Pratilipi Recommendation System

Goal Overview

Building a predictive model to forecast which pratilipis (stories) a user is likely to read in the future, based on historical reading behavior data.

Proposed Methodology

The system leverages both collaborative filtering and content-based approaches. It is built in Python using Pandas for data processing and LightFM for recommendation modeling.

Experimental Approaches

Approach 1:

Implements a collaborative filtering approach using LightFM with:

- Effective reading time as the primary interaction weight
  `effective_read_time = reading_time * (read_percent / 100)`

`Captures reading behavior by combining reading time and read percentage. Accounts for both how long a user spends reading and how much of the content they actually read.`

Approach 2:

Uses raw `read_percent` for SVD - collaborative filtering

Content based feature set included:

- Category features (one-hot encoded using MultiLabelBinarizer)
- Author features (one-hot encoded)
- Normalized reading time

All combined into a unified content feature matrix

Hybrid approach combining:

- Collaborative filtering (70%)
- Content-based similarity (20%) - Content-based features (categories, authors, reading time)
- Popularity score (10%)

Combines collaborative filtering, content-based features, and popularity for a balanced approach. Uses multiple content features like one-hot encoded categories, authors, and normalized reading time. Incorporates item popularity to handle cold-start scenarios.

Approach 3:

Implements a collaborative filtering approach using LightFM

- Creates a **composite weight** that combine reading time with recency that includes time decay: `composite_weight = effective_read_time * composite_recency`

Effective read_time: Incorporates effective reading time, in which the system acknowledges that longer, more complete reads indicate a stronger user preference.

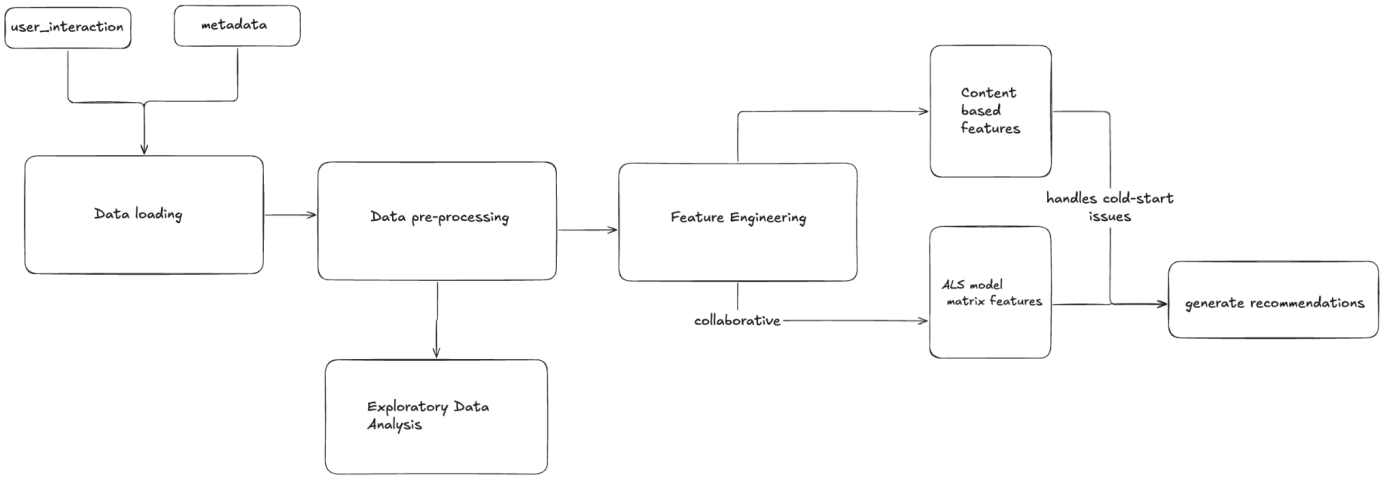
Composite_recency: Incorporates the weighted combination of

- User Interaction Recency: which gives higher weight to recent interactions and discounts older ones, ensuring that recommendations remain relevant to the user's current tastes.
  Publication Recency
- Publication Recency: The number of days between the interaction timestamp and the pratilipi's publication date is calculated. This helps to determine how new or old the content is. Newer pratilipis (with a small number of days since publication) get a higher factor as users are often more interested in recent pratilipis

Comparative Analysis

Experimental Approach	Models used	Features used
Approach 1	ALS	Effective_read_time and Content feature (category)
Approach 2	SVD	read_percent/ multiple encoded content features(category, author, reading time)
Approach 3	ALS	Composite weight- effective read time * composite recency/Content feature (category)

Detailed Workflow (current proposed method)



1. Data Loading & Preprocessing

- **Loading Data:**
  The two CSV files are read using pandas:
  - `user_interaction.csv`: Contains records of user interactions with pratilipis (includes user IDs, pratilipi IDs, reading percent, and timestamps).
  - `metadata.csv`: Contains pratilipi metadata such as category names, reading time, and publication dates.
- **Initial Inspection:**
  The first few rows of both datasets are printed to understand the structure and content. Missing values are also inspected.
- **Preprocessing:**
  - DateTime conversion for temporal features
  - Missing value checks
  - Data type validation

2. Exploratory Data Analysis

Key visualizations:

- Read percentage distribution
- Category popularity
- User activity patterns
- Content popularity trends
- Statistical summaries/descriptions for data quality checks

3. Feature Engineering

Engagement Score

- Normalize read_percent to a 0-1 range.
- Calculate recency weight based on interaction timestamps.
- Combine read percentage and recency into an engagement score.

The engagement score combines read percentage and recency to quantify how much a user is engaged with an item.
Recent interactions are weighted higher, reflecting current user preferences.

Content Length Categorization

- Categorize content into short, medium, and long based on reading time.

Content is categorized into short, medium, and long based on reading time.
Users may prefer content of specific lengths based on their reading habits, thus recommending items that match the user's preferred content length.

Content Age

- Calculate in days the difference from interaction and published date.

The age of content (in days) is calculated to prioritize newer or older items. Users may prefer newer content (e.g., trending topics) or older content (e.g., classics).

Category Preferences

User preferences for specific categories are inferred from their interaction history.
Users often have preferences for specific genres or topics, user's favorite categories, will improve personalization.

4. Model Training

Sparse Matrix Creation

- Create a user-item interaction matrix in sparse format for efficient computation.

Train-Test Split

- Split data into training and test sets based on timestamps.

Model Selection - ALS

- Train the Alternating Least Squares (ALS) model.

The ALS model is configured using the implicit library, which is optimized for implicit feedback datasets (e.g., user interactions like read percentages).

```
print("\n=== Training Model ===")

# Initialize the ALS model
model = AlternatingLeastSquares(
    factors=100,
    regularization=0.1,
    iterations=15,
    random_state=42
)

# Train the model
print("Training ALS model...")
model.fit(model_data['train_matrix'])
print("Model training complete")
```

The ALS algorithm alternates between fixing the user factors and optimizing the item factors, and vice versa. It minimizes the following loss function.

Where:

$$Loss = \sum_{(u,i)\epsilon interactions} (r_{ui} - p_u . q_i)^2 + \lambda\left(||p_u||^2 + ||q_i||^2\right)$$

r_{ui}: Observed interaction (e.g., normalized read percentage).

p_u : User latent factors.

q_i : Item latent factors.

λ: Regularization parameter.

The algorithm iteratively updates the user and item factors until convergence or the specified number of iterations is reached.

After training, the model learns:

- **User Factors:** Latent representations of users in a 100-dimensional space.
- **Item Factors:** Latent representations of items in the same space.

These factors are used to predict user-item affinities and generate recommendations.

5. Recommendation Generation

Collaborative Filtering

- Generate top-5 recommendations for each user using the trained ALS model.

- The engagement score is used as the interaction strength in the user-item matrix.

Cold-Start Handling

- For users with no interaction history, category preferences and popularity are used to make recommendations.
- New users receive relevant recommendations based on popular items or their inferred preferences.

Hybrid Recommendations

- Content-based features (e.g., category, length) are combined with collaborative filtering to enhance recommendations.

A balanced mix of personalized and popular recommendations, improving coverage and relevance.

6. Results

The top 5 recommendations were created for the users and stored in a csv format file:

	user_id	pratilipi_id
0	5506791961876448	1377786228179478
1	5506791961876448	1377786228108450
2	5506791961876448	1377786228204748
3	5506791961876448	1377786228168958
4	5506791961876448	1377786228214891
...	...	...
1218025	5506791985001260	1377786215931338
1218026	5506791985001260	1377786215640994
1218027	5506791985001260	1377786215601962
1218028	5506791985001260	1377786215601277
1218029	5506791985001260	1025741862639304

1218030 rows × 2 columns

Setup Installation

- Clone the repository
- Download the dataset and keep it in the cloned repo folder.
- Move to the applied-recommendation-system folder.
- Install the dependencies by running the requirements.txt
- Change the path of the csv in config.py
- Run the main file.