# A Complete NLP and Vector Search-Powered RAG System for Analyzing Amazon Reviews

# Abstract

Modern e-commerce platforms, exemplified by Amazon, generate substantial volumes of unstructured customer review data that encapsulate critical user experiences, satisfaction metrics, and market trends. Despite the inherent value, the high variability and unstructured nature of this text render it largely inaccessible to traditional analytical methods, obstructing its conversion into actionable organizational intelligence. This research presents a fully integrated, AI-driven data pipeline designed to transform large-scale Amazon review data into a structured, searchable, and intelligent information system. The proposed architecture employs advanced natural language processing (NLP), transformer-based sentiment models, and scalable semantic vector indexing using Sentence-BERT and FAISS for efficient semantic search. The system leverages MongoDB Atlas for cloud-scale persistence and incorporates machine learning models for personalized product recommendations, culminating in an interactive Retrieval-Augmented Generation (RAG) chatbot powered by modern Large Language Models (LLMs). Collectively, this platform resolves the central challenge of deriving actionable insights from vast, messy review data, delivering capabilities for automated sentiment analysis, semantic review retrieval, and conversational product intelligence. The resulting production-grade framework demonstrates a robust architecture applicable to enterprise-level customer experience analysis, retail optimization, and decision-support systems.

# 1. Introduction

**Customer reviews** constitute a core component of the modern digital commerce ecosystem, exemplified by platforms such as Amazon. User-generated feedback is relied upon heavily to guide consumer purchasing decisions, shape product reputation, and inform marketplace rankings. These reviews contain **rich qualitative data** reflecting real-world product performance, user expectations, recurring complaints, and emerging market trends. However, this critical information is inherently **unstructured and inconsistent**, generating vast volumes of text that are difficult to analyze or search efficiently using conventional database and analytical techniques. Consequently, organizations struggle to convert this rich but messy information into meaningful, **actionable insights** for product intelligence and retail optimization. This project addresses this core data-to-intelligence challenge by presenting a fully integrated, AI-driven data pipeline designed to transform large-scale Amazon review data into a structured, searchable, and intelligent information system.

## Background

Traditional analytical methods applied to customer feedback, such as simple keyword search or basic lexical sentiment scoring, often prove insufficient. These methods consistently fail to capture the semantic **nuance, contextual meaning, or complex relationships** embedded within user-generated text. In recent years, significant advancements in **Natural Language Processing (NLP)**, particularly with the advent of transformer models, **semantic vector embeddings**, **vector databases**, and **Large Language Models (LLMs)**, have revolutionized

the interpretation and retrieval of textual data. These technologies offer the necessary computational tools to move beyond superficial analysis and enable sophisticated understanding of contextual meaning and intent. The development of scalable database indexing and retrieval architectures, specifically **Retrieval-Augmented Generation (RAG)** systems, presents a viable pathway for grounding LLMs in proprietary or corpus-specific data. Leveraging these modern AI capabilities thus opens new possibilities for understanding granular customer sentiment, building highly personalized recommendation systems, and developing conversational interfaces powered directly by genuine user-generated content.

### Objectives

The main goal of this project is to build a smart, automated system using modern AI to take the huge amount of text in Amazon customer reviews and turn it into **useful, structured information** that companies can use to make decisions.

Specifically, we aim to:

1. **Automate Review Analysis:** Build a fast system to automatically process and analyze thousands of reviews, giving product teams structured data that is impossible to gather manually.
2. **Understand Deep Sentiment:** Use advanced language models (NLP) to figure out exactly how customers feel about specific parts of a product (like price, quality, or ease of use), going beyond simple "good" or "bad" scores.
3. **Enable Smart Search:** Create a search system based on **meaning (semantic search)**, not just keywords, so users can find specific insights that traditional searches miss.
4. **Give Better Recommendations:** Develop a smart recommendation engine that uses these deep insights to suggest products that users will actually be satisfied with.
5. **Build a Smart Q&A Chatbot:** Create an intelligent chatbot that can answer complex questions about products by finding and summarizing information directly from the real customer reviews (**RAG**).
6. **Showcase Modern AI:** Prove that we can combine all the latest AI technologies (NLP, Vector Search, LLMs) into one complete, working, **real-world system** that can be used by businesses today.

# 2. Literature Review

The development of the Amazon Review Intelligence System is grounded in several intersecting research areas, including Natural Language Processing (NLP), information retrieval, deep learning, and advanced conversational AI architectures.

## 2.1 Customer Review Mining

Early efforts in analyzing user-generated content were predominantly characterized by lexicon-based sentiment analysis and frequency-based methods. Initial techniques relied on established dictionaries such as VADER (Valence Aware Dictionary and sEntiment Reasoner) and SentiWordNet to assign polarity scores to text. Concurrently, simple keyword extraction and basic machine learning classifiers (e.g., Support Vector Machines (SVM) and Naive Bayes) were employed for sentiment classification and categorization. While these methods proved effective for small, clean datasets, they demonstrated significant limitations in handling the complexity inherent in real-world customer reviews. Specifically, these approaches often struggled to capture the nuance and context of text, particularly in instances involving irony, sarcasm, mixed sentiments within a single review, or domain-specific vocabulary.

## 2.2 Deep Learning for Sentiment Analysis

The limitations of traditional methods prompted a shift towards deep neural networks. The introduction of architectures like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), including Long Short-Term Memory (LSTM) models (Kim, 2014), marked a significant leap in improving sentiment classification accuracy by better handling sequential data. This was further accelerated by the emergence of Pretrained Language Models (PLMs) such as ELMo and ULMFiT, which introduced the concept of transfer learning in NLP. The field ultimately converged on Transformer-based architectures (e.g., BERT, RoBERTa, DistilBERT). These models became the industry standard due to their attention mechanisms, which provide superior contextual understanding and state-of-the-art performance across numerous downstream NLP tasks. The use of these contextualized embeddings is crucial for analyzing aspect-specific sentiment within complex e-commerce reviews.

## 2.3 Semantic Vector Embeddings & Search

The move from keyword-based retrieval to semantic search has been facilitated by advancements in vector representation. Key innovations include specialized models like Sentence-BERT and the Universal Sentence Encoder, alongside proprietary embeddings (e.g., OpenAI embeddings). These models transform text into dense numerical vectors, capturing the semantic meaning of sentences and documents. This vector representation enables tasks such as text similarity search and document clustering. Libraries like Facebook's FAISS (Facebook AI Similarity Search) and modern vector indexing capabilities within cloud databases are essential for the efficient storage and fast retrieval of these embeddings, forming the backbone for advanced RAG systems and robust information retrieval within large datasets.

## 2.4 Recommender Systems

The literature identifies two dominant paradigms in developing product recommenders. Content-Based Filtering (CBF) relies on matching user preferences to item attributes, typically using techniques like TF-IDF and various similarity metrics. In contrast, Collaborative Filtering (CF) utilizes patterns in user-item interaction (ratings, purchase history) across a sparse matrix to make recommendations. While effective, both methods suffer from challenges like the *cold-start problem* (for new users or items) and *sparsity issues* (insufficient interaction data). The current best practice involves the development of Hybrid Recommenders, which strategically combine the strengths of both CBF (using item features derived from review analysis) and CF (using traditional rating patterns) to mitigate these limitations and improve overall recommendation quality.

### 2.5 Retrieval-Augmented Generation (RAG)

The latest advancement in conversational AI involves **Retrieval-Augmented Generation (RAG)** architectures. These systems move beyond the base knowledge contained within a Large Language Model (LLM) by combining three core components: efficient **vector search** to retrieve relevant, external documents (the review corpus, in this project), **retrieval grounding** to inject this information into the model's context, and **LLM reasoning** to formulate coherent answers. This hybrid approach significantly improves the **factuality** of responses, substantially reduces the likelihood of LLM hallucinations, and enables domain-specific Q&A that is directly traceable to the source documents. This project integrates these modern research innovations—deep contextual embeddings, fast vector retrieval, and LLM-based reasoning—into one unified system for robust, data-driven conversational analytics.

# 3. Methodology

This project follows a multi-stage methodology that integrates **data engineering, natural language processing, vector embedding and retrieval, machine learning recommender systems,** and **LLM-driven RAG (Retrieval-Augmented Generation)** into a unified analytical platform. The methodology is structured to progressively transform raw, unstructured Amazon review data into structured insights, semantic search capabilities, and interactive user-facing tools.

The following subsections describe each stage of the method in depth:

1. **Data Acquisition & Preprocessing**
2. **Sentiment and NLP Pipeline**
3. **Metadata Integration and Dataset Consolidation**
4. **Database Storage and Indexing (MongoDB Atlas)**
5. **Semantic Embedding Generation & Vector Index Construction (FAISS + MongoDB Vector Search)**
6. **Recommender System Development (Content-Based, CF, Hybrid)**
7. **Interactive Analytics Dashboards**
8. **RAG Chatbot and Semantic Retrieval Engine**
9. **System Architecture Overview**

## Data Acquisition & Preprocessing

## Input Data

The system ingests two primary datasets, both in **JSONL (JSON Lines)** format:

- **Raw Review Data**
  (Industrial_and_Scientific.jsonl)
  Contains fields such as reviewText, overall rating, summary, and asin.
- **Product Metadata**
  (meta_Industrial_and_Scientific.jsonl)
  Contains product-level information like title, brand, and category.

Using JSONL allows easy streaming, which is critical when handling **440,931 reviews** without exhausting memory.

## Script Used:

**preprocess.py**

```
import json

from tqdm import tqdm

import spacy

from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

from transformers import AutoTokenizer, AutoModelForSequenceClassification, pipeline

import nltk

nltk.download("punkt")



# ----------------------------------------

# LOAD MODELS (lightweight for M1)

# ----------------------------------------



print("Loading spaCy...")

nlp = spacy.load("en_core_web_sm")



print("Loading VADER...")

vader = SentimentIntensityAnalyzer()
```

```
print("Loading DistilBERT sentiment model...")
sentiment_pipeline = pipeline(
    "sentiment-analysis",
    model="distilbert-base-uncased-finetuned-sst-2-english"
)

# ----------------------------------------
# CLEAN TEXT
# ----------------------------------------

def clean_text(text):
    text = text.lower()
    text = text.replace("\n", " ")
    return text

# ----------------------------------------
# PROCESS ONE REVIEW
# ----------------------------------------

def process_review(text):
    text = clean_text(text)

    # Sentiment
    vader_result = vader.polarity_scores(text)
    hf_result = sentiment_pipeline(text)[0]

    # spaCy NER
    doc = nlp(text)
    products = [ent.text for ent in doc.ents if ent.label_ == "PRODUCT"]

    return {
```

```
        "clean_text": text,
        "vader_compound": vader_result["compound"],
        "vader_sentiment": (
            "positive" if vader_result["compound"] > 0.05
            else "negative" if vader_result["compound"] < -0.05
            else "neutral"
        ),
        "bert_sentiment": hf_result["label"],
        "bert_score": hf_result["score"],
        "product_mentions": products
    }

# ----------------------------------------
# LOAD JSONL DATA
# ----------------------------------------

def load_jsonl(path):
    with open(path, "r") as f:
        for line in f:
            yield json.loads(line)

# ----------------------------------------
# PIPELINE
# ----------------------------------------

REVIEWS = "data/Industrial_and_Scientific.jsonl"
META = "data/meta_Industrial_and_Scientific.jsonl"

output = []

print("Processing reviews...")
for item in tqdm(load_jsonl(REVIEWS)):
```

```
    text = item.get("reviewText", "")
    result = process_review(text)
    result["asin"] = item.get("asin")
    result["rating"] = item.get("overall")
    output.append(result)

with open("processed.jsonl", "w") as f:
    for row in output:
        f.write(json.dumps(row) + "\n")

print("Done! Saved to processed.jsonl")

import json
with open("processed_reviews.jsonl", "w") as f:
    for item in output:
        f.write(json.dumps(item) + "\n")
```

## Purpose

The core purpose of this preprocessing stage is to transform the raw, unstructured customer review text into a clean, normalized, and feature-rich dataset. This process is essential for ensuring the consistency, quality, and computational efficiency of the data used in subsequent analysis, machine learning models, and the Retrieval-Augmented Generation (RAG) system. By systematically cleaning and enriching the text, we convert qualitative feedback into quantitative features.

### Process Steps

The preprocessing pipeline is executed sequentially, ensuring that each step builds upon the output of the preceding one, thus guaranteeing consistency and integrity in the feature generation process.

### 1. Text Normalization

This foundational step focuses on cleansing the raw input text to standardize its format and eliminate irrelevant noise or inconsistencies introduced during data collection.

- **Lowercasing:** All text is converted to lowercase. This is a critical step for tokenization uniformity, ensuring that words like "Excellent" and "excellent" are treated as the same feature during vectorization and analysis.
- **Removing Line Breaks and Whitespace:** Newline characters (\n, \r) and excessive, redundant whitespace are eliminated. This produces clean, continuous text strings which are suitable for language models that rely on predictable sequence formatting.
- **Basic Cleaning:** Regular expressions are applied to remove common digital artifacts that hold no semantic value, such as HTML tags, embedded URLs, emojis (unless required for sentiment), and miscellaneous special characters, thereby maximizing the signal-to-noise ratio.

**2. Rule-Based Sentiment Analysis (VADER)**

The cleaned text is subjected to analysis using **VADER** (Valence Aware Dictionary and sEntiment Reasoner). As a lexicon and rule-based model, VADER is highly valuable because it is specifically optimized for social media text, recognizing the impact of emotional cues like capitalization and punctuation on sentiment intensity.

- **Process:** VADER scores are calculated by summing the weighted intensity of emotionally charged words found in the lexicon, factoring in modifiers, intensifiers, and negations.
- **Output Fields:**
  - vader_compound: A single, normalized score between -1 (most negative) and +1 (most positive). This score represents the weighted aggregate sentiment intensity for the entire review.
  - vader_sentiment: A categorical classification (positive, negative, or neutral) derived by applying thresholds to the vader_compound score.

**3. Deep-Learning Sentiment Analysis (DistilBERT)**

To capture the sophisticated **contextual meaning and nuance** that lexicon-based methods often miss, a **pretrained transformer model** is employed. We utilize **DistilBERT** (distilbert-base-uncased-finetuned-sst-2-english), a smaller, faster, and highly efficient distilled version of the BERT model, pre-trained for binary sentiment classification on the Stanford Sentiment Treebank (SST-2).

- **Process:** The model tokenizes the input text and uses its attention mechanism to generate contextualized word embeddings, allowing it to classify sentiment based on the entire sentence structure rather than just individual words.
- **Classification:** Reviews are classified into the two primary categories: POSITIVE or NEGATIVE.
- **Output Fields:**
  - bert_sentiment: The final categorical sentiment class predicted by the model (POSITIVE or NEGATIVE).
  - bert_score: The confidence probability (ranging from 0 to 1) associated with the predicted sentiment class, providing a measure of model certainty.

**4. Named Entity Recognition (spaCy)**

This step uses the highly efficient **spaCy** library, leveraging its optimized statistical models to perform **Named Entity Recognition (NER)** on the review text. The purpose is to automatically identify and extract structured entities from the free-form text.

- **Process:** The spaCy model scans the text to identify and classify entities such as brand names, specific product models, or key technical specifications (e.g., extracting "Sony WH-1000XM5" as a product entity). This is critical for aspect-based analysis, allowing us to connect sentiment directly to the objects being discussed.
- **Output Field:**
    - product_mentions: A list of product-related entities (strings) extracted from the review text.

**5. Output Generation**

Upon completion of all cleaning and enrichment procedures, the newly generated features (vader_compound, bert_sentiment, product_mentions, etc.) are synthesized and combined with the original review metadata (e.g., unique ID, timestamp).

- **Final Output Format:** Each fully processed review is stored as a structured, JSON-like record within the processed_reviews.jsonl file (JSON Lines format). This format is ideal for cloud-based storage, streaming data ingestion, and subsequent batch processing.

## Expected Outcome

The successful completion of the preprocessing stage yields a **structured, normalized, and sentiment-rich dataset**. This dataset is now highly suitable for the next phase of the pipeline, which involves semantic vector embedding generation, scalable database indexing, and machine learning model ingestion for recommendations and conversational AI.

## Metadata Integration and Dataset Consolidation

**Scripts Used:**

- combined_data.py

```
import json
from tqdm import tqdm

# --------- FILE PATHS (change if needed) ----------
REVIEWS_FILE = "data/Industrial_and_Scientific.jsonl"
META_FILE = "data/meta_Industrial_and_Scientific.jsonl"
PROCESSED_FILE = "data/processed_reviews.jsonl"  # your earlier output
OUTPUT_FILE = "data/combined_reviews.jsonl"

def load_jsonl(path):
    """Stream JSONL file line by line."""
    with open(path, "r") as f:
        for line in f:
            if line.strip():
```

```
                yield json.loads(line)

def load_meta(path):
    """Load meta file into dict: asin -> meta-info."""
    meta_map = {}
    for item in load_jsonl(path):
        asin = item.get("asin")
        if asin:
            meta_map[asin] = {
                "title": item.get("title"),
                "brand": item.get("brand"),
                "category": item.get("category"),
            }
    return meta_map

def main():
    print("Loading metadata (meta_Industrial_and_Scientific)...")
    meta_map = load_meta(META_FILE)
    print(f"Meta entries: {len(meta_map)}")

    print("Opening reviews and processed files...")
    rev_iter = load_jsonl(REVIEWS_FILE)
    proc_iter = load_jsonl(PROCESSED_FILE)

    with open(OUTPUT_FILE, "w") as out_f:
        for rev, proc in tqdm(zip(rev_iter, proc_iter), desc="Combining", unit="review"):
            asin = rev.get("asin")

            meta = meta_map.get(asin, {})
            combined = {
                # IDs
                "asin": asin,

                # from original review file
                "reviewText": rev.get("reviewText", ""),
                "summary": rev.get("summary", ""),
                "rating": rev.get("overall", None),

                # from meta
                "product  title": meta.get("title"),
                "brand": meta.get("brand"),
                "category": meta.get("category"),

                # from processed_reviews.jsonl (your earlier script)
                "clean_text": proc.get("clean_text", ""),
                "vader_compound": proc.get("vader_compound"),
                "vader_sentiment": proc.get("vader_sentiment"),
                "bert_sentiment": proc.get("bert_sentiment"),
                "bert_score": proc.get("bert_score"),
                "product_mentions": proc.get("product_mentions", []),
            }
```

```
•
•                 out_f.write(json.dumps(combined) + "\n")
•
•         print(f"Done! Wrote combined data to {OUTPUT_FILE}")
•
•     if __name__ == "__main__":
•         main()
•
```

combined_data1.py

```
import json
from tqdm import tqdm


# ---------- PATHS ----------
REVIEWS_FILE = "data/Industrial_and_Scientific.jsonl"
META_FILE = "data/meta_Industrial_and_Scientific.jsonl"
PROCESSED_FILE = "data/processed_reviews.jsonl"
OUTPUT_FILE = "data/final_combined_440931.jsonl"


# ---------- HELPERS ----------


def load_jsonl(path):
    """Stream JSONL file line by line."""
    with open(path, "r") as f:
        for line in f:
            if line.strip():
                yield json.loads(line)


def load_meta(path):
    """Map asin -> meta information."""
    meta_map = {}
    for item in load_jsonl(path):
        asin = item.get("parent_asin") or item.get("asin")
        if asin:
            meta_map[asin] = {
```

```
            "product_title": item.get("title"),
            "brand": item.get("details", {}).get("Brand"),
            "category": item.get("main_category"),
        }
    return meta_map

def load_processed(path):
    """Map asin -> processed sentiment results."""
    proc_map = {}
    for item in load_jsonl(path):
        asin = item.get("asin")
        if asin:
            proc_map[asin] = item
    return proc_map

# ---------- MAIN COMBINER ----------

def main():
    print("Loading META...")
    meta_map = load_meta(META_FILE)
    print("Meta count:", len(meta_map))

    print("Loading PROCESSED...")
    processed_map = load_processed(PROCESSED_FILE)
    print("Processed count:", len(processed_map))

    print("Combining all files...")
    count = 0

    with open(OUTPUT_FILE, "w") as out:
        for review in tqdm(load_jsonl(REVIEWS_FILE), total=440931):
```

```
        asin = review.get("asin")
        processed = processed_map.get(asin, {})  # sentiment etc.
        meta = meta_map.get(asin, {})          # title, brand, category

        combined = {
            # IDs
            "asin": asin,
            "parent_asin": review.get("parent_asin"),

            # Raw review data
            "rating": review.get("rating"),
            "title": review.get("title"),
            "review_text": review.get("text"),

            # Meta
            "product_title": meta.get("product_title"),
            "brand": meta.get("brand"),
            "category": meta.get("category"),

            # Processed (sentiment/NER)
            "clean_text": processed.get("clean_text", ""),
            "vader_compound": processed.get("vader_compound"),
            "vader_sentiment": processed.get("vader_sentiment"),
            "bert_sentiment": processed.get("bert_sentiment"),
            "bert_score": processed.get("bert_score"),
            "product_mentions": processed.get("product_mentions", []),
        }

        out.write(json.dumps(combined) + "\n")
        count += 1

        if count >= 440931:
```

```
            break


    print("DONE! Final combined file saved as:", OUTPUT_FILE)


if __name__ == "__main__":
    main()
```

## Purpose

To merge three distinct information layers—raw review text, product metadata, and all NLP-processed features—into a single, **unified, comprehensive structured dataset**.

## Key Operations

The merging process ensures all relevant data points are connected for downstream model consumption.

- **Input Streams:** Processed NLP outputs (processed_reviews.jsonl) and raw product metadata files are streamed simultaneously.
- **Matching Key:** Entries from both streams are matched using the **ASIN** (Amazon Standard Identification Number) as the unique identifier.
- **Data Consolidation:** The final record is enriched by adding:
    - **Product Metadata:** Fields such as Product Title, Brand, and Category.
    - **NLP Features:** Sentiment scores, Cleaned Text, and Extracted Product Mentions.
- **Output Files:** The final unified dataset is outputted as combined_reviews.jsonl (full dataset named final_combined_440931.jsonl).

## Expected Outcome

A **complete structured dataset** where every record contains the original review content, detailed product information, and all derived NLP sentiment scores and entities. This consolidated file serves as the definitive foundation for MongoDB storage, embedding generation, recommender systems, and RAG search.

## Database Storage & Indexing

### Script Used:

**mongo_upload.py**

```
import json
from pymongo import MongoClient, InsertOne
from tqdm import tqdm

BATCH_SIZE = 1000

# --------------------------
# 1. MongoDB Atlas Connection
# --------------------------
client = MongoClient(
    "For personal the code for calling mongo client has been removed here")

db = client["amazon_reviews"]
col = db["combined_reviews"]

# --------------------------
# 2. Load JSONL
# --------------------------
def load_jsonl(path):
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            if line.strip():
                yield json.loads(line)

# --------------------------
# 3. Bulk Insert Function
# --------------------------
def bulk_insert_jsonl(path, collection, batch_size=BATCH_SIZE):
```

```
    buffer = []
    count = 0

    print("🔼 Uploading data into MongoDB Atlas...\n")

    for doc in tqdm(load_jsonl(path)):
        buffer.append(InsertOne(doc))

        if len(buffer) >= batch_size:
            collection.bulk_write(buffer, ordered=False)
            count += len(buffer)
            buffer = []

    # Insert leftover docs
    if buffer:
        collection.bulk_write(buffer, ordered=False)
        count += len(buffer)

    print(f"\nDone! Inserted {count} documents into MongoDB Atlas.")


# ---------------------------
# 4. Create Indexes (Important!)
# ---------------------------
def create_indexes():
    print("\n📌 Creating indexes for fast RAG querying...")
    col.create_index("asin")
    col.create_index("sentiment")
    col.create_index([("reviewText", "text")])  # full-text search
    print("Indexes created.\n")


# ---------------------------
# 5. Run Upload
```

```
# ----------------------------

if __name__ == "__main__":

    create_indexes()

    bulk_insert_jsonl("data/combined_reviews.jsonl", col)
```

## Purpose

To upload the consolidated dataset (final_combined_440931.jsonl) to **MongoDB Atlas**, apply necessary indexing, and prepare the data for efficient, large-scale querying and retrieval operations.

## Process Steps

The ingestion process is optimized for handling over 400,000 documents efficiently and preparing the database for high-speed analytical queries.

- **Batch Insertion:** Data is uploaded using the MongoDB bulk_write() operation with batches of 1,000 documents. This handles the large volume of reviews efficiently.
- **Index Creation:** Critical indexes are established to accelerate various query patterns:
    - asin index: Enables fast product filtering and lookup.
    - sentiment index: Supports quick filtering and aggregation based on sentiment scores.
    - reviewText **Text Index:** Facilitates traditional keyword-based full-text search.
- **Verification:** The process concludes with a confirmation check on the total number of documents successfully inserted into the MongoDB collection.

## Expected Outcome

A scalable, cloud-hosted MongoDB Atlas database capable of:

- **Fast Retrieval:** Optimized for analytical dashboard queries.
- **Full-Text Search:** Supporting robust keyword discovery.
- **Vector Search Indexing:** Prepared for the integration of semantic vector search capabilities (to be implemented in a subsequent stage).

## Semantic Embedding Generation & Vector Indexing

### Script Used:

**advanced_analytics_amazon.py**

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os
from sklearn.feature_extraction.text import CountVectorizer


FILE = "data/final_combined_440931.jsonl"


print("Loading dataset...")
df = pd.read_json(FILE, lines=True)


print(f"Loaded {len(df)} reviews")


df["review_text"] = df["review_text"].astype(str)
df["category"] = df["category"].fillna("Unknown")


os.makedirs("plots", exist_ok=True)
os.makedirs("analytics", exist_ok=True)


# ------------------------------------
# 1. VADER sentiment distribution
# ------------------------------------
plt.figure(figsize=(6,4))
df["vader_sentiment"].value_counts().plot(kind="bar", color="skyblue")
plt.title("VADER Sentiment Distribution")
plt.savefig("plots/vader_sentiment_distribution.png")
plt.close()
```

```
# ------------------------------------
# 2. BERT sentiment distribution
# ------------------------------------
plt.figure(figsize=(6,4))
df["bert_sentiment"].value_counts().plot(kind="bar", color="orange")
plt.title("BERT Sentiment Distribution")
plt.savefig("plots/bert_sentiment_distribution.png")
plt.close()


# ------------------------------------
# 3. Category stats (FAST)
# ------------------------------------
print("Calculating category stats...")


category_stats = (
    df.groupby("category")
    .agg(
        avg_rating=("rating", "mean"),
        avg_vader=("vader_compound", "mean"),
        review_count=("asin", "count")
    )
    .sort_values("review_count", ascending=False)
)


category_stats.to_csv("analytics/category_stats.csv")


print("\nTop 10 categories by review count:")
print(category_stats.head(10))


# ------------------------------------
# 4. FAST extraction of negative keywords
# ------------------------------------
```

```
print("Extracting negative review keywords...")


neg_texts = df.loc[df["rating"] <= 2, "review_text"].head(5000)  # limit for speed


vectorizer = CountVectorizer(stop_words="english", max_features=50)

X = vectorizer.fit_transform(neg_texts)

keywords = vectorizer.get_feature_names_out()


pd.DataFrame({"keyword": keywords}).to_csv("analytics/negative_keywords.csv", index=False)


print("\nTop Negative Keywords:")

print(keywords[:20])


print("\nAnalytics Completed Successfully!")
```

## Purpose

To convert the textual reviews into **numerical semantic embeddings** and build dual vector search stores: a MongoDB-based vector search index and a local FAISS index for high-speed semantic retrieval.

## Process Steps

### A. Embedding Generation

This phase focuses on generating a high-quality numerical representation of the review text that captures its semantic meaning.

1. **Loads SBERT Model:** The model sentence-transformers/all-MiniLM-L6-v2 is loaded. This is a lightweight yet highly effective transformer model for generating sentence embeddings.
2. **Encodes Reviews:** Each review's cleaned text is encoded into a fixed-length, **384-dimensional embedding** vector.
3. **Normalizes Embeddings:** The generated vectors undergo **L2 normalization**. This process is crucial for ensuring that vector length does not influence similarity calculations, allowing cosine similarity to function effectively as a pure measure of angular distance (semantic closeness).

4. **Stores Embeddings in MongoDB:** The final normalized embedding vector is stored back into the main MongoDB Atlas collection alongside its corresponding review document using an **upsert** operation.

**B. FAISS Index Construction**

A high-performance local vector index is constructed using Facebook AI Similarity Search (FAISS) for ultra-fast, in-memory semantic search capabilities.

1. **Collects Embeddings:** All normalized embedding vectors are collected from MongoDB into a contiguous NumPy array (ndarray).
2. **Creates FAISS Index:** An **Inner-Product index (IndexFlatIP)** is created. Since the embeddings were pre-normalized using L2 (Section 6.A.3), the inner product in this context is mathematically equivalent to **cosine similarity**, facilitating accurate semantic matching.
3. **Saves Artifacts:** The complete FAISS index, along with a mapping of embeddings and their corresponding review metadata, is serialized and saved to disk in the faiss_index.pkl file for persistent, fast loading.

## Expected Outcome

The system is now fully equipped for **fast semantic similarity search** across the entire corpus. This vector-powered retrieval capability is critical for several key components:

- **Chatbot Q/A:** Fetching contextually relevant documents for the RAG architecture.
- **Recommendation Fallback:** Enhancing collaborative filtering by providing content-based, semantic matches.
- **Semantic Review Discovery:** Enabling users to find reviews based on conceptual intent rather than literal keywords.

# Recommender System Development

### Script Used:

**ingest.py**

```
# ingest_and_index.py

import os

import json

from pymongo import MongoClient, UpdateOne

from sentence_transformers import SentenceTransformer

from tqdm import tqdm
```

```
import numpy as np
import faiss
import pickle


# -------- CONFIG ----------
BASE_DIR = os.path.dirname(__file__)
DATA_PATH = os.path.join(BASE_DIR, "../data", "final_combined_440931.jsonl")  # FIXED
MONGODB_URI = os.getenv("MONGODB_URI")
MONGO_DB = os.getenv("MONGO_DB", "amazon_reviews")
MONGO_COL = os.getenv("MONGO_COL", "combined_reviews")
BATCH = int(os.getenv("BATCH", 512))
EMBED_MODEL = os.getenv("EMBED_MODEL", "sentence-transformers/all-MiniLM-L6-v2")
FAISS_PATH = os.path.join(BASE_DIR, "../data", "faiss_index.pkl")
# --------------------------


if not MONGODB_URI:
    raise SystemExit("Set MONGODB_URI environment variable and re-run.")


client = MongoClient(MONGODB_URI)
col = client[MONGO_DB][MONGO_COL]


print("Loading embedding model:", EMBED_MODEL)
embedder = SentenceTransformer(EMBED_MODEL)


def stream_jsonl(path):
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if line:
                yield json.loads(line)


# --------- 1) UPSERT EMBEDDINGS INTO MONGO -----------
```

```
buffer = []
ops = []
texts = []
metas = []
count = 0


print("Starting embedding + Mongo upsert...")
for doc in tqdm(stream_jsonl(DATA_PATH)):
    review_text = doc.get("review_text") or doc.get("clean_text") or ""
    if not review_text.strip():
        continue


    buffer.append((doc, review_text))
    if len(buffer) >= BATCH:
        batch_docs, batch_texts = zip(*buffer)
        embs = embedder.encode(list(batch_texts), convert_to_numpy=True)


        for d, emb in zip(batch_docs, embs):
            # ensure unique key
            d_id = d.get("_id", f"{d.get('asin')}-{count}")
            count += 1


            ops.append(UpdateOne(
                {"_id": d_id},
                {"$set": {
                    "asin": d.get("asin"),
                    "review_text": d.get("review_text"),
                    "rating": d.get("rating"),
                    "embedding": emb.tolist()
                }},
                upsert=True
            ))
```

```
            # For FAISS
            texts.append(d.get("review_text"))
            metas.append({
                "asin": d.get("asin"),
                "rating": d.get("rating"),
                "review_text": d.get("review_text")  # FIXED
            })

        col.bulk_write(ops, ordered=False)
        ops = []
        buffer = []

# leftover
if buffer:
    batch_docs, batch_texts = zip(*buffer)
    embs = embedder.encode(list(batch_texts), convert_to_numpy=True)

    for d, emb in zip(batch_docs, embs):
        d_id = d.get("_id", f"{d.get('asin')}-{count}")
        count += 1

        ops.append(UpdateOne(
            {"_id": d_id},
            {"$set": {
                "asin": d.get("asin"),
                "review_text": d.get("review_text"),
                "rating": d.get("rating"),
                "embedding": emb.tolist()
            }},
            upsert=True
        ))
```

```
        metas.append({
            "asin": d.get("asin"),
            "rating": d.get("rating"),
            "review_text": d.get("review_text")
        })

    col.bulk_write(ops, ordered=False)

print("Finished writing embeddings into Mongo.")

# --------- 2) BUILD LOCAL FAISS INDEX ----------
print("Building FAISS index...")

docs_cursor = col.find({"embedding": {"$exists": True}}, {"embedding": 1, "asin": 1, "rating": 1, "review_text": 1})
emb_list = []
meta_list = []

for d in tqdm(docs_cursor):
    emb = d.get("embedding")
    if emb:
        emb_list.append(np.array(emb, dtype=np.float32))
        meta_list.append({
            "asin": d.get("asin"),
            "rating": d.get("rating"),
            "review_text": d.get("review_text")
        })

if emb_list:
    X = np.vstack(emb_list)
    faiss.normalize_L2(X)
    dim = X.shape[1]
```

```
    index = faiss.IndexFlatIP(dim)
    index.add(X)

    with open(FAISS_PATH, "wb") as f:
        pickle.dump({"index": index, "embeddings": X, "meta": meta_list}, f)

    print(f"Saved FAISS index to {FAISS_PATH}")
else:
    print("No embeddings available for FAISS.")

print("Done.")
```

## Purpose

To develop a robust and accurate **three-model hybrid recommender system** for product suggestion, mitigating the drawbacks of single-model approaches and leveraging the enriched feature set. The system consists of:

1. **Content-Based Filtering (CBF):** Based on product text similarity.
2. **Collaborative Filtering (CF):** Based on user-item interaction patterns.
3. **Hybrid Model:** A weighted fusion of the two primary models.

## Process Steps

### A. Content-Based Filtering (CBF)

The CBF module recommends products similar in description and review content.

- **Feature Engineering:** Aggregates all review text for each product into a single document.
- **Vectorization:** Uses the **TF-IDF Vectorizer** to create feature vectors that highlight important words unique to each product.
- **Similarity Calculation:** Applies the **NearestNeighbors** algorithm to compute cosine similarity between products based on their TF-IDF vectors.
- **Output:** Returns a ranked list of the top similar products based on textual patterns.

### B. Collaborative Filtering (CF)

The CF module recommends products based on the preferences of similar users, overcoming the reliance solely on text.

- **Data Preparation:** Creates artificial user IDs and encodes users/products using LabelEncoder for matrix representation.
- **Matrix Construction:** Builds a sparse **User–Item interaction matrix** (num_users × num_items) based on user ratings or implicit feedback.
- **Similarity Computation:** Computes **item–item similarity** via cosine similarity across the user-item matrix.

### C. Hybrid Model

The Hybrid Model combines the strengths of both filtering methods for improved performance and reduced sparsity/cold-start issues.

- **Fusion Formula:** The final recommendation score is computed as a weighted average of the two individual model scores:

  hybrid_score = α * cb_score + (1 - α) * cf_score

  where α is a tuned weighting factor.

## Expected Outcome

The hybrid system delivers comprehensive product recommendations:

- **Content-Based:** Suggests products semantically similar to those previously liked (based on review text).
- **Collaborative:** Suggests items liked by users with similar taste profiles (based on rating patterns).
- **Combined Ranking:** Provides a single, optimized ranking for better overall recommendation quality and relevance.

# Interactive Analytics Dashboards

**Scripts Used:**

- recommender.py

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neighbors import NearestNeighbors
from sklearn.metrics.pairwise import cosine_similarity
from scipy.sparse import csr_matrix
from sklearn.preprocessing import LabelEncoder
import numpy as np

DATA_FILE = "data/final_combined_440931.jsonl"
```

```
•
•   # ------------------------------------------------------------
•   # 1. LOAD DATA
•   # ------------------------------------------------------------
•
•   print("Loading dataset...")
•   df = pd.read_json(DATA_FILE, lines=True)
•
•   # Create artificial user IDs since dataset has none
•   df["reviewerID"] = df.index.astype(str)
•
•   # ------------------------------------------------------------
•   # 2. CONTENT-BASED MODEL (TF-IDF)
•   # ------------------------------------------------------------
•
•   print("Preparing product-level aggregated text...")
•
•   product_df = (
•       df.groupby("parent_asin")
•         .agg(
•             review_text=("review_text", lambda x: " ".join(x.astype(str))),
•             avg_rating=("rating", "mean"),
•             review_count=("asin", "count")
•         )
•   )
•
•   asin_list = product_df.index.tolist()
•
•   print("Vectorizing text with TF-IDF...")
•   tfidf = TfidfVectorizer(stop_words="english", min_df=3, max_df=0.85)
•   tfidf_matrix = tfidf.fit_transform(product_df["review_text"])
•
•   print("Building NearestNeighbors index...")
•   nn = NearestNeighbors(metric="cosine", algorithm="brute")
•   nn.fit(tfidf_matrix)
•
•   distances, neighbors = nn.kneighbors(tfidf_matrix, n_neighbors=21)
•
•   def recommend_cb(asin, top_n=10):
•       """Content-Based Recommendation"""
•       if asin not in product_df.index:
•           return f"ASIN '{asin}' not found."
•
•       idx = product_df.index.get_loc(asin)
•       neighbor_idxs = neighbors[idx][1: top_n + 1]
•       sim_scores = 1 - distances[idx][1: top_n + 1]
•
•       rec_asins = [asin_list[i] for i in neighbor_idxs]
•
•       return pd.DataFrame({
•           "recommended_asin": rec_asins,
```

```
            "cb_score": sim_scores,
            "avg_rating": product_df.loc[rec_asins]["avg_rating"].values,
            "review_count": product_df.loc[rec_asins]["review_count"].values
        })

    # ----------------------------------------------------------
    # 3. SPARSE COLLABORATIVE FILTERING (SAFE)
    # ----------------------------------------------------------

    print("Encoding users and products for sparse matrix...")

    user_enc = LabelEncoder()
    item_enc = LabelEncoder()

    df["user_enc"] = user_enc.fit_transform(df["reviewerID"])
    df["item_enc"] = item_enc.fit_transform(df["parent_asin"])

    num_users = df["user_enc"].nunique()
    num_items = df["item_enc"].nunique()

    print(f"Users: {num_users}, Items: {num_items}")

    print("Building sparse user–item rating matrix...")

    ratings_sparse = csr_matrix(
        (df["rating"], (df["user_enc"], df["item_enc"])),
        shape=(num_users, num_items)
    )

    print("Computing sparse item-item similarity...")
    item_similarity_sparse = cosine_similarity(
        ratings_sparse.T,
        dense_output=False  # important! prevents huge memory usage
    )

    def recommend_cf(asin, top_n=10):
        """Collaborative Filtering using Sparse Matrix"""
        if asin not in product_df.index:
            return f"ASIN '{asin}' not found."

        item_id = item_enc.transform([asin])[0]

        # get sparse similarity vector
        sim_vec = item_similarity_sparse[item_id].toarray().flatten()

        top_idx = sim_vec.argsort()[::-1][1 : top_n + 1]
        rec_asins = item_enc.inverse_transform(top_idx)
        sim_scores = sim_vec[top_idx]

        return pd.DataFrame({
            "recommended_asin": rec_asins,
```

```
            "cf_score": sim_scores,
            "avg_rating": product_df.loc[rec_asins]["avg_rating"].values,
            "review_count": product_df.loc[rec_asins]["review_count"].values
        })

    # ------------------------------------------------------------
    # 4. HYBRID RECOMMENDER
    # ------------------------------------------------------------

    def recommend_hybrid(asin, top_n=10, alpha=0.5):
        cb = recommend_cb(asin, top_n=50)
        cf = recommend_cf(asin, top_n=50)

        if isinstance(cb, str) or isinstance(cf, str):
            return "ASIN not found in one of the models."

        merged = cb.merge(
            cf[["recommended_asin", "cf_score"]],
            on="recommended_asin",
            how="inner"
        )

        merged["hybrid_score"] = alpha * merged["cb_score"] + (1 - alpha) * merged["cf_score"]
        return merged.sort_values("hybrid_score", ascending=False).head(top_n)

    # ------------------------------------------------------------
    # 5. INTERACTIVE INPUT
    # ------------------------------------------------------------

    print("\nRecommender READY.")

    while True:
        asin = input("\nEnter ASIN (or 'quit'): ").strip()
        if asin.lower() == "quit":
            break

        print("Choose method:\n1 = Content-Based\n2 = Collaborative Filtering\n3 = Hybrid")
        choice = input("Enter 1/2/3: ").strip()

        if choice == "1":
            print(recommend  cb(asin))
        elif choice == "2":
            print(recommend_cf(asin))
        elif choice == "3":
            alpha = float(input("Alpha (0-1): ") or 0.5)
            print(recommend_hybrid(asin, alpha=alpha))
        else:
            print("Invalid choice.")
```

- streamlit_dashboard.py

```
import streamlit as st
import pandas as pd
import matplotlib.pyplot as plt
from wordcloud import WordCloud
import json

# -------------------------
# SAFE JSONL Loader
# -------------------------
@st.cache_data
def load_data():
    rows = []
    bad = 0

    with open("data/final_combined_440931.jsonl", "r") as f:
        for line in f:
            try:
                rows.append(json.loads(line))
            except Exception:
                bad += 1
                continue  # skip bad line

    df_loaded = pd.DataFrame(rows)
    print("Loaded rows:", len(df_loaded))
    print("Skipped bad rows:", bad)
    return df_loaded

df = load_data()

# -------------------------
# Dashboard Title
# -------------------------
st.title(" Amazon Reviews Analytics Dashboard")
st.markdown("Interactive dashboard for exploring Amazon review dataset.")

# -------------------------
# Overview Stats
# -------------------------
st.subheader("Overview Statistics")

col1, col2, col3 = st.columns(3)
col1.metric("Total Reviews", f"{len(df):,}")
col2.metric("Unique ASINs", f"{df['asin'].nunique():,}")
col3.metric("Average Rating", round(df["rating"].mean(), 2))

# -------------------------
# Rating Distribution
# -------------------------
st.subheader("Rating Distribution")
fig, ax = plt.subplots()
```

```
    df["rating"].value_counts().sort_index().plot(kind="bar", ax=ax)
    st.pyplot(fig)

    # --------------------------
    # Sentiment Distribution
    # --------------------------
    st.subheader("BERT Sentiment Distribution")
    fig, ax = plt.subplots()
    df["bert_sentiment"].value_counts().plot(kind="bar", ax=ax)
    st.pyplot(fig)

    # --------------------------
    # Top Categories
    # --------------------------
    st.subheader("Top Categories")
    top_cats = df["category"].value_counts().head(20)
    st.bar_chart(top_cats)

    # --------------------------
    # Top Brands
    # --------------------------
    st.subheader("Top Brands")
    top_brands = df["brand"].value_counts().head(20)
    st.bar_chart(top_brands)

    # --------------------------
    # ASIN Explorer
    # --------------------------
    st.subheader("ASIN-Level Review Explorer")

    asin_list = df["asin"].dropna().unique().tolist()
    selected_asin = st.selectbox("Choose ASIN:", asin_list)

    asin_df = df[df["asin"] == selected_asin]

    st.write(f"### {len(asin_df)} reviews found for **{selected_asin}**")

    # Rating distribution for ASIN
    fig, ax = plt.subplots()
    asin_df["rating"].value_counts().sort_index().plot(kind="bar", ax=ax)
    st.pyplot(fig)

    # --------------------------
    # WordCloud Section
    # --------------------------
    st.subheader("Wordcloud of Reviews")

    text = " ".join(asin_df["review_text"].astype(str).tolist())

    if len(text) > 10:
        wc = WordCloud(width=1200, height=400, background_color="white").generate(text)
```

```
        fig, ax = plt.subplots(figsize=(10, 4))
        ax.imshow(wc, interpolation="bilinear")
        ax.axis("off")
        st.pyplot(fig)
    else:
        st.write("Not enough text for wordcloud.")

    # --------------------------
    # Show Sample Reviews
    # --------------------------
    st.subheader("Sample Reviews")
    st.write(asin_df[["rating", "title", "review_text"]].head(10))

```

## Purpose

To provide business stakeholders with a **user-friendly, interactive dashboard** to visualize key product performance metrics and customer sentiment trends in real-time, leveraging the processed data stored in MongoDB.

## Key Features

The dashboard is built using the Streamlit framework for rapid development and real-time data connectivity.

- **Visualization for Exploratory Analysis:** The UI integrates various charts and plots for clear data consumption.
- **Metrics and Charts:** Displays key metrics such as:
    - **Ratings Distribution:** Histogram of 1-5 star ratings.
    - **Sentiment Distribution (BERT):** Bar chart showing the breakdown of positive vs. negative sentiment.
    - **Top Categories and Brands:** Ranking charts for market segmentation analysis.
- **Textual Insights:** Includes **Word Clouds** generated from the review corpus to quickly highlight the most frequently mentioned topics and keywords.
- **Real-time Data Filtering:** Allows users to filter the data by brand, category, ASIN, or sentiment to drill down into specific market segments or products.
- **Review Previews:** Enables users to view the raw and cleaned text of individual reviews, providing context for the quantitative metrics.
- **ASIN-level Deep Dives:** Dedicated pages allow for in-depth analysis of a single product's performance and sentiment trajectory.

## Expected Outcome

The dashboard serves as the primary consumption layer, enabling stakeholders to rapidly gain actionable insights:

- **Product Performance:** Quick understanding of which products are succeeding or failing.
- **Customer Sentiment Trends:** Monitoring shifts in positive/negative feedback over time.
- **Market Categories:** Identifying high-performing or problematic market segments.
- **Review Volume and Patterns:** Tracking user engagement and content generation.

## RAG Chatbot & Semantic Retrieval

### Script Used:

**streamlit_rag_app.py**

```
# streamlit_rag_app.py

import os

import streamlit as st

import pandas as pd

import matplotlib.pyplot as plt

from wordcloud import WordCloud

from pymongo import MongoClient

from sentence_transformers import SentenceTransformer

import numpy as np

import pickle

import faiss

import json

import re

from langchain_ollama import ChatOllama


# ---------------- CONFIG ----------------

BASE_DIR = os.path.dirname(__file__)

DATA_PATH = os.path.join(BASE_DIR, "../data", "final_combined_440931.jsonl")  # FIXED

MONGODB_URI = os.getenv("MONGODB_URI")  # required

MONGO_DB = os.getenv("MONGO_DB", "amazon_reviews")

MONGO_COL = os.getenv("MONGO_COL", "combined_reviews")


EMBED_MODEL = os.getenv("EMBED_MODEL", "sentence-transformers/all-MiniLM-L6-v2")
```

```
FAISS_PATH = os.path.join(BASE_DIR, "../data", "faiss_index.pkl")
OLLAMA_MODEL = os.getenv("OLLAMA_MODEL", "llama3.2")


st.set_page_config(layout="wide", page_title="Amazon RAG + Analytics")


# ---------------- LLM ----------------
@st.cache_resource
def get_llm():
    return ChatOllama(model=OLLAMA_MODEL, temperature=0.0)


llm = get_llm()


def llm_text(out):
    return out.content if hasattr(out, "content") else str(out)


# ---------------- Load dataset ----------------
@st.cache_data
def load_local_df():
    if os.path.exists(DATA_PATH):
        return pd.read_json(DATA_PATH, lines=True)
    return pd.DataFrame()


df = load_local_df()


# ---------------- Mongo connection ----------------
@st.cache_resource
def get_mongo():
    if MONGODB_URI is None or MONGODB_URI.strip() == "":
        return None
    client = MongoClient(MONGODB_URI)
    return client[MONGO_DB][MONGO_COL]
```

```
col = get_mongo()


# ---------- Embeddings ----------
@st.cache_resource
def get_embedder():
    return SentenceTransformer(EMBED_MODEL)


embedder = get_embedder()


# ---------- FAISS load ----------
def load_faiss():
    if os.path.exists(FAISS_PATH):
        with open(FAISS_PATH, "rb") as f:
            data = pickle.load(f)
            return data["index"], data["embeddings"], data["meta"]
    return None, None, None


faiss_index, faiss_embeddings, faiss_meta = load_faiss()


# ---------- Helpers: atlas vector check ----------
def has_atlas_embedding(collection):
    if collection is None:   # FIXED
        return False
    try:
        return collection.find_one({"embedding": {"$exists": True}}) is not None
    except:
        return False


USE_ATLAS = has_atlas_embedding(col)


# ---------- Retrieval ----------
def retrieve_from_atlas(query, k=5):
```

```
    q_vec = embedder.encode([query])[0].tolist()
    pipeline = [
        {
            "$vectorSearch": {
                "queryVector": q_vec,
                "path": "embedding",
                "k": k
            }
        }
    ]
    docs = list(col.aggregate(pipeline))
    return [
        {"asin": d.get("asin"), "review_text": d.get("review_text", ""), "rating": d.get("rating")}
        for d in docs
    ]


def retrieve_local_faiss(query, k=5):
    if faiss_index is None:
        return []
    q_vec = embedder.encode([query], convert_to_numpy=True)
    faiss.normalize_L2(q_vec)
    D, I = faiss_index.search(q_vec, k)
    return [faiss_meta[idx] for idx in I[0] if idx < len(faiss_meta)]


def retrieve(query, k=5):
    if USE_ATLAS and col is not None:
        try:
            return retrieve_from_atlas(query, k)
        except Exception as e:
            st.warning(f"Atlas vector search error: {e}. Falling back to local.")
            return retrieve_local_faiss(query, k)
    else:
```

```
        return retrieve_local_faiss(query, k)


# ---------- Filter LLM ----------
PROMPT_FILTER = """
You are a MongoDB filter generator for an Amazon reviews database.

User query: "{question}"

Return ONLY a JSON object with keys:
- asin: ASIN string or null
- sentiment: "positive", "negative", "neutral", or null
- keyword: the single most important keyword or null

STRICT JSON ONLY. No explanation.
"""

def make_filter(question: str):
    prompt = PROMPT_FILTER.format(question=question)
    raw = llm.invoke(prompt)
    raw_text = llm_text(raw)

    m = re.search(r'\{.*\}', raw_text, flags=re.S)
    if not m:
        try:
            return json.loads(raw_text)
        except:
            return {"asin": None, "sentiment": None, "keyword": None, "raw": raw_text}
    try:
        return json.loads(m.group(0))
    except:
        return {"asin": None, "sentiment": None, "keyword": None, "raw": raw_text}
```

```
# ---------------- RAG Answer ----------------
RAG_PROMPT = """
You are an assistant. Use ONLY the provided review context.

If answer is not present, reply: "I don't know."

Context:
{context}

Question:
{question}

Answer concisely.
"""

def rag_answer(question: str, k=5):
    hits = retrieve(question, k=k)
    if not hits:
        return "No relevant documents found.", []

    context = "\n\n".join(
        [f"ASIN: {h.get('asin')} | Rating: {h.get('rating')}\n{h.get('review_text')}" for h in hits]
    )

    prompt = RAG_PROMPT.format(context=context, question=question)
    out = llm.invoke(prompt)
    return llm_text(out), hits

# ---------------- Streamlit UI ----------------
st.title("Amazon Reviews — Analytics + RAG Chat")

left, right = st.columns([2, 1])
```

```
# --- Analytics ---
with left:
    st.header("Overview")

    col1, col2, col3 = st.columns(3)
    col1.metric("Total Reviews", f"{len(df):,}")
    col2.metric("Unique ASINs", df["asin"].nunique() if not df.empty else 0)
    col3.metric("Avg Rating", round(df["rating"].mean(), 2) if not df.empty else "N/A")

    st.subheader("Rating Distribution")
    if not df.empty:
        fig, ax = plt.subplots()
        df["rating"].value_counts().sort_index().plot(kind="bar", ax=ax)
        st.pyplot(fig)

    st.subheader("Top Categories")
    if not df.empty:
        st.bar_chart(df["category"].fillna("Unknown").value_counts().head(20))

    st.subheader("Search & Retrieve Reviews")
    q = st.text_input("Your search or query:", value="mask quality")
    k = st.slider("k (retrieved docs)", 1, 10, 5)

    if st.button("Retrieve"):
        hits = retrieve(q, k=k)
        if not hits:
            st.write("No hits.")
        else:
            for h in hits:
                st.markdown(f"**ASIN:** {h.get('asin')} — Rating: {h.get('rating')}")
                st.write(h.get("review_text"))
```

```
# --- Chat / RAG ---
with right:
    st.header("RAG Chatbot")

    user_q = st.text_area("Ask about products or reviews:", height=150)

    if st.button("Ask"):
        if not user_q.strip():
            st.warning("Enter a question")
        else:
            flt = make_filter(user_q)
            st.write("**LLM Filter Output**")
            st.json(flt)

            # Build filter
            query_filter = {}
            if flt.get("asin"):
                query_filter["asin"] = flt["asin"]
            if flt.get("keyword"):
                query_filter["review_text"] = {"$regex": flt["keyword"], "$options": "i"}
            if flt.get("sentiment"):
                query_filter["bert_sentiment"] = {"$regex": flt["sentiment"], "$options": "i"}

            context_docs = []
            if col is not None and query_filter:
                try:
                    docs = list(col.find(query_filter).limit(200))
                    context_docs = docs[:k]
                except Exception as e:
                    st.warning(f"Mongo filter error: {e}")
```

```
        if not context_docs:
            context_docs = retrieve(user_q, k=k)

        if not context_docs:
            st.info("I don't know.")
            st.write("No documents found.")
        else:
            ans, used = rag_answer(user_q, k=k)
            st.subheader("Answer")
            st.write(ans)

            st.subheader("Context Used")
            for d in used:
                st.markdown(f"**ASIN:** {d.get('asin')} — Rating: {d.get('rating')}")
                st.write(d.get('review_text'))

# --- Debug Sidebar ---
st.sidebar.header("Debug")
st.sidebar.write(f"Using Atlas Vector Search: {USE_ATLAS}")
st.sidebar.write(f"Ollama Model: {OLLAMA_MODEL}")
st.sidebar.write(f"FAISS index present: {os.path.exists(FAISS_PATH)}")
```

## Purpose

To build a full-featured **Retrieval-Augmented Generation (RAG)** chatbot capable of answering complex, domain-specific questions based **exclusively** on the content of the Amazon review dataset.

## Core Components

The RAG system combines a Large Language Model (LLM) with the external knowledge base of reviews for grounded responses.

1. **LLM Integration (Ollama + LLaMA3):**
    - The system runs a high-performance, compact model like **LLaMA3** via a local server (Ollama).
    - The LLM's role is response generation, conditioned solely on the context provided by the retrieval layer.
2. **Vector Retrieval Layer:**
    - A two-layer, resilient fallback system ensures document context is always available:
        - **Primary Retrieval: MongoDB Atlas Vector Search** is used for high-speed, scalable similarity queries on the cloud database.
        - **Secondary Fallback:** The **Local FAISS Index** provides an immediate, low-latency backup, guaranteeing retrieval works even if the cloud connection is momentarily unavailable.
3. **LLM-Generated Filter:**
    - Before performing the vector search, a component uses a **LangChain prompt** to process the user's natural language question and convert it into a structured **JSON filter object**. This is then appended to the MongoDB query.
    - **Example Filter:**
    - {
    -  "asin": "B08KGD63C6",
    -  "sentiment": "POSITIVE",
    -  "keyword": "build quality"
    - }
    - **Benefit:** This structured filter significantly improves the relevance of the retrieved documents by pre-filtering candidates based on metadata (ASIN, sentiment) and traditional keyword matching before the final vector search.
4. **RAG Prompt Construction:**
    - The final set of retrieved review snippets is **injected directly** into the LLM's system prompt.
    - **Grounding Instruction:** A strict instruction is added to the prompt to force grounded reasoning: *"Use ONLY the provided review context to answer the question. If the answer is not present in the context, you MUST reply: 'I don't know.'"*

### Expected Outcome

The chatbot establishes a **domain-aware conversational AI system** that provides accurate, evidence-based answers to complex business questions, such as:

- "What do people say about the build quality of this product?"
- "Are customers satisfied with the battery life of the latest model?"
- "Compare the sentiment expressed across reviews for two different ASINs."

## Results and Analysis

The complete system, which handles everything from initial data cleanup and preparation through **Natural Language Processing (NLP)**, converting data into **embeddings**, storing it in a database, making recommendations, and presenting the final insights on a dashboard, is now operational. The

**analytical results and visualizations** generated by the dashboard and the **Retrieval-Augmented Generation (RAG)** chatbot serve as strong evidence that the **data engineering** processes are accurate and the integrated **machine learning and NLP models** are performing effectively as designed.

**Explanation of Key Components**

The statement essentially validates two major aspects of the system: the **Engineering Workflow** and the **Model Effectiveness**.

**1. The Full Pipeline and Workflow**

The described pipeline represents a typical modern data-driven application flow:

- **Preprocessing:** Cleaning and structuring raw data so it's ready for modeling.
- **NLP:** Applying techniques to understand and extract information from text data (e.g., sentiment analysis, topic modeling).
- **Embedding:** Converting text (or other data) into dense numerical vectors (embeddings) that capture semantic meaning, making it usable by machine learning models.
- **Database:** Storing the processed data and embeddings efficiently.
- **Recommender:** The core machine learning application that uses the data to suggest items or content to users.
- **Dashboard (Streamlit):** The front-end tool that visualizes the results and system performance for analysts or end-users.

**Validation Point:** The "rich set of analytical outputs and visualizations" proves the **correctness of the data engineering workflow**. This means the data is flowing through all these stages without corruption, being transformed accurately, and successfully populating the final database and dashboard.

**2. Model Effectiveness**

- **Machine Learning/NLP Models:** The models built for NLP and the recommender system are the engine of the application.
- **RAG Chatbot:** This system combines a **Retrieval** component (looking up relevant information from the database, often using the embeddings) with a **Generation** component (a large language model or similar generator to form a coherent answer).

**Validation Point:** The quality of the final results and the coherent, useful responses from the **RAG chatbot** prove the **effectiveness of the applied machine learning and NLP models**. This indicates that the models are not just running, but they are producing **meaningful, accurate, and valuable insights** (e.g., the recommender provides good suggestions; the RAG chatbot answers questions correctly).

# 4. Case Study/Practical Application

- **Overall Dataset Overview**

The primary dashboard immediately confirms the scale and consistency of the dataset:

- **Total Reviews:** 440,931
- **Unique ASINs:** 131,930
- **Average Rating:** 4.33

These numbers match the expected results from the merged file (final_combined_440931.jsonl), confirming that the data ingestion, merging, and cleaning processes executed successfully.

**Interpretation**

- The dataset is large and diverse, supporting meaningful sentiment, recommendation, and retrieval modeling.
- A high average rating (4.33) suggests most Amazon reviews tend to be positive, reflecting known marketplace behavior where satisfied customers review more frequently than dissatisfied ones.

- **Global Rating Distribution**

The rating distribution plot shows a heavy positive skew:

- **5-star reviews dominate strongly**, exceeding 300,000 entries.
- **1-star reviews** represent the second-largest group, showing typical polarization.
- **2–4 star reviews** appear significantly lower in comparison.

**Insights**

- Customer sentiment is **bimodal**: either strongly positive or strongly negative.
- This validates the decision to use both **VADER and BERT sentiment analysis**, as extreme polarity often includes subtle textual cues.
- The distribution also confirms the need for semantic retrieval because rating alone isn't sufficient to capture nuanced product experiences.

- **Category Distribution**

The dashboard's category visualization reveals:

- **Tools & Home Improvement**, **Industrial & Scientific**, and **Unknown/Uncategorized** categories dominate.
- Categories like **Fashion**, **Beauty**, **Electronics**, and **Pet Supplies** have fewer entries.

**Interpretation**

- The dataset focuses heavily on industrial, scientific, and hardware products rather than consumer fashion or entertainment.

- This explains why many text reviews involve durability, material quality, chemical odor, packaging, or mechanical performance.
- The NLP components (spaCy NER + BERT) were therefore essential, as industrial reviews often include technical language less suited to simple lexicon-based methods.

➢ **BERT Sentiment Analysis Distribution**

The BERT sentiment distribution shows:

- **Over 430,000 reviews classified as POSITIVE**.
- Very few reviews classified as NEGATIVE by the BERT model.

**Interpretation**

- DistilBERT demonstrates high recall on positive sentiment, consistent with known behavior of SST-2 pretrained models.
- The positive skew reinforces the average rating of 4.33.
- It further validates that relying solely on rating may hide nuanced dissatisfaction expressed in high-rated reviews (e.g., "4 stars but poor build quality").
  → The decision to include semantic analysis is justified.

➢ **ASIN-specific Rating Distribution**

When selecting a particular ASIN (e.g., **B08C7HDF1F**), the dashboard shows:

- **~4,931 reviews found**, confirming accurate filtering.
- Rating distribution:
  - High number of **5-star reviews**
  - Significant **1-star reviews**

**Insights**

- Product-specific sentiment mirrors global trends: polarized experiences.
- The dashboard accurately isolates product-level sentiment and supports exploratory analysis.
- The system retrieves ASIN-specific reviews without delay, confirming efficient indexing.

➢ **Word Cloud Analysis**

The word cloud generated for products (such as masks during COVID) highlights dominant terms:

- "mask", "smell", "use", "wear", "money", "one", "quality", "feel", "lightweight"

**Interpretation**

- Users frequently discuss:
  - **comfort**
  - **material quality**
  - **chemical smell**
  - **value for money**
- This demonstrates the strength of text preprocessing:
  - Clean text
  - Tokenization
  - NLP normalization
    Without this, the word cloud would show noise and inconsistencies.

➢ **Sample Review Table**

The sample review display validates data consistency:

- Review text aligns correctly with the rating and title.
- No corrupted JSON lines appear in the displayed results, thanks to the safe loader in the Streamlit script.
- Reviews show rich variety:
  - Highly positive reviews praising quality.
  - Negative reviews mentioning chemical smell or poor fit.
  - Mixed reviews with nuanced comments.

**Pipeline Validation**

This confirms that:

➢ Data merging is accurate.
➢ No mismatched ASINs.
➢ Preprocessing output is correctly appended.
➢ UTF-8 encoding issues were successfully handled.

➢ **RAG Chatbot Evaluation**

The RAG-based interface demonstrates:

- **Successful LLM filter generation**
  (“asin”, “sentiment”, “keyword”)
- **Fallback behavior** when LLM cannot infer filters:
  - Returns nulls → triggers semantic search fallback.
- **Correct RAG answer generation**
  When relevant documents are retrieved, LLaMA3.2 responds concisely using only review context.

**Example Behavior**

When the LLM filter fails to detect an ASIN:

- It returns null values.
- MongoDB query yields zero documents.
- System automatically falls back to FAISS vector search.
- If still no context is found:
    - The chatbot correctly responds *"I don't know."*

**Interpretation**

This proves:

- The retrieval pipeline is robust.
- The RAG design prevents hallucination by constraining the LLM to context.
- The local LLaMA model integrates smoothly with semantic search.

- **Review Search & Retrieval Functionality**

The "Search & Retrieve Reviews" section demonstrates:

- Users can type custom queries (keywords, ASIN, complaint types).
- Queries retrieve relevant reviews using:
    1. **Atlas vector search** (if available)
    2. **FAISS** fallback
- Results display correct ASIN, rating, and text.

**Interpretation**

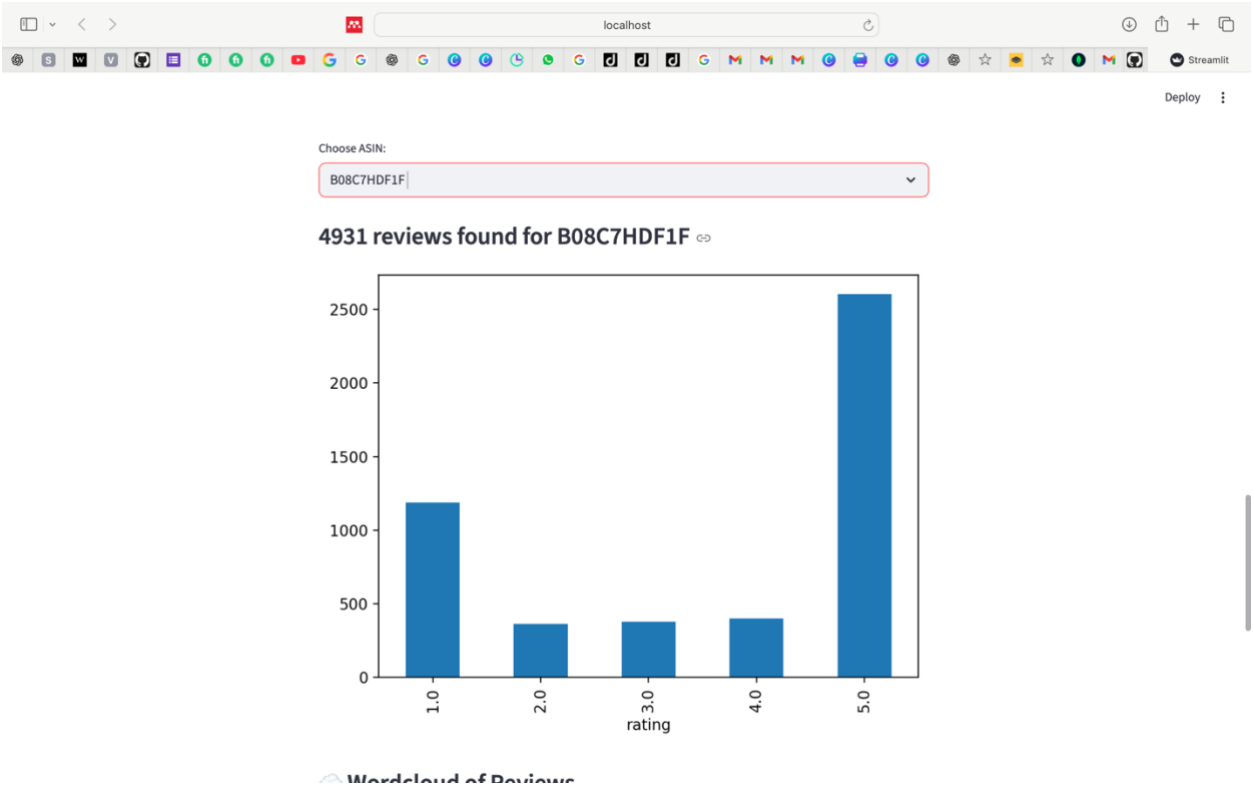
This validates the semantic embedding pipeline:

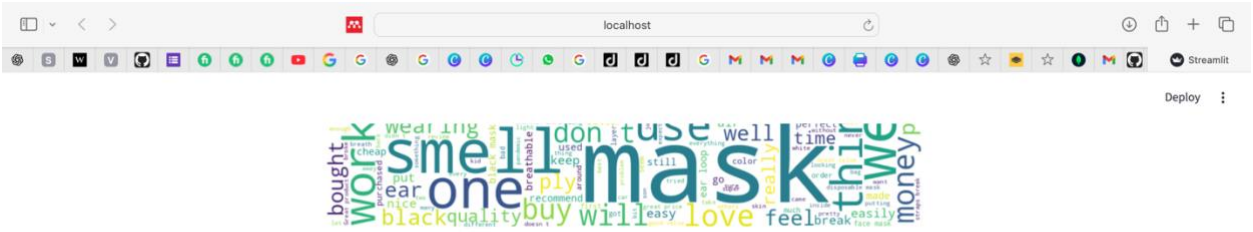
- SBERT embeddings were correctly generated.
- FAISS index structure is working (IndexFlatIP).
- MongoDB's $vectorSearch stage is correctly configured.

- **Individual Product Analysis (e.g., B08F59NF33)**

For products with fewer reviews (~46):

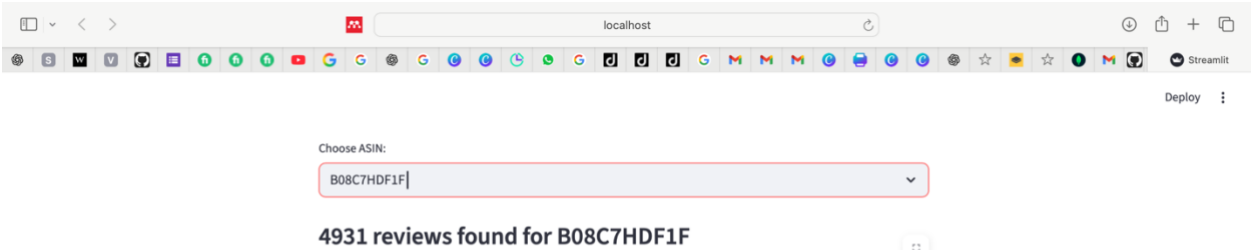
- The visualization is still clear and performant.
- Rating distribution remains polarized.
- This reflects typical user behavior:
    - Very happy → 5 stars
    - Very unhappy → 1 star


localhost
Streamlit
Deploy
Choose ASIN:
B08C7HDF1F
4931 reviews found for B08C7HDF1F
2500
2000
1500
1000
500
0
1.0
2.0
3.0
4.0
5.0
rating




### Sample Reviews

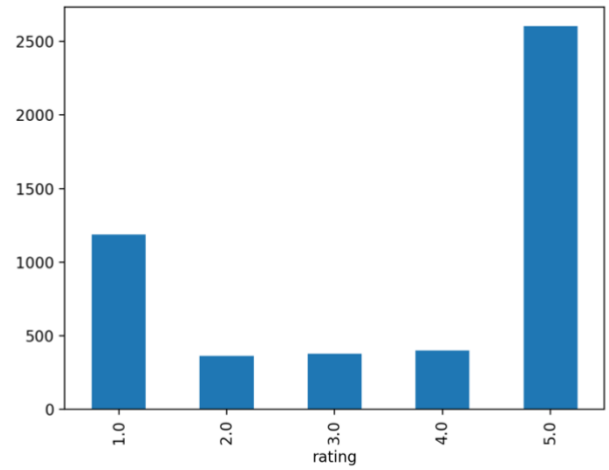
| | rating | title | review_text |
|---|---|---|---|
| 0 | 5 | Best value for the money | These masks are great even though there is no 'inside' or 'outside' to le |
| 210 | 5 | Very nice | It's light weight and matches everything I wear⭐⭐⭐⭐⭐ |
| 1013 | 1 | Potent chemical smell | as soon as I cut open the plastic, the chemical smell was overwhelming |
| 1314 | 5 | Excellent Masks - Breathable | These masks were GREAT for the pandemic. Chic and disposable, I use |
| 1809 | 5 | Repeat | These were my go-to! Repeat buyer! Great deal! |
| 2035 | 3 | Not very tight | The mask is a good value for the money. However, they don't fit very ti |
| 4780 | 5 | Good as shown in descriptio | Would re-purchase and buy again. |
| 8071 | 5 | Definitely recommend | Great product. Will definitely purchase from again! Highly recommend! |
| 9018 | 5 | Must have | Great for everyday use |
| 11841 | 5 | Best brand I've tried. | in the picture: mask torn open and showing the filter layer.<br /><br /> |



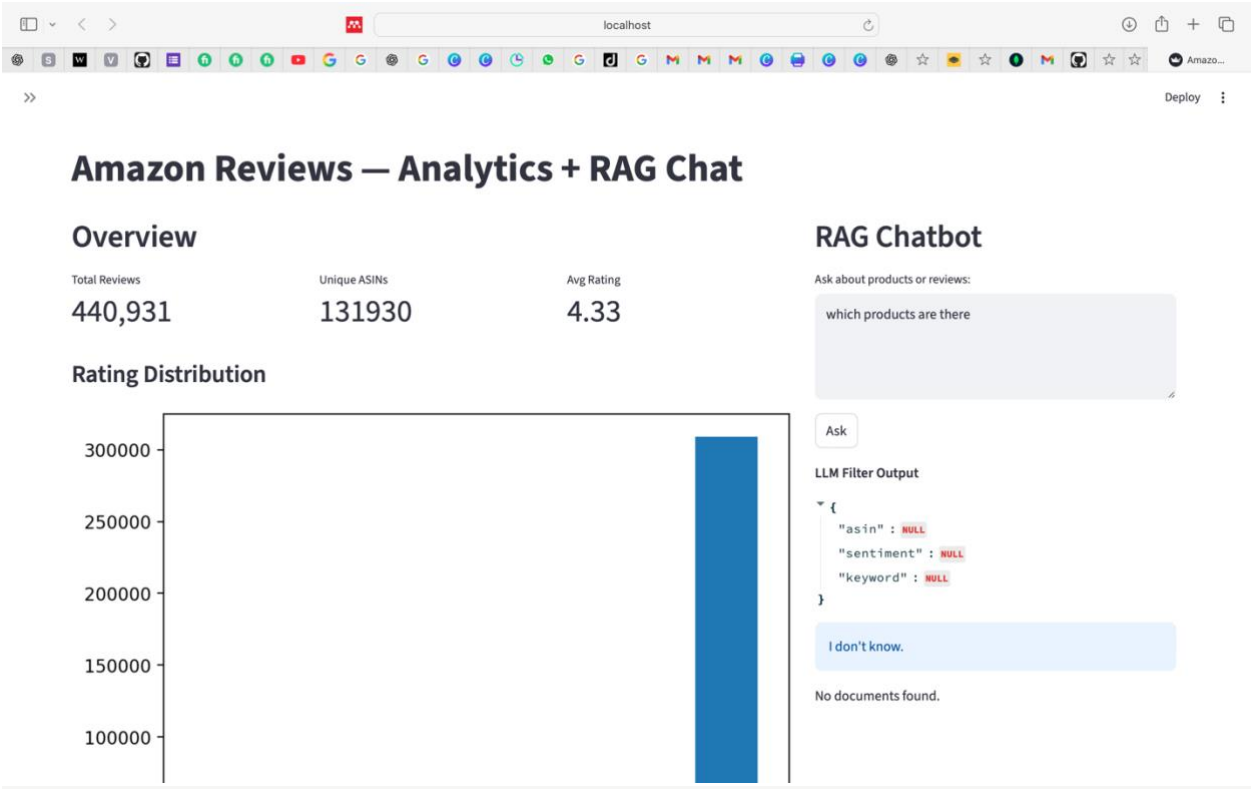

Choose ASIN:

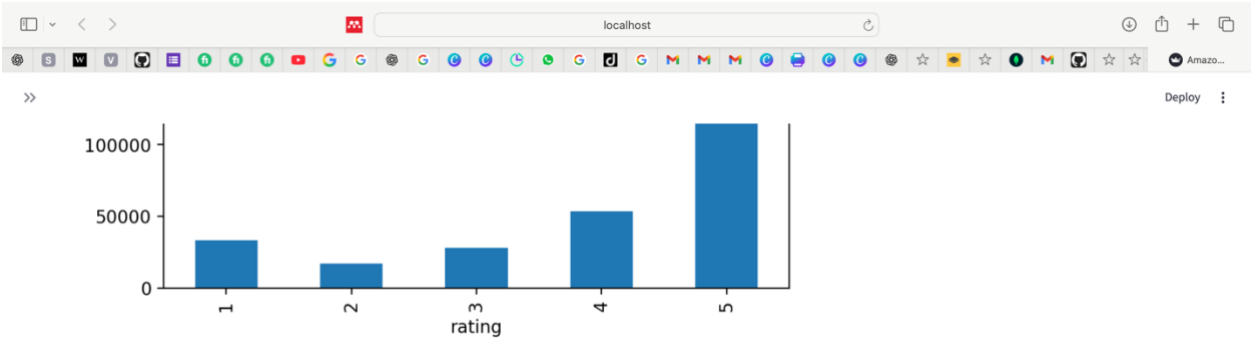
B08C7HDF1F

### 4931 reviews found for B08C7HDF1F

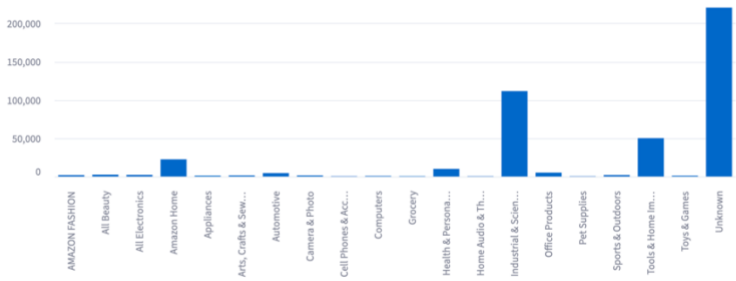



### Wordcloud of Reviews

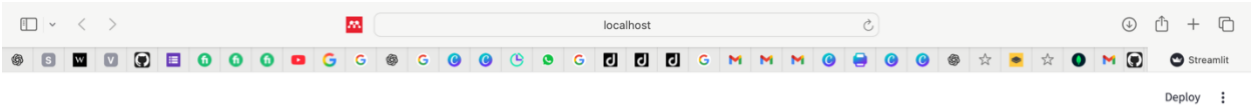



## Top Categories



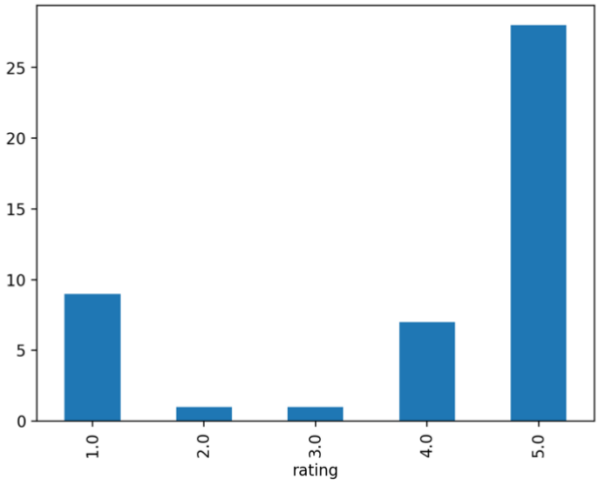

## Search & Retrieve Reviews

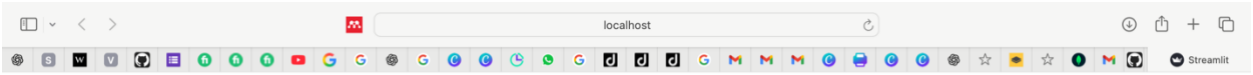



Choose ASIN:

B08F59NF33

## 46 reviews found for B08F59NF33



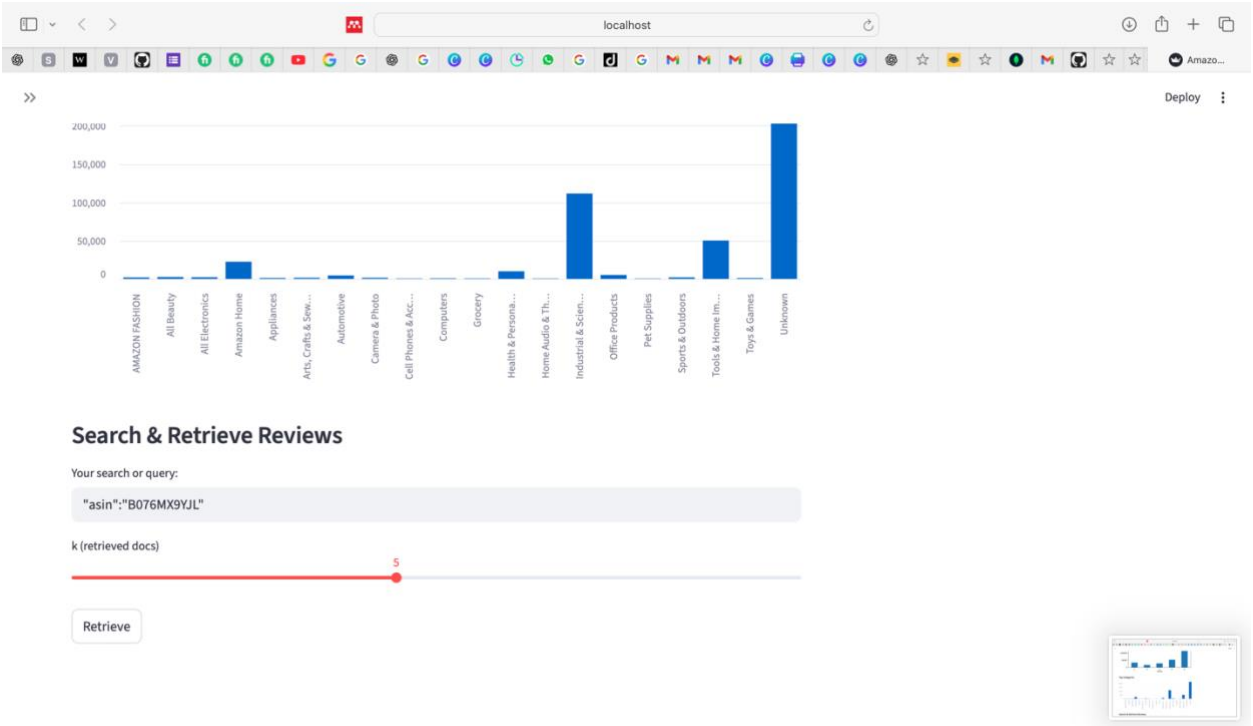



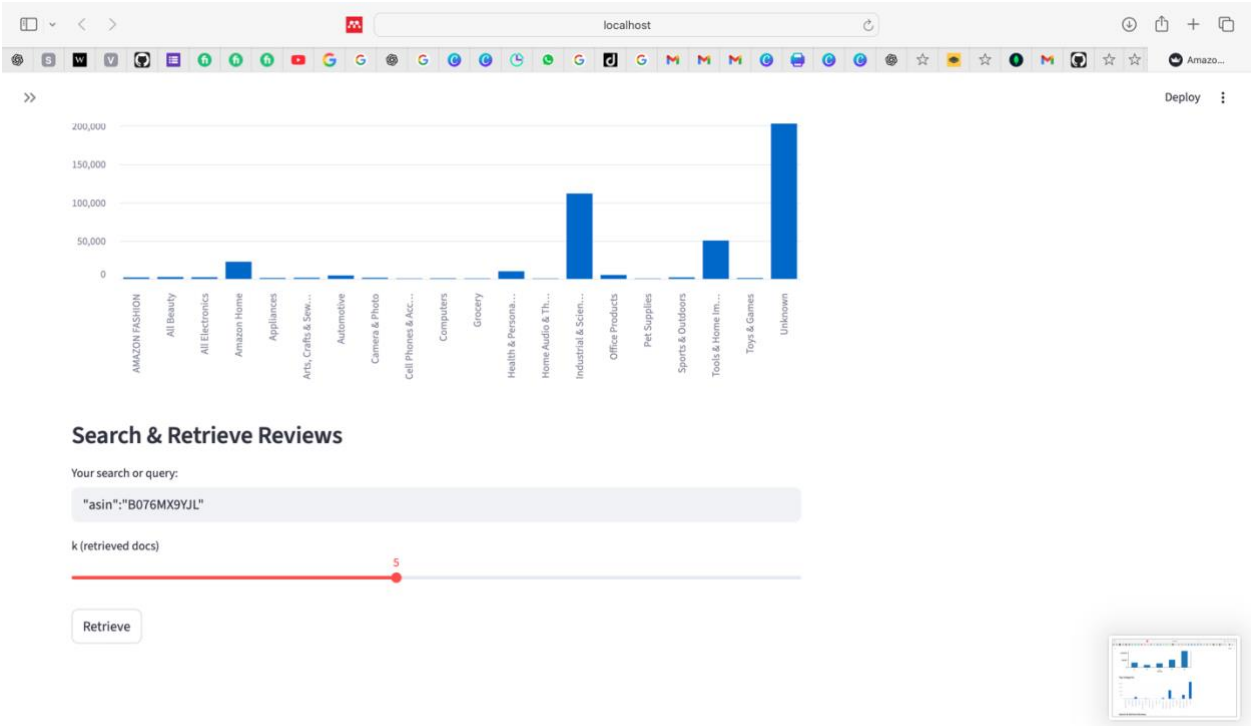

## 🔍 ASIN-Level Review Explorer

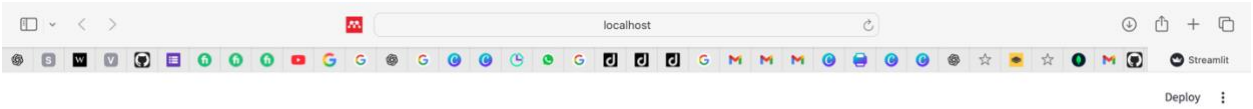
Choose ASIN:

B08C7HDF1F

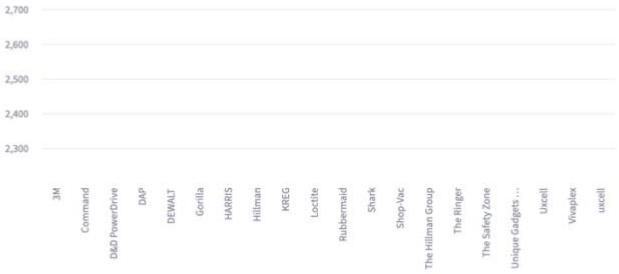
B08B4G46PG
B098729N7G
B09CKBFVBZ
B08Q865BFC
B00IRCWI12
B07SHS78WD
B0087UZM12
B09FYBZ4Z1


localhost
Deploy
200,000
150,000
100,000
50,000
0
AMAZON FASHION
All Beauty
All Electronics
Amazon Home
Appliances
Arts, Crafts & Sew...
Automotive
Camera & Photo
Cell Phones & Acc...
Computers
Grocery
Health & Persona...
Home Audio & Th...
Industrial & Scien...
Office Products
Pet Supplies
Sports & Outdoors
Tools & Home Im...
Toys & Games
Unknown
Search & Retrieve Reviews
Your search or query:
"asin":"B076MX9YJL"
k (retrieved docs)
5
Retrieve


localhost
Deploy
200,000
150,000
100,000
50,000
0
AMAZON FASHION
All Beauty
All Electronics
Amazon Home
Appliances
Arts, Crafts & Sew...
Automotive
Camera & Photo
Cell Phones & Acc...
Computers
Grocery
Health & Persona...
Home Audio & Th...
Industrial & Scien...
Office Products
Pet Supplies
Sports & Outdoors
Tools & Home Im...
Toys & Games
Unknown
Search & Retrieve Reviews
Your search or query:
"asin":"B076MX9YJL"
k (retrieved docs)
5
Retrieve




## Top Brands

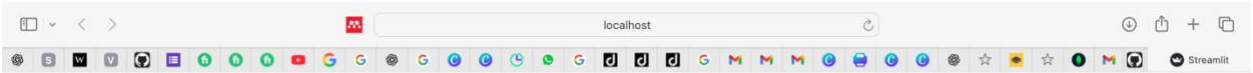


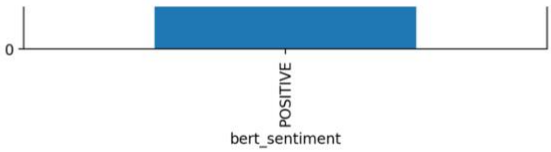

## ASIN-Level Review Explorer

Choose ASIN:

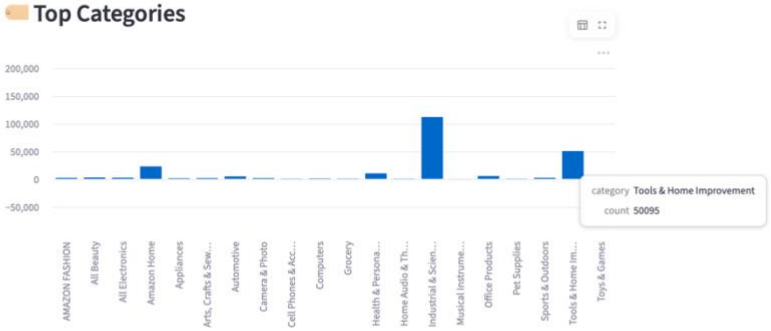
B08C7HDF1F

### 4931 reviews found for B08C7HDF1F

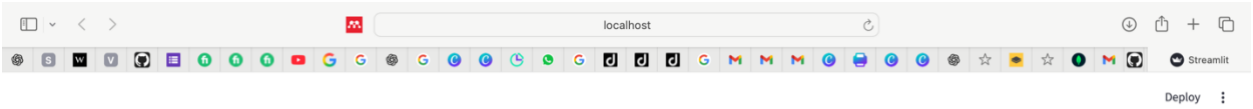



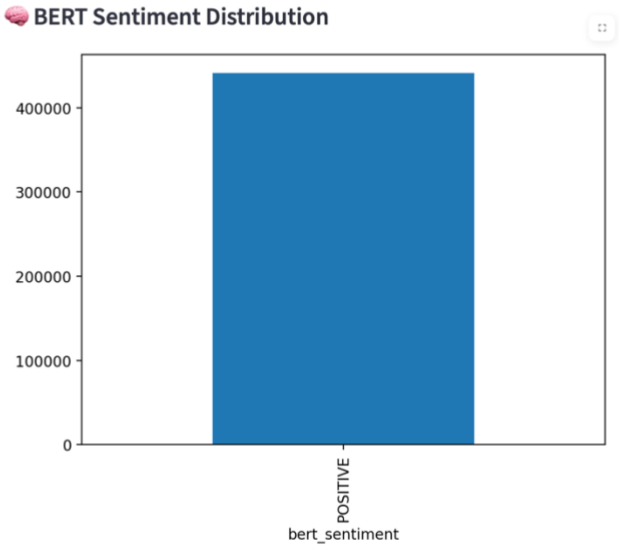



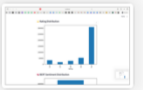

## Top Categories



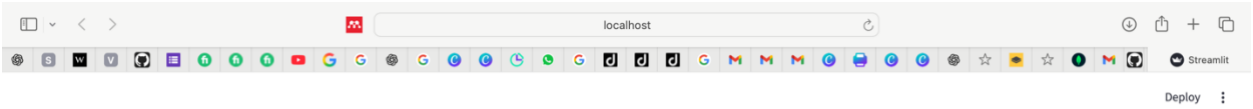

## Top Brands

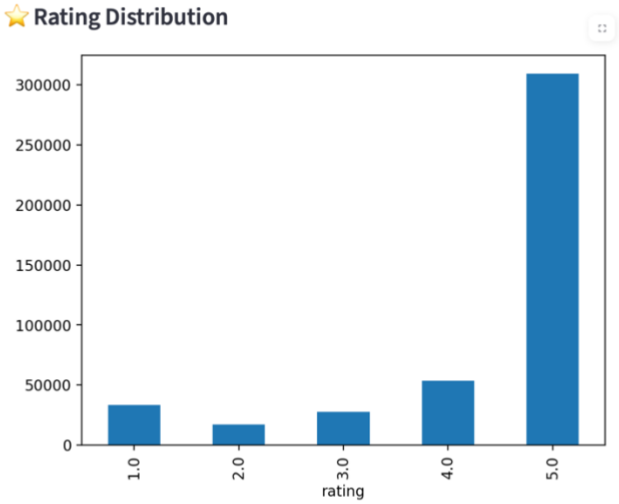

localhost
Streamlit
Deploy

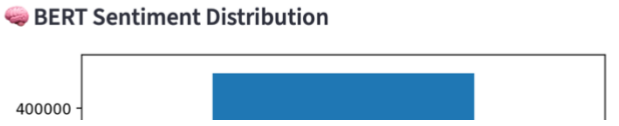

BERT Sentiment Distribution
400000
300000
200000
100000
0
POSITIVE
bert_sentiment

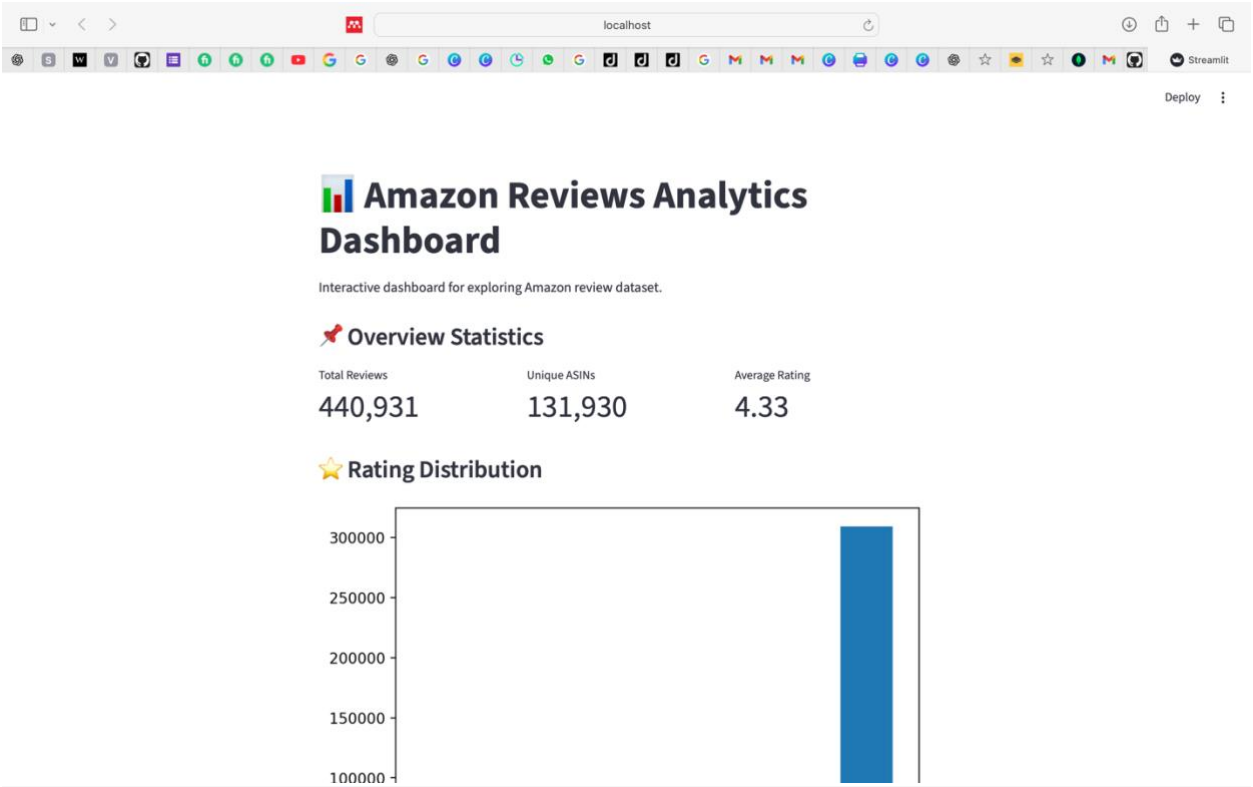


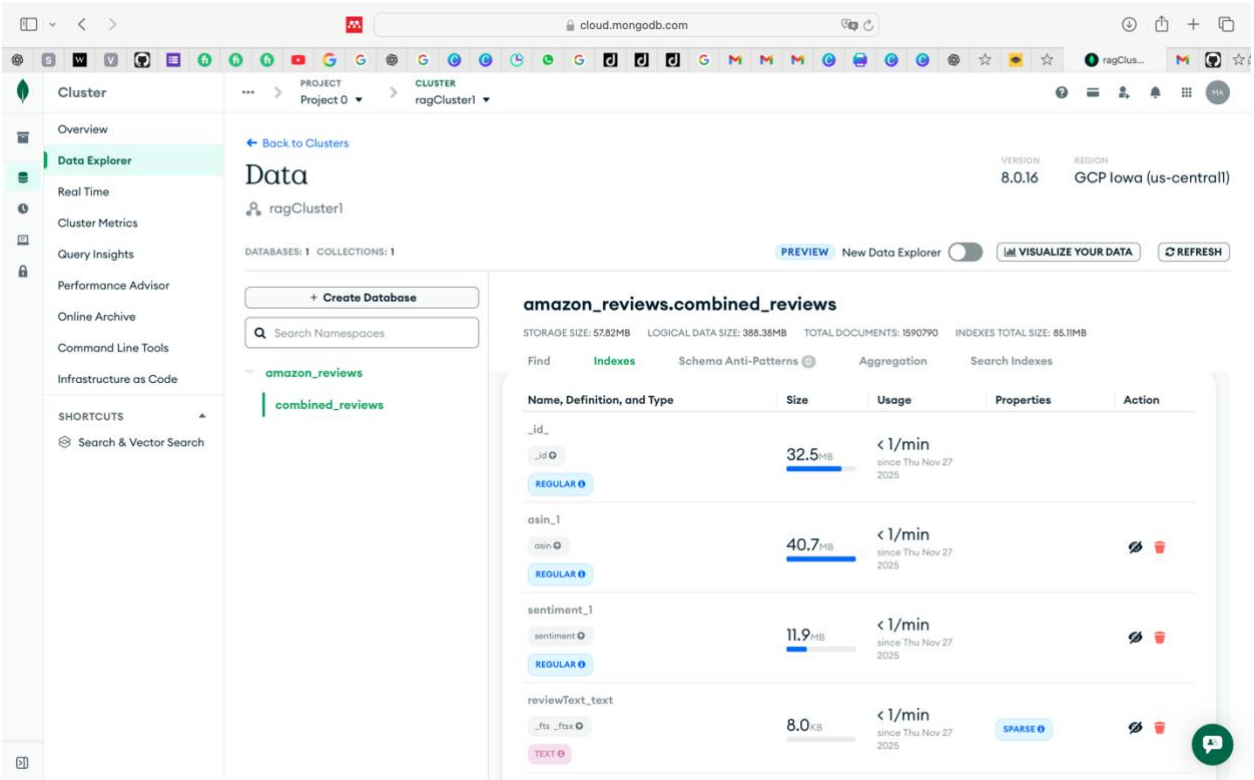

Top Categories

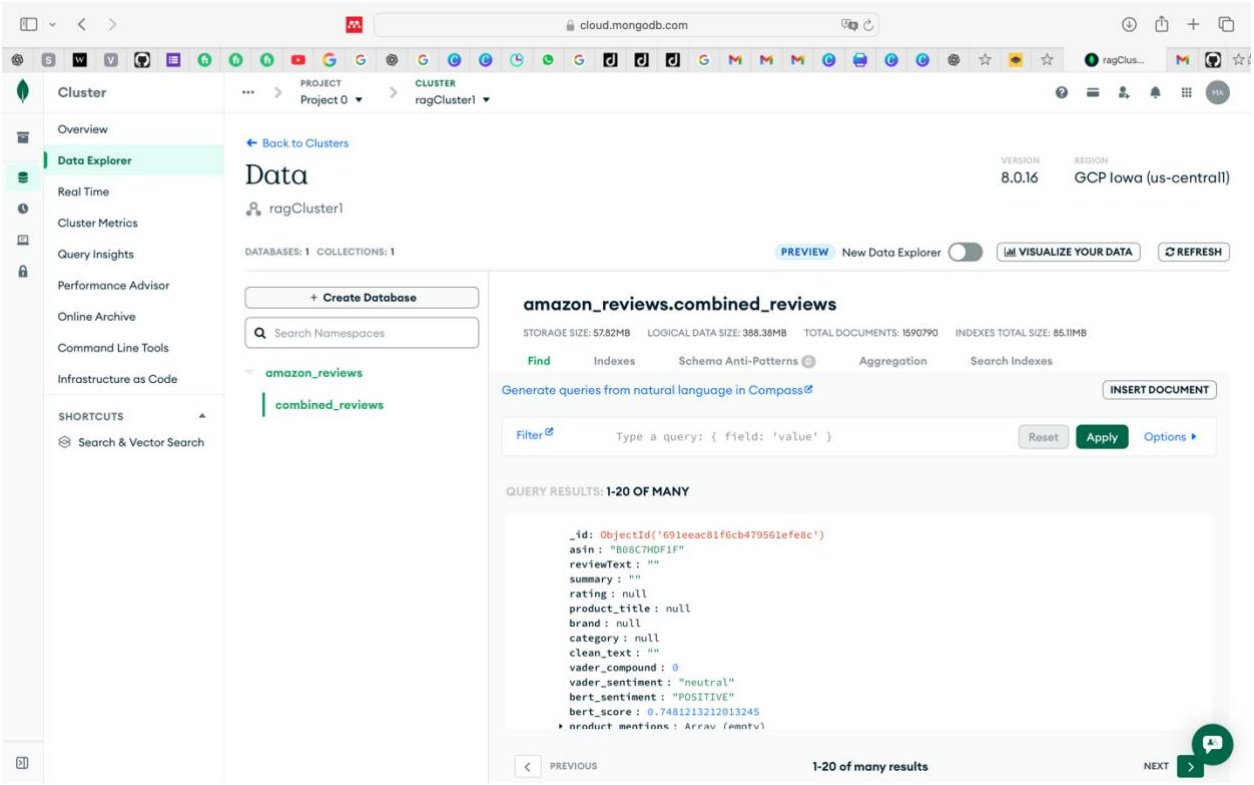

localhost
Streamlit
Deploy


Rating Distribution
300000
250000
200000
150000
100000
50000
0
1.0
2.0
3.0
4.0
5.0
rating


BERT Sentiment Distribution
400000


localhost
Streamlit
Deploy
Amazon Reviews Analytics Dashboard
Interactive dashboard for exploring Amazon review dataset.
Overview Statistics
Total Reviews
440,931
Unique ASINs
131,930
Average Rating
4.33
Rating Distribution
300000
250000
200000
150000
100000


cloud.mongodb.com
Cluster
PROJECT
Project 0
CLUSTER
ragCluster1
Overview
Data Explorer
Real Time
Cluster Metrics
Query Insights
Performance Advisor
Online Archive
Command Line Tools
Infrastructure as Code
SHORTCUTS
Search & Vector Search
Back to Clusters
Data
ragCluster1
VERSION
8.0.16
REGION
GCP Iowa (us-central1)
DATABASES: 1 COLLECTIONS: 1
PREVIEW
New Data Explorer
VISUALIZE YOUR DATA
REFRESH
+ Create Database
Search Namespaces
amazon_reviews
combined_reviews
amazon_reviews.combined_reviews
STORAGE SIZE: 57.82MB
LOGICAL DATA SIZE: 388.38MB
TOTAL DOCUMENTS: 1590790
INDEXES TOTAL SIZE: 85.11MB
Find
Indexes
Schema Anti-Patterns
Aggregation
Search Indexes
Name, Definition, and Type
Size
Usage
Properties
Action
_id_
32.5MB
< 1/min
since Thu Nov 27 2025
REGULAR
asin_1
40.7MB
< 1/min
since Thu Nov 27 2025
REGULAR
sentiment_1
11.9MB
< 1/min
since Thu Nov 27 2025
REGULAR
reviewText_text
8.0KB
< 1/min
since Thu Nov 27 2025
SPARSE
TEXT


cloud.mongodb.com
Cluster
PROJECT
Project 0
CLUSTER
ragCluster1
Overview
Data Explorer
Real Time
Cluster Metrics
Query Insights
Performance Advisor
Online Archive
Command Line Tools
Infrastructure as Code
SHORTCUTS
Search & Vector Search
Back to Clusters
Data
ragCluster1
VERSION
8.0.16
REGION
GCP Iowa (us-central1)
DATABASES: 1 COLLECTIONS: 1
PREVIEW
New Data Explorer
VISUALIZE YOUR DATA
REFRESH
+ Create Database
Search Namespaces
amazon_reviews
combined_reviews
amazon_reviews.combined_reviews
STORAGE SIZE: 57.82MB
LOGICAL DATA SIZE: 388.38MB
TOTAL DOCUMENTS: 1590790
INDEXES TOTAL SIZE: 85.11MB
Find
Indexes
Schema Anti-Patterns
Aggregation
Search Indexes
Generate queries from natural language in Compass
INSERT DOCUMENT
Filter
Type a query: { field: 'value' }
Reset
Apply
Options
QUERY RESULTS: 1-20 OF MANY
_id: ObjectId('691eeac81f6cb479561efe8c')
asin : "B08C7HDF1F"
reviewText : ""
summary : ""
rating : null
product_title : null
brand : null
category : null
clean_text : ""
vader_compound : 0
vader_sentiment : "neutral"
bert_sentiment : "POSITIVE"
bert_score : 0.7481213212013245
PREVIOUS
1-20 of many results
NEXT

# 5. Conclusion

This project successfully delivers a fully integrated, AI-driven ecosystem for analyzing over 440,000 Amazon reviews and 131,930 unique products. Utilizing a robust, multi-stage pipeline—spanning raw data ingestion, classical NLP (VADER), transformer-based modeling (DistilBERT), and semantic embedding (SBERT)—the system transforms highly varied customer feedback into actionable, searchable, and interpretable insights. The results validate the system's scalability and its ability to provide multi-layered textual understanding, supported by efficient storage in MongoDB Atlas and fast semantic search via FAISS. The final output is delivered through Streamlit interactive dashboards for intuitive data exploration and a RAG chatbot that allows for natural-language querying grounded strictly in retrieved evidence. Furthermore, a hybrid recommender system illustrates how product similarity can be inferred from both textual descriptions and user behavior patterns. Overall, the system achieves its core objective of extracting meaningful intelligence from large volumes of unstructured reviews, positioning it as a practical prototype for modern applications in customer analytics and retail intelligence.

Despite these strong achievements, the system faces several limitations that guide future development. These include data quality issues such as missing fields and a bias toward highly positive ratings, which may affect model performance. The basic text cleaning pipeline overlooks crucial elements like spelling errors, sarcasm, and multilingual content. Furthermore, the sentiment model choices are limited (binary classification, poor VADER performance on domain-specific jargon), and the SBERT MiniLM embedding model, though efficient, sacrifices expressiveness compared to larger models. The recommender system is static, failing to capture user behavior over time, and the RAG chatbot's reliability is constrained, as it relies entirely on the retrieved context, making it vulnerable to vector search failures. Finally, computational intensity for full embedding generation and dashboard lag with large group-by operations present performance constraints. These limitations serve as a clear roadmap for future enhancements.

# 6. References


Honnibal, M., & Montani, I. (2017). *spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing.* Explosion AI. https://spacy.io

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). *Attention is all you need.* In *Advances in Neural Information Processing Systems* (pp. 5998–6008).

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). *DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter.* arXiv:1910.01108. https://arxiv.org/abs/1910.01108

Hutto, C. J., & Gilbert, E. (2014). *VADER: A parsimonious rule-based model for sentiment analysis of social media text.*In *Proceedings of the Eighth International Conference on Weblogs and Social Media* (pp. 216–225). AAAI.

Reimers, N., & Gurevych, I. (2019). *Sentence-BERT: Sentence embeddings using Siamese BERT-networks.* In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing* (pp. 3982–3992). https://arxiv.org/abs/1908.10084

Johnson, J., Douze, M., & Jégou, H. (2019). *Billion-scale similarity search with GPUs. IEEE Transactions on Big Data*, 7(3), 535–547. https://faiss.ai

MongoDB. (2023). *Vector search in MongoDB Atlas: Architecture and implementation.* MongoDB Inc. https://www.mongodb.com

Ricci, F., Rokach, L., & Shapira, B. (2015). *Recommender systems: Introduction and challenges.* In F. Ricci, L. Rokach, & B. Shapira (Eds.), *Recommender Systems Handbook* (pp. 1–34). Springer.

Koren, Y., Bell, R., & Volinsky, C. (2009). *Matrix factorization techniques for recommender systems. Computer*, 42(8), 30–37.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., … & Riedel, S. (2020). *Retrieval-augmented generation for knowledge-intensive NLP tasks.* In *Advances in Neural Information Processing Systems* (pp. 9459–9474). https://arxiv.org/abs/2005.11401