Amazon Recommender

# INFO 376

Jah, George, Aidan, Asad, Allen

# Contents

# Meet the team

### Jah Chen

- Cleaning code for readability
- Building embeddings from features
- Preprocessing data into model-friendly matrices

### George Lee

- Cleaning Data
- Creation of Recommendation list based on KNN

### Aidan Bartlett

- Initial Data ETL
- Web App
- Cold-start user profiling

### Asad Jaffery

- Determine features used for rec system
- Clean and merge data to load for system

### Allen Zheng

- Conducted numerous user tests for model evaluation
- Evaluated model efficiency and performance

# Amazon Recommender

**Our goal: Build a recommendation web app to learn about recommender development**

We are using a dataset of Amazon products and reviews, curated by the UCSD McAuley Lab, to develop our recommendation engine

We wanted to experiment with developing a recommendation engine based on real-world data using what we learned during INFO 376

# Project roadmap

**Step 1**

Decide which data we want to focus on, then ingest the data, clean, preprocess, and explore it

**Step 2**

Create the item embeddings

**Step 3**

Create a full-stack web app to obtain an initial user profile and output recommendations

**Step 4**

Combine the recommendation engine with the web app's backend

**Goal**

**Demo a working web app that can make basic recommendations given an initial user profile**

# Initial Dataset

## The more data, the better for us

We chose to use UCSD's data repository, as it gave access to a plethora of data leading to almost 5GB of amazon products and reviews to build our recommendation system from.

# Initial Data Cleaning

### Filling:

Key numerical values such as price were filled based on the factors such as the median price for a category

### Dropping:

High missing cols with little data such as bought_togehter were dropped all together

### Textual Representation Creation:

Key text columns such as title, description, user reviews with write ups were joined into a central column for TF-IDF vectorization.

## Reviews Data

Missing Values Summary:

| | missing_count | missing_% |
|---|---|---|
| title | 804 | 0.02 |
| text | 1033 | 0.02 |
| Unnamed: 0 | 0 | 0.00 |
| rating | 0 | 0.00 |
| images | 0 | 0.00 |
| asin | 0 | 0.00 |
| parent_asin | 0 | 0.00 |
| user_id | 0 | 0.00 |
| timestamp | 0 | 0.00 |
| helpful_vote | 0 | 0.00 |
| verified_purchase | 0 | 0.00 |

Review data was only missing a few titles and text content

## Products Data

Missing Values Summary:

| | missing_count | missing_% |
|---|---|---|
| bought_together | 137269 | 100.00 |
| author | 137007 | 99.81 |
| subtitle | 136919 | 99.75 |
| price | 75261 | 54.83 |
| main_category | 11036 | 8.04 |
| store | 4375 | 3.19 |
| title | 9 | 0.01 |
| parent_asin | 0 | 0.00 |
| details | 0 | 0.00 |
| categories | 0 | 0.00 |
| Unnamed: 0 | 0 | 0.00 |
| videos | 0 | 0.00 |
| description | 0 | 0.00 |
| features | 0 | 0.00 |
| rating_number | 0 | 0.00 |
| average_rating | 0 | 0.00 |
| images | 0 | 0.00 |

Over 50% of items lack clear pricing data, and other important components

# Pipeline Overview

Aggregate data for each item

Turn text into tf-idf matrix

Turn numerics into sparse csr matrix

Combine both to get the hybrid matrix

Create index for tf-idf

Ready for Recommendations

```
# TF-IDF vectorization for textual metadata
tfidf = TfidfVectorizer(max_features=max_features, min_df=min_df, ngram_range=ngram_range, stop_words='engl
tfidf_matrix = tfidf.fit_transform(item_df['text'])

# Select and normalize numeric features
numeric_features = ['price', 'avg_rating', 'num_ratings']
scaler = StandardScaler()

# Convert to numeric safely
numeric_data = df[numeric_features].apply(pd.to_numeric, errors="coerce").fillna(0)
numeric_scaled = scaler.fit_transform(numeric_data)
print(f"Numeric features scaled (columns: {numeric_features})")

# Convert to sparse format for concatenation
numeric_sparse = np.nan_to_num(numeric_scaled)

# Concatenate text and numeric representations
hybrid_matrix = hstack([tfidf_matrix, numeric_sparse]).tocsr()
print(f"Final hybrid matrix shape: {hybrid_matrix.shape}")

# Maintain item lookup for interpretation
tfidf_index = df["parent_asin"].reset_index(drop=True)
print("Created lookup table linking vectors to item parent_asin.\n")
int(tfidf_index.head())

hybrid_matrix, tfidf, tfidf_index
```

# Getting the k-most similar

**Problem:** Once we get the user's explicit ratings, now how do we actually represent them in the item vector space to get their k-nearest items?

**Solution:** Create an embedding for each of the items they rated (+1 and +5), in the same way the rest of the catalog was encoded. Then, find the items from the dataset that are most similar to the products that the user rated highly.

$$\hat{r}_j = \frac{\sum_{i \in R} r_i \cdot \mathrm{sim}(i,j)}{\sum_{i \in R} |\mathrm{sim}(i,j)|}$$

```
# For each review the user has given:
for review in user_reviews:
    parent_asin = review['parent_asin']
    rating = review['rating']

    # Double check validity
    if parent_asin in tfidf_index.values:

        # Get the row index of the item in the hybird matrix, then compute cosine similarity
        row_idx = tfidf_index[tfidf_index == parent_asin].index[0]
        sims = cosine_similarity(hybrid_matrix[row_idx], hybrid_matrix)[0]

        # Top_k determines how many similar items to consider for each item the user has rated
        # runtime gets longer for higher k vals, we can disscuss val later
        k = top_k

        # Get the indices of the top k similar items
        top_k_idx = np.argpartition(sims, -k)[-k:]
        top_k_idx = top_k_idx[np.argsort(sims[top_k_idx])[::-1]]
        top_k_sims = sims[top_k_idx]


        # calculate those similarity scores
        # formula I use is user rating of current item * similarity score
        for neighbor_idx, sim_val in zip(top_k_idx, top_k_sims):

            # Exclude self-similarity
            if neighbor_idx == row_idx:
                continue
            weight = rating * sim_val
            scores[neighbor_idx] = scores.get(neighbor_idx, 0) + weight
            sim_sums[neighbor_idx] = sim_sums.get(neighbor_idx, 0) + abs(sim_val)

# sort our scores, and also normalize.
ranked_scores = {idx: score / sim_sums[idx] for idx, score in scores.items()}
ranked_items = sorted(ranked_scores.items(), key=lambda x: x[1], reverse=True)
```

```
[('B000068WSA', np.float64(4.0)),
 ('B000HWSKNK', np.float64(4.0)),
 ('B00N52DN8Q', np.float64(4.0)),
 ('B001W30G44', np.float64(4.0)),
 ('B0015RCVRM', np.float64(4.0)),
 ('B018K31N68', np.float64(4.0)),
 ('B0001DB6J0', np.float64(4.0)),
 ('B000IN4V6S', np.float64(4.0)),
 ('B00004SV4Y', np.float64(4.0)),
 ('B0030VNLS4', np.float64(4.0))]
```

# Challenges we faced

## Problem 1: Choosing recommendation system

We decided to use a kNN, item-based content-filtering recommendation engine because our data is naturally **very** sparse, too sparse for matrix factorization or SVD.

## Problem 2: Cold start

Providing recommendations to the user given no user history is a classic recommendation problem

## Problem 3: Ghost Town

We have no REAL users that has started to use our platform!

# Initial User Profile

## How should we get likes and dislikes from the user?

### Step 1

Get a random, but thoughtful, selection of real products from the dataset

### Step 2

Present the products to the user and ask them to rate each one positively or negatively

### Step 3

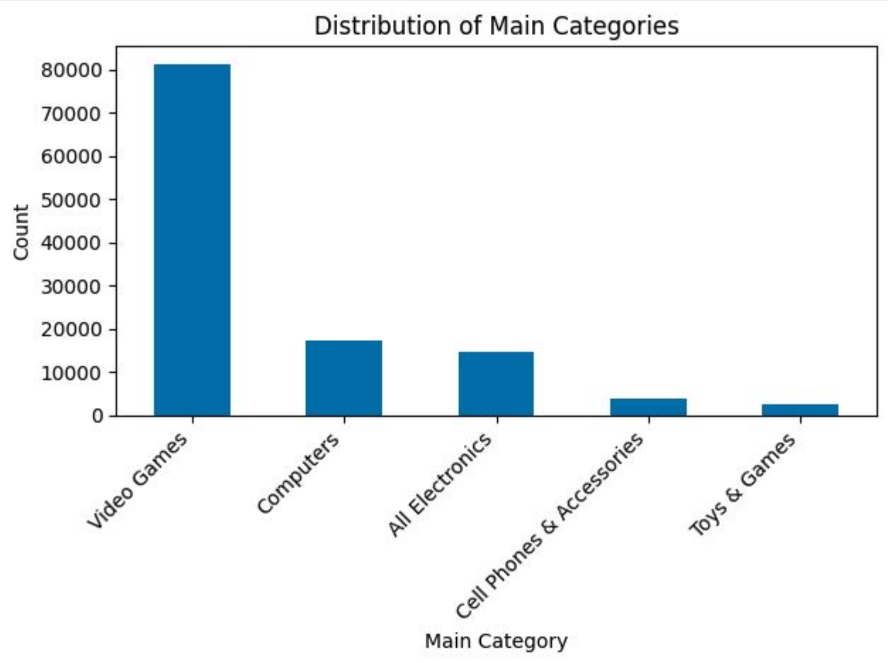
Get the most-similar items to the ones the user rated highly and return them

# Sample Dataset

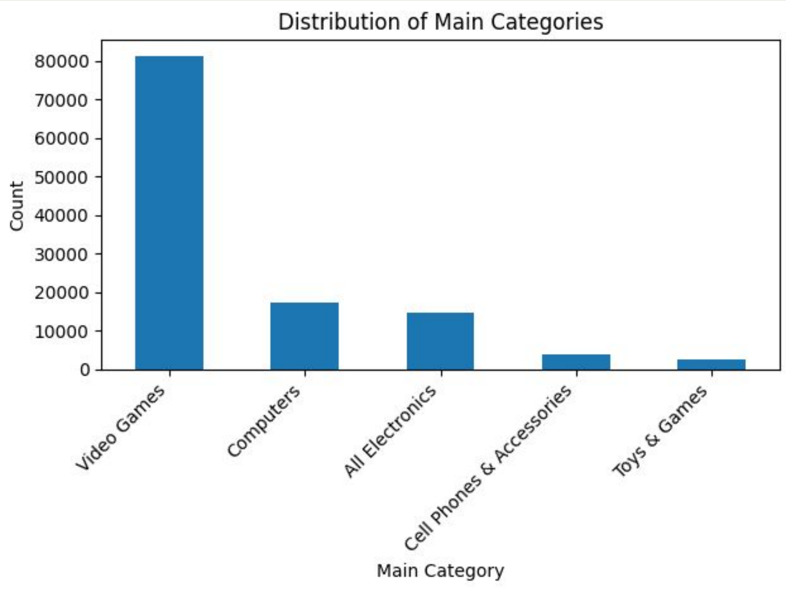
**Problem:** There are 137,269 unique products; how can we give the user a purposeful but unbiased distribution?

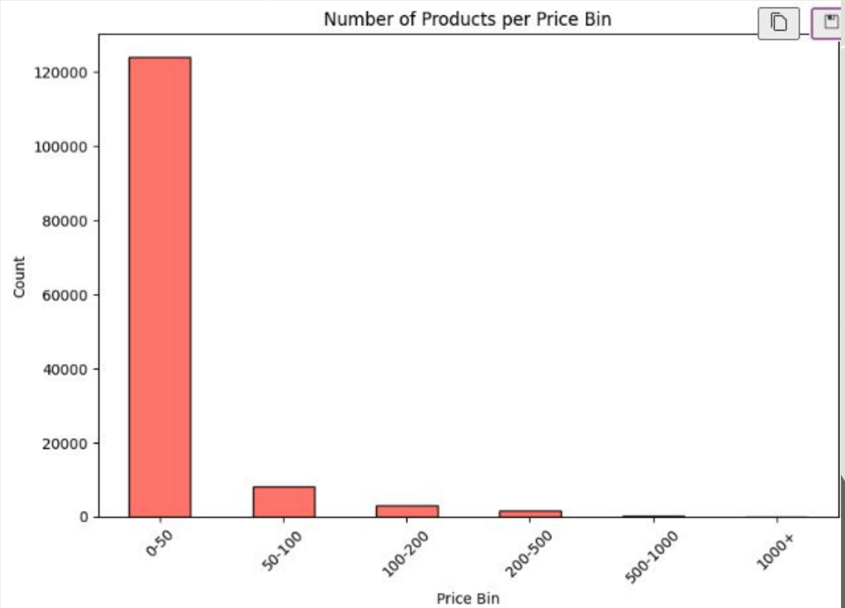
**Strategy:**
Analyze the dataset and pull out most common groups, then randomly select a sample from those groups.

Common key groups of products:
- Main categories
- Price buckets
- Random
- Overall popularity



# Key Product Groups




We also collected the top 10 most-rated products (overall popularity) and a completely random sample

# Demo

# How our model performed?

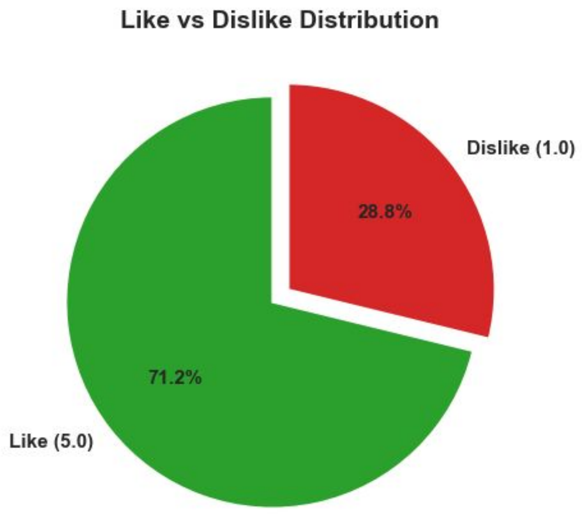
| | Precision | NDCG |
|---|---|---|
| Looking for Racing Games | 0.1 | 0.441 |
| User looking for Mouse and Headset | 0.67 | 0.72 |
| PC gamer looking for game codes | 0.60 | 0.85 |

# EVALUATIONS

**Our Evaluation Method:**

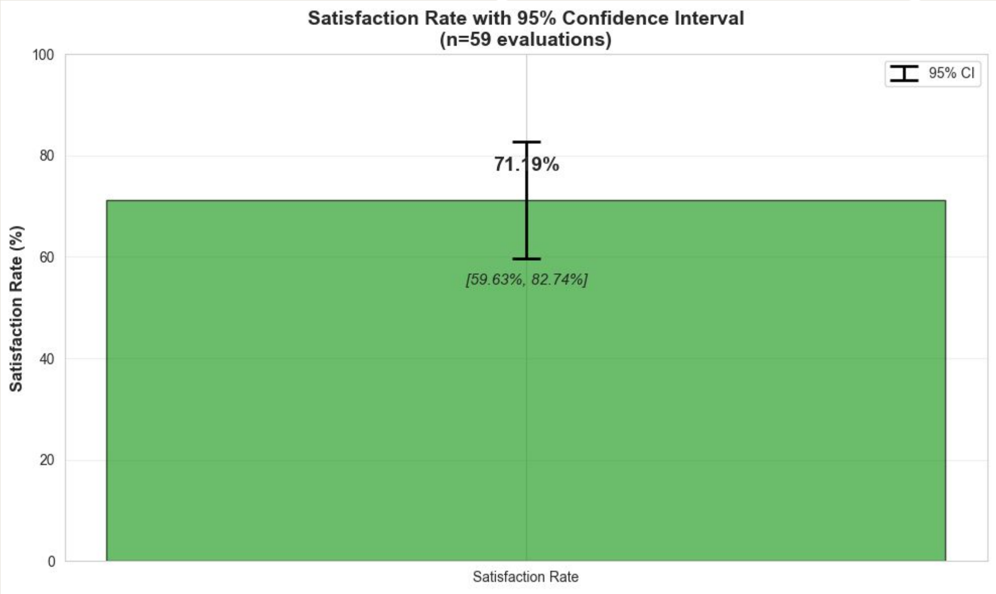
- After users have selected their initial likes and dislikes, recommendations are given to the user
- Users can then provide feedback to whether they like the recommendations that are provided
- We collected data from numerous trial runs focusing on different products and personas each time rating if we liked them or not

# Evaluation Results

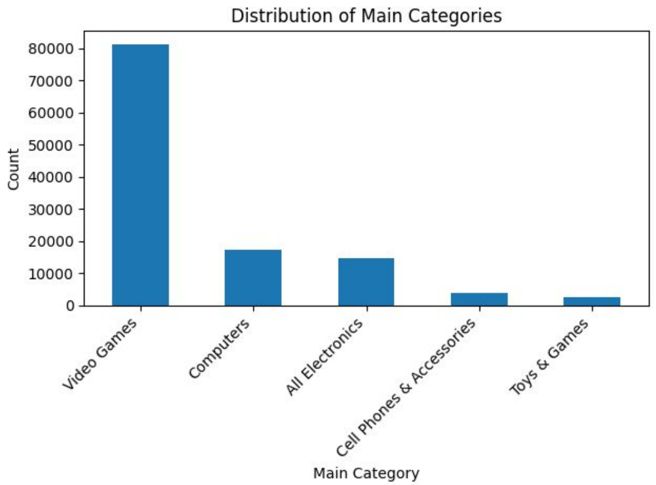



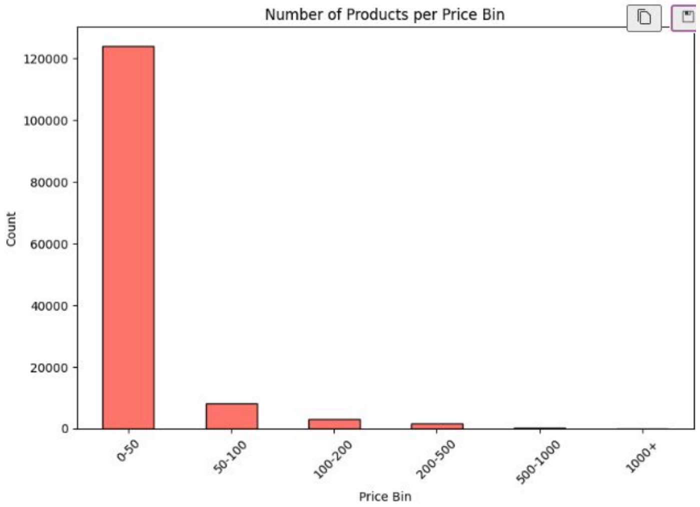
# Evaluation Results

```
================================================================
SUMMARY REPORT
================================================================
                                Metric  Value
                     Total Evaluations     59
                        Total Sessions     11
                     Total Likes (5.0)     42
                  Total Dislikes (1.0)     17
         Overall Satisfaction Rate (%)  71.19
              Overall Dislike Rate (%)  28.81
           Average Ratings per Session   5.36
Average Session Satisfaction Rate (%)  67.32
   Best Session Satisfaction Rate (%) 100.00
  Worst Session Satisfaction Rate (%)   0.00
           Unique Products Recommended     54
    Products Recommended Multiple Times      5
         95% Confidence Interval Lower  59.63
         95% Confidence Interval Upper  82.74
                   Margin of Error (%)  11.56
================================================================
```



# Any questions?


Distribution of Main Categories
Count
0
10000
20000
30000
40000
50000
60000
70000
80000
Video Games
Computers
All Electronics
Cell Phones & Accessories
Toys & Games
Main Category


Number of Products per Price Bin
Count
0
20000
40000
60000
80000
100000
120000
0-50
50-100
100-200
200-500
500-1000
1000+
Price Bin