Master's Degree in Informatics Engineering
Internship
Final Report

Machine Learning in a Recommendation System

João Miguel Gonçalves Garcia
jgarcia@student.dei.uc.pt

Supervisors:
Prof. Ernesto Costa
Eng. Carlos Oliveira
Date: 02 September 2015

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Master's Degree in Informatics Engineering
Internship
Final Report

Machine Learning in a Recommendation System

João Miguel Gonçalves Garcia
jgarcia@student.dei.uc.pt

Supervisors:
Prof. Ernesto Costa
Eng. Carlos Oliveira
Jury:
Prof. Hugo Oliveira
Prof. Tiago Cruz

Date: 02 September 2015

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

Nowadays, given the large availability of different types of information, it is possible to develop intelligent systems that learn from this information and are able to perform certain tasks better than static systems.

The present work reports the development of a recommendation system for Livin'X, a platform for the real-estate market belonging to Ubiwhere. The intention is to provide the users with quality recommendations based not only on house information, but also on the communities they form and, more importantly, the preference the users show for the area a given house is inserted in.

In this work a review of the state of the art on recommendation systems was made, identifying the principles and techniques mostly used. Based on that study it was proposed, implemented and tested an architecture for a recommendation system for the real-estate market. User and city information was gathered in order to feed the system. A genetic algorithm that clusters city areas was developed, allowing for the calculation of the users preference for the area a given house is inserted in. Later, the houses' similarity was calculated using both their characteristics and the user interaction with them. Finally, using collaborative filtering, the developed system was able to provide recommendations more efficiently and with better quality than the previous engine Livin'X was using. In this document it was also reported the difficulties encountered and how they were dealt with. The way the system was validated is also clarified as well as the fine tuning done to optimize the system.

Keywords: City Area Clustering, Machine Learning, Recommendation System

Resumo

Hoje em dia, dada a grande quantidade de diferentes tipos de informação, é possível desenvolver sistemas inteligentes que aprendem usando este tipo de informação e são capazes de realizar certas tarefas melhor que sistemas estáticos.

O presente trabalho reporta o desenvolvimento de um sistema de recomendação para Livin'X, uma plataforma para o mercado imobiliário que pertence à Ubiwhere. A intenção é oferecer aos utilizadores recomendações de qualidade baseado não só na informação das casas, mas também nas comunidades que os utilizadores formam e, mais importante, a preferencia que os utilizadores mostram pelas áreas onde as casa estão inseridas.

Neste trabalho foi feita uma revisão do estado da arte de sistemas de recomendação, identificando os princípios e técnicas mais utilizadas. Baseado neste estudo, foi proposto, implementado e testado uma arquitetura para um sistema de recomendação para o mercado imobiliário. Procurou-se e guardou-se informação de utilizadores e da cidade para alimentar o sistema. Um algoritmo genético que agrupa áreas da cidade foi desenvolvido, permitindo o cálculo da preferência dos utilizadores por dada área da cidade onde dada casa está inserida. Mais tarde, a semelhança entre casas foi calculada usando tanto as suas características como as interações dos utilizadores com elas. Finalmente, usando Collaborative Filtering, o sistema desenvolvido foi capaz de criar recomendações mais eficazmente e de melhor qualidade que o motor de recomendações previamente em uso no Livin'X. Neste documento está também reportado as dificuldades encontradas e como se lidou com elas. A maneira como o sistema foi validado também é clarificada tal como a afinação para otimizar o sistema.

Palavras-Chaves: *Clustering* de áreas de cidade, *Machine Learning*, Sistema de Recomendação

Acknowledgements

I would like to thank Professor Ernesto Costa for his professionalism and dedication in guaranteeing that I had all the conditions to develop this work. His advices in some aspects of this work, especially when handling genetic algorithms, were of great help too.

I would also like to thank Carlos Oliveira for his support throughout this work, making the effort of explaining with detail every decision taken during this project and going the extra mile to guarantee the quality of this work. Also, I have thank everyone at Ubiwhere for providing such great work conditions and giving me the opportunity to work in such an interesting project.

I want to thank my friends that, during the good and bad times, taught me to be ambitious, more hard working and, in general, helped me become a better person.

I also want to thank Fabíola Freitas for all the love and care and for always supporting me in the bad times, always making the effort in making me feel better.

More importantly, I would like to thank my parents and my sister that, throughout my life, have always supported me in many levels and made me what I am today.

Contents

1 Introduction **1**
1.1 Context and Motivation . . . 2
1.2 Objectives . . . 2
1.3 Challenges . . . 5
1.4 Structure . . . 5

2 State of the Art **7**
2.1 Concepts . . . 7
2.1.1 Machine Learning . . . 7
2.1.2 Recommendation System . . . 8
2.1.3 Cold-Start . . . 8
2.1.4 Community . . . 9
2.1.5 Ontology . . . 9
2.1.6 NoSQL Database . . . 9
2.1.7 Graph . . . 10
2.1.8 MapReduce . . . 10
2.2 Techniques . . . 11
2.2.1 Similarity Measures . . . 11
2.2.2 Collaborative Filtering . . . 12
2.2.3 Maximization Algorithms . . . 13
2.2.4 Dimensionality Reduction . . . 15
2.2.5 Spectral Clustering . . . 16
2.2.6 Restricted Boltzmann Machines . . . 16
2.3 Related Work . . . 17
2.3.1 Recommendation Systems . . . 17
2.3.2 City Area Clustering . . . 20
2.3.3 Community Detection . . . 21
2.4 Technologies . . . 21
2.4.1 Apache Storm . . . 21
2.4.2 Apache Spark . . . 22
2.4.3 OrientDB . . . 23

2.5 Conclusions 23

3 Architecture **25**
3.1 General Architecture 25
3.2 City Areas Module 27
3.2.1 City Area Clustering 27
3.2.2 Communities 30
3.2.3 User City Preferences 30
3.3 Interaction and Semantic Similarity Module 31
3.3.1 Matrices Values 32
3.4 Only Input Module 33
3.5 Recommendation Module 34
3.6 Scalability 35

4 Development **37**
4.1 Information 37
4.1.1 Security 38
4.1.2 Events 38
4.1.3 Users 38
4.2 Real Estate Dataset 39
4.2.1 Houses 39
4.3 City Area Preferences 44
4.3.1 City Area Clustering 44
4.3.2 Communities 51
4.3.3 User Area Preferences 51
4.4 Recommendations 51
4.5 OrientDB 51

5 Testing **53**
5.1 Developer Testing 54
5.2 User Testing 59
5.3 Test Conclusions 60

6 Conclusion and Future Work **61**

Appendices **67**

A Methodology **68**

B Feature List **69**

C First Semester Time Allocation **71**

D Second Semester Time Allocation **72**

E Security Levels **73**

F User Profiles **74**

List of Figures

1.1 Livin'X initial questionnaire - Point of Interest (Not final) 3
1.2 Livin'X initial questionnaire - Desired Services (Not final) 3
1.3 Livin'X House Recommendations (Not final) 4
1.4 Livin'X Area Rating (Not final) 4

2.1 The result of area clustering in London [30]. 20
2.2 Storm Topology 22

3.1 High-Level component interaction diagram 26
3.2 City Area Clustering 27
3.3 Community Detection 30
3.4 User City Preferences 31
3.5 Semantic and Interaction Similarity 32
3.6 House Ranking 33
3.7 Recommendation Processing 34

4.1 Number of houses by housing state 40
4.2 Number of houses by type 40
4.3 Number of houses on sale and to rent 41
4.4 Number of houses by typology 41
4.5 Number of houses on sale per price range 42
4.6 Number of houses for rent by price range 42
4.7 Distribution of houses throughout Lisbon 43
4.8 Test 10 fitness evolution 48
4.9 Test 13 number of areas through the generations 49
4.10 Test 19 fitness evolution 49
4.11 Test 19 number of areas through the generations 50
4.12 Areas generated in Test 19 50

5.1 Area results from only input test for user 1 56
5.2 Area results from interaction recommendation test for user 1 56
5.3 Area results from only input test for user 2 57

5.4 Area results from only input test for user 3 58
C.1 Gantt chart representing the first semester time allocation 71
D.1 Gantt chart representing the second semester time allocation 72

List of Tables

4.1 Test parameters 46
E.1 Municipality Security Levels 73
F.1 User Profiles 74

Acronyms

CF Collaborative Filtering

DB Database

PoIs Points of Interest

FDD Feature-Driven Development

RBM Restricted Boltzmann Machine

TSP Travelling Salesman Problem

SVD Singular-Value Decomposition

XML Extensible Markup Language

JSON JavaScript Object Notation

RMSE Root-Mean-Squared Error

DBMS Database Management System

DIMSUM Dimension Independent Matrix Square using MapReduce

Chapter 1

Introduction

Recommendation systems have been a very important factor in the success of e-commerce companies like Amazon, Netflix or eBay. These companies use techniques like Market Basket Analysis, where, given an item, the system recommends other items that are usually bought with it (e.g., men that buy dippers in the supermarket tend to also buy beer), or Collaborative Filtering, where, for instance, a user is compared to other users based on their purchases and gets recommendations based on what similar users have bought. Using data provided by the users, directly (i.e., rating a movie) or indirectly (i.e., time spent on a movie page), the system provides items of interest to them. Both parties come out ahead: the user gets quick access to relevant items and the company is able to offer better service to its costumers.

It is infeasible to generate recommendations to thousands or millions of users with different preferences in an environment constantly changing with static methods. A more intelligent approach is necessary to create a decent recommendation engine. Machine Learning is an area of science involving the study and development of techniques to allow computers to "learn" from data without being explicitly programmed to do so. Its ability to adapt to a constant stream of data is very useful for a recommendation engine.

The present work presents the development of a recommendation system for **Livin'X**, an e-commerce platform for the real-estate market in Lisbon (with the possibility of expanding to other major cities in the future), developed by Ubiwhere, a software company based in Aveiro, although this work was mainly developed in their offices in Coimbra.

1.1 Context and Motivation

Livin'X is a platform that can be used on a web-browser where users insert their preferences (figures 1.1 and 1.2) and are presented certain houses, apartments or rooms according to the input they gave (figure 1.3). The results are then presented in a map which also defines and rates the city areas where the houses are shown on (figure 1.4). This allows the user to only see houses that match their preferences and, additionally, get an idea of the quality of the house's surrounding area, an important factor when choosing a place to live. The current recommendation engine works more like a filter, defining city areas according to the users' preferences and then rating the areas based on the amount and quality of services within the area and their distance to the users' Points of Interest (PoIs) taking into account their preferred means of transportation. Finally, it matches the users to houses within those areas.

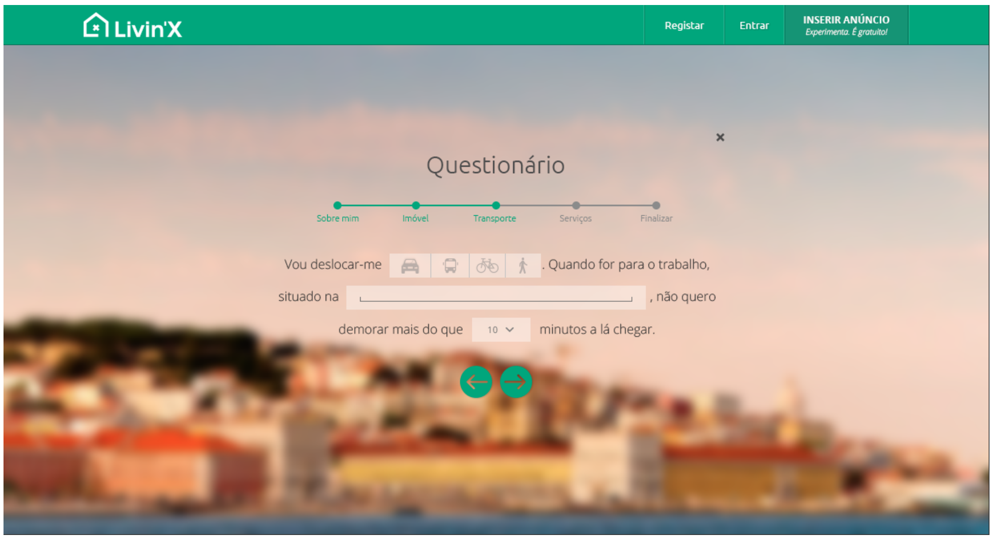
But since there is so much information available, it is possible to develop a robust recommendation system that can obtain better results by processing different kinds of information (e.g., provided by the users, house characteristics, city services) into good, evolving recommendations.

1.2 Objectives

This project intends to create a robust recommendation system for the Livin'X platform that can use the following information to generate relevant recommendations for the user:

- **City Information:** Some examples of this kind of information are: city services and events, people's activities, security information;
- **Community Information:** Use Facebook information the users provide to generate social graphs and infer in which communities each user belongs;
- **User Preferences:** Input given by the user indicating their city and house preferences;
- **User Interactions:** Information about how the user uses the platform (e.g., time spent on a given page, changes made to their preferences);
- **House Characteristics:** This information can be, among others, number of rooms, garages, pets allowed.

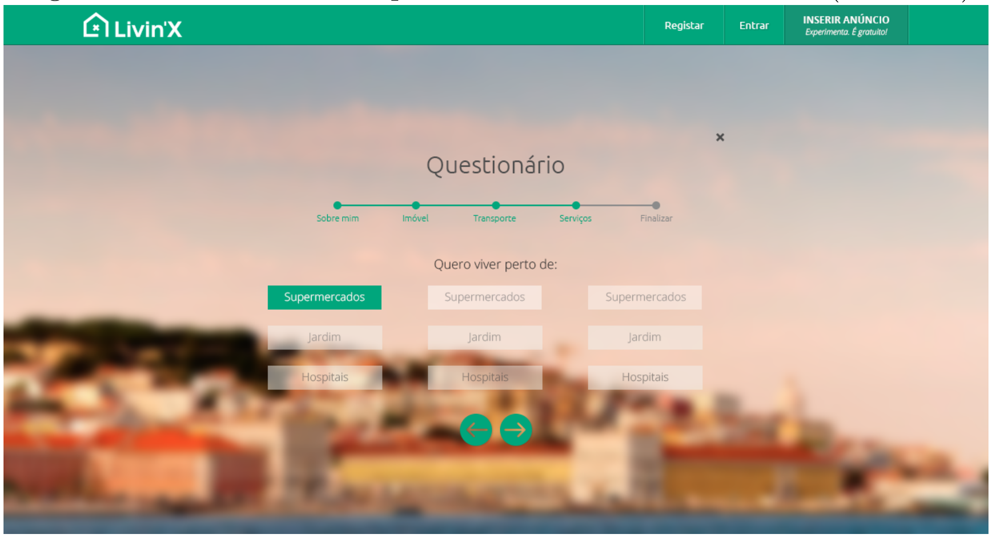
Using the continuous stream of feedback gathered through the users' interactions with the Livin'X platform, the engine must incrementally understand users'



Questionário

Vou deslocar-me . Quando for para o trabalho, situado na , não quero demorar mais do que 10 minutos a lá chegar.

Figure 1.1: Livin'X initial questionnaire - Point of Interest (Not final)



Questionário

Quero viver perto de:

Figure 1.2: Livin'X initial questionnaire - Desired Services (Not final)

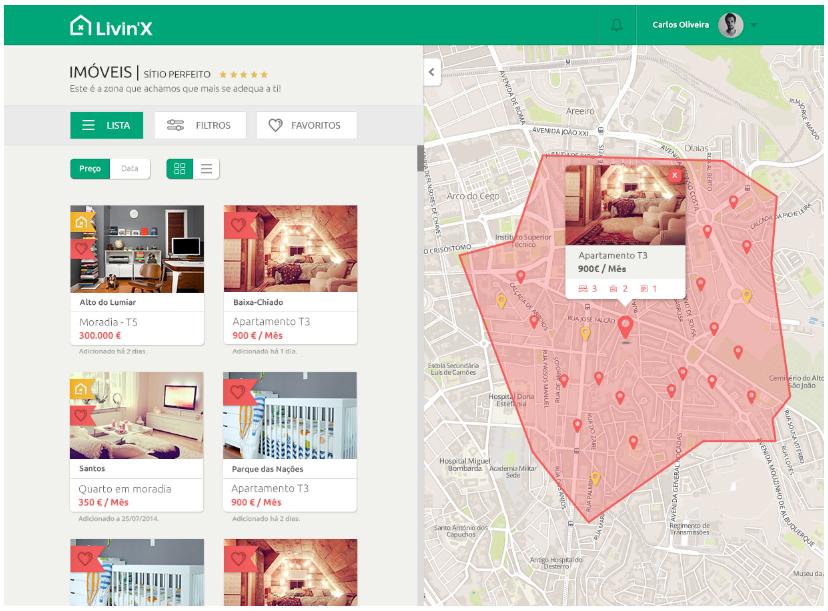


Figure 1.3: Livin'X House Recommendations (Not final)

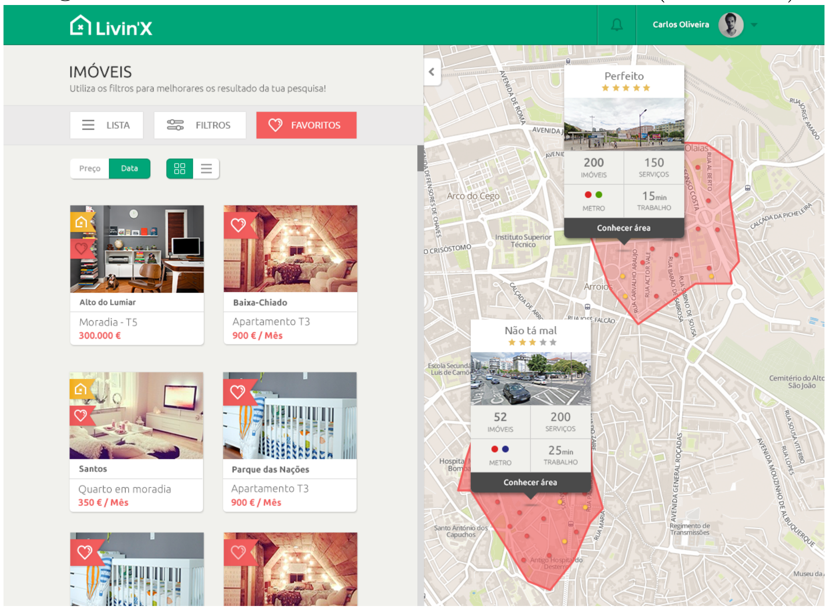


Figure 1.4: Livin'X Area Rating (Not final)

preferences in order to improve the quality of their recommendations. It must also adapt to situations where there is not enough information to infer the users preferences (e.g., new users).

Processing this much information involves heavy computations, so it is important to adopt efficient methods to provide the user recommendations in a reasonable time.

Accomplishing these goals will allow the system to quickly relate the user to areas of the city and find houses that correspond to the users' preferences. If there is no house with the characteristics desired by the users, it will be able to find houses that, while not exactly what the user asked for, are close to the users' preferences and may even be better than what they expected.

1.3 Challenges

Considering these objectives, there are three main challenges in the development of this recommendation engine:

- **Handle Information:** As shown in section 1.2, a large amount of different information is available to generate recommendations. The right techniques must be chosen to make full use of the information provided and, additionally, it is necessary to pick methods that are able to combine all the information and provide the best recommendations possible;

- **Learn from interactions:** The feedback the users continuously provide must be used by the recommendation engine to incrementally improve the recommendations. Although users explicitly provide house and city preferences, the recommendation engine will go a step further and infer a user's preferences through his and other users' feedback taken from their interactions with the platform;

- **Efficiency:** When using the Livin'X platform, most users will not want to wait too much for their recommendations to be generated. With that in mind, the algorithms and technologies chosen to process the information must be able to do it efficiently and return the user's recommendation as quickly as possible, after a request has been made.

1.4 Structure

Chapter 2 presents the results of the study of the state of the art. It starts by describing some concepts and techniques referred throughout the work which might not be familiar to some of the readers. After that, it presents the related work relevant to this project. Finally, the technologies used in the development of the work are presented and described.

Chapter 3 describes the architecture of the solution that will be implemented. It shows how the information is handled and processed internally and how the recommendation engine interacts with the outside.

Chapter 4 reports the development process of the recommendation engine and chapter 5 the tests made to validate its results.

Finally, in chapter 6, the future work is listed and conclusions are drawn from all the work and results obtained from this project.

Chapter 2

State of the Art

This chapter presents the results of the research made on the diverse areas involved in this project. Since the intention is to develop a complex system able to extract various types of information from the given data, research was directed not only to different approaches to create quality recommendation systems, but also at the detection of communities in social graphs, clustering of city areas and the use of semantic information in this kind of systems. Solutions to handle large amounts of data that deliver quick recommendations were also a focus of the research.

2.1 Concepts

This section will present various concepts common to recommendation systems and machine learning in general that are mentioned throughout the work.

2.1.1 Machine Learning

Machine Learning is a field that studies algorithms that build models upon which predictions can be made [15]. These models, instead of following static instructions, develop themselves by learning from data. Machine learning algorithms can be divided in six major areas [16], defined by the way that they learn and by their desired outcomes:

- **Supervised:** Algorithms that learn from data whose desired output is already known. This allows for the algorithm to adjust its results to the desired output;

- **Unsupervised:** Algorithms that operate over unlabelled data with the task of finding some sort of structure in it;

- **Semi-supervised:** Algorithms that use labelled and unlabelled examples to generate a classifier;
- **Reinforcement:** Algorithms that act in a given environment, where every action they make has some reaction. It learns by using the environment's feedback to its actions;
- **Transduction:** Predictions are generated not by using a model but instead by the training's inputs and outputs;
- **Learning to learn:** The learning methods involve learning from past experiences.

A use of Machine Learning in this work can be seen when using Collaborative Filtering (definition in section 2.2.2, use in 3.5). It builds a model that defines the houses similarities by using the user interactions with the houses and their characteristics. Given the user interactions, it returns the houses more similar to the ones he already has shown interest in.

2.1.2 Recommendation System

Recommendation Systems are composed by algorithms that generate recommendations of a given type of item for users based on the information they provide [26]. These items can be of any kind, from books to news articles. Systems of this kind see most of their use in e-commerce platforms. Since they show unseen items to the user that might catch the user's interest, they try generate more purchases and increase the quality of the user experience on the platform.

These systems have some inherent problems, like the need to deal with millions of items, users and the stream of new information that costumers provide, generating recommendations in a reasonable amount of time and lack of information from new costumers.

2.1.3 Cold-Start

The cold-start problem is something to take into account when developing recommendation systems. Some recommendation techniques (e.g., Collaborative Filtering) try to match the target user with other users according to some metric (e.g., rated movies) and recommend other items that similar users liked. The cold-start is the situation where a user has not given any information that allows the comparison with other users (e.g., new users who have not yet rated any movie) and thus make it impossible to provide a recommendation with those techniques [17].

2.1.4 Community

A community can be considered a group of people who share a particular characteristic in common. It can be a family, friends from college or certain hobby enthusiasts [37].

The detection of communities can be helpful in recommendation systems like, in the particular case of this work, to help recommend houses in areas where several members of a person's community live.

Generally, community detection uses information from social networks to infer which communities a certain person belongs to. For example, using Facebook information, a person can be assumed to belong to a certain community through their likes (e.g., music bands, sports club), groups (e.g., friends from college, hobby group) or even from their connections alone (e.g., if a certain number of people have lots of mutual connections, it can be assumed that all of them form a community).

2.1.5 Ontology

An ontology is a representation of knowledge of a given domain through a set of objects, with their given properties, and the describable relationships between them [20]. This allows large amounts of data to be organized, to find relevant information quickly and to make logic operations over the data.

2.1.6 NoSQL Database

NoSQL databases, as any other type of databases, allow the storage and retrieval of data. Contrary to relational databases where the information structure is organized in several tables where each entry has one unique key, the information in NoSQL is not represented by tabular relations but by other types of structures (e.g., documents, graphs).

Database Document

Database (DB) documents are an alternative to a DB table, where the columns refer to a field and the lines to an entry. Documents are normally in a format readable by both humans and computers (e.g., Extensible Markup Language (XML), JavaScript Object Notation (JSON)), able to contain large amounts of information in each. While with tables repeatable fields need an additional table, documents allow all the information relating to one entity to be gathered in the same documents.

ACID Transactions

ACID is a set of proprieties that guarantee reliability in DB transactions (a logical operation on data). ACID stands for:

- **Atomicity:** Requires that a transaction must be complete or, if something fails, the DB must remain unchanged.
- **Consistency:** Requires that a transaction must leave the DB on a valid state, according to all the rules defined.
- **Isolation:** Require that transactions executed concurrently leave the DB in the same state as if they were executed serially.
- **Durability:** Requires that, after an action has been committed, it remains committed in an event of a system failure.

2.1.7 Graph

A graph is a representation of a dataset where objects have some sort of relation with each other. It can be defined as $G = (V, E)$ where G is a graph composed by a V set of vertices and a E set of edges [37]. In a graph, the objects may be connected to each other with an edge, that may represent different kinds of information. For example, an object might be a Facebook user and an edge between users means that they are friends.

Graphs may be directed or undirected: the previous Facebook example represents an undirected graph, since if user u is a friend of user v, the inverse must also occur, user v must also be a friend of user u. However, in the Twitter universe, user u may follow user v, but that does not mean user v follows user u. This is an example of a directed graph, where edges have a direction.

Bipartite Graph

A bipartite graph is a graph where the vertices can be divided into two sets, U and V, where every edge of the graph connects a vertex from U to one from V (i.e., U and V are disjoint sets).

2.1.8 MapReduce

MapReduce is a model of processing and handling large amounts of data with parallel and distributed computing [23]. When developing something with MapReduce, the developer has to write two functions: one will be the *Map* function and the other will be the *Reduce* function. The tasks that handle the *Map* and *Reduce* functions will execute in parallel throughout a distributed system. The *Map* tasks transforms chunks of data into *key-value* pairs. Then, these pairs will be distributed to *Reduce* tasks, where the pairs with the same key will end up in the same tasks. The *Reduce* tasks will perform certain operations to the set of values with the same key.

MapReduce's value comes from its scalability, since it allows the dissemination of tasks throughout a distributed file system.

2.2 Techniques

This section presents several techniques that are commonly used in recommendation systems environments and that shall be mentioned throughout this work.

2.2.1 Similarity Measures

In recommendation systems, similarity measures are used to check how close a user or an object are to another. This allows the engine to make decisions about, for example, what items to recommend to users based on items similar users like.

Euclidean Distance

The euclidean distance, more particularly, the L_2-*norm*, is the measurement more commonly referred as "distance" [24]. It can be defined as:

$$dist([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_n]) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

Although it is useful and intuitive as a distance measurement of various points in a low dimension space, other types of distance measurements might be more useful in measuring similarity in high dimensionality spaces where each dimension assumes binary or normalized values.

Cosine Similarity

Given a user-item matrix M with k users and p items, where the element $m_{u,i}$ indicates the value[1] a user u gives to an item i, there are several different methods to measure the items similarity. A common one is the Cosine Similarity, that

[1]Note that this value can be given directly (i.e., a user rating a movie) or indirectly (i.e., take note if a user read a given article).

measures the distance between two items by the cosine of the angle formed by their user-given value vectors. When the angle is 0°, the cosine similarity is 1 and means that the items are very similar. When the angle is 90° the cosine similarity is 0 and when it is 180°, the similarity is -1, meaning they are completely different. The similarity between objects i_p and i_q can be calculated as follows:

$$sim(i_p, i_q) = \frac{\sum_{u=1}^{k} m_{u,p} \times m_{u,q}}{\sqrt{\sum_{u=1}^{k}(m_{u,p})^2} \times \sqrt{\sum_{u=1}^{k}(m_{u,q})^2}}$$

However, sometimes users rate things differently (i.e., in a movie site, user x might give a rate of 5 stars to every movie he/she likes while user y only gives this rating to perfect movies) and it is important to take that into account when measuring similarity. In these cases, it is possible to apply the *Pearson Correlation Coefficient* (also known as *Adjusted Cosine Similarity Measure*) [33] which normalizes the values, comparing instead the variances of co-rated items. It is calculated in the following way:

$$sim(i_p, i_q) = \frac{\sum_{u=1}^{k}(m_{u,p} - \overline{M_u}) \times (m_{u,q} - \overline{M_u})}{\sqrt{\sum_{u=1}^{k}(m_{u,p} - \overline{M_u})^2} \times \sqrt{\sum_{u=1}^{k}(m_{u,q} - \overline{M_u})^2}}$$

where $\overline{M_u}$ is the average of the ratings given by user k to all the items.

The choice of using cosine similarity throughout this work is due to being very effective when dealing with sparse vectors since it has no need to make calculations for attributes with value equal to 0. This is not the case when using a more conventional Euclidean Distance.

2.2.2 Collaborative Filtering

Collaborative Filtering (CF) is one of the most used techniques used in recommendation systems. In a e-commerce environment, it basically tries to predict what items a user might be interested in using the opinion of other users with similar tastes.

There are essentially two types of CF algorithms, user-based and item-based [33]. **User-based Collaborative Filtering** tries to find other users (*neighbours*) that tend to rate different items the same way the target user does. Then, different algorithms can be used to combine the preferences of the users to create relevant recommendations. **Item-based Collaborative Filtering** tries to create a model based on the users ratings and, using the target user ratings, recommends similar items he/she might find interesting. One good example of item-based CF is Amazon's recommendation system [26] that starts by comparing items based on the rating the users gave them using the cosine similarity and group similar items in clusters. When making a recommendation, it finds items similar to the ones a target users purchased and aggregates them to make a recommendation (based on popularity or correlation).

An advantage of item-based CF is that it makes all the similarity calculations off-line. That way, the on-line process of recommendation scales only based on the users given ratings. On the other hand, user-based CF has to make all the comparisons on-line, making it not very scalable when dealing with millions of users and/or items.

2.2.3 Maximization Algorithms

The problem of finding the best parameters that maximize a given quality function can be quite complex if the search space is large. Comparing even 10% of the possibilities can take too much time. These cases require algorithms that can find, if not the global maximum, a good solution close to the optimum in an efficient way. These algorithms work generally with the same idea: through various iterations, try to find a better solution in the neighbourhood of the current one and repeat this process until no better solution can be found. One of the biggest challenges is to avoid being stuck in a local maximum where the current solution is better than all its neighbourhood but a different, far away, solution that is not being considered returns a much better result.

Hill Climbing

The Hill Climbing technique [32] is used in optimization problems, iteratively making local searches and opting for the best solution.

It starts with a random solution. Then, given a quality function, it changes one element of the solution, which can be called a neighbour, and calculates its quality. If it is better, then the neighbour becomes the current solution and the algorithm repeats until no better solution can be found.

The problem with this is that in more complex search spaces, it can easily converge to a local optimum. A variation to the algorithm is the **Steepest Ascend Hill Climbing** which tests all the neighbours and chooses the best one. Still, it is computationally heavy and does not solve the problem of local maximum. **Stochastic Hill Climbing** observes a neighbour at random and decides if it will be chosen as a new solution based on the amount of improvement it brings.

Random-restart Hill Climbing runs a series of Hill Climbing algorithms starting from different solutions, storing the best solution found. This way it avoids local optimums by trying more diverse solutions, obtaining a solution closer to the global optimum. Another advantage of this sort of Hill Climbing is that it is easily parallelizable, allowing running various instances of the algorithm from different starting points at the same time.

Simulated Annealing

Simulated Annealing is an algorithm inspired in annealing in metallurgy, where the rate at which a metal is cooled is used to control the size of the crystals formed: if cooled slowly enough, large crystal are formed; but, if cooled too quickly, they will contain imperfections.

The algorithm will work similarly to the Hill Climbing presented above, where, given the current solution, it finds a random neighbour and, if it returns a better result, it becomes the current solution. However, in Simulated Annealing, a worse solution can be accepted as the current one following a probability. At each iteration, this probability will get smaller and smaller until it reaches zero (i.e., it is cooling down, where, the less energy there is, the less likely it is to accept a worse solution). This is the fundamental characteristic of the algorithm, for it allows a wider search of search space, something the Hill Climbing algorithm lacks.

Genetic Algorithms

Genetic Algorithms are methods inspired in natural selection and genetics. They start with a population where each individual is represented by its genotype, which can be a vector with various characteristics. Each genotype generates a phenotype and, by applying a fitness function, its quality is calculated (different genotypes can generate the same phenotype). Through various generations, the following operations will be performed over the population:

- **Selection:** A given number of individuals from the population will be selected to reproduce in the current generation. There are several ways of selecting these individuals. For example, the roulette wheel method selects the individual by its relative fitness (i.e., if an individual quality is 20% of the sum of all individuals quality, it has a 20% chance of being selected). Another common one is the tournament selection, where a small number of individuals are selected to compete with each other and the best one "wins" and is selected to reproduce. These methods are repeated until a desired number of individuals have been selected to constitute the reproduction pool. Apart from picking a good set of individuals, the selection methods also have to be able to maintain a good variety of individuals so that the algorithm will not quickly converge to a local optimum and the search space will not be very explored;

- **Crossover:** The genes of the selected pool of individuals chosen to reproduce will be combined, allowing the formation of individuals with good, different, aspects from their parents, creating the possibility of obtaining higher quality individuals. The type of crossover operator used varies a lot depending on the problem, but on its simplest form, given two individuals p_1 and p_2 and their corresponding vector of characteristics, a position in these vectors will be chosen to divide them in two and two individuals will be formed, one with the first half of p_1 and the second half of p_2 and the other with the first half of p_2 and the second half of p_1;

- **Mutation:** The resulting individuals from the crossover operation will have a probability of suffering a mutation to their genes. This mutation might result in better individuals and provides another way of exploring the search space. As with crossover, the type of mutation methods are very dependent on the problem. With a problem such as the Travelling Salesman Problem (TSP), where the objective is to find the shortest route between a set of points passing by one point only once, the gene of each individual can represented in different ways: it can be an integer vector where each parameter is the order each point is visited. The mutation in this case can be switching two elements of the vector with each other; another form of representing the genes is a vector of floats with values between 0 and 1 where the point with the highest value is the first to be visited and so forth. In this case, the mutation can be changing one (or more) value by a small amount;

- **Survivors:** This operation selects the individuals that survive to the next generation. The selected individuals can simply be the ones with higher quality, both parents and offspring, but this method might quickly destroy the diversity of the population, making it difficult to explore more of the search space. Another method is to simply select the offspring but it runs the risk of losing good individuals. A compromise can be found between both these methods by keeping a small percentage of the best parents and the rest are the best offspring. This allows for a good population diversity without the risk of losing the best solutions found.

Regardless of the chosen operators, the fundamental ideas of genetic algorithms are the same: iteratively find better solutions and maintaining a good population diversity to avoid being stuck in local optima.

2.2.4 Dimensionality Reduction

Data contained in a large matrix of values can be more efficiently stored and used when represented by the product of two, slimmer, matrices. To this end, Dimensionality Reduction (or Matrix Factorization) is used [25]. It is the process of, given a matrix M with $m \times n$ dimensions, where m and n have high values, obtain matrices U, with $m \times f$ dimensions, and V, with $f \times n$ dimensions, with f having a low value, where the product of U with V returns a matrix with values similar to M. Each dimension of f can be considered a latent factor that contributes to the values of M. The smaller the value of f, the less accurate will the resulting matrix be, compared to M.

Singular-Value Decomposition

Singular-Value Decomposition (SVD) [25] is a dimensionality reduction technique where the matrix M is represented by $U\Sigma V^T$ where:

- U is a matrix with $m \times f$ dimensions;
- V is a matrix with $n \times f$ dimensions (V^T is the transpose of V);
- Σ is a linear matrix with $f \times f$ dimensions where every entry not on the main diagonal is zero.

The rows of U, V and Σ can be seen as concepts. As an example, if M represents ratings users have given to movies, where $M_{i,j}$ represents the rating user i gave to movie j, the concepts that the columns of U, V and Σ represent might be movie genres like Action or Comedy. In this case, $U_{i,c}$ represents how much user i likes genre c, $V_{j,c}$ how much a movie j belongs to a genre c and $\Sigma_{c,c}$ indicates how much genre c contributes to the rating a user gives to a movie (low values do not contribute much and can even be changed to zero to reduce noise in the data).

2.2.5 Spectral Clustering

Spectral Clustering is a clustering technique that starts by performing dimensionality reduction on a similarity matrix and clusters the elements with some clustering technique based on the new few dimensions.

2.2.6 Restricted Boltzmann Machines

Restricted Boltzmann Machine (RBM) are a type of stochastic Neural Network, that is, a structure based on biological neural networks, with interconnected neurons, capable of approximating functions with some random factors [19]. In RBM, the neurons and their connections form a bipartite graph, which allows a more efficient processing, with a visible layer and a hidden layer. The RBM must be trained to obtain the desirable results and the problem of tuning the parameters of the machine is one of the biggest challanges since the results are highly dependent of it and, in some cases, can be difficult to find the right values.

2.3 Related Work

In this section, related work relevant to the various areas of this project will be presented. These areas are Recommendation Systems, City Area Clustering and Community Detection. While the last two are well defined, the first is a very wide area and will define not only methods, but also important questions such as efficiency, the combination of different methods and ways of dealing with problems that usually arise when using these kinds of systems.

2.3.1 Recommendation Systems

Back in 2003, Amazon needed a recommendation system that was able to handle data of tens of millions of users and millions of items [26]. They developed an item-based CF algorithm that calculates the similarity between items using cosine similarity and then compares the items the user has bought, rated or has in the shopping cart to the rest of the catalogue. A list of the most similar items is created and then sorted according to the items' relevancy and presented to the user. Calculating the similarity between all the items is computationally very expensive but since it is the same for any user, it can be done off-line. That way, the only on-line calculation is looking up similar items in the similarity table, making the algorithm very scalable.

The work in [17] studies how hybrid recommendation systems compare to regular recommendation algorithms. These types of recommendation systems intend to take advantage of the distinct strong points of popular algorithms to generate better results when used together. Also, when using more than one technique, one can compensate for the shortcoming of another. For example, collaborative systems have the well-known cold-start problem that can be compensated by a knowledge-based system. This study will compare the use of collaborative, content-based and knowledge-based systems with strategies that combine them. These strategies are:

- **Weighted:** A combination of the score returned by the different recommendation components;
- **Switching:** The recommendation component with the best result is chosen;
- **Mixed:** Presents the results of the different components together;

- **Feature Combination:** A single recommendation algorithm uses features taken from the other components;
- **Feature Augmentation:** One component computes a set of features that will be used by another component which will generate the final result;
- **Cascade:** Components are ordered with a given priority, where the lower priority ones break the ties of the top priority ones;
- **Meta-level:** One component produces some sort of model which will be the input for another component that generates the final result.

The experimental results showed that the most successful strategies (i.e., those who showed significant improvements over the simple recommendation techniques) were the Cascade and Feature Augmentation. They also showed that Content-Based and Knowledge-Based techniques, which alone are not great, can help to greatly improve CF recommendation since they contribute with a different kind of information (based on the product features instead of the users).

A video recommendation system was developed in [31] that can deliver real-time quality recommendations to millions of users by exploring the emerging cloud computing technology. By using on-demand cloud computing resources, they are able to adapt to variations in the number of items and users in the system.

In [27][28], the authors develop and improve a system for recommending articles using hierarchical topic ontology. It builds the users profiles based on the information the user provides and on his behaviour. It also uses an hybrid approach with both content-based and CF.

Domain knowledge (through an ontology) is used in [34] to augment CF to improve personalized book recommendations. [35] also uses ontologies successfully to recommend PoIs to a mobile user. Their recommendation system implements spatial, semantic and collaborative filters, using the users' preferences and their geographical position at the time to deliver better recommendations.

A semantically enhanced CF technique is developed in [29] where ontologies are used to improve the quality of the recommendations and to compensate for some of CF's shortcomings (cold-start problem and sparse datasets). Singular-value decomposition is also used to reduce the amount of noise in the data and its sparsity. It is also useful since each item representation could end up with more than two thousand dimensions, which takes a lot more time to compute. A weighted linear combination was used to join the similarities calculated using the semantic similarity and the user-item matrix based similarity. The authors empirically test this solution on two datasets: movies and real-estate data. They conclude that the combination of the two techniques is successful in improving the recommendation quality, having a more significant improvement when there is not a lot of user information (cold-start).

The authors of [39] create a recommendation system for real estate websites after a study about user behaviour when searching for a place to live. They detected a pattern where users start by giving input (e.g., location, price) and observe the results returned. Most users then go back and change their search query to obtain more desirable results. A combination of case-based reasoning and an ontological structure is used to create the recommendation system.

A study is performed in [22] on recommendation systems, focusing on the user experience when using them. It states that users tend more to accept recommendations for "low risk" items like books or movies than they do for "high risk" items such as cars or real estate because they will not risk so much based on a recommendation they have no idea how it was made. It suggests that providing explanations on the recommendations can lead to users trusting them more or help them understand and change certain parameters to obtain better ones. Additionally, it refers to various studies that show that users tend to like the explanations and that they help them make decisions on what to purchase.

Netflix Challenge

The Netflix challenge [9] was a contest where the company would give $1.000.000 to the team who could improve the site's movie recommendation engine by 10%. While the recommendation was made only using the rating users gave to movies, it is interesting to take some points from this challenge:

- **Incorporating various solutions:** The algorithm Netflix had implemented at the time was not that good and teams easily beat it with an improvement around 8%. But quickly no one seemed to be able to improve their own solutions. What broke this stalemate were teams with different solutions coming together to integrate them and get better results [36]. In the end, the winning team was in fact composed of three originally different teams.

- **Time of the rating:** The winning team noticed that the time at which the rating was made was important in predicting how users would rate other movies [36]. People tend to give better ratings to movies they just saw. On the other hand, they were more moderate when rating a series of movies in a row that they saw long ago. Another factor also noticed is that people tend to rate movies differently depending on the day of the week.

- **Winning solution was not used:** In the end, Netflix opted not to implement the winning solution [14]. While it gave good results, they were obtained off-line. The gains in accuracy were not considered worthy of all

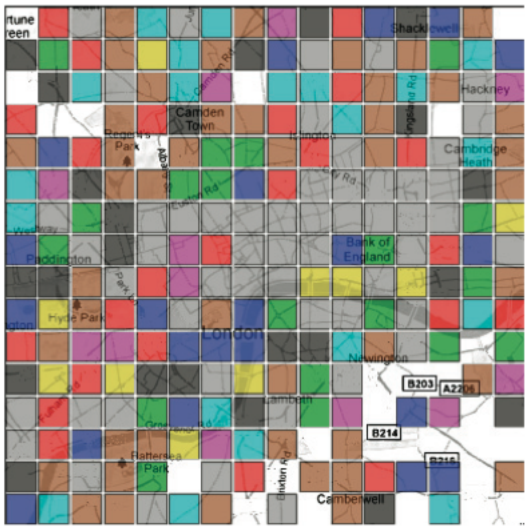


Figure 2.1: The result of area clustering in London [30].

the engineering effort needed to bring it into a productive environment. They ended up using a middle of the road solution, using Matrix Factorization and Restricted Boltzmann Machines.

All these points might be relevant to the project. Using various solutions that, individually do not bring that much, might improve the prediction when used together. Taking into account the user interaction with the platform is also an important factor in a prediction system. Finally, since a personalized system that delivers recommendations on time is desired, it is important to find the balance between the quality of the recommendation and the performance of the system.

2.3.2 City Area Clustering

In [30], areas of cities are modelled and clustered with users' activities information from Foursquare. The city is divided in small sub-sections and the number of each type of activity that is practised in each sub-section is counted. Finally, the sub-sections are clustered using Spectral Clustering. The resulting clusters (example in figure 2.1 where, for instance, the grey areas represent areas where the main activity is "nightlife") ended up representing a main activity practiced in the area, often joined by another, secondary, activity. (i.e., two different clusters where the main activity is "food" but the secondary activities are "shops" and "nightlife" respectively).

CLEVER [18] is an algorithm that finds uniform regions in urban areas. It defines the problem of finding these regions as a maximization problem, where it uses randomized hill climbing to obtain the best value of a fitness function. A cluster's content is represented by its signature, which gives the percentage of the number of each type of building in a given area. The fitness function will reward the purity of a region (i.e., how much a single type of building dominates the rest) and the size of the clusters (the bigger, the better). The importance of this last component to the reward function can be defined by the user. Experimental results showed that, although the quantity of clusters on the same data can vary in a lot on different runs, their quality tends to be similar.

2.3.3 Community Detection

BigCLAM [37] is a method for detecting overlapping communities in large networks. It models community affiliation with a bipartite graph where communities are connected to nodes with an edge that has a given weight representing the degree that each node belongs to a given community. Experiments showed that BigCLAM is very scalable and works well in networks similar to Facebook.

The algorithm CESNA [38] was developed to detected communities in networks, taking advantage of both the network's structure and the nodes' information to more accurately detect overlapping communities. Compared with the state of the art algorithms for this purpose picked by the authors, it scales better and obtains better results. However, when they experimented with Facebook data (which is relevant for the current work), the obtained results, compared with the BigCLAM algorithm, are not significantly better and the performance is slightly worse. It did detect that attributes such as school attended, education received and major are more relevant in the detection of communities.

2.4 Technologies

This section presents the technologies used in the development of the recommendation engine. Since the engine's processes will involve handling large amounts of data as well as streaming information, the chosen technologies must be able to handle this in an efficient way.

2.4.1 Apache Storm

Apache Storm [3] allows reliable processing of streaming data in real-time. A Storm topology (figure 2.2) has three basic components:

- **Stream:** An unbounded sequence of tuples. Tuples are simple key-value pairs;

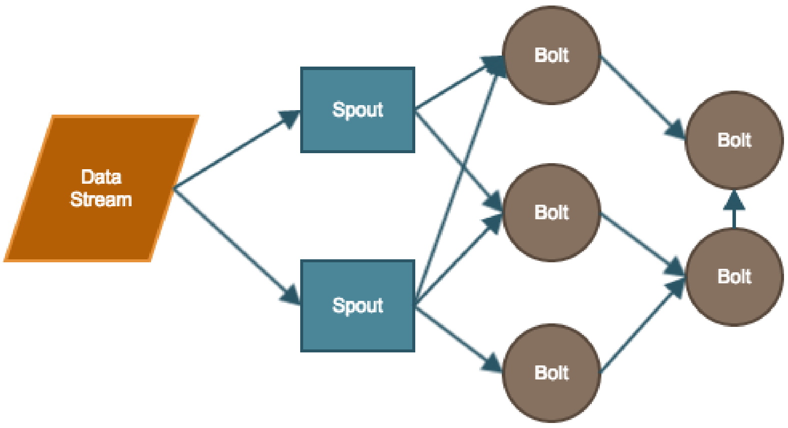


Figure 2.2: Storm Topology

- **Spout:** Spouts are the entry points of data into the topology. They take the raw data, transform it into a tuple and emit it.
- **Bolt:** Bolts are operators that receive the data stream, process it in some way and then might emit it to some other bolts.

An important concept to know when developing an application using Apache Storm is stream grouping. This defines how tuples are distributed among the available bolts. For example, in a matrix multiplication, a spout may receive the two matrices, X and Y, and create tuples where the keys are the positions i and j of the resulting matrix Z and the values are the elements necessary to calculate the value of $Z_{i,j}$ (i.e., line i of matrix X and column j of matrix Y). Then, each tuple can be equally distributed through different bolts (Shuffle grouping) to be processed in parallel. After this process, the resulting values must be gathered to construct the resulting matrix. In order to do this, they have to be grouped by the matrix they belong to, Z (Fields grouping). The final bolt can then join all the values in the same matrix and return it to the user.

If, for some reason, it is found out that Storm does not really fit this project, an alternative can be found in Apache Spark, presented in section 2.4.2, which includes methods that handle streaming data. Spark handles data in small batches, being able to use its in-memory methods to deliver very fast responses.

2.4.2 Apache Spark

Apache Spark [2] provides fast processing of data. Its performance comes from running everything mostly on memory, but there are studies [21] that demonstrate that it can beat Hadoop's MapReduce [5], widely used for this kind of processing, when the input does not fit in memory and it is necessary to make multiple disk calls.

MLlib is Spark's machine learning library. It provides fast and scalable algorithms that are useful for this project, such as CF or the calculation of the cosine similarity on large datasets.

An alternative to Spark can be found in Mahout [7], which implements several scalable and efficient machine learning techniques using the already mentioned Hadoop's MapReduce framework.

2.4.3 OrientDB

OrientDB [10] is the chosen Database Management System (DBMS) for this project. The other DBMS considered was MongoDB [8], which the Livin'X platform already uses. Both are NoSQL DBMS but, while MongoDB is a Document DB, OrientDB is a hybrid Document-Graph engine [11].

Apart from supporting embedded documents, OrientDB also supports relationships by using pointers between records [13]. This allows traversing entire trees or graphs of documents in few milliseconds. This feature is useful when handling large amounts of information since the relationship between documents is assigned only once. This way, the traversing speed is not affected by the size of the DB. Another advantage over MongoDB is that OrientDB implements ACID transactions, guaranteeing reliability.

Nobody in Ubiwhere has used OrientDB, so it will be a new experience within the company and it is unknown if it will be successful or not. In the latter case, MongoDB will be adopted, with which there are enough people with vast experience using it in the company, making the transition and learning of this DBMS quicker and allow the concentration of efforts in other areas of the project.

2.5 Conclusions

Important conclusions for the correct development of the system can be taken from the study of the state of the art.

- **Off-line Computations:** When providing services for a user, waiting times must be avoided in order to keep the user's attention. Yet, these services might require heavy computations that slow the process of returning a response. In these cases, when able, the processes should be made off-line, so that the minimum possible computation is done when users request something, thus reducing response time.

- **Combining Solutions:** Sometimes, some types of data return weak results when used for recommendations. Yet, while their value alone can be considered small, they might help the general recommendations when combined with other types of information. Thus, smart combinations can make irrelevant data relevant to the final results.

- **Geographical Areas Clustering:** Defining city areas based on non-geographical information, like types of buildings, can be a very complex process given the large amount of different combinations possible. As such, non-deterministic algorithms must be adopted to tackle the problem of finding the best areas. To this end, it is also important to create relevant quality functions that evaluate the areas' quality.

- **Understanding Recommendations:** Since Livin'X handles high-risk data for users (real-estate), it is important that they understand why certain houses are recommended. Given two apparently similar houses, making sure that users understand why one is being more highly rated to them than the other will help the users make a more informed decision and raise their trust in the system.

Chapter 3

Architecture

After the study of the state of the art, enough knowledge had been obtained to start the planning of the system.

In this chapter the architecture of this system is described. Given its complexity, it is divided in various modules that operate with different kinds of information.

The architecture was developed with the objectives initially defined in mind. City areas are defined based on the different kinds of city information available, allowing the creation of models that can relate users to houses according to their surrounding area. Users' communities are calculated, opening the possibility of recommending houses based on the how much a community a given user is inserted in belongs to the house's surrounding area. Similarity between houses are calculated using both the users' interactions with the platform and the houses' characteristics. All these factors will be used in the end to provide quality recommendations to the end users.

3.1 General Architecture

The diagram 3.1 presents a general view of how the recommendation engine will communicate with the main Livin'X application. It is desired that both of them are independent from each other to allow an easier development process. There is back and forth communication between the Livin'X platform and the city area and recommendation modules, where requests are made, processed and the results are delivered. Apart from these two, every other module only receives information about new data arriving to the system (e.g., when a user provides new city preferences, these are replicated to the User-City Preferences Module) and process it or use them to update the data structures (e.g., when new user-city preferences are received, new calculations are made regarding the city areas which better adapt to the user's preferences; when a new house arrives to the system, the recommendation engine must update the houses' similarity matrix by adding a new entry and calculating its similarity with the rest).

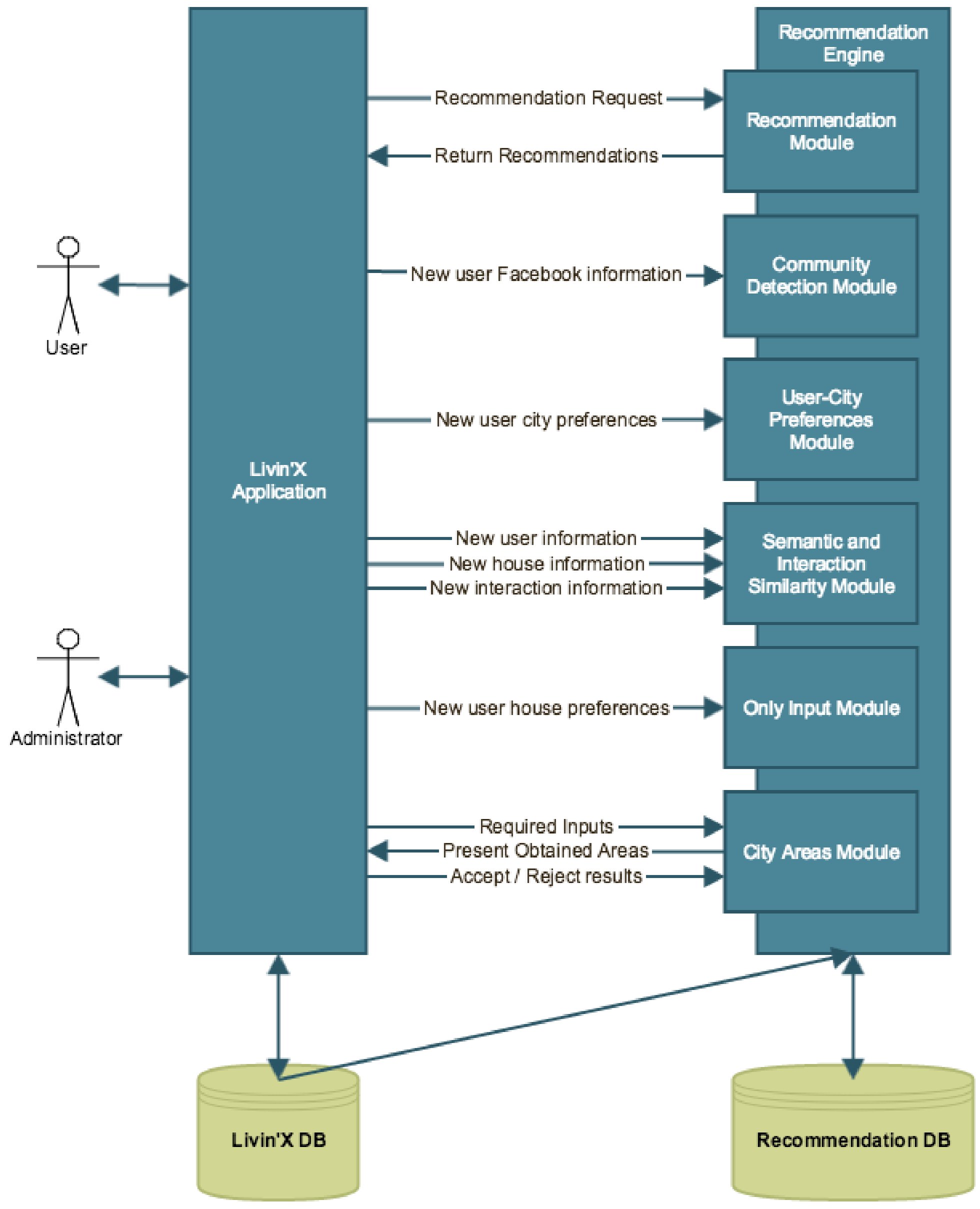


Figure 3.1: High-Level component interaction diagram

The recommendation engine will have its own DB, separate from the Livin'X DB, where it will store all the information used for recommendation calculations (like the matrices detailed in section 3.3.1). However, some data does not require

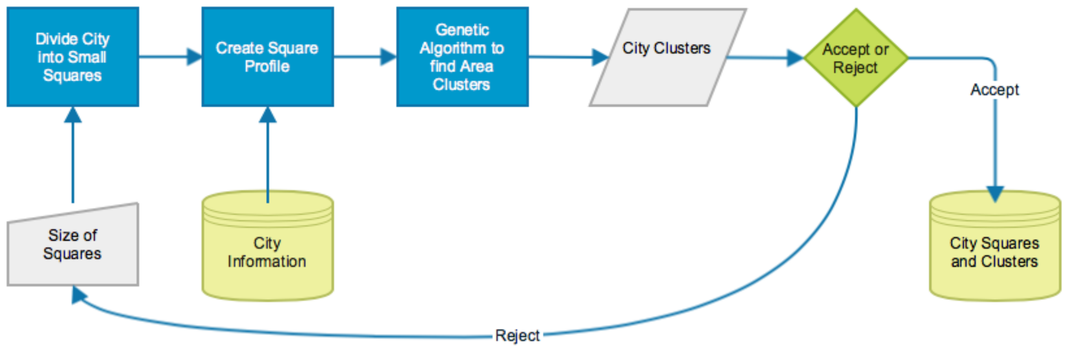

Divide City into Small Squares
Create Square Profile
Genetic Algorithm to find Area Clusters
City Clusters
Accept or Reject
Accept
Size of Squares
City Information
City Squares and Clusters
Reject

Figure 3.2: City Area Clustering

replication to both DBs. For instance, when the engine has a need to check some house's detail, it can access Livin'X's DB to find it instead of having it duplicated in its own DB.

3.2 City Areas Module

This section describes the module where city areas are defined and profiled. Using these areas, the system may relate a user to a given house according to the area where the house is located. This is a very important factor to take into account when choosing a place to live in and one of the core motivators for the creation of the platform.

3.2.1 City Area Clustering

The diagram in figure 3.2 represents how the city areas will be created. The general idea is to divide the city in small squares and create a profile for each of them based on information in the area within the square. The city areas are obtained by maximizing a given cluster's quality function although they are subject to the system administrator approval.

The system starts by receiving input from the system administrator about the size of the squares by which the city shall be divided. Then, it will create a profile for each square based on information about the area: types and reviews of services[1], the security levels[2], types of events[3] and the level of housing[4]. Given

[1]taken from http://foursquare.com

[2]taken from http://www.pordata.pt

[3]taken from http://www.agendalx.pt

[4]information already gathered by Ubiwhere from various real-estate agencies

that the information provided comes in all sorts of different formats, this way of representing the city data allows a more uniform way of handling the data.

The final city areas shall be clusters of the small squares described above. They will be found by maximizing the following cluster quality function for a set of clusters C:

$$qual(C) = \sum_{c} uni(c) \times size(c)^{\beta}$$

where $uni(c)$ represents the uniformity of a cluster, $size(c)$ is the size of a cluster and β is a value that adjusts the importance of the size of the clusters. The uniformity $uni(c)$ can be defined as a linear weight combination (as in [29]) in the following way:

$$uni(c) = \alpha.ser(c) + \sigma.sec(c) + \rho.rev(c) + \epsilon.evt(c) + \lambda.lvl(c)$$

where α, σ, ρ, ϵ and λ represent the weight of their respective functions with their sum being equal to 1. Each function represents the uniformity of a given information within the cluster. The $ser(c)$ function is relative to the types of services in the area, the $evt(c)$ corresponds to the events in the given area and the $lvl(c)$ represents the level of housing within the cluster. These functions deal with information structured in the same way: the percentages of occurrences of a given type in the cluster (i.e., in a cluster, 80% of the events are music concerts and the other 20% are comic conventions). With that in mind, they do their calculations in the same way:

$$function(val_1, ...val_k) = \sum_{i=0}^{k} val_i^2$$

where val_i is the percentage of a given type in the cluster. This way, the more a given type is predominant in the cluster, the greater the cluster's quality is (for example, $0.8^2 + 0.1^2 + 0.1^2$ is greater than $0.4^2 + 0.3^2 + 0.3^2$). On the other hand, the $sec(c)$ function, which deals the the security of the area, and the $rev(c)$, which handles the reviews of the services in a given area, deal with ratings. So, what is desired is a level of discrepancy between ratings within the cluster. That is obtained in the following way:

$$function(val_1, ...val_k) = 1 - \frac{\sum_{i=0}^{k} \frac{|val_i - \overline{val}|}{max - min}}{N}$$

where given a list of ratings, the average of the normalized difference of the ratings with the mean value of the ratings $\overline{val}$ is calculated (max and min represents respectively the maximum and minimum possible rating). It is then subtracted by 1 so that the smaller difference is rewarded. This function obtains greater ratings the smaller the difference between the various ratings within the cluster.

To maximize the quality function, a genetic algorithm (section 2.2.3) without using crossover was chosen. Each genotype will be a vector C containing a certain number of centroids $\{c_1, c_2, ...c_m\}$. With this, and using a clustering operation, the city area clusters (the phenotypes) will be formed. The centroids' value shall be the center of a city square since this reduces the number of possible neighbouring states. The clustering operation will simply be assigning each city square to the cluster with the nearest centroid. This might create situations where a given square is equally distant from two or more centroids. In these cases, the square will be assigned to the cluster where its inclusion will generate the greatest gain in quality (or the least loss).

The mutation operator will be a modification to the vector of centroids. Three different operations can be made, with equal probability: adding, removing or changing a centroid (it is important to note that the centroid vector will not have a fixed length). This allows experimenting with different cluster placement and number of clusters. Survival elitism will be used to pick who reaches the next generation, keeping a small percentage of the best solutions and the rest will be the best new solutions created in the current generation. This allows the best solutions to not be lost and still maintain a good diversity of the population by not discarding solutions that might be worse, to allow a better exploration of the search space.

The main difference between using a genetic algorithm instead of another maximization technique, as presented in section 2.2.3, is the way that it creates a new generation using the whole population, contrary to, say, the hill climbing algorithm, which only explores the neighbourhood of one solution. Random-restart Hill Climbing could be said to have a similar effect but, for example, if the initial population is a and b, where a is a solution with potential and b will not lead anywhere, a genetic algorithm can, in the next generation, discard b and work on different variations of a. On the other hand, at each restart, the hill climbing might not find an equally good solution since it only works with one variation and wanders aimlessly from b without finding a decent solution. The main problem with simulated annealing, applied to this maximization problem, is that it can only process a neighbour at a time, making it very hard to do a good exploration of the large search space.

The difference between this genetic algorithm and others is not bigger because of the choice of not using the crossover operation. This is due to it not being very applicable in this situation. Since the centroid vector C does not have any order in particular, what would end up happening most of the time, if using the crossover operator presented in section 2.2.3 (or some variation of it), is that the resulting vectors would not make much sense. The centroids in one of the offspring could be all close to each other and an area could be left devoid of centroids.

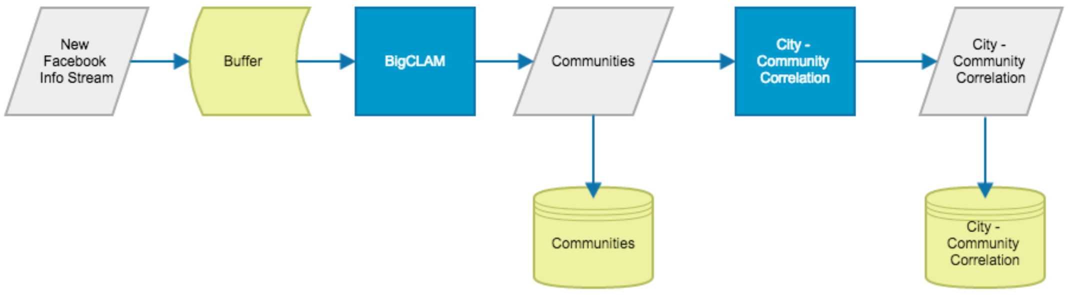

New Facebook Info Stream
Buffer
BigCLAM
Communities
City - Community Correlation
City - Community Correlation
Communities
City - Community Correlation

Figure 3.3: Community Detection

In the end, the city areas are presented to the system administrator, who shall accept or decline. If he/she accepts, the clusters are stored, if not, the administrator can run it again with different inputs to try and obtain better results.

3.2.2 Communities

Diagram 3.3 shows how the system would detect user communities based on Facebook information. To this end, it uses the BigCLAM algorithm [37]. It was chosen based on a study [38] which was already presented on the Related Work section. Basically, it performs as well as other state of the art community detection algorithms and faster on Facebook data. It can detect separated and overlapping communities and communities completely inside other communities.

After the detection of the communities, the algorithm calculates the correlation between the city areas and the communities. Basically, given the a city area a and a community c, their correlation value will be the percentage of c that belongs within the limits of a.

During the BigCLAM and the correlation processing, there will be a buffer that will accumulate new data that comes to the system. When the processing ends, the new batch of information will be processed and the buffer will be cleared.

3.2.3 User City Preferences

Diagram 3.4 shows the process of creating a user's city area rank based on the city preferences the user has provided. It uses Cosine Similarity (section 2.2.1) to measure the similarity between each city area and the user preferences. If the user has given his Facebook information and there is enough information to be relevant (e.g., the communities obtained for a user who only has one friend might be irrelevant), a matrix is created with a weighted linear combination, where the

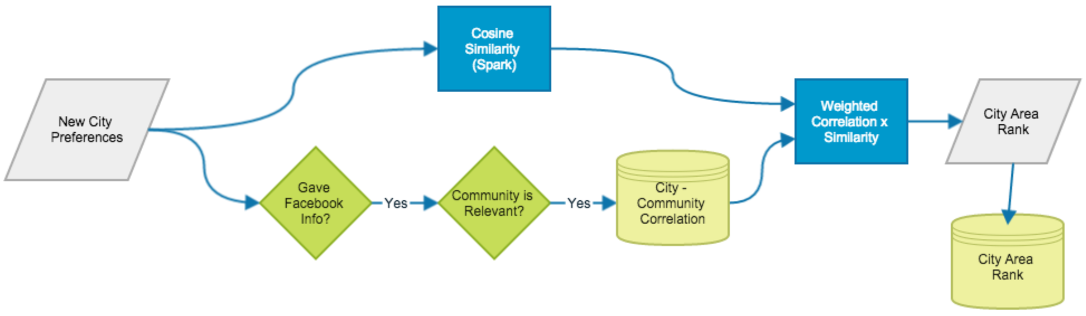


Figure 3.4: User City Preferences

value $val(c)$ for a user's city area c will be:

$$val(c) = \beta.userPref(c) + (1 - \beta).community(c)$$

where $userPref(c)$ is the value obtained from the cosine similarity, $community(c)$ is the highest correlation value a user's community has for c and β is the weight given to each factor: $\beta = 1$ means only the similarity will be used and $\beta = 0$ means only the community correlation will be used. If the user's community information deemed irrelevant, only the similarity measure will be used.

3.3 Interaction and Semantic Similarity Module

Diagram 3.5 shows a section of the system where two types of similarities are calculated:

- **Interaction Similarity:** Similarity based on how the users interacted with the houses. When the same users show interest in the same two houses, they are considered similar;
- **House Similarity:** Similarity based on the houses' characteristics.

Both these measures are calculated using cosine similarity and then the results are combined using the weighted linear combination equal to the one used in section 3.2.3. A buffer system like the one in section 3.2.2 is used to store the incoming request while all the other ones are processed.

The similarity calculation is made using a technique already implemented in Apache Spark called Dimension Independent Matrix Square using MapReduce (DIMSUM) [40]. The main idea is to save time during the calculations by only comparing items that are likely to be similar. It uses the MapReduce model to perform the algorithm's calculations. Given a matrix A with $m \times n$ dimensions

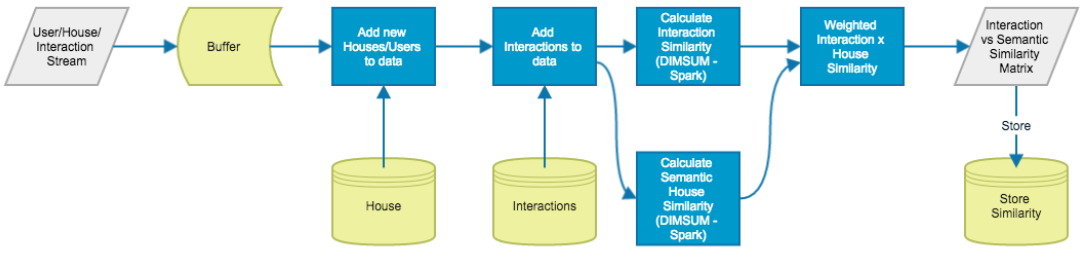

User/House/ Interaction Stream
Buffer
Add new Houses/Users to data
Add Interactions to data
Calculate Interaction Similarity (DIMSUM - Spark)
Weighted Interaction x House Similarity
Interaction vs Semantic Similarity Matrix
Store
House
Interactions
Calculate Semantic House Similarity (DIMSUM - Spark)
Store Similarity

Figure 3.5: Semantic and Interaction Similarity

(with m much larger than n) and considering $a_{i,j}$ an entry of the matrix, the *Map* function emits the product of $a_{i,j}$ and $a_{i,k}$ with a certain probability, it being higher, the more likely elements of column j and k are to be similar. The *mapper* emits the result with the key (j, k). The *Reduce* function, for each key, calculates the similarities between j and k. This method reduces the complexity of the similarity calculation by a significant amount compared to a naive approach (i.e. calculating the similarity of all n elements) with a low error value, as proven by the authors.

This module, apart from handling new user interactions, also deals with new users and new houses. More specifically, it updates the data structures with the new information before the similarity calculations.

3.3.1 Matrices Values

The two resulting matrices from the similarity calculations will have $h \times h$ dimensions, where h is the number of houses in the system and each value of the matrices is the similarity between two houses. However, what has not been explained is the information matrices that feed the similarity calculations.

There are several information matrices The **interaction matrix** I will have $u \times h$ dimensions, where u is the number of users and h the number of houses in the system. The element in $I_{i,j}$ will be a value of the interaction between user i and house j. The element will be the summation of various factors divided by the total number of factors. Theses factors are:

- A value between 0 and 1 relative to the number of visits of a user to a given house. For example, if the maximum number of visits of a given user u to a house was 10, and u visited house h 5 times, the value for h would be 0.5;
- A value of 1 in case the user marked a house as favourite, 0 otherwise;
- A value of 1 in case the user contacted the owner of the house, 0 otherwise;

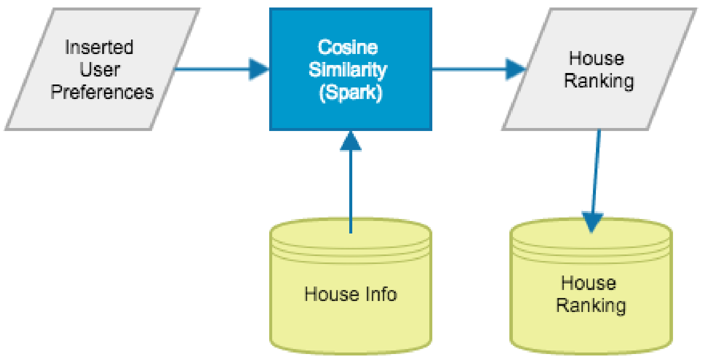


Figure 3.6: House Ranking

- A value of 1 in case the user browsed the images of a given house, 0 otherwise.

Although this value might not be an exact representation of a user's preference for a given house, it is a good approximation.

The **houses characteristics matrix** will have $h \times c$ entries, where h is the number of houses and c is the number of characteristics a house can have, assuming values of 0 and 1. It is pretty straightforward for characteristics like garage, lift or fireplace, it has one or it does not, but for others, like number of rooms, an entry for each possibility has to be created. So, if the possibilities are $1, 2, 3, 4$ or 5+ rooms, five entries will be created to represent them. If the house has 2 rooms, the entry for 2 rooms will be 1 and the rest will be 0. More complicated is the case of values such as rent or buy price which have a lot more different entries than the number of rooms. In this case, a window of values will be created, with the size of the interval to be determined. For example, if the rent of an apartment is €250, than the entry for rents between €200 and €300 will be 1 and the rest of the windows will be 0.

3.4 Only Input Module

The module represented in diagram 3.6 is used when there is not enough information about the users interaction with the system (i.e., when the user starts to use the system). It calculates the similarity between the user preference relative to the house's characteristics and the houses in the database. This similarity is used as a house ranking system.

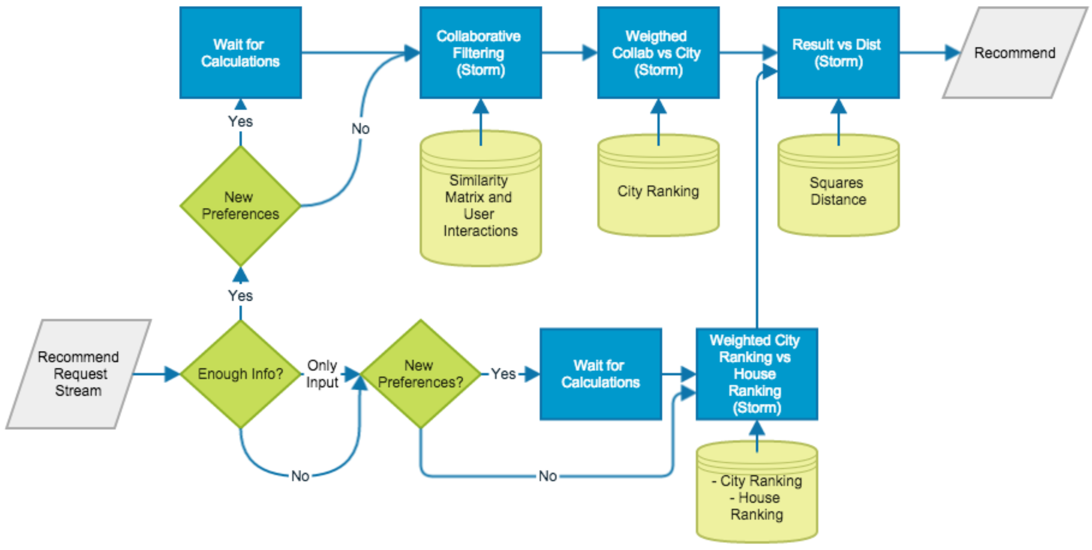


Figure 3.7: Recommendation Processing

3.5 Recommendation Module

Diagram 3.7 shows the piece of the system where the processing of the recommendations occur. When a recommendation request comes in, it can be handled in three different ways:

- **Complete:** When there is enough user interaction information, it will be used to perform the best recommendation possible;
- **Only Input:** When the only information available are the preferences the user has given, they will be compared with the houses and city characteristics to create a recommendation;
- **No Input:** When the user provides no input, a general ranking, independent from users' inputs, must be used to recommend places to users.

When using the Livin'X application, recommendation requests usually come right after the user inserts its preferences. So, before doing the calculations, the system might need to wait for the processes described in section 3.3 to complete. It uses the resulting house similarity matrix S together with the users interaction information to perform CF. To that end, it uses the following formula (based on the formula used in [29]):

$$V_{a,t} = \frac{\sum\limits_{h} (V_{a,h} \times S_{h,t})}{\sum\limits_{h} S_{h,t}}$$

where V is a vector containing the value (predicted or not) of a given house to target user a, t indicating a house without a user interactions (i.e., to be predicted) and h being a house already with a user interaction value. $S_{(h,t)}$ is the similarity value between houses h and t. This way, the system will use the users' collective mind to learn what houses might interest a given user through his and similar users' interactions. Following this calculation, the resulting predictions will be joined with the city area ranking with a given weighted linear combination as presented in section 3.2.3. Finally, the same weight function will be used to join the results obtained from the last operation with a value relative to the distances between the houses and the user's PoIs. This value will be 1 if the house is within the range the user asked, a relative value between 0.5 and 1 in case the house is outside the range but within double of that range and 0.5 if the house is outside the double of the value of that range. It can be noticed that this value does not go below 0.5. This is intended so that a house can not to be completely disregarded just because of its location. All of this will be built on top of a Storm topology (section 2.4.1), where the calculations will be distributed throughout various bolts for fast computation.

In case it is a new user, only the inserted preferences will be available. In this case, the house ranking calculated in section 3.4 will be used, joined with the distance value function presented above.

In cases where the user gives no information, the current solution used by the Livin'X platform will be adopted, where it assumes a set of default inputs for the user following general ideas of what a good living place is (example: near central areas of the city but away from noisy zones like an airport). These inputs will then be handled as the previous case, where the user only gave input but has no interaction with the platform.

3.6 Scalability

Apart from returning good recommendations, it is important to guarantee the efficiency of the engine, even if the number of users rises exponentially. For that purpose, two main ideas were kept in mind while developing the architecture of the system:

- **Off-line Computations:** Whenever possible, do calculations off-line (i.e. don't do all the calculations when a user makes a request). This allows for the user to receive faster recommendations since necessary computations have already been made. One example of this is the house similarity matrices, which are calculated off-line. Since there is only one matrix that is used for the recommendation calculations of all the users and it is not required for it to be completely updated (i.e. an out of date matrix could only be missing a few houses), it saves a lot of time to have it computed off-line while still giving valuable information. So, in conclusion, doing off-line calculations whenever possible and feasible allows for faster calculations of recommendation requests.

- **Distributed System:** Nowadays, it is a lot cheaper to scale out (i.e. add a node to a system) than it is to scale up (i.e. add more resources to a node). With that in mind, the methods chosen to do the computations should be able to them concurrently and the technologies chosen should be able (or meant to) work in this kind of environment. Examples of this in the designed architecture can be seen in the usage of Collaborative Filtering and Storm.

With these measures taken and the necessary resources available, the system shouldn't have much trouble returning fast recommendations to a ever growing community of users.

Chapter 4

Development

This chapter will report the work made in the development in this project, the difficulties encountered and the steps taken to overcome them. The development process follows closely the plan defined in the architecture (chapter 3), so only the divergences from this plan will be mentioned. It is important to note that the work developed for this project is result driven, so integration with technologies such as Apache Spark and Storm have not been implemented, being in the plans for future work. The methodology used throughout this work can be seen in appendix A.

Before beginning with the programming itself, it is necessary to obtain new, relevant, information and to handle the information that already exists. After that, work on creating the city areas can start. Finally, the various types of recommendations can be made and tested.

4.1 Information

In this section, the information to be used in this project will be fetched (if necessary) and analysed, its problems will be reported and the steps to handle them will be defined.

At the conception of this project, the following information, vital to the creation of recommendations, was already obtained by Ubiwhere:

- **Houses:** There are 5400 houses in the database, where, for each one, there is information about its size, typology, features, rent and sell price, house state and type, year of construction and location. It may also include photos of the respective house;

- **Services:** 7312 city services, such as restaurants, bars, transit stations, hospitals, etc. For each service, user ratings may also be included.

So the first step of this project is to obtain the rest of the information that is necessary to feed the recommendation engine, should it follow the designed architecture. That information is the security of the different areas of the city, the events that happen throughout Lisbon and information about the users' interaction with the platform.

4.1.1 Security

Security information helps in the definition of the city areas, as seen in section 3.2.1. It would be more desirable to obtain it at a neighbourhood level but at the moment it is only available at a municipality level, so its use as a way to distinguish areas is limited.

The information is provided in a spread sheet which is exported to the database. The information comes in crimes per 100.000 people. In order to be able to categorize regions in different levels of security, a rank of 1 to 5 is created where, given a value range from the minimum and the maximum value, it is divided in 5 equal parts where each corresponds to a value. This information is obtained through PORDATA [12], a website that provides different kinds of information of areas of Portugal, and the results of the categorization can be seen in appendix E.

4.1.2 Events

As with the security information above, the information of events that happen in Lisbon (e.g. Music shows, theatre, art expositions) is used to define city areas. For this purpose, the import.io web crawler tool [6] was used. It is very easy to use where, given example pages from a website, the user indicates, on the web page, where the information is located and the web crawler visits all of the pages on the website and retrieves the information requested. In this case, the site used was agendalx.pt [1], which promotes events that happen in Lisbon.

The information obtained is the name of the events, the type (open air, arts, cinema, science, children, dance, fair, literature, music, theatre, guided tours), the date and the location. In this last case, only the street name is provided by the site, so, using it and the Google Maps API [4], the latitude and longitude of the event is obtained.

4.1.3 Users

User information, being either their preferences, Facebook friends or their interaction with the platform should come from, obviously, Livin'X users. The problem here is that the platform will not be launched to the public by the end of this work, so it is not possible to evaluate the results of the recommendation engine with the multitude of users that in the future shall use Livin'X.

With this in mind, multiple user profiles will be created to validate the platform. This process will be detailed during the recommendations testing, in section 5.

The interaction with the platform was something that, in the old engine, was not a factor. So time was spent in gathering and storing this information (house visits, house favourites, user-owner messages, image browsing).

4.2 Real Estate Dataset

The visualisation of the data is an important step in the developing of a system of this kind. It allows the finding of some possible limitations (and take steps to avoid these) and to make the development process more efficient (i.e. data might show that a certain method will not work, so might as well not waste time with it or vice versa, it might show that a method might work really well on this kind of data).

4.2.1 Houses

There are 5400 houses available to be recommended. While it is a big number, given the diversity that can exist (e.g. different typologies, features, prices and locations), the lack of houses with a given set of characteristics might make it difficult to recommend anything to a user. This is a situation that should only exist while the platform is not launched, so, while it will not be a limitation in the future, for the purpose of testing and obtaining results, a watchful eye has to kept on this situation.

In figure 4.1 it can be seen that a predominance of houses in an old state exists. While there is still a significant amount of new houses, the rest is almost negligible, making it hard to appear in the recommendations.

Figure 4.2 shows that most of the available houses are apartments and that there are no rooms for rent of any kind. In figure 4.3 it can be seen that most houses in the database are for sale (even though selling and renting are not mutually exclusive).

Figures 4.4, 4.5 and 4.6 show more balanced distributions, showing that the choice of each of these factors (typology and sell or rent price range) should not affect the quality of the recommendations during testing.

Finally, figure 4.7 illustrates how the houses are distributed throughout Lisbon, where a more intense shade of red indicates stronger presence of houses in that section. It is clear that the majority is in center Lisbon, with a decent amount

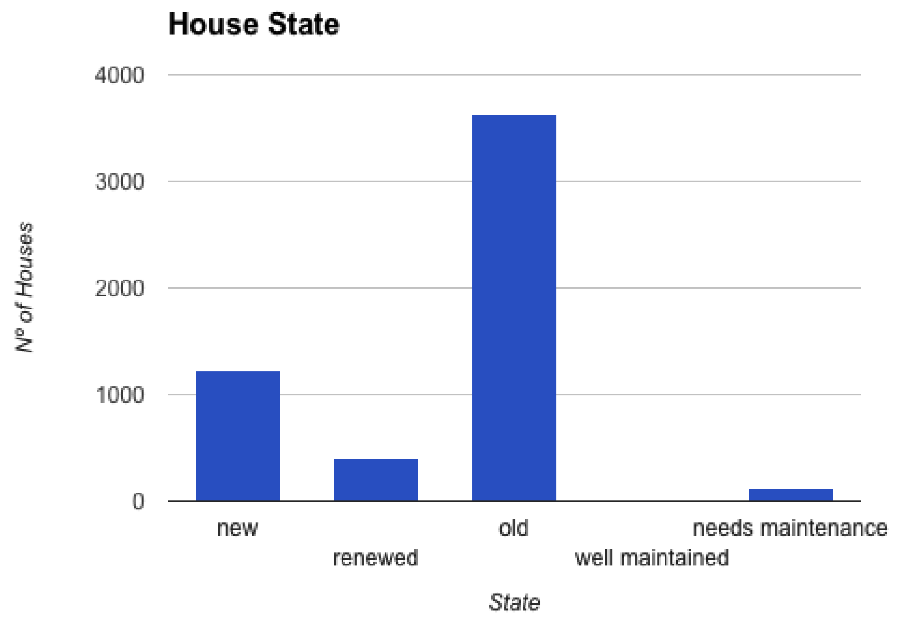


Figure 4.1: Number of houses by housing state

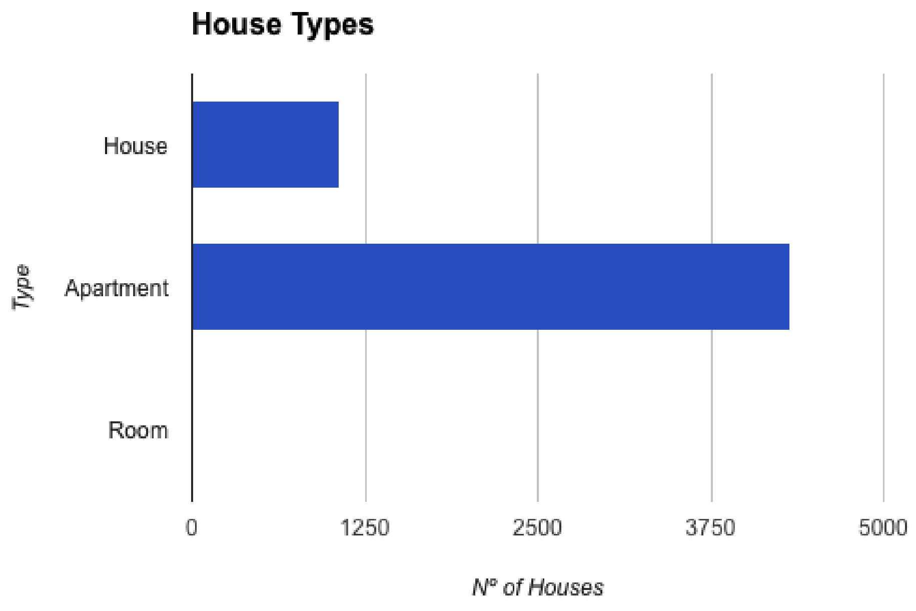


Figure 4.2: Number of houses by type

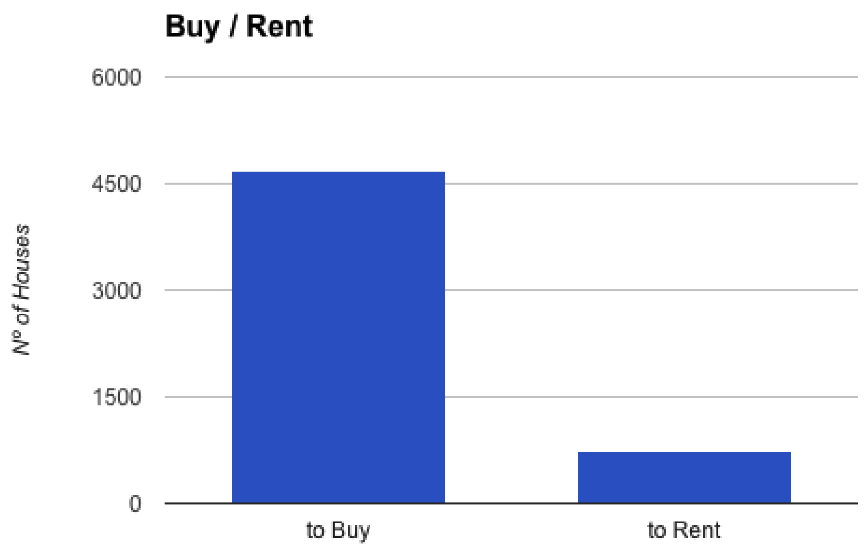


Figure 4.3: Number of houses on sale and to rent

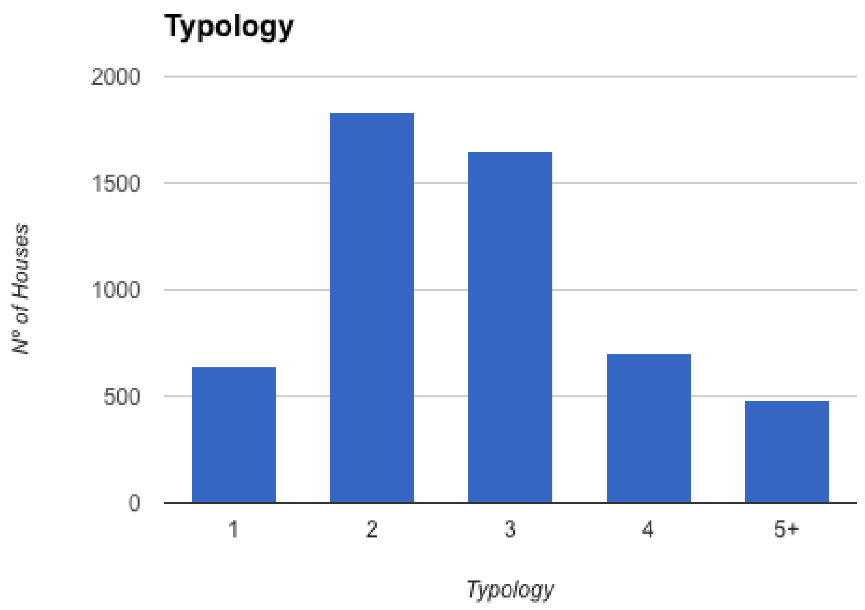


Figure 4.4: Number of houses by typology

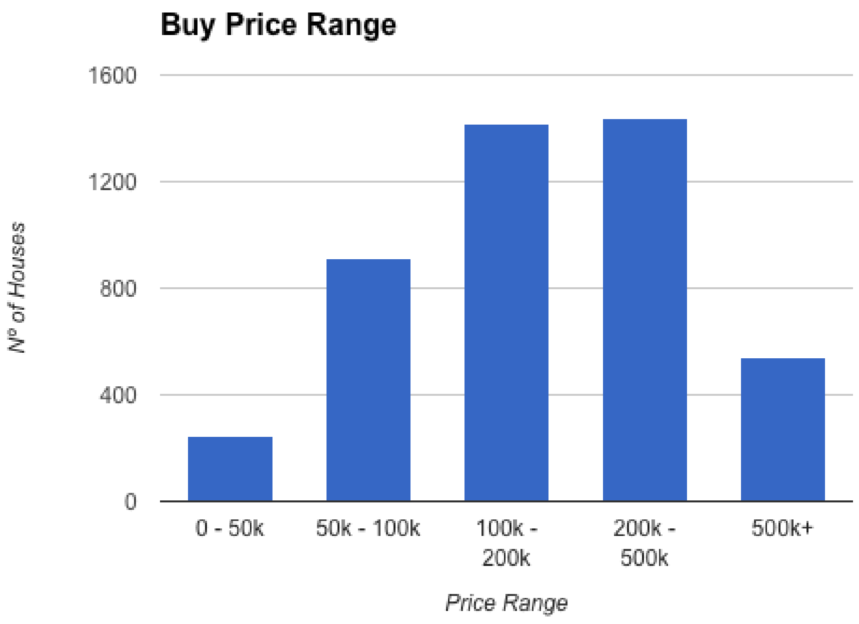


Figure 4.5: Number of houses on sale per price range

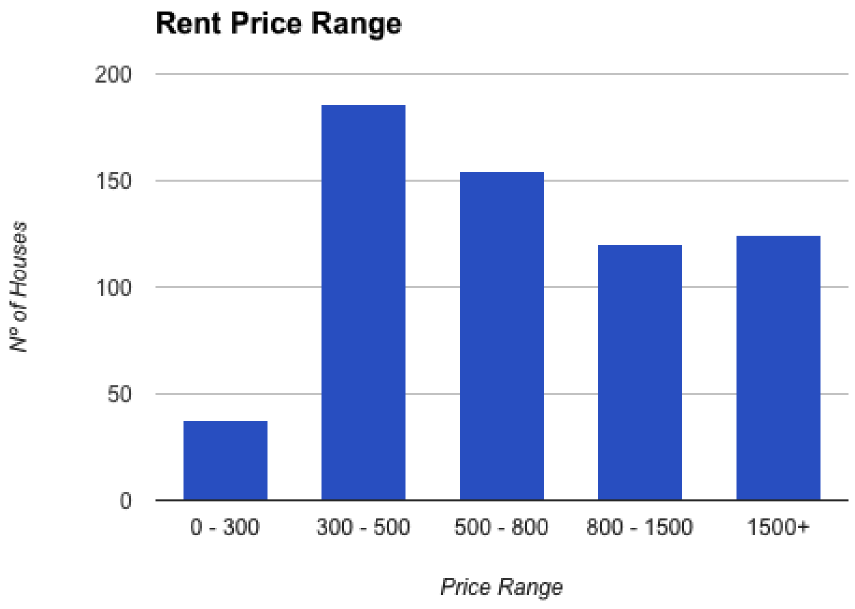


Figure 4.6: Number of houses for rent by price range

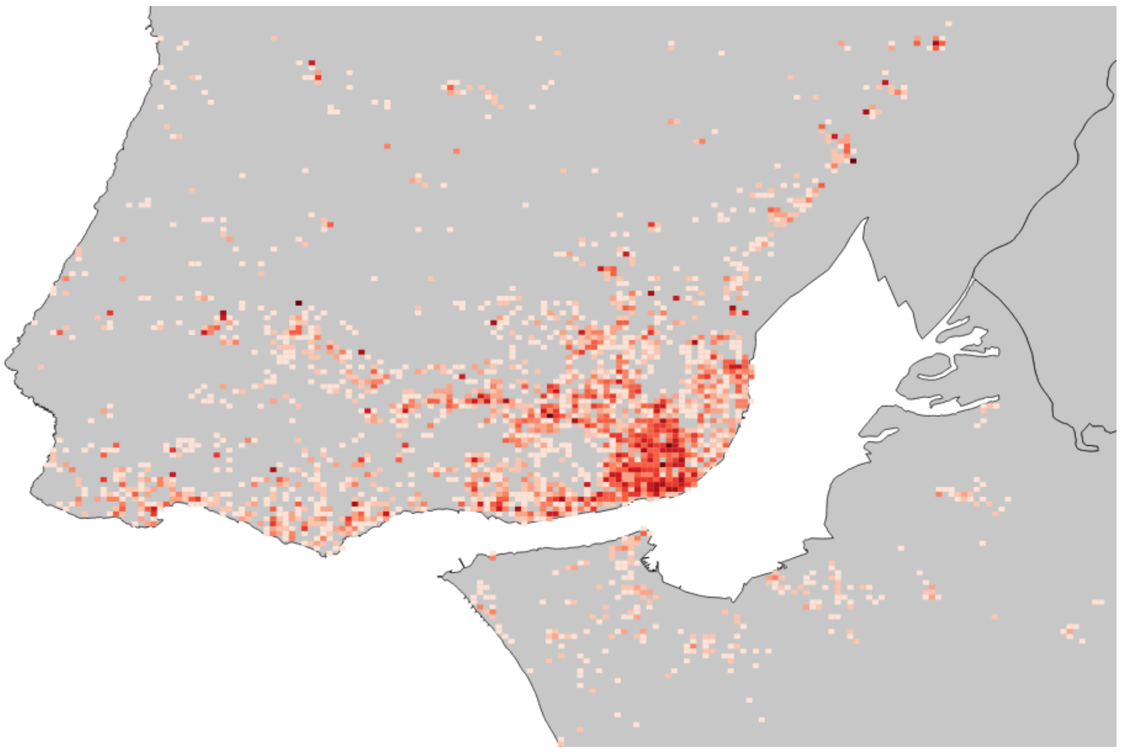


Figure 4.7: Distribution of houses throughout Lisbon

to the west of the center of the city. Outside these regions, there are few, sparse, houses.

With this, it can be concluded that the choice of house state, type, whether to buy or rent and the location of a house can, during testing, affect the quality of the recommendation and should be taken into consideration when analysing the results. This, however, reflects the actual availability of houses for sale/rent in Lisbon.

4.3 City Area Preferences

This section will report the development of the City Areas Module designed in section 3.2. This part of the project took a lot longer than expected due to three main reasons:

- **Programming:** The code for this module was all made from scratch and is large, making the programming of this module take in itself a long time;
- **Debugging:** The complexity and the scale of the implementation make it harder to debug at later stages;
- **Efficiency:** While it is not imperative that the genetic algorithm that computed the city areas be very fast, since it should be run only from time to time, it should run in a decent time for testing purposes. Initially, a run of 100 generations could take from 5 to 10 hours. Since 100 generations was proven not to be enough, it was imperative to make this process faster. In the end, 400 generations could be run in 5 hours. The process of making the algorithm more efficient also took time from the rest of the project.

4.3.1 City Area Clustering

The desired result from the genetic algorithm was to obtain distinguishable city areas, with smaller, more detailed areas where there is a bigger concentration of information (e.g. center Lisbon) and wider areas in sparser regions. The parameters to be tested are:

- **Uniformity weights:** The weights used to compute the uniformity of an area (presented in section 3.2.1). This value will not be presented in the various tests made because they were initially given the same values and were not tinkered with much;

- **Size importance value:** The value that changes the importance of the size of the regions to the computation of the quality of a given set of areas (also presented in section 3.2.1);
- **Population:** The number of individuals maintained throughout the genetic algorithm's computations. Since this value has always been set at 100, it will be omitted from the test table in 4.1;
- **Initial Centroids:** The number of areas existing at the beginning of the genetic algorithm's computations;
- **Generations:** The number of generations that the genetic algorithm will run;
- **Tournament Size:** The size of the tournament used to select individuals that will reproduce in each generation;
- **Survival Rating:** Percentage of individuals that survive through the generation. As with the uniformity weights, this value was set to 5% initially and was never changed throughout testing.

Table 4.1 details the parameters used for each test. The first test generated large areas everywhere, which is not desirable. To counteract this, for the following test the size weight was reduced in order to not award so much score to big areas. Additionally, the initial centroids were set to 150, which helps making the generated areas smaller, given that there are more areas occupying space. This did help a bit in obtaining smaller areas, but not enough. The following test (3), the size weight was reduced even more. Here, it was observed that, from the initial centroids, throughout the generations, the algorithm tends to increase the number of areas without showing signs of stopping. To try and fight this, the initial centroids were upped to 300. Also, it was noted that in the tournament selection, individuals with a rating below 40 were rarely selected, so the size of the tournament was reduced, initially to 4 and finally to 3, as it remained throughout the rest of the test, since it gave satisfactory results.

The results obtained (4) reduced the size of the areas, but in central Lisbon it was still noted that there were areas with too much content while others had almost nothing. Also, the constant rising in number of centroids continued. In test 5, the initial number of centroids were upped to 500, and, while the rise of number of centroids was less steep, it still happened. Also, the areas throughout the region were all very small and nonsensical. This obviously was not the correct path. At this point, a problem was noticed that areas in the outer region that had zero content in them contributed nothing to the overall score (because their uniformity

Table 4.1: Test parameters

Test Id	**Size Weight**	**Init. Centroids**	**Generations**	**Tournament Size**
1	1	100	100	5
2	0.5	150	100	5
3	0.1	150	100	5
4	0.1	300	100	4
5	0.1	500	100	3
6	0.1*	100	30	3
7	0.1*	300	100	3
8	0.1*	500	100	3
9	0.1*	200	100	3
10	0.1*	200	200	3
11	0.1*	200	100	3
12	0.1*	300	100	3
13	0.1*	300	200	3
14	1*	150	200	3
15	1*	200	200	3
16	1*	200	300	3
17	1.02*	200	300	3
18	1.02*	200	300	3
19	1.02*	200	400	3

score would be zero) but still wasted computational power just by existing. Also, they were undesirable since they contributed nothing to the recommendations and did not allow bigger regions to form in the sparser areas. To try and overcome this, in the following (6 and 7) tests, empty regions would affect negatively the overall score of an individual. This way, the genetic algorithm should tend to cluster these small, empty regions to bigger ones with some sort of content.

While test 6 was inconclusive due to having few generations, in test 7 it was noticeable the desired effect of penalising empty regions. There was still the problem that the algorithm did not tend to any solution, the number of areas was still growing constantly throughout the generations. In test 8 the number of centroids was raised to try to address this issue, but with no success. In test 9, instead of penalising empty areas, it penalised areas with a number of content less than 10 (be it number of houses, services, etc) since areas with that little content were as undesirable as empty areas. The results were positive, with the number of areas not growing much. There were other problems, though. Even though the number of areas stopped growing uncontrollably, the best and average fitness value of each generation continues to grow linearly. It was also noted that, even though Lisbon was composed of small areas (as it is intended), two or three of these areas contained all of the content while others were nearly empty. At this point, a new idea was tried. During the mutation process, besides the already implemented creation, deletion and change of centroids, add two more operations: one that merges areas with low content and another that splits areas with large amounts of content.

In test 10, besides implementing the new idea mentioned above, the number of generations was also upped to 200, in order to try to make the fitness values tend to some value. The results were mostly positive, with wider areas outside of Lisbon and smaller areas in the center, although it still was not perfect, there were still areas with too much content in central Lisbon. In image 4.8, it is noticeable that the fitness values don't tend to anywhere. To try and check which factor was making the results better, the number of generations or the split/merge in the mutation, a test 11 used the initial number of generations, but the results weren't great. Test 12 tries the same with more initial centroids but the results don't vary much.

Test 13 raises the number of generations to 200, since it is clearly necessary to raise this value. There is still no change in the behaviour of the fitness evolution. This time, taking a closer look to the number of areas of the best result in each generation (figure 4.9), it is possible to see that, while it goes up and down a lot, the value tends to rise. Taking a closer look at one of the articles ([18]) used to create the function that computed the value of a set of areas, they state that the value that measures the importance of the size of the areas should not be smaller than 1. It makes sense, since, if the value is lower than 1, the result will be better

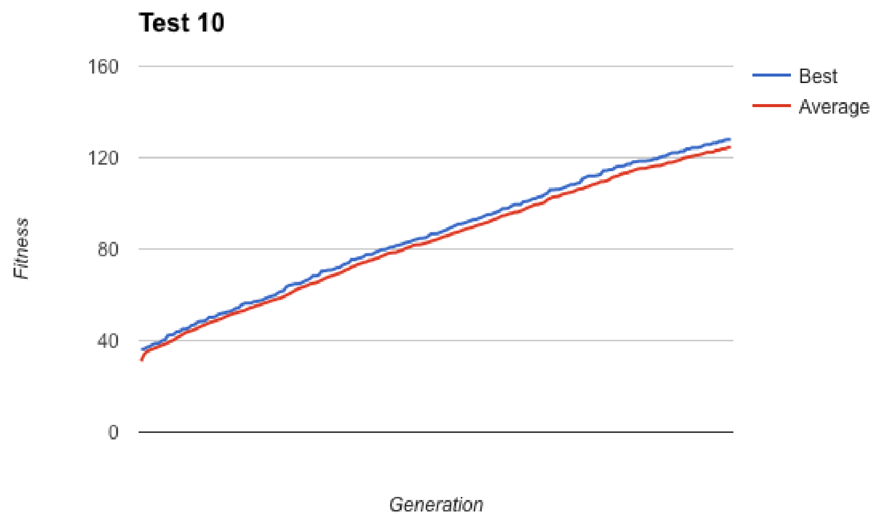


Figure 4.8: Test 10 fitness evolution

the larger number of areas there is, since the size value will not be relevant.

On test 14, the size weight was set to 1. The number of areas went down through the generations, which is expected, since in the first generations the algorithm tends to merge the sparser areas. Either way, the fitness values still don't tend to anywhere. On test 15, where the initial generations were set to 200, the results were slightly better and it was possible to see that the fitness values stop raising so fast at some point. Still, it clearly needs more time to run.

The rest of the tests returned good results, where it was noticeable that, when the size weight was upped to 1.02, the areas obtained in more denser regions were better (i.e. the content was balanced between them, no areas with too much content and others with too little). In image 4.10 can be seen how the fitness changed through the generations. Although it is for test 19, it is similar to the 16, 17 and 18. It starts to tend to some point, but does not quite reach it yet. Having more generations could help find that point, but resource and time limitations make it impossible to go any further. Similarly, in image 4.11, it can be seen how the number of areas change. It generally starts by going down steeply (while the sparser areas are being merged) and at some point that decline stops and starts to raise slowly (when the areas in more denser regions are being formed).

The areas generated in test 19 can be seen in image 4.12. The areas are as they were expected and desired, although only when the system is tested with real users can it be seen if the areas are beneficial or a detriment to the recommendations' quality.

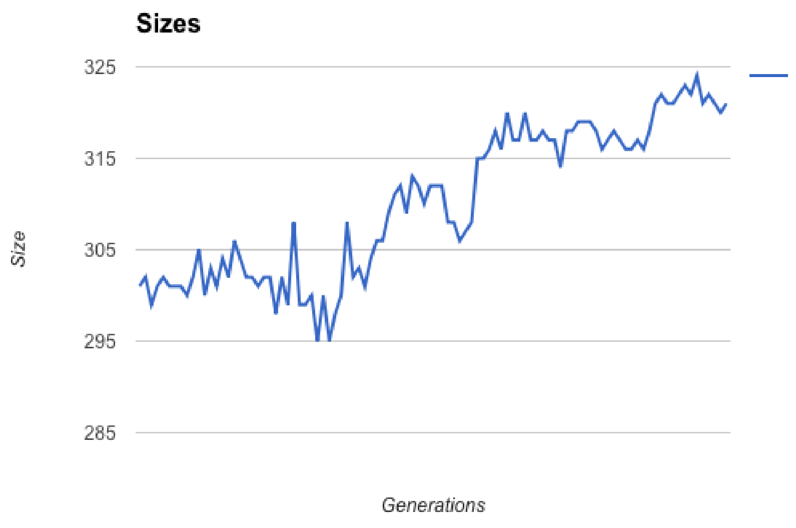


Figure 4.9: Test 13 number of areas through the generations

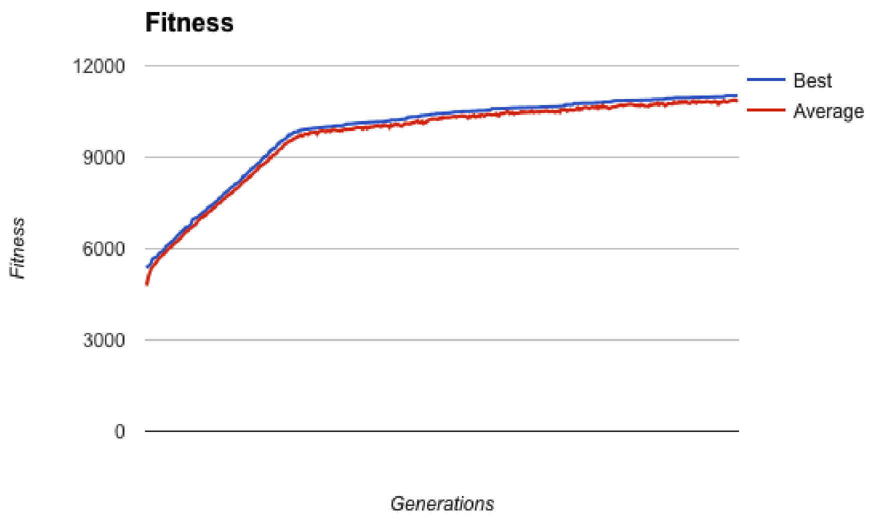


Figure 4.10: Test 19 fitness evolution

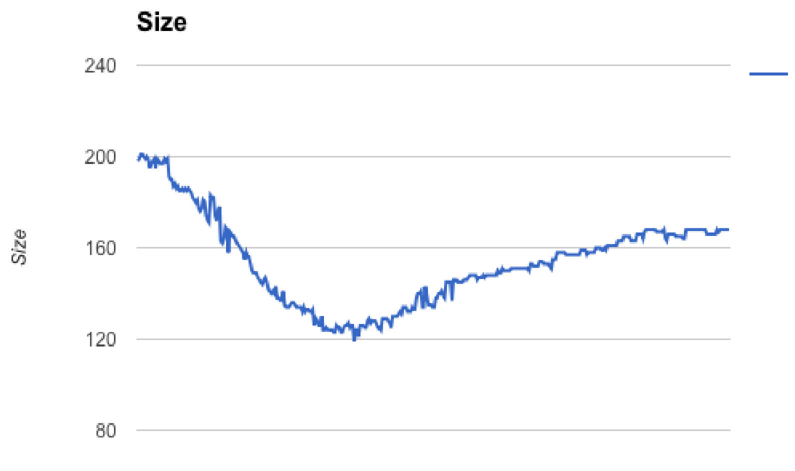


Figure 4.11: Test 19 number of areas through the generations

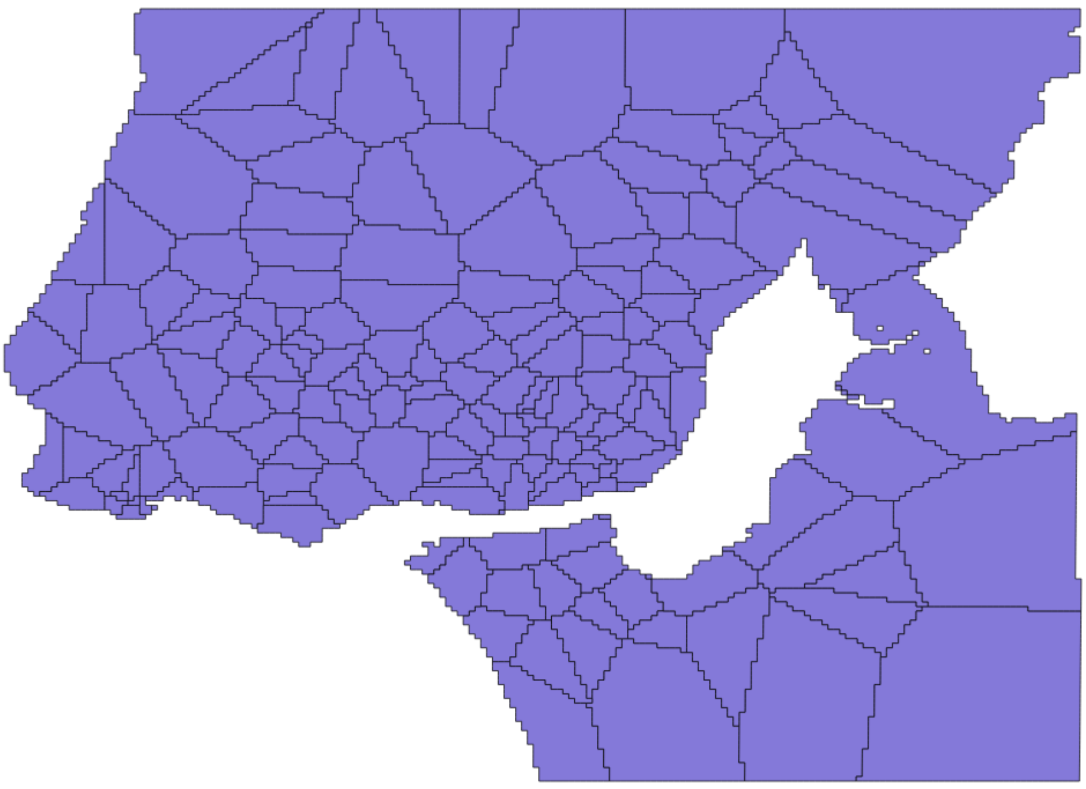


Figure 4.12: Areas generated in Test 19

4.3.2 Communities

The computation of communities of users in the Livin'X platform was not developed in this project. The reason for this is that there are no real users using the platform yet, so it is impossible to obtain real communities in Lisbon. Fake users and connections between them could be generated to test the quality of the algorithm in detecting the communities, but it still wouldn't be possible to test the communities contribution to the general recommendations. This contributed to lowering the priority of the community detection algorithm, which was then chosen not to be implemented. It is, as such, future work for the platform.

4.3.3 User Area Preferences

This part of the module intends to compute the level of preference each user has for each area of the city. Since the community calculation has not been developed, as mentioned in the section above, it is simply a calculation of the similarity of the user's preferences with the areas' characteristics. The quality of this factor will be verified during the user tests.

4.4 Recommendations

The recommendation engine designed in section 3.5 was implemented as it was planned, apart from not using the technologies foreseen to improve scalability and performance. Since the main objective is to obtain results from the recommendation engine, the use of the technologies was not fundamental to this stage of the project. Either way, even without these technologies implemented, the developed engine proved to be more efficient in its calculations than the previous one.

The tweaking of the engine was reported in chapter 5 since the quality of it's results come from the users' feedback.

4.5 OrientDB

One object of interest of this project was how OrientDB would perform in a system of this kind. The most positive mention goes to its query language, adapted from SQL, that is very intuitive and easy to use, allowing the performance of a large amount of different types of queries without much difficulty. The graph structure allowed for complicated, relational, queries to be made very quickly. One the other hand, due to it's graphic nature, the creation and deletion of millions of edges can take very long, more than half an hour. This didn't have a great effect on this system because computations of this kind were done off-line.

Chapter 5

Testing

In this chapter, the tests performed over the engine will be reported, as well as the changes made to it following the tests' feedback. The results from both the previous and the current recommendation engine will be compared using a number of metrics:

- **Response Time:** Time spent computing the recommendations;
- **Recommendation Quality:** The quality of the recommendations through a user's perspective;
- **Presented Area Quality:** The quality of the recommended areas.

While the first metric is an objective value, the last two are not. It is hard to put a value on the quality of the results of a system of this kind if there is not a solid user base (which there is not, in this case) to evaluate them. So, various user profiles were created (appendix F) in order to be used to create realistic interactions with a system, following a clear set of desires from someone looking for a house. When creating these profiles, the data limitations mentioned in section 4.2.1 were taken into account. The profiles will be used in two stages:

- **Developer:** In order to test if the methods are correctly implemented and if they are behaving as intended, the developer will use the user profiles to verify the results from the engine;
- **Test Users:** Volunteers from inside and outside Ubiwhere will help test the engine in order to obtain different experiences from users using the platform. Since none of them are (probably) looking to buy a house in Lisbon, they will also follow the user profiles when using the platform.

There are different types of recommendations to be tested as well: when the only information about the user in the platform is the preferences the user inserted and when there is also interaction information which can be used to recommend houses similar to other houses the user has shown interest in. In the later category, there are also two different situations to keep in mind: if the house similarity calculations use the overall user interactions or not. This later case should happen in the early stages of launching the platform, when the amount of user interactions with the platform is not great enough to obtain decent results from them. In this case, only the houses' characteristics are used in the calculation of the house similarity.

5.1 Developer Testing

Before reporting the individual tests, some general results will be presented. As for the recommendation computation time, the developed the engine in this project is clearly superior. In the previous engine, calculations took up 20 seconds or more, while in this one the calculations not using interactions are almost instantaneous, the one using interaction are generally under 10 seconds, although there were cases they got up to 20. Still, these times have the potential to be much smaller when technologies such as Apache Spark or Storm are implemented. Another note is that, while the focus of Livin'X is to use the areas the houses are situated in to generate recommendations, the new engine presents the houses ordered by their individual quality as opposed to the old one, that only uses the quality of the areas to order the houses. This allows, when using the new engine, for a user to get a more direct access to the houses with the most quality quicker.

As for the house recommendation results, the difference in quality of the old engine to the new one when not using interactions is not big, presenting houses according to the user inserted preferences. The only area where it differentiates more is in the areas presented, since the new engine considers more factors, it is able to present areas more to the users' preferences. On the other hand, when the new engine uses the users' interactions to calculate the recommendations, the difference is notable, presenting results of better quality by learning from the users' interactions and adapting to their tastes. Finally, on a last note, both engines suffer when the sort of house the user is looking for does not have much presence in the database. But, while the new engine tries to look for the closer results it can to what the user asked for, the old engine might not even return any kind of results.

To start the tests off, the user profile 1 (which can be seen in appendix F) was used. Looking at the house data in section 4.2.1 and comparing it the user 1 preferences, it can be concluded that there shouldn't be a problem in finding a house, since there should be plenty available that satisfy the preferences of the user. The first recommendations use only the inserted preferences of the user and the results are as expected, returning houses within the user's input, without much exception. In image 5.1, it can be seen the results on the map, where the green dot represents the user's point of interest, to which he wants to be close to, and the other dots are recommended houses where the darker the dot, the more recommendation value it has. It can be observed that most of the recommendations (and the higher valued ones) are in the areas near the green dot but that there are many other houses recommended in further away regions of Lisbon with lower values. The results included houses with no bathrooms, and houses with a garage clearly stood out because this information is clearly visible. The visibility of the information may affect how a user perceives a recommendation, influencing the results measured, something to account for when testing the engine with real users.

Having acquired some user interaction information, it is possible to make recommendations based on them. In this case, only the house characteristics similarity was used in the similarity matrix since there is no other user interaction information. The first results obtained with this method were of mixed quality. Observing the 20 first results, there were three or four houses which were good recommendations, either for being cheaper houses or new/well maintained houses within the price range. Another positive note was that no houses without bathroom were presented, showing that the engine could "understand" the user's desires. On the other hand, there were a lot a houses with only one room recommended. It is expected, and even desired, that the recommendations don't follow all the specifications the user demands (i.e. as said before, a user looking for old houses certainly will not complain if the engine finds one in good state within the price range), someone looking for a house with at least two rooms will not be happy to find a house with one room. In this case, this happens because, given the low price range, there are a lot of houses with one room under all the other user preferences indicated. To try and avoid this behaviour, the function that calculates the similarity between houses was changed so that the typology contributed to half of the similarity value. This time around, the engine returned a lot less one-room houses, presenting results closer to what the user might desire. The results can be seen in image 5.2, where the areas are a lot less distributed than the previous results.

User 2 is similar to user 1, but does not mind being further away from his point of interest as user 1. The houses returned in the first recommendation are similar in terms of characteristics to the ones in the previous test with user 1. The main difference is in the general location. Figure 5.3 shows the distribution of the recommendations. It can be seen that, even though the range was upped, the areas are closer to the point of interest than in the previous test (as seen in 5.1). This happens because, given the wider range, a lot more houses that are within the user

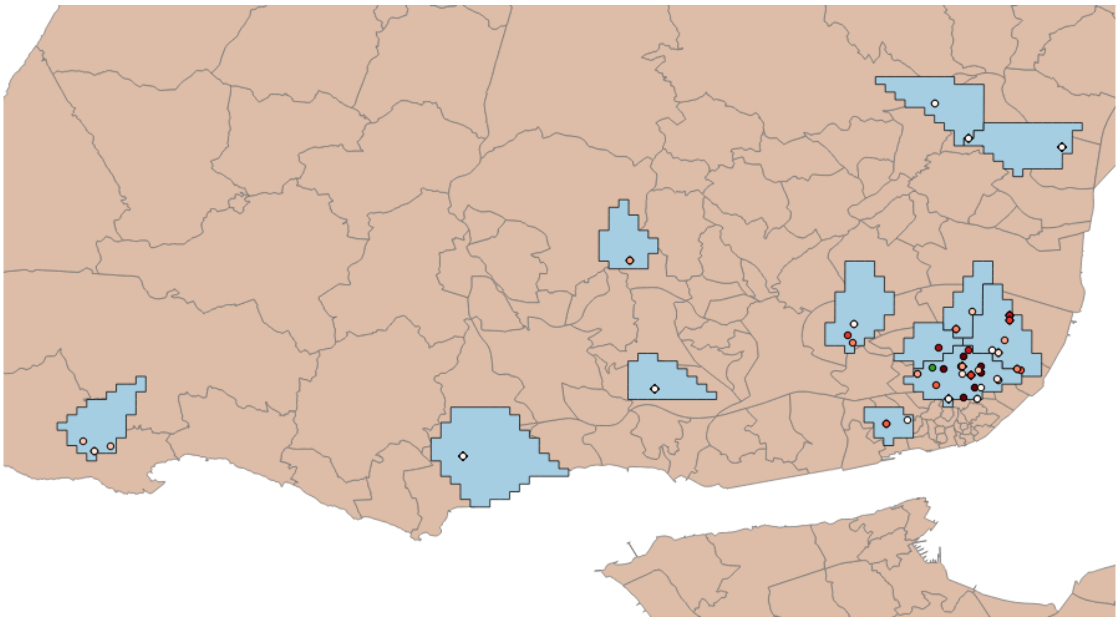


Figure 5.1: Area results from only input test for user 1

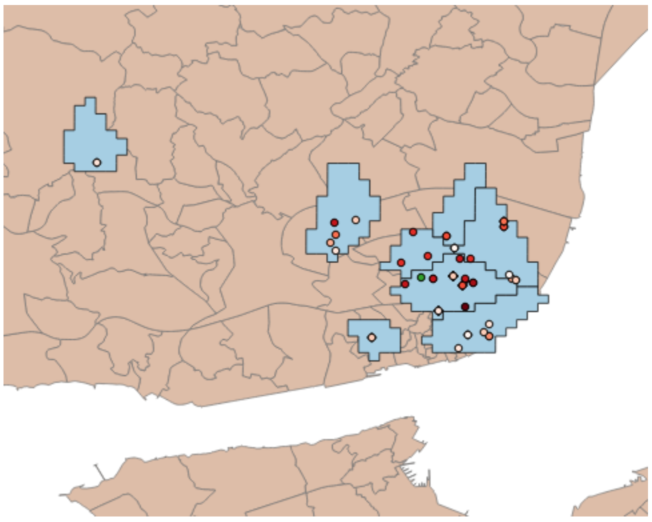


Figure 5.2: Area results from interaction recommendation test for user 1

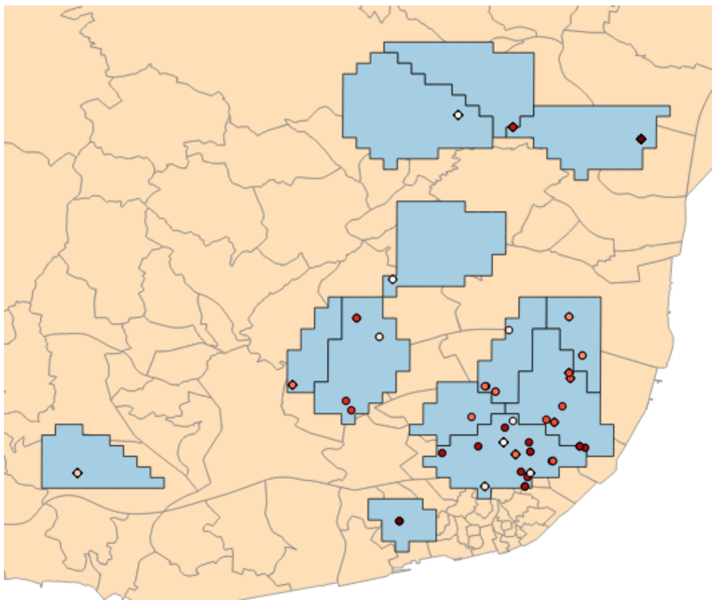


Figure 5.3: Area results from only input test for user 2

preferences are not penalized due to being further away from the point of interest. In the previous text, only, for example, 20 houses could be within the range, the rest of the recommendations (of lower value) were all equally penalised for their distance to the point of interest and, as a consequence, could appear in any place on the map. Another consequence is that, in this test, all areas in general have houses with high recommendation value, contrary to the previous test, where the majority of high value houses were in the same area of the point of interest.

The results from the interaction recommendations were very good. Apart from presenting some houses (not apartments) initially, the rest of the recommendations are of great quality, presenting apartments of great value. It is interesting to note that, at this stage, most of the recommendations are of typology three, contrary to user 1, where it's recommendations were generally of typology two, even though the inserted preferences were the same except for the range to the point of interest. This happens because, in the first phase of recommendations, since user 2 had a wider range, houses outside central Lisbon are cheaper and, therefore, it was able to find a lot more typology 3 houses within the price range and, consequently, user 2 has more interactions with houses of this kind. Further recommendation requests returned recommendations of similar quality.

The preferences for user 3 are quite different. Since the user is looking for a house and not an apartment, the other preferences have to change too (e.g. houses are more expensive than apartments, so the price range should be increased). On this first map, the results presented on the map, which can be seen in figure 5.4,

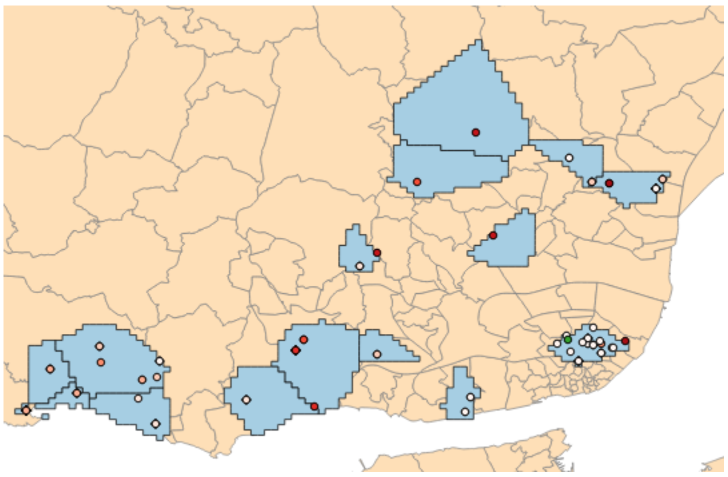


Figure 5.4: Area results from only input test for user 3

are very different compared to the previous users, but this was to be expected. The majority of the houses with good recommendation value are far away from the point of interest of the user. This happens because central Lisbon is mainly composed of apartments and most houses are in the outer regions. The first round of recommendations, taking only the user preferences into account, does not show any surprises compared to the previous tests. The following recommendations, using the user interactions, present a couple of apartments. This is not necessarily a bad behaviour, users looking for houses might find an apartment more interesting for some reason, be it price or quality, so no steps were taken to avoid this behaviour. Further recommendations show even more apartments, but this is due to there being a considerably less amount of houses in the database compared to apartments.

User 4 is looking for new houses (as in, houses where nobody has lived before). Since these are more expensive, the price range was increased. The results from the first recommendations have nothing out of the ordinary, be it in the map or in the recommended houses. The recommendations using the user interactions presents problems, though. The houses recommended are old and of low quality. While a user looking for a new house might be happy with an old one but in good state, houses in bad state will not be satisfactory. This behaviour happens because, after the houses browsed in the first interaction, which are according to the user preferences, there are not many more similar to these ones. Also, all the houses browsed in the first interaction were to the user's liking so, while the user was able to "show" the engine his preferences, there was no chance to make the engine understand which house features the user dislikes. So, given these two factors, the engine isn't left with much information to compute the following recommendations. In this case, when presented with low quality houses, the user has to “show” dissatisfaction to them and request new recommendations. After this, the recommendations are of good quality. The houses presented are used, relatively recent, in good state and in good conditions.

Users 5, 6 and 7 did not present any surprising results compared to the previous users.

5.2 User Testing

The user testing, apart from validating the results from the recommendation engine, can obtain important user experience information. The results from these tests will be presented in a more general way, since most users made similar comments about the platform and its results. Also, it is important to note that, contrary to the last section, the interaction similarity is an important component of the house similarity matrix, since this time around there is a lot more interaction information provided by the users.

5 people within Ubiwhere and 5 people from outside the company will be the testers of the engine. Each will use the platform three times, each time taking up a different user profile (appendix F): one will be user 1, the other two will be randomly selected. With each user profile, they will test both the new and the old engine, comment what they like and dislike and in the end they will state which one they liked the most and why.

Starting with comments about the performance of the engine, the users tended to prefer the recommendations computed with the interaction information more than the recommendations done only with their inserted preferences. More specifically, they appreciated that the engine didn’t restrict itself to the inserted preferences, presenting houses of good quality, even if not completely within the user desired characteristics. On a more negative note, some users complained that some of the houses presented have nothing to do with what the users asked for. These are houses of low recommendation value that happened to be within the 100 houses with the best score and, thus, presented to the user. This behaviours was easily dealt with by reducing the number of presented houses to 50.

Users usually started by browsing houses through the list on the left (Figures 1.3 and 1.4), although at some point they start to use the map to browse a given area to their liking. They wished for important information to be displayed in the snapshot of the houses, such as the number of bathrooms, presence of a garage and total area. As it is now, the only information displayed is the typology and price, making the user visit a lot of house pages just to see if the house is remotely to their liking or not. Another complaint was that, after visiting a house’s page, when returning to the browsing area, the house list returns to the top instead of going back to the same spot.

Comparing their experience with the new engine and the old one, the users generally preferred the results from the new. While they did not make many comments comparing the area results of the engines, they did noticed, using the new engine, that the results presented in the sidebar were ordered by their quality and appreciated that. And, as been said before, they also appreciated the recommendations made using the users' interactions.

5.3 Test Conclusions

From these tests, it can be concluded that the results obtained from the engine developed in this project are an improvement compared to the old engine. It returns areas the users can relate more to, sorts houses by their recommendation quality, returns faster results and an engine that can adapt to the users by using their feedback.

Chapter 6

Conclusion and Future Work

In this internship, a recommendation system was developed for Livin'X, a real-estate market platform. It can use a large amount of different information available, from city area information, such as services and events, to user information, such as their preferences or their interactions with the platform, in order to make quality recommendations to the end user. This was achieved through a study of state of the art methods and technologies used to process each different kind of information and the design of an architecture that could combine the best methods to obtain good recommendations using all the different available data.

The major setback of the project was the lack of real users using the platform, since it was not open to the public. This did not allow to fully test the quality of the results. Still, using volunteers from inside and outside Ubiwhere, it was possible to test the engine, verifying if the algorithms were well implemented and the results' quality to outside users. Another positive note is that the engine was developed in such a way that it can be easily deployed and used for other cities apart from Lisbon.

This work's contributions were the following:

- Study of the state of the art of various fields affecting this project (Machine Learning, Recommendation Systems, City Area Clustering, Community Detection);
- Designing of an architecture able to use all the available data to create quality recommendations efficiently;
- Development of a real-estate recommendation engine that is an improvement to the old engine, being able to make full use of city, house and user information to provide quality recommendations;
- Testing of the quality of the results obtained from the engine.

The tests made for this project showed that the algorithms were all implemented and that the engine was able to obtain good results. Still, only when real users use the platform can it be possible to say for certain that the recommendation engine is useful for them. This can be measured in several ways: if they find what they were looking for; if they return to the platform; if they don't give up on the platform midway through; if they recommend Livin'X to their friends. This is a typical sales funnel (Acquisition, Activation, Retention, Revenue, Referral) and it will be each of these steps that will be measured in a production environment.

The community detection algorithm, while not a priority, has the possibility of providing valuable information for the recommendations. Its performance is a bit dependable in the user location information that it is possible to obtain, but can be a good addition either way.

Technologies like Apache Spark and Storm will be vital to the performance and scalability of the engine. At this point, the computation that presents a more worrying situation is the recommendation calculations that use the interaction information. It's current complexity is $\mathcal{O}(i^n)$, where i is the number of houses the user has interactions with and n the maximum number of houses the user has no interactions with that are within a limit of similarity with the houses the user has interactions with (as was defined in 3.5). Since, in general, n will always be larger than i, as i rises, the complexity rises as well. While this complexity in general will not go out of control since the limit of similarity is controllable, these are still heavy calculations that would work better if they were made concurrently. This would be achievable with Storm.

While acceptable values for the multiple variables that affect the recommendations were found, there is room for improvement. Testing techniques such as A/B testing or multivariate testing should be applied to find the values that generate the best recommendations according to the users feedback. Also, since this system should be able to be applied to any other city in the world, it should also be tested in other cities in order to evaluate its adaptability.

While there is still work to be done and challenges to overcome, the results have shown that this new version of the recommendation engine is a great improvement over the old one in terms of result quality and performance. It is more prepared, equipped and dynamic to tackle real world problems.

Bibliography

[1] Agenda cultural de lisboa. http://www.agendalx.pt.

[2] Apache spark. https://spark.apache.org/.

[3] Apache storm. https://storm.apache.org/.

[4] Google maps api. https://developers.google.com/maps.

[5] Hadoop. https://hadoop.apache.org.

[6] import.io. http://www.import.io.

[7] Mahout. http://mahout.apache.org.

[8] Mongodb. http://www.mongodb.org.

[9] Netflix prize. http://www.netflixprize.com/.

[10] Orientdb. http://www.orientechnologies.com/.

[11] Orientdb vs mongodb. http://www.orientechnologies.com/orientdb-vs-mongodb/.

[12] Pordata. http://www.pordata.pt.

[13] Why orientdb? http://www.orientechnologies.com/why-orientdb/.

[14] Why netflix never implemented the algorithm that won the netflix 1 million challenge. https://www.techdirt.com/blog/innovation/articles/20120409/03412518422/why-netflix-never-implemented-algorithm-that-won-netflix-1-million-challenge.shtml/, 2012.

[15] Taiwo Ayodele. *New Advances in Machine Learning*, chapter Introduction to Machine Learning. InTech, 2010.

[16] Taiwo Ayodele. *New Advances in Machine Learning*, chapter Types of Machine Learning Algorithms. InTech, 2010.

[17] Robin Burke. Hybrid web recommender systems. In *The adaptive web*, pages 377–408. Springer, 2007.

[18] Zechun Cao, Sujing Wang, Germain Forestier, Anne Puissant, and Christoph F Eick. Analyzing the composition of cities using spatial clustering. In *Proceedings of the 2nd ACM SIGKDD International Workshop on Urban Computing*, page 14. ACM, 2013.

[19] Li Deng and Dong Yu. Deep learning: Methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2013.

[20] Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[21] Derrick Harris. Databricks demolishes big data benchmark to prove spark is fast on disk, too. `https://gigaom.com/2014/10/10/databricks-demolishes-big-data-benchmark-to-prove-spark-is-fast-on-disk-too/`, 2014.

[22] Joseph A Konstan and John Riedl. Recommender systems: from algorithms to user experience. *User Modeling and User-Adapted Interaction*, 22(1-2):101–123, 2012.

[23] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining Massive Datasets*, pages 24–25. 2014.

[24] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining Massive Datasets*, pages 92–94. 2014.

[25] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining Massive Datasets*, pages 405–425. 2014.

[26] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80, 2003.

[27] Stuart E Middleton, David C De Roure, and Nigel R Shadbolt. Capturing knowledge of user preferences: ontologies in recommender systems. In *Proceedings of the 1st international conference on Knowledge capture*, pages 100–107. ACM, 2001.

[28] Stuart E Middleton, Nigel R Shadbolt, and David C De Roure. Capturing interest through inference and visualization: Ontological user profiling in recommender systems. In *Proceedings of the 2nd international conference on Knowledge capture*, pages 62–69. ACM, 2003.

[29] Bamshad Mobasher, Xin Jin, and Yanzan Zhou. Semantically enhanced collaborative filtering on the web. In *Web Mining: From Web to Semantic Web*, pages 57–76. Springer, 2004.

[30] Anastasios Noulas, Salvatore Scellato, Cecilia Mascolo, and Massimiliano Pontil. Exploiting semantic annotations for clustering geographic areas and users in location-based social networks. *The Social Mobile Web*, 11, 2011.

[31] Rafael Pereira, Hélio Lopes, Karin Breitman, Vicente Mundim, and Wandenberg Peixoto. Cloud based real-time collaborative filtering for item-–item recommendations. *Computers in Industry*, 65:279–290, 2014.

[32] Stuart J. Russel and Peter Norving. *Artificial Intelligence: A Modern Approach*, pages 122–125. Pearson Education, 3 edition, 2010.

[33] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001.

[34] Ahu Sieg, Bamshad Mobasher, and Robin Burke. Ontology-based collaborative recommendation. *Computing*, 2010.

[35] Guillermo González Suárez, Tatiana Delgado Fernández, José Luis Capote Fernández, and Rafael Cruz Iglesias. Context–aware recommender system based on ontologies.

[36] Eliot van Buskirk. How the netflix prize was won. `http://www.wired.com/2009/09/how-the-netflix-prize-was-won/`, 2009.

[37] Jaewon Yang and Jure Leskovec. Overlapping community detection at scale: a nonnegative matrix factorization approach. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 587–596. ACM, 2013.

[38] Jaewon Yang, Julian McAuley, and Jure Leskovec. Community detection in networks with node attributes. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 1151–1156. IEEE, 2013.

[39] Xiaofang Yuan, Ji-Hyun Lee, Sun-Joong Kim, and Yoon-Hyun Kim. Toward a user-oriented recommendation system for real estate websites. *Information Systems*, 38(2):231–243, 2013.

[40] Reza Bosagh Zadeh and Gunnar Carlsson. Dimension independent matrix square using mapreduce. *arXiv preprint arXiv:1304.1467*, 2013.

Appendices

Appendix A

Methodology

This project was divided in two major parts: the designing of the architecture (with the necessary study of the state of the art) and the development of the engine. Being of a very different nature, different methodologies were applied to each part.

For the study of the state of the art and the designing of the architecture, it was straightforward: it was known what was desired from the platform, so the various fields involved were studied and at the end the architecture was made. In appendix C it can be seen the time allocation for this period of the work.

For the development of the system, Feature Driven Development was used. This lightweight agile methodology intends to iteratively release working versions of the software. At the end of each milestone, a senior coder will review the code and integration testing will be done at the end. The features were defined initially and can be seen in appendix B and the time allocation for each one of these features can be seen in D.

Appendix B

Feature List

- **Feature 1** - Information Gathering:
  - Contribute to the integration of spatial dimensions: City limits and parishes;
  - Contribute to inputting city neighbourhoods in the platform;
  - Extract parish security information.
- **Feature 2** - Genetic Algorithm:
  - Implement cell profiling mechanism;
  - Implement genetic algorithm;
  - Implement city cluster storage mechanism.
- **Feature 3** - Community Detection (minor milestone):
  - BigCLAM implementation.
- **Feature 4** - Area Rankings (major milestone):
  - Implement correlation detection between city clusters and communities;
  - Implement cosine similarity;
  - Create city area ranking.
- **Feature 5** - User Interactions (minor milestone):
  - Implement user interaction information gathering methods;
  - Calculate interaction similarities.
- **Feature 6** - Global House Similarity (minor milestone):

  - Calculate characteristics house similarity;
  - Calculate interactions vs house similarity matrix.
- **Feature 7** - Only Input Recommendations:
  - Only input and no input flows in production website.
- **Feature 8** - No Input Recommendations:
  - Integrate complete input flow in production website.

Appendix C

First Semester Time Allocation

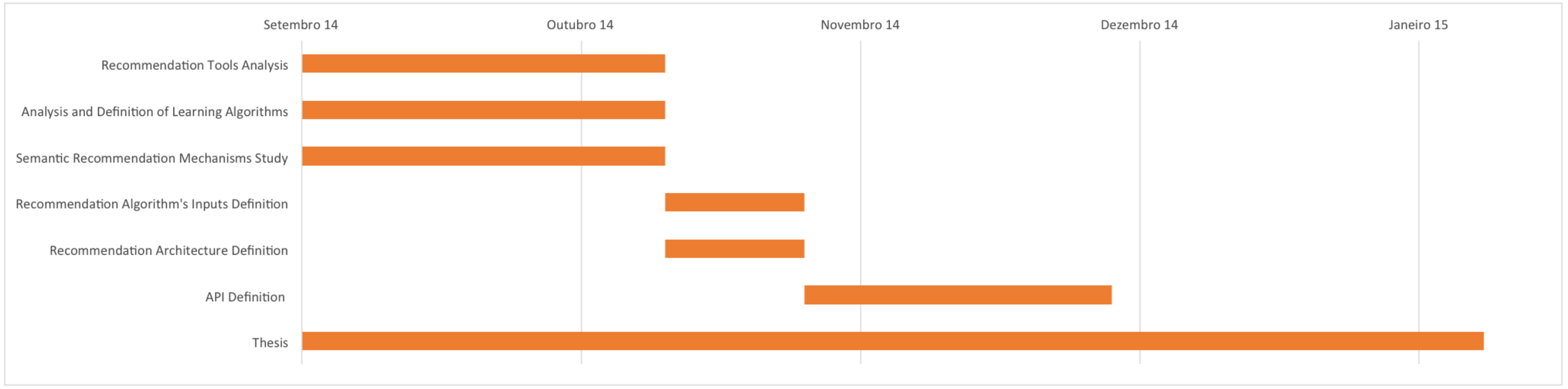


Figure C.1: Gantt chart representing the first semester time allocation

Appendix D

Second Semester Time Allocation

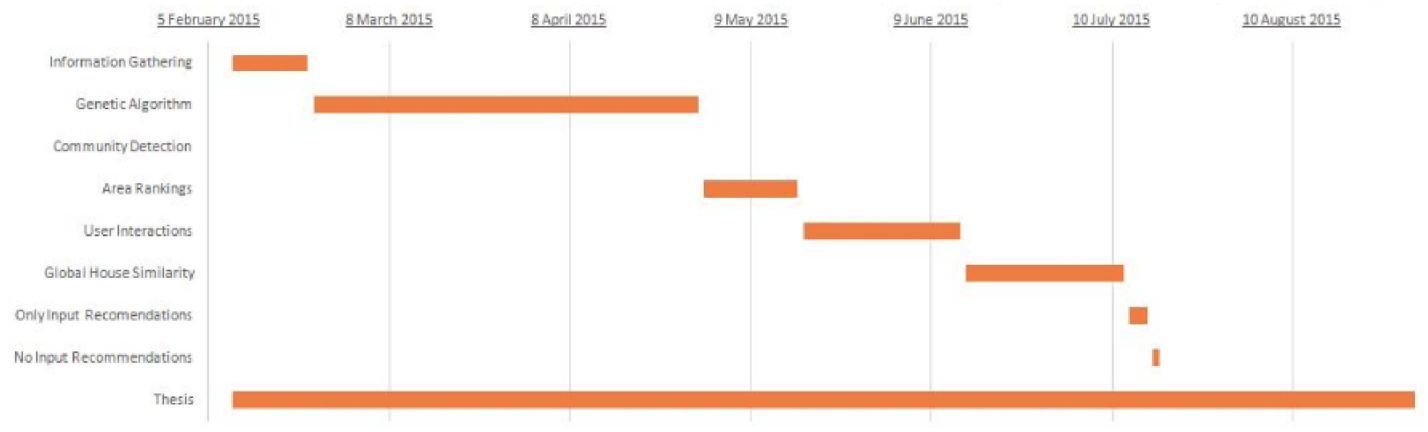


Figure D.1: Gantt chart representing the second semester time allocation

Appendix E

Security Levels

Table E.1: Municipality Security Levels

Municipality	Crime Rating
Amadora	4
Cascais	5
Lisboa	1
Loures	4
Mafra	5
Odivelas	5
Oeiras	5
Sintra	5
Vila Franca de Xira	5
Alcochete	5
Almada	4
Barreiro	4
Moita	4
Montijo	4
Palmela	3
Seixal	5
Sesimbra	3
Setúbal	4

Appendix F

User Profiles

Table F.1: User Profiles

User Profile	**Type**	**State**	**Buy/Rent**	**Price**	**Rooms**	**Mobility**	**Location**	**Time**	**Categories**
1	Appartment	Used	Buy	0 - 200k	2 - 3	Public	Saldanha	5	Restaurant, Cafe
2	Appartment	Used	Buy	0 - 200k	2 - 3	Public	Saldanha	15	Restaurant, Cafe
3	House	Used	Buy	200k - 400k	2 - 4	Car	Saldanha	15	Restaurant, Cafe
4	Appartment	New	Buy	0 - 350k	2 - 3	Public	Saldanha	5	Restaurant, Cafe
5	Appartment	Used	Rent	0 - 800	2 - 3	Public	Saldanha	5	Restaurant, Cafe
6	Appartment	Old	Buy	0 - 400k	4 - 5	Public	Saldanha	5	Restaurant, Cafe
7	Appartment	Old	Buy	0 - 200k	2 - 3	Public	Saldanha	5	Subway, School

User Profile - Profile Id; **Type** - Type of housing; **State** - Condition of the house; **Buy/Rent** - If looking to buy or to rent; **Price** - Desired price range; **Rooms** - Desired number of rooms; **Mobility** - User method of transportation; **Location** - Location user wants to be near to; **Time** - Maximum time user wants to be from given location; **Categories** - Types of services user wants to be near to.