Smart Substitution Model – Architecture Options Documentation

Overview

This document details six architecture options for the Smart Substitution Model in DiscountMate, designed to recommend cost-effective alternatives for items in a user's cart (e.g., swapping a $4 milk for a $3 equivalent). The model leverages the finalized dataset with columns: `product_code`, `name`, `brand`, `brand_confidence`, `brand_tier`, `category`, `subcategory`, `original_price`, `sale_price`, `std_item_size`, `std_item_size_unit`, `item_size`, `price_per_unit`, `unit_type`, `size_band`, `tags`, and `similarity_score`. Each option explains how it works, incorporating relevant columns, pros, cons, and implementation hints. The choice will depend on factors like explainability, scalability, and dataset size, with potential for hybridization (e.g., rule-based + embeddings).

1. Rule-Based Baseline

Description: A straightforward system using predefined rules to filter and select substitutes from the dataset. It prioritizes matches on `subcategory`, `unit_type`, and `size_band`, while favoring higher `brand_tier` and lower `sale_price`. For example, scan the dataset for items in the same `subcategory` with compatible `unit_type`, within ±10% `size_band` or `item_size`, and swap if the alternative's `sale_price` is lower, using `price_per_unit` for normalization.

How it Works:

- Filter by exact `subcategory` match (1 if same, else 0).
- Ensure `unit_type` compatibility (e.g., 'g' vs. 'ml').
- Check `size_band` or `item_size` proximity (±10% range using `std_item_size` if available).
- Prefer same or higher `brand_tier` based on `brand_confidence`.
- Select swap if `sale_price` < original, calculating savings via `(original_price - sale_price) / original_price`.

  Pros: Highly explainable (e.g., "Swapped due to same subcategory and 20% cheaper"); fast implementation without ML training.

  Cons: Inflexible for nuanced cases, like ignoring `tags` or `name` semantics; may miss creative swaps.

  Implementation Hint: Use Pandas filtering on the dataset; no training needed, just query logic.

2. Weighted Scoring System

Description: Assigns scores to potential substitutes using a formula that weights multiple columns, ranking options for the best swap. This balances similarity

(`similarity_score`, `size_band`) with savings (`sale_price`, `price_per_unit`).

How it Works:

- Formula example: `score = 0.4 * subcategory_match + 0.2 * size_similarity + 0.2 * brand_similarity + 0.2 * price_saving`.
- `subcategory_match`: 1 if matches `subcategory`, else 0.
- `size_similarity`: Normalized difference in `item_size` or `std_item_size` (e.g., 1 - abs(diff) / max_size), using `size_band` for banding.
- `brand_similarity`: Based on `brand_tier` delta and `brand_confidence` (e.g., 1 if same tier, 0.5 if adjacent).
- `price_saving`: `(original_price - sale_price) / original_price`, normalized to 0-1.
- Rank candidates and select top score above threshold.
  Pros: Customizable weights allow tuning per `category`; incorporates more columns like `tags` for bonuses.
  Cons: Requires empirical tuning to prevent biases (e.g., over-emphasizing price); less adaptive to new data.
  Implementation Hint: Implement in Python with NumPy for vectorized scoring; test weights on sample data.

3. Clustering Approach (K-Means / HDBSCAN)

Description: Groups similar products into clusters using feature vectors from key columns, then swaps within clusters to the cheapest viable option based on `sale_price`.

How it Works:

- Feature vector: Encode `subcategory` (one-hot), `unit_type`, normalized `price_per_unit`, `std_item_size/item_size`, and `brand_tier`.
- Apply K-Means (fixed clusters) or HDBSCAN (density-based) to form groups.
- For a query item, find its cluster and rank alternatives by lowest `sale_price`, filtering by `size_band` and `similarity_score` > threshold.
  Pros: Automatically discovers product groups without manual rules; scalable for large catalogs like our 20k+ items; handles multi-dimensional similarity.
  Cons: Cluster quality needs validation (e.g., silhouette score); may group unrelated items if features are imbalanced.
  Implementation Hint: Use scikit-learn for clustering; preprocess with StandardScaler; visualize with PCA for debugging.

4. Embedding-Based Similarity

Description: Generates vector representations combining structured data and text from `name` and `tags`, using similarity metrics to find and rank substitutes.

How it Works:

- Text embeddings: TF-IDF or pre-trained models (e.g., Sentence-BERT) on `name` + `tags` (e.g., embedding "fresh whole milk 2L" ≈ "organic dairy milk 2000ml").
- Hybrid vector: Concatenate embeddings with encoded `subcategory`, `unit_type`, `size_band`, `brand_tier`, and `price_per_unit`.
- Compute cosine similarity to rank candidates; swap top ones with `sale_price` savings and `similarity_score`boost.
  Pros: Captures semantic nuances in messy `name` data; flexible for partial matches (e.g., via wildcard in `tags`).
  Cons: Lower explainability ("similar due to vector distance"); requires embedding computation overhead.
  Implementation Hint: Use scikit-learn's TfidfVectorizer or Hugging Face transformers; FAISS for efficient nearest-neighbor search on large datasets.

5. Graph-Based Approach

Description: Models products as a graph where nodes are items, and edges represent similarity weighted by column differences, finding swaps via nearest neighbors.

How it Works:

- Nodes: Each row by `product_code`.
- Edges: Connect within `subcategory/category`, with weights = inverse of differences in `item_size`, `price_per_unit`, `brand_tier` (e.g., multi-attribute distance).
- For a query, traverse graph to find connected nodes with lower `sale_price`; incorporate `similarity_score` as edge boost.
  Pros: Intuitive explanations ("connected by similar size and brand"); natural for reasoning chains (e.g., multi-hop swaps).
  Cons: Graph construction/maintenance is computationally heavy for growing datasets; requires libraries like NetworkX.
  Implementation Hint: Build with NetworkX; use shortest path or PageRank for ranking; precompute for efficiency.

6. Multi-Objective Optimisation

Description: Frames substitution as optimizing multiple goals under constraints, balancing similarity and savings using columns like `similarity_score` and `sale_price`.

How it Works:

- Objective: Maximize `λ1 * similarity_score + λ2 * price_saving + λ3 * brand_quality`, where `price_saving = (original_price - sale_price) / original_price` and `brand_quality` from `brand_tier/brand_confidence`.
- Constraints: Same `subcategory/unit_type`; `size_band` match; `item_size` within range.

- Solve per query using libraries like SciPy.optimize.
  Pros: Explicitly handles trade-offs (e.g., slight size mismatch for big savings); adaptable per `category`.
  Cons: Tuning lambdas (λ) is iterative; slower for real-time queries without approximation.
  Implementation Hint: Use linear programming if objectives linear; test on subsets with grid search for λ values.

Next Steps

- **Evaluation**: Test each on sample data using metrics like swap acceptance rate, savings generated, and precision (via manual review).
- **Hybridization**: Combine (e.g., rules for filtering + embeddings for ranking).