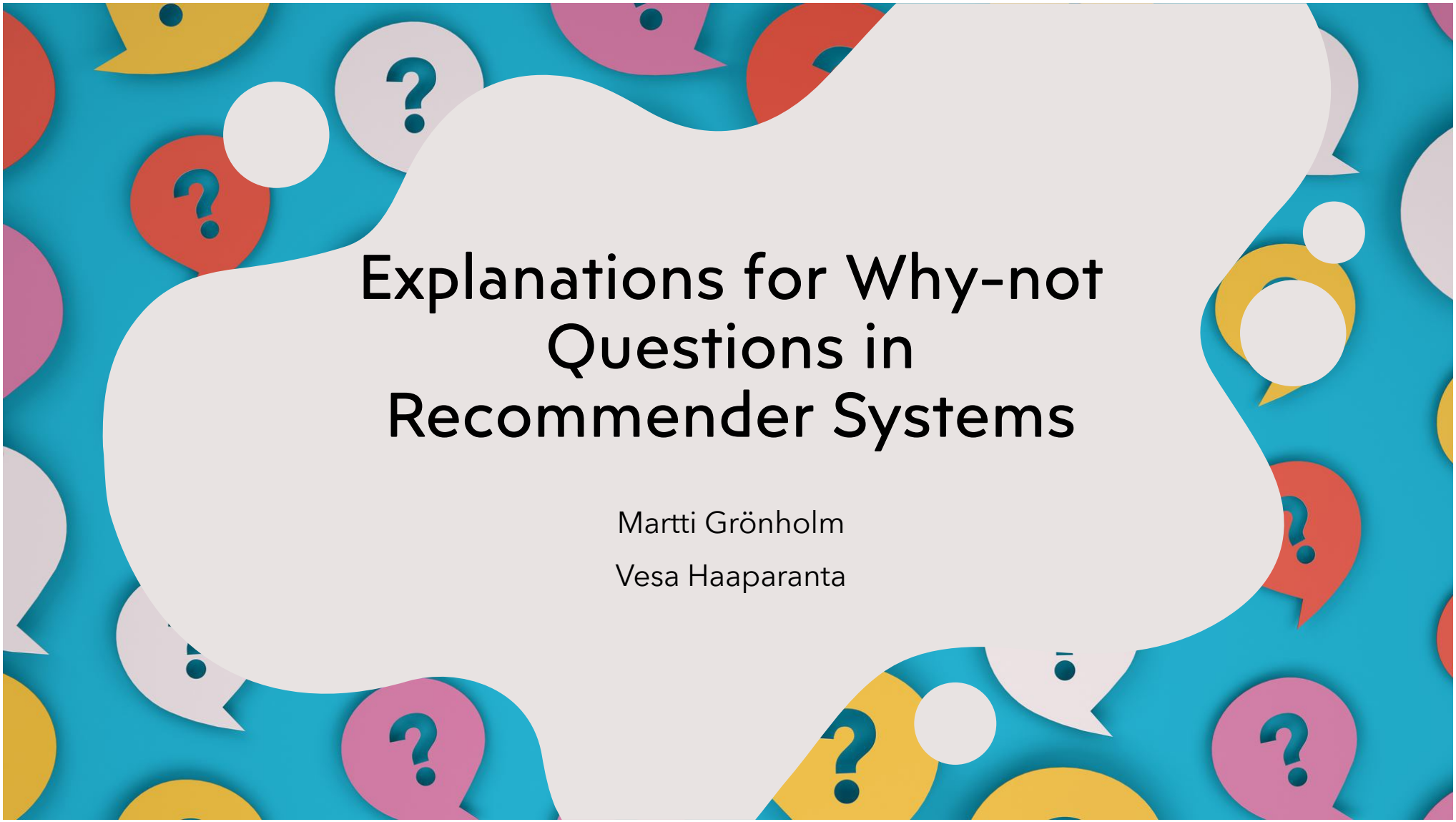
Explanations for Why-not Questions in Recommender Systems
Martti Grönholm
Vesa Haaparanta

How the method works

The method produces explanations for group recommendations for the granularity case for atomic case, group case, and position absenteeism case.

By leveraging a kNN approach and multiple sequences, it delivers diverse movie selections while maintaining transparency in the selection process.

This methodology emphasizes understanding individual movie recommendations, group genre preferences, and the significance of movie rankings. Such insights enhance user comprehension and trust in the recommendation system.

The emphasis on transparency in explaining the inclusion or exclusion of movies contributes to a more user-friendly and understandable recommendation process. Ultimately, this method enriches group experiences by tailoring recommendations to diverse tastes within the group.

Function overviews

```python
def generate_group_recommendations_with_info(user_group, ratings_matrix, top_n=10, num_sequences=3):
    group_recommendations = []
    recommendation_info = {'considered_movies': {}, 'selected_movies': {}}

    for sequence in range(num_sequences):
        knn_model = NearestNeighbors(metric='cosine', algorithm='brute')
        knn_model.fit(ratings_matrix)

        sequence_recommendations = []
        sequence_considered_movies = {}  # Track considered movies for each sequence

        for user_id in user_group:
            user_idx = ratings_matrix.index.get_loc(user_id)
            # Use all other users as potential neighbors
            distances, indices = knn_model.kneighbors(ratings_matrix.iloc[user_idx, :].values.reshape(1, -1), n_neighbors=len(ratings_matrix) - 1)

            # Flatten indices and distances, exclude the first one (self)
            flat_indices = indices.flatten()[1:]
            flat_distances = distances.flatten()[1:]

            # Pair distances with indices and sort them
            sorted_neighbors = sorted(zip(flat_distances, flat_indices))

            # Select top N similar users based on sorted distances
            similar_users_indices = [idx for _, idx in sorted_neighbors[:top_n]]

            similar_users_ratings = ratings_matrix.iloc[similar_users_indices]
            user_movies = ratings_matrix.iloc[user_idx]
            unrated_movies = user_movies[user_movies.isna() | (user_movies == 0)].index
            avg_ratings = similar_users_ratings.mean(axis=0)
            avg_unrated_ratings = avg_ratings[unrated_movies]

            # Update considered movies with average ratings
            sequence_considered_movies.update(avg_unrated_ratings.to_dict())

            # Select top N movies based on average ratings, not randomly
            top_movies = avg_unrated_ratings.nlargest(top_n).index.tolist()
            sequence_recommendations.extend(top_movies)

            # Update selected movies
            for movie in top_movies:
                recommendation_info['selected_movies'].setdefault(movie, []).append(avg_unrated_ratings[movie])

        group_recommendations.append(sequence_recommendations)
        recommendation_info['considered_movies'][sequence] = sequence_considered_movies
```

```python
def explain_atomic_case(movie_id, recommendation_info, ratings_matrix, movies_data):
    # Handle the case where the movie ID is not in the movies_data dictionary
    title = movies_data[movie_id]['title'] if movie_id in movies_data else f"Movie ID {movie_id}"

    if movie_id in recommendation_info['selected_movies']:
        return f"'{title}' was recommended."
    elif movie_id in recommendation_info['considered_movies']:
        # Ensure avg_rating is retrieved correctly
        avg_ratings_list = recommendation_info['considered_movies'].get(movie_id, [])
        avg_rating = sum(avg_ratings_list) / len(avg_ratings_list) if avg_ratings_list else 0

        # Calculate the group's average rating for the movie
        group_avg_rating = ratings_matrix.get(movie_id, pd.Series()).mean()

        # Determine the reason for not selecting the movie
        reason = "lower than group's average rating" if avg_rating < group_avg_rating else "not aligning with group's preferences"
        return f"'{title}' was considered but not selected due to {reason}."
    else:
        return f"'{title}' was not considered in the recommendation process."
```

```python
def explain_group_case(genre, movies_data, recommendation_info):
    considered_titles = []
    for movie_id in recommendation_info['considered_movies']:
        if movie_id in movies_data and genre in movies_data[movie_id]['genres']:
            considered_titles.append(movies_data[movie_id]['title'])

    selected_titles = []
    for movie_id in recommendation_info['selected_movies']:
        if movie_id in movies_data and genre in movies_data[movie_id]['genres']:
            selected_titles.append(movies_data[movie_id]['title'])

    if selected_titles:
        return f"Movies from the genre '{genre}' like {', '.join(selected_titles)} were recommended."
    elif considered_titles:
        return f"Movies from the genre '{genre}' like {', '.join(considered_titles)} were considered but not selected."
    else:
        return f"No movies from the genre '{genre}' were considered."
```

```python
def explain_position_absenteeism(movie_id, recommendation_info, ratings_matrix, movies_data):
    title = movies_data[movie_id]['title'] if movie_id in movies_data else f"Movie ID {movie_id}"

    if movie_id in recommendation_info['selected_movies']:
        # Ensure avg_rating is a single scalar value
        avg_ratings_list = recommendation_info['selected_movies'].get(movie_id, [])
        avg_rating = sum(avg_ratings_list) / len(avg_ratings_list) if avg_ratings_list else 0

        # Calculate the group's average rating for the movie, ensuring it is a single scalar value
        group_avg_rating = ratings_matrix.get(movie_id, pd.Series()).mean()

        # Determine the reason for not ranking the movie first
        reason = ("diversity considerations" if avg_rating < group_avg_rating
                  else "there were movies with higher average ratings or better matching the group's preferences")
        return f"'{title}' was recommended but not ranked first due to {reason}."
    else:
        return explain_atomic_case(movie_id, recommendation_info, ratings_matrix, movies_data)
```

Function explanations

generate_group_recommendations_with_info

- Generates group movie recommendations in multiple sequences
- The use of multiple sequences with a kNN model for each allows capturing varied aspects of user preferences. By considering unrated movies from similar users, the recommendations are likely to align with the users' interests while maintaining diversity. Tracking both considered and selected movies enables detailed explanations for the recommendation logic.

explain_atomic_case

- Provides an explanation for why a specific movie (atomic case) was or was not recommended to the user group.
- Crucial for understanding the reasons behind the inclusion or exclusion of individual movies in the recommendation list.

explain_group_case

- Explains why movies of a particular genre were or were not recommended to the user group

explain_position_absenteeism

- Explains why a specific movie was not ranked first in the recommendation lis

Conclusion

We can see that the method outputs top10 recommend movies in three sequences for the user group to watch together.

Then there are atomic case, group case, and position absenteeism explained for example movies.