Java E-Commerce Recommendation Project

Step 4 and Step 6 Documentation

Detailed Code Analysis and Logic Explanation

Antigravity AI

January 12, 2026

Contents

Chapter 1

Step 4: Recommendation Model (Deep Dive)

This chapter provides a comprehensive technical breakdown of the Recommendation Engine. This system is a **Hybrid Content-Based Recommender** enhanced with **Probabilistic Sentiment Analysis**, designed to solve the problem of information overload in e-commerce.

1.1 System Architecture

The recommendation pipeline consists of three distinct mathematical components that function in unison:

1. **Text Vectorization (TF-IDF)**: Converts unstructured product text into a structured Numerical Vector Space.
2. **Sentiment Classification (Naive Bayes)**: A probabilistic AI that determines the qualitative quality of products based on user reviews.
3. **Composite Scoring Engine**: A ranking algorithm that aggregates vector similarity, sentiment probability, and product metadata into a final score.

1.2 Component 1: Text Vectorization (TF-IDF)

Class: `TFIDFVectorizer.java`

To compare products mathematically, we must first convert their textual titles and descriptions into numbers. We use the **Vector Space Model (VSM)**.

1.2.1 Mathematical Theory

The weight of a term t in a document d is calculated as the product of its *Term Frequency* and *Inverse Document Frequency*.

1. Term Frequency (TF)

Measures how often a word appears in a specific product description.

$$TF(t, d) = \frac{\text{count}(t, d)}{\text{total words in } d}$$

Rationale: If "OLED" appears frequently in a description, it is likely significant to that product.

2. Inverse Document Frequency (IDF)

Measures how informative a word is across the entire catalog.

$$IDF(t) = \log\left(\frac{N}{1 + DF(t)}\right)$$

Where:

- N: Total number of products (documents).
- $DF(t)$: Number of products containing term t.

Rationale: Common words like "the", "and", or "with" appear in almost all documents ($DF(t) \approx N$), resulting in an IDF near 0. Rare words like "GeForce" or "4K" have a high IDF.

3. Final Vector Calculation

The vector representation V_d of a product is an array where each dimension corresponds to a word in the vocabulary:

$$V_d = [TFIDF(t_1, d), TFIDF(t_2, d), ..., TFIDF(t_n, d)]$$

1.2.2 Code Implementation

```java
// Code snippet from TFIDFVectorizer.java
public double[] transform(String text) {
    List<String> tokens = tokenize(text);
    double[] vector = new double[vocabSize];

    for (String term : tokens) {
        if (vocabulary.containsKey(term)) {
            int index = vocabulary.get(term);
            double tf = (double) countOccurrences(term, tokens) / tokens.
    size();
            double idf = idfWeights.get(term);
            vector[index] = tf * idf; // The core formula
        }
    }
    return vector;
}
```

1.3 Component 2: Sentiment Analysis (Naive Bayes)

Class: `NaiveBayesClassifier.java`

While typical recommenders only look at product features, our system analyzes the *human element* using a **Supervised Machine Learning** algorithm: Multinomial Naive Bayes.

1.3.1 Mathematical Theory

We calculate the probability that a given review D belongs to a class C (Positive or Negative) using **Bayes' Theorem**:

$$P(C|D) = \frac{P(D|C) \cdot P(C)}{P(D)}$$

Since the denominator $P(D)$ is constant for comparison, we maximize the numerator. We assume independence between words ("Naive" assumption):

$$P(C|D) \propto P(C) \cdot \prod_{i=1}^{n} P(w_i|C)$$

Log-Likelihood Optimization

To prevent floating-point underflow (multiplying many small probabilities results in 0), we sum the logarithms:

$$\log P(C|D) \propto \log P(C) + \sum_{i=1}^{n} \log P(w_i|C)$$

Laplace Smoothing

To handle the "Zero Frequency Problem" (where a new word has 0 probability and zeros-out the entire calculation), we use Add-One Smoothing:

$$P(w|C) = \frac{\text{count}(w, C) + 1}{\text{count}(AllWords, C) + |Vocabulary|}$$

1.3.2 Seed Data Training (Cold Start)

Since the application connects to CSVs without labeled sentiment data, we solve the **Cold Start Problem** by pre-training the classifier with a "Seed Dataset" in memory during initialization.

```
// Code snippet from NaiveBayesClassifier.java constructor
private void trainWithSeedData() {
    // Teaching the AI what "Positive" looks like
    train("The battery life is amazing", true);
    train("Fast shipping and generic quality", true);

    // Teaching the AI what "Negative" looks like
    train("Terrible battery life drains fast", false);
    train("Broken screen and bad service", false);
```

```
}
```

1.4 Component 3: The Recommendation Engine

Class: `RecommendationEngine.java`

This class acts as the orchestrator. It orchestrates the Training Phase and the Inference (Search) Phase.

1.4.1 Phase 1: Startup & Training

When the application launches, the `trainModel()` method performs the following heavy-lifting:

1. **Corpus Construction**: Aggregates all product titles and descriptions.
2. **Vocabulary Learning**: Runs `vectorizer.fit()` to calculate IDF weights.
3. **Vector Caching**: Pre-calculates the TF-IDF vector for every product and stores it in memory (‘productVectors‘ Map).
4. **Sentiment Scoring**: Iterates through all reviews, runs the Naive Bayes prediction on each, averages the result, and caches it (‘sentimentCache‘ Map).

1.4.2 Phase 2: Relevance Calculation

When a user searches for a query Q, the engine performs **Cosine Similarity** between the Query Vector V_q and Product Vector V_p:

$$\text{Similarity}(Q, P) = \cos(\theta) = \frac{V_q \cdot V_p}{||V_q|| \cdot ||V_p||} = \frac{\sum_i A_i B_i}{\sqrt{\sum A_i^2}\sqrt{\sum B_i^2}}$$

This results in a score between 0 (No match) and 1 (Perfect semantic match).

1.4.3 Phase 3: Composite Scoring (Ranking)

The final score determines the ranking. It is a weighted sum designed to balance relevance, quality, and popularity:

$$\begin{aligned}\text{Score} = &(\text{Similarity} \times 0.35) \\ &+ (\text{Normalized Rating} \times 0.25) \\ &+ (\text{Normalized Reviews} \times 0.15) \\ &+ (\text{Inverse Price} \times 0.15) \\ &+ (\text{Sentiment Probability} \times 0.10)\end{aligned}$$

```java
// Implementation in RecommendationEngine.java
private double calculateCompositeScore(Product p, double simScore) {
    double ratingScore = p.getAvgRating() / 5.0; // 0.0 to 1.0
    double sentimentScore = (sentimentCache.get(p.getId()) + 1.0) / 2.0; // Map -1..1 to 0..1

    return (simScore * WEIGHT_SIMILARITY) +
           (ratingScore * WEIGHT_RATING) +
           (sentimentScore * WEIGHT_SENTIMENT) + ...;
}
```

This formula ensures that a product which matches the query perfectly but has terrible AI-detected sentiment will be ranked lower than a slightly less relevant but highly praised product.

1.5 Data Model: RecommendationResult.java

A simple POJO (Plain Old Java Object) to hold the output data.

```java
public class RecommendationResult implements Serializable {
    private String productId;
    private String title;
    private double score; // Composite recommendation score
    private int rank; // Ranking position (1 = best)
    // ... getters and setters ...

    public String getRankBadge() {
        switch (rank) {
            case 1: return "[1st] BEST";
            case 2: return "[2nd]";
            case 3: return "[3rd]";
            default: return "#" + rank;
        }
    }
}
```

Explanation

- Implements ‘Serializable‘ for potential network transmission or caching.
- Stores both raw product data (‘title‘, ‘price‘) and computed metadata (‘score‘, ‘rank‘).
- ‘getRankBadge()‘: A UI-helper method that returns a medallion emoji for top 3 results, enhancing the user experience.

1.6 Advanced OOP Architecture

This project adheres to professional software engineering principles by utilizing custom Abstraction and Generics.

1.6.1 Abstract Class: AbstractRecommendationEngine

Instead of a monolithic class, we use an **Abstract Base Class** to define the template for any future recommendation algorithms.

- **Encapsulation**: Shared fields like ‘allProducts‘ are protected.
- **Polymorphism**: Subclasses must provide their own implementation of ‘trainModel()‘.

1.6.2 Generic Class: ScoredItem<T>

To decouple the ranking logic from the ‘Product‘ model, we implemented a custom **Generic Wrapper**.

```
public class ScoredItem<T> {
    private T item;
    private double score;
    // ... Any type T can be ranked!
}
```

Benefit: The engine can now rank *any* object type (Products, Advertisements, Users) using the same generic algorithm, proving high-level system flexibility.

Chapter 2

Step 6: Tests

Quality assurance is handled via JUnit 5 tests. These ensure the logic remains correct as the codebase evolves.

2.1 Engine Tests: RecommendationEngineTest.java

```
14 public class RecommendationEngineTest {
15
16     private RecommendationEngine engine;
17     private List<Product> mockProducts;
18
19     @BeforeEach
20     public void setUp() {
21         mockProducts = Arrays.asList(
22                 new Product("p1", "Samsung Galaxy S23", 800.0, ...),
23                 new Product("p2", "iPhone 15 Pro", 1200.0, ...),
24                 new Product("p3", "Dell XPS 13", 1500.0, ...)
25         );
26         engine = new RecommendationEngine(mockProducts);
27     }
28
29     @Test
30     public void testGetRecommendations() {
31         // Search for "Samsung"
32         List<RecommendationResult> results = engine.getRecommendations("
   Samsung", 0, 2000, "All Categories", 5);
33
34         assertFalse(results.isEmpty());
35         assertEquals("p1", results.get(0).getProductId());
36     }
```

Explanation

- **Mocking**: Instead of loading the real huge CSV file, we create a small list (‘mockProducts‘) with controlled data. This makes tests fast and deterministic.
- **@BeforeEach**: Re-initializes the engine before every test to ensure a clean state.
- **testGetRecommendations**: Verifies that searching for ”Samsung” actually returns the Samsung product as the top result (‘p1‘), confirming the TF-IDF and scoring logic works.

2.2 Vectorizer Tests: TFIDFVectorizerTest.java

```
38      @Test
39      public void testTransform() {
40          List<String> docs = Arrays.asList("apple banana cherry", "banana
    cherry date");
41          vectorizer.fit(docs);
42
43          double[] vector = vectorizer.transform("apple banana");
44
45          assertNotNull(vector);
46          assertEquals(vectorizer.getVocabSize(), vector.length);
47      }
```

Explanation

- Verifies that the vectorizer can handle simple strings.
- Ensures the output vector dimension matches the vocabulary size (which determines the Vector Space dimensions).

2.3 Data Loader Tests: DataLoaderTest.java

```
10      @Test
11      public void testParsePrice() {
12          assertEquals(99.99, DataLoader.parsePrice("99,99 €"), 0.001);
13          assertEquals(1250.0, DataLoader.parsePrice("1 250,00 €"), 0.001);
14      }
```

Explanation

- Tests the robustness of parsing logic. European formats often use commas (‘,‘) for decimals and spaces for thousands separators.
- ‘assertEquals(expected, actual, delta)‘: The delta ‘0.001‘ allows for tiny floating-point differences.