Tripendum: your next trip in Trentino

GitHub: `https://github.com/gzemo/bdt-project-tourism-recommendation-system`

Abstract — **Recommendation systems correspond to a set of methodologies and techniques implemented to offer an "enhanced" user experience by sorting through clients' preferences from a vast amount of data. Such systems can assist users in various decision-making processes, and are mostly suitable to the extent of suggesting goods to buy or, generally, experiences to make.**

The current project aims at presenting *Tripendum,* a recommendation system implemented specifically to suggest tourism-related experiences to potential clients visiting Trentino-Alto-Adige, whose historical Regional County Seat, Trento, was originally known as *Tridentum*.

Keywords — **Collaborative Filtering, Alternating Least Squares (ALS), Trentino-Alto-Adige, tourism.**

I. Introduction

Among many recommendation system techniques, one of the most reliable algorithms implemented is collaborative filtering, which is based on gathering and analyzing customer's data to provide suggestions based on the similarity between the same user and a pool of other clients (*user-user* approach) [1].

In this project such technique was implemented adopting an hybrid approach, since it converges the memory-based approach comparing the previous user-item interactions and the model-based approach predicting the outcome of the interactions for the items the user did not interact with. The purpose of this project is therefore to provide a new client with a set of recommended activities or experiences according to past users' individual preferences who have enjoyed similar experiences.

A recommendation system has been built by simulating synthetically generated data and by taking inspiration from a real world platform such as TripAdvisor. A strategy was devised to encourage the first initial N users (the initial *baseline* pool of potential clients) to interact with the system multiple times, visiting places or experiencing different activities and leaving reviews. This approach aimed to establish a solid foundation of the initial set of reviews, mitigating the issue of data sparsity. The recommendation system proposed takes into account an *hybrid* approach which aims at merging both the memory- and model- based paradigms. The core model algorithm involved was implemented by adopting the Alternating Least Squares (ALS) technique that has been proved to efficiently handle large-scale sparse datasets [2]. This feature indeed is commonly hindering the recommendation procedure of describing and predicting complex user-item latent relationships in order to foresee future ratings and thus recommend accordingly.

The present demo addressed exclusively all those municipalities belonging to the Trentino-Alto-Adige's territory. This initial set of data was gathered with the purpose of testing the workflow of the whole system and, consequently, the recommendation engine by exploiting real data. Nonetheless the application has been developed by ensuring to be independent with respect to the regional boundaries involved and can be therefore extended and generalized to be adapted to collect and process POIs regardless of the geographical location.

II. System Model

A. *System architecture*

In the current project a conceptual, logical and physical data model, described in *Figure 1*, has been defined for each of the following entities:

- *Municipalities*: cities where a given POI may belong to.
- *Users:* clients whose initial simulated registration and suggestion request, based upon the set of preference, simulate a real pool of customers using the application.
- *POIs*: node-like object defined by its spatial coordinates, the facility's name and additional tags.
- *Ratings*: outcome gathered from the interaction between the user and the suggested POI.

<u>Data Collection</u>: the initial set of raw POIs is retrieved by querying for all those nodes that satisfy a given set of attributes from *Overpass*, a freely available API service which provides a ready to use query and filtering syntax to directly interact with the *OpenStreetMap*'s API [3]. The system architecture is best described by the flowchart in *Figure 2*, which depicts the hierarchical process and the order of the stages involved, and will be discussed as follows.

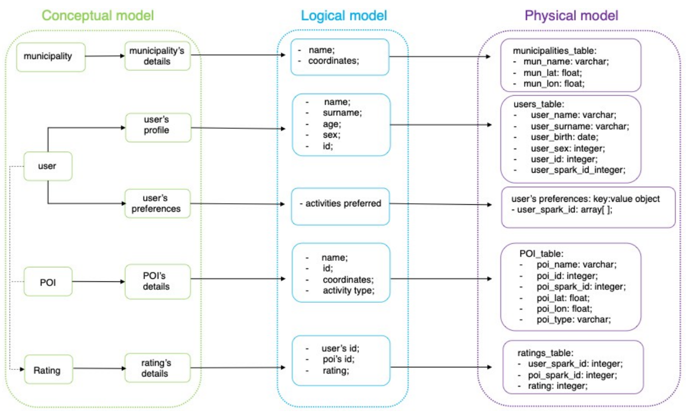

Figure 1: conceptual, logical and physical data model.

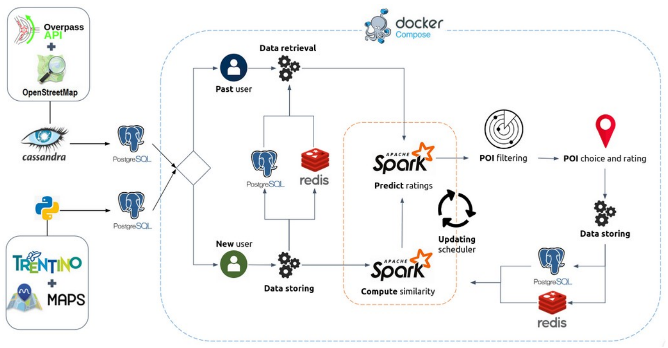

Figure 2: project's workflow.

An additional API endpoint (*Geocoding*) [4] has been used with the purpose of retrieving latitude and longitude for each previously gathered municipality to further filter the resulting recommendation output. Each request has been carried out by jointly maximizing and minimizing the nodes' retrieval and the number of actual *Overpass API* requests respectively by executing a single *Overpass API* query for each trip category currently under examination. For this purpose an initial set of 8 trips' categories had been initialized and the related best corresponding *Overpass* query command had been chosen to capture a wide range of nodes defined under the same category (*see Figure 3*).

<u>Data Generation:</u> users' demographic attributes and ratings were synthetically generated using the python *Faker* library [5], due to the unavailability of data of this kind and since such data is considered personal and mostly owned by private companies which allows for limited free access. Users were also simulated to choose between a set of given journey categories to better mimic the initial trip's preferences selected by the user.

<u>Data Ingestion and Cleaning</u>: The ingestion of Raw POIs, which come as semi-structured objects, was sequentially managed to be performed to be stored directly. The motivation under this choice relies on the fact that the currently gathered data are made available by a third-party relatively low-end server which reserves the right of hindering user's access whether a high rate of request is encountered. To avoid this danger the resulting semi-structured object is directly stored into a Cassandra cluster to be further retrieved and processed. Raw POIs objects are then parsed and filtered in order to save all those items that are actually referring to a real distinct POI thus excluding any unspecified or common facility resulting in 2219 retained nodes.

<u>POIs Storing</u>: To accommodate the different types of data retrieved effectively, we have opted for different storage solutions. A Cassandra cluster is implemented to store the raw POIs data retrieved from *OpenStreetMap*'s API in order to be further parsed and cleaned to be saved in the persistent relational storage.

A PostgreSQL database is used first for storing filtered POIs' details, such as name, latitude and longitude, category type and eventually the address, and secondly for storing municipalities' names along with the corresponding coordinates.

<u>Encapsulation:</u> a Docker network is initialized to allocate distinct containers through which the following pipeline's stages are made available and performed.

<u>User Baseline Simulation:</u> a pool of clients has been initialized in order to fill the initial set of reviews that will be further loaded to fit and maintain the recommendation engine. Each user identity is sequentially initialized and a random set of POIs visited and corresponding ratings given is provided. To deal with the severe sparsity each user is then spawn and processed to provide several ratings over different POIs.

<u>Request submission and storing:</u> the pipeline allows then to simulate a real queue of users requesting suggestions to the recommendation system which can be either previously registered ones or new clients. A PostgreSQL database is used for storing users' profiles, which includes personal details such as name, surname, age and sex, and the users' ratings to the points of interest visited. Redis was adopted for storing the users' vector of preferences, which is a binary representation of the categories preferred which will be further passed as input for estimating users' similarity score.

<u>Recommendation engine:</u> the core recommendation system relies on the following three stages.

- *Similarity estimation:* when a new, unregistered, user requests a suggestion the system firstly needs to find a pool of previously registered users that match its initial set of trip's preferences to provide a final recommendation which is based upon what other clients with similar preferences had chosen before. This stage fulfills the need for predicting its future ratings given that, being its first request, no ratings have been provided yet. Starting from an initial threshold and to the vector of user's preferences the system iteratively searches and filters the pool of

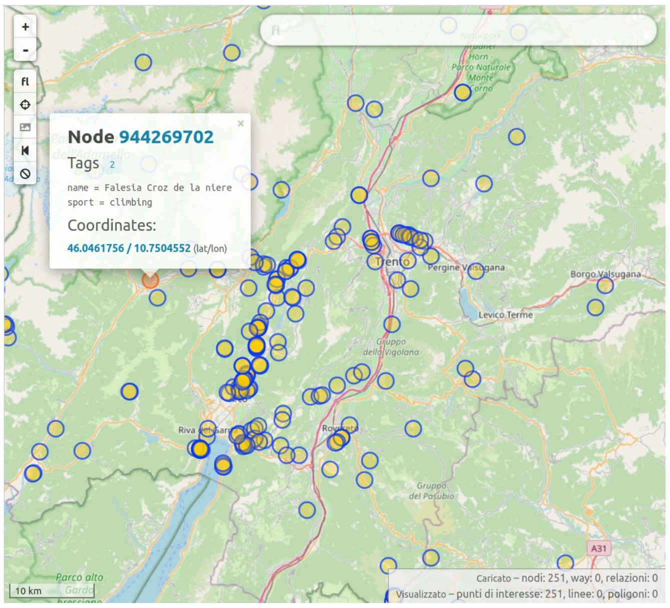
Node 944269702
Tags 2
name = Falesia Croz de la niere
sport = climbing
Coordinates:
46.0461756 / 10.7504552 (lat/lon)
Trento
Pergine Valsugana
Borgo Valsugana
Levico Terme
Gruppo della Vigolana
Riva del Garda
Rovereto
Gruppo del Pasubio
Parco Naturale Monte Corno
Parco alto Garda bresciano
A31
10 km
Caricato – nodi: 251, way: 0, relazioni: 0
Visualizzato – punti di interesse: 251, linee: 0, poligoni: 0

Figure 3: example of resulting nodes gathered from an OpenStreetMap request: the figure displays all facilities related to any "*climbing*" sport facility. The highlighted node defines a single raw node with an unique identifier, latitude and longitude, and 2 tags (name and sport).

baseline users to find a subset of neighbours by means of the Jaccard similarity's score.

- *Ratings prediction:* once the subset of most similar users is found, we employed the aforementioned ALS algorithm to predict the missing user-item matrix, which contains all the predicted ratings from POIs not already visited by the pool of baseline users. All POIs are then displayed along with their rating and averaged.
- *Recommendation filtering*: the user will be then advised with the top N POIs ordered by the highest rating given (or, indeed, predicted by the former phase). A set of filtering procedures are then initialized to ensure that the resulting POIs suggested are among those belonging to the preferences chosen by the user and within a given maximum distance previously defined.

Final storing and model update schedule: once the resulting filtered POIs are rendered the system simulates the act of a user reviewing the suggested experience. The process then saves the chosen POIs' id by updating the set of already visited places within the Redis cluster and the triplet $\{user\_id, poi\_id, rating\}$ is updated in the ratings relational table. Since new client's ratings predictions are provided by consulting all records belonging to the initial baseline pool of users an updating schedule is then implemented in order to routinely re-train the ALS model by also including the batch of all those $\{user\_id, poi\_id, rating\}$ excluded from the initial set of training data. This task is triggered when a certain number of new users' requests are received and successfully stored and may also be addressed with a time schedule (currently under development).

B. *Technologies*

All technologies and data sources involved in the present report were primarily chosen to address two key concepts: *a)* to rely entirely on open-source only and easily deployable software in straightforward and highly customizable manner; *b)* to be ready to scale effectively in light of any future unpredicted usage.
An exhaustive list of softwares follows.

PostgreSQL: since many of the datasets implemented in this project can be conceptualized as relational tables, and therefore structured in a tabular form, we have opted to employ PostgreSQL as a storage solution due to the robust support for managing structured data with a defined schema on write [6].

Redis: is an in-memory open-source storage that allows replications to be made to efficiently distribute and retrieve information by a search-by-key pattern [7]. It offers fast response times and is specifically designed to scale effectively and to dynamically manage changes in the structures of input stored. The software has been implemented to store the user's preferences and POI visited to handle the retrieval of all users preferences vector for the similarity estimation and to help in the computation of the similarity score, for which we need to retrieve the feature vectors of all existing users, which are stored in Redis, and parse them to a Spark dataframe, which is more efficient and faster when utilizing an in-memory dataset for users' preferences, as opposed to storing it on disk (e.g., in PostgreSQL).

Apache Spark: provides a powerful framework for handling large-scaled dataset computation enabling to potentially distribute the data under examination in a clustered environment that can be scaled horizontally across multiple infrastructures [8]. It was responsible for mainly addressing the managing of the user preferences and the user-item datasets that holds users' ratings alongside with the relative POI. Spark has been leveraged jointly with the *MLib* python API layer to compute the Jaccard similarity score based on users' preferences in order to identify a representative sample of the most similar existing clients to be closely related with new user within the system and to fit the ALS model for ratings imputation.

Apache Cassandra: is an open-source Wide-Column data storage built upon a masterless architecture [9]. The choice fell on Cassandra firstly due to the highly scalability potential and since it was originally thought as an initial storage for all raw POIs nodes gathered from *Overpass API*, for which a fast writing performance storing was needed to lower the chances of losing data. Indeed, with the idea of scaling the project to potentially map a wider geographical area, a fault-tolerant distributed software was needed to prevent any future pitfalls and to be able to scale horizontally. Secondly, with the specific needs of storing semi-structured raw data that shared a common pattern, we decided to exploit the multi *primary key* design to efficiently retrieve the related set of unstructured tags by a specific POI feature.

Docker: to ensure portability and compatibility, we have chosen to deliver a ready-to-use system via Docker containers [10]. For each aforementioned technology a container is provided through Docker Compose which enables the integration of each key step of the presented work's pipeline and consequent implementation regardless of the operating system currently in use. An additional "master" container in charge of orchestrating the workflow simulation is also provided.

III. Implementation

Any technology involved has been firstly installed and developed locally, to be efficiently tested and included in the project in a *module by module* fashion.

To facilitate the simulation, the original datasets composed of raw *OpenStreetMap API* nodes and Trentino's municipalities will be made available from the beginning in JSON and CSV files respectively, in order to exclude the API requests from the Docker based simulation. Nonetheless the complete project, including also the *Overpass API* requests, are made available following the commands described in the github repository.

Initialisation: the simulation begins with two processes. A parser (*./POI/OSMparser.py*) object is instantiated to connect, retrieve and store *OpenStreetMap* nodes into Cassandra by means of several *Overpass API* queries. All POIs satisfying the following two conditions are retrieved:

- The node is located into the ray of 50km from the origin (which, in the current case, corresponds to latitude and longitude of Trento).
- At least one tag of the node is contained into the *./POI/overpass_query_block.txt* file which lists all Overpass API queries that are to be performed.

All the POIs retrieved for each *Overpass* query (which defines, namely a "block" of nodes sharing the same trip's category) are stored into Cassandra along with the category they belong to: the pair $\{id, block\_name\}$ works as primary keys and will then be used to retrieve back and finally store the filtered node into a PostgreSQL dataset. Beside, by connecting to the *Geocoding*'s API, the coordinates (latitude and longitude) for each municipality contained in *./POI/municipalities_of_trentino.txt* are then retrieved and stored into the MUNICIPALITIES dataset.

User's registration: the user gives their personal details to the server, which are simulated by the python *Faker* package. Then the user selects their preferences for the activities proposed, which correspond to the same set of categories used to parse the POIs (see *./POI/overpass_query_block.txt*) and retrieved each time from the POI dataset, and the municipalities dataset respectively, to maintain the available pool continuously updated (see *./clients/common.py*). The storage process for the user profile is computed into two parallel steps: first, connecting to the PostgreSQL database, the user personal attributes are stored in the USERS dataset (see *./clients/common.py*). Secondly, a key-value pair containing the id of the current user and a binary vector based on the user's selected categories is inserted into the Redis database with the attribute preferences (see *./clients/common.py*).

Data preparation: an intermediate step is needed in order to retrieve data from the previously mentioned storages to build the preferences (from Redis) and ratings (from Postgres) Spark DataFrames to further compute the similarity estimation and the ratings prediction respectively. While for the latter this was performed by directly connecting the database to Spark for the former we explicitly iterated over the currently stored keys matching the given lookup pattern.

Recommendation algorithm: depending on the user, whether it is already-subscribed or a new coming one, the ALS algorithm works differently to provide a pool of recommended activities (see *./run/recommend.py*).

For *new users* the system identifies a pool of existing most similar neighbours from those already registered based on the similarity of the corresponding attributes in the PREFERENCES dataset. The process progressively extends the pool of most similar users by increasing incrementally the maximum allowed similarity threshold until a minimum number of neighbours is found. On this restricted pool of users the ALS algorithm predicts the ratings of all the points of interest in the POI dataset and then the averaged score is performed for each point to obtain a final set of possible destinations sorted by the predicted rating.

For *already registered* users the system firstly retrieves all those POIs' id corresponding to the experiences the user already had to get all POIs not visited yet and then ALS prediction is automatically applied to predict the missing ratings.

Filtering procedure: Using the list of suggested POIs, a temporary Pandas data frame is created with the following details for each POI suggested:

$$\{POI\_name, POI\_distance, POI\_category\}$$

The table is then filtered in order to obtain *N* rows corresponding to the points recommended, sorted by distance, and with an activity correspondence with the user's preferences (see *./POI/selection_POI.py*). From this new list of recommended points of interest, the user will randomly select one.

Rating assignment: supposing that the user will go directly to the point of interest suggested and that will be willing to leave a review for the point of interest visited, the simulation proceeds with storing the review for the current user and the point of interest visited into the RATING dataset.

IV. Results

After successfully completing the simulation, some statistics are computed on the data that have been stored (see *./run/statistics.py*). Namely, one can appreciate which POI was most visited and which got the highest average rating and which clients that took the highest number of trips or the most satisfied by recommendation system. Currently an implementation to keep track of user-item density estimation in order to evaluate the percentage of filled entries over the total is under development.

In the GitHub repository it is possible to see some pictures of the whole simulation to have a better understanding of the demo (see *./simulation_screens*).

Video of the simulation available at: https://youtu.be/2GPTJ6xAG38

V. Conclusions

The implemented work showcases a collaborative filtering recommendation system designed for tourists in specific locations. The dual stage recommendation fulfills the task of suggesting new, unseen destinations by jointly exploiting the properties of memory- as well as model-base recommendation technologies along with a *user-user* approach. The prediction of rating, from which the consequent suggestion is gathered, is routinely updated in order to provide a model which is continuously learning from incoming user reviews. Moreover, even though not explicitly implemented, the current system may easily allow for trips' categories and user preferences updates, potentially dealing with the circumstances in which new sets of holiday experiences are made available to the public in that specific area.

Nonetheless several suboptimal implementation choices are actually hindering the system to achieve the highest efficiency: future updates may address the following limitations.

Due to time and system constraints, any implementation discussed so far has been integrated and tested without explicitly providing a cluster-like replication with more than a single node partition for technologies like Cassandra, Redis and Spark. In addition to this, no backup implementation was addressed to save an emergency copy of the relational-like storage such as user profiles and ratings.

The use of the technologies implemented as storage solutions could have been more efficient, exploiting technologies' features and packages or even adopting other solutions. For the former, examples are the coordinates of points of interest and municipalities, which would better fit a Redis storage to exploit the Redis Geodist-related functions. For the latter, we could have used Spark not only as the main computation engine but also as a temporary storage by duplicating the flow of information coming to the platform by directly loading the $\{user, item, rating\}$ triplets into a Spark DataFrame. If so, implementing a mini-batch loading of the data system would have resulted in a more efficient implementation rather than using a continuous loading process from the persistent storage. Furthermore, the connection between Redis and Spark is thought of being suboptimal since it has been not implemented through the *spark-redis* package [11] which requires the use of Scala and Jedis, but was performed iteratively through python.

An additional limitation regards the *./POI/municipalities_of_trentino.txt* file, which was manually created for this project, since a better solution would have been implemented by a prompt-based scraping and parsing module to extract the municipalities for a given geographical area.

Regarding the recommendation system it should also be considered that, by using randomly generated customers with a likewise random set of preferences, it is not expected to provide accurate performances as real world data would provide. Furthermore the whole project has been developed without relying on the GitHub maintenance system, yielding manual intervention to fulfil the task of integrating project's updates and code modules.

A final limitation for this project was the absence of any pub/sub implementation (like Kafka or Redis) to handle the flow of customers at the beginning of the ingestion process.

In conclusion, although the system implemented for this project is proved to be suitable for small demos with a limited number of users and POIs and it would in principle require to be adjusted to organize an increased amount of data, it provides a reliable starting solution to be efficiently enriched by future improvements.

References

[1] Jia, Zhiyang & Yang, Yuting & Gao, Wei & Chen, Xu. (2015). User-Based Collaborative Filtering for Tourist Attraction Recommendations. Proceedings - 2015 IEEE International Conference on Computational Intelligence and Communication Technology, CICT 2015. 22-25. 10.1109/CICT.2015.20.

[2] Paleti, L., Radha Krishna, P. & Murthy, J.V.R. Approaching the cold-start problem using community detection based alternating least square factorization in recommendation systems. *Evol. Intel.* 14, 835–849 (2021). https://doi.org/10.1007/s12065-020-00464-y

[3] Overpass official website. Available at: https://overpass-turbo.eu/

[4] Geocoding official website. Available at: https://geocode.maps.co

[5] Faker official website. Available at: https://pypi.org/project/Faker/

[6] PostgreSQL official website. Available at: https://pypi.org/project/Faker/

[7] Redis official website. Available at: https://redis.io/

[8] Apache Spark official website. Available at: https://spark.apache.org/

[9] Apache Cassandra official website. Available at: https://cassandra.apache.org/_/index.html

[10] Docker official website. Available at: https://www.docker.com/

[11] Repository for reading and writing data from and to Redis with Spark: https://github.com/RedisLabs/spark-redis/tree/master