BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

TripTales: A multifactor hybrid recommender system for travel destinations

Supervisor
Lect. Ph.D. Pop Andreea Diana

Author
Munteanu Tudor-Constantin

2023

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENŢĂ

TripTales: Un sistem hibrid de recomandare multifactorial pentru destinații de călătorie

Conducător științific
Lect. Ph.D. Pop Andreea Diana

Absolvent
Munteanu Tudor-Constantin

2023

ABSTRACT

The recommendation problem refers to the challenge of providing personalized suggestions or advice to users in various domains, such as movies, music, books, products, or in this case, travel destinations. The objective of this thesis is to develop a web application similar to a social media one, where people interact with each other and post about their travel adventures. This is the basis of the integrated recommender system that predicts the top three new places that a user would like to visit. The proposed method uses a hybrid approach that involves user-based collaborative filtering, content-based filtering, and implicit feedback. It aims and succeeds to handle the cold start problem and to ensure diversity, leading to promising results. They may leave room for further experiments with various information to be taken into account that describes the travel destinations when making recommendations.

Contents

1 Introduction **1**

1.1 Subject overview 1

1.2 Thesis overview 2

1.3 Original contribution 2

2 Literature review **4**

2.1 Overview of Recommender Systems 4

2.2 Collaborative Filtering 4

2.2.1 Memory-based techniques 5

2.2.2 Model-based techniques 7

2.2.3 Content-based filtering 8

2.3 Netflix recommendation algorithms 8

3 Proposed recommender system **14**

3.1 Dataset 14

3.2 The recommendation algorithm 16

3.2.1 The implementation 16

3.2.2 Adding new places to dataset 19

3.2.3 K-means 20

4 Methodology: The Web Application **21**

4.1 Architecture 21

4.2 Frameworks and tools 23

4.2.1 Server 23

4.2.2 Database 24

4.2.3 Caching 25

4.2.4 Frontend 26

4.3 Design patterns 27

4.3.1 Cached singleton 27

4.3.2 Provider 27

4.3.3 Builder 28

4.4 Functionalities and implementation 29
4.4.1 Features 30

5 Conclusion **37**
5.1 Comparison with SOTA. Results 37
5.2 SWOT analysis 38
5.3 Future improvements 39

Bibliography **41**

Chapter 1

Introduction

1.1 Subject overview

Travel planning can be a challenging task, especially for individuals who are unfamiliar with some particular destinations, undecided, or simply do not have knowledge about traveling. There are many of us who wish to travel more and explore new places, but it is difficult due to the increasing availability of online data that is hard to be filtered in a timely and efficient manner.

Recommender systems have emerged as a promising approach to assist travelers in making informed decisions about where to go and what to do. Despite that, the recommendation problem still exists. For an accurate recommender system, there are many metrics and approaches that have to be included in experiments with respect to the problem. They use machine learning algorithms to analyze user data and provide personalized recommendations. Generally, they are based on the user's behavior, interactions, tendencies, popularity, and other factors, like geographical location, time, and so on. In this way, information is selected and presented to people in an appealing manner.

For example, if one wants to watch a movie, it would be easier with a Netflix or HBO Max account, because the content would be presented and recommended according to his profile. It is the same when it comes to buying products; Amazon, for example, uses recommendation systems and has a big part of its revenue coming from that. These examples can be regarded as state-of-the-art in this field, but their aim is to keep the users engaged within their application in order to increase their incomes.

Regarding traveling, people use travel agencies because they are exempt from all the searching work, and it is easier for them to make decisions from a variety of given options. Searching for everything by themselves and finding the right thing can be tiring and time-consuming because in order to draw conclusions about a

place, one has a lot to take into consideration. However, the services offered by these travel agencies are often expensive.

1.2 Thesis overview

This thesis focuses on developing a recommender system for travel planning. The project is a web application, named TripTales, that allows users to share their travel experiences and provide recommendations to them. They can interact like in a social media app, by liking, saving, and viewing posts. The system uses a hybrid approach to suggest personalized travel destinations based on user preferences.

Another key feature is the itinerary recommendation. If a user wants to go to a place for a defined number of days, he can get a full itinerary for each day, which also includes 3 accommodation options. The itinerary consists of recommended activities to do when visiting that place, as well as food, restaurants, and museums. Therefore, any kind of specific activity that is known to be popular in the visited place will be included.

The main objective of the thesis is to optimize and evaluate the recommendation system for the purpose of travel planning. The thesis is organized as follows: the second chapter provides a review of the literature on recommendation systems, including an overview of the different techniques that are being used, evaluation metrics, and applications that integrate them. Chapter 3 outlines the methodology used to develop the recommender system, including data collection and algorithm selection. The fourth chapter describes the implementation of the web application, including the user interface, the other available features, and the integration of the recommender system and the itinerary generator. Finally, Chapter 5 presents the results and analysis of the system's performance, including future work, limitations, and conclusions.

1.3 Original contribution

After more experiments on my randomly generated dataset, more contributions were achieved:

- As hybrid recommendation systems can incorporate diverse approaches in authentic and creative ways, TripTales uses content-based filtering, user-based collaborative filtering, and implicit feedback from the users.
- The presented hybrid approach proposes an original idea because of the way it combines a variety of features regarding travel destinations and posts. The fact

that users indirectly interact with places through posts is unique. Therefore, all the posts about a certain place aggregate the data that will precisely describe it from the user's perspective. Also, a user's rating regarding a place is built in a unique manner, because of how it weighs the ratings and bonuses which were fine-tuned to study the impact on the outcome, as explained in Subsection 3.2.1.

- During the research, I did not find a real dataset that would satisfy my needs, meaning users that rated travel destinations or places. That led to building my own randomly generated dataset with realistic data that is discussed in detail in Subsection 3.1.
- The places within the application are clustered based on their geographic location using the K-means algorithm. Therefore, the actual distribution of the data is taken into account, not some predefined fixed geographic limits. The number of clusters increases as the data grows and the centroids will not remain fixed.
- The cold start problem was overcome by recommending the top three most popular places that the user did not visit. This is done through an original popularity score that is computed as a weighted sum, using the place's number of likes and view count.

Chapter 2

Literature review

This chapter presents the most popular and effective recommendation systems, in order to introduce the reader to this field. There are many factors and metrics to be taken into account while implementing, but I decided to settle on the ones that are the most important and relevant to the present. The concrete results will be analyzed, and a closer look will also be taken at the details.

2.1 Overview of Recommender Systems

Recommender systems have become increasingly popular in recent years due to the explosion of online data and the need to provide personalized content. They are designed to suggest items of interest to users based on, for example, their past behavior, preferences, or similarities to the other users. There are many companies that built their own recommendation engines. One example would be Amazon, which generates increases in sales yearly this way. Also, Netflix uses such a system in order to increase the viewers' activity and keep them happy regarding the watched content.

There are several types of recommender systems, but the main ones would be those that use collaborative filtering, content-based filtering, or hybrid approaches. Of course, there are many other diverse and interesting types, so we will explore them further in the next sections.

2.2 Collaborative Filtering

This type of content filtering is a technique used by recommendation systems to make predictions about user preferences. A research paper from 2018, published at Vrije Universiteit Amsterdam [Mol18], also explains some key concepts related to it. Basically, the idea is that people who have agreed in the past, regarding their

preferences, behaviors, or tendencies, tend to agree in the future. A basic example would be if a user likes a movie and it is similar to another user, this would be a sign that the second one would also enjoy that movie. This kind of approach assumes building the user-item matrix V of dimensions n x m, where u_i with $i = 1, 2, \ldots, n$ is the set of users, and p_j with $j = 1, 2, \ldots, m$ is the set of products, meaning the items. Thus, in rows, there are users and in columns the items, so v_{ij} is the rating given by the users to the items. If there are no interactions between a user and an item, then $v_{ij} = 0$.

$$V = \begin{array}{c} \begin{matrix} p_1 & p_2 & & p_j & & p_m \end{matrix} \\ \begin{bmatrix} v_{11} & v_{12} & \ldots & v_{1j} & \ldots & v_{1m} \\ v_{21} & & & & & \\ & & \ddots & & & \\ \vdots & & & v_{ij} & & \vdots \\ & & & & \ddots & \\ v_{n1} & & \ldots & & & v_{nm} \end{bmatrix} \end{array} \begin{matrix} u_1 \\ u_2 \\ \\ u_i \\ \\ u_n \end{matrix}$$

Figure 2.1: User-item matrix

As collaborative filtering is based on information about similar users or objects, it can be divided into memory and model-based techniques.

2.2.1 Memory-based techniques

The two main approaches are user- and item-based. The first finds similar users based on their past interactions and recommends items that they have liked or rated highly. With the second approach, items are compared instead of users, and their similarities are calculated based on user interactions. If a user has liked or interacted with a particular item, he/she is likely to be interested in similar items, so that is what it is going to be recommended.

User-based Collaborative filtering

The idea is that the users most similar to the target must be found, taking into account the ratings of the same items. Therefore, their top-rated items that are new to the target user will be suggested. To compute the similarity metric between the users, there are many methods that can be applied, such as Cosine Similarity or Pearson Correlation.

The Cosine Similarity takes two vectors, representing the user's ratings regarding items [BHK98]. It is calculated by taking the dot product of them over the product of their norms. Measures the cosine of the angle between the two vectors, which

indicates the similarity of their directions. A result of 1 shows that the vectors are identical, while a value of 0 indicates no similarity.

$$\cos(u_i, u_k) = \frac{\sum_{j=1}^{m} v_{ij} v_{kj}}{\sqrt{\sum_{j=1}^{m} v_{ij}^2 \sum_{j=1}^{m} v_{kj}^2}} \tag{2.1}$$

Pearson correlation quantifies the strength and direction of the linear association between two data point sets [MMN02]. The resulting coefficient can be a value ranging from -1 to +1. -1 represents the perfect negative linear relationship, +1 the perfect positive linear relationship, and 0 means no linear relationship.

$$S(i,k) = \frac{\sum_{j=1}^{m} (v_{ij} - \bar{v}_i)(v_{kj} - \bar{v}_k)}{\sqrt{\sum_{j=1}^{m} (v_{ij} - \bar{v}_i)^2 \sum_{j=1}^{m} (v_{kj} - \bar{v}_k)^2}} \tag{2.2}$$

Here, $S(i,k)$ is the similarity between u_i and u_k users, v_{ij} is the rating of u_i to the item p_j, $\bar{v}_i$ is the mean rating of the same u_i, and m is the number of items.

Using one of these methods, the similarity score is calculated between all users. Based on that, the most similar k users to u_i are identified, and the items they rated the highest are selected, except those already rated u_i. This is weighed using similarities, and the values are added. This provides predictions of the rate the target user would give to each item. Finally, the top k items are selected on the basis of the predicted ratings.

Item-based Collaborative filtering

If the previous algorithm is based on the similarities of users, to make recommendations, this one is the opposite, trying to determine how similar are two items.

The similarity of the items p_i and p_j is calculated using the users who already rated them and then calculating the cosine similarity or Pearson correlation, as before. For the first approach, the difference in the rating scale between users is ignored, and for the second approach, the users who rated both p_i and p_j must be isolated. The formulas remain the same; the only difference is that $\bar{v}_i$ means the average rating of the j-th film. Finally, the item similarity matrix is built, and the items that the user has previously rated are selected in order to choose the closest ones to them. They are weighed to obtain the predicted rating. Based on that, recommendations are made.

2.2.2 Model-based techniques

Memory-based techniques are not so fast and scalable, especially in a real system that generates recommendations in real time based on a large dataset. To achieve these objectives, model-based recommendation systems are used. Ratings are used to build a dataset and implement a model that will extract information from the data. Machine learning algorithms can be applied to construct models based on matrix factorization, an unsupervised method of machine learning to reduce dimensionality. In essence, it learns the user and item's latent preferences from the ratings and uses the dot product of them to predict the missing ratings. One of the techniques that might be applied is Singular Value Decomposition (SVD), one of the most effective, and thus it will be further discussed in detail.

SVD

This is a matrix factorization technique that uses a matrix A of rank r and m x n dimensions with the aim of decomposing it into three component matrices by this formula:

$$\text{SVD}(A) = U \cdot S \cdot V^T \tag{2.3}$$

U and V are orthogonal matrices of dimensions m x n respectively n x m, while S represents the singular matrix [GPB13]. This represents the decomposition of the original matrix into linearly independent vectors. The first r columns of U and V are eigenvectors of $A \cdot A^T$ and $A^T \cdot A$, respectively, representing singular vectors left and right of A. The best low-rank approximation of A is achieved by keeping the first k diagonal values of S and removing r - k columns from U and r - k rows from V:

$$A_k = U_k \cdot S_k \cdot V_k^T \tag{2.4}$$

The actual dimensions of U, S, and V will be m x k, k x k, and k x n respectively.

SVD can be applied to the user-item rating matrix in recommender systems. It assumes that the matrix contains latent structures which can be captured by transforming them into lower dimensions. It relates to all the users, or to the majority of them. The transformed matrix represents users and items in a k-dimensional space. The matrix product $U_k \cdot \sqrt{S_k^T}$ stands for the M pseudo users, while $\sqrt{S_k \cdot V_k^T}$ represents N pseudo items in the k-dimensional space. Each element of the first matrix represents a feature of the corresponding item. As an example, in the movie domain, it may show if a movie is a comedy or not. Similarly, the second matrix indicates whether or not a user likes these features. The prediction of the u-th user rating on

the i-th item can be computed as follows:

$$\hat{r}_{i,u} = U_k \cdot p \cdot (S_k^T(u) \cdot p) \cdot (S_k \cdot V_k^T(i)) \tag{2.5}$$

2.2.3 Content-based filtering

This approach is intended to recommend items alike to those previously liked by users. The difference between this and Collaborative Filtering is that its recommendations are not based only on rating similarity, but also on the information from the items. For movies, it may be the title, the year, or the actors. Information that describes each item and a sort of user profile that sketches what the user likes is necessary. The goal is to learn the user's preferences and recommend items similar to them. The data must be structured, each element being described as a vector based on the same set of attributes. The central part is creating the user's preferences model based on these vectors.

Content-based filtering solves a few of the problems of collaborative filtering, such as "cold start" because the system can recommend even if the user did not evaluate any element. The popularity bias problem is also solved by recommending items with rare features so that users with unique tastes can receive effective recommendations. This technique also has its own drawbacks; the implementation depends on the item metadata, which demands a rich description of the items. Therefore, it leads to "limited content analysis", another problem that users may be limited in their ability to explore new items.

2.3 Netflix recommendation algorithms

Due to its robustness, business purpose, and value, the Netflix recommender system may be considered a state-of-the-art in this field. In 2015, Netflix's CPO and VP of Product Innovation published an article [GUH15] in which they discussed the approach and motivation behind the recommendation algorithm, but also what makes it global and language conscious. They put in words the fact that Netflix lies where the Internet and storytelling intersect, thus being the inventors of Internet television.

The recommendation problem is also discussed. Internet TV is different from traditional broadcast, making it all about the user's choice: what to watch, when, and also where. Even if this is a massive improvement, people may feel overwhelmed due to a large number of options, leading to loss of interest and poor choices.

As time passed, the recommendation algorithm became a collection of algorithms, and the star rating became history. Nowadays there is a vast amount of

data that is considered to define Netflix user behavior and preferences such as the device, time of the day, day of the week, the intensity of watching, and the place in the product in which each video was discovered. All these come together on the Netflix homepage, as seen in Figure 2.2, which is the most important, being the first one seen by any user.

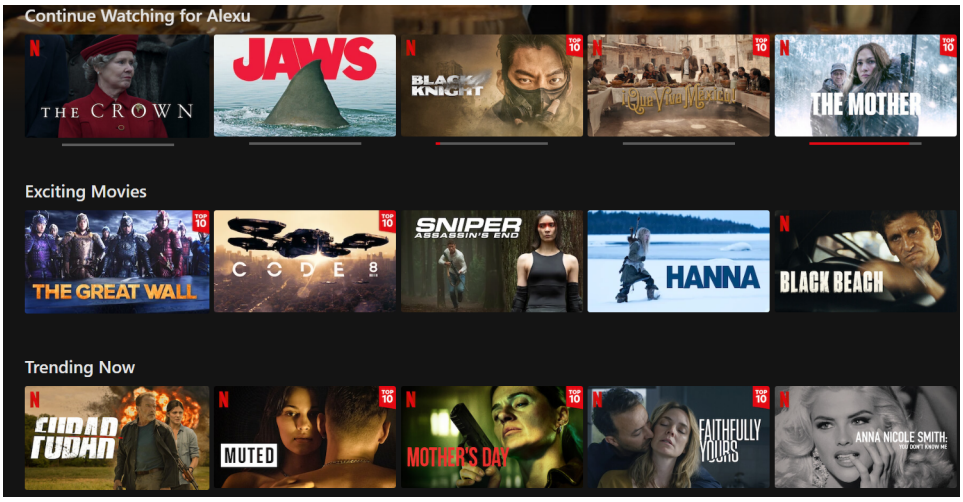

Figure 2.2: Netflix homepage

Usually, there are about 40 rows on the homepage, where the videos on each row are typically from one algorithm. For example, it may be:

- **Personalized Video Ranker.** It sorts the collection of videos (filtered by genre or other criteria) for each user in a custom manner with respect to its profile, adding at the same time some unpersonalized content to provide diversity too.
- **Top-N Video Ranker.** It is used for the top picks row and aims to find the best personalized recommendations in the entire catalog.
- **Trending Ranker.** This algorithm works better when it comes to identifying trends that may repeat yearly, such as watching movies similar to Home Alone during the Christmas period in North America.
- **The video-video similarity.** Because you watched rows, use this. It builds a ranked list of similar videos for every video in the system. The ranking is not personalized, but the picks for the rows are personalized.
- **Evidence selection.** Every piece of evidence that can be shown for each recommendation is evaluated by this algorithm. This can be something like: the movie won an Oscar, or it is similar to something that the user watched.

- **Search algorithms.** Most of the time, users search for genres, actors, and titles, which is fine when the system contains exact data on search queries. This may again become a recommendation problem when general concepts or similar data are searched as in the previous example, which is not part of the Netflix catalog. An example may be to search for "free", also illustrated in Figure 2.3. This would put to work several algorithms, and one of them would retrieve the film "Friendzone", another one matches the concept of "French movies", and the third one recommends movies for this.

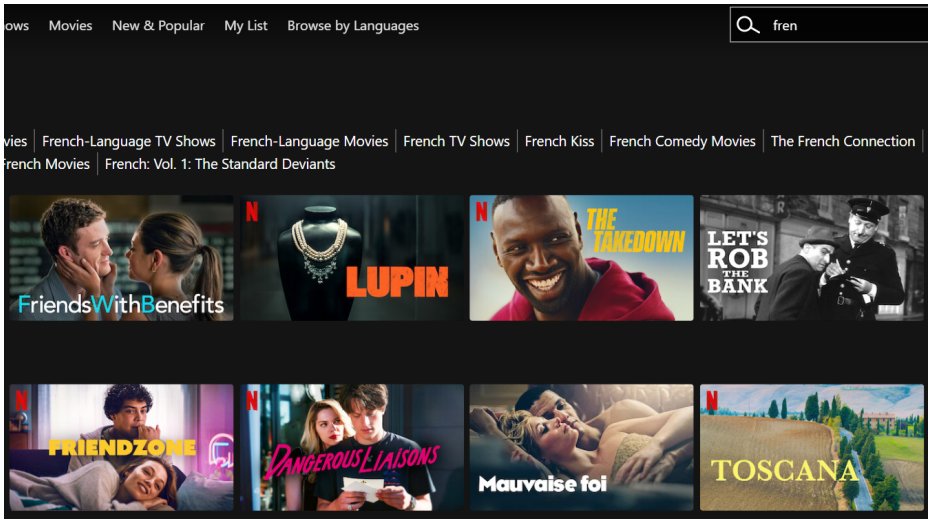

Figure 2.3: Netflix search recommendation page

Thus, almost everything at Netflix is based on recommendations. Hence, the scalability is enabled and the core of the app is defined by that because here recommendation is similar to business value. The user will always be engaged within the application and will also be pleased with the content served, preventing them from abandoning the service for another one. Without personalization, all users would get the same videos recommended, which is not fair because of the differences between people as human beings.

In order to improve their algorithms, Netflix uses A/B testing, a user experience research method. For example, on a website, other users may get a different version of the same component to determine which is better. In the previously cited article, it is described how Netflix implements this. Because the desire is to listen to customers and differentiate good recommendations from great ones, they choose new users and existing users to conduct these tests. The advantage of the old members is that they have previous experience with the application, a fact that helps measure the impact on the immediate change of the product. The new ones are also preferred due to their lack of interaction with other versions of the application; thus, they would indicate the efficiency of the new version, rather than the old to

a new change. An example would be these video sets presented in Figure 2.4, for the "House of Cards" search. The bottom ones seem more relevant; they include the original version of the movie, but turn out to be worse because the others have a stronger popularity influence.

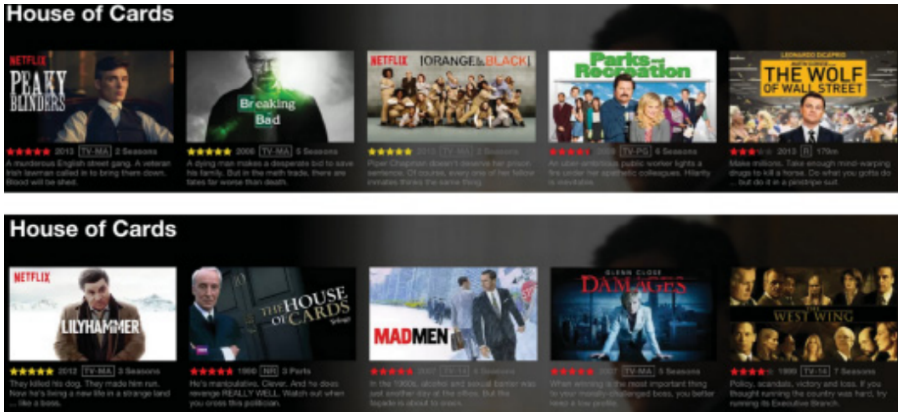

Figure 2.4: Similar sets of videos for "House of Cards" used at A/B testing [GUH15]

Another article [SBE+21], again published by a Netflix research team in 2021, aims to introduce deep learning into their various recommendation tasks. This created meaningful improvements in user retention, compared to Gomez-Uribe and Hunt from 2015 who outlined their techniques from the before deep learning period. The modeling approaches were split into the bag of items approaches and sequential models, as Figure 2.5 presents, which excel at mixing heterogeneous features.

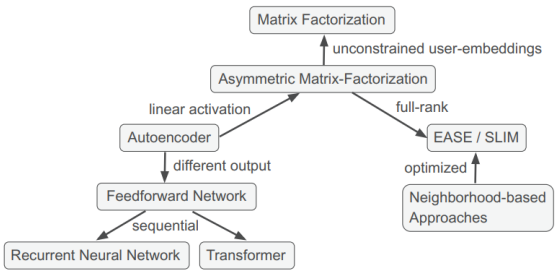

Figure 2.5: Simplified relationships among models [SBE+21]

Autoencoding consists of encoders and decoders which are made up of a variety of hidden layers with non-linear activation functions. The input vector is converted by the encoder to a low-dimensional embedding and then decoded to an output vector. The goal is to minimize reconstruction errors by creating a relatively close output vector to the input vector. A user's play history encoding serves as an input and output vector for the recommendation task. If the user has watched the video,

each member of the vector can take the value 1; otherwise, it can take the value 0. Continuous-valued vectors can also be used to record how long a user viewed a video.

- Bag of items approaches:
  - The factorization of an asymmetric matrix is similar to a linear autoencoder with a single hidden layer. The video embedding matrix of the encoder and decoder is learned, and the user embedding is limited to being the average embedding of those matrices for the already seen videos. Recommendations for a new user or updates to new data would not include an optimization procedure, as required by standard matrix factorization. The use of asymmetric matrix factorization allows the calculation of recommendations by passing the input vector through the encoder and the decoder.
  - Neighborhood-based approaches are also a specific example of autoencoders because they are based on an item-to-item function. A full-rank model can be generated by increasing the dimension of the hidden layer in a linear autoencoder until it equals the number of elements in input and output vectors. With regard to model parameters, it is identical to a full-rank model with a single item-matrix. This is equivalent to the matrix used to measure item-item similarity in neighborhood-based methods. The perspective of neighboring techniques such as autoencoding offers a rational way to optimize such a matrix, although it is often created with heuristics such as cosine similarity.
- Sequential models:
  - The sequential nature of the user's play history contains important information lost in the bag-of-items model. Many sequential models created for NLP tasks can be modified to be used for recommending. The recommendation model is designed to forecast the subsequent item that the user will engage with, rather than the next word in a phrase. The sequence of items that users visit during their sessions is the only information known about them. Repeated neural networks (e.g. long-term memory [HS97]) and transformer models (e.g. BERT [DCLT18]) are some of the sequential models that have been tested in recent years. Despite not being strictly speaking sequential models, the transformers' attention mechanism has a comparable impact. The attention mechanism offers a novel and intriguing method to find an explanation for each recom-

mended film, in addition to increasing the accuracy of the recommendation.

Deep learning models are effective at utilizing additional information beyond user feedback, so integrating time as a unique heterogeneous feature demonstrates their power. One of their models, using raw continuous timestamps to record when a user played a video and the current time when making a prediction, showed improvement. This approach yielded a 30% increase in offline metrics compared to discretized time. In this context, offline metrics are used to evaluate the performance of the system based on historical data without direct interaction with users. These metrics may be Root Mean Squared Error, or Precision and Recall for example. However, deep learning models are less explainable, and there is an open question of how far into the future these methods can predict and the speed of performance degradation without retraining.

Chapter 3

Proposed recommender system

In this chapter, the focus will be on the implementation and theoretical part of my recommendation algorithm. We will see in-depth how my hybrid approach to recommending travel places works, the results, and how I achieved them.

3.1 Dataset

My implementation is a hybrid one and uses a combined approach that contains some components:

- User-based collaborative filtering
- Content-based filtering
- Implicit feedback

Due to this, the system may encounter an initial problem, cold start. This comes from the lack of data related to users and items, and also from the lack of interactions between them. This may appear for the new users or for the system bootstrapping case when the system is new and there is almost no information that can be relied on. This matter will be discussed later in detail, and now I will present the way I have built the data set in order to get some true and accurate recommendations based on the system's capabilities and not on a fallback solution.

Having the model and the database representation, which is explained in Subsection 4.2.2, it comes to the seeding part that I will explain.

- Users:
  - With the help of faker-js, a node package capable of generating massive amounts of random, fake but realistic data, I've added to the database 500 users. I also stored 25 people's portrait photos and randomly assigned one to each of them.

- Places and Clusters:
  - I seeded 250 places, using 25 centers. I have built a list with those places chosen from all the continents, including also Antarctica and Australia, taking into account to be popular travel destinations too. Having the address for each place, I made use of Google's Geocoding API [MKL19] to translate them into coordinates. The next and final step was to use K-means (algorithm described in Subsection 3.2.3) to cluster these places, taking into account their distribution around the initial centroids, which are displayed in Figure 3.1. In addition, the clusters will be recomputed once a day, because places are added (they are never deleted), having an impact on the recommender system. This determines chaining changes: the change of the centroids that causes differences in recommendations.

Figure 3.1: Clusters map

- Likes and Saved posts:
  - In order to provide user interaction and diversity, I provided each user with a number of 400 likes and 20 saved posts. Both are for random posts that are not created by the user that performs the action.
- Posts:
  - To seed the posts, I chose to create 50 of them per user, meaning a total of 15000. Each post gets a random rating on a scale of 1 to 10, and a random number of users who viewed it. The view of a user is considered

only once because, in this way, content that reaches a wider audience is prioritized. For the rating, I used a weight-based function to assign the value with a higher likelihood between 4 and 8, a medium likelihood between 9 and 10, and the lowest possibility between 1 and 3. Thus, a bell curve was achieved, also emphasized in Figure 3.2, which represents that the set of chosen values tends to have central, normal values, which is more realistic. When seeding, it is also taken into account that the user will have posts about random and different places. Storing 15000 photos for the beginning would be too much, taking into account that it is only dummy data that tests the recommender system. Regarding this, I have used the same approach as that for the users.

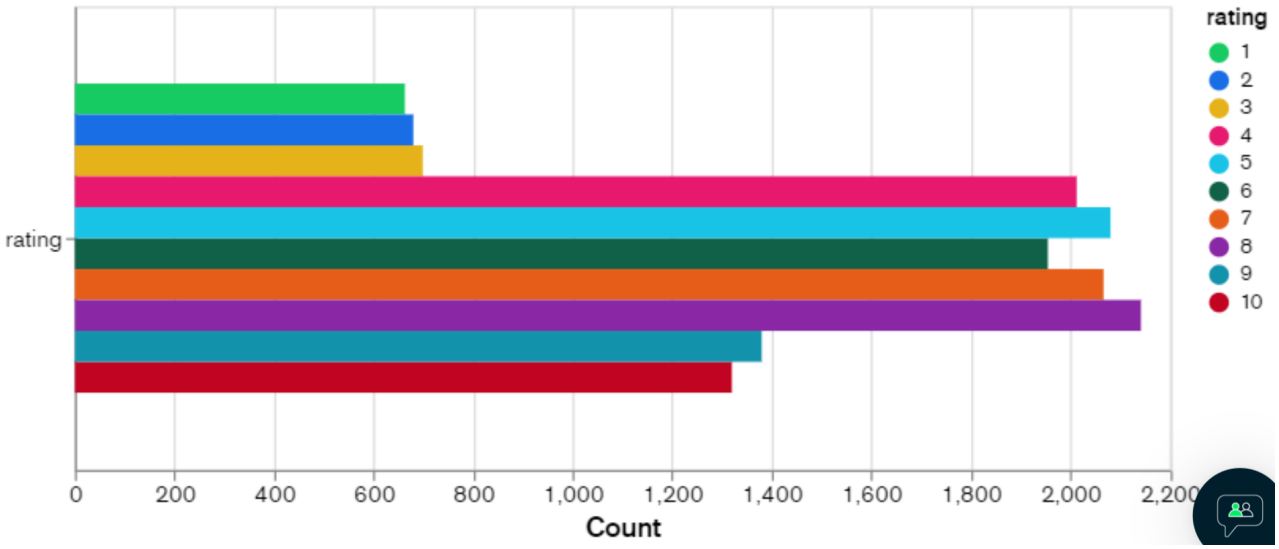

Figure 3.2: Ratings distribution

3.2 The recommendation algorithm

3.2.1 The implementation

First of all, my algorithm's approach is to prepare and create the user-item matrix, based on the places that come from the posts. The goal is to construct a matrix that contains the score of each user for each place, a score that is composed using more weights. What the algorithm will do further is predict the missing values of the matrix (the zeros), based on the similarities between the users, and recommend the places with the highest predicted rating that were not already visited.

In order to accomplish that, I extract the clusters that were visited by the user, then group the posts by the place where they are. As the next step, each place has

aggregated the ratings with the users who gave them, likes, views, or saved them. Having this data, I use the following formula to compute the place score:

$$Min(10, (rating + likeBonus + clusterBonus + savedPostBonus) * viewWeight) \quad (3.1)$$

Min is used because the rating is on a 1 to 10 scale and the computed score may exceed it. The likeBonus is 1 if the user liked a post containing the place, and the clusterBonus is 0.7 if the user visited any place in the same cluster. SavePostBonus is 1.35 if the user has any saved posts with that place, and the rating represents the mean of the ratings given by the user to his posts about that place. Regarding the viewWeight, I use this formula:

$$1 + log10(viewCount)/scalingFactor \quad (3.2)$$

The scaling factor is 30 and if the viewCount is 0, the viewWeight will be 1. I chose this function and scaling factor because with a larger number of views, the harder the value of the function grows. This is desired because, in this way, the less viewed places will not be discriminated against and not recommended instead of the most viewed ones.

These hyper-parameters were chosen as a result of experimenting and fine-tuning, in order to optimize the performance of the recommender system with respect to the dataset. They are designed to be smaller than the central value of the rating because are less definitory, and they also highlight that a user may be more interested in a place if one saves posts about it, rather than liking them. Less important than both previous factors would be that a user may be interested in places from a certain geographical area, so that is why the cluster bonus is used. On the other hand, someone may want to discover new places, but for that is the weight regarding the number of views, to recommend trending places, and to introduce diversity.

Now, that the data is well prepared, the recommendation algorithm first checks the number of interactions of the user with the places. As a backup solution, it recommends the most popular K places (K = 3) if the user interacted with fewer than 10 places. If there are enough interactions, as a next step, user-based collaborative filtering is done using cosine similarity. This metric is often used in high-dimensional positive spaces, like when comparing vectors in text analysis. A concrete example would be as follows: User1 and User2, have both rated PlaceA and PlaceB. If User1 rated PlaceA with 5 and PlaceB with 1, and User2 rated PlaceA with 4 and PlaceB with 1, their rating vectors are quite similar. Calculating the cosine similarity between these two vectors would give a high value, indicating that these two users have similar preferences. Two identical users should have a cosine similarity of 1, while two completely different users should have a score of 0 for this metric.

The top K most similar users are extracted for the person to whom the recommendations are made, and the ratings of the unvisited places are predicted based on interactions with those users. The predicted rating for each of them is a weighted sum of the ratings given by the previously extracted users, adjusted by user bias, item bias, and a randomization factor, also called noise, which can increase or decrease the rating by 0.125. Its role is to add some diversity to the recommendations.

The user bias represents the average rating of all items by the given user. It is a measure of the user's general tendency to rate items high or low. For instance, if a user rates items very high in general (an average of 9 out of 10), and another user rates items very low in general (an average of 3 out of 10), then a rating of 7 by the first user is considered low, while a rating of 7 by the second user is considered high.

The item bias is the average rating of a given item among all users. It is a measure of the general agreement among users about the quality of the item. For example, if an item tends to be rated high by all users, then it likely has high quality, while an item that tends to be rated low by all users likely has low quality.

The predicted ratings are normalized using the Z score. This is achieved by subtracting the mean and dividing it by the standard deviation. This recalibrates the ratings to have a mean of 0 and a standard deviation of 1. The idea behind this is to account for variance among user rating behaviors. Some users might tend to rate harshly, others more generously, and yet others might use the whole scale from 1-10. Normalizing these ratings allows for a more direct comparison by mitigating the influence of these rating habits. This would be the last part of the algorithm; it is only left to filter out places the user has already visited, sort them by their predicted ratings in descending order, and select the top K.

$$predictedRating = zScore[userBias + (\sum_{i=1}^{k}(r_i - b) \cdot similarity_i) / \sum_{i=1}^{k} similarity_i + noise] \tag{3.3}$$

The formula depicts the predicted rating of an item, where r_i is the rating of the i^{th} most similar user to the item, b is the bias of that item, and $similarity_i$ is the similarity score of the same similar user.

In order to exemplify how the whole algorithm works and see some concrete results, I will pick two users. One with little interaction within the app, and another one with enough interaction so that the fallback solution will not be used.

For example, the first user from my dataset, Adriel Pollich, is the most similar to the users:

- Cale Beer, similarity: 0.6406343036490246
- Dayton Wilderman, similarity: 0.6098617212230214

- Merritt Bailey, similarity: 0.6009505958956299

She got as recommendations the following places:

- Las Vegas, USA, predicted rating: 9.802449427949695
- New York City, USA, predicted rating: 9.45679722008553
- Nuuk, Greenland, predicted rating: 5.597389987350885

The results obtained indicate a moderate to high level of similarity between the target user and the top K similar ones. Their interests and behaviors align fairly well. Most of the users in the dataset have a maximum similarity score of around 0.55-0.65, so, with respect to that, these can be considered very high.

The other case would be when the recommendation algorithm is run for a user with not enough interactions within the application, which leads to recommending the most popular places that were not already visited. It means taking into account the view count and the likes of the posts. Using weights, it is emphasized that a like is 1.5 times more important than a view. The results for the initial dataset would be:

- Riga, Latvia, popularity score: 22710 (21192 views and 1012 likes)
- Vienna, Austria, popularity score: 21421.5 (19821 views and 1067 likes)
- Brazzaville, Congo, popularity score: 21379.5 (19701 views and 1119 likes)

3.2.2 Adding new places to dataset

When adding a new place, it should be taken into account that it must belong to a cluster. For the moment, it is added to the closest centroid that is found using the Haversine formula, which determines the straight line distance between two points on a sphere given their longitudes and latitudes:

$$a = \sin^2\left(\frac{\Delta\varphi}{2}\right) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right) \tag{3.4}$$

$$c = 2 \cdot \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right) \tag{3.5}$$

$$d = R \cdot c \tag{3.6}$$

Here, φ_1, φ_2 are the latitudes of the first point and second point in radians, $\Delta\varphi$ is their difference, $\Delta\lambda$ is the difference of their longitudes, R is the radius of the Earth, and d is the distance between the two points.

At 1 AM UTC (out of business hours), a cron job runs that recomputes the clusters. As the number of places grows, so will the number of clusters, the job being

also responsible to keep them at the value of 10% of the places. I chose this approach because it is time-consuming and doing this every time a place is added would not be the best user experience due to the long wait.

3.2.3 K-means

This unsupervised machine learning algorithm is regarded as one of the most powerful and popular because of its clustering capabilities. As also presented in the 2012 published article at the Xinyang Agriculture College, by the computer science department [LW12], it is an iterative algorithm, used to partition a dataset into K distinct non-overlapping clusters, with the goal of minimizing the sum of squared distances between the data points and their assigned cluster centroids. It works in the following manner:

- **Initialization.** Select K random points as the initial centroids. This is a parameter of the algorithm and is the number of clusters in which you want to classify your data.
- **Assignment.** Each data point is assigned to the nearest centroid. This is usually done using a distance measure such as the Euclidean distance. After this step, the initial set of clusters is obtained.
- **Update.** For each previously formed cluster, the new centroid is calculated as the mean of all its data points.
- **Iteration.** Assignment and update steps are repeated until the centroids do not change significantly, or a certain number of iterations have been reached.

Chapter 4

Methodology: The Web Application

This section is where the focus will be on the built web application, TripTales, which incorporates my recommendation algorithm and all the other features, too. The architecture, implementation details, and used technologies will be discussed, including the motivation for using them.

4.1 Architecture

To build the application from scratch, a strategy must be established first. For that, software architecture comes in handy because it defines the structure of the system while describing how the aspects that make it work and behave should be incorporated.

Regarding this aspect, due to the fact that the application follows the client-server model, I chose to use an N-tier architecture, as represented in Figure 4.1. This provides a modular and scalable structure because it enforces the separation of concerns, allowing better reusability and maintainability. The goal is to divide the software system logically and physically into distinct and independent layers, where each of them focuses on a specific concern. Physical separation means that each layer can be deployed independently, so they can all run on different machines. There are many advantages of this, load balancing can be enabled, and also the performance is optimized because you can allocate dedicated resources for specific requirements. The tolerance to fault is reduced, and the overall availability of the system is enhanced. If one of the layers fails, all the others will be up and running, which is different from a monolith, and it will also be easier to identify the source problem. With this approach, you can change things in one layer without affecting the others. Thus, the code has a better organization, which makes development easier.

For example, in my application, I chose to separate everything into four tiers:

- **Presentation.** This is represented by the frontend application, responsible for rendering the user interface and handling the user interactions. I chose to use React and deploy it as a single-page standalone application on Firebase.
- **Application.** It contains the business logic and enforces its rules. The tier is also responsible for data manipulation, algorithm implementation, and computations. It is implemented using Node.js along with Express.js. For deployment, the option was Azure, as a standalone REST API.
- **Caching.** This is an optional layer because the traditional N-tier architecture has only three layers in most cases. In the current example, it is used as a buffer between the application and the data source. To perform data caching, the application uses Redis, which is hosted on AWS.
- **Data.** This tier has to manage the interaction with the database. It handles tasks such as data retrieval, storage, update, and deletion. For the database, the option was a NoSQL one, MongoDB, due to the ease of using it together with Node. It contains only a cluster, also hosted on AWS.

Finally, on top of this N-tier architecture, the combination of the backend, controllers that handle Mongoose models, and the frontend with its React components that serve as views, both as a whole, can be seen as an MVC design pattern. Moreover, the backend itself follows the MVC principles. It separates the concerns of data models, business logic (controllers), and views (JSON responses to the client).

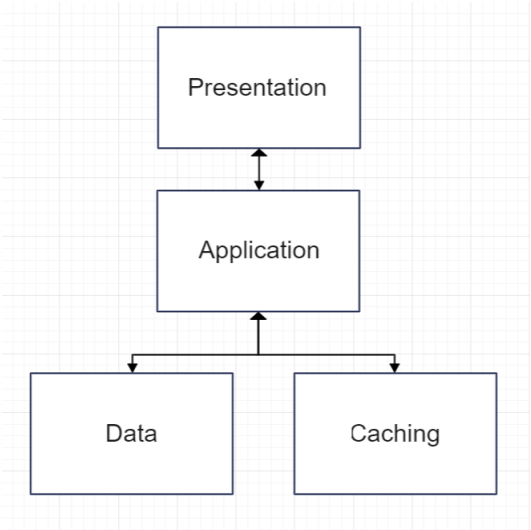

Figure 4.1: 4-tier architecture

Even if more technologies were already mentioned, the next section will be specially dedicated to them. They will be analyzed in detail.

4.2 Frameworks and tools

Now that the architecture is well-defined and the chosen technology stack has been mentioned, the focus will be on the implementation of each layer.

4.2.1 Server

To build a web application that incorporates a variety of features, the application layer is a must. To create it, I have chosen Node.js together with Express.js, a powerful and popular web application framework that works on top of Node. In the following sections, we will take a closer look at them while explaining what I made use of during the development.

Express.js

Even if it is a lightweight solution, it provides a solid foundation for building APIs. Therefore, it simplifies the development process, allowing you to focus on the application logic rather than the low-level details. Express also contains a flexible set of features that come in handy when creating robust and scalable server-side applications. Some key features that helped me would be:

- **Middleware.** They are functions that have access to the request and response objects and can perform tasks in the middle of the request-response cycle, being used to process in a modular and reusable way. Middleware functions can perform tasks such as authentication, request parsing, logging, error handling, and more. Therefore, multiple middlewares can be composed into a single middleware chain using ”app.use()”. This allows you to group related middlewares together and apply them to specific routes or globally to the entire application.

- **Routing.** It provides a simple and intuitive way to define routes and handle HTTP methods such as GET, DELETE, POST, or PUT, with the help of Express router. Multiple routes with different URL patterns can be defined and associated with specific handler functions to execute. When a request hits the server, the router’s configured middlewares are passed through one by one to the one that matches the URL.

- **Error handling.** It has built-in error-handling mechanisms, including custom error middleware functions. These functions can be used to catch and process errors that occur during the request processing, in a centralized manner, allowing you to handle them gracefully and return appropriate error responses to clients.

- **Static file serving.** Express allows you to serve static files, such as images, or other assets, directly from a specified directory.

Node.js

Node.js is an open-source server-side environment that allows Javascript to run outside the browser. It has a lot of helpful features, like being cross-platform or being designed around an event-driven, non-blocking I/O model. It is well suited for handling concurrent operations, and uses callbacks, Promises, or async/await syntax to handle asynchronous operations efficiently. NPM is also a great aspect, it is a package manager that provides access to a vast ecosystem of open-source libraries while making it easy to install, manage, or share code packages.

In order to achieve the implementation goals, I used many node packages that help me extend the functionalities. For example, I used Multer in order to implement my own middleware that handles file upload, or body parser that extracts the request's body as a JSON. I used Mongoose, an ODM that provides a higher level of abstraction over MongoDB, to simplify the interaction and to be able to build schema, models, and relationships between data. Other examples would be Axios, to make requests to Google APIs, or Nodemon for development, because it automatically restarts the application when file changes are detected.

4.2.2 Database

For this layer, the application uses a Database-as-a-Service, MongoDB Atlas. This is a fully managed cloud database service provided by MongoDB that allows users to easily deploy, manage, and scale this type of database in the cloud without the need for infrastructure management. The choice was AWS, as already mentioned in the architecture section. Because it is a NoSQL database running in the cloud, scalability, and elasticity are then ensured.

Atlas also offers robust security to protect your data, such as encryption at rest and in transit, role-based access control, and network access control. High availability is also ensured by automatically replicating the data across multiple nodes within a cluster. In the event of node failure or disruption, the database system can automatically redirect requests to available nodes, allowing uninterrupted access to the data.

Another advantage of using Atlas is the variety of tools and services that integrate with it. Except for Mongoose, which was an obvious reason for choosing this stack of technologies, it nicely integrates MongoDB Compass for visual database management, and MongoDB Charts for data visualization and dashboarding.

In TripTales, the collections that I use to store the entities required by the recommendation system are the following: User, Post, Place, and Cluster. In the database diagram, which is represented in Figure 4.2, the following relationships can be observed:

- The user has an array of saved posts and another one of created posts.
- The post contains a user as its creator, an array of likes that has all the users who liked it, and also a place that the post is about.
- Each place contains a cluster assigned on the basis of its geographic location.
- The cluster contains a list of places that are part of it.

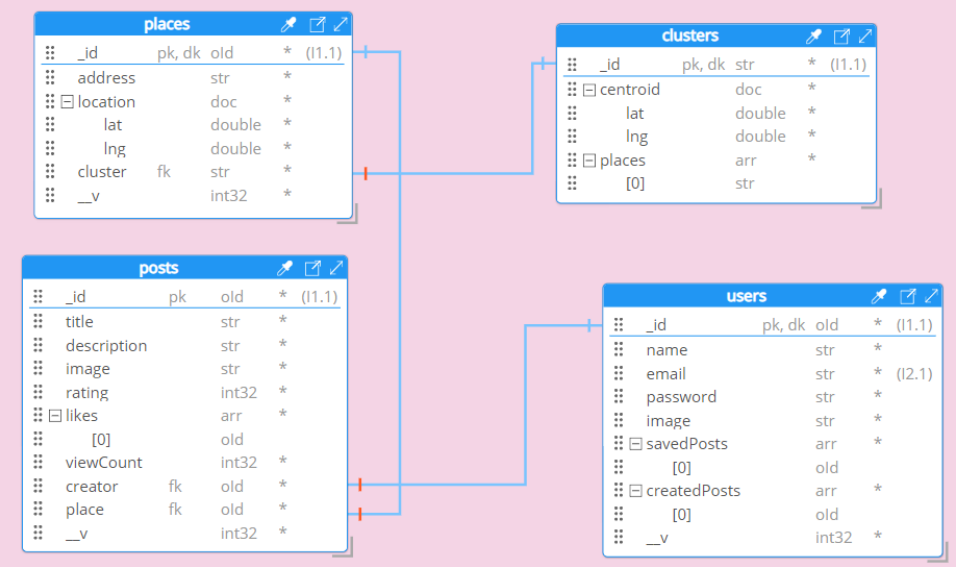

Figure 4.2: Database diagram

4.2.3 Caching

Using a caching layer, the application can significantly reduce the load on the backend and improve response times. Caching frequently accessed data helps avoid expensive database queries or computationally intensive operations, resulting in improved performance, and a better overall user experience. In the case of the actual application, the need for caching is demanded by the place recommendation algorithm, which performs more time-consuming computations.

The most comfortable option was Redis, an open source in-memory data structure store that allows the storage and retrieval of various data structures such as strings, lists, hashes, etc. The choice was supported by Redis Labs, a fully managed in-cloud Redis service, similar to Atlas due to its care of infrastructure setup,

maintenance, and monitoring. Therefore, advanced features like high availability, automated scaling, and data persistence are provided. It supports various deployment options, so I chose AWS.

4.2.4 Frontend

For the client side, the choice was React, a popular Javascript library that is used to build highly reactive and modern user interfaces. It is widely adopted because of its flexibility, performance, and developer-friendly features, which make it a go-to for creating interactive and dynamic web applications.

Despite these, React has many key features that fit this application. The component-based architecture where the user interface is divided into reusable and independent components is a major advantage. I made use of functional components because they are simpler and can be passed properties from a parent component that allows customizability, following the one-way data flow, known as ”one-way binding”. This helps to maintain a predictable state and makes it easier to understand how data change throughout the application. React also uses a virtual DOM because the original HTML DOM was not created for dynamic content. Instead of re-rendering the whole element, only a minimal amount of operations to update the DOM is performed.

Another advantage of using this library is the JSX syntax, a powerful extension for Javascript that allows you to write HTML-like code. It allows you to define your components' structure and appearance in a more intuitive and expressive way.

Last but not least, one of the most representative things for React are the hooks. These allow for using the state management and lifecycle features without writing classes. Hence, it is possible to maintain stateful logic in functional components, by building custom hooks or by using the inbuilt ones. For example, some of the most popular and effective hooks, that I have also used are:

- **useState.** The hook lets you add a state to your functional components. It returns an array with two items: the current state and a function to update the state. You can call this function from an event handler or somewhere else to trigger a state update and rerendering of the component.

- **useEffect.** This hook lets you perform side effects on your function components. Side effects are operations that affect the world outside of the function scope, such as updating the DOM or making a network request. It takes two arguments: a function where you put your side-effect logic, and an optional array of dependencies. An empty array as the second argument means that it runs only once after the initial render. If you pass a list of variables, the effect will be re-run when any of those change.

- **useCallback.** The hook returns a memorized version of the callback function that only changes if one of the dependencies has changed. It is useful when passing callbacks to child components that rely on reference equality to prevent unnecessary renders.

In addition to these, custom hooks were also implemented, which helped during development. One of them is named useHttpClient, and its purpose is to handle HTTP requests using the fetch API. It abstracts the complexities while providing a simple interface to send requests, track the loading state, and handle errors. With the help of useRef, it holds an array of active HTTP requests' abort controllers that will persist for the component's whole lifetime. They attach their signal to the respective request, hence aborting active requests when the component is unmounted is possible because at that moment a cleanup useEffect is run.

4.3 Design patterns

Apart from the MVC already used and discussed, some other design patterns were used to facilitate the development.

4.3.1 Cached singleton

For example, I used cached singleton, implemented like in Figure 4.3, to establish the database connection to make sure there is only one instantiated object. Therefore, I decided to make it simple and take advantage of the Node.js caching mechanism, meaning that after the first time a module is loaded, it is also cached. So, every call to it returns the same object if it resolves to the same file.

This implementation is not similar to the classical one, as in Figure 4.4, because there is no private constructor or the getInstance method, which solves the problem with one class instead of two. The trick is done because instead of exporting the class, an instance of it will be exported. Node.js will cache it and reuse it whenever it is needed.

4.3.2 Provider

In React, the provider design pattern is used to share state or behavior among several components without passing props manually at every level of the component tree (prop drilling). It is implemented using React's Context API, which allows the creation of a context object that is given default values. My implementation from Figure 4.5, may serve as an actual example. This can be seen as a global state

```
class DatabaseConnection {
  init(app) {
    this.databaseConnectionUri = `mongodb+srv://${p
    this.app = app;
  }

  getNewDbConnection() {
    mongoose.connect(
        this.databaseConnectionUri
      ) // mern = db name
      .then(() => {
        this.app.listen(process.env.PORT || 5000);
      })
      .catch((err) => {
        console.log(err);
      });
  }
}
module.exports = new DatabaseConnection();
```

Figure 4.3: My own cached singleton

```
class PrivateSingleton {
    constructor() {
        this.message = 'I am an instance';
    }
}

class Singleton {
    constructor() {
        throw new Error('Use Singleton.getInstance()');
    }

    static getInstance() {
        if (!Singleton.instance) {
            Singleton.instance = new PrivateSingleton();
        }
        return Singleton.instance;
    }
}

module.exports = Singleton;
```

Figure 4.4: Classical singleton

for a set of components. After that, the Provider is used to share the data and functionality with the component tree that will be wrapped in. It allows a value property that contains an initialization of the context that will be provided. This allows any descendent component to access them by calling the useContext hook, without wrapping it inside a Consumer.

I chose to implement it in the application for the login part, to manage some global states that would be used across the entire application. Thus, the whole router in App.js, at a very high level, was wrapped inside the Provider, which also receives the value property where its properties get values using the custom useAuth hook. Therefore, all components rendered for the available routes to a user, depending on its login state, will have access to the context.

```
import { createContext } from "react";

export const AuthContext = createContext({
    isLoggedIn: false,
    userId: null,
    token: null,
    login: () => {},
    logout: () => {} ,
    setIsLoading: () => {}
});
```

Figure 4.5: Context creation

4.3.3 Builder

This pattern is a creational one, and it aims to simplify object creation by separating the construction from the object representation, enforcing the separation of

concerns this way. The construction of the object is done step by step, where each of them corresponds to setting a particular property or configuration option., using the builder's provided methods.

In the actual context, the builder pattern was chosen to be used to create multiple Mongoose models. For example, Figure 4.6 shows how the user model uses it because, during the sign-up process, it facilitates code readability and control.

```
const createdUser = new UserBuilder()
  .setName(name)
  .setEmail(email)
  .setImage(req.file.path)
  .setPassword(hashedPassword)
  .setSavedPosts([])
  .setCreatedPosts([])
  .build();
```

Figure 4.6: User builder

4.4 Functionalities and implementation

The following section is about getting into the details of the web application implementation. Each feature will be discussed, as well as the approaches used to accomplish the functionality.

First of all, the CORS policy is established as a middleware function because the server has to share resources when responding to requests from different origins (the React client in this case). Figure 4.7 shows the specified origins from which the requests may come, along with the possible headers and methods allowed.

```
app.use((req, res, next) => {
  res.setHeader("Access-Control-Allow-Origin", "*");
  res.setHeader(
    "Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept, Authorization"
  );
  res.setHeader("Access-Control-Allow-Methods", "GET, POST, PATCH, DELETE");
  next();
});
```

Figure 4.7: Cors policy

There are two types of features, those available when logged-in, and the others available when logged-in or unauthenticated. To illustrate all possible actions for a user, the use case diagram in Figure 4.8 is the one that clearly shows them.

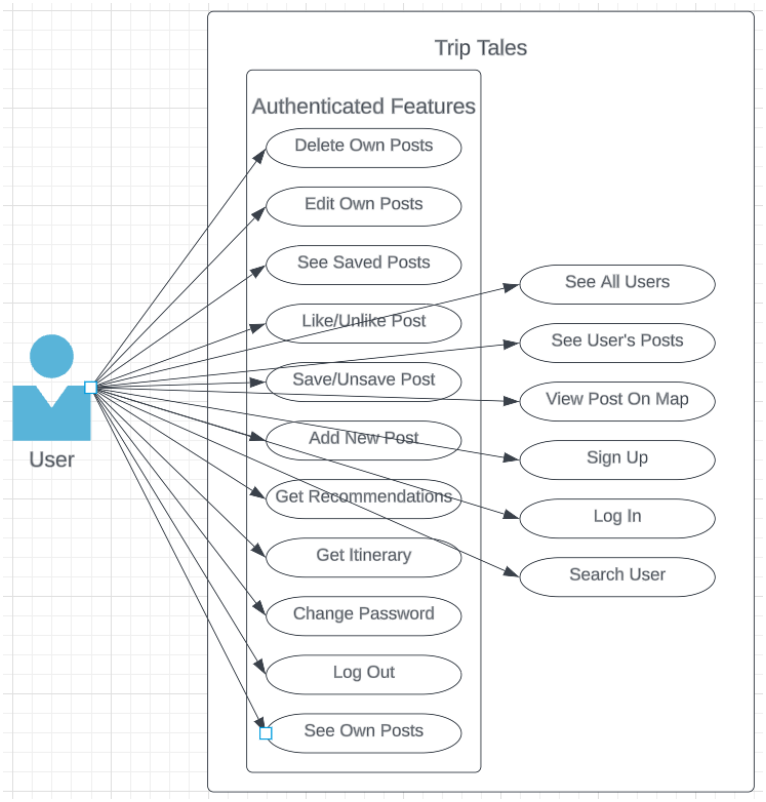

Figure 4.8: Use case diagram

4.4.1 Features

Sign up

Some conditions have to be respected in order to have a successful use case. The new user must upload a photo with a valid extension, that is, PNG, JPG, or JPEG, which should have a maximum size of 5MB. Here, Multer comes into action, it saves the file to a specified disk path, changing its name to an uuid. The user also has to introduce a valid email, a name, and a password which must contain at least 1 uppercase, 1 lowercase, 1 digit, 1 special character, 8 characters, and its confirmation. These field validations are performed using the express-validator package. The server checks with the email if the user already exists. If it does, a 422 error is sent, otherwise, the user is created, using bcrypt node package to hash the password with more salt rounds, and jsonwebtoken package to generate a token to log in immediately after signing up. The token takes as payload some user data, uses a secret key, and is set to expire in one hour. Signing up also leads to deleting cached data.

Log In

This is somewhat similar to the sign-up because the same packages and ideas are used. The difference is that the credentials inputted in a form as the one from Figure 4.9, are checked and a new token is emitted if they are valid. The related user data is stored in the browser using local storage, and a callback is set to automatically execute the logout at the time of token expiration. To make the login persistent and survive on page refresh while the token is available, my trick is to auto-login at every page refresh, with the help of useEffect hook, which is incorporated in my own custom useAuth hook.

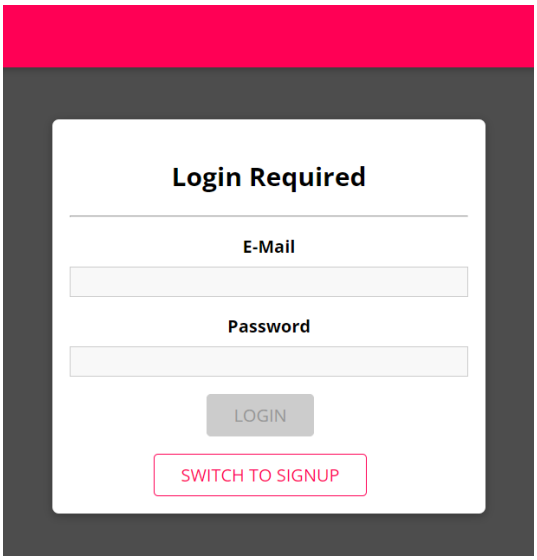

Figure 4.9: Login screen

Log Out

This is achieved by clearing the related local storage and states. This can be done by pressing the log-out button or automatically when the session token expires.

Change Password

If a user wants to change his password, the old password, new password, and new password confirmation must be entered. If the first one is wrong or the last two do not match or meet the security requirements, an error modal is shown like the one in Figure 4.10; otherwise, the data is updated and a new token is not emitted in order to re-login the user because it is not based on this piece of data.

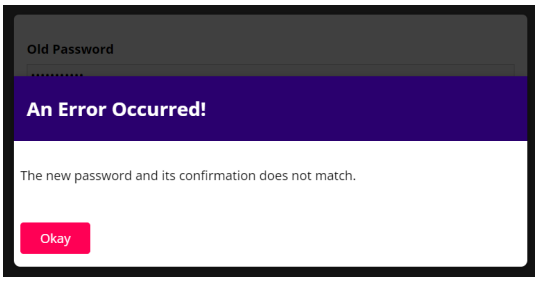

Figure 4.10: Error modal example

Edit Post

This is done using a Patch request. It allows a user to update his own posts. Only the title, rating, and description can be updated. The only validations are to introduce a non-empty title, a description of minimum five characters, and a rating between 1 and 10. I chose to leave the photo uneditable because the same applies to most social media platforms.

See All Users

The homepage is available to both authenticated and unauthenticated users and presents all users within the application. Every user is displayed within a Card component, with its avatar image and name. Clicking on it leads to the next features.

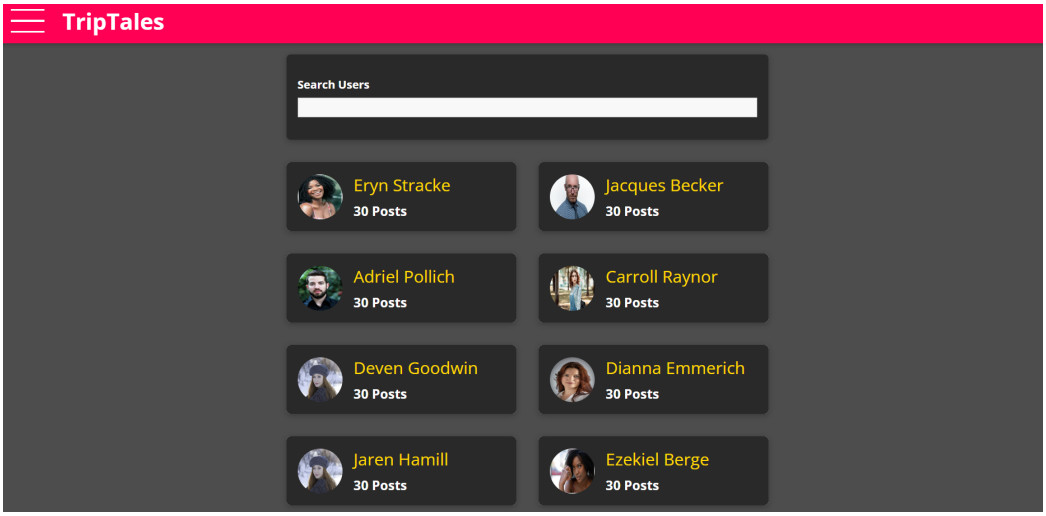

Figure 4.11: Homepage

See User's Posts

All its posts are each displayed within the same Card component but styled differently from the user's one. Here, the user can like/unlike or save/unsave the post

because he is logged in. Authenticated or not, you can also view the place on the map, as in Figure 4.13, which is rendered on a Modal component using the Google Maps SDK for Javascript. The post's view count is also incremented if the user is logged in and has not already seen it.

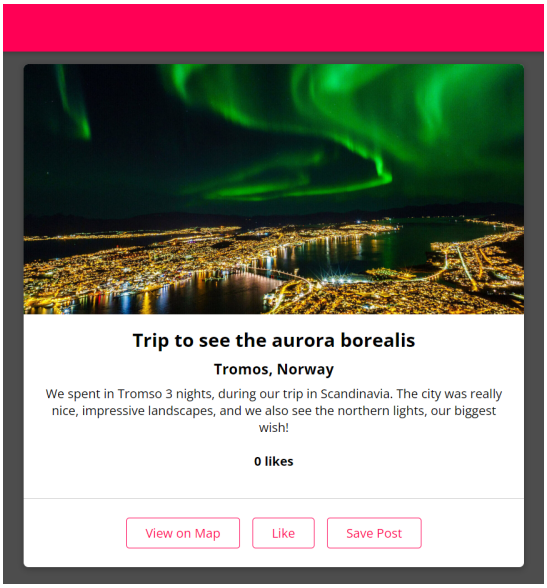

Figure 4.12: User's post

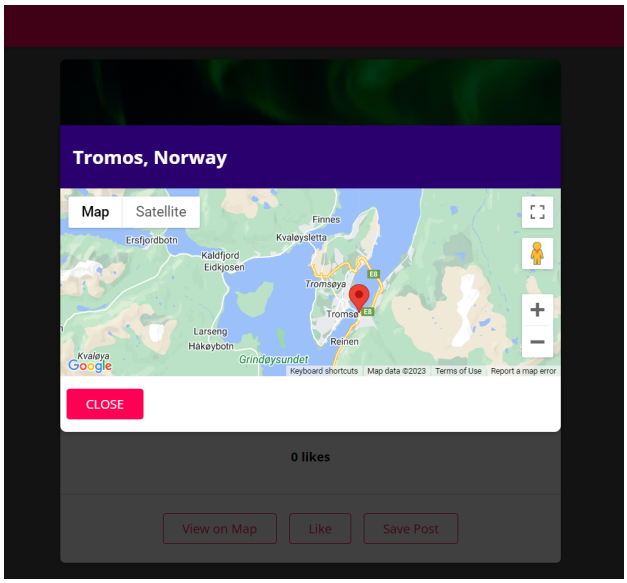

Figure 4.13: Rendered map

Like and Save Post

I put them in the same subsection because they act quite similarly. If the button is white with red text, it means that you did not like the post. When pressing it, your user is added to the post's likes list, which contains all the users that liked it. Then, the button's colors are inverted. The opposite is applied if you already liked the post and press the button. Regarding the save post button, it works the same, only that the user has its own saved posts list that will get added or removed the post that is interacted with. The buttons can be seen in Figure 4.12.

Get Recommendations

Based on the algorithm described in Chapter 3, the application recommends to the user the top 3 destinations of travel which have not yet been visited. The server will retrieve the cached data, or compute and cache it for one hour and a half. Then, the result is sent to the user by email in the format from Figure 4.14, using the node-mailer package. If the application is in a very initial phase and there is not enough data, an appropriate message is shown that recommendations cannot be done yet.

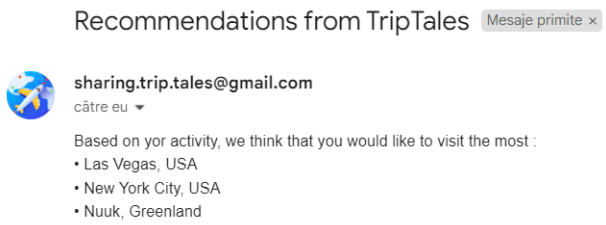

Figure 4.14: Mail with place recommendation

Delete Post

When a user wants to delete one of his own posts, a confirmation modal pops up. If it is confirmed, a simple Delete request is sent, deleting the post and its related photo, using the fs module.

Add Post

This is exactly the same as updating the post, except that it uses a Post request instead of a Patch and you have to add an address and upload a photo. All inputs are passed in a form similar to that in Figure 4.15. The photo validations are the same as the ones described at sign-up. For the address, Google Maps Geocoding API [MKL19] is used because it is able to translate it into coordinates that are further used to render the map or compute the clusters for the recommendation algorithm. If the place that the post is about does not exist in the database, a new one is created and assigned to a cluster as described in subsection 3.2.2.

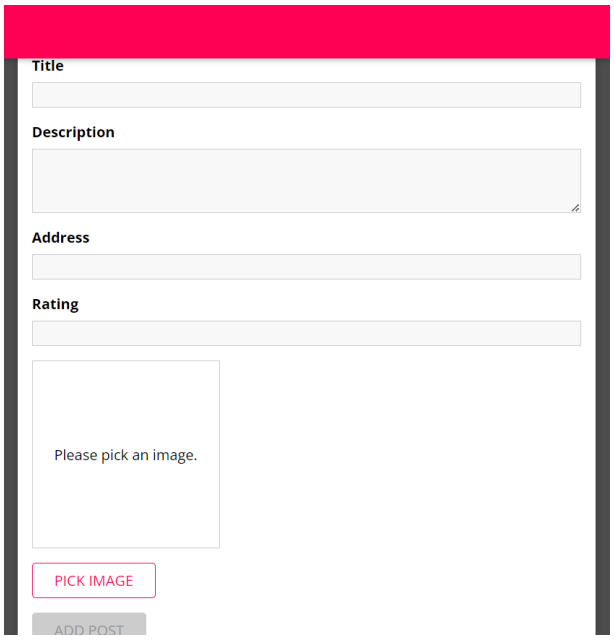

Figure 4.15: Add post

See Own Posts

This is basically the same thing as seeing some other user's post, only that you cannot save your own posts. Instead, you can edit or delete them, as can be seen in Figure 4.16. When seeing your own posts, the view count is not incremented.

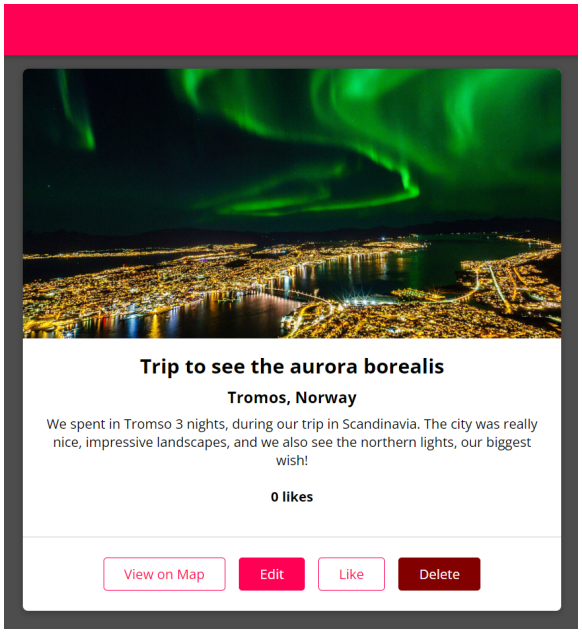

Figure 4.16: Own post

Search User

To search users within the whole application, there is an input on the homepage, as seen in Figure [4.11], that takes a name. The search query is sent to the server with a Get request, where it is trimmed, and all the users that contain it in their names are displayed, as Figure 4.17 exemplifies.

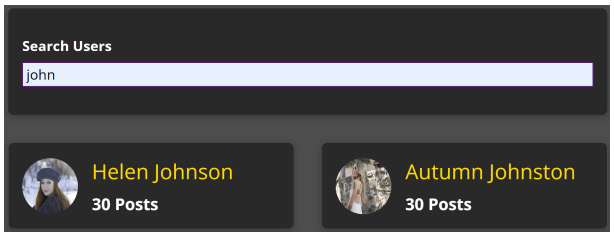

Figure 4.17: Search user example

Get Itinerary

In a form similar to that used to add a post, a user must fill in the destination, the number of days, and the number of people. The server will process the data and

send it to the OpenAI API that uses the GPT (generative pre-trained transformer) models which are specialized in understanding and generating natural language. It will respond with a full itinerary for each day of the trip and with three different recommended accommodations. Furthermore, all this content will be sent to the user via email, in a format similar to the one from Figure 4.18.

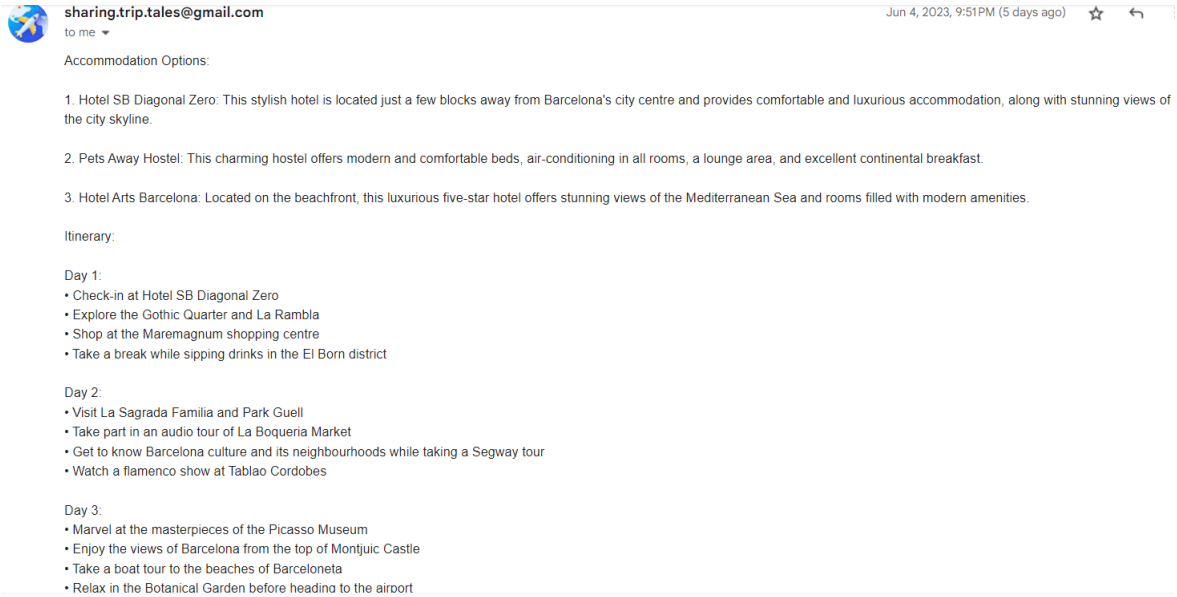

Figure 4.18: Mail with itinerary

Chapter 5

Conclusion

5.1 Comparison with SOTA. Results

Due to the fact that the data was completely random (a person who is enthusiastic about cold places could also go, for example, to Sahara), it is impossible to make an accurate estimate of the overall accuracy of the model. Without a proper logic, in chaotic data, it is impossible to divide the database into a test and "train" set, so as to evaluate the accuracy of the recommendation algorithm only knowing the "train" subpart of the data. Basically, the similarities found in users could be disrupted at each step, due to the adding of a completely random rating that "does not make sense". However, one way to evaluate the recommender system, as described before, is the one where I went through the predictions and saw that they are aligned with the similar users, and have their encapsulated logic that is correct and would be correct in a not-so-random (real) system, due to the isolation of similar users.

This is the creative nature of hybrid recommendation systems such as the one I developed, which involves collaborative filtering based on user input, content filtering, and implicit feedback. It involved a lot of fine-tuning, localizing similar clusters, and trying to obtain the best and logical assumptions of future ratings. Other SOTA recommenders, for instance, Netflix used metrics such as RMSE to evaluate the difference between predictions and real ratings. However, the Netflix database is rooted in reality, so they were able to extract a test subset of the data and evaluate on it. Amazon, for instance, used item-based collaborative filtering through others, to determine how similar items are and to recommend according to the user's preferences. A/B testing and metrics like Precision at K were used to evaluate and improve their recommendation engine, but again, these are not applicable to my system because of the need for real users.

In conclusion, the experiment to create this system has shown that obtaining useful and logical recommendations from randomly generated data is a complex

process. Although traditional metrics may not apply, the developed recommender has demonstrated its potential by aligning predictions with similar users and making sense within the context of the data.

5.2 SWOT analysis

Strengths

- TripTales is an easy-to-use and engaging application. Its features are user-friendly and intuitive. It can also be used by anyone because it is deployed and accessible through any Internet browser.
- Due to the caching system, recommendations are made almost immediately.
- The application has the potential to be utilized at a large scale, as a social media app by bringing together travel enthusiasts who want to discover more new places. This is also a nice environment to share posts about experiences and thoughts regarding past travel experiences, which may be helpful for others.
- The itinerary recommendation feature can be helpful for people to organize their trips and to find new attractions in a certain place. It also helps people find accommodation because three are recommended.
- The application itself would be easy to scale because its components are deployed independently.

Weaknesses

- When the recommendation algorithm does the computations, the load is huge because there is a lot of data to process, with 15000 posts and 500 users being the greatest numbers among all the models. They may also increase as TripTales acquires more users. When running it after the deployment, until the fix was done, the application crashed because the JavaScript heap was out of memory. These heavy computations are a long-running process that leads to the next weakness, which is somehow a compromise.
- Using cached data, with an expiration time, to avoid heavy computations, results in slightly outdated recommendations. Also, the first recommendation after the cache expires will take more time than usual.

Opportunities

- The application can be extended to a real scenario, regarding the recommendation system. Real data can be easily brought to the database; therefore, a unique dataset can be created to be used for scientific purposes about traveling or recommendation systems, for example.
- The application has the potential to be utilized at a large scale, as a social media app by bringing together travel enthusiasts who want to discover more new places. This is also a nice environment to share posts about experiences and thoughts regarding past travel experiences, which may be helpful for others.
- Seeing such posts may encourage people to travel more and pursue their dreams regarding this.

Threats

- As the amount of data grows, it would be harder and harder to perform the computations regarding the recommendation algorithm.
- The need for RAM and CPU increases, and so do the costs, due to the need for a more performant server.

5.3 Future improvements

Several areas regarding the application and research of the recommendation system can be explored.

For example, caching can be improved. When a change that would affect the recommendations occurs, such as liking, saving, or adding a new post, a task can be queued to be executed as a background process to update the cache. This would not interfere with the main process, so the user experience is not affected, and the recommendations will be kept up to date.

Another idea is to introduce comments to the posts, which would make the application more interactive. Thus, more data would be available about places. Google NLP has an API that can be useful to extract sentiments or emotions from comments. They can be attached to places as labels and considered when making recommendations.

A friendship or follow-based system would be interesting because it connects similar users in a more interactive manner. This may open the door to providing personalized content to each user.

A chatting feature would also be something to consider. The direct connection between users increases, allowing them to share more details about trips, places, or travel tricks.

To enhance the recommendation algorithm, more features can be considered, in addition to likes, saved posts, view count, and cluster bonus. Time would also be crucial in observing trends, and the most recent likes would be more relevant, for example. Experimenting with real users and data should also be considered as a further experiment, which would require fine-tuning to update the weights for the features.

In conclusion, this recommendation system uses an original hybrid approach and can be further explored in different research areas, improving the way recommender systems are implemented.

Bibliography

[BHK98] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, page 43–52, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018.

[GPB13] Mustansar Ali Ghazanfar and Adam Prügel-Bennett. The advantage of careful imputation sources in sparse data-environment of recommender systems: Generating improved svd-based recommendations. *Informatica (Slovenia)*, 37:61–92, 2013.

[GUH15] Carlos A. Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manage. Inf. Syst.*, 6(4), 2015.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[LW12] Youguo Li and Haiyan Wu. A clustering method based on k-means algorithm. *Physics Procedia*, 25:1104–1109, 2012. International Conference on Solid State Devices and Materials Science, April 1-2, 2012, Macao.

[MKL19] Heeket Mehta, Pratik Kanani, and Priya Lande. Google maps. *International Journal of Computer Applications*, 178:41–46, 05 2019.

[MMN02] Prem Melville, Raymod J. Mooney, and Ramadass Nagarajan. *Content-Boosted Collaborative Filtering for Improved Recommendations*. American Association for Artificial Intelligence, USA, 2002.

[Mol18] Leidy Esperanza Molina. Recommendation system for netflix. 2018.

[SBE+21] Harald Steck, Linas Baltrunas, Ehtsham Elahi, Dawen Liang, Yves Raimond, and Justin Basilico. Deep learning for recommender systems: A netflix case study. *AI Magazine*, 42(3):7–18, Nov. 2021.