DM

Preference Aggregation for a Group Recommendation System

MASTER DISSERTATION

Diogo Oliveira Gouveia
MASTER IN INFORMATICS ENGINEERING

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

September | 2023

Faculdade de Ciências Exatas e da Engenharia

Mestrado em Engenharia Informática

Diogo Oliveira Gouveia

Orientado por:

Eduardo Fermé

Jorge Fernandes

18 de dezembro de 2023

Resumo

Este relatório de projeto debruça-se sobre o desenvolvimento de um módulo de agregação de preferências para um sistema de recomendação de grupo integrado no projeto Knowledge-Based Artificial Intelligence (KBAI). O projeto KBAI tem como objetivo criar uma ferramenta prática para interação com uma base de conhecimento dinâmica e colaborativa, bem como para o processamento racional de conhecimento. O sistema será baseado numa base de conhecimento e tem como objetivo desenvolver uma plataforma que permita a gestão de quantidades consideráveis de conhecimento. As bases de conhecimento constituem uma forma de representação do conhecimento que, com o auxílio de *chatbots*, perfis dinâmicos e agregação de preferências, permite a construção de poderosos sistemas de recomendação. Para este efeito, será feita uma pesquisa teórica que combine o conhecimento académico e científico das instituições de investigação com a experiência empresarial e de desenvolvimento das equipas de profissionais que integram o projeto. Para além disso, será descrito o desenvolvimento do *software* destinado à agregação de preferências, desde a sua prototipagem até à versão mais recente à data da redação deste documento.

Keywords: base de conhecimento · escolha social · agregação de preferências

Abstract

This project report will delve into the development of a preference aggregation module for a group recommendation system integrated into the Knowledge-Based Artificial Intelligence (KBAI) project. The KBAI project aims to create an interactive tool for engaging with a dynamic and collaborative knowledge base, as well as for the rational processing of accumulated knowledge. This system will be based on a knowledge base and it aims to develop a platform that allows for the management of large amounts of knowledge. Knowledge bases constitute a way of representing knowledge which, with the aid of chatbots, dynamic profiling and preference aggregation, allows for the builing of powerful recommendation systems. In order to achieve this, theoretical research will be made so as to combine both academic and scientific knowledge from investigation institutions with the business and development experience from the professionals teams integrating the project. Furthermore, the development of the software destined for aggregating preferences will be described, from its prototyping to the latest version as of the writing of this document.

Keywords: knowledge base · social choice · preference aggregation

Agradecimentos

Agradeço profundamente ao meu orientador, o professor Eduardo Fermé, por dispôr a sua experiência e sabedoria, rever os conteúdos deste documento e oferecer preciosos pareceres ao longo do projeto.

Agradeço à minha família, sem cujo apoio o desenvolvimento deste projeto não teria sido possível.

Agradeço aos meus dois colegas, Katherin Varela e Rodrigo Vieira, pela colaboração e ajuda neste projeto conjunto, assim como todos os membros do projeto KBAI, incluindo o meu coorientador e líder do projeto, Jorge Fernandes.

Agradeço ao meu bom amigo, Pedro Rodrigues, pela confiança e apoio que me deu ao longo da realização do projeto.

Agradeço aos docentes da Faculdade de Ciência Exatas e Engenharia, pelas lições que moldaram o meu saber com as ferramentas necessárias para o meu futuro.

Este projeto recebeu financiamento ao abrigo da candidatura n.º. M1420-01-0247-FEDER-000073 no âmbito do Programa Operacional da Região Autónoma da Madeira (Madeira 14-20).

Table of Contents

List of Figures

Acronym List

AI Artificial Intelligence

API Application Programming Interface

HTTP Hypertext Transfer Protocol

ID Identification

IDE Integrated Development Environment

JPEG Joint Photographic Experts Group

JSON JavaScript Object Notation

KB Knowledge Base

KBAI Knowledge Base Artificial Intelligence

PDF Portable Document Format

PNG Portable Network Graphics

SVG Scalable Vector Graphics

SWF Social Welfare Functions

UML Unified Modeling Language

1 Introduction

This report is integrated into the Knowledge-Based Artificial Intelligence (KBAI) project, which aims to create an interactive tool for engaging with a dynamic and collaborative knowledge base, as well as for the rational processing of accumulated knowledge. Supported by a central platform, KBAI facilitates the registration and correlation of various concepts. The objective of this dissertation is to develop an engine that is able aggregate a group of preferences belonging to distinct profiles saved to the knowledge base as a group in order to allow a different system to provide recommendations to said group. In order to achieve this, multiple investigations and experimental developments will be made so as to combine both academic and scientific knowledge from investigation institutions with the business and development experience from the professionals teams integrating the project. In the system's kernel, various mechanisms should provide a constant update of the knowledge base's beliefs (by verifying their validity or adding new ones to the mix through learning or similar methods). The project will thus focus on the wide field of AI, with special focus on social choice and belief change, with the research mainly concerning the modeling of decision problems, analyzing existing models and attempting to provide an overview of the basic notions behind the representation of choice, preference and belief. Though the theories have been the subject of much study through the last years, the objective of this research in particular is to help provide a basic understanding of existing concepts to pinpoint which ideas may effectively aid the KBAI project's development.

A mentioned before, this system will be based on a knowledge base and it aims to develop a platform that allows for the management of large amounts of knowledge. Knowledge bases constitute a way of representing knowledge which, by using chat-bots and dynamic profiles, make it possible to create algorithms that allow for the aggregation of multiple preference sets in an effort to achieve a cohesive recommendation to a group of users based on each of their criteria. This recommendation system will be supported by an interface developed by the team and will cover three case studies: project management, news recommendation and touristic guiding. The project is split in several phases the first of which is a research of the current state of the art surrounding multiple topics touching on decision theory and related fields. Specifically, the study made in this project aims to aid the construction of an algorithm that allows for aggregating preferences to a group of users while allowing to update each of the user's profiles according to feedback provided.

The project will be developed by multiple teams, each working on different components of the system, mainly in collaboration with Katherin Varela and Rodrigo Vieira's work (see Appendix D).

In short, the aggregation will take a group's profiles created by the dynamic profiling and chat-bots, aggregate them into a single profile and save it to the knowledge base for the recommendation module to generate suggestions/recommendations for.

This dissertation will concern with a specific part of the developer team that aims to implement a system that aggregates the preferences of a group of users in an effort to provide a suggestion suitable to all the individuals' tastes, but future work will see extensive cooperation with the front-end team in order to integrate the resulting work in a cohesive interface. To this end, a preliminary exploration was made on the fields of decision theory, preference relations, social choice/judgment aggregation and belief change.

In the end, this project aims to obtain a running executable that receives requests through HTTP protocols, retrieves information from the knowledge base, aggregates a group's preferences and returns the result, ready to be used by the recommendation module of the system. This executable should be run in container environment, and be reliable enough to be scalable, asserted by future tests.

2 Planning

This project was planned to span from the September of 2022 to June of 2023. In order to better illustrate the project's timeline, a simple Gantt diagram was built (see Figure 1). This diagram constituted a vague estimate of what would be done, and as such some phases ended up stretching further along than expected and testing was postponed altogether.

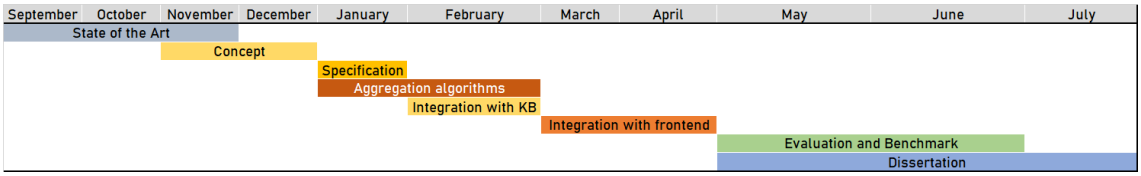

Fig. 1. Gantt diagram depicting the project's timeline.

The initial phase of this work has centered around research of the topics mentioned before, starting from three main sources: [1], [3] and [4]. From these, works that were cited by these works (or cited them in turn) were searched according to perceived relevance to the topics required to build a recommendation system that accounts for feedback in order to update a user's preferences.

The second phase had the main concepts of the system be elicited in an effort to understand the architecture and the entities at play with the knowledge base and obtain the generalized view of what the system will do once implemented.

The next phase, specification, cemented the concepts obtained before in data structures which will hold the information needed to provide acceptable recommendations to groups of users. This includes profiles, preferences, queries and any such data that allows the system to be more robust in its recommendations. This phase also saw a particular algorithm be selected for implementation after discrimination.

Afterwards, the selected algorithm was implemented, with the next phase concerning the integration with the knowledge base as it relates to the wider KBAI project. After integration, the remaining crucial step is the interaction with the user, working closely with the frontend team in order to build the interface to allow use of the algorithm.

The last phases concerned testing, benchmarking and the final documentation of the project towards the delivery date.

3 Research

3.1 Introduction

For the purposes of this dissertation, a collection of books and articles were consulted in an effort to introduce basic notions needed for a recommendation system destined to groups of user.

Decision (or choice) theory is a term that includes an incredible variety of fields (mathematics, philosophy, statistics, economics and many more) [1] and their respective theoretical work. On a more fundamental level, decision theory aims to build frameworks with which to try and understand the intricacies behind the choices that lead us to act. Decision theory, however, is far from being an isolated field. Indeed, it is more of an umbrella term than includes multiple study fields that each touch on similar but distinct notions that relate how we think and act.

Studying decision theory will serve as a basis of introduction to the entirety of the field, allowing a gateway into the remaining subjects. As far as this study goes, only the more essential basics around modeling decisions will be taken into account and this document thus will describe the work already done in the project and the remaining work that will follow until its conclusion.

The chapter is structured as such: after the introductory paragraphs, revision of research made until this point will be detailed, divided in four topics, with a small conclusion detailing future work.

This research went through multiple authors' works in order to grasp a solid understanding of the underlying complexities surrounding these fields, starting by Decision Theory proper with [1], where the basic concepts surrounding choice and preference are defined and exposed. From here, [5], [6], [7], [8], [9] and [10] provide us with an overview of the nuances of representing preferences and the relations between them. With multiple agents in play, social choice and judgment aggregation become a pressing concern, as discussed in the works [11], [12], [13], [2] and [14]. Finally, the field concerning the epistemic properties of belief and how it changes and evolves is touched upon by the works [4], [3] and [15].

3.2 Decision Theory

Decision theory joins the fields of economy, mathematics, philosophy, sociology and statistics in an effort to systematize the decision making of rational agents. Through this section, agent will be used to refer to any given entity that makes a decision. The theory's application encompasses contexts both abstract (speculation about what is a rational decision) and practical (analyzing human behaviour). Within it, we may find a multitude of research fields like the exploration of the variety of rules for decision making (and their logical consequences) or of the mathematical properties behind representing rational behaviour.

In [1], Resnik creates a useful overview on the field of Decision Theory. For the purposes of this study, the first four chapters were put at the forefront.

One very fascinating use of these theories is found at the hands of social scientists, where experiments and social surveys aim to try to understand how real people act in events that require a choice. Decision theory studies may be split into two general branches:

– Prescriptive, where it is attempted to systematize decision making in a way that reveals what is the most rational decision;

– Descriptive, where it is investigated how we, as people, act and make decisions by studying real life situations.

This is not a hard ruled distinction as information and experience from the descriptive branch is what allows prescriptions to be made about how rational decisions should be made in the other branch, and thus any given research is almost guaranteed to have both sides involved. One of the best ways of understanding a problem is modeling it. When it comes to modeling decision problems, this study will explore one that discriminates each choice in **act**, **state** and **outcome**. Let us look at a practical example: say that an agent walks by a random dog in the street. As they go to pet it, they stop and remember that the dog may not be friendly at all, and that they might get bitten. This situation can be represented graphically in a decision table, as seen in figure 2.

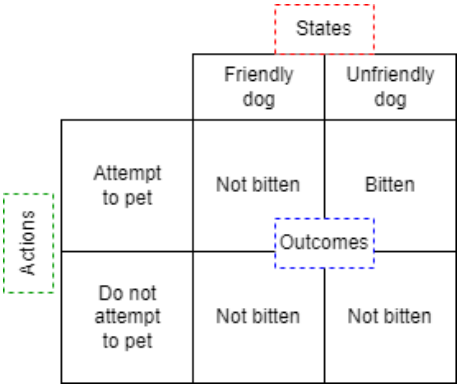

Fig. 2. Decision table with acts, states and outcomes of a particular situation. Based on [1].

This type of modeling is very useful to understand the kernel of a decision problem. In this case, this only one out of four outcomes that is deemed unfavourable. This process of identifying the acts, states and outcomes (and of drawing the associated decision table) is called the problem specification. Properly specifying a decision problem is key to understanding and analyzing possible courses of action. How one depicts and interprets the problem in the specification may impact what acts seem the most rational, as it may change which act is the dominant one by outlining its particular advantages due to modeling choices. An act is dominant over another if its outcomes are more favourable overall (in Figure 2, not attempting to pet the dog is the dominant act, as none of the outcomes are negative).

It is important to note that there might exist a case of illegitimate problem specification. For example, describing states in the vein of "win", "success" or "good outcome" are all roughly equivalent to "making the right decision": these type of specifications prove redundant, because if the agent already knows the right choice there is no reason not to choose it. While it may seem straightforward to identify an illegitimate specification, it is not as easy to conclude how proper a state description truly is. For example, states may have different levels of relevancy to the agent in a given decision (whether or not it is a full moon or not should not affect a decision on what meal to have, at least in theory) or states that could be switched to "making the right decision" are not immediately ruled out (whether or not the agent will have success in a given career is easily seen as "make the right decision", but should still be a factor in deciding the outcomes of a career choice). Another way of representing this problem is the decision tree, illustrated in Figure 3.

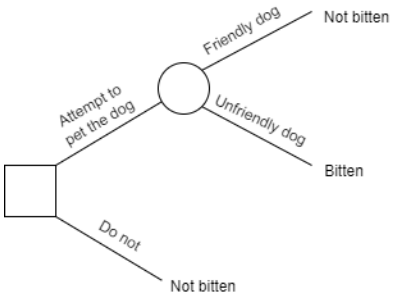

Fig. 3. Decision tree with acts, states and outcomes of a particular situation. Based on [1].

This way, it is remarkably easy to expand on the resulting states, resulting in a chain of choices, as seen in Figure 4. Such modeling also allows for a more complete representation of a problem that requires multiple decisions, leading to multiple states, outcomes or other subsequent decisions.

Building a tree from a given decision table is also a simple task, each square is a point of action from which depart as many lines as there are possible actions, each of those leading to a circle that itself possesses as many lines departing from it as there are possible states that lead to the respective outcome.

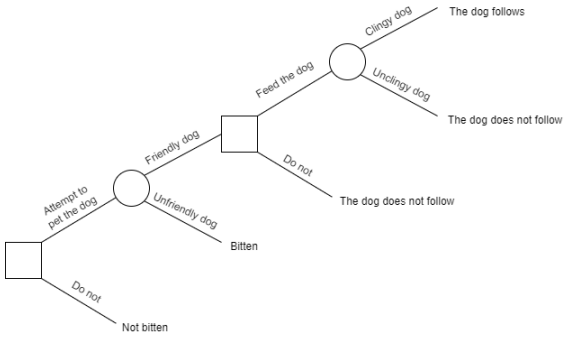

Fig. 4. Expanded decision tree with acts, states and outcomes of a particular situation. Based on [1].

It should be noted that the tree represented in Figure 3 was obtained from the decision table in Figure 2, although with the "Do not attempt to pet" act leading only to "Not bitten" as a result of the act being irrelevant regarding the outcome in that state, simplifying the tree. It is however, not such a simple ordeal to convert a tree into a table. A way to do this is with strategies, in which an agent decides a sequence of decisions based on the successive outcomes: for example, in the situation described in figure 4, a possible strategy would be "Attempt to pet the dog, if it does not bite, attempt to feed it" and another could also just be "Do not attempt to pet", both leading to different ends of the tree. There is no sure way to convert a tree into a table, but condensing them into strategies is an interesting way of analyzing a decision problem.

With the modeling of the decision problems covered, there is also the matter how an agent chooses a course of action. While choosing the dominant acts seems a reasonable choice, at least in theory, it is not always clear which act is the dominant one, as sometimes it comes down to what the agent prefers (for example, an animal lover would likely prefer to interact with the dog,

despite not interacting with it being the dominant act). This leads to the question that will be explored next: what is preference? The next section will delve into the topic of preferences and preference relations, resorting to commonly cited authors to explore common ways of representing and modeling preference.

3.3 Preferences and Preference Relations

A fundamental aspect to making a decision is the discrimination between actions depending on how preferable their outcome is. This is a common occurrence through daily life: what clothes to wear, meals to eat or shows to watch are all examples decisions made based on the preference of the individual. One of the most challenging aspects in formalizing these relations lays on trying to understand what it means for an agent to prefer one thing to another.

Preference relations can thus be described as being binary relations which can be used to express an agent's preferences in decision problems. Despite the concept they represent being so inherent to decision making and wider idea of rationality, there is no formal standard for the representation of preference relations, though many of the conceived models share a fair deal of similarities. In general, all of the existing models allow to describe the basic notion that given two alternatives, one of them is either less, equally or more preferable than the other.

The relations between preferences are normally represented in any such way which allows for the basic notion that given two options A and B, A is at least as preferable as B. Such two models include:

- **Cardinal preferences**, in which each agent is able to evaluate utility for each option and thus ordering them by said utility.
- **Ordinal preferences**, in which each agent has a preference between each option, thus also ordering them but without giving them any proper value beyond what the comparison affords them.

Other preference models include dichotomy, semi-ordering, intervals, diffusion and many more exist, but cardinal and ordinal tend to be more reliably explored in the literature.

In [9], Hansson displays the fundamental axioms for preference relations, represented by the binary operator P such that xPy means that an option x that is preferable to an option y, both of them belonging to a preference field. These options, x and y, are seen as logical propositions and as such may have logical operators from propositional calculus applied to them, such that $(x \wedge y)Px$ would mean that having both x and y happen is preferable to only x happening. As is noted in Hansson's work, this does not mean that all rules surrounding logic operators apply to preferences, regardless if their meaning is the same. In his work, rather than P, the relation R (meaning "at least as equally preferred", rather than the strict preference represented by P) is used as the most essential in order to formalize the axioms. As a binary relation, R obeys the following axioms:

1. Transitivity: $(xRy \wedge yRz \implies xRz)$;
2. Totality: $(xRy \vee yRx)$.

It is thus defined that a preference relation is one that fulfills both axioms, while a partial preference relation fulfills only the first one and the property xRx. By means of R we can define the relations P and R:

1. P is such that $xPy \equiv xRy \wedge \neg(yRx)$, meaning that x is strictly preferred to y;
2. S is such that $xSy \equiv xRy \wedge yRx$, meaning that x is equally preferred to y.

It follows also from the axioms of transitivity and totality that, as shown by Sören Hallden in [6]:

1. $xPy \implies \neg(yPx)$
2. $xPy \wedge yPz \implies xPz$
3. xSx
4. $xSy \implies ySx$
5. $xPy \wedge ySz \implies xPz$
6. $xPy \vee xSy \vee yPx$

Hansson notes additionally that an additional axiom, $xPy \implies \neg yP\neg x$, is applicable if the negation operator $\neg$ has any meaning in the preference field. His work goes further into more axioms which will not be delved into here, as the ones listed here serve as a solid base with which to represent preference relations in a clear way.

Sören Hallden, in [6], can be seen as the forerunner on the formalization on the logic of preference, with Von Wright's later work [16] presenting the first axiomatization. Since then many other contributions of value have been made (including Hansson's [9] explored before), but there the modeling of preferences remains unstandardised over the literature. The work in [8] aims to provide a survey on non-conventional preference modeling. Listing a variety of models, the authors go in detail about the different advantages and disadvantages arisen from each one, providing a comprehensive view of the dilemma around modeling preferences in an appropriate way, remarking that there is no single best way to model preference relations. The eventual choice of model often revolves on the nuances the problem at hand that are relevant to its proper representation: for example, proper numerical solutions tend to be simpler to work with, but are likely to rely on a large number of assumptions, while more nuanced and rich models give up on strong formalization in order to allow for flexibility. Any given model will have its downsides, but that does not discredit them, as their strengths are often reflected on such downsides.

Continuing with the exploration of preference relations, in [7] Xu explores various types of preference relations, summarizing their advantages and shortcomings. A comprehensive survey is exposed in the paper detailing a group of preference relations, of which two are put at the forefront: **multiplicative** and **fuzzy**. For either of these, Xu defines two sets with which to analyze them:

1. $X = \{x_1, x_2, ..., x_n\}$ of multiple alternatives;
2. $w = \{w_1, w_2, ..., w_n\}$ of preference weights (where the preference weight of x_n is represented by w_n) such that they all add up to 1.

These two sets are used in order to represent the alternatives and their preference weight through the exploration of the multiplicative and fuzzy relation.

Multiplicative preference relations are based on a reciprocal matrix $P = (p_{ij})_{n\times n} \subset X \times X$ where n is the number of alternatives and:

$p_{ij} > 0, p_{ij} \cdot p_{ji} = 1, p_{ii} = 1$ with $i, j = 1, 2, ..., n$

$$\begin{bmatrix} 1 & 0{,}5 & 0{,}2 \\ 2 & 1 & 0{,}4 \\ 5 & 2{,}5 & 1 \end{bmatrix}$$

Fig. 5. Matrix representing the multiplicative preference relations of an agent.

This means each column and row represent a specific alternative, with the corresponding value representing the ratio of preference of the row's alternative to the column's, such that if $p_{ij} > 1$, alternative i is p_{ij} preferred to j (this also means there can be no $p_{ij} = 0$, as then we would have $p_{ij} \cdot p_{ji} = 0$, breaking the condition. For example, assuming an agent determines his preferences on three distinct alternatives, the resulting matrix would have three columns and three rows, as seen in Figure 5. The value on the last row, first column indicates, for example, that the agent prefers the third alternative five times as much than the first, as $p_{31} = 5$ (and, as such, prefers the first alternative a fifth as much than the third, as $p_{13} = \frac{1}{5}$). It is worth noting the diagonal of ratios of 1 (this is, all p_{ij} where $i = j$), denoting that any alternative is equally preferred to itself. The matrix in the figure represents what is called a consistent multiplicative relation, because it possesses the transitivity property $p_{ij} = p_{ik} \cdot p_{kj}$, seen as $p_{13} = p_{12} \cdot p_{23} \Leftrightarrow 0.2 = 0.5 \cdot 0.4$, and thus every value is given by $p_{ij} = \frac{w_i}{w_j}$ (the ratio between the preference weights of each alternative). In practice, multiplicative relations are seldom transitive (and thus, are almost always inconsistent), a topic the author goes into further detail in his work while defining other ways transitivity can occur.

Fuzzy preference relations concern themselves with a more nuanced approach to the vague nature of preferences. Much like multiplicative, there fuzzy relations are represented in a matrix $R = (r_{ij})_{n \times n} \subset X \times X$, where n is also the number of alternatives, but rather than reflexive it is complementary and is such that:

$r_{ij} \geq 0, r_{ij} + r_{ji} = 1, r_{ii} = 0.5$ with $i, j = 1, 2, ..., n$

Like the multiplicative relations, each value in the matrix defines the preference relation regarding the row's alternative to the column's, but unlike multiplicative, fuzzy relations are, in practice, a percentage. While the value still reflects a ratio of preference (totally relative to the alternatives at hand), the degree of preference from alternative i to alternative j adds up to 1 if added with the degree of preference of j to i (meaning that if $r_{ij} \geq 0.5$, then i is preferred to j, for example). A fuzzy preference relation matrix is represented in Figure 6.

$$\begin{bmatrix} 0{,}5 & 0{,}5 & 0{,}2 \\ 0{,}5 & 0{,}5 & 0{,}4 \\ 0{,}8 & 0{,}6 & 0{,}5 \end{bmatrix}$$

Fig. 6. Matrix representing the fuzzy preference relations of an agent.

Because this is an additive relation, a_{ij} can be equal to zero, meaning j would be completely preferred to i (in other words, $r_{ij} = 0$, thus $r_{ji} = 1$ so that the $r_{ij} + r_{ji} = 1$ condition is respected). Consistency is also a property of these relations, which are called additive consistent if they are

such that, given a third k alternative, $r_{ij} = r_{ik} - r_{kj} + 0.5$, where each relation is given with $r_{ij} = 0.5(w_i - w_j + 1)$. Like multiplicative relations, a multiplicative consistent fuzzy relation exists if $r_{ij}r_{jk}r_{ki} = r_{ik}r_{kj}r_{ji}$ where each relation is given by $r_{ij} = \frac{w_i}{w_i + w_j}$. The paper lists further desirable properties which will not be explored here, but complement these notions.

Xu goes in further detail, exploring sub-types of multiplicative and fuzzy preference relations based on the criteria listed before, including the relationships between them, while also delving into two other types of preference relations: **linguistic** and **incomplete**.

Linguistic relations are focused on the idea that decision making involves a description of preferences through linguistic labels that work around ordered discrete sets on the style of $S = \{s_i | i = -t, \ldots, t\}$ where s_t is the linguistic variable attributed to the linguistic label t. This type of relation can be seen as a matrix $L = (l_{ij})_{n \times n} \subset X \times X$ such that:

$l_{ij} \in S, l_{ij} \bigoplus l_{ji} = s_0, l_{ii} = s_0$ with $i, j = 1, 2, \ldots, n$

Incomplete preference relations in turn are also explored by Xu in further detail in the paper.

The conclusions drawn by the article split the relations in roughly two types, the ones with complete knowledge of all elements (like multiplicative and its derivatives) and the ones without it (like fuzzy and its derivatives).

Concluding this section on preference relations, an overview of the nuances surrounding preference of an agent was explored, with the exploration of essential axioms as well as some models of representation of preference relations, where a key observation is made: no modeling of preference presents itself a silver bullet against any and all problems surrounding decision and preference and how one decides to model a problem comes down to the needs of that particular situation in regards to what aspects are seen to be the most critical.

Until now, single agent preferences have been explored, but people are rarely isolated and free of concern on what others around them prefer. Herein lies an interesting question: how to deal with multiple agents with multiple preferences among them? The next section will cover social choice and judgment aggregation in order to better understand the key issues behind the aggregation of multiple agents' related preferences on any given matter.

3.4 Social Choice and Judgment Aggregation

Let us look into John Donne's words: "No man is an island entire of itself; every man is a piece of the continent, a part of the main; [...]". Indeed, humankind is an inherently social species, for any kind of benefit modern society brings us is achieved through the collective effort of multiple individuals sharing their skillsets to create something bigger than the sum of its parts. Thus the community is formed.

As communities are made of different individuals, with varied world views, ideals and experiences, decisions that affect the whole group need to be taken considering each of the individual members' judgments. It is this phenomenon that the fields of social choice and judgment aggregation aim to study and systematize, with the goal of better understanding how multiple agents with different opinions work in collaboration to decide on the course of action that best fits the whole group.

Defined in [12] as the design and formal analysis of methods for aggregating the preferences of multiple agents, social choice theory constitutes a field originating in economics and political

science. Aggregation methods are used to allow groups of agents to choose among a set of alternatives. For example, choosing a leader in an election or deciding on how to allocate resources through a team are processes that involve aggregation methods. Computational social choice can thus be seen at the computational framework with which to study social choice. As defined in [5], computational social choice is an ever growing field that joins classical topics like economics and voting theory with more modern topics like artificial intelligence, multiagent systems, and computational complexity. Preference relations come into play in particular within the voting theory field, where economists, political theorists, mathematicians and other specialists have dedicated a lot of effort in studying the properties of voting. Within this field, its computational possibilities were first explored in the 1990s [12] by Bartholdi, Orlin, Tovey and Trick, who correlated the complexity theory of computer science to social choice and positing that complexity itself could be a barrier to manipulation.

Pragmatically, any social choice problem can be seen as an optimization problem. Optimization problems generally consist in finding a solution that obeys all constraints while satisfying the most preferences possible. In this context, we may look at constraints as "hard" preferences, which may not be disregarded by the solution, while other preferences can be seen as "soft" preferences which may be overlooked if no other choice is presented. These last ones may be represented by different models, such as fuzzy, weighted, probabilistic and others.

The question of how to aggregate preferences remains. This study will concern itself with a straightforward method, voting rules. While many diverse voting rules exist, all of them involve the same initial phase where all the voters denounce their preferences by ordering all the alternatives. From those ordered preferences, whatever rule in effect is applied the agents' preferences in order to obtain the winner. We may see these rules as functions called social welfare functions, which will be discussed later.

At the time of the French Revolution, Nicolas de Condorcet was a mathematician and philosopher advocating for the ideas of the then-ending Enlightenment. He studied social choice formulated two insights which will be touched on this report, Condorcet's Jury Theorem and Condorcet's Paradox. Applying mathematics to social sciences, he built two major observation on social choice, listed in [2]:

1. Condorcet's Jury Theorem: In essence, assuming all the juries have the same chance of passing a correct judgment (from two choices, one of which is seen as "correct"), and that chance is higher than 50%, then the theorem holds that the more judges there are the more likely it is for their aggregated judgments to be correct.

2. Condorcet's Paradox: This paradox relates to the phenomenon that an aggregation of rational agents' judgments may result in irrational judgments, such as when there is no Condorcet winner (i.e., a winner that is strictly preferred to every other candidate in vote). This is exemplified in Figure 7.

It is worth noting that Condorcet studied a particular form of voting: majority voting. In this, he discovered a surprising compound of related issues that are to this day a core concern of social choice theory.

Lastly, the marquis also is behind a fundamental notion of the Condorcet Winner. This is any such voted candidate who is overall preferred to any other candidate, meaning that the majority of voters ranked said candidate over all the alternatives.

Voter	Preference		
	#1	#2	#3
X	A	B	C
Y	C	A	B
Z	B	C	A

A is preferred to B
C is preferred to A
B is preferred to C

Fig. 7. Condorcet paradox. Based on [2].

In [12], credits social choice theory's foundations as a formal scientific field to Kenneth Arrow, responsible for bringing together the axiomatic method and the exploration of aggregation methods, while also being the author of the Impossibility Theorem (now also known as Arrow's Theorem). After his work, properties like Pareto-optimality, monotonicity and manipulability have been criteria used to study the possibility of aggregations methods.

Building on Condorcet's studies concerned with majority voting, Kenneth Arrow, a Nobel Prize winner, was the author of a more general outlook on the study of social choices.

In his studies of social choice theory, Arrow based his work on social welfare functions (SWF). These are aggregation methods that collect varied **ballots** from a multitude of **voters** with the aim of aggregating them according to a **rule of voting**. As put in [14], SWFs consist of a framework for preference aggregation, that is, a way to join the preferences of multiple agents in a single preference ordering that reflects all of their judgments, such that with a finite set N consisting of two or more agents, a finite universe U consisting of at least two options, the SWF uses the strict preference ordering made by each of the agents in N on the options in U (this is, their **ballots**) in order to obtain an aggregation.

These functions are typically judged according to properties that would, theoretically, consist of a just voting system. In this context, Arrow formulated a paradox using some particular properties (as defined in [12] and [2]):

1. Pareto-optimality: The result of the function should be congruent with all the agents' preferences, such that if any given voting alternative is preferred to another in the resulting aggregation, then all the individual agents also possess that very same preference relation. Formally, this means if a strict and unanimous choice is reflected in the SWF's result (such that if a is preferred to b by the end result, then all voters prefer a to b), then is it pareto-optimal;

2. Independence of irrelevant alternatives: This property requires that the social preference between any pair of alternatives is influenced only by that pair alone, such that the insertion of new alternatives does not alter it. This means if a is preferred to b, no matter the other alternatives that may be added or removed, then the property is obeyed;

3. Non-dictatorship: An SWF is non-dictatorial, if there is no voter such that if a is preferred to b by that voter, then the result necessarily reflects that a is preferred b. Basically, this means no voter decides the election by themselves.

Arrow's theorem is thus stated:

For any aggregation problem, if there are more than two voters, then there exists no preference aggregation rule that satisfies pareto-optimality, independence of irrelevant alternatives and non-dictatorship simultaneously.

This particular theorem assumes the domain to be universal, meaning all logically possible profiles of complete and transitive individual preference orderings are valid ballots. What does this theorem mean, in practice? Put simply, this means that a Pareto-efficient rule that respects independence of irrelevant alternatives will have a dictator, thus being liable to manipulation as one voter's preferences dictate the overall result.

Though there is much to discuss on the topic of preference aggregation, the essential concepts have been laid in this section, allowing for a light understanding of the nuances that voting systems often hide. From the smallest group's decisions to a nation's elections, the fundamental conclusions that social choice theory has reached represent an attempt to understand how to create aggregation methods which all involved agents believe to be fair and representative. The KBAI project will focus on smaller groups, but even then the prospect of creating recommendations that satisfy all the individuals in such groups seems a complex task that requires further study of various voting methods. The variety present through the literature means it would be unfeasible to merely list them here, but throughout the projects timeline a good amount will be studied in order to choose the method most appropriate to the recommendation system. A detailed list of voting methods explored will be presented in a following section.

No matter how satisfactory a voting rule, often consistent voting is needed as the human mind is in constant change in congruence with all it perceives through life, thus adding another layer of complexity to the topics already discussed, decision theory and preference relation. With social choice and preference aggregation covered, this research then follows into the last topic: belief change.

3.5 Belief Change

In a constantly changing environment, what an agent knows or does not know is in permanent change. As new information becomes available to the decision maker, their perception of what they hold to be true, their beliefs, adapt to be the most consistent as possible.

As professed by [4], this notion has been explored in a general sense since the times of Antiquity, but only in the past century has it become a formal field of study at the hands of the likes of William Harper, Isaac Levi, Georg Henrik von Wright and Jaakko Hintikka. These last two laid the first groundwork in the systematization of the logic behind beliefs, while Harper brought forward the fact that Carnap's inductive logic lacked the capacity of belief revision. Levi is credited with much of the formal framework of the field. Along with these names, Carlos Alchourrón, Peter Gärdenfors and David Makinson are also extremely important, as they provided a new formal framework that allowed for a better study on the field of Belief Change, known by their surnames' initials - the AGM model.

To discuss how to model belief change, we can look into the notions Gardenförs explored in his work [3], namely the concepts of epistemic states and the changes they go through.

The epistemic **state** represents the cognitive state of an agent at a given point in time. At any moment, the cognitive state might change through an epistemic **commitment**, obtained from an epistemic **input**. This will then result in an epistemic **equilibrium**, where the epistemic state is now consistent with itself after the commitment has provoked all the changes. These changes are such that, depending on the epistemic *attitude* might add, update them or discard beliefs through epistemic **expansions**, **revision** or **contraction**.

Epistemic states are specifically the idealizations of human cognitive states and are, as such, subject to criteria that are not often affirmed in practice. This means that if we see an epistemic state as a collection of beliefs, they might or not be consistent with each other, though, ideally, they certainly should if we wish to have what is perceived as a rational state. Any given state can be tested by being but under criticism of internal sources, under which it reaches epistemic equilibrium in case it satisfies all internal criticism. As internal sources tend to push a state towards equilibrium, external ones tend to push states towards *different* equilibria. To model these states, we may recur to the Bayesian model, propositional model or the model of possible worlds.

For any given state, any belief is judged in what Gardenförs summarizes into three epistemic attitudes in the propositional model: acceptance, in which the belief in part of the epistemic state, rejection, in which its negation is accepted by the state, or indetermination, in which neither it is accepted or rejected and thus not a part of the proposition set. In a probability model, the same three attitudes apply whilst attributing probabilities to beliefs such that $P(A) = 1$ means acceptance, $P(A) = 0$ means rejection and $0 < P(A) < 1$ means it is undetermined, with the numerical valuation allowing for a more nuanced approach. Any of these attitudes however share their dependency on epistemic **judgments**, which consist of natural language idioms that reflect the attitude ("A is likely" or "A is not likely" are both indetermination in a probability model, but indicate tendencies towards acceptance or rejection).

Any such attitudes on beliefs are the result of epistemic inputs, which catalyze all changes to a belief state in what are seen as external forces pushing a state from one equilibrium to another. This usually involves a change in attitude towards a certain belief, whose status in the state was put into question due to a conflicting input, throwing the state out of equilibrium. The change is begotten through a series of internal forces pushing to make the state consistent again. As such we may see inputs as new constraint thrown at the state: it is in equilibrium, until an input comes along and makes it inconsistent by invalidating some sort of belief (and/or its implications), requiring epistemic commitments that make it consistent again. These changes of belief are, in turn, guided by some functional rule by which the changes take hold. These can be called **epistemic commitment functions**. These functions take a given state, make all the changes needed to host the new input and return the new state of belief.

Gardenförs explores many models, but we will have belief sets at the forefront as it constitutes a well defined way of modeling an epistemic state. In Hansson's [15], the author does in further detail of this modeling with a relevant focus on computer science, where programmers have constructed databases with the ability to be updated through procedures. With the advancement of artificial intelligence, database updating has advanced in tandem.

The work's examples use a computer that behaves similarly to an agent, holding beliefs in the same way. Assuming the computer allows for communication with a database containing information on a given subject. The machine works in such a way that any question about its beliefs is answered. In a first example, the book uses binary question which are answered with "yes" or "no". This allows to know which beliefs are represented in the database (but not how!). For example, "Do you believe cats bark?" would be met with a "No", for example. This brings us to a first level of abstraction in studying belief change. To this end, the only the properties of a database are relevant, not the intricacies of its data's subject. Rather, the model of question "Do you believe A?" forms a pattern where A may represent any sentence. Hansson takes this sentence to obey to sentential logic.

With this in mind, let's consider sentences p,q and r which will be used to question the database:

"Do you believe p?" $\triangleright$ Yes

"Do you believe q?" $\triangleright$ No

In this study, we will have truth-functional connectives serve the purpose of negating and combining sentences, with which the computer should be able to answer questions:

"Do you believe $\neg r$?" $\triangleright$ No

In the vast majority of cases, consistency will be desired from the system, such that it does not simultaneously believe r and $\neg r$, for example. Such a system would logically inconsistent. This is, interestingly, not broken by the case in which neither r nor $\neg r$ are believed, which would mean the agent does not possess enough data to hold any of the beliefs.

This goes back to Gärdenfors formulated epistemic attitudes. For any given sentence p, the machine's attitude of belief might be of:

1. Acceptance (belief in p / acceptance of p, implying disbelief in $\neg p$ / denial of $\neg p$);
2. Rejection (disbelief in p / denial of p, implying belief in $\neg p$ / acceptance of $\neg p$);
3. Indetermination (suspension of disbelief on p, neither denying nor accepting p or $\neg p$).

This allows for different questions to be made ("Is it the case that p?") to which the system can answer with "I do not know" rather than just "yes" or "no". We may also add probabilities to this representation of belief in such a way that, for example, assuming the system believes p has 1% chance of being true, it answers on the lines of:

"How likely is p?" $\triangleright$ Five per cent

"Is it possible that p?" $\triangleright$ Yes, but very unlikely

These types of answers add complexity to the modeling but allow for a more realistic approach to belief representation. For now however, only the three main answers ("yes", "no" and "I do not know") will be used to study belief sets, but extended scopes of answers are further studied in Hansson's book.

With the basic notions of interaction with the hypothetical computer established, it is now time to study inputs and how they change the database's beliefs. For these inputs to be made, there will be two different instructions that allow the belief on a sentence p to be changed:

"Believe p!"

"Do not believe p!"

In a practical example, assuming the database holds no information on p at first, these commands would be used as thus:

"Do you believe that p?" $\triangleright$ No

"Do you believe that $\neg p$?" $\triangleright$ No

"Believe p!"

"Do you believe that p?" $\triangleright$ Yes

"Do you believe that $\neg p$?" $\triangleright$ No

"Do not believe p!"

"Do you believe that p?" $\triangleright$ No

"Do you believe that $\neg p$?" $\triangleright$ No

In this example, the first command leads to incorporation of belief, while the second leads to its contraction. It needs be reminded that asking the computer to not believe p is not equivalent to asking it to believe $\neg p$. Rather, it is the same of asking to "forget" the information held on p in case it accepts it, which is shown above where the computer does not suddenly believe that the opposite of p is true.

These types of inputs allow for the current state of belief in the database to change. At any given time, the collection of beliefs held by the database can be called its current belief state (also called epistemic state by Gärdenfors), such that:

1. $s(K)$ represents the belief set associated to the belief state K;
2. A sentence p belongs to $s(K)$ if asking "Do you believe that p?" is answered with "Yes" by a machine whose belief state is K.

This lack however one crucial element of rational beliefs: that the agent needs to believe the logic consequences of the beliefs held at any given state K. These consequences are represented by $Cn(K)$, which holds all that follows from the sentences in $s(K)$. This means that if p and q are believed in the state K and are part of $s(K)$, it follows that, by $Cn(K)$, $p \wedge q$ is also a part of $s(K)$ even if the agent was not told explicitly to believe so. Any sentence, α, that is the logical consequence of a sentence set A, $Cn(A)$, can have its relation with A represented as $\alpha \in Cn(A)$ or $A \vdash \alpha$. As a result, this means $s(K) = Cn(s(K))$, making the set closed under logical consequence.

Belief sets have however have glaring disadvantages, counter-intuitive results of the idea that a logically closed set holds that $s(K) = Cn(s(K))$. For example, if $A \in s(K)$ and $A \vdash \alpha$, then $\alpha \vee \beta \in s(K)$, as the $\vee$ connective needs only one of the operands to hold true for the expression to also hold true. This makes belief sets functionally infinite. For example, if an agent believes "pigs can not fly", combining this sentence with any other with "or" will always result in a sentence that belongs to $s(K)$, no matter how absurd or unrelated ("pigs can not fly or pigs are deities").

Because of this, it is relevant to talk about belief bases, which help to deal with the fact that belief sets are as large as the language for the sentences allows it to be (if it is infinite, so it the set). Belief bases are states represented by sentences which are not logically closed, such that, considering a belief state K in which $s(K) = S$, a set of sentences A is a belief base if and only if $S = Cn(A)$. This definition does not mean they are necessarily finite, but are limited realistically by the size of A. It is further defined:

1. α is a belief iff $\alpha \in Cn(A)$;
2. α is a basic belief iff $\alpha \in A$;
3. α is a derived belief iff $\alpha \in Cn(A)$;

Having in mind the KBAI project is in itself a computational system, belief bases present themselves as a much more feasible option for representing belief in a way very natural to workings of a computer. Approaching belief in this way means any changes are made to the original belief base, with any derived beliefs changing in tandem to accommodate the changes. This gives the

beliefs in $Cn(A)$ a lot more ephemeral existence, not being worth holding on their own unless a logical consequence of A and thus being removed as soon as the basic belief that made it hold is gone. This idea of removing all derived beliefs after the basic beliefs in which they buttress is gone is called disbelief propagation. Hansson furthers the exploration of belief bases, including how they update after inputs, but for the sake of brevity, this study will not cover those notions directly.

Thus is closed the last topic of research, belief change. There is much more to this than what was shown here, as the field is quite extensive, but with the brief exploration made here the main concepts needed to understand the basic notions on belief change were mostly covered. As with the previous topics, one may note the lack of a formal standard in belief representation and modeling, balanced by a rather rich literature which holds many proposals and models to better understand how an agent holds and updates their beliefs as new information is given to them, forcing them to criticize what they believe in order to hold (what seems to them) a consistent belief state. With the research covered, a conclusion now follows with a prospect of the future work surrounding this project.

3.6 Conclusion

This document covered the planning and initial research parts of the KBAI project, going through a brief research of the existing literature. It is worth noting, again, that none of the fields explored constitute independent fields, not truly: decision problem usually come down to preferences the relations between them, both preference and decision can easily come down to how the agent feels about its environment (that is, their beliefs) and social choice can be succinctly explained as the art of merging the preferences of multiple agents.

The KBAI project aims to build a recommendation system through the use of a knowledge base, for which this study aims to contribute a functionality of preference aggregation. In 3.2, the essentials of decision problems were covered in order to better understand the remaining fields, explored in sections 3.3, 3.4 and 3.5. The contents included in 3.3 aim to help model preferences and relationtions between such preferences, while 3.4 covered the necessary notion to aggregate them. The last field, approached in 3.5, will come into play later in the project, in order to revise user profiles.

These study of decision, preference and belief reveal how truly complex human rationality can be to model. The nuance needed in order to properly represent each small aspect of choice making by rational agents are, thus, fundamentally at odds with the (seemingly) simpler mathematical approaches applied to other scientific fields. This essentially means that any kind of modeling will either seem erratic and lacking in strong rules in order to represent a problem or that, in the other hand, will way be too strict to account for the more unpredictable aspects of an agents mind.

Despite how fundamental these concepts are for understanding human rationality, from the state of the art it is abundantly clear that no formalized modeling exists, though some models take the spotlight in comparison to others. This does not always mean that they are strictly better, as [8] dutifully reminds the reader: the best model is dependent on the desired approach one desires to take with any given problem. Though the conclusion in that work refer specifically to preference relations, they can easily be said to apply to any of the fields studied here, as no problem is quite like the other, neither any modeling fits perfectly into every situation. A common allegory given in respect to this conundrum lies in the London Underground railway system and its map. It is

very possible to fit the line into a common map and leave it at that, but what use does the map's user have for every minutiae when the only thing they need to know is which station is next to the other station? It is true that sometimes the user does not know exactly where they need to go, but the vast majority of the Tube's users are local commuters all across Greater London and know exactly where they want to drop off. Thus it makes perfect sense to reduce the map to the subway lines, their stops and what stations allow for line switching. The very same idea is what drives the conclusions in [8]: not all information is relevant for any given problem, and sometimes (most times) modeling involves dropping redundant information in order to better understand the kernel of a problem.

The research made here serves, as such, more as an introductory exploration to the themes of decision making and related scientific fields. The next section of this document will concern itself with the steps towards the implementation of a group recommendation system based on the concepts laid down in this study. From the bibliography listed, on the very least [13] will be frequently consulted in order to list aggregation methods and voting rules with which to build the recommendation system.

4 Voting rules

4.1 Introduction

This section contains a collection of voting rules which were explored in the initial phases of development with the aim of choosing one of them for implementation. First, five main properties are going to be listed, followed by the collected voting rules. The final part of this section will detail the chosen method of aggregation.

4.2 Properties

There are a number or properties a voting rule may possess which reflect how fair they are. These rules also require a binary preference relation so each voter may express their vote, such that, for two alternatives a and b:

- $a \succeq b$ means "a is at least as preferable as b";
- $a \succ b$ means "a is strictly preferable to b";
- $a \simeq b$ means "a and b are equally preferable".

With these relations defined, in order to describe the properties, let us have that [12] [2]:

- R is a voting rule;
- $A = \{a_1, a_2, \ldots, a_n\}$ is the set of n alternatives being voted on, such that $n \geq 2$;
- $V = \{v_1, v_2, \ldots, v_k\}$ is the set of k voters, such that $k > 2$;
- $P = <\succeq_1, \succeq_2, \ldots, \succeq_k>$ is the profile of k preference relations, such that $\succeq_i$ is the voter v_i's preference relations concerning the alternatives in A, with $1 \leq i \leq k$.
- $\succeq_R$ is the social preference relation obtained from applying R to P.

From these definitions, $a_1 \succeq_i a_2$ means the voter v_i believes a_1 is at least as preferable as a_2, for example.

Universality of domain

R's domain is the set of all logically possible P profile, such that each of the preference orderings are complete and transitive.

This means that any voter v_k has an ordering $\succeq_k$ in which, for any two alternatives a and b belonging to A, $a \succeq_k b$ or $b \succeq_k a$. In other words, all the voters in V compare all the alternatives in A with each other.

Ordering

For any profile P, the relation $\succeq_R$ obtained from applying R is complete and transitive.

This means that for any two alternatives a and b belonging to A, $a \succeq_R b$ or $b \succeq_R a$.

Pareto-optimality

For any profile P, if the relation $\succeq_R$ obtained from applying R is such that $a_i \succeq_R a_j$, then every relation in P is such that $a_i \succeq a_j$.

This means that the preferences contained in the relation $\succeq_R$ are reflected in all voters.

Independence of Irrelevant Alternatives

For any profiles P and P' and any two alternatives $a_i, a_j \in A$, if for any voter $v_i \in V$ it stands that $a_i \succeq_i a_j$ and $a_i \succeq'_i a_j$, then $a_i \succeq'_R a_j \longrightarrow a_i \succeq_R a_j$.

This means that, for two different profiles, if all voters have the same preference ordering in both, then the resulting aggregation must also maintain the same preference ordering between the two profiles.

Non-dictatorship

For any profile P, there is no voter $v_i \in V$ such that, for two alternatives $a_i, a_j \in A$, $a \succeq_i b \longrightarrow a \succeq_R b$.

This means that no voter has the ability to change the outcome to their preference, since if the condition $a \succeq_i b \longrightarrow a \succeq_R b$ were to be met for a given v_i, that voter would have the ability to decide the outcome of R themselves, making the rule dictatorial.

4.3 Voting rules

Here, the collected voting rules are listed. The symbols here used obey the definitions show previously to definine the properties.

Plurality

For a profile P, v_i's vote is the alternative preferred in the relation $\succeq_i$ to all the others. The alternative with the most votes is the winner.

Scoring rules

In this rule, each voter in V assigns a value to each alternative in A, such that each one has a weight w, where the weight of a_p is w_p. The candidate in A whose sum of the weights assigned to them is the greatest is considered the winner.

The way the weights are assigned varies. Let $(w_1, w_2, \ldots, w_n)$ be the vector representing the weights assigned to each alternative in A by a given voter in V:

1. $(1, 0, \ldots, 0)$ represents an analog to the rule in 4.3, in which the voter selects only one of the alternatives (given a weight of 1), denying all the others (with a weight of 0).
2. $(1, 1, \ldots, 0)$ represents an inverse of 4.3, and is called the veto rule or *anti-plurality*, in which the voter selects the alternative that *does not* prefer at all (giving it a weight of 0), and showing indifference between the others (with a weight of 1).
3. $(n-1, n-2, \ldots, 0)$, called the *Borda* rule. In practice, the weights are assigned so that, when ordered, the vector reflects the preference ordering of the corresponding voter. Like the rule in 4.3, it allows the voter to choose a preferred alternative, but also offers a nuance in allowing an opinion on the remaining alternatives by explicitly ordering them.

Cup

A rule based on the idea of a tournament where pairs of alternatives in A compete in pairs through a vote of the kind set out in 4.3. The winner then competes with the winner of another

pair, repeating the process until no alternative remains but the winner. This method can be seen as a binary tree in which each node contains the winning alternative and its leaves contain the pair of alternatives that competed. The root of the tree contains the alternative chosen in the method.

Copeland

In essence, each alternative a_p receives a Copeland score, given by the sum of alternatives to which a_p is most preferred, subtracted by the sum of alternatives preferred to a_p. The winning alternative holds the highest score, with ties being decided by choosing the alternative whose defeated competitors (i.e. to which a_p is preferred) have the highest summed score.

Simpson

For this rule, the sum of preferred alternatives is again used. For each alternative a_p, the number of alternatives that are preferred to it is calculated. The alternative with the lowest sum wins the election.

Plurality with runoff

This method reinforces the rule in 4.3, in which for an alternative to be considered the winner, it must have an absolute majority, i.e. more than $k/2$ voters in V must vote for the alternative for it to win. If this is not the case, the two alternatives with the most votes enter a new vote in order to extract the winner.

Single Transferable Vote (STV)

Similar to the 4.3 method, several consecutive votes are also taken in order to decide the winner. Each voter in V orders the alternatives in A according to their preferences, so that the alternative that the fewest voters put first is eliminated and a new round of voting begins. The winning alternative is the one that obtains a majority in which $k/2$ of the voters selected it as their first preference.

Baldwin

Similar to the rule in 3, but iterated so that, after each vote, the alternative in A with the lowest score among the alternatives as a whole is eliminated and the next vote is taken, repeating until the winner is obtained.

Nanson

Also similar to the rule in 3, but iterated so that, after each vote, the alternatives in A with less than the average of the scores of the alternatives as a whole are eliminated and the next vote is taken, repeating until the winner is obtained.

Approval

In much the same way as the 1 and 2 methods of the 4.3 rule, each voter in V selects the candidates they approve of (giving them a weight of 1 in terms of comparison), leaving the rest unrated (in practice, giving them a weight of 0). The alternative with the most approvals is chosen as the winner.

Range

This works in the same way as 4.3, but instead of restricting each voter in V to accepting or denying alternatives in A, they are given a range of scores with which to evaluate them and the one with the highest score in the end wins.

Cumulative

A method that requires more deliberation, but proceeds in the same way as described in 4.3, where the range is replaced by a 'budget' to be distributed by each voter in V on each alternative in A, again winning the election the one with the highest score.

Bucklin

Using this rule, the winner of this rule is the alternative in A that obtains more than $^k/_2$ votes that puts it in first place. If no alternative meets this condition, the one with more than $^k/_2$ votes that puts it in second place wins. The process is repeated in this way (third place, fourth place, etc.) until the alternative that meets the condition is found. In the event of a tie, the alternative with the most votes wins.

4.4 Final aggregation algorithm

The collection of literature allowed for the parsing of a variety of voting rules, through which an algorithm was constructed. Let us have that:

- i is an item a profile may have an interest for;
- v is a value of preference, such that v_k is the value of preference towards i_k, with $k \in N$;
- p is a profile with a set of $k \in N$ preferences $\{i_1, i_2, \ldots, i_k\}$ and the respective values $\{v_1, v_2, \ldots, v_k\}$;
- G is a group of $n \in N$ profiles $\{p_1, p_2, \ldots, p_n\}$;
- v_{n_k} is the preference value of the profile p_n over the interest i_k, $n, k \in N$;
- p_G is the aggregation of the profiles in G;
- W is a function that calculates the weight of a profile in a group, such that $W(p)$ represents the p profile's weight.

Thus, in a group G with $n \in N$ profiles, for any interest i_k, p_G's preference value towards it is given by:

$$v_{G_k} = \frac{\sum_1^n (v_{n_k} \cdot W(p_n))}{\sum_1^n W(p_n)}$$

This algorithm allows certain profiles to have a different say in the final aggregation, be it more of less. It is useful for groups whose members do not share equal power of decision, such that if $W(p) = 0$, then p has no weight in the aggregation. Furthermore, if no profile as any weight in the group, then the resulting aggregation will possess no preferences (i.e., it will contain indiference towards all interests).

4.5 Conclusion

Exploring voting rules and their properties allowed for a more complete understanding of the social choice theoretic field.

Using the notions collected from existing literature, an algorithm was built in order respond to the necessities of the KBAI project.

This algorithm was verified by the project leaders and was thus greenlit for implementation, discussed in the following sections of this report.

5 Tools

5.1 Introduction

In this chapter, the tools used throughout the project will be described both on their nature and their application during development. For each of the described tools, the reasoning behind the choice that led to their use will be provided when appliable in order to contextualise how their properties allow furthering the goals of the project.

5.2 Graph drawing software

In the preliminiary stages of development, diagrams were used to describe the general architecture and functions of the system. To create them, a drawing software was needed.

To that end, *diagrams.net* [17] was chosen. It is a cross-platform graph drawing software used to create diagrams such as flowcharts, wireframes, UML diagrams, organizational charts, and network diagrams. Parts of its source code are provided under the Apache 2 open-source license and it is available freely as an online web app and as an offline desktop application, the latter of which was used in the project. Supported storage and export formats to download include PNG, JPEG, SVG, and PDF, though the app also provides a .drawio format for diagrams it creates for later editing.

5.3 Knowledge base

The centerpiece of the KBAI project, as noted by its title, is the knowledge base, a repository where all information may be stored and processed, serving as a resource for other modules within the project. An ontology was developed to represent the knowledge base. However, due to confidentiality agreement, the ontology and its accompanying description can not be provided.

5.4 Programming language

The implementation of the aggregation module for the KBAI project required the choice of a programming language. Because the modules all work separately from each other, communicating only through the knowledge base, the choice comes down to each of the modules' developers. For preference aggregation, *Rust* was chosen.

Rust [18], or sometimes *Rustlang*, is a multi-paradigm, general-purpose programming language designed with a heavy emphasis on performance, type safety, and concurrency. Memory safety is ensured by making sure all references point to valid memory without the use of a garbage collector or reference counting mechanisms often used in other languages.

Enforcing memory safety and preventing data races is done thanks to its "borrow checker" that tracks all references' object lifetimes in a program during compilation. *Rust* makes use of concepts from functional programming, such as static types, immutability, higher-order functions and algebraic data types. Coupled with the language is a build system and package manager *Cargo* allows the download, compilation and distribution of libraries (named *crates*) mantained in an official registry.

The final version of the module will likely be executed on containers, thus requiring the attributes of both safety and performance present in *Rust*. This factor motivated the choice.

5.5 IDE

The choice of IDE came down to general flexibility and compatibility with the chosen language, as well as performance in lower end machines.

Visual Studio Code [19] was chosen both for being lightweight and powerful enough to allow multiple customizations that tailor the interface and functionalities to the developer. The fact *Rust* is supported by extensions provided by the IDE's *Marketplace* and that it has integration with the *GitHub* platform cemented it as a clear choice for development.

5.6 Conclusion

In conclusion, the tools chosen for any project always bear weight in how development proceeds. The choices made for KBAI were done with the most possible care in order to not reach serious obstacles while conceiving all the parts necessary for progressing towards the objectives.

It is worth noting that, aside from the use of the knowledge base, the modular property of the system naturally allowed for each of the teams in the KBAI project to chose their own tools according to their discretion and taste. The *diagrams.net* and *Visual Studio Code* were used for it's familiarity and frequent usage in other projects. On the other hand, the choice of *Rust*, aside from its objective qualities, came from a place of personal interest on the possibilities provided by applying a language not used before in a real, practical setting.

Moving forward, the choices made during this project ultimately shaped how the preference aggregation module would come to be built. The next chapter deals with the various steps of development towards it.

6 Developemnt

6.1 Introduction

In this chapter, the development of the aggregation engine will be described thoroughly, going from the initial explorations to the last version delivered as of the writing of this document. Firstly, an explaination of what is being implemented, followed by a description of the iterative model of development. As such, this part will touch on the main milestones of implementation up to the last one, followed by an analysis of what future version could look like.

The first step of development was the conception of a dataflow diagram that allowed for the clear outlining of the necessary processes. For implementation, in turn, the following milestones may be established:

1. minimal functional executable;
2. data retrieval from knowledge base;
3. data processing from retrieval;
4. aggregation process;
5. data upload to knowledge base;
6. HTTP service for handling API requests;
7. command line options for customization;
8. support for characteristic aggregation;
9. veto functionality.

These steps do not all share the same length nor the same overall relevance to this dissertation's main theme, the preference aggregation capabilities. Nevertheless, they form a single final product by the end and as such will be included in this chapter.

Any references to the KBAI's API address will be redacted in the form of an ellipsis.

After these steps were concluded, the final documentation was built in order to ensure the aggregation module's mantainability, which will also be listed further into this chapter.

6.2 Prototyping

The first step in development required a solid decision on which algorithm would be implemented. Using Python (see Appendix B), five different options were reviewed, and the leadership settled on the one that most suited the project's needs. Five voting rules were tested:

1. Plurality;
2. Scoring Borda Rule;
3. Scoring Veto Rule;
4. Cumulative;
5. Cumulative Average.

A simple JSON (see Appendix B.2) with a mock group for testing was written using the code in B.1, with three users with different preferences on candidates. Running the *main.py* file outputs the following:

```
1  Plurality: [('A', 3), ('B', 2), ('C', 1)]
2  Scoring Borda Rule: [('A', 15), ('B', 13), ('C', 8)]
3  Scoring Veto Rule: [('A', 6), ('B', 5), ('C', 2)]
4  Cumulative: [('A', 1.9000000000000001), ('B', 1.3999999999999997), ('C',
       0.10000000000000012)]
5  Cumulative Average: [('A', 0.3166666666666667), ('B',
       0.23333333333333328), ('C', 0.016666666666666687)]
```

This allowed for a clear analysis of how the changes to *group.json* affected the aggregation's results. In the end, the Cummulative Average was chosen as it already possessed compatibility with the recommendation module's value range (defined in advance with the project collaborators), property afforded by the averaging and assuming all the preference values in the group's data fluctuate between -1 and 1.

6.3 Minimal functional executable

The creation of the project using Cargo was the first step, through the command:

```
1  cargo new motor
```

This prepared a new folder, "motor", with the bare minimum contents:

- a *.git* folder and a *.gitignore* file for interoperability with Git;
- a *src* folder containing the package's source code, with a *main.rs* file containing the instructions required to output the common "Hello, World!" test;
- a *Cargo.toml* file used to control the package's information about itself and its required dependencies.

Running the executable at this point outputted "Hello, World!" to the console.

6.4 Data retrieval from knowledge base

Retrieving the data from the knowledge base was done with its team's assistance, who had by this point provided the template used for retrieving a mock group's preferences. A single identification number, 41, was provided for the group, along with the appropriate template's, 14, called "Groups with Interests".

To first assert if the template worked, the following link was accessed in an internet browser:

```
.../Template/Main/get?templateId=20&rootNodeId=41
```

This yielded the group's data, as shown in Appendix A.3, which is always formated as an array (*nodes*) even for single results. Within it there is a *customer* representing a group containing three members within its *nodes* array. Translating this process into code resulted, broadly, in the following excerpt:

```
1       let url : String = format!(".../Template/Main/get?templateId={
            GET_GROUP_CUSTOMERS_AND_INTERESTS_TEMPLATE_ID}&rootNodeId={
            root_node_id}");
2       let response : Response = send_http_request(Method::GET, url, String::
            new())?;
3
4       let body : &Vec<u8> = &response.bytes().unwrap().to_vec();
5       let data : String = String::from_utf8_lossy(&body).to_string();
6       let json : Value = serde_json::from_str(&data)?;
```

Going by order, the first line defines the link to the template, while the second one sends it and assigns the retrieved data to a variable *response*, after which the data is converted through lines four, five and six into a more practical JSON format.

The associated code is present in Appendix C.3.

6.5 Data processing from retrieval

A JSON containing a group's data needs now to be processed into matching data structures in order to better handle it going foward. Firstly these structures will be described, followed by the actual method of converting the JSON into them.

The data structures consist of:

1. *Preference*, a particular preference of a profile;
2. *Profile*, a single user's profile, with its preferences included;
3. *Group*, a group of user profiles.

A Preference consists of the particular interest's ID in the knowledge base and its corresponding preference value (how much the profile likes/dislikes the interest):

```
1 pub struct Preference
2 {
3     pub id : u64,
4     pub value : f64
5 }
```

A *Profile* contains a customer's ID in the knowledge base, its weight in the group and a vector of Preferences:

```
1 pub struct Profile
2 {
3     pub id : u64,
4     pub weight : u64,
5     pub preferences : Vec<Preference>
6 }
```

A *Group* contains its matching knowledge base ID along with its constituents' profiles:

```
1 pub struct Group
2 {
3     pub id : u64,
```

```
4       pub profiles : Vec<Profile>
5   }
```

With the structures defined, all that is needed is to parse the JSON's contents obtained from the knowledge base through the template and convert it into a *Group* type object. This is done, in code, by a three functions (one for each structure) nested within eachother, in which the *Group*'s building function runs the *Profile*'s building function through the group's *nodes* JSON array, which in turn runs the *Preference*'s building function through each of the customer's preference arches to build a vector of *Preference* objects.

Below, the relevant code for the JSON's processing:

```
1   pub fn build_group(group_data : &Value) → Group
2   {
3       let group_id : u64 = group_data["nodes"][0]["existingNodeId"].as_u64().
            unwrap();
4       let group_profiles : Vec<Profile> = build_profiles(&group_data);
5
6       let group : Group = Group { id: group_id, profiles: group_profiles };
7
8       return group;
9   }
10
11  fn build_profiles(group_data : &Value) → Vec<Profile>
12  {
13      let mut profiles : Vec<Profile> = Vec::new();
14
15      let user_data_array = group_data["nodes"][0]["nodes"].as_array().unwrap
            ();
16
17      for user_data in user_data_array
18      {
19          let profile = build_profile(user_data);
20          profiles.push(profile);
21      }
22
23      return profiles;
24  }
```

The cycle in line seventeen (in the *build_profiles* function) calls upon the *build_profile* function for each customer in the group:

```
1   fn build_profile(customer_data : &Value) → Profile
2   {
3       let preferences : Vec<Preference> = build_preferences(&customer_data["
            arches"]);
4
5       let profile : Profile = Profile { id: customer_data["existingNodeId"].
            as_u64().unwrap(), weight: DEFAULT_PROFILE_WEIGHT ,preferences :
            preferences};
6
```

```
7       return profile;
8   }
9
10  fn build_preferences(customer_arches_data : &Value) → Vec<Preference>
11  {
12      let mut preferences : Vec<Preference> = Vec::new();
13
14      for arch in customer_arches_data.as_array().unwrap()
15      {
16          if arch["typeId"].as_u64().unwrap() == PREFERENCE_TYPE_ID
17          {
18              let preference = build_preference(arch);
19              preferences.push(preference);
20          }
21      }
22
23      return preferences;
24  }
```

And in turn, the cycle in line fourteen (in the *build_preferences* function) calls upon the *build_preference* function for each arch:

```
1   fn build_preference(arch_data : &Value) → Preference
2   {
3       let preference_id_json = &arch_data["node"]["existingNodeId"];
4
5       let preference_weight_json = match arch_data["properties"].get("ExpValue
            ")
6       {
7           Some(value) ⇒ value,
8           None ⇒ &arch_data["properties"]["ImpValue"],
9       };
10
11      let preference_id = preference_id_json
12          ∘ as_u64()
13          ∘ unwrap();
14
15      let preference_weight = preference_weight_json
16          ∘ as_str()
17          ∘ unwrap()
18          ∘ trim_end()
19          ∘ replace("λ"", "")
20          ∘ parse::<f64>()
21          ∘ unwrap();
22
23      let preference : Preference = Preference { id : preference_id, value :
            preference_weight };
24
25      return preference;
26  }
```

A preference value may contain explicit and/or implicit values. In line 5 (in the *build_preference* function), the *match* block asserts that if an explicit value is present, use it; otherwise, use the implicit one.

The JSON by this point has been converted into a *Group* object, and is now ready to be further handled by the aggregation module.

The associated code is present in C.3.

6.6 Aggregation process

With the JSON turned into a *Group* object, it is now possible to proceed with the aggregation using the data provided by the knowledge base. The preference aggregation function, in short, calculates a weighted average for each preference. For this, all the group's preferences are compiled into a hash map with key/value pair in wich the key is an interest's ID and the value is a vector of *WeightedPreference* objects. A *WeightedPreference* is a structure created to associate a *Profile*'s weight in the group with one of its preference values:

```
1  pub struct WeightedPreference
2  {
3      pub profile_weight : u64,
4      pub preference_value : f64
5  }
```

Using this structure, all the group's profiles are parsed with the *compile_preference_weights* function:

```
1  fn compile_preference_weights(group : &Group) → HashMap<u64, Vec<
       WeightedPreference> >
2  {
3      let mut preference_weights : HashMap<u64,  Vec<WeightedPreference> > =
           HashMap::new();
4
5      for profile in &group.profiles
6      {
7          for preference in &profile.preferences
8          {
9              let weighted_preference = WeightedPreference{profile_weight :
                   profile.weight, preference_value : preference.value};
10
11             match preference_weights.entry(preference.id) {
12                 Entry::Vacant(weights) ⇒ { weights.insert(vec![
                       weighted_preference]); },
13                 Entry::Occupied(mut weights) ⇒ { weights.get_mut().push(
                       weighted_preference); }
14             }
15         }
16     }
17
18     return preference_weights;
19 }
```

In brief words, the *compile_preference_weights* function cycles through each profile within the group, and for each it parses their preferences. For each of a profile's preference, an object of type *WeightedPreference* is created pairing the profile's weight with the its preference towards a given interest. Then, that *WeightedPreference* is put into the hash map using the interest's ID as key. This function is called by the *aggregate_preferences* function, which uses the resulting hash map to execute the aggregation:

```
1  fn aggregate_preferences( group : &Group, aggregated_profile : &mut Profile)
       → ()
2  {
3      let group_preference_weights : HashMap<u64, Vec<WeightedPreference>> =
           compile_preference_weights(group);
4
5      for (interest_id , preference_weights) in group_preference_weights
6      {
7          let added_weights : f64 = preference_weights.iter().map(|w| w.
               profile_weight).sum::<u64>() as f64;
8          let added_values : f64 = preference_weights.iter().map(|w| w.
               preference_value * w.preference_weight).sum();
9          let preference = Preference { id : interest_id , value :
               added_values/added_weights };
10         aggregated_profile.preferences.push(preference);
11     }
12 }
```

The function receives the *Group* object containing the information and a *Profile* type variable into which the resulting aggregation will be saved. Using the compiled weighted preferences, for each of the key values in the hash map, all the profile weights are added into the *added_weights* variable (line 7) and all the products of preference values and profile weights are added into the *added_values* variable (line 8). All that remains now is creating a *Preference* object with the hash map's key as the *preference_id* field and the the weighted average (obtained by dividing *added_values* by *added_weights*) as its *value* field (line 9).

After going through all the key values in the hash map, the *Profile* argument passed initially now contains the resulting aggregation.

The associated code is present in C.1.

6.7 Data upload to knowledge base

The project required that the handled group's data be sent back to the knowledge base after aggregation, done yet again through a template, this using "Customer with Interests". This template, numbered 14, is accessed similarly to template 20, "Groups with Customers":

```
.../Template/Main/get?templateId=14&rootNodeId=<group_id>
```

However, because it is now needed to send information as opposed to retrieving it, the "get" is changed to a "populate":

```
.../Template/Main/populate?templateId=14&rootNodeId=<group_id>
```

To send the information, the HTTP request must include a properly formated JSON within its body resembling one that would be received when retrieving a customer's information (Figure 8), with the aggregated preference values within its *arches* array. The *Rust* language allows the implementation of methods for its structures, so this feature was used to format the desired JSON string, both for the $Profile$ and $Preference$ structures (see Appendix C.3, from lines 162 through 233).

Fig. 8. Template 14, "Customer with Interests", example.

The resulting JSON (see Appendix A.4 for a working example) is then sent as an HTTP request to the knowledge base in order for the aggregation to take effect:

```
1 let mut preference_wrapper_map : serde_json::Map<String, Value> = serde_json
      ::Map::new();
2 preference_wrapper_map.insert("nodes".to_owned(), Value::Array(vec![group.
      as_json_with_preferences()]));
3
4 let preference_output : String = serde_json::to_string_pretty(&Value::Object
      (preference_wrapper_map)).unwrap();
5 let preference_url : String = format!("../Template/Main/populate?templateId
      ={POPULATE_CUSTOMER_AND_INTERESTS_TEMPLATE_ID}");
6 let response : Response = send_http_request(Method::POST, preference_url,
      preference_output)?;
```

The HTTP response contains the information of whether or not the operation was done correctly, which at this point is merely outputted to the console, regardless of success.

6.8 HTTP service for handling API requests

Before the executable is to be released to the remaining teams of the project, HTTP service was required to be embedded into the aggregation module so it could be ran in a container to receive requests. To this end, the *rocket* crate was used:

```
    let config : Config = match Config::build(Environment::Development)
    ∘ port(8080)
    ∘ finalize()
    {
        Ok(value) ⇒ value,
        Err(error) ⇒ { println!("{error:?}"); return;},
    };

    rocket::custom(config)
        ∘ mount("/aggregation", routes![aggregation_request_handler])
        ∘ launch();
```

This created a route that matched any requests sent to *localhost* : 8080/*aggregation*/ < *group_id* > to the following handler:

```
#[get("/<group_id>")]
fn aggregation_request_handler(group_id : u64) → JsonValue {
    match handle_group(group_id)
    {
        Ok(_) ⇒
        {
            let _ = log(format!("Aggregated preferences of group {group_id
                }."));
            JsonValue(json!(Response{reply : format!("SUCCESSFULY AGGREGATED
                GROUP {group_id}")}))
        }
        Err(e) ⇒ JsonValue(json!(Response{reply : format!("FAILURE ⇒ {e}")
            }))
    }
}
```

This handler calls on the function *handle_group* to use the given group's ID in order to execute the aggregation using the processes described on the previous interations:

```
pub fn handle_group(group_id : u64) → Result<(Vec<String>,Vec<Response>),
    Box<dyn Error> >
{
    let group_data : Value = get_group_data(group_id)?;
    let group : Group = build_group(&group_data);
    let aggregated_profile : Profile = aggregate_group(&group);

    let responses : (Vec<String>,Vec<Response>) =  return_group_data(
        aggregated_profile)?;

    Ok(responses)
}

fn aggregate_group( group : &Group ) → Profile
{
    let mut aggregated_profile : Profile = Profile { id : group.id, weight :
        DEFAULT_PROFILE_WEIGHT, preferences : Vec::new()};
```

```
15
16      aggregate_preferences(group, &mut aggregated_profile);
17
18      return aggregated_profile;
19  }
```

In short, what was once hardcoded (the group ID 40 used through development) is now received through a simple HTTP request made through the address:

```
1       [module's IP address]/aggregation/<group_id>
```

This initial implementation opens on port 8080 of *localhost*, as was now ready to be sent to the KBAI project's other teams for usage in testing.

The associated code is present in Appendices C.1, C.3 and C.7.

6.9 Command line options for customization;

This iteration consisted of an addition that allowed the specification of the port when running the executable in the command line:

```
1       let port : u16;
2
3       if args.len() > 1
4       {
5           port = match args[1].parse::<u16>()
6           {
7               Ok(value) ⇒ value,
8               Err(_) ⇒ { println!("Error while parsing the port value.");
                    return;},
9           };
10      }
11      else {
12          port = 8080;
13          println!(" » Port unspecified; using {port}.λn");
14      }
```

If a number is provided, it is used as port if valid or an error is raised if invalid. If none is provided, it defaults to 8080 used through development.

The associated code is present in C.7.

6.10 Support for characteristic aggregation

Midway through development, the requirement was imposed on the role that characteristics would play while aggregating various profiles into a collective one.

The nature of these characteristics are by and large a parallel to how preferences are represented in the knowledge base, a *characterisic* connected by an *Is* arch to a *Customer*. As such, a structure was created to represent both entities (the characteristic's ID and its value as inscribed within the arch's properties) within the code:

```
pub struct Identity
{
    pub id : u64,
    pub value : String
}
```

The *Profile* structure was thus altered accordingly to accomodate this addition:

```
pub struct Profile
{
    pub id : u64,
    pub weight : u64,
    pub preferences : Vec<Preference>,
    pub identities : Vec<Identity>
}
```

These characteristics were obtained through template 51, "Customer with Interests", for each of the group's members individually, as there was no template similar to 20 ("Groups with Interests") which provided a group and all its members' characteristics. After retrieving the data from the knowledge base pertaining to all the group's characteristics, these were compiled into a hash map in such a way that, for each individual characteristic ID (the key) existed a vector containing all existant values related to it (the value). This means, in an hypothetical case, two members might have *Is* arches connecting to the same *Characteristic* node representing language with two distinct values, "Tagalog" and "Hebrew", in which case the node's ID would be paired with a vector containing both values. Each of these values are then pushed into the group's aggregated profile, more specifically into its *identities* field, as two different *Identity* objects.

Saving the aggregated characteristics follows the same method used for preferences, formating the aggregated profile and its identities into a properly formated JSON (shown in Appendix A.2) and sent as an HTTP request to the address:

```
    ◦ ../Template/Main/populate?templateId=51&rootNodeId=<group_id>
```

The process of incluing characteristics into the overall aggregation will not be described, but is available in Appendix C.

6.11 Veto functionality

The final step of development was dedicated to the creation of an additional HTTP service to be provided by the module, which included the possibility of vetoing a given interest for varying lengths of time. This part was made largely without any changes to the previous parts discussed in this section, as its processes do not meddle with the general aggregation system developed.

The vetoing service was required to provide a way for a particular *Customer*'s *Prefers* arch to a particular *Interest* to have its value changed to zero (vetoing it in the eyes of the recommendation module) either permanently or until a given point in time. In this latter case, the value present pre-vetoing had to be reset after the expiration.

Vetoing required the sending of an HTTP request in the form of the address:

```
    [module's IP address]/feedback/veto/<user_id>/<interest_id>/<veto_length
        >
```

This would cause the *Customer* of ID *user_id* to have its *Prefers* arch's value to the *Interest* of ID *interest_id* to be set to zero for how many hours provided in *veto_length*.

If the value for the length is zero, then it is permanent; otherwise, the veto's information, including the pre-veto value, are then saved in a local text file, regularly parsed by the module in order to assert if any have expired and resetting the value if any is found to have run its course.

The associated code is present in Appendix C.4.

6.12 Final Iteration

The final executable allows for conversion of a group with multiple profiles into a single collective one by aggregating each of the individual profile's preferences and characteristics. Additionally, it also provides a way for a user to have an interest vetoed.

From the time the executable is run, its functioning may described as such:

1. The server starts up, opening up in the specified port;
2. Await requests for aggregation or vetoing;
3. If a request for aggregation is received, the group's information is retrieved, aggregated and sent to the knowledge base;
4. If a request for vetoing is received, the values get updated in the knowledge base and temporary vetos have their expiration recorded locally.

Not described in this section are the handling of errors (see Appendix C.2) and minor logging utilities (see Appendix C.5). Errors are handled in a unique manner in the *Rust* language in comparison to most others, in such a way that required the creation of custom errors to accomodate to the specifics of this project.

As mentioned previously, this software will now be run from a container mantained by other members of the KBAI project, with the aim of future user testing.

6.13 Conclusion

Summarily, the development of the aggregation module followed a number of milestones that may be used to frame the various steps that built up to the last realease of the executable. The aggregation is proven to be working by the output JSON produced by the module (see Appendix A.4).

Some of these steps are more closely related to preference aggregation than others (namely, up to the sixth iteration), as requirements shifted constantly throughout the project's timeline. In the end, features not initially planned for were included, such as characteristics and an explicit vetoing mechanism.

With the development covered in this section, the resulting product and future works will now be discussed in the closing paragraphs of this dissertation.

7 Conclusion and Future Work

Closing this dissertation, it is assertable that the main practical objetive was achieved, such that an executable was devised that allowed:

1. Aggregating group preferences;
2. Aggregating group characteristics;
3. Explicit vetoing.

Although all of these contribute considerably to KBAI's objective, only the first one was truly set on the beggining of development. As such, most of the work in this document focuses on the preference aggregation aspect of the project.

This document covered the planning and initial research parts of the KBAI project, going through a brief research of the existing literature. This study of decision, preference and belief reveal how truly complex human rationality can be to model. The nuance needed in order to properly represent each small aspect of choice making by rational agents are, thus, fundamentally at odds with the simpler mathematical approaches applied to other scientific fields. This essentially means that any kind of modeling will either seem erratic and lacking in strong rules in order to represent a problem or that, in the other hand, will way be too strict to account for the more unpredictable aspects of an agents mind.

The choice of tools for any project weights in how development comes to be. The choices made in this project were done with all the information possessed at the time so as to not run into significant obstacles while building the software that would allow for the objectives to be met.

As for the development of the aggregation module, it followed a number of milestones that may frame the project's timeline up to the last realease of the executable. Requirements shifted constantly throughout the project's timeline and in the end, features not initially planned for were included, such as characteristics and an explicit vetoing mechanism, though not at the expense of the initial objetive.

The project provided ample opportunity to explore previously unfamiliar areas of software development, both through simple tasks and more complex endeavours. The usage of a knowledge base afforded a fascinating alternative to conventional databases, while the data exchange through HTTP APIs allowed for a practical introduction to a different architectural paradigm in the industry (as opposed to network sockets) not previously studied through the curriculum. The usage of the Rust language also provided a chance to explore an unfamiliar programming tool, with very intriguing new mechanics such as the borrowing of data, lifetimes for variable and the absence of commonly used exceptions. On a more personal level, it made way for insertion in a real workplace environment and imbuing valuable notions needed for a future career in the technological field.

Future work on the developed software should, mainly, concern itself with the integration of the UI modules of the KBAI project, asserting the reliability of the executable in a practical context, fine tuning the algorithm if user testing deems it necessary and the substitution of the veto's save file by knowledge base entities in order to ensure permanency when deploying the executable in volatile containers. Though not extensive code-wise, a more comprehensive documentation would also ensure mantainability into the KBAI project's future.

References

[1] M. D. Resnik, *Choices: An Introduction to Decision Theory*. University of Minnesota Press, 1987. [Online]. Available: http://www.jstor.org/stable/10.5749/j.ctttshgd

[2] C. List, "Social Choice Theory," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2022.

[3] P. Gärdenfors, *Knowledge in Flux*. College Publications, 1988.

[4] E. Fermé and S. O. Hansson, *Belief Change: Introduction and Overview*, 1st ed. Springer Publishing Company, 2018.

[5] F. Rossi, K. B. Venable, and T. Walsh, *A Short Introduction to Preferences*. Springer International Publishing, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-3-031-01556-4

[6] S. Halldén, *On the Logic of 'Better'*. Lund, Sweden: C.W.K. Gleerup, 1957.

[7] Z. Xu, "A survey of preference relations," *International Journal of General Systems*, vol. 36, no. 2, pp. 179–203, 2007.

[8] P. Vincke, *Preference Modeling*. Boston, MA: Springer US, 2009, pp. 3055–3058. [Online]. Available: https://doi.org/10.1007/978-0-387-74759-0_521

[9] B. Hansson, "Fundamental axioms for preference relations," p. 423–442, Oct 1968. [Online]. Available: http://dx.doi.org/10.1007/BF00484978

[10] G. Grätzer, *Lattice Theory: Foundation*. Springer Basel, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-3-0348-0018-1

[11] J. C. Heckelman and N. R. Miller, "Introduction: issues in social choice and voting," p. 1–12. [Online]. Available: http://dx.doi.org/10.4337/9781783470730.00005

[12] V. C. F. Brandt and U. Endriss, *Computational Social Choice*, 2012.

[13] D. Grossi and G. Pigozzi, *Judgment Aggregation: A Primer*. Springer International Publishing, 2014. [Online]. Available: http://dx.doi.org/10.1007/978-3-031-01568-7

[14] H. Moulin, *Handbook of Computational Social Choice*. Cambridge University Press, 2016.

[15] S. O. Hansson, *A Textbook of Belief Dynamics*, ser. Theory Change and Database Updating. Kluwer Academic, 1999.

[16] G. H. von Wright, "The logic of preference," *Studia Logica*, vol. 30, pp. 159–162, 1963.

[17] draw.io. [Online]. Available: https://www.drawio.com/

[18] Rust programming language. [Online]. Available: https://www.rust-lang.org/

[19] Visual studio code - code editing. redefined. [Online]. Available: https://code.visualstudio.com/

Appendices

A Templates

A.1 Customer with Interests

A.2 Customer with Characteristics

A.3 Groups with Interests


```
nodes:
  0:
    existingNodeId:          41
    type:                    "Customer"
    typeId:                  5003
    properties:
      reference:             "Grupo do Verão"
    nodes:
      0:
        existingNodeId:      42
        type:                "Customer"
        typeId:              5003
        properties:
          reference:         "Diogo Costa"
          Birthdate:         "04/08/1998"
          Surname:           "Costa"
          Name:              "Diogo"
          contextdescription: "Role: Admin"
        arches:
          0:
            type:            "Prefers"
            typeId:          5056
            properties:
              ImpValue:      "0.74"
              ExpValue:      "-1.0"
            node:
              existingNodeId: 29
              type:          "Interest"
              typeId:        5000
```

A.4 JSON preference aggregation output


```
{
    "nodes": [
        {
            "arches": [
                {
                    "node": {
                        "existingNodeId": 29,
                        "type": "Interest",
                        "typeId": 5000
                    },
                    "properties": {
                        "Value": "−0.11249999999999999"
                    },
                    "type": "Prefers",
                    "typeId": 5056
                },
                {
                    "node": {
                        "existingNodeId": 25,
                        "type": "Interest",
                        "typeId": 5000
                    },
                    "properties": {
                        "Value": "0.37833333333333335"
                    },
                    "type": "Prefers",
                    "typeId": 5056
                },
                {
                    "node": {
                        "existingNodeId": 27,
                        "type": "Interest",
                        "typeId": 5000
                    },
                    "properties": {
                        "Value": "−0.21500000000000002"
                    },
                    "type": "Prefers",
                    "typeId": 5056
                },
                {
                    "node": {
                        "existingNodeId": 24,
                        "type": "Interest",
                        "typeId": 5000
                    },
                    "properties": {
                        "Value": "0.9"
```

```
                    },
                    "type": "Prefers",
                    "typeId": 5056
                },
                {
                    "node": {
                        "existingNodeId": 23,
                        "type": "Interest",
                        "typeId": 5000
                    },
                    "properties": {
                        "Value": "0.843"
                    },
                    "type": "Prefers",
                    "typeId": 5056
                }
            ],
            "existingNodeId": 41,
            "type": "Customer",
            "typeId": 5003
        }
    ]
}
```

B Prototype

B.1 group_template.py

```
1  import json
2  import time
3
4  group =[
5      {
6          "voter" : "1",
7          "preferences" : [
8              { "candidate" : "A", "weight" : 50 },
9              { "candidate" : "B", "weight" : 60 },
10             { "candidate" : "C", "weight" : -30 }
11         ]
12     },
13     {
14         "voter" : "2",
15         "preferences" : [
16             { "candidate" : "A", "weight" : 50 },
17             { "candidate" : "B", "weight" : 30 },
18             { "candidate" : "C", "weight" : 20 }
19         ]
20     },
21     {
22         "voter" : "3",
23         "preferences" : [
24             { "candidate" : "A", "weight" : 20 },
25             { "candidate" : "B", "weight" : 10 },
26             { "candidate" : "C", "weight" : -30 }
27         ]
28     }
29 ]
30
31 file =open(f"group_{time.time()}.txt", "w+")
32
33 file.write(json.dumps(group, indent=4))
34
35 file.close()
```

B.2 group.json

```
[
    {
        "voter": "1",
        "preferences": [
            {
                "candidate": "A",
                "weight": 50
            },
            {
                "candidate": "B",
                "weight": 60
            },
            {
                "candidate": "C",
                "weight": −30
            }
        ]
    },
    {
        "voter": "2",
        "preferences": [
            {
                "candidate": "A",
                "weight": 50
            },
            {
                "candidate": "B",
                "weight": 30
            },
            {
                "candidate": "C",
                "weight": 20
            }
        ]
    },
    {
        "voter": "3",
        "preferences": [
            {
                "candidate": "A",
                "weight": 20
            },
            {
                "candidate": "B",
                "weight": 10
            },
            {
                "candidate": "C",
```

```
                "weight": −30
            }
        ]
    },
    {
        "voter": "4",
        "preferences": [
            {
                "candidate": "A",
                "weight": 50
            },
            {
                "candidate": "B",
                "weight": 30
            },
            {
                "candidate": "C",
                "weight": 80
            }
        ]
    },
    {
        "voter": "5",
        "preferences": [
            {
                "candidate": "A",
                "weight": 10
            },
            {
                "candidate": "B",
                "weight": −10
            },
            {
                "candidate": "C",
                "weight": −15
            }
        ]
    },
    {
        "voter": "6",
        "preferences": [
            {
                "candidate": "A",
                "weight": 10
            },
            {
                "candidate": "B",
                "weight": 20
            },
```

```
98                {
99                    "candidate": "C",
100                   "weight": −15
101               }
102           ]
103       }
104   ]
```

B.3 main.py

```
1  import sys
2  import prototype
3
4  file_name ="group.json"
5
6  if len(sys.argv) >1:
7      file_name =sys.argv[1]
8
9  group =prototype.get_group(file_name)
10
11 prototype.aggregate(group)
```

B.4 prototype.py

```
from copy import copy
from voting_rules import *
import json

def get_group(file_name):
    file =open(file_name)
    group =json.load(file)
    print(f"Grupo carregado (ficheiro '{file.name}') \n")
    #print(group)

    for voter in group:
        for preference in voter["preferences"]:
            preference["weight"] /=100

    return group

def order_preferences(group):
    for voter in group:
        voter["preferences"].sort(key =lambda preference : preference["weight"], reverse =True)

    print("Preferências do grupo ordenadas")
    print(json.dumps(group,indent=2), "\n \n")

    return group

def aggregate(input_group):

    group =copy(input_group)

    group =order_preferences(group)

    print(f"Pluralidade: {plurality(group)}")

    print(f"Borda: {scoring_borda_rule(group)}")

    print(f"Veto/Aprovação/Nanson: {scoring_veto_rule(group, cutoff =0)}")

    print(f"Cumulativa: {cumulative(group)}")

    print(f"Média Cumulativa: {cumulative_average(group)}")

    print("\n")

    return ()
```

B.5 voting_rules.py

```
def favorite_candidate(user):
    return user["preferences"][0]["candidate"]

def sort_result(result):
    return sorted(result.items(), key =lambda candidate: candidate[1], reverse =True)

def plurality(group):
    votes =[]

    for user in group:
        votes.append(favorite_candidate(user))

    counting ={}

    for vote in votes:
        counting[vote] =0

    for vote in votes:
        counting[vote] =counting[vote] +1

        #print(f"Candidate {vote} gets 1 score ({counting[vote]} total)")

    return sort_result(counting)

def scoring_borda_rule(group):
    counting ={}

    for preference in group[0]["preferences"]:
        counting[preference["candidate"]] =0

    n =len(group[0]["preferences"])

    for round in range(n):
        for voter in group:
            candidate_id =voter["preferences"][round]["candidate"]
            counting[candidate_id] +=(n - round)

            #print(f"Candidate {candidate_id} gets {n} - {round} +{1} ={n - round} score ({
                counting[candidate_id]} total)")

    return sort_result(counting)

def scoring_veto_rule(group, cutoff):
    counting ={}

    for preference in group[0]["preferences"]:
        counting[preference["candidate"]] =0

```

```
48      for voter in group:
49          for candidate in voter["preferences"]:
50              candidate_id =candidate["candidate"]
51              candidate_weight =candidate["weight"]
52
53              if (candidate_weight >cutoff):
54                  counting[candidate_id] +=1
55                  #print(f"Candidate {candidate_id} gets 1 score ({counting[candidate_id]} total)")
56              else:
57                  #print(f"Candidate {candidate_id} gets 0 score ({counting[candidate_id]} total)")
58                  pass
59
60      return sort_result(counting)
61
62  def cumulative(group):
63      counting ={}
64
65      for preference in group[0]["preferences"]:
66          counting[preference["candidate"]] =0
67
68      for voter in group:
69          for candidate in voter["preferences"]:
70              candidate_id =candidate["candidate"]
71              candidate_weight =candidate["weight"]
72
73              counting[candidate_id] +=candidate_weight
74
75              #print(f"Candidate {candidate_id} gets {candidate_weight} score ({counting[
                    candidate_id]} total)")
76
77      return sort_result(counting)
78
79  def cumulative_average(group):
80      counting_total =cumulative(group)
81
82      counting_average =[]
83
84      for candidate in counting_total:
85          counting_average =counting_average +[(candidate[0], candidate[1]/len(group))]
86          #print(f"Candidate {candidate[0]} gets {counting_average[-1][1]} score")
87
88      return counting_average
```

C Code

C.1 aggregator.rs

```
1  use reqwest::blocking::Response;
2  use serde_json::Value;
3
4  use crate::{structs::*, data_handling::*};
5  use std::{collections::HashMap, collections::hash_map::Entry, error::Error};
6
7  pub fn handle_group(group_id : u64) ->Result<(Vec<String>,Vec<Response>),Box<dyn Error»
8  {
9      let group_data : Value =get_group_data(group_id)?;
10     let group : Group =build_group(&group_data);
11     let aggregated_profile : Profile =aggregate_group(&group);
12
13     let responses : (Vec<String>,Vec<Response>) =return_group_data(aggregated_profile)?;
14
15     Ok(responses)
16 }
17
18 fn aggregate_group( group : &Group ) ->Profile
19 {
20     let mut aggregated_profile : Profile =Profile { id : group.id, weight :
           DEFAULT_PROFILE_WEIGHT, preferences : Vec::new() , identities : Vec::new() };
21
22     aggregate_preferences(group, &mut aggregated_profile);
23     aggregate_characteristics(group, &mut aggregated_profile);
24
25     return aggregated_profile;
26 }
27
28 fn aggregate_preferences( group : &Group, aggregated_profile : &mut Profile) ->()
29 {
30     let group_preference_weights : HashMap<u64, Vec<WeightedPreference» =
           compile_preference_weights(group);
31
32     for (preference_id , preference_weights ) in group_preference_weights
33     {
34         let added_weights : f64 =preference_weights.iter().map(|w| w.profile_weight).sum::<u64>() as
               f64;
35         let added_values : f64 =preference_weights.iter().map(|w| w.preference_value *w.
               preference_weight).sum();
36         let preference =Preference { id : preference_id , value : added_values/added_weights };
37         aggregated_profile.preferences.push(preference);
38     }
39 }
40
41 fn aggregate_characteristics( group : &Group, aggregated_profile : &mut Profile) ->()
42 {
```

```
43      let group_identity_values : HashMap<u64, Vec<String» =compile_identity_values(group);
44
45      for (characteristic_id , characteristic_values ) in group_identity_values
46      {
47          for characteristic_value in characteristic_values
48          {
49              let identity =Identity { id : characteristic_id , value : characteristic_value };
50              aggregated_profile.identities.push(identity);
51          }
52      }
53  }
54
55  fn compile_preference_weights(group : &Group) →HashMap<u64, Vec<WeightedPreference»
56  {
57      let mut preference_weights : HashMap<u64, Vec<WeightedPreference» =HashMap::new();
58
59      for profile in &group.profiles
60      {
61          for preference in &profile.preferences
62          {
63              let weighted_preference =WeightedPreference{profile_weight : profile.weight,
                    preference_value : preference.value};
64
65              match preference_weights.entry(preference.id) {
66                  Entry::Vacant(weights) ⇒{ weights.insert(vec![weighted_preference]); },
67                  Entry::Occupied(mut weights) ⇒{ weights.get_mut().push(weighted_preference); }
68              }
69          }
70      }
71
72      return preference_weights;
73  }
74
75  fn compile_identity_values(group : &Group) →HashMap<u64, Vec<String»
76  {
77      let mut identity_values : HashMap<u64, Vec<String» =HashMap::new();
78
79      for profile in &group.profiles
80      {
81          for identity in &profile.identities
82          {
83              match identity_values.entry(identity.id) {
84                  Entry::Vacant(values) ⇒{ values.insert(vec![identity.value.to_owned()]); },
85                  Entry::Occupied(mut values) ⇒{ values.get_mut().push(identity.value.to_owned()); }
86              }
87          }
88      }
89
90      return identity_values;
```

56

91 }

C.2 data_error.rs

```
1
2  use std::error::Error;
3  use std::fmt::Display;
4
5  #[derive(Debug, Clone)]
6  pub struct EmptyDataError;
7
8  impl Display for EmptyDataError
9  {
10     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) →std::fmt::Result {
11         write!(f, "Error: API request returned empty data! Does the given ID belong to a group?")
12     }
13 }
14
15 impl Error for EmptyDataError {
16     fn source(&self) →Option<&(dyn Error +'static)> {
17         Some(&EmptyDataError)
18     }
19 }
20
21 #[derive(Debug, Clone)]
22 pub struct ApiRequestError;
23
24 impl Error for ApiRequestError {
25     fn source(&self) →Option<&(dyn Error +'static)> {
26         Some(&ApiRequestError)
27     }
28 }
29
30 impl Display for ApiRequestError
31 {
32     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) →std::fmt::Result {
33         write!(f, "Error: API request returned error code! Does the given ID exist?")
34     }
35 }
```

C.3 data_handling.rs

```
1  use std::error::Error;
2  use crate::{structs::*, data_error::*};
3  use serde_json::Value;
4  use reqwest::{self, blocking::*, header::{CONTENT_TYPE, HeaderMap}, Method};
5
6  pub const DEFAULT_PROFILE_WEIGHT : u64 =1;
7
8  const CUSTOMER_TYPE_ID : u64 =5003;
9  const CUSTOMER_TYPE : &str ="Customer";
10 const PREFERENCE_TYPE_ID : u64 =5056;
11 const PREFERENCE_TYPE : &str ="Prefers";
12 const INTEREST_TYPE_ID : u64 =5000;
13 const INTEREST_TYPE : &str ="Interest";
14 const IDENTITY_TYPE_ID : u64 =5057;
15 const IDENTITY_TYPE : &str ="Is";
16 const CHARACTERISTIC_TYPE_ID : u64 =5004;
17 const CHARACTERISTIC_TYPE : &str ="Characteristic";
18
19 const GET_GROUP_CUSTOMERS_AND_INTERESTS_TEMPLATE_ID : u64 =20;
20 const GET_CUSTOMER_AND_CHARACTERISTICS_TEMPLATE_ID : u64 =51;
21 const GET_CUSTOMER_AND_INTERESTS_TEMPLATE_ID : u64 =14;
22 const POPULATE_CUSTOMER_AND_CHARACTERISTICS_TEMPLATE_ID : u64 =51;
23 const POPULATE_CUSTOMER_AND_INTERESTS_TEMPLATE_ID : u64 =14;
24
25 fn send_http_request(method : Method, url : String, body : String) →Result<Response,Box<dyn Error
       »
26 {
27     let client : Client =Client::new();
28
29     let mut headers =HeaderMap::new();
30     headers.insert(CONTENT_TYPE, "application/json".parse().unwrap());
31     let response : Response =client.request(method, &url).headers(headers).body(body).send()?;
32
33     match response.status().is_success()
34     {
35         true ⇒Ok(response),
36         false ⇒Err( Box::new(ApiRequestError{}) )
37     }
38 }
39
40 pub fn get_group_data(root_node_id : u64) →Result<Value,Box<dyn Error»
41 {
42     let url : String =format!(".../Template/Main/get?templateId={
           GET_GROUP_CUSTOMERS_AND_INTERESTS_TEMPLATE_ID}&rootNodeId={
           root_node_id}");
43     let response : Response =send_http_request(Method::GET, url, String::new())?;
44
45     let body : &Vec<u8> =&response.bytes().unwrap().to_vec();
```

```
46      let data : String =String::from_utf8_lossy(&body).to_string();
47      let json : Value =serde_json::from_str(&data)?;
48
49      match json["nodes"][0]["existingNodeId"].as_u64()
50      {
51          None ⇒Err( Box::new(EmptyDataError{})),
52          Some(_) ⇒Ok(json),
53      }
54  }
55
56  pub fn return_group_data(group : Profile) →Result<(Vec<String>,Vec<Response>),Box<dyn Error»
57  {
58      let mut preference_wrapper_map : serde_json::Map<String, Value> =serde_json::Map::new();
59      preference_wrapper_map.insert("nodes".to_owned(), Value::Array(vec![group.
            as_json_with_preferences()]));
60
61      let preference_output : String =serde_json::to_string_pretty(&Value::Object(
            preference_wrapper_map)).unwrap();
62      let preference_url : String =format!(".../Template/Main/populate?templateId={
            POPULATE_CUSTOMER_AND_INTERESTS_TEMPLATE_ID}");
63      let response_preferences : Response =send_http_request(Method::POST, preference_url.clone(),
            preference_output.clone())?;
64
65      let mut characteristic_wrapper_map : serde_json::Map<String, Value> =serde_json::Map::new();
66      characteristic_wrapper_map.insert("nodes".to_owned(), Value::Array(vec![group.
            as_json_with_identities()]));
67
68      let characteristic_output : String =serde_json::to_string_pretty(&Value::Object(
            characteristic_wrapper_map)).unwrap();
69      let characteristic_url : String =format!(".../Template/Main/populate?templateId={
            POPULATE_CUSTOMER_AND_CHARACTERISTICS_TEMPLATE_ID}");
70      let response_characteristics : Response =send_http_request(Method::POST, characteristic_url.
            clone(), characteristic_output.clone())?;
71
72      Ok((vec!{preference_output,characteristic_output},vec!{response_preferences,
            response_characteristics}))
73  }
74
75  pub fn set_veto(user_id : u64, interest_id : u64, duration : u64) →Result<(Veto,Response),Box<dyn
        Error»
76  {
77      let old_preference =get_user_preference_value(user_id, interest_id)?;
78
79      let old_preference_value =match old_preference
80      {
81          Some(preference) ⇒preference.value,
82          None ⇒0.0
83      };
84
```

```
85      let veto : Veto =Veto { user_id, interest_id, preference_value : old_preference_value, duration };
86      let response =set_preference_value(user_id, interest_id, −1.0)?;
87
88      Ok((veto,response))
89  }
90
91  pub fn set_preference_value(user_id : u64, interest_id : u64, preference_value : f64) →Result<
        Response,Box<dyn Error»
92  {
93      let preference : Preference =Preference { id : interest_id, value : preference_value };
94
95      let profile : Profile =Profile { id : user_id, weight : DEFAULT_PROFILE_WEIGHT, preferences :
            vec![preference], identities : vec![]};
96
97      let mut wrapper_map : serde_json::Map<String, Value> =serde_json::Map::new();
98      wrapper_map.insert("nodes".to_owned(), Value::Array(vec![profile.as_json_with_preferences()]));
99
100     let output : String =serde_json::to_string_pretty(&Value::Object(wrapper_map))?;
101     let url : String =format!(".../Template/Main/populate?templateId={
            POPULATE_CUSTOMER_AND_INTERESTS_TEMPLATE_ID}");
102     let response : Response =send_http_request(Method::POST, url.clone(), output)?;
103
104     Ok(response)
105 }
106
107 pub fn get_user_preference_value(user_id : u64, interest_id : u64) →Result<Option<Preference>,
        Box<dyn Error»
108 {
109     let user_preference_data : Value =get_user_preference_data(user_id)?;
110
111     let user_profile : Profile =build_profile(&user_preference_data);
112
113     for preference in user_profile.preferences
114     {
115         if preference.id ==interest_id
116         {
117             return Ok(Some(preference));
118         }
119     }
120
121     Ok(None)
122 }
123
124 fn get_user_preference_data(root_node_id : u64) →Result<Value,Box<dyn Error»
125 {
126     let url : String =format!(".../Template/Main/get?templateId={
            GET_CUSTOMER_AND_INTERESTS_TEMPLATE_ID}&rootNodeId={root_node_id}")
            ;
127     let response : Response =send_http_request(Method::GET, url, String::new())?;
```

```
128
129     let body : &Vec<u8> =&response.bytes().unwrap().to_vec();
130     let data : String =String::from_utf8_lossy(&body).to_string();
131     let full_json : Value =serde_json::from_str(&data)?;
132
133     let user_exists =full_json["nodes"][0]["existingNodeId"].as_u64().is_some();
134
135     match user_exists
136     {
137         false =>Err(Box::new(EmptyDataError{})),
138         true =>
139         {
140             let user_json =full_json["nodes"][0].clone();
141             Ok(user_json)
142         },
143     }
144 }
145
146 fn get_characteristic_data(root_node_id : u64) →Result<Value,Box<dyn Error»
147 {
148     let url : String =format!("…/Template/Main/get?templateId={
            GET_CUSTOMER_AND_CHARACTERISTICS_TEMPLATE_ID}&rootNodeId={
            root_node_id}");
149     let response : Response =send_http_request(Method::GET, url, String::new())?;
150
151     let body : &Vec<u8> =&response.bytes().unwrap().to_vec();
152     let data : String =String::from_utf8_lossy(&body).to_string();
153     let json : Value =serde_json::from_str(&data)?;
154
155     match json["nodes"][0]["existingNodeId"].as_u64()
156     {
157         None =>Err(Box::new(EmptyDataError{})),
158         Some(_) =>Ok(json),
159     }
160 }
161
162 impl Preference
163 {
164     pub fn as_json(&self) →Value
165     {
166         let mut interest_json : serde_json::Map<String,Value> =serde_json::Map::new();
167         interest_json.insert("existingNodeId".to_owned(), Value::Number(self.id.into()));
168         interest_json.insert("type".to_owned(), Value::String(String::from(INTEREST_TYPE)));
169         interest_json.insert("typeId".to_owned(), Value::Number(INTEREST_TYPE_ID.into()));
170
171         let mut preference_properties_json : serde_json::Map<String,Value> =serde_json::Map::new()
                ;
172         preference_properties_json.insert("ExpValue".to_owned(), Value::String(self.value.to_string()))
                ;
```

```
173
174             let mut preference_json : serde_json::Map<String,Value> =serde_json::Map::new();
175             preference_json.insert("type".to_owned(), Value::String(String::from(PREFERENCE_TYPE))
                    );
176             preference_json.insert("typeId".to_owned(), Value::Number(PREFERENCE_TYPE_ID.into()
                    ));
177             preference_json.insert("properties".to_owned(), Value::Object(preference_properties_json));
178             preference_json.insert("node".to_owned(), Value::Object(interest_json));
179
180             return Value::Object(preference_json);
181         }
182     }
183
184     impl Identity
185     {
186         pub fn as_json(&self) →Value
187         {
188             let mut characteristic_json : serde_json::Map<String,Value> =serde_json::Map::new();
189             characteristic_json.insert("existingNodeId".to_owned(), Value::Number(self.id.into()));
190             characteristic_json.insert("type".to_owned(), Value::String(String::from(
                        CHARACTERISTIC_TYPE)));
191             characteristic_json.insert("typeId".to_owned(), Value::Number(
                        CHARACTERISTIC_TYPE_ID.into()));
192
193             let mut identity_properties_json : serde_json::Map<String,Value> =serde_json::Map::new();
194             identity_properties_json.insert("CharValue".to_owned(), Value::String(self.value.to_string()));
195
196             let mut identity_json : serde_json::Map<String,Value> =serde_json::Map::new();
197             identity_json.insert("type".to_owned(), Value::String(String::from(IDENTITY_TYPE)));
198             identity_json.insert("typeId".to_owned(), Value::Number(IDENTITY_TYPE_ID.into()));
199             identity_json.insert("properties".to_owned(), Value::Object(identity_properties_json));
200             identity_json.insert("node".to_owned(), Value::Object(characteristic_json));
201
202             return Value::Object(identity_json);
203         }
204     }
205
206     impl Profile
207     {
208         pub fn as_json_with_preferences(&self) →Value
209         {
210             let preference_arches_json : Vec<Value> =self.preferences.iter().map( |e| { e.as_json() }).
                    collect();
211
212             let mut profile_json : serde_json::Map<String,Value> =serde_json::Map::new();
213             profile_json.insert("existingNodeId".to_owned(), Value::Number(self.id.into()));
214             profile_json.insert("type".to_owned(), Value::String(String::from(CUSTOMER_TYPE)));
215             profile_json.insert("typeId".to_owned(), Value::Number(CUSTOMER_TYPE_ID.into()));
216             profile_json.insert("arches".to_owned(), Value::Array(preference_arches_json));
```

```
217
218             return Value::Object(profile_json);
219         }
220
221         pub fn as_json_with_identities(&self) →Value
222         {
223             let identity_arches_json : Vec<Value> =self.identities.iter().map( |e| { e.as_json() }).collect();
224
225             let mut profile_json : serde_json::Map<String,Value> =serde_json::Map::new();
226             profile_json.insert("existingNodeId".to_owned(), Value::Number(self.id.into()));
227             profile_json.insert("type".to_owned(), Value::String(String::from(CUSTOMER_TYPE)));
228             profile_json.insert("typeId".to_owned(), Value::Number(CUSTOMER_TYPE_ID.into()));
229             profile_json.insert("arches".to_owned(), Value::Array(identity_arches_json));
230
231             return Value::Object(profile_json);
232         }
233     }
234
235     pub fn build_group(group_data : &Value) →Group
236     {
237         let group_id : u64 =group_data["nodes"][0]["existingNodeId"].as_u64().unwrap();
238         let group_profiles : Vec<Profile> =build_profiles(&group_data);
239
240         let group : Group =Group { id: group_id, profiles: group_profiles };
241
242         return group;
243     }
244
245     fn build_profiles(group_data : &Value) →Vec<Profile>
246     {
247         let mut profiles : Vec<Profile> =Vec::new();
248
249         let user_data_array =group_data["nodes"][0]["nodes"].as_array().unwrap();
250
251         for user_data in user_data_array
252         {
253             let profile =build_profile(user_data);
254             profiles.push(profile);
255         }
256
257         return profiles;
258     }
259
260     fn build_profile(customer_data : &Value) →Profile
261     {
262         let preferences : Vec<Preference> =build_preferences(&customer_data["arches"]);
263
264         let identities : Vec<Identity> =match get_characteristic_data(customer_data["existingNodeId"].
                as_u64().unwrap())
```

```
265     {
266         Ok(customer_characteristics_data) ⇒build_identities(&customer_characteristics_data["nodes
                "][0]["arches"]),
267         Err(_) ⇒Vec::new(),
268     };
269
270     let profile : Profile =Profile { id: customer_data["existingNodeId"].as_u64().unwrap(), weight:
            DEFAULT_PROFILE_WEIGHT ,preferences : preferences , identities : identities };
271
272     return profile;
273 }
274
275 fn build_preferences(customer_arches_data : &Value) →Vec<Preference>
276 {
277     let mut preferences : Vec<Preference> =Vec::new();
278
279     for arch in customer_arches_data.as_array().unwrap()
280     {
281         if arch["typeId"].as_u64().unwrap() ==PREFERENCE_TYPE_ID
282         {
283             let preference =build_preference(arch);
284             preferences.push(preference);
285         }
286     }
287
288     return preferences;
289 }
290
291 fn build_preference(arch_data : &Value) →Preference
292 {
293     let preference_id_json =&arch_data["node"]["existingNodeId"];
294
295     let preference_weight_json =match arch_data["properties"].get("ExpValue")
296     {
297         Some(value) ⇒value,
298         None ⇒&arch_data["properties"]["ImpValue"],
299     };
300
301     let preference_id =preference_id_json
302         ∘as_u64()
303         ∘unwrap();
304
305     let preference_weight =preference_weight_json
306         ∘as_str()
307         ∘unwrap()
308         ∘trim_end()
309         ∘replace("λ"", "")
310         ∘parse::<f64>()
311         ∘unwrap();
```

```
312
313     let preference : Preference =Preference { id : preference_id, value : preference_weight };
314
315     return preference;
316 }
317
318 fn build_identities(customer_arches_data : &Value) →Vec<Identity>
319 {
320     match customer_arches_data.as_array()
321     {
322         Some(arches) ⇒
323         {
324             let mut identities : Vec<Identity> =Vec::new();
325             for arch in arches
326             {
327                 if arch["typeId"].as_u64().unwrap() ==IDENTITY_TYPE_ID
328                 {
329                     let preference =build_identity(arch);
330                     identities.push(preference);
331                 }
332             }
333             identities
334         },
335         None ⇒Vec::new(),
336     }
337 }
338
339 fn build_identity(arch_data : &Value) →Identity
340 {
341     let characteristic_id_json =&arch_data["node"]["existingNodeId"];
342     let characteristic_value_json =&arch_data["properties"]["CharValue"];
343
344     let characteristic_value =characteristic_value_json
345    ∘as_str()
346    ∘unwrap()
347    ∘trim_end()
348    ∘replace("λ"", "");
349
350     let identity : Identity =Identity { id : characteristic_id_json.as_u64().unwrap(), value :
          characteristic_value };
351
352     return identity;
353 }
```

C.4 feedback.rs

```
use reqwest::blocking::Response;

use crate::{data_handling::*, structs::Veto};
use core::time;
use std::{thread, error::Error, time::{UNIX_EPOCH, SystemTime}, fs::OpenOptions, io::{Write,
    BufReader, BufRead}};

const VETO_FILE_NAME : &str ="veto_cache.txt";

pub fn handle_feedback(user_id : u64, interest_id : u64, duration : u64) ->Result<(Veto,Response),Box
    <dyn Error>>
{
    let response =set_veto(user_id, interest_id, duration)?;
    let veto =response.0.clone();
    if veto.duration >0
    {
        log_veto(veto);
    }

    Ok(response)
}

pub fn log_veto(veto : Veto) ->()
{
    let mut file =OpenOptions::new()
        .write(true)
        .append(true)
        .open(VETO_FILE_NAME).unwrap();

    let now =SystemTime::now();
    let veto_limit_seconds =now.duration_since(UNIX_EPOCH).unwrap().as_secs() +veto.duration
        *60*60;

    file.write_all(
        format!(
            "{},{},{},{}\n",
            veto.user_id, veto.interest_id, veto.preference_value, veto_limit_seconds
        )
        .as_bytes()
    ).unwrap();
}

pub fn unlog_veto(mut line_index : u8) ->()
{
    let mut initial_file =OpenOptions::new()
        .read(true)
        .open(VETO_FILE_NAME).unwrap();

```

```
46      let lines =BufReader::new(initial_file).lines();
47
48      for line in lines
49      {
50          if line_index != 0
51          {
52              let mut file =OpenOptions::new()
53             ∘write(true)
54             ∘open(VETO_FILE_NAME).unwrap();
55
56              file.write(line.unwrap().as_bytes());
57
58              line_index −= 1;
59          }
60          else {
61
62          }
63      }
64
65  }
66
67  fn check_vetoes() →Result<(),Box<dyn Error»
68  {
69      let file =OpenOptions::new()
70         ∘read(true)
71         ∘open(VETO_FILE_NAME)
72         ∘unwrap();
73
74      for line in BufReader::new(file).lines()
75      {
76
77          let line_string =line?;
78
79          if !line_string.is_empty()
80          {
81              let line_parts : Vec<&str> =line_string.split(',').collect();
82
83              let expiration_date =line_parts[3].parse::<u64>()?;
84              let now_date =SystemTime::now().duration_since(UNIX_EPOCH)?.as_secs();
85
86              let user_id =line_parts[0].parse::<u64>()?;
87              let interest_id =line_parts[1].parse::<u64>()?;
88              let preference_value =line_parts[2].parse::<f64>()?;
89
90              if get_user_preference_value(user_id, interest_id).unwrap().unwrap().value ==−1.0
91              {
92                  if now_date >expiration_date
93                  {
```

```
94                          println!("Veto expired; renewing old preference value ({preference_value}) of user {
                                user_id} for interest {interest_id}");
95                          set_preference_value(user_id, interest_id, preference_value)?;
96                      }
97                  }
98              }
99          };
100
101         Ok(())
102     }
103
104     pub fn veto_validation() →()
105     {
106         loop
107         {
108             thread::sleep(time::Duration::from_secs(10));
109             let _ =check_vetoes();
110         }
111     }
112
113     pub fn launch_veto_validation() →()
114     {
115         thread::spawn(veto_validation);
116     }
```

C.5 logging.rs

```
1  use chrono::{DateTime, Utc};
2  use std::{error::Error, fs::OpenOptions, io::{Write}};
3
4  const LOG_FILE_NAME : &str ="logs.txt";
5
6  pub fn log(message : String) →Result<(), Box<dyn Error»
7  {
8
9      let mut file =OpenOptions::new()
10         ∘write(true)
11         ∘append(true)
12         ∘open(LOG_FILE_NAME)?;
13
14     let now: DateTime<Utc> =Utc::now();
15
16     let log_message =format!(
17          "#{} »{}λn",
18          now, message
19     );
20
21     print!("{log_message}");
22
23     file.write_all(
24          log_message
25         ∘as_bytes()
26     )?;
27
28     Ok(())
29 }
```

C.6 main.rs

```
1  #![feature(proc_macro_hygiene, decl_macro)]
2
3  mod structs;
4  mod data_handling;
5  mod aggregator;
6  mod server;
7  mod data_error;
8  mod feedback;
9  mod logging;
10
11 use std::env;
12 use server::launch_server;
13 use feedback::launch_veto_validation;
14
15 fn main()
16 {
17     let args: Vec<String> =env::args().collect();
18
19     launch_veto_validation();
20
21     launch_server(args);
22 }
```

C.7 server.rs

```
use rocket::{*, config::*};
use rocket_okapi::{openapi, routes_with_openapi};
use rocket_contrib::json::JsonValue;
use rocket_okapi::swagger_ui::{make_swagger_ui, SwaggerUIConfig};
use serde_json::json;

use crate::aggregator::handle_group;
use crate::feedback::handle_feedback;
use crate::logging::log;

#[derive(serde::Serialize)]
struct Response {
    reply: String,
}

#[openapi]
#[get("/<group_id>")]
fn aggregation_request_handler(group_id : u64) ->JsonValue {
    match handle_group(group_id)
    {
        Ok(_) =>
        {
            let _ =log(format!("Aggregated preferences of group {group_id}."));
            JsonValue(json!(Response{reply : format!("SUCCESSFULY AGGREGATED GROUP {
                group_id}")}))
        }
        Err(e) =>JsonValue(json!(Response{reply : format!("FAILURE =>{e}")}))
    }
}

#[openapi]
#[get("/veto/<user_id>/<interest_id>/<veto_length>")]
fn feedback_request_handler(user_id : u64, interest_id : u64 , veto_length : u64) ->JsonValue {
    match handle_feedback(user_id, interest_id,veto_length)
    {
        Ok(_) =>
        {
            let _ =log(format!("Registered veto of interest {interest_id} by user {user_id} (length: {
                veto_length} hour(s))."));
            JsonValue(json!(Response{reply : format!("SUCCESSFULLY VETOED INTEREST {
                interest_id} FOR USER {user_id} (DURATION: {veto_length} HOUR/S)")}))
        }
        Err(e) =>JsonValue(json!(Response{reply : format!("FAILURE =>{e}")}))
    }
}

fn get_docs() ->SwaggerUIConfig {
    use rocket_okapi::swagger_ui::UrlObject;
```

```
46
47      SwaggerUIConfig {
48          url: "/aggregation/openapi.json".to_string(),
49          urls: vec![UrlObject::new("Aggregation", "/aggregation/openapi.json"),UrlObject::new("
                Feedback", "/feedback/openapi.json")],
50          ..Default::default()
51      }
52  }
53
54  pub fn launch_server( args : Vec<String> ) ->()
55  {
56      let port : u16;
57
58      if args.len() >1
59      {
60          port =match args[1].parse::<u16>()
61          {
62              Ok(value) =>value,
63              Err(_) =>{ println!("Error while parsing the port value."); return;},
64          };
65      }
66      else {
67          port =8080;
68          println!("» Port unspecified; using {port}.λn");
69      }
70
71      let config : Config =match Config::build(Environment::Development)
72      .port(port)
73      .finalize()
74      {
75          Ok(value) =>value,
76          Err(error) =>{ println!("{error:?}"); return;},
77      };
78
79      rocket::custom(config)
80          .mount("/aggregation", routes_with_openapi![aggregation_request_handler])
81          .mount("/feedback", routes_with_openapi![feedback_request_handler])
82          .mount("/swagger", make_swagger_ui(&get_docs()))
83          .mount("/", make_swagger_ui(&get_docs()))
84          .launch();
85  }
```

C.8 structs.rs

```
1  pub struct Preference
2  {
3      pub id : u64,
4      pub value : f64
5  }
6
7  pub struct Identity
8  {
9      pub id : u64,
10     pub value : String
11 }
12
13 pub struct Profile
14 {
15     pub id : u64,
16     pub weight : u64,
17     pub preferences : Vec<Preference>,
18     pub identities : Vec<Identity>
19 }
20
21 pub struct Group
22 {
23     pub id : u64,
24     pub profiles : Vec<Profile>
25 }
26
27 pub struct WeightedPreference
28 {
29     pub profile_weight : u64,
30     pub preference_value : f64
31 }
32 #[derive(Clone)]
33 pub struct Veto
34 {
35     pub user_id : u64,
36     pub interest_id : u64,
37     pub preference_value : f64,
38     pub duration : u64
39 }
```

D Division of work