UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

STUDY OF EVENT RECOMMENDATION IN EVENT-BASED SOCIAL NETWORKS

Autor: ORIOL JIMÉNEZ GONZÀLEZ

Director: DRA. MARIA SALAMÓ
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, June 27, 2018

Abstract

Recommendations are in our every day life: *streaming services, social media, web pages...* are adopting and using recommender algorithms. Recommendation algorithms benefit both parts: clients can find more easily products that they like, and the companies make more benefits because clients use their services more. The recommendation problem presented in this work is a non-traditional variant of this problem as it recommends events. Events, unlike books or videos, cannot be recommended in the same way, because users cannot rate an event until the day it happens, and then no new users can rate it again after that. This magnifies a problem called "cold start problem" where every new event has no ratings, which greatly complicates the recommendation problem. This work studies Event Recommendation for a social media called Meetup[1] where users can attend a selection of events created by the community. Although users do not leave a rating of the event, we have a signal called RSVP[2], which is a non-obligatory mark on whether the user has the intention to attend the event or not. In this work we will be exploring how different recommender algorithms perform to recommend events based on RSVPs and also propose three new algorithms. The analysis will be done with 5 datasets extracted from Meetup during the months between November 2017 and April 2018. The results show that hybrid versions containing collaborative and contextual-aware algorithms rank the best among all the algorithms tested.

Resum

Les recomanacions estan presents en el nostre dia a dia: *streaming services, social media, webpages...* estan adoptant i utilitzant algoritmes de recomanació. Els algoritmes recomanadors beneficien a les dues parts: els clients poden trobar més fàcilment productes que els hi agraden, i les companyies tenen més beneficis perquè els clients utilitzen més els seus serveis. La recomanació presentada en aquest treball és una variant no tradicional que recomana events. Aquests, a diferència d'altres productes com son llibres o vídeos, no es poden recomanar de la mateixa manera, perquè els usuaris no poden valorar un event fins que passa, i un cop passat cap usuari nou el pot valorar. Això magnifica un problema anomenat "cold start problem" on cada nou event no té valoracions, el qual complica significativament el problema de recomanació. Aquest treball estudia la recomanació d'events per una xarxa social anomenada Meetup[1] on els usuaris assisteixen a una selecció d'esdeveniments creats per la comunitat. Encara que els usuaris no donen una valoració de l'event, tenim una senyal anomenada RSVP[2], que és una marca no obligatòria que indica si el usuari té la intenció d'anar a l'event o no. En aquest treball explorarem com diversos algoritmes recomanadors es comporten recomanant events basant-se en els RSVPs, i a demés proposarem tres algoritmes nous. L'anàlisi es farà amb 5 datasets extrets de Meetup durant els mesos de Novembre 2017 a Abril 2018. Els resultats mostren com versions hibrides que contenen algoritmes col·laboratius i contextuals obtenen millors resultats entre tots els algoritmes provats.

[1] www.meetup.com
[2] RSVP: Répondez s'il vous plaît

Resumen

Las recomendaciones están en nuestro día a día: *streaming services, social media, webpages...* están adoptando y utilizando algoritmos de recomendación. Los algoritmos recomendadores benefician a las dos partes: los clientes pueden encontrar más fácilmente productos que les gustan, y las compañías generan más beneficios porque los clientes usan más sus servicios. El problema de la recomendación presentado en este trabajo es una variante no tradicional ya que recomienda eventos. Los eventos, a diferencia de otros productos como son libros o vídeos, no se pueden recomendar de la misma manera, ya que los usuarios no pueden valorar un evento hasta el día que ocurre, y una vez pasado ningún nuevo usuario puede valorarlo. Esto magnifica un problema llamado "cold start problem" donde cada nuevo evento no tiene valoraciones, lo que complica significativamente el problema de recomendación. Este trabajo estudia la recomendación de eventos para una red social llamada Meetup[1] donde usuarios asisten a una selección de eventos creados por la comunidad. Aunque los usuarios no dan una valoración del evento, tenemos una señal llamada RSVP[2], la cual es una marca no obligatoria que indica si el usuario tiene intención de atender el evento o no. En este trabajo exploraremos como diversos algoritmos recomendadores se comportan recomendando eventos basándose en los RSVP y además propondremos tres nuevos algoritmos. El análisis se hará con 5 datasets extraídos de Meetup durante los meses entre Noviembre 2017 y Abril 2018. Los resultados muestran como versiones híbridas que contienen algoritmos colaborativos y contextuales obtienen mejores resultados de entre todos los algoritmos probados.

[1] www.meetup.com

[2] RSVP stands for the French expression "Répondez s'il vous plaît", meaning "please respond"

Contents

Chapter 1

Introduction

1.1 Problem

Recommender Systems (RS) [10] try to answer a common problem in the digital world: with so many options of products, users have to scroll through a large quantity of items to find what is the best of for them. This process is often long and tedious for the user, so having a system that can help in that decision is often a key factor in web services, social networks and E-commerce, to name a few. In order to do that, RSs have to gather data from both users and items, either by item and user descriptions, by search and purchase history of users, or even by mobile data position and mobility patterns. After that, RSs creates rules and relations between items and users to determine the best product for each user.

The main focus of our work will be seeing and comparing different algorithms that can employ a gathering of historical user data to make predictions in a collaborative filtering recommender. Collaborative Filtering (CF) [10] is a technique used in RSs that relies only in data of other users to make their predictions, and bases how a user will rate an item based only on how other users rated it in the past. The premise of this idea is that users with similar tastes will like similar things. Later in this work we will be also exploring Context-Aware (CA) [10] alternatives: a RS that exploits external factors like mobility and time patterns of the users to predict items that are more affine to the user.

In this work we will be focusing our work in a social network called Meetup. Meetup is an network in which users can attend to events created by the community and join different groups, based on their preferences. Each group has a profile and a description, and organizes meetings or events with their own members. In this Social Network, members of a group inform their intention to go to a presented event or not using a RSVP. As we will see later, this is a rare occurrence and most of the time it does not happen, but because we cannot tell if a user goes to an event or not in any other way, we will be using this metric to make recommendations. The assumption we will be having with RSVP is that if a user marks a positive RSPV, we will suppose the user went to the event, and if he marks an event with a negative RSVP, he dislikes the event and he did not go to said event. Finally, we choose this platform as it allowed us to gather data as it has an API[1] which has data from their users, events, and RSVPs.

Social Networks (SN) have been using RSs for quite some time now, and it is well-documented field of work, but we will be working with an specification of this case: Event-Based Social Networks (EBSN) [2]. IN EBSN, users explore and attend events based on their preferences, and join groups that organize certain types of events. EBSN has a few of added challenges given by the context that we will be exploring

[1] Application Programming Interface. Link: secure.meetup.com/meetup_api

in this work. The most important problem in EBSN is the "cold start problem", which also happens in traditional recommender systems, but in EBSN is greatly magnified. The "cold start problem" is a well-known problem in computer-based information systems that concerns the issue of a new user or item. New users come with no information at the start, so the system is unable make assumptions of which preferences it has and thus cannot compare the new user with any existing user in the database. To alleviate this problem, is common practice for social networks to make new users "pass a test" of which categories they prefer, so the recommender has something to work with, even if broad and general (in particular, Meetup does this). However, most of the time that is not enough, and some further measured need to be taken to alleviate this problem. Also, for new items the problem is different, unless the items have some type of description (and that only happens in recommender systems with Content-Based filtering (CB) [10] or hybrid methods [10], not for collaborative filtering), so this issue is really important in EBSN and alleviating this problem is key to success.

Another big problem in EBSN that affects recommendations is the transience of events versus items in traditional RS: an event is not "consumed" in the same way a movie or an ad can be. Events are temporal, which means the system cannot get a rating of the event in real-time. If we tried to implement an EBSN as a traditional recommender system, we would have to wait until the day of the event to get the rating of the users, and at that point the data is irrelevant because the event is already over anyway. Specifically, in EBSN, events will never have proper ratings, because events are temporal, so we will need to use a metric called RSVP[2] which is a non-obligatory indicator that users make to confirm assistance to an event.

Formally, given that the number of items is quite large and the number of events a user is interested in is pretty small, we want to optimize the RS so that the best items that it recommends are the best for the users, more than listing all the events that the user might like. Thus, the problem presented in this work can be stated as a **ranking problem**: given a target user, and a set of attended events or contextual signals, which of the available events are more likely to be attended by this user? We will need to compute a ranking of the N more likely events for each user. In order to do that we have to compute an array $U\ x\ E\ x\ R$ (User U, Event E, RSVP Rating R). Operating this array will give us a rating like hood $f(u, e, r) \in \mathbb{R}$. We will compute this rating for each of the user-item pair, and then compute the top-n best recommendations as following:

$$topN(u) := \underset{e \in E/E_u}{\overset{n}{\operatorname{argmax}}} f(u, e, r)$$

Where E/E_u denotes the subset of E of events that the user u has the attended. This function will then return a ranking of the n best events to the user. Using this ranking we will compute statistics of success and performances of the algorithms.

1.2 Motivation

Nowadays, every major social network or selling platform has a RS attached to it. RS can be used in a variety of cases, ranging from product recommendation to ad placement. These systems benefit all the parts involved: users can find the right product for them, and companies cater to a wider variety of users. That interaction also benefit companies because users become more invested in the platform and generate more revenue. In some cases, recommendations make up a big part of all the income of a company. In the case of EBSN it is also really important, specially living in a city, because the number

[2]RSVP: from the french "répondez s'il vous plaît"

of groups and events is really high, and a single user could never filter through all the information. If instead of that, we can show a short list of events sorted by how relevant are to the user, we are going to see a big difference in how users engage with the platform.

EBSN is still a relatively new field of work, and there still needs more research. We want to study the nuances of EBSN in regard to traditional RS: what extra challenges we have, why do they appear, what does the context tell us about the way users interact with other users and items and the system, what techniques work best and why, etc. In this work we primarily explore collaborative filtering and context-aware although there are more approaches to the recommendation problem (we cover this in the Section 1.3, *Field of work*). We decided to go with collaborative filtering because it is the most standard approach and it also let us try and compare a wider range of algorithms that are inside this category. We will later explore in this work what downsides does Collaborative Filtering have, and what tools do we have to alleviate those problems. Context-aware data is also a common choice for EBSN, as the context let us exploit user patterns like mobility or time. We seek to explore how different algorithms can work under these circumstances, and how we can fix said problems, for example by using a hybrid algorithm combining the two approaches.

1.3 Field of work

In the early stages of the Internet, RS were simpler versions of what are today, mainly being document search using content-based information, like tags. Most of the sources seem to point the origins of RS as we know it now in some web journals in the mid-1990s, like Tapestry[3] [12] which would forward content to relevant readers. As the web grew, the amount of information that could be stored and computed grew with it. The first use of collaborative filtering was GroupLens, which added a system that would let users rate items, and the RS would pair up users to create correlations between users, which was one of the first Nearest-Neighbor approach algorithms. After that, more and more services and platforms found this new technology useful and created implementations of a RS tailored to their problem. Amazon, Google with their ranking algorithm PageRank, eBay... After some time, different algorithms would surface, different enough to be able to make a classification of them, each one with their own strengths and weaknesses, see Figure 1.1. The most common classification [10] is:

- **Content-based:** The system learns to recommend items that are similar to the ones the user liked in the past. The similarity of items is calculated based on the features associated with the compared items. In order to do that, there needs to be a *Content Analyzer*, which extracts relevant information of the items either by keyword matching or via TF-IDF which creates a vector of features, a *Profile Learner* which constructs a user profile based of past actions and ratings, and a *Filtering Component* that matches user profiles with item profiles using similarity metrics, which are context dependent.
- **Collaborative Filtering:** The simplest and original implementation of this approach recommends to the target user items that other users with similar tastes liked in the past. The similarity in taste of two users is calculated based on the similarity in the rating history of the users and affects how much that opinion is weighted to compute the final rating. Usually ratings are not present for each pair of user-item, so inherently to this technique appear a set of problems related to it: Firstly, the aforementioned "cold start problem", which indicates that Collaborative Filtering is not considered good for a new user or item until we get some information of it, usually in the form

[3] Tapestry.com

of ratings. Secondly, the sparsity of the datasets lead to bad results, because with less ratings the system has less comparisons to make and also worse confidence rates. Finally, tastes are almost never popular, and most of the time we will have a distribution called "long tail problem", where users with uncommon tastes add up to 80% of the dataset. Even all its flaws, Collaborative filtering is still considered to be the most popular and widely implemented technique in RS.

- **Demographic:** This type of system recommends items based on the demographic profile of the user. The assumption is that different recommendations should be generated for different demographic niches. While these approaches have been quite popular in the marketing literature, there has been relatively little proper RS research into demographic systems.

- **Knowledge-based:** Knowledge-based systems recommend items based on specific domain knowledge about how certain item features meet users needs and preferences and, ultimately, how the item is useful for the user. These RSs do not fully estimate the utility for the user but instead apply some extra heuristics to determine what is best for the user. Case-based RS use a similarity function made to find what is best following a set of rules, while Constrained-based RS use explicit rules for how to pair users with items.

- **Community-based:** Also called social recommender systems, this type of system recommends items based on the preferences of the user's friends. Evidence suggests that people tend to rely more on recommendations from their friends than on recommendations from similar but anonymous individuals. Although it has a similar approach as Collaborative Filtering model, it has the advantage that it does not require data of the user, but rather to know preexisting users, which means that it is more resistant to the new user problem.

- **Hybrid:** These RSs are based on the combination of the above mentioned techniques. The usual idea behind this is that combining techniques helps fix the disadvantages of one another. The most important type of hybrid RS are context-aware systems, which are usually paired up with Collaborative Filtering to boost the performance, as collaborative data can be sparse in some cases and context-aware can solve fill the gaps with extra information. Adding that data to the current RS can be a challenge, and three different algorithmic paradigms exist: reduction-based, contextual post filtering and contextual modeling. In reduction-based (also called contextual pre-filtering), information about the context is used for selecting or constructing the relevant set of data, so later a traditional recommender can work with the filtered dataset. In contextual post filtering, the recommendation algorithm ignores the context information and computes ratings. Then, the resulting set of recommendations gets adjusted with the contextualized data. In contextual modeling, context data is explicitly used in the prediction model as part of rating estimation.

EBSN are a specification of a larger type of recommender systems which are social networks. In standard social networks, content aware, collaborative filtering, context-aware and community-based RS are all popular. However, EBSN relies most of the time in collaborative filtering and context-aware because of some reasons, mainly that events have time and location and the lack of rating information. One subclass of EBSN exists, and that is Offline Ephemeral Social Networks, or offESN for short. OffESN are networks created ad-hoc for a specific location and a specific purpose and lasting short periods of time, like a network created for attending to events in a convention. This field is also new and gaining some traction, but we will not be covering this field as it has its own set of challenges.

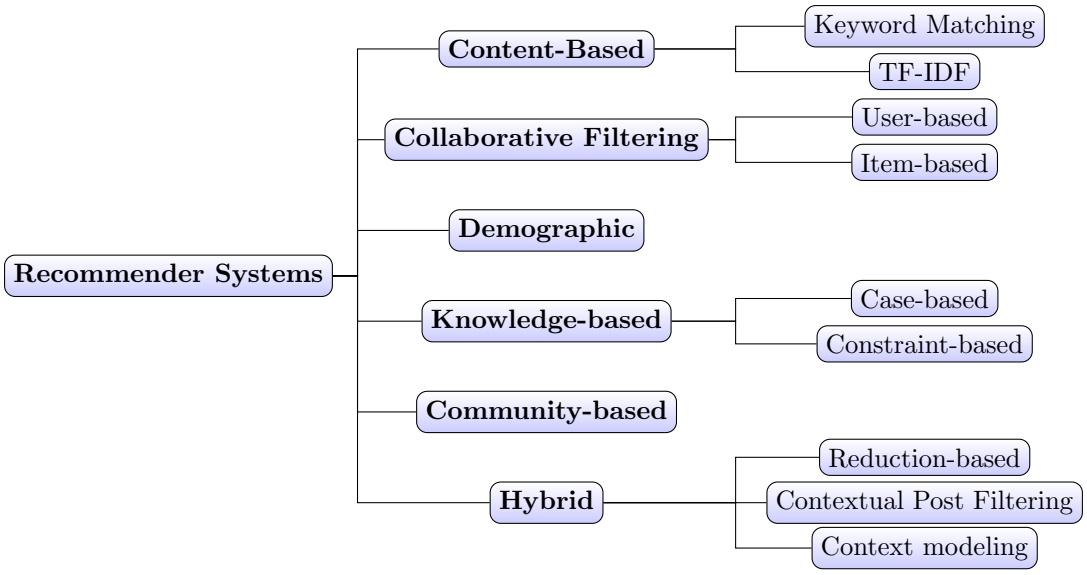


Figure 1.1: Basic classification of Reccommender Systems.

1.4 Objectives

In this work we aim to explore the field of EBSN: what features does this context have, what kind of data it has, what options are there to make recommendations. In order to study EBSN we plan to make a collection of datasets using Meetup data, a well-known EBSN, using the different endpoints in their API to get useful information from five different cities in the US. Then, we will process the datasets, and look into the properties of it and of the context, and in the end we intend to use the datasets to test a list of algorithms with the aid of a Java library called *Librec*[4]. After examining the results, we plan to explore another approaches to the recommendation problem using context-aware solutions. The main objectives we want to accomplish on this work are:

- Generate a collection of datasets using data from Meetup API. In order to test algorithms, we need an algorithm that holds information about users, items and ratings as well as some contextual data. To make the datasets more reliable, we want to download data from at least 5 different cities in different locations in the US.

- Observe and compare the performances of multiple algorithms when tested under our datasets in different scenarios. We will be using a list of algorithms, from simple and naive algorithms (*Random* , *Most Popular*) to some generic ones (*ItemKNN* or *UserKNN*) and some state-of-the-art recommender algorithms (*Biased Recommender MF*, *Bayesian Personalized Ranking (BPR) [14]*, *SVD++*). We will be testing all the algorithms under different conditions, like different cities, sparse and dense datasets, etc. and comparing who does best in each scenario. The goal is not so much to find the best algorithm, but to find how they vary under different circumstances and why.

- Study the properties of the context and the datasets: When and why are RSVP created? What can we extract from them? Is there any way to alleviate the "cold start problem" and the sparsity inside the datasets? Do different cities rate different? Are differences in users relevant to the problem? Asking all these questions about the context is really important to understand how to tackle the problem and will give us insight in the recommendation problem.

[4] www.librec.net

- Extract contextual data of the context and demonstrate how context-aware RS can give a better solution to the "cold start problem". In particular, study whether or not distance and time are important factors to recommend events. Furthermore, we are interested in hybrid RS using both Collaborative Filtering and context-aware information. We will apply the same tests applied to non-hybrid RS and determine if they boost the prediction performance.

1.5 Report Overview

In chapter "Related Work", we will be analyzing 7 related works of the EBSN literature and finding out what new approaches and solutions they bring to the table to combat common problems in EBSN. In chapter "Analysis", we will be describing the whole process of constructing a group of datasets, starting with downloading the data to storing and filtering it, and finally structuring it to start our tests. In chapter "Implementation and Design", we will discuss diverse properties of our datasets and also explain all the procedures and methodology of the experiments, giving reasoning to the setting of our tests, and arguing the results.

Chapter 2

Related Work

Even though event recommendation in EBSN is a pretty new field of work, we already have a few works discussing it, each one pitching their own idea. Because rating information is very sparse in this field (RSVP or other ratings are usually optional) most of the works seem to go for context-aware solutions, and it seems like the best solution to combat the "cold-start problem" is to get additional sources of information, e.g. digital traces, geographical traces, etc. There are also a few works that use mobile data to infer user position and preferences. Mobile data information is getting more and more prevalent as it becomes more widespread and easy to obtain, and seems like the field is moving in that direction. Seems important to note that no method is yet regarded as 'best' by the community: the context of the recommender (amount of data and kinds of data) can change the approach used. Finding a model that fits the context and the datasets seems to be the common factor to getting the best performances.

Event Recommendation in Event-based Social Networks

Macedo and Marinho [2] study EBSN and all their properties. They claim that is an unexplored area in recommendation problems and extensive analysis of the different features of EBSN such as sparsity, event life time, co-participation of users and geographical features are needed. They also use RSVP as an indicator of whether a user liked an event, even though they say it is a sparse indicator, getting to 99% sparsity. Mainly, they want to address the following points regarding EBSN:

- How sparse is the RSVP data and how does it affect collaborative filtering problems?
- In which point of the event lifetime users tend to RSVP?
- How the geographically distance between the users home and active events affect their decision at attending these events?
- Are past RSVP useful for predicting future RSVP?

Their data gathering is being made using Meetup API from 3 major cities in the USA: Phoenix, Chicago and San José. Macedo and Marinho chose these cities for the quantity of users and their variation in location, so they have a diverse enough dataset. In this case, they retrieved user, event, and RSVP data. Then they move on to a section which shows interesting insights about RSVP which will also be studied in our work or at least mentioned. They show a cumulative distribution to showcase the sparsity of RSVP over events. The graphic reveals that 90% of users have at most 10 RSVP, and most of them much less. They also study event lifetime and RSVP occurrence, in order to understand which temporal features

affects when a user RSVPs an event. They find that while events have a somewhat long timespan (5 to 100 days), users tend to notice events much later, as the RSVP usually happen a few days before. They close the observation by indicating that collaborative vs. content based options should be determined based on time left of the event, as events with short lifespan will not get many RSVP.

Macedo and Marinho also mention that co-participation is an important factor, with 30% of the users co-participating in two or more events.

Finally they also study distance as a factor to event recommendation, and make the observation that 50% of all ratings users made were positive around 10KM distance, while users did not provide great ratings passed the 100KM.

Macedo and Marinho also present their own results obtained with the data gathered and a few recommender algorithms. They selected 6 time stamps, and created a train-test partitions based on if the events was before or after said timestamp. They also split the datasets by sparsity, as we will be doing later in this work, by separating the datasets by number of RSVP, and study separately the results. The evaluation metric will be NDCG (Normalized Discounted Cumulative Gain) which is a well-known metric used in top-n ranking problems. The algorithms tested in these experiments are a grand total of 6: Random, Most Popular, Location-aware, BPR-MF, user KNN, item KNN, logistic regression (which is a hybrid algorithm containing all the previous recommenders except random).

The results show how KNN algorithms tend to score better, and that most popular is as good as location aware, which seems to indicate that popularity has a role as important as distance in event recommendation. Also their logistic regression ranks slightly better that user and item KNN. Also as expected, all algorithms (except location aware and random) rank better with less sparse datasets.

Macedo and Marinho end drawing some conclusions of these experiments: RSVP is a really sparse metric, and the majority of events are cold-start. Even in these conditions, KNN options seem to rank best, while some well-known algorithms as Matrix Factorization options does not seem to work as in other fields.

Context-Aware Event Recommendation in Event-based Social Networks

Macedo et. al. [9] study how certain algorithms and context-aware recommendations work in EBSN, specifically using data from Meetup, downloading it from the Meetup API and using RSVP solely to measure user-event interaction. Macedo et. al. try to find alternatives to the "cold start problem" which makes recommendations in EBSN unsatisfactory to the users. In their work, they propose different context-aware RS exploiting social signals based on group membership, location signals, temporal signals, etc. Their assumption is that each one of these signals have a positive relevance on the user's decision to attend an event.

Macedo et. al. formalize this problem as: given a target user as a set of contextual signals, which one of the avaliable events are more likely to be attended by the user? Formally, they define for the target user the set of users U and events E, the user's temporal preferences T, the set of groups G, user's geographically distance preferences D, and the textual contents of events C. The set S is defined as $S \subseteq U \times E \times T \times G \times C \times D$ of the relations between user, event, time, group, content, distance. The objective is to find $\hat{S} : U \times E \times T \times G \times C \times D \to \mathbb{R}$ that assigns a preference score for the candidate events, so the top-n can be defined as the top n best scores of that preference function.

They propose different contextual models exploring the different defined social signals:

- **Social Aware:** Recommender based on the number of events the target user attended in the group that he belongs to. Formally, assuming that g_e represents the group associated with the candidate event e:

$$\hat{s}_{s1}(u, e) := \frac{|E_{u,g_e}|}{|E_u|},$$

where $|E_{u,g_e}|$ is the number of events created by g_e attended by u and $|E_u|$ is the number of events that user u attended to. Furthermore, Macedo et. al. also made another implementation of this social signal using a Bayesian Personalized Ranking (BPR) [14], which has as parameters $\Theta := U, E, G$ to compute the n-ranking with.

- **Content Aware:** Content-aware references mainly to event descriptions, which help identify similar events and recurring events. They created this signal with a bag-of-words model: Each user is represented as a TF-IDF vector of the words extracted past events attended. Additionally, they weight each event by a decay function by its time of recommendation instant. Formally, this is defined as:

$$\vec{u} : \sum_{e \in E_u} \frac{1}{(1+\alpha)^{\tau(e)}} \times \vec{e},$$

with $\vec{e}$ being the TF-IDF of the event candidate and $(1+\alpha)^{\tau(e)}$ represent the decay function over time. The candidate events are then ranked on the cosine similarity $\hat{s}_C(u, e) := cos(\vec{u}, \vec{e})$.

- **Location Aware:** Macedo et. al. propose a kernel-based density distribution, stating that it models user mobility patterns better than other, more basic options. Formally, they define

$$\hat{f}_l := \frac{1}{|L_u|} \sum l' \in L_u K_H(l - l'),$$

where l are the lat-lon coordinates of the event, K_H is the Gaussian kernel and and L_u is the lat-long coordinates of all events attended by the user u.

- **Time Aware:** The hour and day in which the user attends events can also be important in event recommendation. Formally, each event $\vec{e}$ can be modeled as a 24x7 vector as the space of all hours/days of the week, with 0 if the event did not happen in that hour and day, and 1 if it happened. Then, we can represent $\vec{u}$ as $\vec{u} = \frac{1}{E_u} \times \sum_{e \in E_u} \vec{e}$, and the recommender as the cosine similarity between user and candidate event: $\hat{s}_T(u, e) = cos(\vec{e}, \vec{u})$.

After all these implementations, Macedo et. al. explain their experiments, detailing their dataset which contains information about Chicago, Phoenix and San José, information about groups, events, and RSVPs. Their evaluation protocol is a complex sliding window system of splits created from 12 stamps, so they can realistically test their algorithms. They also document the recommenders they use in the tests: Most Popular as baseline, BPR, BPR-NET, which is an implementation of an approach created by Z.Quiau et. al. in their work, and the recommender created in this work, a multi-contextual learning to rank events (MCLRE), which fuses all the aforementioned factors into a single RS, and has a gradient descent to assign the importance of each factor. In their results can be seen that this last algorithm outperforms all the others by a wide margin, and Macedo et. al. end by commenting how the use of multiple contexts can lead to high accuracy and mitigate the "cold start problem", and how these context aware information seem more important than the standard RSVP.

Recommending Social Events from Mobile Phone Location Data

Quercia et. al. [13] study how geolocation through mobile data can break the "cold start problem" in EBSN. Their study consists of two datasets: a sampling of location estimations of one million mobile phone users in the Great Boston area, and a gathering of data of social events in the same area. Their goal is to create different algorithms using location information to figure out the best solutions to the common problems in EBSN, specifically help alleviate sparsity problems in RS.

As we said, Quercia et. al. use and combine data from two separate datasets; the first one is from Airsage Inc. and gives information about where users live and go. It has location estimates from one million mobile phone users (20% of the total population) in the Great Boston area during summer 2009. From this dataset, they made a subsample of 80.000 users. The second dataset is extracted from the "Boston Globe Calendar" which has information about social events that happened in a $15Km^2$ area of Great Boston, which then they subdivided in a grid-like pattern to discretize positions. After merging the two datasets, the final data has 53 events and 2,519 unique users, which is 97,3% sparse. Quercia et. al. then subdivide the users by postcode, shrinking down the matrix from 53×2519 to 53×49, which lowered the sparsity to 50,50%.

Then Quercia et. al. move onto explaining the six recommendation algorithms they will be using:

- **Popular events (Most Popular)**: The ranking score is proportional to how many users attended the event.
- **Geographically close events**: The ranking is inversely proportional to the distance user-event.
- **Popular events in the area**: Most popular among residents in the area (rank proportional to the number of users who live in the same location that attend the event).
- **Term Frequency Inverse Document Frequency (TF-IDF)**: much like the popular TF-IDF algorithm used in information retrieval, tries to extract the popular events everywhere and only select the regionally popular events.
- **K-nearest locations**: Tries to find the k-similar locations to the one where the user lives. The similarity of two zones is weighted according to the number of events that users attend living in one zone separately and together.
- **K-nearest events**: similar to k-nearest locations, this algorithm finds the k-most similar events to the event we are trying to predict. Similarity is computed by the cosine similarity.

Quercia et. al. decide to use a metric called percentile-ranking. The percentile-ranking $rank_{u,j}$ relates of how far up or down the event j is in the list of recommended events for u, ranging from 0 to 1. If the event is the best recommendation for user u, the percentile ranking will score 0, and if it is the worst scoring event for the user, it will score 1. Percentile ranking have the advantage over absolute ranks in the fact that they are independent to the number of total events. 0,5 of ranking percentile is random strategy. Their testing algorithm is leave-one-out cross validation (loocv), as their dataset is pretty small.

The results of the experiments seem to point out that the most popular in the area is the best recommender, scoring 0.33 in percentile ranking, while TF-IDF scored 0.36, and geographically close scored 0.44. They also do another test comparing results for specific events, showing that these techniques are not always better, but rather context-dependent.

Finally, Quercia et. al. conclude by remarking the importance of mobile data position to event recommendations and how hard is to find large datasets containing geographical information.

Immersive Recommendation: News and Events Recommendations Using Personal Digital Traces

Cheng-Kang et. al. [3] propose a new user-centric recommendation model, called Immersive Recommendation, that incorporates cross-platform and diverse personal digital traces into recommendations. Their context-aware topic modeling algorithm traces from different contexts personal profiles, item profiles and ratings. The focus with this work is to personalize news and local event recommendations.

Cheng-Kang et. al. claim the usefulness of digital traces, and link it to the "interest development theory", which states that people become interested in the field that they are invested in. For that matter, they created Immersive Recommendation, a new user-centric recommendation model that leverages user's diverse personal digital traces from different platforms to make recommendations.

Formally, given a user i, the input of the user profiling problem is a set $N_i = \{n_{i,m} = (c_{i,m}, E_{i,m}), m = 1, \ldots, M\}$, where $n_{i,m}$ represents each instance of the user i digital traces. $n_{i,m}$ can be an email thread, a set of tweets, a set of Facebook posts, etc. $c_{i,m}$ is the context, and $E_{i,m}$ is the content of it. The item profiling has as a goal to transform N_i into a feature vector of the user u_i.

In order to create that user profile, they extracted information from the content using a topic modeling algorithm called Context - Aware Latent Dirichlet Allocation (CA-LDA), which has the advantage of being independent to the platform that is being used. That creates an algorithm that is unsupervised, does not use personal data, and gives better results than other representation learning algorithms, such as *doc2vec* [8].

The basic LDA profile has three steps: Train LDA model with item corpus, use the trained model to infer the topic distributions for each instance $n_{i,m} \in N_i$ and define the user profile u_i as the weighted sum of the topics based on relevance. Cheng-Kant et. al. explain then how this approach has insufficient coverage and is not tolerant to context specific noise (different context use different registers and different keywords: email, words, like, share...). In CA-LDA they co-train item copora and digital traces corpora. CA-LDA assures that all corpora share a superset of "salient topics", but each corpus individually has its own background topic that is associated with the context dependent noise.

As for the recommender algorithm, Cheng-Kant et. al. propose a hybrid collaborative filtering algorithm, called collaborative user-item regression, which is build on top of the foundation of regression-based latent factor. This model introduces two latent offsets, one for user and another for items, which will be tuned in a way that makes up for the difference between user rating and user/item profile relevance, i.e. this model captures the preference information that is missing from the user and item profiles.

For the experiment Cheng-Kant et. al. chose 63.053 "Medium.com" and 50.000 Meetup users that declared their Twitter handle in the profile and crawled their Twitter profiles. The algorithms tested were *doc2vec* [8], LDA [1] and CA-LDA[1]. For the evaluation technique, they chose for each user a set of 10 topics that the user liked and 190 that did not in the case of Medium, and 5 and 95 events in the case of Meetup. The idea is that a good profiling would be able to discern which are liked by the user. CA-LDA outperformed the two other profiling algorithms in every case.

For the recommendation, 5 algorithms were tested: The presented in this work (ImmRec), Content-based, Most Popular, Probabilistic Matrix Factoring (PMF) and Collaborative Topic Modeling (CTM). Content-based is tested to see the effect of a pure content-based approach, Most Popular is the standard baseline algorithm to compare with, PMF makes no use of user or items profile and represents traditional

[1]Context Aware LDA: https://github.com/changun/CA-LDA , 2015

collaborative filtering methods, and CTM is a state-of-the-art recommender that uses PMF an user and item profiles.

Their testing in all recommenders was a 5.000 sample of users, separated in train and test by timestamps. The evaluation metrics used were Average Recall Rate@M : $\frac{\text{number of items the user liked in top M}}{\text{total number of items the user liked}}$, Mean Reciprocal Rank (MRR) which measures the ranking of the first correct item and averages over all the users:

$$MRR = \frac{1}{|I|}\sum_{i=1}^{I}\frac{1}{rank_i},$$

with $rank_i$ being the first correct rank for user i.

The results obtained were that ImmRec outperformed popular, which outperformed CTM and DMF. Finally, Cheng-Khan et. al. also conducted a small user study to explore the utility of Immersive Recommendation in an interactive setting. Two stages were tested: cold start (ImmRec, Popular, Random) and post-cold-start (ImmRec, Popular,PMF,CTM). ImmRec showed superior results in all cases, although CTM and PMF got close in the last case.

Cheng-Khan et. al. finish explaining how the techniques used in their work could be used in other fields and their plans for expanding this algorithm in future works.

A Random Walk Around the city: New York Venue Recommendation in Location-Based Social Networks

Anastasios et. al. [11] present a three-fold recommender that could help alleviate common problems in location recommender systems: using behavioral, social and spatial factors. Their work starts by explaining that the adoption of location-enabled smartphones and new trends to leave notifications of check-ins to friends have led to some services being able to hold huge datasets of user's mobility. In that regard, Anastasios et. al. seek to answer the following questions:

- How ofter do people tend to visit new places? After some research, they discover that between 60-80% of the users check-ins (visits) are to venues never visited in the past month.
- What assumptions do recommender systems make about human mobility? Anastasios et. al. will talk later how each recommender algorithm has its own assumptions, and hence its flaws.
- How can be recommendation quality improved by combining different sources of data? In this work they propose a method based on a random walk with restart which seamlessly and simultaneously combines all the available signals into a high dimensional graph.

Anastasios et. al. collected data from two popular location-based services: Foursquare[2] and Gowalla[3]. They restricted the analysis to the eleven most popular cities across both services. The data is conformed of user, places, check-ins. Gowalla has fewer users than Foursquare for all cities, with the most popular city Austin having 4.008 users, compared to Foursqare's New York which has 16.131.

Anastasios et. al. now move on to explaining the new venue problem: given a sample of check-in data over time t, set U of users, and a set L of venues, they aim to predict $\Psi_i^t = \frac{\Theta_i^{t+1}}{\Theta_i^t}$, where Θ_i^t is all the venues the user i checked in at time stamp t. Thus, the test and train sets have to be hold-out separated by a time stamp t.

[2] foursquare.com
[3] Officially shutdown in March 2012

After that, Anastasios et. al. start explaining all the recommenders that will be tested against the random walk with restart method:

- **Visiting popular venues**: Most Popular assumes that the like hood of check-in is proportional to how many people have checked in before. Users will check only at the most popular places: $\sum_{i \in U} c_{ik}$.
- **Attending venues by category, a content-based approach**: This method creates a vector of user preferences based on a set of categories given by the dataset.
- **Following friends**: Ranks venues by summing the number of check-ins performed by a user's friends at each place. $\sum_{j \in \Gamma_i} c_{jk}$, where Γ_i is the set of friends of i.
- **Staying close to home**: ranks venues by distance to user location; as they do not have that information, they just assume that user location is the most visited venue by the user. This assumes that users like to find options close to where they usually go.
- **Like-mindedness and similarity**: collaborative filtering assumes that historically like-minded people will continue to have shared preferences in the future.

Finally, they present their alternative, a Random Walk Approach. Each method presented until now leverages one unique aspect from the data, but Random Walk can automatically combine each of those features. A Random Walk over a linked structure is based on the idea that the connections between items encode information able to rank them in a useful way. The random walker will transition between the graphs different nodes according to transition probabilities of each node. A known algorithm in this field is the PageRank [**?**] algorithm. In a random walk over a network, the transition probabilities can be arranged in a matrix $Q = \alpha W + (1-\alpha)R$, formed by two factors, a structural one and a random one. W is the transition probability according to the network structure, whereas R models a random probability of jumping to another node. α is the tuning factor, used to change the behavior of the algorithm. Another algorithm that Anastasios et. al. tested is RWR, Random Walk With Restart, which is a sub variant with a small chance to restart from the first point. Another version is WRWR, Weighted (and directional) Random Walk with Restart. A link from user i to user j is weighted as $\frac{1}{|\Gamma_i|}$, inversely proportional to the total of friends. Link from user i to place k is $\frac{c_{ik}}{||\Theta_i||}$, or proportionally to the user's check-ins to that venue over the total number of check-ins for that user.

For the evaluation, they partitioned the dataset into train and test splits of 30 consecutive days. For the metrics, they used Precision@N, recall@N, using $N = 10$ and Average Percentile Ranking (APR).

$$APR = \frac{\sum_{u \in U} \sum_{s \in L} i_{u,s} \times rank_{u,s}}{\sum_{u \in U} \sum_{s \in L} i_{u,s}},$$

$rank_{u,s}$ is a metric that measures how far up the list of preferences for user u the venue s is. When $rank_{u,s} = 0$, means that s is ranked first on the preference list. If $rank_{u,s} = 1$, s is last on the list.

Analyzing the results, Anastasios et. al. comment on how KNN and Matrix Factorization (MF) are not able to surpass Most Popular, while RWR achieves an improvement of 5% up to 18% over Most Popular, depending on the city. Furthermore, the results between the two datasets are consistent with each other.

Anastasios et. al. finishes commenting on how the abundance of social-based and contextual data will be more and more common in the future, and using this data to improve user's recommendation will be a challenge to overcome.

A dual-perspective latent factor model for group-aware social event recommendation

Yogesh and Yi [4] present in their work a dual-perspective latent factor model for group-aware recommendations in EBSN. They modeled two factors that are important to user recommendation: one from the user perspective (e.g. if the event has topics of interest for the user) and the other from the event perspective (e.g. if the events are done in a way that has affinity to the user, mainly event planning and organization). The goal is to utilize group factors to alleviate the cold-start problem without having to use textual content information. As it happens in this work, Yogesh and Yi will be downloading data from Meetup and using RSVP to determine the attendance of users. The objectives of their work are stated as follows:

- Characterize two different complementary perspective of groups, created to alleviate the cold-start problem.
- Proposing a probabilistic latent factor with two sets of latent factors and pairwise learning probability via Logistic and Probit sigmoid functions.
- Make the model flexible enough to add more kinds of data.
- Evaluate the results.

Yogesh and Yi downloaded user, group, and RSVP information from four US cities (New York, San Francisco, Washington, and Chicago) using the Meetup API. For all cities, the number of groups per user is in the range of 7,78 to 9,88 groups per user. They also analyzed data from venues as it could be a deciding factor in the recommendations. Because RSVP is so sporadic and so much more biased towards positive events (users usually do not bother to rate negatively a event), they consider this problem as a top-N ranking. This ranking will be made from RSVP and group data.

In this work, they set up a similar algorithm to Bayesian Personalized Ranking: Formally, (u, i, j) is a preference of item i over j for user u. In this work, the rating preference is a positive RSVP over no RSVP (not seen or the user does not care), which is better than a negative RSVP. They define $x(u, i, j) = s(u, i) - s(u, j)$, with $s(u, i)$ being the ranking score of i for u. The pairwise ranking optimization criterion is the log like hood of the observed preferences, which can be defined as:

$$max_{\Theta}\Psi(\Theta) = \sum_{(u,i,j)\in D_s} log\sigma(x(u, i, j)) - Reg(\Theta),$$

Θ is the set of all parameters, $Reg(\Theta)$ is a regularization term to prevent over fitting, $\sigma(x)$ defines probability of pairwise preference, the probability of item i being preferred over j. This function is a monotonically increasing function. This function can be a Logistic function: $\sigma(x) = \frac{1}{x+exp(-x)}$. In that case, the model would be the same as Bayesian Personalized Ranking. Another approach is to model $\sigma(x)$ as a Probit sigmoid function. Formally, users and events are projected into a $f << min(m, n)$ space, m being the number of users and n the number of items. In the most basic form, $p_u \in \mathbb{R}^f$ and $q_t \in \mathbb{R}^f$, and their ranking score is the inner product, $s_{u,i} = p_u^t q_t$. In this group-aware latent factor model (GLMF), this operation uses other functions, factors linked to groups and user preferences. After this point, Yogesh and Yi decide to use stochastic gradient descent, where an update of the parameters is being made for each preference instance (u,i,j). The parameters move in the direction of the gradient with a leaning rate α until convergence as follows:

$$\Theta \leftarrow \Theta - \alpha \frac{\partial \Psi}{\partial \Theta}$$

Preference instances (u, i, j) are extracted from the dataset based on the user ratings. However if the user has only one type of rating (positive or negative), a preference cannot be created. If the user only has positive ratings, a random event with missing RSVP is given a negative RSVP and if the user only has negative ratings, a random event from one of his groups will be given a positive rating.

In the experiments, Yogesh and Yi test a huge number of algorithms, ranging from basic algorithms, to state-of-the-art, to multiple versions of their algorithms with small differences. The proposed dual-perspective group-aware models vastly outperformed state-of-the-art algorithms. Venue-aware models performed the best. Both Logit and probit versions seem pretty competitive. Finally they test an experiment with only new users, to test cold-start performances. Again, a sub variation called GLFM-VDP-logit and probit gave the best results, outperforming all state-of-the-art algorithms in that regard.

Yogesh and Yi end by commenting how their approach improved performance over basic recommender, and how this dual view could be applied to other contexts. They also plan on adding content and social clues to the proposed recommender to further boost the performance.

Hybrid Event Recommendation using Linked Data and User Diversity

Houda and Raphaël [5] propose a novel hybrid approach using both Context-based and Collaborative Filtering RSs. This hybrid model is also enhanced by the integration of a user diversity model designed to detect user propensity towards specific topics. The basis behind this approach is that user decisions are inherently complex and hard to model with a content-based or collaborative filtering alone. Their belief is that a structured event model lets them cope with such complexity, giving a more straightforward way to explore the data. Houda and Raphaël are aware of the problems of event recommendation compared to traditional RS, mainly they comment on RSVP (their dataset is 98% sparse) and the transience of the events. This is why they decided to make the hybrid model, so the content-based component would alleviate these problems. Collaborative Filtering is still necessary as collaborative data plays a huge role in event recommendation.

The principle of content-based RS is to suggest new items similar to those the user liked in the past. The similarity could be computed using the internal descriptions of the items and a distance like Pearson correlation, cosine similarity, Latent semantic analysis or the popular TF-IDF. Houda and Raphaël decided to use the latter. In their work though, this is a little different to most cases as they use Semantic web and RDF datasets. RDF datasets have all the information of an event condensed, including information about the type of event, where it happened, when it happened, and who assisted the event. The datasets were collected from three event web directories: Eventful[4], Last FM[5], and Upcoming[6], and published in the Linked Data Cloud[7]. To compute item similarity, they use a Vector Space Model comparing triplets from the RDF structure, with a triple being < subject, property, object >.

This process projects the data into a 3-dimensional tensor. A non-null weight is assigned to each entry X_{ijp} for each existing triple < i-th subject, p-th property, j-th object >. The representation of event e_i

[4] https://losangeles.eventful.com/events
[5] https://www.last.fm/
[6] https://upcoming.org/
[7] http://linkeddata.org/

is a t-dimensional vector. The similarity is calculated using the cosine distance. Similarity values between events are then used to obtain a ranked list of recommended items, as following:

$$rank_{cb}(u_i, e) = \frac{\sum_{e_j \in E_u} \sum_{p \in P} \alpha_p \beta_p sim^p(e_i, e_j)}{|P| \times |E_u|},$$

e_i is the event to recommend, P is the set of properties, E_u is the set of past events attended by u, α_p is the learned weight of the parameters. The parameters are the ones that are present in the RDF structure of the datasets. β_p is a weight estimated by training methods, and quantifies interest peaks in the user. It is also always $\beta_p = 1$ if the property p is the subject of the event. The peaks of user interest are detected using Latent Dirichlet Allocation [1], a modeling technique using co-occurrence of terms.

Collaborative filtering models co-participation between events, in the traditional sense. Houda and Raphaël modelled not only considering similarity between users, but also the contribution of a group of friends:

$$rank_{cf}(u_i, e) = \frac{\sum_{j \in C} a_{i,j}}{|C|} \times \frac{|E_i \cap (\cup_{j \in C} E_j)|}{|E_i|},$$

where C is the set of co-attendees of the user u_i that will attend the event e, E_i is the set of attended events by the user u_i, and a_{i_j} represents the fraction of common events between the users u_i and u_j. Note that in the first term, the contributions of each co-attendee is taken separately, while in the second is taken as a group.

Finally, the hybrid model has a merge of the two ranking functions:

$$rank(u, e) = rank_{cb}(u_i, e) + \alpha_{cf} rank_{cf}(u_i, e),$$

where α_{cf} is again a learning parameter which will tell the importance of the collaborative filtering component.

To learn all these rank weights, Houda and Raphaël use a linear regression with gradient descent, and then two evolutionary computational methods, a base Genetic Algorithm (GA) and a Particle Swarm Optimization (PSO). In GA algorithm, a population is a set of chromosomes (candidate solutions) and each chromosomes denotes a set of genes. A fitness function ranks each of the solutions. The best solutions are recombined and mutated for the next generation. The fitness function in this case will be precision@N and recall@N. PSO optimization is a similar approach, but easier to implement because it has less parameters.

The dataset has 2,426 events, 481 user, and 12,729 consumptions. The first experiment was done using the same recommender but different stochastic and testing how each method would optimize the parameters of the RS. PSO and GA showed a better performance than linear regression and unary. Recommender algorithms were also tested, and Collaborative Filtering + Content-based vastly outperformed the other variants without CF, more than doubling in Precision and Recall.

Finally, Houda and Raphaël conclude noting how models that contemplate user diversity improve event recommendation. In the future they plan on working on adding more features like popularity and temporal indexing to boost the performances even more.

Conclusions

We chose this selection of works because it offers variety in the approaches they take into the recommendation problem. The datasets chosen for these works were primarily RSVP from Meetup, although a few works decided to use other platforms: Quercia et. al. [13] used mobile location data, Yogesh and Yi [4] used RSVP in conjunction with group data to add more relations to the dataset. All works use Collaborative Filtering as the primary recommender, some used Content-Based factors, and almost all works reviewed included a context-aware signal in one way or another (distance, time, groups, etc.) . In all cases, the final product proposed was always a hybrid implementation of at least two RS together, even more in some examples like Macedo et. al. [9]. In general, it seems like the common consensus is that RSVP data is sparse and must be leveraged with other factors to achieve a good recommendation.

Chapter 3

Analysis, Implementation and Design

In this chapter we will be covering all the processes needed to achieve our objectives for this work: firstly we will be explaining how we used the Meetup API to download different kinds of data needed for the experiments and how we formed the final collection of datasets from said data; after that we will describe the inner workings of a Java Library called *Librec*[1], which we will later use to conduct our experiments.

3.1 Obtaining the Datasets

First of all, we need to know which kind of data we require to gather in order to run our algorithms with. We will be requiring two kinds of data: for the collaborative filtering algorithms, we need information about events, the RSVPs each event had and which users gave RSVP to said events. For the context-aware algorithms, we want to get location and time data related to users and events. Thankfully, Meetup makes all that information accessible through their different endpoints that they have set up in their Application Programming Interface (API)[2], which lets us download all the kinds of data we need and more.

Obviously, with more data we can run tests with better results and better confident rates. This is why we have gathered as much data as possible from the five most popular[3] cities in United States: **New York, Los Angeles, Washington, Chicago, and San Diego**. We selected those cities as they present the highest member count of all and the most activity, but also are separated enough (2 of these are East Coast, 2 are West Coast and one in neither). For all cities, we decided to gather all the data in a 100 km radius, which covers the city and the general area around it, that way we could consider changing the radius to a smaller one if we needed it. We also collected information during a five month period, since November 1st of 2017, to March 1st of 2018.

To download the data, we need to call a endpoint with a set of parameters to specify how we want the query to be made. For each call, Meetup returns one or multiple JSON objects, representing each object returned with all of their attributes as JSON parameters. We stored all the information collected by the endpoints as raw data in JSON format. Among the attributes of each object, usually there is an

[1] https://www.librec.net/

[2] www.meetup.com/meetup_api/docs

[3] According the their own API, using the endpoint 'Cities' (https://www.meetup.com/meetup_api/docs/2/cities/).

ID, which represents each object unmistakably, and we will be linking IDs of different kinds of objects together in order to get all the information of events, RSVP, and users (see Figure 3.1). Given how certain endpoints are set up, the plan that we found worked best is to gather all the events of a certain city, then get all the RSVP related to the downloaded events, and get the users who made the RSVP. With that linking information and extra contextual information that Meetup makes public, we can run both collaborative filtering and context-aware RS.

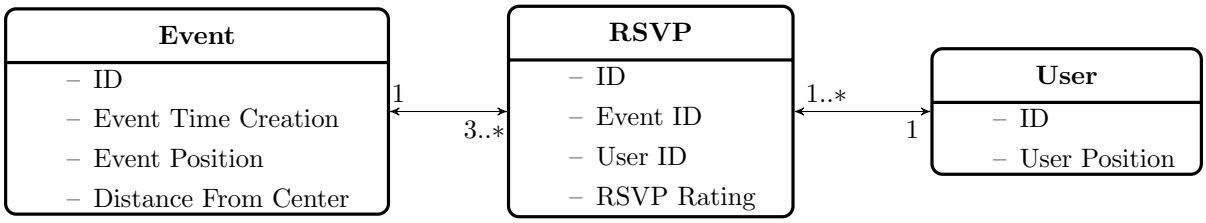


Figure 3.1: Basic types of data and their relations.

Meetup has a well documented, easy to use API where anybody with a API key can make requests to their server and retrieve information[4]. There are multiple endpoints, each one with different names and different HTTP methods, and require a different set of parameters. To make it easier for us, we wrote a small program that automatizes calls to endpoints. Because we want to retrieve data from their end, we will be only using GET methods. The list of methods we will be using is the following:

1. **Open Events**[5]: This endpoint lets us download all the events given a city, a radius and a window of time. This method is the core of our dataset, and will allow us to capture all the other information we need, but has a few drawbacks: it only returns information about events with at least 3 RSVPS, so the datasets will not have unpopular events. Also, the time of the events can only be one month in the past at most, so in order to gather large quantities of data we must do frequent and periodic calls to this endpoint. On the bright side, it lets us download exactly the data we want in a quick fashion: the endpoint supports up to 200 events for each call. Each call also has a link to the following query with the remaining results if there are any. This can go on until it reaches a max of 10.000 events, so we made each call encompass a whole week instead of a month, because in some cities retrieving one whole month could have surpass that maximum and lose events past the 10.000 mark. The information retrieved from this endpoint was really useful, as it gave not only the id of the event, but also additional information as description, distance from the center of town, position and time of the event which will be used in context-aware RS.

2. **RSVP**[6]: This method returns the list of RSVPs of a given Event. The information included in the response has the ID of the user who did the RSVP, the event identifier, and the result of the RSVP (positive or negative). Basically, this method gives us the collaborative information we want in the dataset: User - Event - Rating. Along with the Events endpoint, we can combine the two and get all RSVP of a chosen city.

3. **Members**[7]: Returns information from a given user ID. Although it gives us interesting data, more than anything we want to gather information about the approximate position of users, and this is the only way to get it as the only information that we get in the RSVP endpoint about the user is the user ID. This information will be used in context-aware RS.

[4] https://www.meetup.com/meetup_api/
[5] www.meetup.com/meetup_api/docs/2/open_events/
[6] www.meetup.com/meetup_api/docs/ urlname/events/ event_id/rsvps/#list
[7] www.meetup.com/meetup_api/docs/2/member/#get

In order to make our work as efficient as possible, we created a small *Java* project responsible for making automated calls to the different endpoints, saving the responses, and later we will explain how the program also restructures and filters our datasets. Without going into much detail, the program has a few classes that hold the same attributes as the Queries in the API, and we would instantiate and fill the attributes needed, and create an URL holding these parameters to be send to a GET endpoint:

```
https://api.meetup.com/2/open_events?&page=10000&country=US&state=CA
&city=Los+Angeles&time=1522540800000%2C1523210400000&radius=100
&status=past&sign=true&key=******************************
&photo-host=public&limited_events=false&text_format=plain
&format=json&order=time
```

Figure 3.2: A basic query example that returns all the Events of Los Angeles during a week in a radius of 100KM

After creating the URL, we need to send it and process the result. We would first open the connection and execute the query stored in the URL by making the connection, and then storing the response. First of all, we need to check the *responseCode*, as it holds information of how the query went. If the code is incorrect (i.e. 403, 404, 410, 500, etc) we retry the connexion a few times in case it is a connection problem on their end. If the problem persists, means that Meetup API does not let us get that information. That could be by a couple of reasons, but usually means the data is private and cannot be viewed. In these cases, we do not have an option but to delete the original data, as we cannot extract anything from it.

3.1.1 Problems encountered

Although Meetup API gave us all the information we needed, there were a few problems that we had to deal with when collecting data. Firstly, some of the ID retrieved by the endpoints had inconsistent format: most of the time were numbers, but from time to time we had strings of characters representing the same types of ID. That made us take the decision to convert the ID from their system to a new system that took every new ID and gave it a new number, starting from 1. Secondly, some calls to the API were impossible to make because of privacy reasons. When an event is private, it has reduced information shown, and we could not access RSVP information of that event, so after some testing we decided the best was to delete the original event also. Doing so, we make sure that the information stays consistent across all data types. Finally, as we mentioned, we downloaded events in bulk, but after that, RSVP and user information had to be downloaded by ID, which means that we needed to make one call to the endpoint for every event or every user in the dataset. Meetup API, like most other API, has a limit of number of calls that can be made from the same API key in some amount of time, so we had to set up a timeout of 1 second to make all the calls we needed to the system without having the API key banned or temporally disabled.

3.2 Data Structure

After downloading all the necessary information using Meetup API, we ended up with 100 data files containing Event information and 100 data files containing RSVP information. Each one for every week (for simplicity, 1 week = 1/4 of a month) of the 5 months collected for every one of the 5 cities included

in our dataset. It is important to note that all the information that we got by the Meetup API is in JSON format. In order to read and save in that format, we used JSON parser libraries[8]. Using a Java project, we operated the data by reading, crossing, filtering and saving it in different formats with the final objective being to create a final group of datasets containing all the information needed in a specific format that *Librec* could understand.

The data processing consists basically in parsing and extracting relevant data from the JSON structures, filtering data, and crossing it with the three types of data we have: User ID, Event ID, RSVP Rating. Also, we need to do a similar process with the extra data we will be using in context-aware RS. In order to keep track with all the information, we named the files containing different data:

1. **Events JSON File**: This file contains all the gathered information from the "open_events" endpoint as it was downloaded. The format of this file is JSON. Every JSON object in this file contains information about an event, such as an internal ID, position and time of the event, description of the event, group that created the event, etc. The information might vary between events, sometimes due to privacy levels. Although the data has some interesting data, like information about the group that created the event or venue information, we primarily want three attributes: one is the internal ID of the event which is used to download their RSVP information, another one is the distance of the event to the center of the city, and the last one is the position of the event. The first will be used in collaborative filtering, while the other two will be used in context-aware datasets.

2. **RSVP JSON File**: Same as the previous file, this file also contains all the gathered information from a endpoint, in this case the "RSVP" endpoint, as raw text, in JSON format. As we said earlier, all the events are guaranteed to have at least three or more RSVP. Each JSON object in this file contains a list of all RSVP, always indicating the ID of the user who made it, the rating of the RSVP, and the event ID. As these datasets have all the collaborative information, we will primarily work with this data file to form the final collaborative filtering dataset.

3. **Filtered RSVP DataFile**: After we have all the data together, we can start structuring it the way we want. First of all, we need to apply a filter to all the events and RSVP to only select events that are inside a specific distance radius. Doing this will allow us to make experiments with varying distances and to filter out far away events we do not consider relevant. Since information about location does not appear in the RSVP JSON File (see Figure 3.1), we had to read all Event data, and using the convenient "distance" attribute, which states the distance of the event location to the center of the town, we filtered out all the events from the events datafile and consequently all the RSVP related to said events, and the result is saved in a processed RSVP DataFile. The format of this file is still JSON.

4. **Processed DataFile**: A Processed DataFile is the name we gave to a datafile that has the format to work with the Java library *Librec*. This file can be synthesized from all available RSVP JSON File or Filtered RSVP DataFile, as the format is the same in both files. It gets the event ID, user ID and rating for each RSVP and the time from the corresponding event and saves it in this file with the format [userID,eventID,RSVP,Timestamp]. In the *Librec* documentation, this format is called *UIRT*, which stands for User - Item - Rating - Timestamp, and it is accepted by the library for all the algorithms implemented. The information saved here is not exactly the same as it was in the JSON files, though. We talked previously on Subsection 3.1.1 how Meetup API has inconsistent formats when it comes to identifiers, so we created our own internal ID, that assigns to every unique user or event a new ID starting from 1. Also, we saved RSVP assigning a $+1$ to a positive rating, and a -1 to a negative one.

[8] github.com/stleary/JSON-java

5. **Splitted Processed DataFile**: One final optional step we can take on the structuring of the datasets is to split the datasets by density. This idea was first introduced to us in one of the works reviewed in Chapter 2 by Macedo et. al. [2] , where they do experiments on different sections of the datasets depending on the number of ratings per user. With more ratings per user, less sparsity of the dataset. We also did that, and for that matter we have for each city 7 sections with different level of ratings per user, which from now on we will call **Splits**. Each split has a subset of all users that has a specific number of ratings in the dataset, which are the following:

$$[\,1\,,\,2\,,\,3\,,\,4-5\,,\,6-10\,,\,11-20\,,\,>20]$$

   The splits are selected so they roughly have a similar amount of users, although it depends on the city. Each one of the datasets are also ordered by the last parameter of each line, the timestamp of the time of the event. This will be useful later when running the tests.

6. **Item Position DataFile**: For the context-aware algorithms, we need to load extra data from items and users into the algorithm inside the *Librec* library. *Librec* does not have context-aware implementations, but inside the library we can set up a algorithm that after loading the collaborative filtering Datafile also loads location data. That way the algorithm will be able to factor both information in the way it chooses to. So, we need to have a collection of datasets that has the relation of new event ID - event position. Thankfully we already had this information in the datafiles saved. The Events JSON File can hold 2 attributes called “lat” and “lon” inside the information related to the venue the event is in. This information is not always present, but if it is we can just save that information in the new file. In the case that it is not present, we can just skip it for now and make an exception inside the context-aware RS for missing information.

7. **User Position DataFile**: With the same idea as before, we need a second group of datasets that holds contextual information about the user to load in the contextual-aware RS. In this case it is a little harder, as user position information does not appear in any of the data we had stored. The only way to obtain it is downloading it from Meetup API (“members” endpoint), parsing the position as latitude and longitude coordinates, and save the relation between new user ID and coordinates.

3.3 Librec

Librec is a open-source Java library for recommender systems aiming specially at the problem of rating prediction and item ranking. In its latest version 2.0 *Librec* implements a high repertory of state-of-the-art recommender algorithms, similarity functions, data splitter algorithms and ranking metrics, and is relatively easy to implement one of your own (Figure 3.3). Also, each one of the algorithms are highly configurable in its parameters.

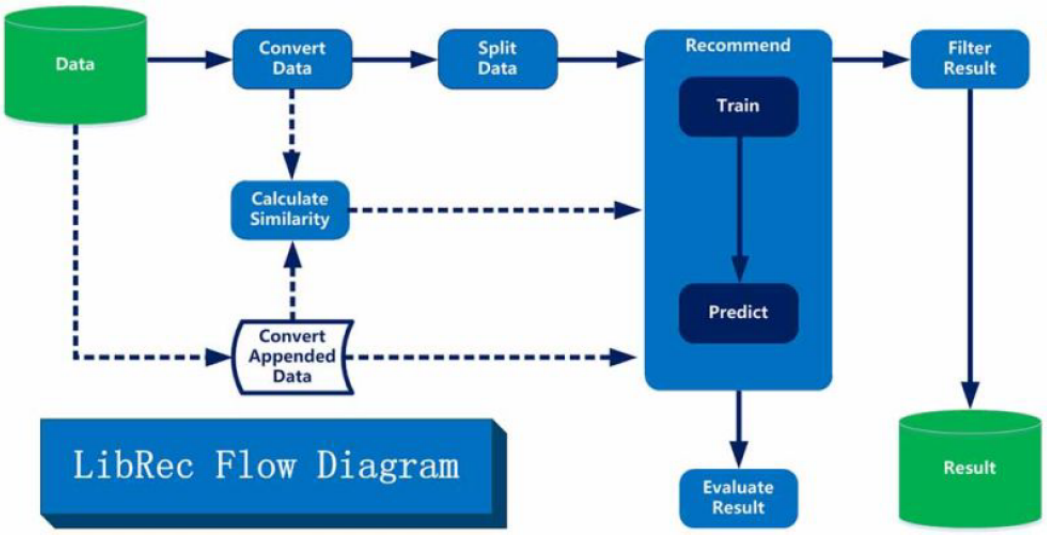


Figure 3.3: *Workflow of Librec.*

For each execution, *Librec* needs a configuration file. A configuration is a file of format *.conf* that holds all the information needed for the execution. Even though the parameters can be changed inside the code, most of the time we will load all the configurations parameters via configuration file as it is easier. A configuration file holds the following information:

1. **File paths**: path of the folder containing all the data files used in the execution, paths of the output and log files.

2. **Data Model parameters**: Type of the data model (Text or ARFF). In this work we will only be using Text as ARFF is only compatible with a few algorithms. We can also define the format of the datafile, which in our case is UIRT (User-Item-Rating-Timestamp).

3. **Recommender algorithm**: Name of the class that will be used as recommender in the execution. The name can be either the full class name or an abbreviation to the name. Alongside the name of the algorithm, a set of parameters related to the algorithm can be defined in the configuration file. The full reference of the full list of parameters and algorithms possible is described in their official webpage[9].

4. **Splitter algorithm**: *Librec* supports a variety of cross-validation algorithms that let us split the data between train and data: holdout, K-fold, leave-one-out, givenN, etc. Each one with a few unique parameters: for example, we can choose the ratio of train-data in the holdout or the number of partitions in K-fold.

5. **Similarity function**: only used in a few recommender algorithms, a function similarity will rate the how close two users or two events are to each other. Some of the options *Librec* has are: Pearson's correlation, Cosine Similarity, Binarized Cosine Similarity, etc.

[9]https://www.librec.net/dokuwiki/doku.php?id=AlgorithmList

6. **Metric function**: After the execution, *Librec* will output the results in the form of metrics. Metrics measure the performance of the algorithm, using different criteria. Because the problem presented in this work is a ranking classification problem, the metrics that we will use will be classification metrics rather than regression metrics. Some parameters can also be defined for the metrics, like whether it is a ranking problem or the top N classifications to take into account.

7. **Other parameters**: Parameters of the general execution of the program, like the random seed of the execution.

Librec also puts a large quantity of recommender algorithms to our disposal. Among these algorithms we can find naive solutions, like recommending at random or always giving the average rating in the dataset, and also some more advanced algorithms. *Librec* has an abstract class called *AbstractRecommender*, which is the class where all the recommender algorithms in the library extend from. This means all the algorithms have a few steps in common: each recommender starts with the function *setup* being called, then *trainModel* and finally *predict*. In the setup, the algorithm starts loading parameters from configuration files, initializing parameters and the train model, which is a matrix created from the datasets which is $|U_{tr}|$ by $|E_{tr}|$ in dimension, U_{tr} and E_{tr} being the users and events in the train model. In some cases, additional matrices are load to help in the prediction process. After that, the algorithm predicts a score for each new user and item based on the trained model and its prediction function.

Firstly, the Collaborative Filtering algorithms we will use in this work are the following:

1. **Random Guess**: In this experiments it is common practice to add simple and naive algorithms to have baseline reference points to compare the algorithms with. Random Guess is always supposed to be the worst algorithm. Before the execution, the algorithm defines the minimum and maximum ratings of all the dataset. For each pair of user - item, returns a random value between the maximum and minimum rating:

$$predict(u, e) = random(minvalue, maxvalue)$$

   This algorithm is extremely fast as it only initializes two variables in the setup, does not train the model, and the prediction function is trivial.

2. **Most Popular**: Same as Random Guess, Most Popular is usually also added to the tests to show how a "base" or simple algorithm that takes a simple approach. This algorithm only takes into account the popularity of an event. Once the rating matrix is fully loaded, this algorithm counts how many RSVP each event has. Then, for each pair of user - event, returns this count as the rating of the pair:

$$predict(u, e) = |R_e \neq 0|,$$

   with R_e being the column of the event e in the train split. Note that this recommender only uses the item data to compute the result, not user data, and does not train the model in any way as it does not need that information.

3. **User KNN**: User KNN is one of the most common and standard recommender algorithms. Usually called user-based collaborative filtering, the idea behind this algorithm is that users with similar tastes and preferences might like and dislike things similar to the target user. Basically, this algorithm tries to predict how useful an item is to a certain user by how well other users similar to him rated it:

$$predict(u, e) = \sum_{j=1}^{m} \alpha_{u,u_j} R_{u_j,e},$$

Where α is the similarity function, which takes two users and computes a value between 0 and 1 depending on how similar the two users are. The intuition with this formula is that we value opinions directly proportional to the similarity to the user we are trying to compute the rating. Because it relies on the number of ratings, this algorithm is affected by the sparsity of a dataset: with more ratings the datasets have, the higher the chance of having similar users that also have rated different event. Choosing a good similarity function is pretty important to the performance of this algorithm. From all the repertory of similarity functions that *Librec* lets to our disposition, we choose to use the Binarized Cosine Similarity as the one we will be using for all our tests. This function is a specific implementation of the Cosine Similarity, which uses the Euclidean dot formula to process a value of how close two vectors of ratings are to each other:

$$similarity(u, e) = \frac{v_1 \times v_2}{\sqrt{v_1 \times v_2}\sqrt{v_2 \times v_1}}$$

This metric works well when the vectors are sparse, as only the non-zero dimensions need to be considered. We also have the advantage that our ratings are only positive or negative, so this is an implementation that fits the context. Some test also showed that this similarity function yields better results than other similarities like Pearson's or Jaccard's.

4. **Item KNN**: Item KNN is the same algorithm as User KNN but with the reversal of the roles of user and item. Instead of recommending items by similarity of users, this algorithm recommends computing a similarity between two items:

$$predict(u, e) = \sum_{i=1}^{n} \alpha_{e,e_i} R_{u,e_i},$$

α is the function similarity on items. This change of users to items can be beneficial in some cases if the datasets have more ratings by user than by item. Similar to User KNN, this algorithm still suffers from a sparse dataset.

5. **Bayesian Personalized Ranking (BPR) [14]**: A state-of-the-art recommender optimized for ranking where each user has a likehood function created from the Bayesian probability function. Instead of using scores $S(i, u)$ of the pair item-user, BPR uses probabilities given by the function $P(i > j; u)$: the probability that user u prefers item i over user j. Unlike other rating algorithms, BPR does not train over user-item ratings but rather trained using a function of 'preference' which needs a user,item,item triples. This function of preference is based off the rating of item i and item j:

$$P(i > j; u) = \sigma(S(i, u) - S(j, u)),$$

with σ being a sigmoid function that smooths the results: $\sigma(x) = \frac{1}{1+e^{-x}}$.

6. **Biased MF (MF) [6]**: In general Matrix Factorization decomposes the train model into three matrices. The goal is to create a dimensionally reduced representation of all user and item vectors $\vec{p_u}$ and $\vec{q_i}$, which instead of being in the user and item space, are in the reduced feature space. The final rating is computed by the three factorized matrices: $R = P\Sigma Q^T$, where P is called the user-feature affinity matrix, $\sum$ is a diagonal feature weight matrix, and Q is the item-feature relevance matrix.

7. **Singular Value Decomposition (SVD++)** [**7**]: SVD++ is a Matrix Factorization model, in fact one of the most widely used models. SVD++ integrates the implicit feedback information from a user to items and the user latent factors are complemented by the latent factors of the items to which the user has provided implicit feedback. *Librec* offers a wide variety of Matrix Factorization models, and we decided to choose two that were selected in some other works reviewed: in this case, both Biased MF and SVD++ were showcased in Yi Fang and Yogesh Jhamb [4] dual-perspective approach.

The second set of tests that we will do is related to context-aware. *Librec* does not have any kind of contextual-aware recommender by default, so we had to create our own implementations. Context-aware are going to have almost all the same properties as the last ones, except of the algorithms implemented in this test are going to be new algorithms implemented into the the *Librec* library and will load extra contextual information by the side. The implementations of the contextual-aware RS are the following:

1. **Distance Aware Recommender**: This recommender uses location data from both users and events in order to compute a rating. The location information will be loaded by the side at the time of setup by the two contextual files presented before: the user location data, which has approximated location of all the users that attended any event, and the event location data, which has the declared position in the application. We explored different approaches for the prediction function: The most simple approach would be computing a rating that would be inversely proportional to the distance between user and item. This approach, while valid, would not get good ratings for users that attended to some events from far away. The second implementation we did would get all the previous events the user RSVP'ed to, get their positions and computed distance to the user. For each pair of user - event, returns a rating:

$$predict(u, e) = \frac{1}{\sqrt{1 + mean(D_u) - distance(u, e)}},$$

   Where D_u are the distances of E_u, all the events that the user RSVP'ed to. Basically, instead of getting the distance user - event and computing the rating with that, we decided to get the average distance of all events, and see how close that distance is to the event we are trying to get a rating on. Doing so, we made the algorithm invariable to the threshold of distance traveled to attend an event: the rating would be equally good if it was from a user in the city or from a near town. After that, with just computing a function that makes the rating proportionally inverse to the distance was enough, but adding a square root made the values smoother, and it gave slightly better results. This algorithm is an original implementation.

2. **Time Aware Recommender**: Similar to Distance Aware Recommender, this recommender uses time data from events to make predictions. We use timestamp information of the events to get the hour and day of the week the event happened. We think this view of the information is the one is going to give us the best performances. After that, we define a representation of the event $\vec{e}$ as a $24x7$ matrix which has a 1 whenever that event happened in that day of the week and hour of the day. We can define the representation of u as:

$$\vec{u} = \frac{1}{E_u} \times \sum_{e \in E_u} \vec{e}$$

   Once we have $\vec{u}$ and $\vec{e}$, which are the matrix representations of both the user and event we want to rate, we use cosine similarity:

$$predict(u, e) = \frac{\vec{u} \times \vec{e}}{\sqrt{|\vec{u}|}\sqrt{|\vec{e}|}} = \frac{\vec{u_{ij}}}{\sqrt{|\vec{u}|}}$$

The simplifications can be made because $\vec{e}$ is a matrix of zeros and a single one in position i, j (day and hour of the event), and $\sqrt{|\vec{e}|}$ is 1.

This recommender algorithm is an implementation of Macedo's Time Aware Recommender [9], and we found it really interesting approach and yielded better results than other alternatives.

3. **Hybrid Recommender**: Alongside the two said algorithms, we are also testing how an hybrid algorithm between one of the context-aware and non context-aware solutions might work. The concept behind is, even thought context-aware algorithms may contribute to the prediction of events, they lack information about user and item correlations. It seems logical to mix the two in a single recommender, and see if they offer better results. In this recommender we will be having a factor called α which will be dictating how much of each recommender we take in the final prediction:

$$predict(u, e) = \alpha \times recommender_1(u, e) + (1 - \alpha) \times recommender_2(u, e)$$

The first recommender will be either distance-aware recommender or time-aware recommender and the second recommender has to be a collaborative filtering recommender. We think User KNN can be a good fit as it is the most common in this kind of problems and it is generic enough.

In order to make *Librec* work as we want to, we only have to call a method called *runJob()*, which makes a full execution of the code, with a certain configuration file loaded. The full execution will starting by reading and generating a data model partitioning the data into train and test, selecting and operating whichever recommender and similarity function was selected, and computing the result with some metrics. After running the recommender, *Librec* will output the results of the algorithms given a set of metrics. We will be collecting said outputs for each algorithm city and more factors in order to compute the results.

Chapter 4

Experiments

In this chapter we will showcase the final datasets and analyze some properties of them, explore the methodology of the experiments, covering all the algorithms and techniques being used in each step, and at the end we will show the final results and comment on them. Studying the data will give us an idea on how to approach the tests, as well as some preferred parameters. We are also going to compare the differences of both approaches: collaborative filtering versus context-aware, exploring their datasets and algorithms separately.

4.1 Data

In this section we will show the final datasets obtained from the Meetup API that will be used in both Collaborative Filtering RS and Context-Aware RS and analyze them. From the collaborative filtering dataset, we are more interested in the number of ratings between users and items, and what can we do with them to combat the ever present sparsity that affects all datasets. In the contextual datasets, we will explore the spatial and temporal distribution of the events and users in order to decide if they are interesting factors to study.

4.1.1 Collaborative Filtering data

The dataset that we are going to use in collaborative filtering has information about all the events gathered during a 5 month periods, the RSVP rating of those events, the users who RSVP'ed any event and the timestamp in which the event began. After collecting and computing all the data, we put together all the results to make a quick summary of all the cities, as we can see in Table 4.1.

City	Users	Events	RSVP	Sparsity
New York	221802	62304	902448	99.993%
Los Angeles	152358	54722	630420	99.992%
Washington	124099	35132	476359	99.989%
San Diego	93709	34586	389989	99.987%
Chicago	69027	19419	278829	99.979%

Table 4.1: *General Information about the datasets (100KM).*

Out of the five cities, New York has the most data by a wide margin in all categories, followed by Los Angeles and Washington, and with some distance by San Diego and Chicago. Comparing this information with city population, we can affirm that Washington users are the most active across all cities, as Washington has the lowest population of all the cities studied in this work but it is the third city in terms of data in the dataset. Chicago is the least active, as it is the third city in the US by population but the fifth by data here. Apart from that, the data seems pretty standard, the cities with more RSVP also have more Users and Events in the dataset.

RSVP Sparsity

The results in Table 4.1 shows that we gathered a large number of users and events, but we found that even though the number of RSVP ratings is larger than the other two, is not is not that much higher in comparison, which might indicate that each user and event does not have many RSVP linked to them. To formalize this intuition, we computed the levels of sparsity for each city and found out that the levels were too high, with the lowest of them all being Chicago at 99.979%. We calculate sparsity with the following formula:

$$Sparsity = \frac{|R|}{|U| \times |E|} \times 100,$$

with R being the set of RSVP, U being the set of all users, and E being the set of all events, all that by each city. Sparsity gives us a understanding of how many relations are between users and events: what is the chance of any user i to not have a rating of any item j? As we can see, most of the time we will not have that kind of information. Seems like users usually do not bother to give RSVP to an event if they do not feel strongly about it. This has huge repercussions in any RS using collaborative data, as it hinders the performance. Collaborative filtering RS benefit a lot of having a dense dataset, because with less ratings by user and item the RS cannot find similar users or items to the one we are trying to recommend. Making a generalization, the large majority of users rate a small percentage of events, as we can see in Table 4.2.

City	RSVP	Avg. RSVP/User	Avg. RSVP/Event
New York	902448	4,069	14,485
Los Angeles	630420	4,138	11,520
Washington	476359	3,127	13,559
San Diego	389989	4,162	11,276
Chicago	278829	4,039	14,359

Table 4.2: *Average RSVP per user and events in all cities (100KM).*

Despite this information, we know that some users and events might have different RSVP usages. Instead of showing the average values, we can also look at the total distribution of RSVP per user or item (Figure 4.1 and Figure 4.2).

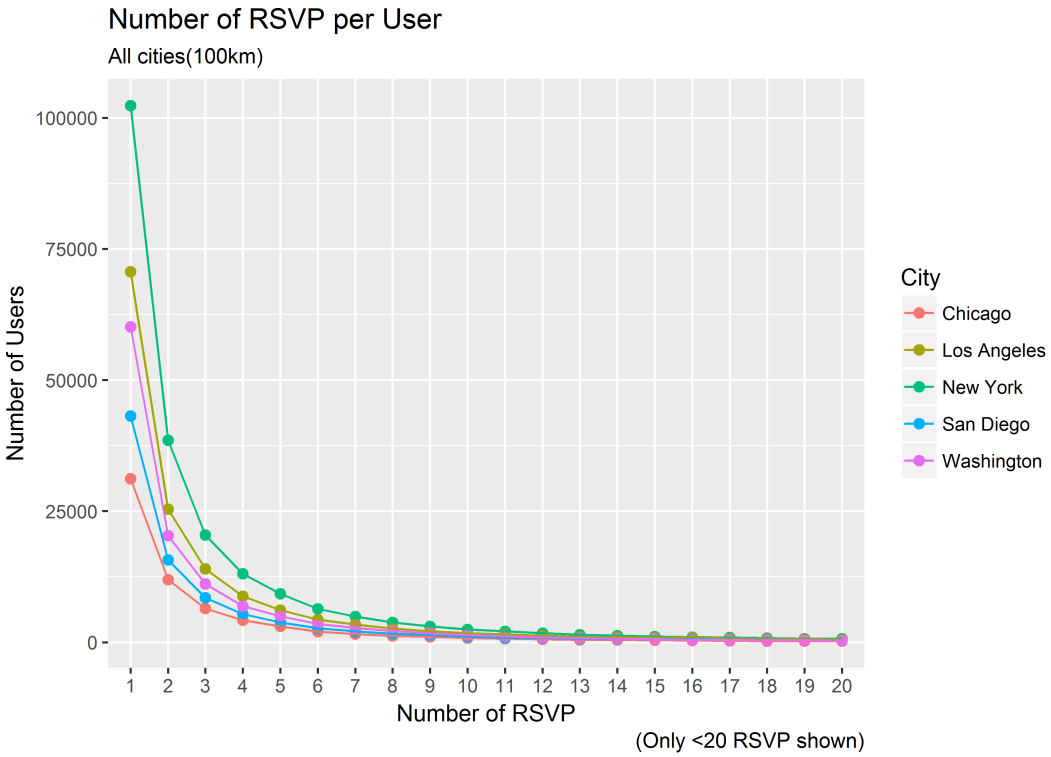


Figure 4.1: *Amount of RSVP per User in different cities. (100KM)*

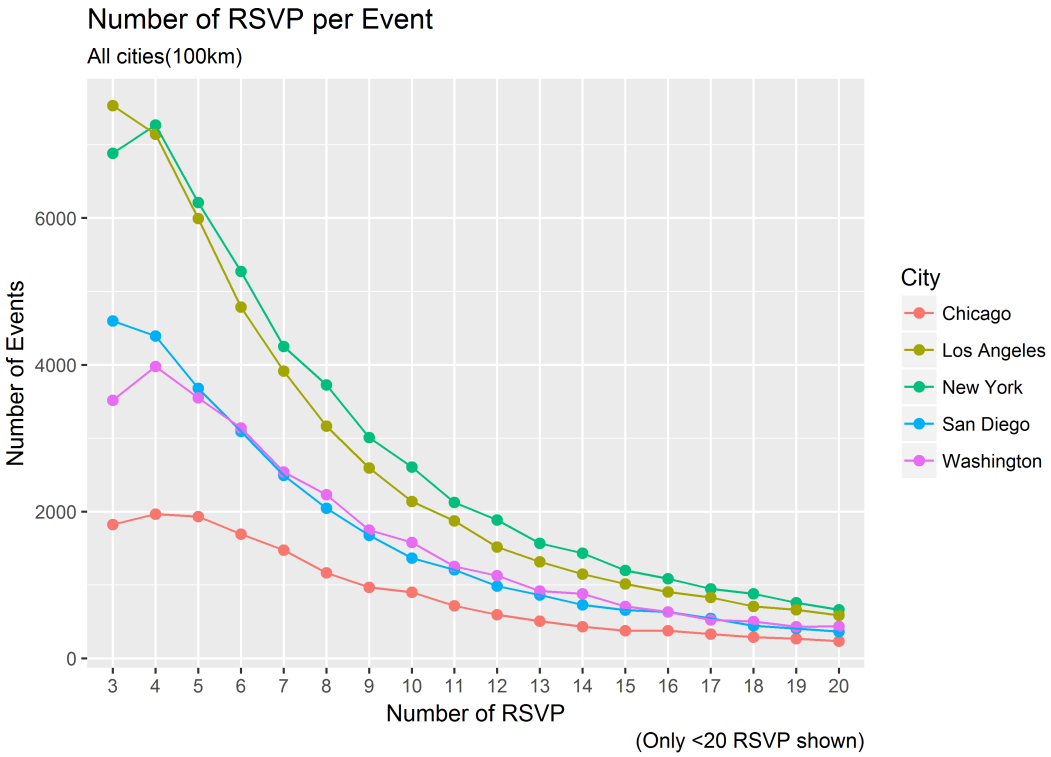


Figure 4.2: *Amount of RSVP per Event in different cities. (100KM)*

In all cases, seems like the function of users over RSVP follows a exponential distribution, which means the large majority of users only RSVP a few of events in the course of 5 months. The second graph shows

a different distribution, starting from 3 because of the restrictions with Meetup API that made that we could only retrieve events with at least 3 RSVP. Some of the cities have a peak at 4 RSVP per event, but in general seems to go down with more RSVP and has a more lineal relation than users, which shows that there are more popular events than users.

The circumstances when users decide to RSVP events are also interesting to study: according to Macedo et. al. [2], RSVP occur more close to the date of the event, in some long-term events even having more than 80% of ratings in the last 20% of their life times. When it comes to recommending, users have a bias: most of the ratings observed are positive RSVP. One explanation for this phenomenon could be that users that are not interested in a event could just simply not RSVP as it is non-obligatory, only doing so in events presented by certain groups or in certain events where the number of assistants is important. Figure 4.3 shows how different cities have their rating usage.

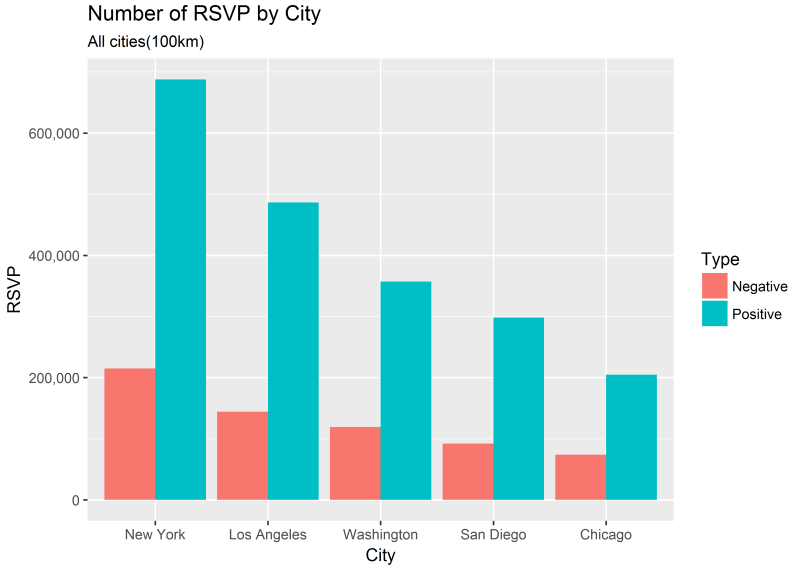


Figure 4.3: *Distribution of positive and negative RSVP in different cities. (100KM)*

Figure 4.3 shows the usage of both positive and negative RSVP for each city. In all cities, positive ratings seem to be much more common than negative, making up to roughly two-thirds of all ratings at 100 KM. Even though we see that each city has different amounts of RSVP ratings, in all cases seems like RSVP bias is consistent. According to Macedo et. al. [2], RSVP bias changes with distance, and while in some cities explored this was the case (in New York we found a change of 10% in events at 10KM and 100KM), in most cities we could not draw conclusions from the data.

Size of the dataset

In traditional recommender algorithms, adding more data to the dataset most of the time means better recommendations because the algorithm has more relations between users and items on average on the dataset to learn from. As we talked before, this is not the case in our work and in general in EBSN, because of a problem that we talked before which is the transience of events. Events, unlike other items in traditional RS, have location and time, which complicates the recommendation problem. That makes adding data way less helpful because of two main reasons: The first is that adding data from larger windows of time (temporally) would not actually make the dataset more dense, as new users cannot make relations (i.e. RSVP) with old events. In particular, adding extra information that occurred after

the date of the dataset will not add new RSVP to any existent event. It can add more information of the users though, as users have a longer timespan than events and adding data can potentially affect the same users again. So adding data temporally is still useful, just not as much as in traditional RS. The second reason is that adding data by distance (adding data from a bigger radius from the center of a city) also will not make the dataset denser. In the same sense as before, adding more data from the external area of the city will not create many relations with existing users and events, because each user usually tends to attend events close to them, and most of the population cannot virtually attend every event. In some cases can be even detrimental to have this data, as most of the time there are less events and users in the surroundings of a city, and it only adds noise to our recommender algorithms and lowers the density.

For example, if we filter from the datasets the events made only in a 25Km radius of the center of the city, we would get a collection of smaller datasets as shown in Table 4.3.

City	Users	Events	RSVP	Sparsity
New York	152546	35546	580785	99.989%
Los Angeles	92771	29427	357121	99.987%
Washington	89982	22551	337114	99.983%
San Diego	37502	12103	148491	99.967%
Chicago	52028	12178	198993	99.969%

Table 4.3: *General Information about the datasets (25KM).*

The sparsity of the datasets is lower than in Table 4.1, even though we reduced the number of RSVP by roughly 60%. One reason why sparsity might be less in these datasets is that having less distance lets the user attend to more events in that area, unlike in the 100Km. Another reason could be that users closer to the center of the city are more prone to go to events inside the 25Km radius, making the average of RSVP per user higher. Given this reasoning, the more we shrink the datasets the better sparsity levels we should get. We can see this effect in Figures 4.4 and 4.5.

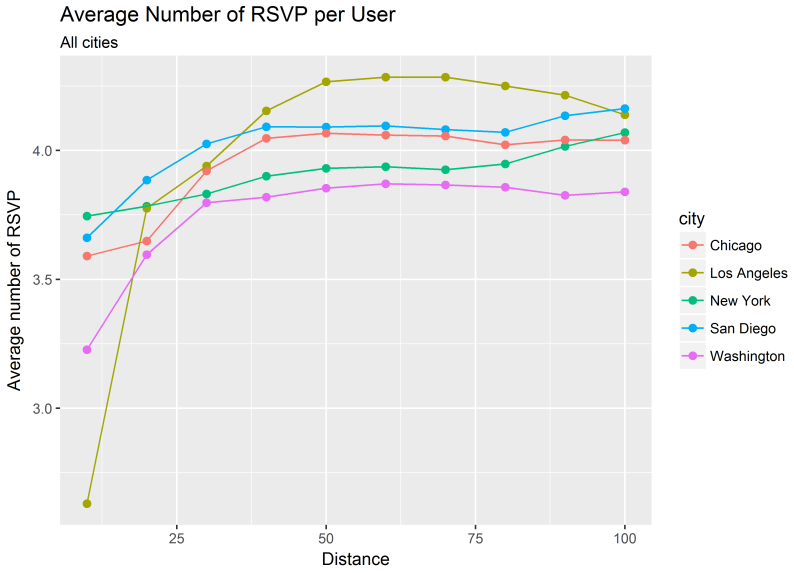


Figure 4.4: *Amount of RSVP per Event in different cities. (25KM)*

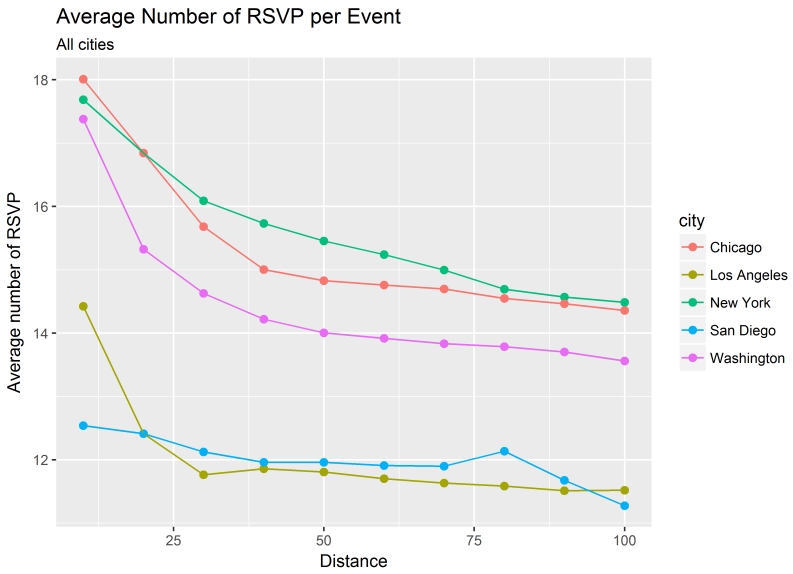


Figure 4.5: *Amount of RSVP per Event in different cities. (25KM)*

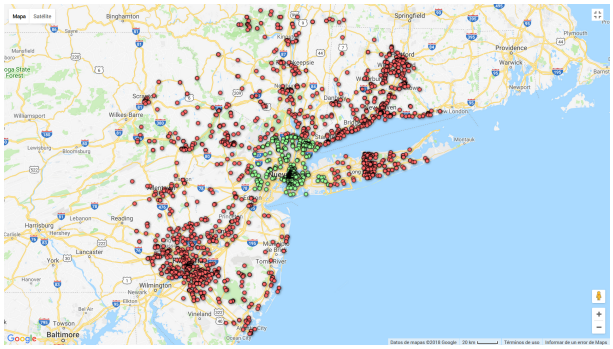
As we can see, the average number of RSVP by user and event seem to be affected by distance and the two of them seem to go in opposite directions: while events seem to have more RSVP the closer they are to the center of a big city, users tend to RSVP more the farther they are to the center. This information seems to point out that events in the city seem to be more popular and that users that live in the same city do not RSVP as much as users that live far away, probably because users do not bother to RSVP to a very close event or it is a given, while far away users feel that confirmation of attendance is more important. Also, the average of RSVP per event seem to be much lower in the two cities that are from the west coast (Los Angles, San Diego) than in the other cases. In the end, it is still better to go down in distance as users are bigger in number in the datasets. The two trends do not seem to be linear for most cities, and looking at the data seems like a good point to shrink the data would be at some distance between 25 and 35 KM, where averages tend to stabilize. In this work, we will be using filtering events from 25 KM, as they are easier to process.

4.1.2 Contextual data

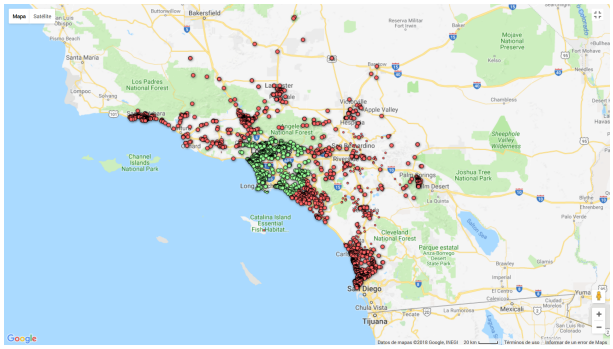
Apart from the collaborative filtering, we will also explore context-aware options, and for that we need contextual data from both users and events. As we said previously, we will study two main factors: distance and time. For distance, we computed another collection of datasets that hold user and item positions. For time, we already have that information inside the collaborative filtering datasets. We talked briefly in the Section 3.2, but that datasets have 4 parameter by line: User - Item - Rating - Timestamp. That last timestamp will be used in our time-aware RS.

Location Data

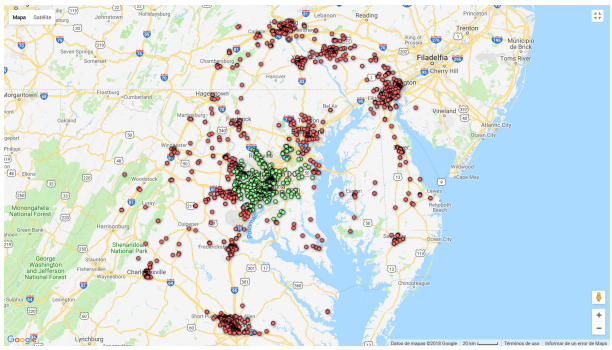
For distance-aware RS we will use two datasets: one containing user location, and another using event location. Both locations are saved in the datasets as a pair of latitude and longitude coordinates. To get an idea of the distribution of the data, we can plot it into a map, like in Figure 4.6.



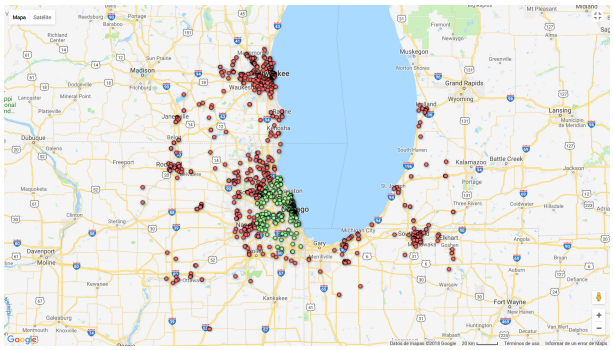
(a) New York



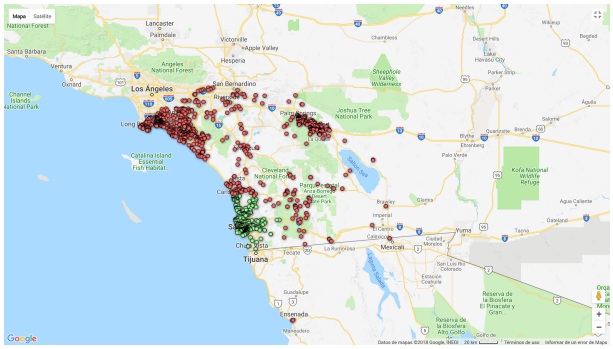
(b) Los Angeles



(c) Washington



(d) Chicago

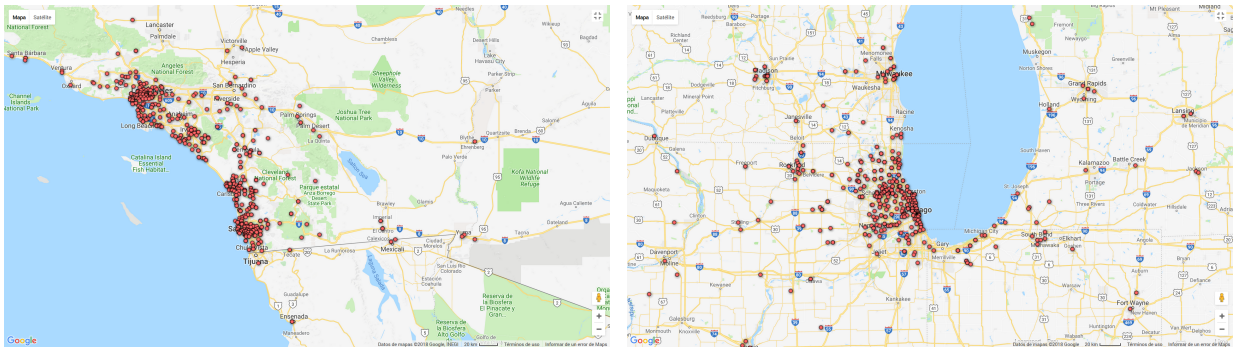


(e) San Diego

Figure 4.6: *Spatial distribution of events across all cities (100KM and 25KM). Green points mark the events presents in the 25KM dataset.*

The representations give us a good estimate of a few factors. Firstly the difference in number of total events are very visible on the map, with New York being the best example. The distribution of events across all cities also show where events are created primarily: while the center of town usually has the best density of events, in some cases cities or areas around the city have big densities of events too. That is another reason to keep the maximum distance close to the city: having a bigger radius would make the algorithms mix up events from different nucleus of events. The worst offenders are Los Angeles and San Diego, which are actually so close together that they have events from one another, and in the case of San Diego loses events in 100KM because of it being so close to the border. Keeping the datasets at 25KM (only the green points) keeps the datasets inside the center of the city for the most part and not mix the datasets with other close towns.

We can also do the same representation with the user location information we gathered. Unlike event positions, Meetup only declares said positions as inferred approximations of the user's home location, as users do not have to inform of that information in the platform. As we talked previously in Section 3.2, user location data was not present and it was needed to be downloaded from an extra endpoint. Because that operation takes time, we only have that information about two cities: Chicago and San Diego. This also implies that the context-aware RS that we will test can also be used in said cities. User location can also be plotted into a map, like in Figure 4.7



(a) San Diego (b) Chicago

Figure 4.7: *Spatial distribution of users across all cities (100KM).*

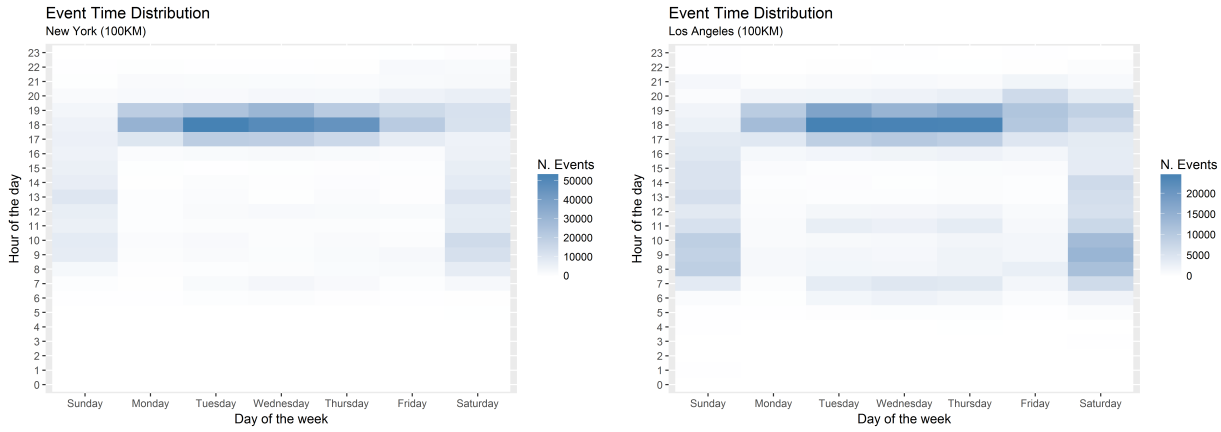
As we can see, the distributions in these cases show a more scattered dataset than the events did. Actually, there is a non-trivial amount of users that are from really far away, for example the opposite coast of US, Europe or even Asia. It seems understandable that during a 5-month period users from different parts of the world would attend at least one event, for example going on vacations or for business reasons.

All the representations have been made using a Google tool called *Fusion Tables*[1], which lets us plot a large quantity of points in a map and apply all sorts of filters. The datasets introduced to make the representation has three values: latitude, longitude and type (whether the point was inside the 25KM radius or not) in case of event locations.

Time Data

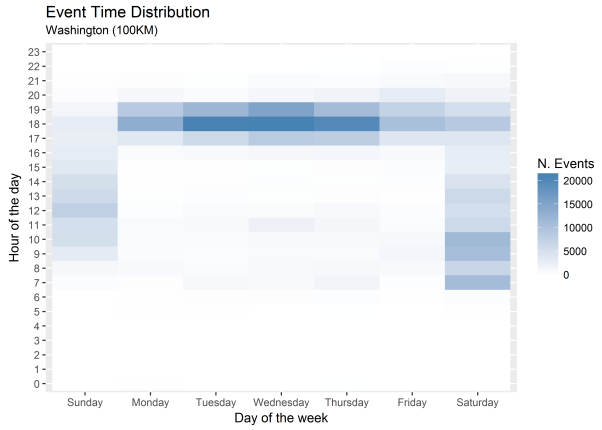
The other context-aware option we will study in this work is time-aware. The data needed in time-aware RS can be obtained inside the same collaborative filtering datasets described in Subsection 4.1.1. The last parameter in each RSVP is the timestamp of the start of the corresponding event. One useful way we can explore data of the events is extracting the hour of the week of the timestamp. According to the Meetup API, that timestamp represents the time of start of the event. We are going to dissolve the timestamp, into day of the week and hour of the day to explore the weekly and daily patterns of users (Figure 4.8).

[1] https://fusiontables.google.com/

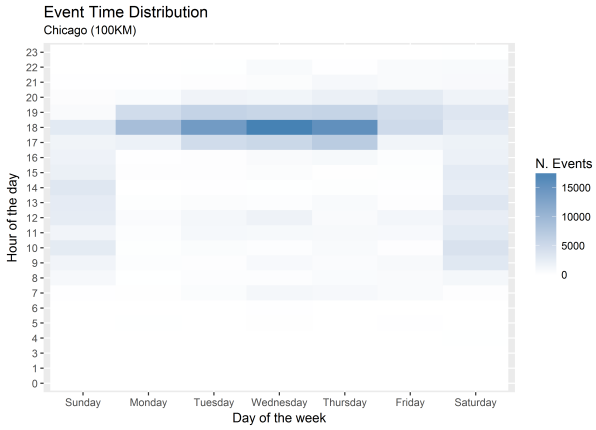


(a) New York

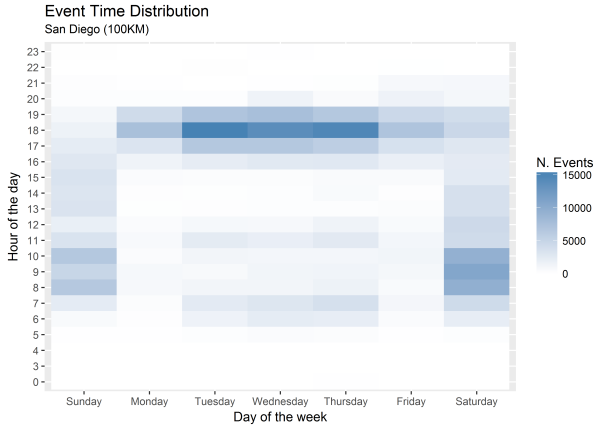
(b) Los Angeles



(c) Washington



(d) Chicago



(e) San Diego

Figure 4.8: *Temporal distribution of RSVP across all cities (100KM).*

Analyzing the data we can conclude that most events (and the most popular ones) are made after work hours in work days (5PM through 7PM seem the most active) in all cities, specially in Tuesday, Wednesday and Thursday. In weekends, events are scattered through all the day, although in some cities seem to be a trend towards events in the morning (8AM through 10AM). One reason the peaks are higher in weekdays is that people who work in the city tend to stay around to do events after work hours, while in

the weekend users prefer to go out of the city, with only a fraction of the users staying. In general, these plots show strong trends across all cities.

4.2 Methodology

The problem presented in this work is a ranking problem: we are not so interested in the classification of the events (whether an even was classified good or bad for the user), but rather that the best events are ranked best for the user. Each one of the algorithms presented will try to find the best N events for each user, in other words, given a set of previous ratings by the user, will compute a *topN* for each user. The application of this problem would be the equivalent as opening the Meetup Application and seeing a list of "recommended for you" events. Instead of only recommending good events in no particular order (which could be a long list), we want to compute a concise list of the best events, so the user can filter the best options for them and move on. As some of the works reviewed, we are going to compute a *topN* ranking of $N = 20$.

The main objective of the experiment is to determine and compare how certain algorithms perform under different circumstances: principally, we are going to test each recommender in every one of the 5 cities separately and also for each city we are going to split each dataset into 7 levels of sparsity, called Splits. We are going to compare a set of 7 Collaborative Filtering algorithms which use the same information as input but use that data in different ways to compute the results, and 4 Context-aware which will use extra datasets. In between all those implementations, there are going to be naive, traditional, and state-of-the-art RSs, as seen in Section 3.3.

Each one of the algorithms presented in this work is going to be tested for different reasons: in the case of Random Guess and Most Popular, we want to see how simple algorithms would perform with our datasets: while we expect Random Guess to be the worse algorithm, Most Popular has proven to be better than some algorithms shown in Chapter 2. Then we present some typical Collaborative Filtering algorithms such as User and Item KNN, that will represent traditional RS. The rest of algorithms (BPR, SVD++, Biased MF) are state-of-the-art algorithms that have proven to be better than traditional Collaborative Filtering algorithms, but not in contexts like EBSN. BRP is actually optimized for ranking problems, so in theory it should perform okay, while SVD++ and Biased MF are variants of Matrix Factorization and we do not know how they will perform.

In order to make our tests as realistic as possible, we will test the algorithms separating the train and test set temporally. Temporal data is all linked implicitly, and reordering or randomizing the data can break the relation and skew the results. Keeping the order will allow us to test the prediction capabilities of the algorithms based on previous ratings by the user. In the train portion of the dataset we will have $N-1$ ratings of each user, which will serve to train the algorithm with the preferences of each user, while in the test portion we will have the last rating of each user by date. This way we can test our algorithms in a way that mimics a real case scenario: the algorithm should be able to predict based on last ratings of the user. Other approaches used for this problem are splitting the dataset by a certain timestamp, which could leave users entirely on the train or test portion of the dataset. We think this approach is correct and interesting paired up with the splitting by sparsity of the datasets. Because each split of the datasets have users who have a certain amount of ratings, the number of ratings of each user that will stay in the train set varies on each split:

$$\text{Splits} = [\,1\,,\ 2\,,\ 3\,,\ 4-5\,,\ 6-10\,,\ 11-20\,,\ >20]$$

$$\text{Trained Ratings} = [\,0\,,\ 1\,,\ 2\,,\ 3\,,\ 5\,,\ 10\,,\ 20]$$

As we can see, there are a few problems to this: the first one being the first split. The first split only holds users that during the 6 months have made a single RSVP, so we cannot separate the ratings in train and test in this case. In this split we will be using a holdout splitting algorithm which separates the datasets in 80% train and 20% test at random. The second thing to note is that in splits with more than one number of ratings per user $(6-10, 11-20, >20)$ we have to choose one lower than the minimum value in the range. Otherwise, we would have users entirely in the train set.

In ranking problems, there are a few metrics to compute the performance of algorithms, but the most common one and the one we will be using in this work is NDCG (Normalized Discounted Cumulative Gain), which is an advanced version of DCG (Discounted Cumulative Gain). The premise of DCG is that highly relevant events to the user should appear on top of the ranking. This metric penalizes in cases where this does not happen, using the following formula:

$$DCG_p = \sum_{i=1}^{p} \frac{rel_i}{log_2(i+1)},$$

rel_i is the graded relevance of item i. NDCG uses a logarithmic function to give more value to the first results in the topN ranking. The results this metric gives is heavily dependent on the list length, as the logarithmic function is not scaled with the length of the ranking list. NDCG fixes this by normalizing the weight of each value in the list:

$$NDCG_p = \frac{DCG_p}{IDCG_p},$$

where $IDCG_p$ is the Ideal Discounted Cumulative Gain, which has the following formula:

$$IDCG_p = \sum_{i=i}^{|REL|} \frac{2^{rel_i} - 1}{log_2(i+1)},$$

which is the same formula as Discounted Cumulative Gain, but instead of adding elements by some arbitrary order, it does so by order of relevance. This means that Ideal DCG is the maximum value of DCG, and in a perfect ranking algorithm $DCG = IDCG$, which would yield values of NDCG of 1.0.

Although NDCG will be the most important metric for us and was used in some works reviewed previously on Chapter 2, we will also be using Precision@N and Recall@N. While not as useful as NDCG to measure the overall performance of an algorithm, will give us insight on their classification. We define $Precision@N$ and $Recall@N$ as follows:

$$Precision@N = \frac{\text{Number of items the user liked in Top N}}{\text{Items in Top N}}$$

$$Recall@N = \frac{\text{Number of items the user liked in Top N}}{\text{Total number of ratings by the user}}$$

Given the nature of the experiments, usually we will have only a few ratings of each user in the test set. NDCG and Recall are invariant to the quantity of items in the top N ranking, but Precision is expected to be low, as most of the time the best case scenario will be $\frac{1}{20} = 0.05$.

4.3 Results

In this section we will show and comment the results of the various experiments we performed during this work. We will try to make educated guesses on the factors that explain the performances of each algorithm for each city and split.

4.3.1 Collaborative Filtering

We will comment on the results in **Figure 4.3.1**. For this result, we are comparing all 7 Collaborative Filtering algorithms. Every algorithm has an unique color line, which is explained in the legend at the top. The X axis represents the splits we previously described, each one having diverse sparsity levels. All the different cities are represented side by side, while each one of the three different metrics is being represented in each row of results.

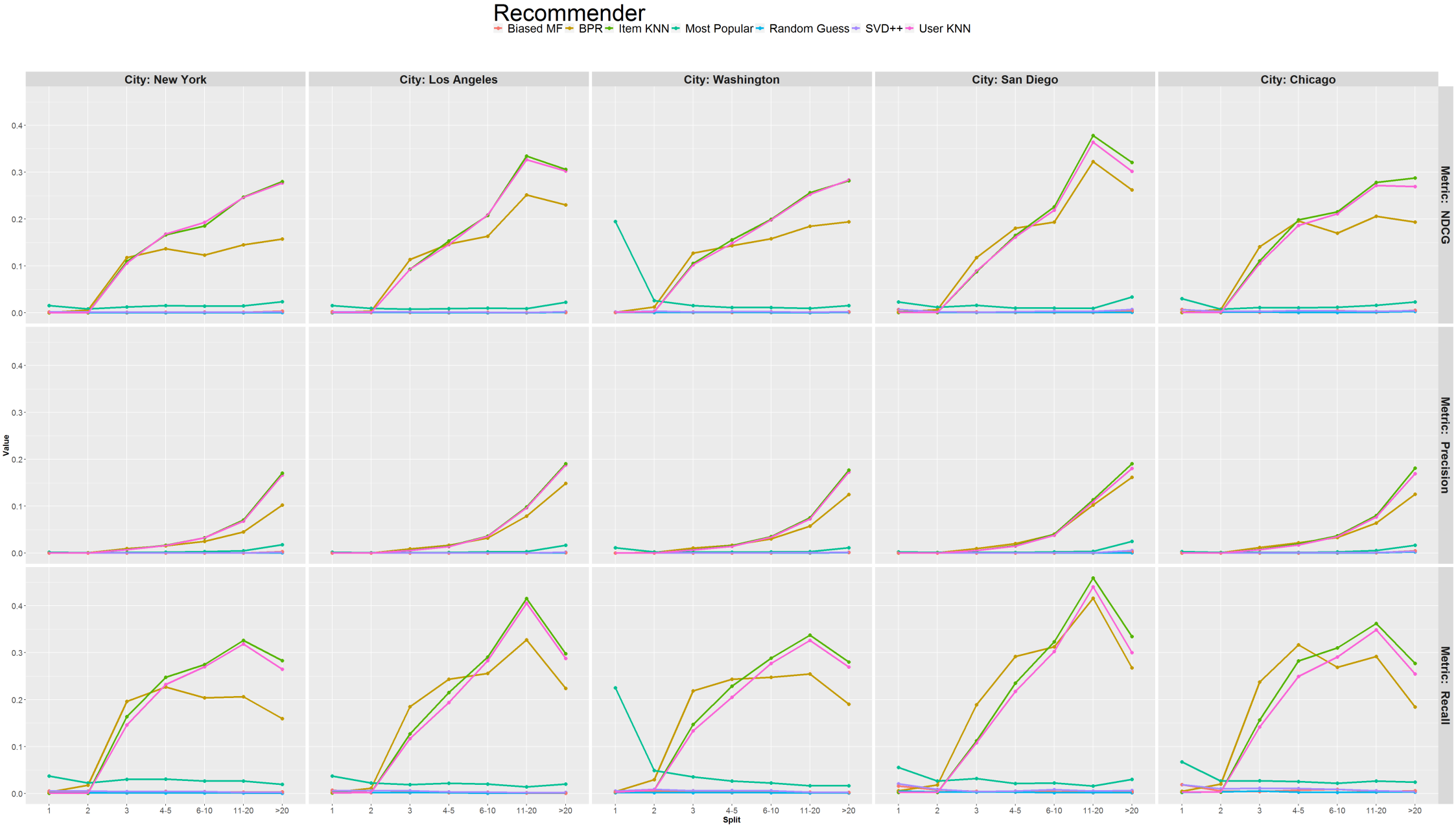
As we can see, there results create two clear groups of recommender algorithms: the ones that surpass 'Most Popular' (User and Item KNN, Bayesian Personalized Ranking) and the ones that do not (Biased MF, Random Guess, SVD++). In general, the trends seem pretty consistent across all cities, with only a few exceptions really perceptible like 'Most Popular' first split on Washington. That was due an 'out lier' event which had an enormous popularity: the wreath replacing day at Arlington cemetery, which had the astounding number of 13.471 RSVP. The two cities in the west coast (Los Angeles and San Diego) have also an interesting trend of peaking abruptly at the $11 - 20$ split, and also show the best performances of all cities.

In all cities we can say that all algorithms have better performances with lower sparsity levels or at least maintain it. This is expected, as higher splits mean we have more data to predict each user with, and collaborative RS like User or Item KNN work better in those conditions. All the metrics show improvements, and in specific NDCG seems to do the best with the changes in sparsity.

The algorithms that seem to do best with these datasets are User KNN and Item KNN, specially at higher density. Although KNN are generic algorithms, they do not seem affected much by the low sparsity levels. Also seems that user and item KNN are pretty interchangeable in terms of performance, with no algorithm being always on top of the other. BPR is the only contender, showing great performances in the first splits but then not increasing as much with more data. BPR also is a lot more faster, so we could make the observation than BPR is preferable with higher sparsity levels.

Most Popular does not have a great performance, but stays consistent through all the splits. It has a paper being the absolute best algorithm with almost no RSVP information, like in the first and even second split. This behavior makes sense, since Most Popular is not going to be affected by sparsity levels, because it does not use relations between users and events, only the number of RSVP of items.

On the other side, we have a few algorithms that could not surpass Most Popular in terms of performance: Biased MF and SVD++ seem to perform rather poorly in this scenario. Our first guess would be that sparsity is too much for the algorithms to handle, and that forces the algorithm to factorize the matrix without the needed information. In general, Matrix Factorization algorithms yield usually the best performances in traditional RS, but seems like in EBSN we cannot achieve that level of density that the algorithms would want.


Recommender
Biased MF
BPR
Item KNN
Most Popular
Random Guess
SVD++
User KNN
City: New York
City: Los Angeles
City: Washington
City: San Diego
City: Chicago
Metric: NDCG
Metric: Precision
Metric: Recall
Value
0.0
0.1
0.2
0.3
0.4
Split
1
2
3
4-5
6-10
11-20
>20

4.3.2 Context-Aware

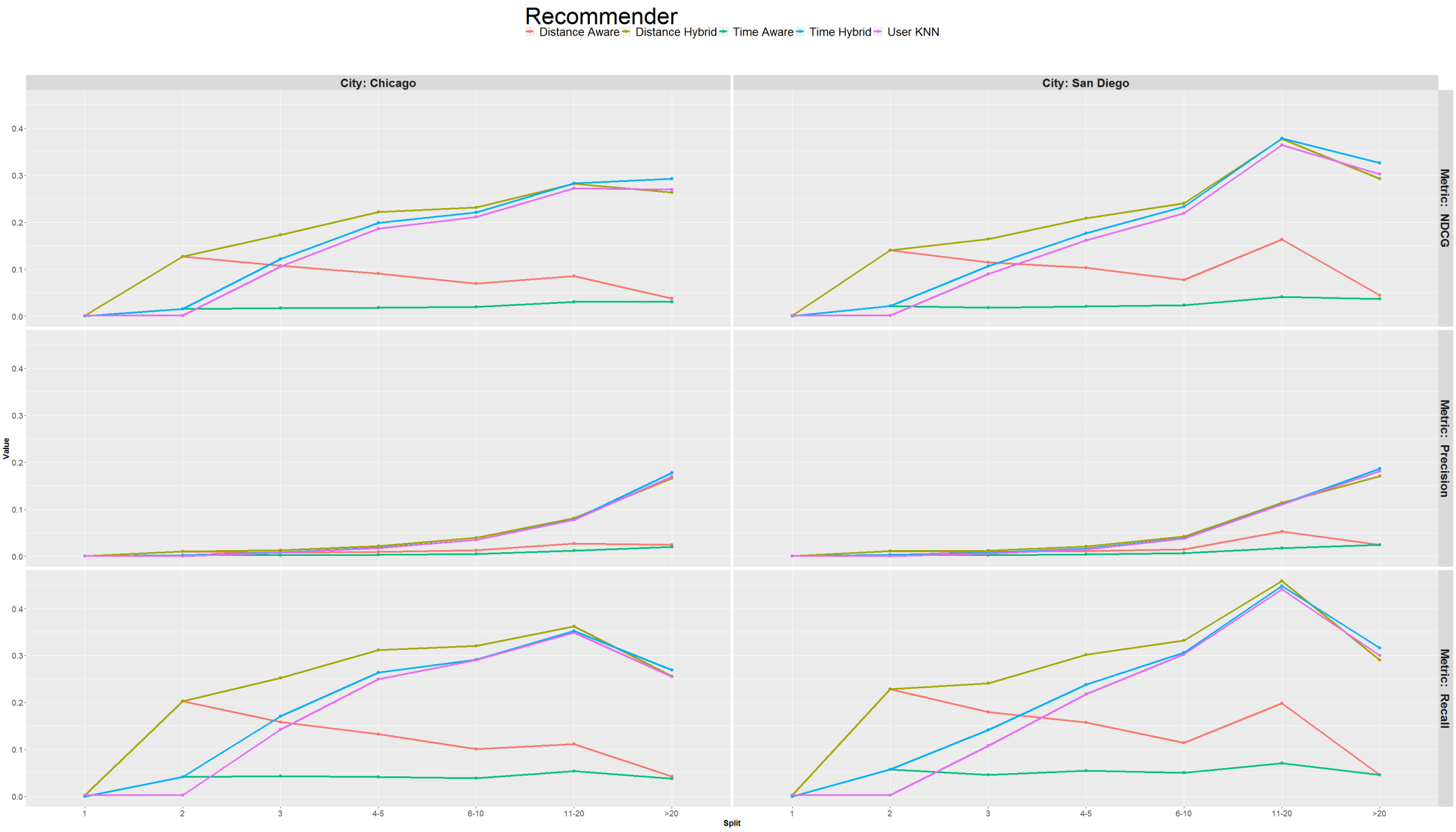
Now we will on the results shown in **Figure 4.3.2**. For this result, we are testing the algorithms under the exact same conditions as the experiment before, the only difference being that now we are testing context-aware RS. We are going to test the effectiveness of the two contextual aware factors: distance of the event and time of the event. We are also going to show the performance of two hybrid RS, using each one of the two contextual-aware algorithms and a basic collaborative filtering that also adapts pretty well to any situation: User KNN. The hybrid version depends on one parameter α, which is usually decided using gradient descent.

The plot shows shows the base and hybrid versions of each context-aware RS, and also User KNN as a point of reference. User KNN has the same values as it has in Figure 4.3.1.

The results show a consistent trend: contextual algorithms, while better than some collaborative-filtering, still do not outperform User KNN in most cases. The main problem with it is that each contextual algorithm only exploits one single factor of the recommendation problem, which is a lackluster approach. In theory, adding more and more factors should yield better performances than by separate. Only analyzing the base algorithms, seems interesting how Distance Aware does very well at early splits, but then Time Aware gets close to it. The same trend also appears on their hybrid versions.

Hybrid models do outperform consistently User KNN, specially at the first splits. This confirms the intuition we had that context-aware and collaborative filtering can have a constructive relation in a hybrid environment. While collaborative filtering has strong relations in the data, context-aware can help fill the gaps, as RSVP is really sparse and most of the time we will not have that information for each pair of user and item. However, we can have contextual information for almost all users and items, which can differentiate between items with subtle differences when RSVP information is not present. After doing some tests, we found that the best parameter of α is around $\alpha = 0.85$. The fact that the algorithm found this as the best value means that the best configuration of the two algorithms is to have Collaborative Filtering have the lead whenever can offer something, and contextual aware help in the cases where it does not.

This effect can also be seen in the different splits of a same city: in first splits is where hybrid heavily outperforms User KNN. User KNN and collaborative filtering in general struggle the most to recommend items in the first splits, as it lacks rating information. Context-aware does not, so combining the two approaches seems good because it creates a good all-around RS that can predict better in all situations.


Recommender
Distance Aware
Distance Hybrid
Time Aware
Time Hybrid
User KNN
City: Chicago
City: San Diego
Metric: NDCG
Metric: Precision
Metric: Recall
Value
0.0
0.1
0.2
0.3
0.4
1
2
3
4-5
6-10
11-20
>20
Split

Chapter 5

Economic Cost

In Economic Cost we are going to make an approximation of the cost of the research done, based on the different phases the work has passed and the number of hours dedicated to each task. In terms of time done on each task, we can differentiate a few parts:

- **Investigation**: Exploring the inner workings of the Meetup API, understanding the differences between endpoints and deciding which data to use in order to create the final product. We also include figuring out the inner workings of the *Librec* library so we could understand the implementations of the algorithms and the general process in the library to add extra algorithms, and also time reading related works and other documentation.
- **Coding (Analysis and Implementation)**: Creating a setup to download all the data for us in an automatic fashion, which included having a constructor of Queries, that could send and receive, and also manage errors. After that, creating a project that could save, filter, cross the information in order to form the final body of datasets. Also, the time coding the implementations of the contextual-aware algorithms.
- **Librec tests (Analysis)**: Testing and configuring the experiments that we showed. This include the selection of all the techniques selected for each work and the reasoning behind the tests.
- **Code executing (Implementation)**: while this could be done while doing other tasks, it also took a large quantity of time, specially in two process: downloading the data of RSVP and User locations and computing the final results using *Librec*.
- **Documentation**: This includes creating this document and also the Annexes. It also includes the time making the images that go along with it.

The exact time spent in each task can be a bit blurry, but we can make somewhat accurate approximations based on some information:

During all the time working on this project, I tried to follow a setup that included doing at least 16 hours each week split usually in two days, although at the start and during exams weeks I could not follow through. I started the investigation around September 2017, and around November I started downloading the first data. Investigation of Meetup and deciding which data to download was the lengthier part of this process. During last of 2017 and first of 2018 is where I started coding, and testing *Librec* settings. Coding consisted in testing formats of data and testing how the library could process it. The executions made in this work took a long time: Meetup API has restrictions that force each user and RSVP to be downloaded individually and with a maximum number of calls per minute. That made the operation

Task	Hours
Investigation	100
Coding	250
Testing	80
Execution	250
Documentation	60
Total	**740**

Table 5.1: *Distribution of Hours in this work.*

really lengthy: at the bare minimum, we can account for a single call for every event, in order to get their RSVP, and a single call for every user in the two cities we collected user location data. That accounts for at least 737.798 calls, one being made each second. Without taken into account connection and processing time, the bare minimum time is 205 hours of gathering data. On top of that, counting the time of the *Librec* executions, can easily add up to 250 hours and even more (tests, final results, A/B testing, etc). The only redeeming part about that number of hours is that I could be doing more work on the side while that was going on.

After approximating work hours, we can compute the economic cost:

Task	Hours	Price/Hour	Total Task
Investigation	100	4	400€
Coding	250	8	2000€
Testing	80	4	320€
Execution	250	1	250€
Documentation	60	1	60€
Total	**740**		**3030€**

Table 5.2: *Total economic cost of this work.*

Testing, Investigation, and Documentation have reduced price per hour as they are easy work, and execution even less price as it only needs a capable computer and electricity to do the job. The total ascends to a little over 3.000€, reasonable price for a few months of work.

Chapter 6

Conclusions and future work

In this work we explored the field of work that is EBSN and non-traditional social networks in general. We gave an introduction into the subject as well as objectives and the reasons we had to tackle this problem. We explored and gave insight to the problems of Recommender Systems and in special EBSN, and counter measures to help alleviate said problems. The focus of this work was to generate research rather than find a new approach of the problem. Collaborative Filtering is a type of RS that is has been established as the best type of recommender in EBSN, and the number of works that pair that with another type of RS such as Context-Aware or Content-Based is quite large, as we have seen in Chapter 2.

As a research job, we feel that we explored all the important factors in EBSN recommender systems. We covered common problems like "cold start problem" that happen in most RS, to some more advanced problems like the transience of events or sparsity of the data by time and distance. Historically there have been a few works that covered some properties of EBSN, such as RSVP bias and RSVP ratings by distance, but as far as we know we also covered more properties than other works. With RSVP data being so sparse, new inventions have to appear in order to overcome the challenges of the new context. The results obtained point out that hybrid approaches and context-aware RS seem to perform better in almost all cases, and that sometimes traditional RS were not the answer.

Before starting this work, we had three main points that we wanted to address in this work: downloading and creating a collection of datasets to make experiments using Meetup API was the first one. This resulted to be the most time-extensive objective, as we wanted to gather large amounts of data, and also figuring out how to cross the data in order to create the datasets and write the Java project. We ended up with a lot of transformations of the datasets that helped us tackle different problems such as the contextual datasets needed in contextual-aware RS. In future work some more work could be done in transforming the datasets in other formats or even could be changed into other systems like DataBases for example.

The second main objective we proposed at the start of this work was to be able to compare the effectiveness of different algorithms in different contexts such as different cities and different sparsity levels. This objective was also accomplished with the help of *Librec*: with the flexibility it offers we could download data from different cities and different splits of it and let the library process and compute the results, which was a time saver, considering that the tests in itself were time-extensive.

We also took the time in our work to analyze some properties of the dataset, which we regard as important to understanding the context of EBSN. We think that a good understanding of the context of the cities and

the datasets helps us understand and give meaning to the result we can later find. Some of the aspects we analyzed were lead by some other works similar to ours, and some others were simply interesting information that appeared during the exploration of the data.

Finally, we also explored context-aware and hybrid approaches to the same problem and compare the results of all the approaches. We even found hybrid alternatives that offered better results that collaborative filtering alone could offer. We would have liked to explore even more context-aware approaches, but sadly we did not have that much time to add more factors. In future work, we would much like to explore even more approaches to the recommendation problem, such as content-aware or group-aware, and probably add a logistical regression model merging all the approaches into a single multi-aware Recommender System.

All and all, we think we accomplished successfully all the objectives we set to ourselves for this work. We think that the problems presented in EBSN are a important barrier that RS have to cross in order to be available to a lot of new services that would benefit from it. EBSN is still a new and exciting field that will for sure see some development in the future.

Bibliography

[1] D. Agarwal and B.-C. Chen. flda: matrix factorization through latent dirichlet allocation. In *In Proceedings of the third ACM International Conference on Web Search and Data Mining*, page pages 91–100, 2010.

[2] Augusto Q. de Macedo, Leandro B. Marinho, Federal University, of Campina, Grande Federal, University of Campina, and Grande. Event recommendation in event-based social networks. 2014.

[3] Cheng-Kang Hsieh, Longqi Yang, Honghao Wei, Mor Naaman, and Deborah Estrin. Immersive recommendation: News and event recommendations using personal digital traces. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 51–62, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[4] Yogesh Jhamb and Yi Fang. A dual-perspective latent factor model for group-aware social event recommendation. *Information Processing Management*, 53(3):559–576, may 2017.

[5] Houda Khrouf and Raphaël Troncy. Hybrid event recommendation using linked data and user diversity. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 185–192. ACM, 2013.

[6] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009.

[7] Yehuda Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model, 2008.

[8] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents.

[9] Augusto Q. Macedo, Leandro B. Marinho, and Rodrygo L.T. Santos. Context-aware event recommendation in event-based social networks. In *Proceedings of the 9th ACM Conference on Recommender Systems*, RecSys '15, pages 123–130, New York, NY, USA, 2015. ACM.

[10] N.N. *Recommender Systems Handbook*. Springer US, 2010.

[11] Anastasios Noulas, Salvatore Scellato, Neal Lathia, and Cecilia Mascolo. A random walk around the city: New venue recommendation in location-based social networks. In *Privacy, security, risk and trust (PASSAT), 2012 international conference on and 2012 international confernece on social computing (socialcom)*, pages 144–153. Ieee, 2012.

[12] Ivens Portugal, Paulo Alencar, and Donald Cowan. The use of machine learning algorithms in recommender systems: A systematic review. 2014.

[13] D. Quercia, N. Lathia, F. Calabrese, G. Di Lorenzo, and J. Crowcroft. Recommending social events from mobile phone location data. In *Proc. IEEE Int. Conf. Data Mining*, pages 971–976, December 2010.

[14] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback.

Annex A: User and Developer Manual

In this section we are going to explain how to set up this project from scratch and how to modify or add new features to the code. As the code is already commented, we are going to explain the inner workings of the code in a more broad instance, getting a feeling of the general purpose of each class and their interactions.

Meetup Data Manager

Meetup Data Manager is the name of the Java project used for downloading and managing the data from Meetup endpoints. This project is capable of creating Queries to the different endpoints of the Meetup API and automatically download all its contents. It also can save this information in multiple formats and cross ID between different files. As we have seen previously, the objective of this project is to make datafiles for the library *Librec* .

The installation of this project is as straightforward as it can be, import it with any Java IDE just like any other Java project. We recommend having a folder called "DataFiles" or similar to store the information by city.

The main class of the project features an operation selector: any operation that can be done in this project can be called using this selector. First of all we will explain the operations related to downloading information and then we will talk about how to store and structure the saved data. The list of possible operations using this project are:

1. **Get Events**: Given a city and and two timestamps, it creates a set of four queries to the "open_events" endpoint. Each query aims to get all the events of the city in a 100 KM radius, but it does so one week at a time. This query is made specifically for retrieving all the events of the past month. As we talked previously in Section 3.1, this endpoint has event data only from events one month in the past. Each query has a max of 10.000 events returned. If it ever gets the case where any query is returning that number, is probably capping the retrieved information, and losing events past the 10.000 event mark. The files are saved in the root of the project by default.

2. **Get RSVPS**: Same as the first operation, this operation creates a query to the "rsvp" endpoint. In order to download RSVP data, we first need an Event id. That is why this operation also loads a file from a specific file name format: `'city' + Events + 'number'`. After loading the event file, it reads all the event id and makes a single call for each one. After all is done, the result is saved in another file called RSVP file.

3. **Filter RSVP**: Loads all the Event and RSVP files in the folder and then proceeds to filter out RSVP that are from events farther than X distance from the center of the city. The filtering distance is measured from the event position to the center of the town, and appears in the Event file with a parameter called "distance".

4. **RSVP to TXT**: This operation converts a RSVP file into the *Librec* format. It loads all the Event and RSVP files and for each RSVP creates a line of the file with the format UIRT (User - Item - Rating - Timestamp). The first three appear on the RSVP file, while the latter appears on the Events file. A boolean can be passed to the function. If it is true, the function creates two extra files, one containing all the user ID in a text file and another containing item ID. This will come in handy in operation 6 and 7.

5. **Split RSVP**: This operation takes a file of the *Librec* format and splits it into files by number of ratings the user has in the whole dataset. The number of splits and the ranges of each split are hard coded but can be changed easily inside the code. This is a final optional step in the collaborative filtering data processing.

6. **User Position**: From the additional files produced in operation 4, one of them was a unfinished set of user IDs. We can use that file to process the location of each user by ID. This operation calls the "members" endpoint with each user ID in the input file, and saves a file with the ID and the latitude and longitude of the approximated user positions. Same as operation 2, a Query gets made, sent and processed for every ID, so it takes time, even more considering that there are substantially more users than events in the dataset. The result of this operation is a file that can be introduced in the distance-aware recommender.

7. **Event Position**: From the additional files produced in operation 4, one of them was a unfinished set of event IDs. Same as operation 6, we can load that file and complete each id with its longitude and latitude coordinates. This information does not need to be downloaded, because it already appears in the Events file. The result of this operation is a file that can be introduced in the distance-aware recommender.

8. **Events to Map**: Operation 8 and 9 were created solely to map and plot information about the dataset, like in Section 4.1. Since they are useful in some way, they remain in the project. This operation loads all the event files and saves the longitude and latitude. This file can later be used easily in a tool like *Fusion Tables*[1] to plot the data in a map.

9. **Event to Time**: Similar to operation 8, this one loads a file in the *Librec* format and reads the timestamp data, computes the day of the week and hour of the day of said event and saves it. This can be used later in some plot tool.

This set of operations might be might not very flexible as it is now. In fact, most of the operations have hard coded parameters, so it is almost mandatory to enter the code in order to use it. Of course, it can be changed easily to download information from other endpoints, combine it in different ways, change the structure of the files, etc. Knowing the function of each class inside the code can help in modifying or adding new features to it:

- **Main class**: Class that hold the operation selector. It also has some utility functions, like getting the state of a city (hard coded).

- **QueryBuilder classes**: Each of the QueryBuilder classes represents the input parameters of a endpoint that can be called in the Meetup API. Because the number of attributes is really high

[1] https://fusiontables.google.com/

```
QueryEvent qe = qeb
    .country("US")
    .state("CA")
    .city("Los Angeles")
    .nResults("10000")
    .convertedTime(1522540800000L + "," + 1523210400000L)
    .status("past")
    .radius("100")
    .buildQuery();
```

Figure 6.1: Code that creates a QueryEvent object. First we set the parameters, in no order in particular, and at the end we can create the object with `buildquery()`

and usually most of them are null we decided, after some thought, to model this classes after a Builder pattern. Instead of creating a long and mostly empty constructor for the class, we modeled two classes: the first one has setters of the parameters, and upon calling the method called `buildQuery()` it creates a class with the parameters instantly. Even though this system is more complex, it simplifies the main class when creating new Queries so it is easier to create specific Queries and more readable (Figure 6.1).

- **FileConverter class**: FileConverter does haves all the conversion operations from file to file. It is just an utils file that holds the functions.
- **FileUtils class**: Utils class that loads and saves files with JSON or text format. It is also able to get all the files of one type in a folder (by file name patterns).
- **QuerySender class**: Takes care of the connection between the program and the Meetup API endpoint. If the connection fails, it retries a few times, because the error could be from an error in their end, or because the information cannot be obtained, or because the parameters are incorrect. If the return code is an error code (403, 410, etc.) The program knows Meetup is returning these values because the information either it is missing or is private, and then it generates an exception. Any type of Query can be sent and processed, independently of the endpoint.

Librec

In this section we will talk about the implementations of the contextual-aware algorithms implemented on *Librec* specially for this work. The library can be used in two ways, by console or by code. By console it is possible to do a simple runs of the code indicating the parameters of the execution or even indicating the route of a Configuration File. In general though, this mode is not advised as the number of options is limited. We suggest creating a Java project or even adding code in the same library. With code, a function called `runJob()` can be called, which does the same as calling by console. Importing the library into another project it is also possible to make a custom implementation of the function, instead of using the one *Librec* uses.

In order to create new algorithms, an easy way is to copy an already existing algorithm that extends from the `AbstractRecommender` class and change the code. In order to use that new recommender, the full name of the class must be declared in the configuration file, for example:

```
rec.recommender.class = net.librec.recommender.baseline.DistanceAwareRecommender
```

All the recommender classes have the same functions: `setup()` which initializes general variables, `trainModel()` which is used to fully fill the train matrix, and `predict(int userIdx, int itemIdx)` which returns a rating for each pair user - item in the test set. The main part of the algorithm is going to be on this last function.

For the implemented Distance-aware and Time-aware algorithms, files with contextual data were loaded in the `setup()` function. The names of the files right now are hard coded. *Librec* has a configuration object loaded in almost all classes during the execution, but we could not find how to load extra variables, like an extra path name. In future work this could be added. In the `predict(int userIdx, int itemIdx)` function the two indexes are translated back into the id of the datasets inversing the `dataModel` Map, which holds the information loaded from the datasets. After getting the original id, we can use the custom files we loaded to get any information about items or users. In Distance-aware, we load the user position and item position from all items rated by the user, and the target item too. We use that information as described in Section 3.3 to compute the final rating. In Time-aware, we use the same mechanism to load times of items rated by the user and also the item of the target item. One last thing to note is that we get the user relations from the train portion of the dataset (using the global variable `trainMatrix`). Never use `testMatrix` as it would render the train-test split pointless.