Query Auto-Completion System

End-to-End Pipeline Design

Stage 1: Feature Engineering

Input Data:

- `df_pool`: 150K unique queries
- `df_query_features`: catalog_clicks, orders, volume, catalog_views

Processing Steps:

1. Merge pool with query features on 'query' column
2. Create lowercase version: `query_lower`
3. Build 3-char prefix index: `prefix_3char`
4. Normalize features using min-max scaling
5. Compute popularity score with weighted formula

Popularity Score Formula:

$$p = 0.4 \cdot \text{orders\_norm} + 0.3 \cdot \text{clicks\_norm} + 0.2 \cdot \text{volume\_norm} + 0.1 \cdot \text{views\_norm}$$

Output: `df_pool_enhanced` with popularity scores

Stage 2: Dual Encoding Pipeline

Dense Path: Semantic Embeddings

1. Load `all-MiniLM-L6-v2`
2. Batch encode: 64 queries/batch
3. Original: 384-dimensional
4. Apply PCA: 384D $\rightarrow$ 128D
5. Save as `.npy` (float32)

Why PCA?

- Reduces unnecessary dimension.
- Bearable accuracy loss

Sparse Path: Lexical Matching

1. Initialize BM25 with HashingVectorizer
2. Set $n\_features = 2^{14} = 16384$
3. Fit on pool queries (IDF computation)
4. Batch transform: 10K queries/batch
5. Apply sparsification: threshold = 0.01
6. Save as CSR matrix (`.npz`)

Stage 3: Index Construction

FAISS Dense Index:

- Type: `IndexFlatIP` (Inner Product for cosine similarity)
- Dimension: 128D
- Vectors: 150,000
- Index size: $\sim$75 MB
- Search complexity: O(n) but highly optimized with SIMD

Prefix Lookup Table:

- 3-character prefix mapping
- Example: `"nik"` $\rightarrow$ [123, 456, 789, ...]
- Reduces search space: 150K $\rightarrow$ 1-5K queries
- Stored in DataFrame index for O(1) lookup

All Components Serialized:

- BM25 model, PCA transformer, FAISS index, Dense embeddings, Sparse matrix, Pool metadata

Stage 4: Inference & Scoring

Query Processing Flow:

```
# 1. Prefix filtering (if len >= 3)
mask = df_pool_enhanced['prefix_3char'] == prefix.lower()[:3]
filtered_indices = mask[mask].index.to_numpy()

# 2. Encode query with both paths
dense_vec = model.encode([prefix])[0]
dense_vec = pca.transform([dense_vec])[0]  # 128D
sparse_vec = bm25.transform_query(prefix)   # BM25 IDF

# 3. Normalize and weight (alpha = 0.6)
hdense = normalize(dense_vec) * alpha
hsparse = normalize(sparse_vec) * (1 - alpha)

# 4. FAISS search on filtered/full pool
dense_scores, indices = faiss_index.search(hdense, top_k * 2)

# 5. Sparse similarity with CSR matrix
sparse_scores = pool_sparse_matrix[indices].dot(hsparse.T).flatten()

# 6. Combine with popularity boost (w = 0.15)
final_scores = dense_scores + sparse_scores + w * popularity[indices]

# 7. Top-K selection with argpartition
top_indices = np.argpartition(final_scores, -k)[-k:]
```

Performance & Optimizations

Speed Optimizations:

- **Numba JIT**: Score combination in compiled code
- **Batch encoding**: 5x faster than sequential
- **Prefix filtering**: 30x search space reduction
- **FAISS SIMD**: Hardware-accelerated similarity
- **CSR sparse matrix**: Memory-efficient sparse ops

Configuration:

- $\alpha = 0.6$ (dense/sparse balance)
- $w = 0.15$ (popularity weight)
- $k = 100$ (candidates)

Approach-2: Overview & Motivation

Core Philosophy

Optimize for **speed** over semantic complexity using lightweight, deterministic methods

Key Advantages:

- Extremely fast - no neural models at inference
- Simple deployment - single Python script
- Explicit typo handling via fuzzy matching
- Low memory footprint

Expected Limitations:

- Semantic gaps - misses paraphrases
- Lower hit rate on unseen queries
- No learning from query relationships

Trade-off: Sacrifice semantic understanding for millisecond-level response times

Approach-2: Four Core Components

1. **Historical Frequency Lookup**
   - Direct prefix $\rightarrow$ query mapping from training data
   - Captures actual user selection behavior
   - *Example:* prefix ‘sho” $\rightarrow$ [‘shoes”: 450, “shorts”: 320, “shoulder bag”: 180]

2. **Prefix Index (Inverted Index)**
   - Maps each possible prefix to all queries starting with that prefix
   - Memory-efficient: limit queries stored per prefix
   - Enables instant lexical matching

3. **Fuzzy Matching**
   - Handles typos and spelling variations
   - Computes similarity between input prefix and popular queries
   - Threshold-based candidate selection
   - *Example:* “shoez” $\approx$ “shoes” (similarity > 0.8)

4. **Popularity-Based Backfill**
   - Fallback mechanism when < 150 candidates found
   - Uses most popular queries from entire pool
   - Ensures complete 150-query output

Scoring, Ranking & Implementation

Weighted Scoring Function

Final candidates scored using weighted combination:

$$\text{Score}(q|p) = w_1 \cdot S_{\text{hist}}(q,p) + w_2 \cdot S_{\text{prefix}}(q,p) + w_3 \cdot S_{\text{fuzzy}}(q,p) + w_4 \cdot S_{\text{pop}}(q)$$

where:

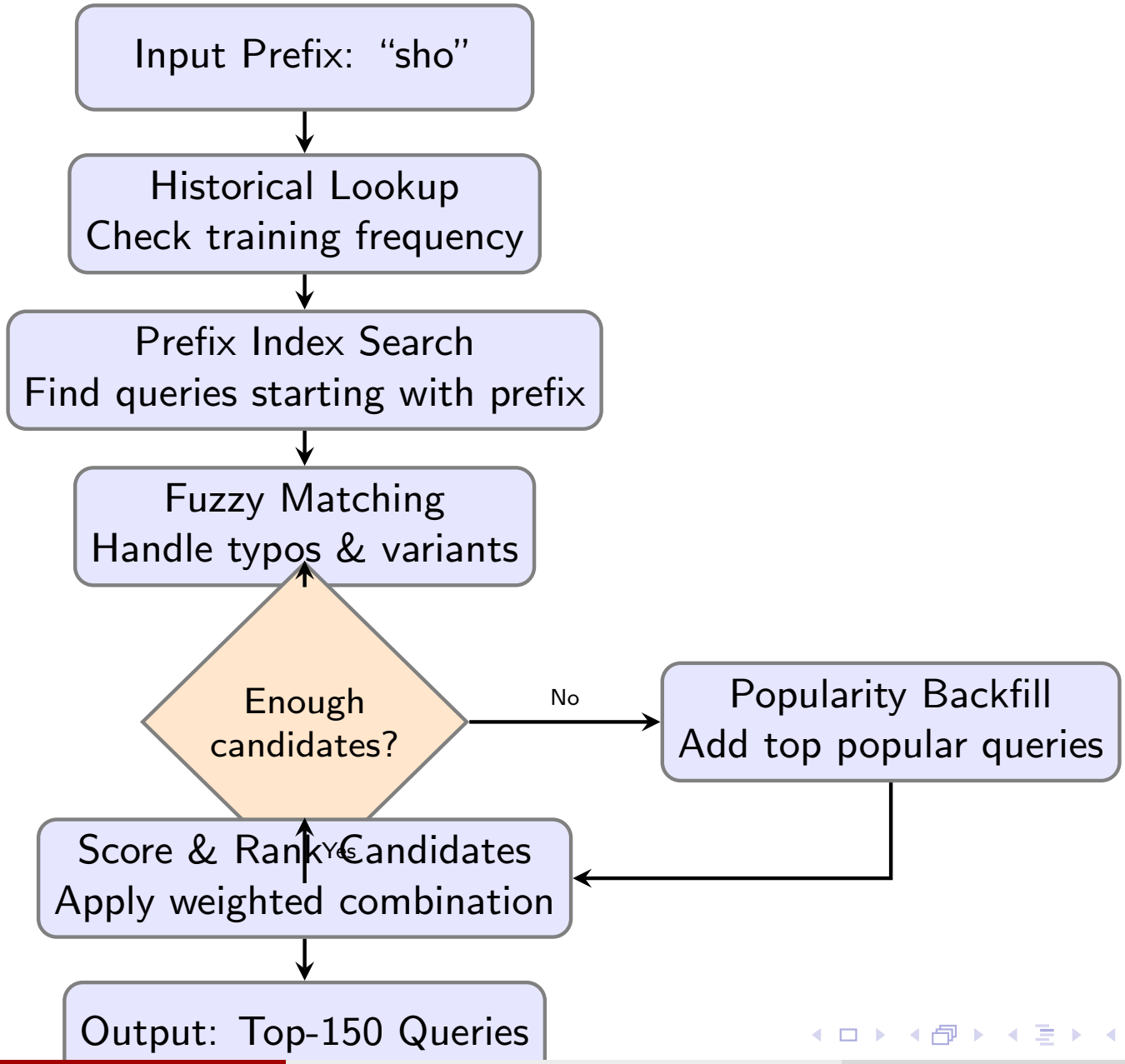
- S_{hist} = Historical match score (normalized frequency)
- S_{prefix} = Direct prefix match score (binary or position-based)
- S_{fuzzy} = Fuzzy similarity score (e.g., Levenshtein-based)
- S_{pop} = Normalized popularity score from engagement metrics

Scoring, Ranking & Implementation

Implementation Strategy

- **Technology:** Single Python script with standard libraries
- **Data structures:** Dictionaries for frequency maps, prefix indices
- **Fuzzy matching:** RapidFuzz or similar lightweight library
- **No neural models loaded** $\rightarrow$ instant startup, minimal RAM
- **Output:** Top-150 queries ranked by composite score

Complete Retrieval Pipeline



Approach 3: Generative LLM (GPT-2 + LoRA)

Hypothesis: Treat auto-complete as a text generation task rather than retrieval.

Model Architecture & Setup

- **Base Model:** GPT-2 Small (124M parameters).
- **Fine-tuning Strategy:** LoRA (Low-Rank Adaptation).
- **Trainable Parameters:** $\approx$ 600K (0.5% of total).
- **Input Format:** `Prefix: <input> \n`
- **Target Output:** `Completion: <term1>, <term2>...`

Objective: Test if the model can learn the query distribution and generate valid completions based on prefix alone.

LLM Approach: Technical Failures

Experimentation revealed three fundamental blockers for production usage.

1. **Inference Latency (Critical)**
   - **Observed:** 3-4 seconds per request.
   - **Cause:** Autoregressive generation is sequential ($O(N)$). And LLM slower comparitively.

2. **Hallucination & Constraints**
   - **Issue:** Model generates valid English words that are *not* in our specific query pool.
   - **Stat:** lot of generations were "out-of-vocabulary" regarding the database.
   - **Correction Cost:** Post-filtering invalid terms increases latency further.

3. **Stochasticity**
   - Auto-complete requires deterministic ranking based on popularity.
   - LLM outputs vary based on temperature/seed (inconsistent user experience).

Architecture & Model

- **Model**: OpenAI CLIP ViT-B/32
- **Embedding dim**: 512
- **Normalization**: L2 (cosine similarity)

Data Pipeline:

- Dataset: Amazon Reviews 2023 (All_Beauty)
- Filter: images.notna(), title.len() < 10
- Sample: 5000 products
- Batch size: 32

Image Encoding

```
# Batch processing with CLIP
image_inputs = torch.stack([
    preprocess(img) for img in batch_images
]).to(device)

with torch.no_grad():
    features = model.encode_image(image_inputs)
    # L2 normalization for cosine similarity
    features = features / features.norm(dim=-1, keepdim=True)
```

Key Operations:

- Image preprocessing: PIL $\rightarrow$ CLIP transform
- Encoding: ViT-B/32 encoder
- Output: normalized 512-dim vectors

FAISS Indexing & Search

```
# FAISS Inner Product index
embeddings_np = embeddings.numpy().astype('float32')
dimension = embeddings_np.shape[1]  # 512
index = faiss.IndexFlatIP(dimension)
index.add(embeddings_np)

# Text query encoding
text_tokens = clip.tokenize([query]).to(device)
text_features = model.encode_text(text_tokens)
text_features /= text_features.norm(dim=-1, keepdim=True)

# Similarity search
similarities, indices = index.search(query_np, top_k)
```

Search: IndexFlatIP (Inner Product = Cosine Sim)

Comparative Analysis & Conclusion

Final Decision

- **Reject Approach 3 (LLM):** Too slow, uncontrollable output.
- **Reject Approach 4 (CLIP):** Solves wrong problem (semantic vs lexical), lacks image data.
- **Recommendation:** Proceed with **Approach 1** (BM25 + SBERT Re-ranking) or **Approach 2** (Trie/Frequency) for production.