Mini Project: Recommender Systems

Alexander Stæhr Johansen
Department of Computer Engineering
Aarhus University
Aarhus, Denmark
201905865@post.au.dk

Henrik Tambo Buhl
Department of Computer Engineering
Aarhus University
Aarhus, Denmark
201905590@post.au.dk

Liulihan Kuang
Department of Computer Engineering
Aarhus University
Aarhus, Denmark
201906612@post.au.dk

Instructors:
Christian Fischer Pedersen & Christian Marius Lillelund

Team 3

Contents

1. Introduction

A recommender system, also named a recommendation system, belongs to the class of machine learning algorithms that aims to predict user preferences and interests based on their past behavior and provide personalized recommendations accordingly [1]. As the amount of data available on every user of the Internet has increased significantly, the accuracy of these systems has as well, and they are now considered a crucial part in many industries such as e-commerce, media and entertainment, travel and hospitality healthcare.

It is therefore of interest to explore the most common implementation techniques of recommender systems theoretically, and practically, and compare them.

2. Methods

In this section, we will elaborate on the three most common implementation techniques of recommender systems: content-based filtering, collaborative filtering, and hybrid systems.

2.1. Content-based filtering

Content-based filtering relies on product/service-specific features to determine the most relevant recommendations for the user [2]. The technique analyzes the user's past behavior, encapsulates this as a set of features, and compares them to the features of a proposed product/service. It then compares the degree of similarity between the two sets of features, and if the degree exceeds a certain threshold, the recommendation is extended to the user. For example, if a user frequently watches children's cartoon movies, the content-based filtering technique will suggest other popular children's cartoon movies to the user.

The technique encapsulates the features in two feature vectors, i.e. one of the user's profile, and one of the proposed product/service, also called an item. Examples of features in the user vector include viewing history, ratings given by the user to other items, search queries, purchase history, demographic information, or declared interests. Features in the item vector could include genre, cast crew, plot summary, keywords, language, release date, and user ratings.

The technique then compares the degree of similarity by using a quantitative measure, such as the cosine angle, as defined below:

$$Similarity = cos(\theta) = \frac{\boldsymbol{A} \cdot \boldsymbol{B}}{\|\boldsymbol{A}\|\|\boldsymbol{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{1}$$

Visually, the similarity is determined as shown in the figure below:

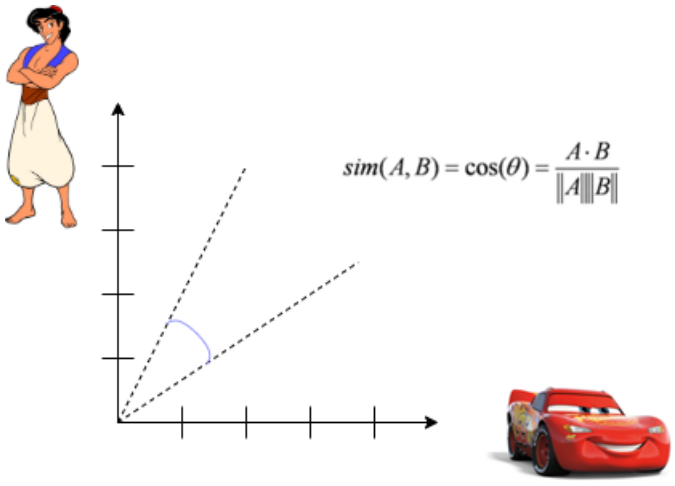

Fig. 1: Cosine similarity visualization.

The technique then determines the relevant recommendation according to the maximization of the similarity score.

2.2. Collaborative filtering

Collaborative filtering is a recommendation technique that leverages the behavior and preferences of users to recommend items [3]. In this approach, the system identifies users who have similar preferences and recommends items that similar users have liked or interacted with. Collaborative filtering can be done using two main approaches: user-based collaborative filtering and item-based collaborative filtering.

Formally, let U be the set of users and I be the set of items. In user-based collaborative filtering, the system generates a user-item matrix M, where each entry $M(u, i)$ represents the user u's preference for item i. The preference can be binary (for example liked or not liked), ordinal (for example a rating from 1 to 5), or continuous (which could be purchase amount). Given a user u, the system identifies other users who have similar preferences to u, based on their historical interactions with the items. This is typically done by computing a similarity score, for example, cosine similarity, as explained in the previous section. Once similar users are identified, the system recommends items that those similar users have liked or interacted with but that the user u has not yet interacted with. The recommended items are typically ranked by their predicted preference score, which can be computed using various methods, such as weighted sum, matrix factorization, or neural networks.

In item-based collaborative filtering, the system generates an item-item matrix N, where each entry $N(i, j)$ represents the similarity score between item i and item j. The similarity score can be computed based on various features of the items, such as genre, artist, or keywords. Given a user u, the system identifies items that the user has interacted with in the past and computes the similarity score between those items and all other items in the item-item matrix. The system then recommends items that are most similar to the items the user has interacted with, but that the user has not yet interacted with.

Collaborative filtering has been widely used in various applications, such as movie recommendations, music recommendations, and e-commerce product recommendations. The reason is that it has the advantage of being able to recommend items that the user may not have known or thought of before, based on the preferences of other users with similar tastes.

2.3. Hybrid system

In the context of recommender systems, a hybrid system is a recommendation approach that combines two or more recommendation techniques to provide personalized recommendations to users [4]. The goal of a hybrid system is to leverage the strengths of different recommendation techniques while mitigating their weaknesses.

Formally, let U be the set of users and I be the set of items. A hybrid system combines two or more recommendation techniques, each represented by a recommendation function that takes as input a user u and returns a set of recommended items. Let $R_1(u)$ be the set of items recommended to user u by the first recommendation technique, and $R_2(u)$ be the set of items recommended to user u by the second recommendation technique. The hybrid system then combines the recommendation sets $R_1(u)$ and $R_2(u)$ into a single set $R(u)$ of recommended items. Some ways of combining the recommendation sets include:

- **Weighted hybrid:** In this approach, each recommendation technique is assigned a weight that reflects its relative importance in the overall recommendation. The final recommendation set $R(u)$ is computed as a weighted sum of the individual recommendation sets $R_1(u)$ and $R_2(u)$.

- **Switching hybrid:** In this approach, the system decides which recommendation technique to use based on the user's context or behavior. For example, if any given user has a long history of interacting with the system, the system might switch to a collaborative filtering approach that leverages the user's historical interactions. Alternatively, if the user has just joined the system, the system might start with a content-based filtering approach that leverages the features of the items.

- **Cascade hybrid:** In this approach, the output of one recommendation technique is used as input to another recommendation technique. For example, the system might first generate a set of recommendations using a content-based filtering approach, and then use the output of that approach as input to a collaborative filtering approach to generate a refined set of recommendations.

Hybrid systems have been shown to be effective in improving the performance of recommender systems, particularly when dealing with sparse data, cold-start problems, or diverse user preferences.

3. Materials

In this section the dataset and the code repository will be introduced.

3.1. Code repository

The reader can find the code repository clicking HERE.

3.2. Dataset

The Small MovieLens dataset, provided by the GroupLens Research Project at the University of Minnesota, is a valuable resource for students and researchers working on recommender systems and machine learning projects [5]. Comprised of 100,000 movie ratings collected from approximately 600 users and spanning nearly 9,000 movies, this dataset offers a representative sample of user preferences and movie metadata.

In the context of this project, we have chosen the Small MovieLens dataset as the foundation for developing and evaluating our movie recommendation algorithms. The dataset includes user ratings on a scale of 0.5 to 5 stars, along with movie titles, genres, and timestamp information. This rich set of data enables us to explore various techniques, such as collaborative filtering and content-based recommendations, to provide personalized movie suggestions to users.

4. Implementation

In this section, we will present the implementation of three different movie recommendation systems using different techniques: collaborative filtering, content-based filtering, and a hybrid of the two. It is possible to test the systems individually to obtain recommendations, but the below implementations are modified to allow for the calculation of common performance metrics.

```
# Import libraries
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

# Load the movielens Small dataset from local storage
movies = pd.read_csv('movies.csv')
ratings = pd.read_csv('ratings.csv')
```

Listing 1: Importing libraries and the dataset.

4.1. Content-based recommendation system

This recommender uses content-based filtering by comparing movie genres to find movies that are similar to the ones the user has already rated. The predicted rating for each recommended movie is the mean rating the user has given to their previously rated movies.

The recommender is defined with the signature ”content_based_recommender”, which takes the following input arguments:

- **user_id:** The ID of the user for whom we want to generate movie recommendations.
- **cosine_sim:** A precomputed cosine similarity matrix between movies based on their genres.
- **df_movies:** The movies DataFrame containing movie information such as movieId and genres.
- **df_ratings:** The ratings DataFrame containing user ratings for different movies.
- **n:** The number of recommendations to generate (default value is 10).

The function then does the following:

1) Filters the user ratings for the given user_id and selects only the ’movieId’ and ’rating’ columns.
2) Calculates the mean user rating for a given user.
3) Finds the indices of the movies rated by the user in the df_movies DataFrame.
4) Calculates the cosine similarity scores between the user’s rated movies and all other movies, based on the precomputed cosine similarity matrix.
5) Sorts the movies based on their similarity scores in descending order.
6) Selects the top N most similar movies and exclude the movies already rated by the user.
7) Calculates the predicted ratings for the recommended movies by assigning the mean user rating to each of them.
8) Returns a DataFrame containing the recommended movieIds and their predicted ratings.

The code is inserted here such that the reader can inspect the implementation in detail.

```python
def content_based_recommender(user_id, cosine_sim, df_movies, df_ratings, n=10):
    user_ratings = df_ratings[df_ratings['userId'] == user_id]
    user_ratings = user_ratings[['movieId', 'rating']]

    # Compute the mean rating for the user
    mean_user_rating = user_ratings['rating'].mean()

    # Find the indices of the movies rated by the user
    movie_indices = user_ratings['movieId'].apply(lambda x: df_movies[df_movies['movieId'] == x].index
    [0])

    # Calculate the cosine similarity scores between the user's movies and all other movies
    sim_scores = cosine_sim[movie_indices].mean(axis=0)

    # Sort the movies based on the similarity scores
    sim_scores = sorted(list(enumerate(sim_scores)), key=lambda x: x[1], reverse=True)

    # Get the top N most similar movies (excluding the movies rated by the user)
    sim_scores = [score for score in sim_scores if score[0] not in movie_indices][:n]
    recommended_movie_indices = [i[0] for i in sim_scores]

    # Calculate the predicted rating for each recommended movie
    predicted_ratings = [mean_user_rating] * n

    # Return the recommended movieIds and their predicted ratings
    recommended_movies = df_movies.iloc[recommended_movie_indices]['movieId'].tolist()
    return pd.DataFrame({'movieId': recommended_movies, 'predicted_rating': predicted_ratings})
```

Listing 2: Importing libraries and the dataset.

4.2. Collaborative-filtering

This recommender uses collaborative filtering by finding users who are similar to the target user based on their movie ratings. It then recommends movies that similar users have rated highly but the target user has not seen yet. The predicted rating for each recommended movie is the average rating given by similar users. The recommender is defined with the signature ”collaborative_filtering_recommender”, which takes the following input arguments:

- **user_id:** The ID of the user for whom we want to generate movie recommendations.
- **cosine_sim:** A precomputed cosine similarity matrix between users based on their movie ratings.
- **user_item_matrix:** A user-item matrix containing users’ ratings for different movies.
- **movies:** The movies DataFrame containing movie information such as movieId and genres.
- **n:** The number of recommendations to generate (default value is 10).

The function then does the following:

1) Converts the user_id to the corresponding index in the user_item_matrix.
2) Calculates the cosine similarity scores between the target user and all other users, using the precomputed cosine similarity matrix.
3) Sorts the users based on their similarity scores in descending order, and the top N most similar users are selected (excluding the target user).
4) Converts the indices of the similar users back to user IDs.
5) Aggregates the movies rated by similar users, and the sum of their ratings is calculated. The movie IDs are sorted in descending order based on the sum of their ratings.
6) Identifies the movies already seen by the target user.
7) Selects the top N movies that have not been seen by the target user as recommendations.
8) Extracts the recommended movie IDs from the movies DataFrame.
9) Calculates the average rating for the recommended movies based on the ratings given by the similar users.
10) Returns the DataFrame containing the recommended movie IDs and their predicted ratings.

The code is inserted here such that the reader can inspect the implementation in detail.

```python
def collaborative_filtering_recommender(user_id, cosine_sim, user_item_matrix, movies, n=10):
```

```
    # Convert the user_id to the corresponding index in user_item_matrix
    idx = user_item_matrix.index.get_loc(user_id)

    sim_scores = list(enumerate(cosine_sim[idx]))
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
    sim_scores = sim_scores[1:n+1]
    similar_users_indices = [i[0] for i in sim_scores]

    # Convert the similar users' indices back to user IDs
    similar_users = user_item_matrix.iloc[similar_users_indices].index.tolist()

    movies_rated = user_item_matrix.loc[similar_users].sum().sort_values(ascending=False)
    movies_rated = movies_rated.index.values.tolist()

    movies_seen = user_item_matrix.loc[user_id]
    movies_seen = movies_seen[movies_seen > 0].index.values.tolist()
    movies_to_recommend = [m for m in movies_rated if m not in movies_seen][:n]

    recommended_movie_ids = movies.loc[movies['movieId'].isin(movies_to_recommend)]['movieId'].tolist()

    # Calculate the average rating for the recommended movies
    recommended_movie_ratings = user_item_matrix.loc[similar_users][recommended_movie_ids].mean().tolist
    ()

    # Return the recommended movie IDs and their predicted ratings
    return pd.DataFrame({'movieId': recommended_movie_ids, 'predicted_rating': recommended_movie_ratings
    })
```

Listing 3: Importing libraries and the dataset.

4.3. Hybrid system

This hybrid recommender combines content-based and collaborative filtering recommenders by generating recommendations based on both movie features and user similarities. The alpha parameter allows for control of the balance between the two recommendation approaches, and the final hybrid score is a weighted average of the predicted ratings from both models. The recommender is defined with the signature ”hybrid_recommender”, which takes the following input arguments:

- **user_id:** The ID of the user for whom we want to generate movie recommendations.
- **cosine_sim_content:** A precomputed cosine similarity matrix between movies based on their features (e.g., genres, keywords, etc.).
- **cosine_sim_cf:** A precomputed cosine similarity matrix between users based on their movie ratings.
- **user_item_matrix:** A user-item matrix containing users' ratings for different movies.
- **movies:** The movies DataFrame containing movie information such as movieId and genres.
- **ratings:** The ratings DataFrame containing user ratings for different movies.
- **n:** The number of recommendations to generate (default value is 10).
- **alpha:** A weight parameter for combining the content-based and collaborative filtering recommendations (default value is 0.5).
- **num_individual_recommendations:** The number of individual recommendations to generate from each model (default value is 100).

The function then does the following:

1) Generates the top num_individual_recommendations movies and their predicted ratings from both the content-based and collaborative filtering recommenders.
2) Merges recommendations from both models on the movieID.
3) Calculates a hybrid score for each movie by taking a weighted average of the predicted ratings from both models, using the alpha parameter as the weight.
4) Sorts the combined recommendations based on the hybrid score in descending order.
5) Returns the top N recommended movies along with their predicted ratings.

The code is inserted here such that the reader can inspect the implementation in detail.

```
def hybrid_recommender(user_id, cosine_sim_content, cosine_sim_cf, user_item_matrix, movies, ratings, n=10, alpha=0.5, num_individual_recommendations=100):
    # Get the top recommended movies and their predicted ratings from content-based and collaborative filtering recommenders
    content_based_recommendations = content_based_recommender(user_id, cosine_sim_content, movies, ratings, n=num_individual_recommendations)
    collaborative_filtering_recommendations = collaborative_filtering_recommender(user_id, cosine_sim_cf, user_item_matrix, movies, n=num_individual_recommendations)

    # Merge the two recommendation sets and compute the hybrid score
    combined_recommendations = content_based_recommendations.merge(collaborative_filtering_recommendations, on='movieId', how='inner')
    combined_recommendations['hybrid_score'] = alpha * combined_recommendations['predicted_rating_x'] + (1 - alpha) * combined_recommendations['predicted_rating_y']

    # Sort the recommendations based on the hybrid score and return the top N recommended movies
    combined_recommendations = combined_recommendations.sort_values(by='hybrid_score', ascending=False).head(n)
    return combined_recommendations[['movieId', 'hybrid_score']].rename(columns={'hybrid_score': 'predicted_rating'})
```

Listing 4: Importing libraries and the dataset.

5. Experiments and results

To measure the performance of the recommenders, we used the following three metrics:

- **Mean Absolute Error (MAE):** MAE measures the average absolute difference between the predicted ratings and the true ratings. It is computed as the sum of the absolute differences between the predicted and true ratings, divided by the number of ratings. Mathematically, it can be defined as:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_{\text{true}_i} - y_{\text{pred}_i}|$$

- **Mean Squared Error (MSE):** MSE measures the average squared difference between the predicted ratings and the true ratings. It is computed as the sum of the squared differences between the predicted and true ratings, divided by the number of ratings. Mathematically, it can be defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{\text{true}_i} - y_{\text{pred}_i})^2$$

- **Root Mean Squared Error (RMSE):** RMSE is the square root of the MSE. It measures the average magnitude of the differences between the predicted ratings and the true ratings, giving a higher weight to larger errors. Mathematically, it can be defined as:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_{\text{true}_i} - y_{\text{pred}_i})^2}$$

In these formulas, n is the number of ratings, y_{true_i} denotes the true ratings, and y_{pred_i} represents the predicted ratings. For the reader's convenience, the testing function is inserted as below:

```
import numpy as np
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.model_selection import train_test_split

# Define the evaluation metrics
def evaluate_recommendations(test_data, recommendations):
    # Filter test_data to keep only the recommended movies
    test_data_filtered = test_data[test_data['movieId'].isin(recommendations['movieId'])]

    # If there's no intersection between recommended movies and test data, return None
```

```
    if test_data_filtered.empty:
        return None

    # Reorder recommendations according to test_data_filtered
    recommendations = recommendations.set_index('movieId').loc[test_data_filtered['movieId']].reset_index
    ()

    y_true = test_data_filtered['rating'].values
    y_pred = recommendations['predicted_rating'].values

    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)

    return {'mae': mae, 'mse': mse, 'rmse': rmse}
```

Listing 5: Importing libraries and the dataset.

The experiments were then conducted using the MovieLens dataset, which was split into training (80%) and testing (20%) sets using the train_test_split function from the sklearn.model_selection module.

5.1. Results

Conducting these experiments, the results below appear.

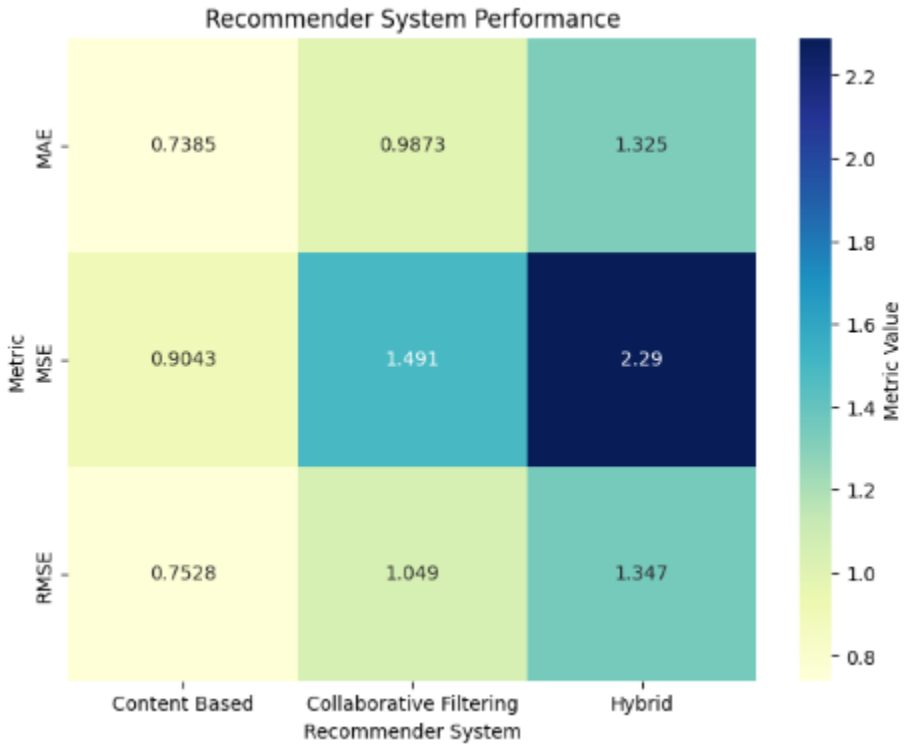

Fig. 2: Results comparison as a heatmap.

6. Discussion

From the results, it can be observed that the Content-Based Recommender outperforms both the Collaborative Filtering and Hybrid Recommenders in terms of all the evaluation metrics. This suggests that using content-based features (e.g., movie genres) to generate recommendations is more effective than using user-item interaction data (e.g., movie ratings) alone or combining both types of information in a hybrid approach.

One possible explanation for the superior performance of the Content-Based Recommender is that content features provide a more stable and reliable basis for generating recommendations, as they are less susceptible to the cold-start problem, which

occurs when there is insufficient user-item interaction data available. In contrast, the Collaborative Filtering Recommender relies heavily on user-item interaction data, making it more vulnerable to the cold-start problem and potentially less accurate when data is sparse.

The Hybrid Recommender, which combines both content-based and collaborative filtering approaches, does not seem to improve performance over the individual methods. It is possible that the weight parameter used to combine the two methods (alpha) was not optimal, and further exploration of different weight combinations could yield better results. Additionally, other hybrid approaches or more advanced techniques, such as matrix factorization or deep learning-based methods, could be explored to enhance recommendation performance.

7. Conclusion

In this project, we implemented and compared three recommender systems: Content-Based, Collaborative Filtering, and Hybrid, to provide personalized movie recommendations. Our evaluation metrics, including Mean Absolute Error (MAE), Mean Squared Error (MSE), and Root Mean Squared Error (RMSE), demonstrated that the Content-Based Recommender outperformed the other two methods in terms of prediction accuracy.

The results suggest that leveraging content features, such as movie genres, can provide more accurate recommendations compared to relying solely on user-item interaction data or combining both types of information in a hybrid approach. However, there may be opportunities for improvement by exploring alternative hybrid methods, refining weight parameters, or incorporating advanced techniques such as matrix factorization [6] or deep learning-based methods [7].

In conclusion, the Content-Based Recommender proved to be the most effective method in this study, but future work should investigate additional content features and alternative approaches to further enhance the quality of movie recommendations.

References

[1] Francesco Ricci et al. *Recommender Systems Handbook*. Boston, MA: Springer, 2011.

[2] Michael J Pazzani and Daniel Billsus. "Content-based recommendation systems". In: *The Adaptive Web*. Berlin, Heidelberg: Springer, 2007, pp. 325–341.

[3] Badrul Sarwar et al. "Item-based collaborative filtering recommendation algorithms". In: *Proceedings of the 10th International Conference on World Wide Web*. 2001, pp. 285–295.

[4] Robin Burke. "Hybrid recommender systems: Survey and experiments". In: *User Modeling and User-Adapted Interaction* 12.4 (2002), pp. 331–370.

[5] F. Maxwell Harper and Joseph A Konstan. "The MovieLens Datasets: History and Context". In: *ACM Transactions on Interactive Intelligent Systems (TIIS)*. Vol. 5. 4. ACM. 2016, p. 19.

[6] Yehuda Koren, Robert Bell, and Chris Volinsky. "Matrix factorization techniques for recommender systems". In: *Computer*. Vol. 42. 8. IEEE. 2009, pp. 30–37.

[7] Heng-Tze Cheng et al. "Wide & Deep Learning for Recommender Systems". In: *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. ACM. 2016, pp. 7–10.