A Content-Based Movie Recommender System Pipeline Using Textual and Metadata Features

Özlem Karabulut(6872575)
Department of Computational Linguistics / University of Tübingen
`oezlem.karabulut@student.uni-tuebingen.de`

Abstract

Recommendation systems have become an integral component of digital platforms, enabling personalized experiences and improving user engagement. This paper presents the design and implementation of a movie recommendation system pipeline that experiments with three major approaches: content-based filtering, collaborative filtering, and a hybrid model combining both. Using publicly available movie and rating datasets, the system was implemented in Python with machine learning libraries to process data, compute similarities, and generate recommendations. Experimental results demonstrate the strengths and limitations of each approach, although not tested formally according to the evaluation baselines.

1 Introduction

The exponential growth of digital media has created an overwhelming number of choices for consumers. Movie recommendation systems play a crucial role in addressing the problem of information overload by helping users discover relevant content. In recent years, recommendation systems have become integral to platforms such as Netflix and Amazon Prime Video, where user satisfaction and engagement are closely tied to effective personalization. These systems not only improve user experience but also drive revenue by increasing watch time and reducing churn. This work aims to design and implement a movie recommendation system that demonstrates key approaches: demographic-based filtering, content-based filtering, collaborative filtering, a hybrid model, and a simple LM-based recommendation system. By comparing these methods with different similarity metrics and metadata combinations, I analyze their performances.

2 Related Work

Recommendation systems have been widely studied across e-commerce, social media, and entertainment industries. Maybe the simplest algorithm is demographic-based filtering. It takes the demographic data of a user to determine which items may be appropriate and then recommends items based on the description of formerly evaluated items and information obtainable from the content(Ryngksai and Chameikho, 2014). Later, two dominant paradigms have emerged: (1) Content-based filtering (Pazzani and Billsus, 2007), which relies on item attributes to suggest similar items, and (2) Collaborative filtering, recommending items based on user behavior, typically leveraging ratings or interaction history. Collaborative filtering can be user-based, where recommendations are made by finding similar users, or item-based, where items are recommended based on similarity in user rating patterns (Sarwar et al., 2001). Hybrid methods combine the two to overcome limitations such as cold-start problems and data sparsity. The implemented system builds on these foundations, comparing these approaches using the same datasets.

3 Methods

The system was developed using Python in a Jupyter notebook environment. The workflow consisted of:

1. Dataset Selection: Movies and ratings data were loaded from a publicly available dataset (See Section 3.1).

2. Exploratory Data Analysis: Data analysis is crucial for working with any kind of recommender systems. Understanding the distribution of the data, how many missing values exist, is part of this process, and design decisions such as how to handle missing values are decided based on data analysis, which is explained in detail in Section 3.2.

3. Handling Missing Values: There are many

ways to handle missing values, ranging from column/row deletion to drop empty fields, mean/mode imputation, to duplicate, assigning indicator/placeholder values, or handling them with the help of other values: data augmentation. Different design decisions are made in this work to handle missing values (See Section 3.3.

4. Data Augmentation: With the use of an instruction-tuned LM, a field in the dataset is populated.

5. Demographic-Based Content Filtering: This was done by simply retrieving the most popular movies watched by the audience.

6. Content-Based Content Filtering: Different similarity matrices were computed from various movie feature vectors to generate recommendations.

7. Collaborative Content Filtering: Both user-based and item-based collaborative filtering were tested, using similarity scores derived from user rating patterns.

8. Hybrid Model: The system integrated predictions from both approaches, weighting them to optimize accuracy.

9. LM Embeddings-Based Model: This model aims to test whether contextual embeddings achieve better accuracy in recommendation quality, although it has not been fully explored.

3.1 Datasets

Two (textual) datasets from Kaggle are used for this project:

- **MovieLens-latest-small**[1]: This dataset contains `movies.csv`, which includes simple movie data such as ID, title, and genres. `ratings.csv` serves as the backbone, particularly when building a user-based recommender system. It serves as a baseline, combined with the average ratings from the TMDb dataset, which is on a scale of 10, while MovieLens user ratings are on a scale of 5. `tags.csv` includes additional info, which is useful for enhancing our recommender system with additional data, such as the most highlighted features of a movie. `links.csv` will be used for further improvements when we want two datasets. This is especially useful for multimodal recommender systems, since TMDb dataset also includes virtual data.

- **Tmdb-5000**[2]: This dataset is highly comprehensive and has pre-calculated average rating data, which I use in the ranking system. Also, it is richer in various data fields such as plot, crew, cast, director, original title, and many more. For more information, please refer to the link provided. `credits.csv` consists of cast, crew, and director data.

3.2 Explaratory Data Analysis

In this part, I utilize simple libraries and their functions to explore the datasets. This step is crucial and valuable as it lays the foundation for the subsequent processes. It is essential to reduce the data to the required size to address the issue of sparsity. Additionally, handling missing values is another important aspect of resolving this issue, and this can also be accomplished through data exploration. Thus, I inspect dataframes using the `pandas` library, identifying unnecessary columns and rows with empty values, and making design decisions based on the number of null values present.

After import, initial shapes are inspected (using `.info()` and `.describe()`). For instance, the TMDb table contains 4,803 movies with $\sim$20 columns (budget, genres, overview, vote_average, etc.), while MovieLens ratings span $\sim$100,836 entries (rating mean $\approx$3.5).

For TMDb, JSON-like columns are parsed for inspection (*e.g.*, genres is a list of genre dicts). Converting genres to strings and counting, I find the most common genres are Drama ($\approx$2,297 films), Comedy (1,722), Thriller (1,274), etc.. These counts confirm film variety and guide any genre-based recommendations. I also note many nulls in a small number of fields: tagline or homepage. The vote_count and vote_average fields vary widely, motivating a weighted scoring later (see Section 3.8).

In MovieLens ratings, each user has rated dozens of movies on a 0.5–5 scale.

I also extract and flatten nested lists: after parsing, the `genre_names` column (a list of genre names

[1]https://https://www.kaggle.com/datasets/grouplens/movielens-latest-smal

[2]https://https://www.kaggle.com/datasets/tmdb/tmdb-movie-metadata

per film) is exploded to count frequencies for EDA.

Overall, EDA establishes data sizes and highlights missing values that must be addressed. Additionally, throughout the similarity calculation processes, we must convert our data into matrices. During this, data preprocessing steps are executed, including cleaning text by removing extra spaces and adjusting case sensitivity, as well as creating new metadata fields by combining columns.

3.3 Handling Missing Values

I merge the TMDb main table with the credits data (on `movieID`) so that each movie now has cast and crew JSON fields. These will later feed into feature construction (*e.g.*, director name, lead actors).

I clean the newly merged *TMDb+credits* dataframe(`tmdb_data`) by dropping or filling missing fields. For instance, the homepage URL column is mostly null and not necessary for our systems, so I dropped it. Any movies still missing a critical field are removed: *e.g.*, I found 3 movies with no overview text (which is central to content-based methods) and 1 with null release_date; I drop those entries. After this, all remaining movies have non-null overview, release date, and tagline (The latter is done by data augmentation, described in the next section).

3.4 Data Augmentation

For this additional step, I chose to augment one field that I will use later with the help of an LM. The TMDb movie data has missing `tagline` text for some films. To address the significant proportion of missing `tagline` fields, which included popular and highly rated movies, I opted to augment these fields using a fine-tuned language model (LM), rather than simply omitting rows or assigning empty strings. Instead, I leveraged the TmDB dataset's rich contextual information: giving movie titles, overviews, and also combining available tag columns from the MovieLens dataset, which adds additional rich contextual and creative data, as prompt inputs to generate plausible taglines:

```
prompt = (
    "You␣are␣a␣creative␣movie␣marketer.\
       n"
    "Write␣a␣single,␣catchy,␣one-
       sentence␣movie␣tagline␣based␣on␣
       the␣info␣below.\n"
    "Return␣only␣the␣tagline.\n\n"
    f"Title:␣{title}\n"
    f"Overview:␣{overview}\n"
)
```

Initially, I experimented with small bidirectional autoregressive models such as `T5` and `BART` for text generation, as these models have demonstrated efficacy in similar tasks. However, probably due to the limited data, text generation span capabilities of these small models predominantly produced repetition of the input prompts as outputs. Subsequently, I employed `Flan-T5`, a fine-tuned variant of `T5`, which yielded more varied and contextually appropriate taglines. Nevertheless, many of the generated taglines were still simple repetitions of the movie titles. The most promising results were achieved using the instruction-tuned model `Llama-3.2-3B-Instruct`[3]. With a carefully crafted prompt structure, this model generated taglines that closely resembled real-world data, offering a compelling road for future improvements in data augmentation. This step showcases how LMs can augment datasets, demonstrating advanced data engineering.

3.5 Filtering Approaches

The three most commonly used approaches of recommendation systems are tested with different methods. These approaches are:

1. **Demographic-Based Simple Filtering:** This offers generalized recommendations to every user based on some properties, such as movie popularity. The basic idea behind this recommender is that movies that are more popular and more critically acclaimed will have a higher probability of being liked by the average audience. This model does not give personalized recommendations based on the user.

   The implementation of this model is extremely trivial. I first sort movies based on ratings and display the top movies. I compute a weighted score akin to IMDb's formula, seen in Section 3.8. We apply this to all TMDb movies with at least m votes, yielding $N \approx 480$ films. Sorting by this score produces a popularity ranking.

2. **Content-Based Filtering:** This method recommends items to a user based on the attributes of the items and the user's past behavior. The core idea is that if a user liked a particular movie, they are likely to enjoy other

[3]https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct

movies with similar attributes. Content-based filtering creates a profile for each user based on the features of the items they have interacted with. Similarity between items can be computed using cosine similarity, Euclidean distance, or other distance metrics. The major advantage of this method is its ability to provide personalized recommendations without relying on other users' data. However, it suffers from the "serendipity problem," meaning it may fail to suggest items outside a user's known preferences.

I build content-similarity recommenders using movie metadata. The notebook experiments with several similarity measures on different feature sets:

- **Overview-based TF-IDF + Cosine similarity:** I vectorize each movie's overview text with `TfidfVectorizer(stop\_words='english')`. For example:

```
from sklearn.feature_extraction.
    text import TfidfVectorizer
tfidf = TfidfVectorizer(
    stop_words='english')
tfidf_matrix = tfidf.
    fit_transform(tmdb_data['
    overview'])
```

  Then compute the pairwise cosine similarity matrix via `sklearn.metrics.pairwise.linear_kernel`. Given a target movie title, I look up its index and sort other movies by similarity. The function `get_recommendations()` returns the top 10 similar titles. For example, querying *The Dark Knight Rises* returns several Batman-related titles (*The Dark Knight, Batman Forever, Batman Begins, etc.*), as expected: these share high-overlap words like "Batman," "Gotham," etc.
- **Overview TF-IDF + Euclidean distance:** For sanity, I also try unnormalized TF-IDF with Euclidean distance (converted to similarity by $1/(1+d)$). This produces very similar results to cosine, confirming consistency.
- **Genre Jaccard similarity:** Since genres are categorical, I encode each movie's `genre_names` list with a multi-hot binary vector (`MultiLabelBinarizer`) and compute Jaccard similarity (1 - Jaccard distance) between genre sets. For example, *The Notebook* (romance) is closest (by this measure) to titles like *A Beautiful Mind* (drama and romance) and *Mona Lisa Smile* (period drama and romance).
- **Cast/Crew Bag-of-Words:** I concatenate key metadata (lead cast, director, genres) into a text string per movie and vectorize via `CountVectorizer` and TD-IDF. The resulting cosine matrix yields similarity based on shared actors/directors. For *The Dark Knight Rises*, the top content-based matches included *Batman Begins*, *The Prestige*, etc. (all sharing Christopher Nolan or the cast).

Each content-based variant yields a ranked list of similar movies to a given query movie. I note that no single metric is "best" in all cases; hence, I implement and compare multiple (cosine, Euclidean, Jaccard) and even hybridize them later.

3. **Collaborative-Filtering:** The previous method is only capable of suggesting movies that are close to a certain movie, not capable of capturing tastes and providing recommendations across genres. Thus, it is not really personal in that it doesn't capture the personal tastes and biases of a user. Anyone querying the engine for recommendations based on a movie will receive the same recommendations for that movie, regardless of who she/he is. This method takes people's ratings into account. There are 2 ways to do that:
   i. User-based collaborative filtering. These systems recommend products to a user that similar users have liked. For measuring the similarity between two users, we can either use Pearson correlation or cosine similarity. Although computing user-based CF is very simple, it suffers from several problems. One main issue is that users' preferences can change over time. It indicates that precomputing the matrix based on their neighboring users may lead to bad performance. To tackle this problem, we can apply item-based CF.
   ii. Item-based collaborative filtering. Instead of measuring the similarity between users, the item-based CF recommends items based on their similarity with the items that the target user rated. Likewise, the similarity can be computed with Pearson or cosine

similarity. The major difference is that, with item-based collaborative filtering, we fill in the blank vertically, as opposed to the horizontal manner that user-based CF does.This method successfully addresses the challenge of changing user preferences since item-based collaborative filtering (CF) is more stable. However, there are still several issues. The main problem is scalability. The amount of computation needed increases with the number of customers and products. In the worst case, it can become very complex with O(mn), where m is the number of users and n is the number of items. Another issue is sparsity. To tackle both scalability and sparsity in CF, we can use a latent factor model to find similarities between users and items. Essentially, we want to turn the recommendation process into an optimization task. We need to see how well we can predict a user's rating for various items. A common way to measure this is Root Mean Square Error (RMSE); a lower RMSE indicates better performance. A latent factor refers to a characteristic that users or items possess. For example, in music, a latent factor could be the genre of a song. Singular Value Decomposition (SVD) reduces the size of the utility matrix by extracting its latent factors. This process places each user and item into a shared latent space, making it easier to understand the relationships between them.

- I use the `surprise` library to perform Singular Value Decomposition on the user–item rating matrix. I load the ratings into surprise's dataset, then run 5-fold cross-validation of `surprise.SVD()`. The mean test RMSE is about 0.87 (with MAE≈0.67). This indicates a reasonably good fit to predict held-out ratings. After training on the full data, I obtain a model `svd` that can predict any user's rating for any movie. For instance, predicting User 1's rating of *The Dark Knight Rises* yields about 4.24 (on a 5.0 scale). I can then sort all movies by the predicted rating for a user to recommend top titles.
- Item-Based CF (Pearson): I also compute an item–item similarity matrix via Pearson correlation of columns in the user–movie rating pivot table. Concretely:

```
rating_matrix = ratings.pivot(
    index='userId', columns='
    movieId',values='rating')
item_sim = rating_matrix.corr(
    method='pearson',
    min_periods=5)
```

  Given a target user, I take all movies they have rated and weight each candidate movie's score by the weighted sum of similarities to those rated items. This yields personalized recommendations. For example, for user id 1, the top-10 CF recommendations included *Brave Little Toaster*, *Rush Hour 3*, *Wind River*, etc., none of which user 1 had rated. (This list is drawn from similarity to the user's past rated movies.) I also computed each user's actual top-rated films: *e.g.* user 1's favorite rated movies include *Dr. No*, *Mad Max 2*, *Star Wars*, etc..

These collaborative methods capture patterns like "users who liked A also like B". The SVD approach generalizes well but requires retraining with new ratings, while item–item scales directly to more items.

4. **Hybrid Recommender:** Hybrid recommendation systems combine multiple approaches to leverage the advantages of each. Common hybrids include content-based + collaborative filtering, or demographic + collaborative filtering. By combining methods, hybrid recommenders can overcome the limitations of individual models, such as the cold-start problem (difficulty recommending new items to new users) or overspecialization in content-based systems. Hybrids may be implemented sequentially (one model refines the output of another) or in parallel (a weighted combination of scores from multiple models). This approach often leads to improved accuracy and more personalized recommendations.

   To combine the strengths of content and CF, I implement a simple hybrid: given an input movie and user, first take the top 25 content-similar movies (by TF-IDF cosine on overview), then re-score those by the CF model for that user. In code, for user U and movie T:

   (a) Find indices idx of movies most similar to T.
   (b) Form a candidate set of these movies (*e.g.* excluding T itself).
   (c) For each candidate movie with TMDb ID, use the SVD model to predict `svd.predict(U, candidate_movieId)`.
   (d) Sort candidates by this predicted rating and output the top-N.

5. **Transformer-Based Recommendation:** Modern recommendation systems increasingly leverage deep learning models, particularly transformer-based architectures like BERT or SBERT, to capture semantic similarities between items and users. Instead of relying solely on metadata or ratings, embeddings are generated for items and user profiles in a shared vector space. Recommendations are then made by finding items with embeddings close to a user's preferences. This approach allows the system to capture complex relationships and subtle nuances in content. I explore a transformer-based semantic search approach, using the *SentenceTransformer* model *all-MiniLM-L6-v2* to embed movie metadata into a vector space. I use previously created `key_metadata`. Then, encode all movies into embeddings:

```
from sentence_transformers import
    SentenceTransformer
model = SentenceTransformer('
    sentence-transformers/all-MiniLM
    -L6-v2')
movies_embeddings = model.encode(
    tmdb_data['key_metadata'].tolist
    ())
```

   To get recommendations, the model accepts a natural-language query (*e.g.* "Romantic comedies released in the 1990s") and encodes it into the same embedding space. Then cosine similarity between the query vector and all movie embeddings is calculated, returning the top-N most similar movies.

3.6 Vectorization Methods:

- **TF–IDF on single values (*e.g.*, overview):** Term Frequency–Inverse Document Frequency (TF–IDF) converts text data, such as movie overviews or plot summaries, into numerical vectors that reflect the importance of words relative to the corpus. This method allows the system to compute similarity between items based on their textual content and is particularly useful for content-based recommendations.

- **CountVectorizer on combined metadata**: It transforms categorical or textual metadata into numerical vectors by counting the occurrences of each token. When applied to combined metadata, it creates a sparse representation capturing the presence or absence of features. This representation enables similarity computation across multiple attributes and improves recommendations based on metadata rather than just textual descriptions.

- SBERT / embeddings: Sentence-BERT (SBERT) and other embedding-based models map items into a dense vector space that preserves semantic meaning. Unlike TF–IDF or CountVectorizer, embeddings capture contextual relationships and deeper semantic similarities. Similarity between items can then be computed using cosine similarity or other distance metrics, providing richer and more accurate recommendations.

3.7 Similarity Metrics:

- **Cosine similarity:** It is commonly used for TF–IDF and CountVectorizer outputs and measures the cosine of the angle between two vectors, capturing their directional alignment in high-dimensional space.

- **Euclidean similarity:** Euclidean distance measures the straight-line distance between two vectors. For similarity scoring, Euclidean similarity is often converted using $\text{similarity} = \frac{1}{1+\text{distance}}$. This metric provide an alternative to cosine similarity and can serve as a sanity check.

- **Jaccard distance:** Measures dissimilarity between two sets by dividing the size of their intersection by the size of their union. Particularly useful for sparse binary data, such as the presence/absence of genres or keywords.

- **Pearson similarity:** Computes the correlation between two vectors, accounting for differences in magnitude and centering around the mean. This metric is commonly used in collaborative filtering to compare user or item rating patterns.

[]

Title	Vote Count	Vote Average	Score
The Shawshank Redemption	8205	8.5	8.0589
Fight Club	9413	8.3	7.9390
The Dark Knight	12002	8.2	7.9198
Pulp Fiction	8428	8.3	7.9043
Inception	13752	8.1	7.8630

Table 1: Comparison of vote statistics and scores for selected films.

3.8 Ranking:

For ranking, the previously used IMDb formula(IMDb, 2016) is used:

$$\text{Weighted} = \frac{v}{v+m} \cdot R + \frac{m}{v+m} \cdot C \quad (1)$$

Where:

- R is the average rating for the item,
- v is the number of votes for the item,
- m is the minimum votes required,
- C is the mean rating across all items.

3.9 Recommendation Function:

A modular recommendation function takes the movie title and similarity metric as input and outputs the top-10 similar movies based on the metric given as an argument. Another improved recommendation function filters movies by ranking them based on the popularity field.

3.10 Results

Each technique outputs ranked lists; I highlight a few key results:

- **Demographic-based Simple Filtering:** The top movies by this metric are listed in Table 1:

  This list matches intuition: classics like *Shawshank Redemption* rank high. This demographic-based recommender is non-personalized but often serves as a baseline or a fallback for brand-new users.

- **Content-based Filtering:** Example output for *Blade Runner* includes *The Matrix*, *Interstellar*, and *Gravity*, reflecting similar sci-fi themes. Euclidean TF-IDF also gave comparable results. For genre Jaccard on *The Notebo*k, I got *A Beautiful Mind* and *Mona Lisa Smile*, capturing similar romance/drama categories.

- **Collaborative Filtering:** SVD's cross-validation reported RMSE $\approx$0.873, indicating reasonable predictive accuracy. For a test user (id 1), SVD predicted a rating of $\sim$4.24 for *The Dark Knight Rises*. The Pearson-based recommender for User 1 produced a list including unexpected but relevant titles given the user's history. In sum, it achieved strong personalization but struggled with new users and movies (cold-start problem).

- **Hybrid Recommender:** The hybrid recommendations for *Dr. No* prioritized James Bond sequels and associated films, balancing content-similarity and CF scores. This demonstrates that the hybrid can, *e.g.*, surface *From Russia with Love* for a Bond fan (which pure CF might not if the user hadn't rated early Bond films). To summarize, hybrid methods outperformed individual approaches by combining the strengths of both, offering a balanced solution that reduced sparsity and improved overall recommendation quality.

- **Transformer-Based Recommender:** For the query "Romantic comedies released in the 1990s" yields movies like *Four Weddings and a Funeral* (1994) and *Sleepless in Seattle* (1993) among the top hits, correctly matching both genre and era, while also having many year mismatches (*e.g.*, *Two Weeks Notice* (2002)). This method allows free-text queries rather than specifying an exact movie, demonstrating the flexibility of transformer-based embeddings, and with further experiments, a sufficiently large model can perform better.

Together, these results indicate that each method works as expected. Visual inspection of examples confirms quality, though I did not compute a unified metric (since I lack explicit "ground truth" preferences).

Also, some other highlights I gained from the experiments are as follows:

- Cosine similarity usually works best for TF–IDF vectors.

- CountVectorizer method yielded better results compared to TF-IDF when working with metadata fields consisting of longer text and more diverse data fields.
- Metadata features like genres and top cast can improve relevance.
- The same recommender system with different metadata (tagline+ overview vs. cast+crew+genre) showed intuitively more relevant results with the metadata combining cast, crew, and genre.
- Popularity filtering prevents obscure, low-rated movies from appearing, or at least pushes them to the end of the list. With better methods, they can be hidden. (*e.g.*, *Batman&Robin* with 4 rating is still in *The Dark Knight Rises* listing).

Conclusion

I experimented with different methods for a movie recommender, demonstrating the spectrum from simple popularity-based ranking to advanced transformer-based semantic search. The content-based filters effectively retrieve similar movies based on descriptions and metadata. The collaborative filtering leverages user behavior to personalize recommendations; the SVD model achieved a test RMSE $\approx$0.87, showing decent accuracy. The hybrid system combines these to tailor suggestions to both movie context and individual taste. Lastly, the transformer-based model enables intuitive text queries (“kid-friendly animated films”) to retrieve matching movies by meaning.

Limitations: Each approach has trade-offs. Popularity-based filtering ignores individual preferences. Pure content-based filtering cannot recommend outside the immediate “content neighborhood” of a movie. CF models struggle with new movies or sparse users (cold start). The hybrid recommender partially alleviates this but still depends on having at least one known movie or rating. The transformer search relies on the quality of metadata, a good similarity metric, a carefully chosen LM, and a tailored prompt. A big lack of the system is that I could not have an evaluation based on user demographic data or real-time feedback. Therefore, it is not sensible to make certain evaluations on the performances of different methods without having a proper, well-structured evaluation baseline.

Future work: Integrating user profiles or social data to better personalize is a promising step. Also, deep learning approaches (e.g., neural collaborative filtering, knowledge graphs, or end-to-end transformer models) might capture deeper patterns but require exhaustive computational resources. Additional modalities like poster images (via multimodal embeddings) could enrich recommendations. Most importantly, having a strong evaluation baseline, such as A/B testing or using held-out ratings to quantitatively compare these models’ efficacy, should be the immediate step to have a solid foundation for the claims of this study.

Nonetheless, this project achieved its goal of demonstrating a comprehensive pipeline testing, rather than having strong theoretical claims. From raw data through augmentation, cleaning, multiple algorithms, and example outputs, reflecting current industry practices in movie recommendation is successfully conducted in this phase of the project and will be improved further.

Acknowledgments

References

IMDb. 2016. Imdb top tv chart. https://web.archive.org/web/20160315073311/http://www.imdb.com/chart/toptv/. Accessed: 15 March 2016.

Michael J. Pazzani and Daniel Billsus. 2007. *Content-Based Recommendation Systems*, pages 325–341. Springer Berlin Heidelberg, Berlin, Heidelberg.

Iateilang Ryngksai and L. Chameikho. 2014. Recommender systems: Types of filtering techniques. *International Journal of Engineering Research & Technology (IJERT)*, 3(11).

Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. *Proceedings of ACM World Wide Web Conference*, 1.