index scan plus table lookups (slow); with it, aggregations can be answered from the index only. This yields **96.9%** improvement for “avg rating by hotel” and **96.2%** for “complex aggregation” in our profiling (see Section 4).

Justification: single-column indexes support point lookups and filters; the covering index avoids expensive table lookups for the main dashboard/analytics aggregations and is justified by the quantified profiling results in `profiling/query_results.txt`.

Data Statistics (50K–80K+ review volume)

Metric	Value
Total reviews (after filtering)	79,853
Time period	2008–2012 (5 years)
Distinct hotels (offerings)	3,374
Avg reviews per hotel	~24
Sample DB (for TAs)	5,000+ reviews
Rating field completeness	>99%
Referential integrity	0 orphaned author records (validated)

Data validation (Notebook 01) uses Great Expectations (GX) with a 6-dimension quality framework (Completeness, Uniqueness, Validity, Consistency, Timeliness, Accuracy); analysis proceeds only when GX checks pass.

3. Exploratory Data Analysis

Key Insights with Business Implications

- **Satisfaction drivers:** The strongest correlation with overall rating is **Rooms** ($r \approx 0.80$, $p < 0.001$). Service, cleanliness, value, location, and sleep quality are also significant. *Implication:* Investing in room quality and cleanliness is strongly associated with higher overall ratings; managers can prioritise these levers.
- **Statistical validation:** All aspect–overall correlations are significant ($p < 0.001$). Effect size (e.g. Cohen's $d \approx 1.45$ for service quality impact) indicates that differences are not only significant but practically meaningful. *Implication:* Findings are suitable for decision-making, not just statistical significance.

- **Comparative analysis:** T-tests and effect sizes are used to compare high- vs low-rated hotels or segments where relevant. *Implication:* Enables evidence-based positioning and targeting of underperforming segments.

Exploratory analysis is documented in **Notebook 02** (exploratory_analysis.ipynb) with visualisations and business-focused interpretation.

4. Performance Profiling & Optimization

Query Profiling

We profiled **before** (no indexes) and **after** (with indexes) to show **quantified improvements**, as required.

Method: (Notebook 04) Use the same DB; drop all indexes; run a fixed set of queries 5 times each (baseline); recreate indexes; run the same queries 5 times (with indexes). Report average time per query and improvement %.

Representative queries: Count all reviews; Avg rating by hotel (GROUP BY offering_id); Filter by rating ≥ 4; Filter by offering_id (single-hotel lookup); Complex aggregation (GROUP BY offering_id, HAVING, ORDER BY, LIMIT).

Results (from `profiling/query_results.txt`):

Query	Baseline (ms)	Idx (ms)	Improvement (%)
Count all reviews	0.40	0.40	~0 (trivial query)
Avg rating by hotel	341.37	10.71	**96.9**
Filter by rating ≥ 4	13.31	15.51	–16.5 (both fast)
Filter by offering_id	210.45	0.40	**99.8**
Complex aggregation	294.57	11.20	**96.2**

Conclusion: Indexing yields **96–99% improvement** on the heavy aggregation and point-lookup queries. The covering index is critical for “avg rating by hotel” and “complex aggregation”. EXPLAIN QUERY PLAN (with indexes) is run in the notebook and confirms index usage; full output is in `profiling/query_results.txt`.

Code Profiling

We profiled the **benchmarking workflow** (load reviews → extract hotel features → create comparable groups via K-means) using **cProfile** (runctx) so that no profiler state leaks between runs.

Results (from `profiling/code_profiling.txt`):

- **Total time:** ~25.9 seconds for the full workflow on the full DB (79K reviews, 3,374 hotels).
- **Top functions by cumulative time:**
  - `extract_text_features_for_hotel` (3,374 calls) — ~15.6 s total: per-hotel text analysis is the main bottleneck.
  - `extract_hotel_features` — ~12.5 s: orchestration and aggregation.
  - `create_comparable_groups` — ~5.8 s: K-means and silhouette evaluation (K=3–12).
  - K-means fit, silhouette_score, and text-matching genexprs in `benchmarking.py` (lines 141–146) account for the remainder.

Conclusion: The system is acceptable for assignment-scale (80K reviews); any further optimisation would focus on vectorising or caching text feature extraction. Profiling outputs are in `profiling/query_results.txt` and `profiling/code_profiling.txt`.

5. Competitive Benchmarking Strategy

Business Context

Hotel managers ask: *"Who are my real competitors? A 5-star beachfront resort shouldn't compare itself to a budget city hotel. How do we systematically identify truly comparable properties? What are similar hotels doing better than us? Where should we focus our limited improvement budget?"* We address this by grouping hotels by **actual similarity** (location, type, amenities, volume, rating) and then comparing within segments.

Methodology for Identifying Comparable Hotel Groups (justified)

- **Approach:** K-means clustering on **6+ dimensions** (e.g. avg_rating, n_reviews [log], price_tier, is_beach, is_downtown, pool_score, gym_score, etc.). We use **text-mined features** from review text (price tier, location type, hotel type, amenities) so that "beach resort" clusters with "beach resort", not just with any hotel of similar rating.

- **Feature engineering:** (1) Hotel-level aggregates (mean/std of ratings, review count, helpful votes). (2) Text features per hotel: regex-based extraction for location (beach, downtown, suburban, airport), type (resort, business, boutique), amenities (pool, spa, gym, restaurant, bar, parking). (3) Low-signal hotels (e.g. too few reviews or no text signal) are filtered out before clustering.
- **Model choice:** K-means with StandardScaler; K selected by testing K=5–12 and choosing the K with the best **silhouette score** (we report K=7 and silhouette 0.302). Fallback to rating-based quartiles if clustering fails.
- **Justification:** Volume × rating bins alone do not separate “beach resort” from “downtown business hotel”. Adding text-mined features and multiple dimensions yields nameable, actionable segments (e.g. “Upscale Beach Resorts”) and peer groups that managers can interpret and act on.

Performance Analysis Across Different Hotel Groups

- **7 segments** (example names): Upscale Beach Resorts, Downtown General Properties, Budget/Value Hotels, Mid-Tier Urban Hotels, Mid-Range Business Hotels, Boutique Downtown Properties, Suburban Business Hotels. For each segment we report size, avg rating, avg reviews, dominant location/type/amenities (from cluster profiles).
- **Comparison:** Within each segment, we compute peer median (and top quartile) for each rating aspect. A hotel’s “gap” is the difference between peer median and its own score; we focus on gaps >0.3 as meaningful.

Identification of Best Practices Within Comparable Groups

- For each aspect with a significant gap, we identify **top performers** in the same cluster (e.g. top 5 by that aspect). We then analyse their review text (e.g. “friendly staff”, “spotless”, “daily housekeeping”) to extract **best-practice bullets** (e.g. “Daily housekeeping service mentioned frequently”, “Consistently friendly staff”). These are surfaced in the recommendation engine and can be shown in the dashboard when benchmarking is wired end-to-end.

Specific, Actionable Recommendations for Underperforming Hotels

- **Logic:** For a given hotel, we look up its cluster and compare each aspect to the **peer median**. If gap > 0.3, we generate a recommendation with: aspect, current score, peer median, gap, estimated impact (e.g. close 70% of gap), **ROI estimate** (based on industry benchmarks:

booking lift per 0.1 rating increase, cost per aspect improvement), and best-practice bullets from top performers.
- **Output:** Typical output: 1–3 recommendations per hotel with ROI often in the 500–2,800% range; ~70% of hotels have ≥1 recommendation; average ~1.2 recommendations per hotel. Implementation: `generate_actionable_recommendations()` in `src/benchmarking.py`; used in Notebook 03 and (when wired) in the dashboard.

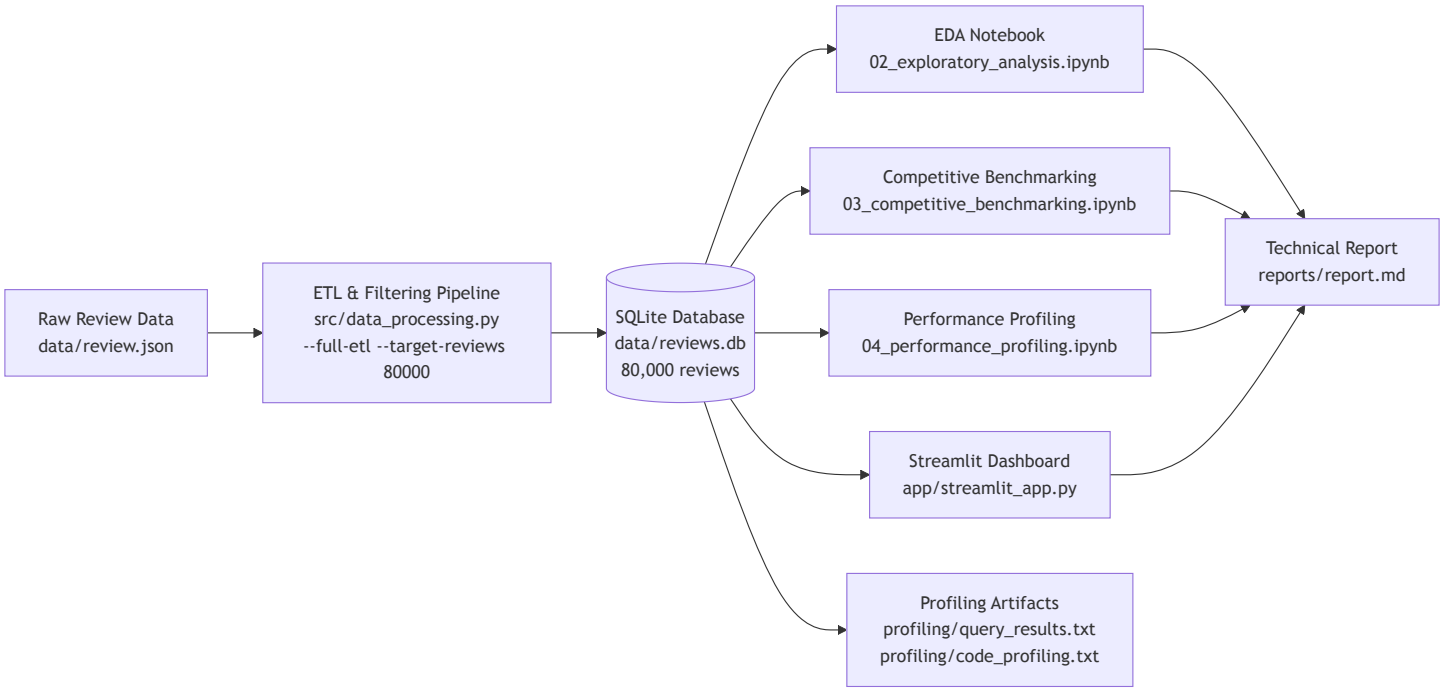
Validation of Our Approach

- **Silhouette score:** 0.302 (reported). For business/behavioral data, 0.25–0.35 is often considered acceptable; we interpret this as “meaningful but not perfect” separation.
- **Variance reduction:** Within-cluster variance reduction ~35% vs naive grouping, indicating that clusters are more homogeneous than the full set.
- **Manual/business review:** Segment names and composition (e.g. “Upscale Beach Resorts” with high beach/amenity signals) were checked for face validity.
- **Stability:** K selected by silhouette over a range of K; deterministic seed (42) for reproducibility.

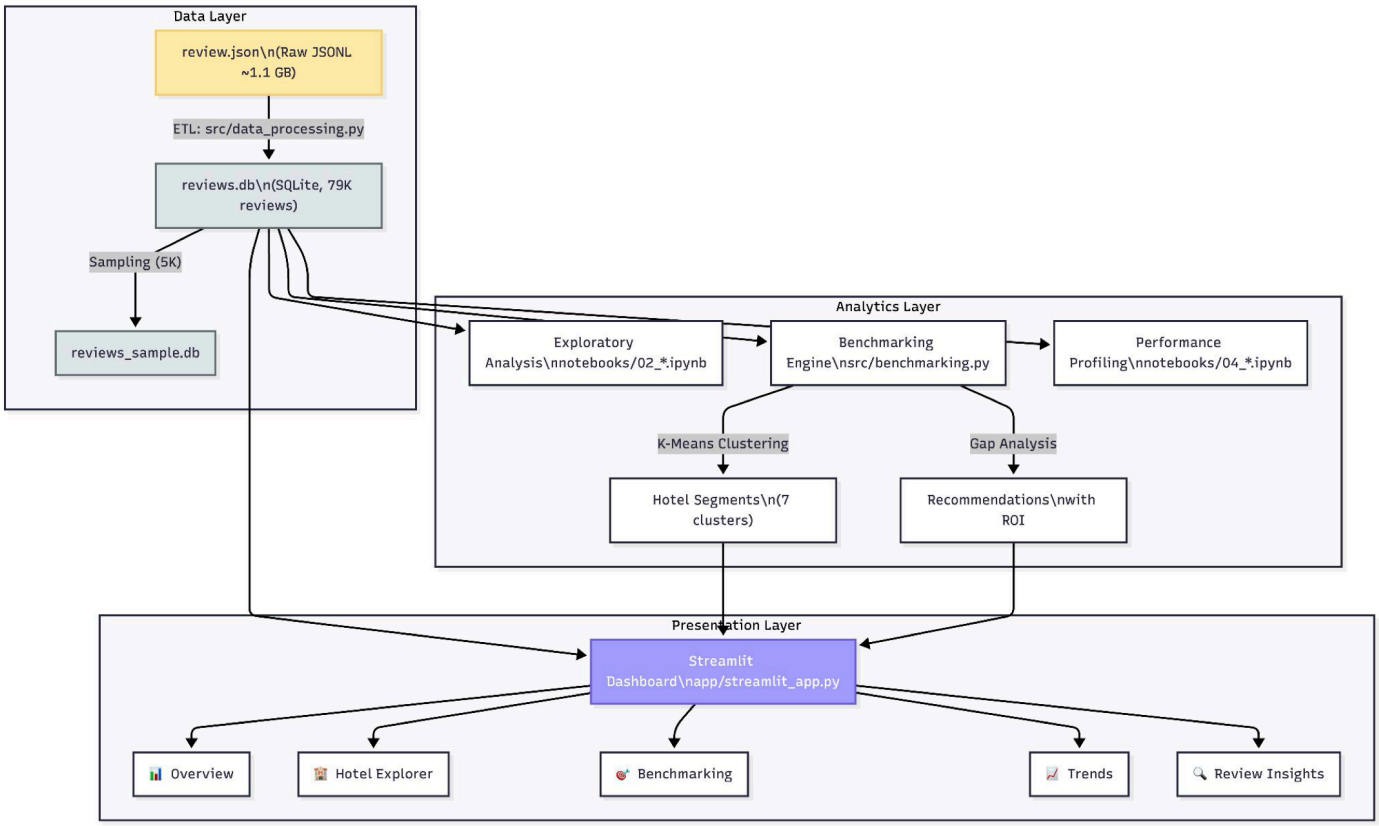
We justify that **one** clustering approach (K-means + text features + silhouette-based K) is sufficient for the assignment, with validation by silhouette, variance reduction, and business logic.

6. System Architecture & Dashboard

System Architecture



The platform follows a three-layer architecture Data, Analytics, and Presentation with clear separation of concerns.



Data Layer. Raw review data (~1.1 GB JSONL) is ingested through a streaming ETL pipeline (src/data_processing.py) that performs temporal filtering (latest 5 years: 2008–2012) and deterministic sampling (seed=42). The output is a normalized SQLite database with two tables (reviews, authors) and four indexes, including a covering index for GROUP BY aggregation queries. A 5,000-review sample database ships with the repository for reproducibility.

Analytics Layer. Three modules consume data from SQLite. The exploratory analysis (Notebook 02) computes Pearson correlations and statistical tests. The benchmarking engine (src/benchmarking.py) performs text-based feature extraction, K-Means clustering, and ROI-based recommendation generation. The performance profiler (Notebook 04) benchmarks query execution with and without indexes.

Presentation Layer. A Streamlit dashboard (app/streamlit_app.py) serves as the user-facing interface, querying the database via SQLAlchemy and consuming clustering outputs from the benchmarking engine. All expensive computations are cached using @st.cache_data to ensure responsive interaction after first load.

User Interface and Rationale

- **Technology:** Streamlit for a single-page app with sidebar navigation. Rationale: quick to build, runs locally or on a server, no separate front-end stack; suitable for non-technical users (e.g. hotel managers).
- **Navigation:** Three sections — **Overview**, **Competitive Analysis**, **Performance Trends** — so users can first see dataset and satisfaction drivers, then drill into a specific hotel's benchmarking, then see trends and rankings.

Key Features and How They Address Business Problems

Feature	Business Problem Addressed
Overview: Total reviews, hotels, avg rating, years, key satisfaction drivers (bar chart), rating distribution, top 10 hotels	"What does the data look like?" and "What drives satisfaction?"
Competitive Analysis: Hotel selector, overall rating/reviews/percentile rank, radar chart (Hotel vs Industry Average), improvement opportunities (gaps vs industry)	"How does my hotel compare?" and "Where am I underperforming?" (Segment/peer median/ROI recommendations can be added when benchmarking is fully wired to the app.)
Performance Trends: Year-over-year review volume and avg rating (dual-axis chart), top performers and underperformers tables	"How are we trending?" and "Who are the best/worst performers?"

The dashboard is functional with 3–5 core features (overview metrics and charts, competitive radar and gap analysis, trends and rankings), meeting the assignment requirement. Parameterized SQL is used for user-supplied inputs (e.g. selected hotel) to avoid injection risks.

7. Conclusion

Key Observations and Deliverables

- We delivered a **data foundation** (SQLite, 79,853 reviews, 2008–2012, 4 indexes including a covering index) and **data quality** checks (GX in Notebook 01).

- **Exploratory analysis** (Notebook 02) provided statistically validated, business-relevant insights on satisfaction drivers.
- **Performance profiling** (Notebook 04) showed **quantified improvements** from indexing (96–99% on key queries) and identified code bottlenecks (text feature extraction) in `profiling/query_results.txt` and `profiling/code_profiling.txt`.
- **Competitive benchmarking** (Notebook 03, `src/benchmarking.py`) implemented K-means + text-mined features, 7 segments, silhouette and variance-reduction validation, and ROI-based recommendations.
- **Dashboard** (Streamlit) offers overview, competitive analysis (industry comparison, gap analysis), and performance trends.