FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Ein generisches Framework für Entscheidungsunterstützende Systeme im medizinischen Bereich

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Daniel Gepp, B.Sc.
Matrikelnummer 01228976

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gernot Salzer

Wien, 1. Jänner 2001

Daniel Gepp   Gernot Salzer

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

A Generic Framework for Medical Decision Support Systems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Daniel Gepp, B.Sc.
Registration Number 01228976

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gernot Salzer

Vienna, 1st January, 2001

Daniel Gepp	Gernot Salzer

Erklärung zur Verfassung der Arbeit

Daniel Gepp, B.Sc.
Naglern 19, 2113 Karnabrunn

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Jänner 2001

Daniel Gepp

Acknowledgements

At first I would like to thank my parents for supporting me in every possible way during my studies.

Furthermore, I would like to thank my advisor, Gernot Salzer, for the possibility of contributing to his research through this thesis. Your advices on numerous occasions were very important for me.

Kurzfassung

Diese Arbeit zielt darauf ab einen Prototypen eines generisches Framework für Entscheidungsunterstützende Systeme im medizinischen Bereich zu implementieren. Für die ausgewählte medizinische Domäne erstellt das Framework ein Backend, welches eine REST Schnittstelle zur Verfügung stellt, und ein einfaches Frontend, welches die Eingabe von Informationen von Patienten, bzw. Krankheiten ermöglicht. Da das Framework von Personen ohne professionellem Hintergrund in Informatik verwendet werden soll, wird eine abstrakte Methode spezifiziert um Merkmale und Krankheiten abzubilden.

Für das Ranking von Krankheiten werden zwei Ansätze präsentiert und implementiert: ein probabilistischer Ansatz und ein Ansatz, welcher lose auf der Idee von term frequency-inverse document frequency (TFIDF)-Ähnlichkeit basiert. Ein System, welches mit dem vorgestellten Framework erstellt wird, wird mit dem bestehenden, Entscheidungsunterstützende System Dermtrainer verglichen, welches für dermatologische Erkrankungen optimiert ist.

Das resultierende Framework basiert auf einem yeoman Codegenerator. Dieser Generator erstellt ein Spring Boot backend und ein Vue.js frontend entsprechend den Eingaben des Benutzers. Für die Repräsentation der Merkmale entschieden wir uns gegen eine benutzerdefinierte domain specific language (DSL) zugunsten einer einfachen JSON-Struktur mit dazugehörigen JSON Schemas für die Validierung von Nutzereingaben. Krankheiten werden mit Hilfe von Spreadsheets abgebildet und dem System übermittelt.

Um die Korrektheit des Frameworks zu zeigen, konstruieren wir ein System, welches auf der Domäne von Dermtrainer basiert. Wir vergleichen die gestellten Diagnosen beider Systeme mit Patienten- und Krankheitsdaten, welche im originalen Dermtrainer-Projekt erstellt wurden.

Die Ergebnisse zeigen, dass ein System, welches mit dem vorgestellten Frameworks erstellt wurde, vergleichbare Qualität in puncto Ranking der Krankheiten, wie Dermtrainer erreicht.

Obwohl das vorgestellte Framework von einem Laien im Bereich der Informatik alleine nicht realistisch bedienbar ist, können wir einen Anwendungsfall präsentieren, in welchem die Verwendung des Frameworks durch einen Laien realistisch ist.

Abstract

This thesis aims to implement a prototype of a framework for medical decision support systems. For the specified medical domain, the framework will generate both the backend providing REST functionality and a basic frontend that provides simple means of input for patient information. Since the framework is to be used by persons without a background in computer science, we will specify an abstract method to characterize features and diseases.

For the ranking of diseases, two generic approaches will be presented and implemented: a probabilistic approach and an approach loosely based on the idea of term frequency-inverse document frequency (TFIDF)-similarity. Finally, a system built with the framework will be compared to Dermtrainer, which is an optimized dermatology decision support system.

The resulting framework is built as a yeoman code-generator. This generator builds a Spring Boot backend and a Vue.js frontend according to the input of the user. For representing features we decided against a custom DSL in favor of a simple JSON structure alongside with JSON Schemas for validating user input. Diseases are represented as spreadsheets.

In order to demonstrate the correctness of the framework, we construct a system based on the domain of Dermtrainer with the framework. We compare the diagnoses made of the systems with patient and disease data from the original Dermtrainer project.

The results show that a system constructed with the proposed framework achieves comparable overall quality when compared to Dermtrainer.

Although, the implemented framework is not usable by a layperson in the field of computer sciences, we can show a use case in which a layperson can realisticly use the framework, when the initial setup is already done.

Contents

CHAPTER 1

Introduction

Medical Expert Systems provide support for general practitioners in helping them diagnose rare illnesses of medical fields that they are not specialized in. While there is ongoing research and projects on Medical Expert Systems for specialized fields, there is little research aiming at frameworks for decision support systems.

In a recently finished project funded by FFG the “Theory and Logic” group of the Technical University of Vienna developed a decision support system for dermatological diseases. Analysing the final system it seems to be possible to generalize the reasoning component as well as the user interface to arbitrary medical fields, provided the data can be organized in a similar way.

The aim of this thesis is to understand the conditions under which it is possible to use a similar approach and to develop a generic framework for such medical decision support systems. This framework then can be used to generate specialized decision support systems like the dermatological one.

One particular problem of generic frameworks is what sometimes is called the knowledge acquisition bottleneck: They require expertise in computer science as well as in the application domain. For this reason we intend to define an interface that allows medical processionals to set up a decision support system without extensive knowledge in computer science. The overall aim of this thesis is to examine under which conditions a decision support system derived from a generic framework can compete with a system developed for specifically for the domain.

1.1 Aim of the Work

The aim of this work is to implement a framework for medical decision support systems. For the specified medical domain, the framework will generate both the server providing REST functionality and a basic frontend that provides simple means of input for patient information. Since the framework is to be used by persons without a background in computer science, we will develop an appropriate user interface for configuring the framework, like a domain specific language. For ranking the diseases, two generic approaches will be presented and implemented: a probabilistic approach and an approach loosely based on the idea of TFIDF-similarity.

The following research questions will be addressed:

- How can we specify the features and characteristics of diseases in an abstract way?
- Is it possible to generate a specific decision support system from a generic framework that yields results comparable to an existing system that is optimized for diagnosing a specific medical field (dermatology in our case)?

1.2 Methodological Approach

1. Literature Review: background information build the theoretical basis for this thesis.
2. Definition of Disease metadata and Disease Format
3. Implementation of the Framework with an eye on user-friendliness.
4. Evaluation of the framework
   - Evaluation of correctness: Comparison of a medical decision support system created by the framework with an already existing system for dermatology that has been specifically constructed and tuned.
   - Evaluation of usability: A proper evaluation would have to involve physicians not scared by computers. While we know some from prior projects, it is unrealistic to assume that they will devote their time to this project. Therefore we will analyse the proposed interface for configuring the framework regarding the required skills. It is likely that this interface will have to be adapted in future projects.

CHAPTER 2

Related Work

Although this thesis describes a new framework for decision support systems, due to the conceptual similarity of decision support systems to recommender systems, we present related work to both types of system and frameworks.

Furthermore, we describe the Dermtrainer project that this thesis is partially based on.

2.1 Decision Support Systems

A decision support system is a information system used to provide support in the process of making decisions for the user. The main goal of decision support systems is to make knowledge of specialists, from a specific domain, available to non-specialists. Components of a decision support system comprise a knowledge base, an inference engine, a user interface and tools for knowledge acquisition [1].

State of the Art

We could not find recent research regarding medical decision support system frameworks aiming at non-technical users. However, work can be found that focuses on the inference component of medical decision support systems.

There is a framework using weighted scoring and the first inference, then aggregate (FITA) method for inferencing. Additionally it uses a neural network for learning, which is decoupled from the decision support system that aims at helping to identify similar diagnoses. The system takes patient symptoms, history and laboratory data as input. The framework is explored theoretically and an exemplary patient data flow is given. It comprises five parts: initial gathering of subjective information, generation of probable diagnoses, gathering objective evidence, evaluating the hypothesis, confirmation of the disease [2].

Abudahab et al. [3] propose a rule based framework that is constructed so that physicians are able to model the rule and knowledge base. Additionally, the system supports inferencing based on this data. The framework itself provides no diagnosing of patients.

The *Generic Medical Fuzzy Expert System for Diagnosis of Cardiac Diseases* proposed by Sikchi et al. [4] uses a fuzzy rule system. Besides patient symptoms, this system also takes laboratory parameters into consideration, when diagnosing a patient. In the study, the framework was tested in the domain of cardiac diseases. The system is based on Matlab Fuzzy Logic Toolbox.

Much more work can be found regarding specialized decision support systems. Recent research does not seem to focus on probabilistic or term frequency-inverse document frequency (TFIDF) based classifiers, but rather fuzzy logic [1, 5, 6, 7], machine learning [8, 9, 10] or rule based approaches [11, 12].

Although extensive literature review has been conducted, no recent research for completely generic decision support systems could be found. It seems that developed frameworks need to have a domain for which it can be used in a generic way.

2.2 Recommender Systems

Recommender systems are algorithms and information systems that suggest items to a user, based on the preferences of the user and certain predefined constraints. Recommender systems need to collect preferences from users in order to complete this computational task. Preferences can be classified as explicitly expressed, e.g., ratings given on other items, or as inferred preferences. Inferred preferences may be for instance the repeated visit of a product page [13].

State of the Art

It seems that a lot more research is conducted in the area of recommender system frameworks than in the area of decision support system frameworks.
This could be explained, inter alia, by the fact that recommender systems play an important role in profitable Internet services like Amazon, Netflix and YouTube, with Netflix awarding one million dollar to the team[1] that would develop a system that could beat the recommender system of Netflix [13].

Therefore, research also led to the development of generic recommender system frameworks that are successfully used and further developed outside the academic domain:

MyMediaLite [14] is a framework for developing recommender systems, implemented in C#. It includes libraries for recommendation algorithms and tools for evaluating recommender systems. The frameworks features k-nearest neighbor, simplest baseline methods and matrix factorization for the rating of predictions. Command line tools enable developers to train recommender systems without having to write code.

[1]Which was awarded to team “BellKor’s Pragmatic Chaos” on September 21, 2009. https://www.netflixprize.com/

A more recently developed framework is RankSys [15] which was developed with Java 8. RankSys is specialized in the evaluation and experimentation on recommendation technologies. The framework includes collaborative filtering recommendation algorithms, such as nearest neighbor, matrix factorization and factorization machines, as well as quality metrics and re-ranking algorithms.

Another recent framework is the, also for Java developed, recommendation library LibRec [16]. LibRec features 65 recommendation algorithms and data structures used in the most common tasks for recommender systems.

While the above mentioned frameworks are actively used and under development, there is more recent research in the field of general recommender system frameworks conducted like context-aware [17, 18] or location-aware [19, 20] frameworks.

Since more and more data of the health states of patients, diagnoses and medication is stored in databases, recommender system frameworks are also a field of research in medical computer science.

OrderRex [21] is a recommender system that predicts hospital admission and the outcomes based on the data of the patient that is inserted on the hospital counter. The broad goal of the framework is to be completely integrated in the clinical workflow and make clinical order suggestions based on data in the electronic medical record (EMR) of the patient. For this, EMR data has to be preprocessed, such that clinical data is mapped on time points per patient. Recommendations are built from this timeline using co-occurrence statistics.

A recommendation framework based on neighborhood based collaborative filtering for skin diseases is presented in [22]. Grasser et al. also integrate evidence based exclusion rules, which remove inappropriate recommendations. Although the evaluated data set in this study is settled in the domain of dermatology, the underlying framework is designed to be flexible enough to be used in other medical disciplines.

[23, 24] independently describe frameworks for the recommendation of drugs by general practitioners.

All of these frameworks provide algorithms and tools commonly needed to start the development of a recommender system and therefore mainly focus on professional developers. They are well equipped for experimenting and applying metrics on algorithms, but are not intended to be setup by a person not having a professional computer science background[2].

Additionally, setting up a system with one of the mentioned libraries requires the user to write additional code, which is not the case in the framework presented in this thesis.

[2]An exception here is the framework presented in [25] where an already set up system can be configured by "non-technical users". However, it is not described to what extent the system can be configured.

2.3 Dermtrainer

As the resulting framework partially builds on the findings made during the development of Dermtrainer [26], the project is briefly described in this section.

The goal of the project was to develop a prototype of a dermatology decision support system that assists general practitioners and serves as a training platform for dermatologists in education. As a result of the interdisciplinary nature of the system, the following teams were involved in the development:

- Dermatologists from Medical University of Vienna (MUW) - contributed a systematic way of diagnosing skin diseases, description of the diseases and images of skin lesions. MUW also carried out a clinical study that compared the accuracy of general practitioners diagnosing skin lesions using Dermtrainer vs. a standard encyclopedia.
- Software Engineers from emergentec biodevelopment GmbH - implemented the GUI and the infrastructure for storing disease data and images.
- Computer Scientists from Vienna University of Technology (TUW) - developed and analysed methods for selecting diagnoses based on observations on the patient.

2.3.1 Patient Data

Patient information is entered by the physician via a web interface which provides information about arrangement, localization, morphology and color of the lesions, timing, and additional signs that are observed. The form does not have to be completed, although more information leads to more precise diagnoses.

2.3.2 Disease Data

The disease knowledge base contains 620 diseases, each described by 131 fields relevant for diagnosis.
Due to the uncertainty of the real probability that a symptom occurs with a disease, most of these fields hold one of three values that are estimated by dermatologists:

- “yes” - the symptom can be observed with the disease
- “no” - the symptom cannot be observed with the disease
- “unlikely” - the symptom is unlikely to be observed with the disease

The overall frequency of a disease is an exception with higher precision. The epidemiology of a disease can be classified from “exceptional” to “very common” in six steps.

2.3.3 Ranking of Diseases

The ranking of diseases can be divided into 3 phases:

1. Elimination of diseases where certain symptoms that are observed on the patient, but correspond to a “no” in the disease. This approach can only be used for symptoms that cannot be mistaken. Due to subjective judgement of non-specialists that differs to the dermatologists opinion, certain symptoms are not excluded, but rather penalized. In this case a “no” can be seen as a “more unlikely than unlikely”.
2. Computation of scores that reflect the similarity of diseases to the patient data. The computer scientists of TUW developed two methods: one probabilistic approach and one based on the TFIDF method[3]. Each method results in two metrics per disease.
3. Selection of the diagnoses to be displayed. Diseases are shown in the order of the internal ranking, but no scores are shown to the user. These scores could give a wrong impression of precision.

2.3.4 Evaluation

For the evaluation of the decision support system a total of 422 test cases were created. These cases consist of virtual patients[4] that were diagnosed by residents of MUW and physicians from different disciplines of a hospital in New York. The analysis of these cases led to modifications in the disease database and improvements of the scoring methods.

In the final version, Dermtrainer ranks the correct diagnosis in 94% of the test cases among the displayed diseases. On average seven diseases are shown.

2.3.5 Current status

The rights on Dermtrainer have been sold. A current version of the system is served on `http://www.dermtrainer.com/`.

[3] In the final version of Dermtrainer, only the probabilistic method is used.
[4] A virtual patient comprises images of the lesions and additional information that can be observed.

CHAPTER 3

Methods for Ranking Diseases

One of the biggest challenge for physicians in diagnosing lesions lies within the diagnosis itself. Besides the high grade of subjective judgement of the physician, making a diagnosis almost non-repeatable [27], the fuzziness of medical data and knowledge leads to a grade of uncertainty in diagnoses. Due to this uncertainty and subjectivity of features, a physician can observe within a lesion, methods for diagnosing have to take into account this uncertainty. In this thesis we decided to rely on variations of the methods already used in Dermtrainer. This way we can also draw direct comparisons with the system.

Each of the presented methods results two metrics:

- *similarity*: a measure of similarity of a patient input to a disease that ignores epidemiology
- *rank*: the *similarity* of a disease to a patient input combined with the overall frequency of the disease

Throughout this chapter the notations presented in table 3.1 are used.

Additionally the following helper functions are used in both algorithms:

- Function for checking whether feature m_i belongs to the category c:

$$c_{check}(m_i, cat) = \begin{cases} 1 \text{ for } cat = c(m_i) \\ 0 \text{ otherwise} \end{cases} \tag{3.1}$$

- Function that counts the number of features of a category c given by the patient input $\vec{p}$

$$nf_{patient}(\vec{p}, c) = \sum_{l=0}^{j} c_{check}(m_l, cat) \tag{3.2}$$

D	a disease in the disease knowledge base
$\vec{p} = (m_1 \dots m_j)$	feature vector representing a patient with $j <= k$ observed features
$P(D)$	the overall *frequency* of a disease d
$P(m_i \mid D)$	*likelihood* that a feature m_i occurs given a disease D
$c(m_i)$	function assigning a category c unambiguously to a feature m_i
$w(c)$	function that assigns the weight for each category c. If no weight is specified this function returns 1.

Table 3.1: Base notations used by both methods.

3.1 Probabilistic Approach

The idea behind the probabilistic algorithm is that the probability of a disease after the observation of a number of features can be computed with the probabilities of the features given the disease and the probability of the disease itself.

The base form for this equation is derived from Bayes' theorem:

$$P(D \mid \vec{p}) = \frac{P(\vec{p} \mid D) \cdot P(D)}{P(\vec{p})} \tag{3.3}$$

with

$P(D \mid \vec{p})$	... probability for D after observing $m_1, \dots, m_j$
$P(\vec{p} \mid D)$	... probability of observing $m_1, \dots, m_j$ given disease D
$P(D)$	... probability of disease D
$P(\vec{p})$	... probability of observing $m_1, \dots, m_j$ simultaneously

We assume that all observations m_i are disjoint events[1] which allows us to rewrite $P(D \mid \vec{p})$ as:

$$P(D \mid \vec{p}) = \frac{P(m_1 \mid D) \cdot P(m_2 \mid D) \cdots P(m_j \mid D) \cdot P(D)}{P(m_1) \cdot P(m_2) \cdots P(m_j)} \tag{3.4}$$

Since the probability of observations $P(\vec{p})$ is the same for all diseases, we can ignore it for ranking diseases, which results in $R(D \mid \vec{p})$:

$$R(D \mid \vec{p}) = P(m_1 \mid D) \cdot P(m_2 \mid D) \cdots P(m_j \mid D) \cdot P(D) \tag{3.5}$$

[1] We are aware that medical observations are rarely disjoint events. The here presented equation represents an approximation of the probability of a disease given certain observations. Probabilities of the observations can only be estimated anyway.

n	number of diseases in the knowledge base
$m_{i_{yes}}$	number of diseases d that contain the feature m_i
$idf(m_i) = log(\frac{n}{m_{i_{yes}}})$	IDF of the feature m_i

Table 3.2: Notations used by the method based on TFIDF.

Inserting weights for the respective categories that express the difference in relevance leads to:

$$R(D \mid \vec{p}) = P(m_1 \mid D)^{w(c(m_1))} \cdot P(m_2 \mid D)^{w(c(m_2))} \cdots P(m_j \mid D)^{w(c(m_j))} \cdot P(D) \tag{3.6}$$

To lower the computational effort, this equation only uses the logarithmic form of probabilities. We then add the reciprocal value of $nf_{patient}$ to attenuate the influence of features, based on the number of features of the corresponding category that are observed. This leads to the *rank* being defined as follows:

$$rank_p(\vec{P}, D) = \sum_{i=0}^{j} (log(P(m_1 \mid D)) * \frac{w(c(m_i))}{nf_{patient}(\vec{P}, c(m_i))}) + log(P(D)) \tag{3.7}$$

The probabilistic *similarity* of a patient $\vec{P} = (m_1 \dots m_j)$ is then defined as:

$$sim_p(\vec{P}, D) = \sum_{i=0}^{j} (log(P(m_1 \mid D)) * \frac{w(c(m_i))}{nf_{patient}(\vec{P}, c(m_i))}) \tag{3.8}$$

3.2 Approach based on TFIDF

Basically, the idea behind this algorithm is that a disease, as well as a patient can be seen as documents that contain features instead of terms. Since we should consider that symptoms are not equally meaningful, we use IDF to calculate the weight of a symptom. If a symptom occurs in many diseases, this symptom can be declared as less significant, which results in a lower IDF weight and vice versa. This leads to a feature significance weighted scoring.

3.2.1 TFIDF Foundations and Notations

Additionally to the base notations, the definitions presented in table 3.2 are used in this section.

The basic concept behind the TFIDF method, a method used in information retrieval, is to weight a document *doc* based on the terms it comprises with a search term t_i. The term frequency (TF) is defined as $TF(doc, t_i) = |\text{occurrences of } t_i \text{ in } doc|$, which means the weight of the document is higher, the more often the search term occurs in it.

On the other hand, the inverse document frequency (IDF) is defined as $IDF(t_i) = log(\frac{N}{n_i})$, where N is the overall number of documents and n_i is the number of documents containing the term t_i. This value is higher if the number of occurrences of the search term is low. The intuition behind the inverse document frequency is that a term that occurs in many documents is not a good discriminator and should get less weight than a term that occurs in few documents.

Calculating the TFIDF weight is defined as follows [28]:

$$TFIDF(doc, t_i) = TF(doc, t_i) * IDF(doc, t_i) \tag{3.9}$$

With the TFIDF weight we can determine the weight of a document regarding to a term. To use TFIDF for computing the similarity of two documents (a patient and a disease in this case) we need a metric of similarity. Therefore, the *cosine similarity* can be used. We denote $\vec{V}(doc)$ the vector of a document *doc* with one dimension for every term in the dictionary. These vector components are computed using the TFIDF weight. The cosine similarity of two documents d_1, d_2 is then defined as the cosine of the angle between the two document vectors [29]:

$$cos_{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|} \tag{3.10}$$

Since the algorithm used in this thesis uses a combination of TFIDF and probabilistic elements we need to establish the relationship between the IDF weight and the probability $P(t_i)$ that a term t_i appears in a document [28, 30]:

$$IDF(t_i) = -log(P(t_i)) \tag{3.11}$$

3.2.2 TFIDF based scoring Algorithm

The following definition represents a helper function used for the algorithm, such that the weighting factor is only applied to already positive values:

$$w_f(x, k) = \begin{cases} x * w(k) \text{ for } x > 0 \\ x \text{ otherwise} \end{cases} \tag{3.12}$$

The TFIDF based similarity of a patient $\vec{P} = (m_1 \ldots m_j)$ and a disease D is defined as follows:

$$sim_t(\vec{P}, D) = \sum_{i=0}^{j} \left(w_f \Big((idf(m_i) + log(P(m_1 \mid D))) * \frac{1}{nf_{patient}(\vec{P}, c(m_i))}, c(m_i) \Big) \right) \tag{3.13}$$

Although this method is based on the concept of TFIDF the equation given above shows that it is merely a combination of IDF values and the probabilities of features. More

Metric	**Threshold**
rank	-4
similarity	-8

Table 3.3: Thresholds used for the respective metrics. NOTE: Due to the logarithmic representation of all scores, so are the thresholds.

specifically it is an extension of the algorithm described in section 3.1 adding $idf(m_i)$ and conditional weighting achieved by the helper function w_f.

The *rank* is defined analogously to the *rank* in the probabilistic method:

$$rank_t(\vec{P}, D) = sim_t(\vec{P}, D) + log(P(D)) \qquad (3.14)$$

3.3 Mixing of Rank and Similarity Metrics

During the development of Dermtrainer it has been discovered that the overall quality of the ranking could be improved by ranking diseases with a combination of the *similarity* and *rank* metric, compared to the usage of only one metric. The result of this combination is referred to as *mixed* ranking in the context of this thesis.

A pseudo code representation of the algorithm for mixing the metrics is given by algorithm 3.1. The thresholds used in this algorithm have been determined empirically during the development of Dermtrainer and can be seen in table 3.3.

3.3.1 Algorithm for ranking diseases

Algorithm 17 shows the pseudo code used for ranking diseases that combines the calculation and mixing of metrics. The overall structure is based on Dermtrainer.

The initialization in line 1 loads the disease database and creates structures used for the calculation of metrics, dependent on the type of score used.

The structure of the algorithm comprises two main for-each loops. The first loop checks for every disease profile whether it conflicts with the patient data. A conflict condition is fulfilled when a disease has a “no” as value for a feature that is observed in the patient, which results in the disease being removed from the list of potential diagnoses. Two circumstances can prevent the check of a conflict for a category:

1. The category does not have hard conflicts.
2. The category is in the list of excluded categories of the current disease.

The second loop computes *similiarity* and *rank* metrics for the remaining diseases with the selected scoring method.

Algorithm 3.1: Ranking of diagnoses by mixing *similarity* and *rank* metrics.

```
Data: list of all diagnoses D
Result: ranked list of diagnoses
r := empty diagnose list;
for score s in ['rank', 'similarity'] do
    r_1 := empty diagnose list;
    n := 6;
    t := threshold(s);
    sort D after score s in descending order;
    for diagnose d in D do
        if n <= 0 OR score s of d < t then
            break;
        end
        if score 'similarity' of d >= threshold('similarity') then
            add d to r_1;
        end
        n−−;
        if n = 0 then
            n := MAX of Integer;
            t := score s of d;
        end
    end
    add all diagnoses of r_1 to r;
end
eliminate duplicates in r;
return r;
```

Then, all scores are normalized by determining the maximum *similiarity*, respectively *rank* of all scores and subtracting it from each score[2].

At the end of the algorithm, the top diagnoses are ranked, using mixed ranking.

[2]The maximum is subtracted because all scoring methods are using logarithmic scale.

Algorithm 3.2: Generic algorithm to calculate the ordered diagnoses for a patient input.

```
   Data: Patient input $P$
   Result: A ordered list of disease scores
 1 load disease profile list $D$ from database;
 2 $S \leftarrow$ empty List;
 3 foreach disease profile $d_i \in D$ do
 4     foreach non-empty category $c_i \in P$ do
 5         if $c_i$ has hard conflict and not $d_i(c_i)$ has conflict exclusion then
 6             if $P(c_i)$ conflicts $d_i(c_i)$ then
 7                 remove $d_i$ from $D$;
 8             end
 9         end
10     end
11 end
12 foreach disease profile $d_i \in D$ do
13     calculate $sim_i$ and $rank_i$ of $d_i$ and $P$;
14     add ($d_i$, $sim_i$, $rank_i$) to $S$;
15 end
16 normalize $rank$ and $sim$ of all scores in $S$;
17 return mixScores(S)
```

CHAPTER 4

Feature Definition and Disease Format

For the setup of a medical decision support system, a wide range of categories and disease features, respectively different constraints between them have to be specified.

Based on the requirements that emerged during the development and analysis of Dermtrainer, we specified the information needed in order to generate a flexible and complete framework. This chapter describes the structures used for defining medical data and the main ideas behind it.

4.1 Running Example

For the sake of clarity we construct a small running example of disease feature categories which will be used throughout this thesis[1]: *site, color, amount of lesions, formation, border, age, gender.* The features of these categories can be seen in table 4.1.

Additionally, the following constraints apply to the example:

- *site* has transitive features, e.g. if a lesion is on the *toes* of the patient it is also on the *feet* of the patient.
- A value for the category *formation* may only be given if the a disease has an *amount of lesions* of *many.* This applies to both, disease and patient input.

[1]Due to copyright reasons we cannot include the original feature categories of the disease knowledge base used in Dermtrainer. The values of this example are for explanatory purposes only and do not comprise a working system.

category	**features**
site	head, torso, breast, stomach, arms, hands, fingers, legs, feet, knees, toes
color	red, white, brown, black
amount of lesions	single, few, many
formation	scattered, cumulative
border	delimited, irregular
age	infant, child, adult, elder
gender	male, female

Table 4.1: Category features of the running example.

- *age* is a category without hard conflict, meaning that if the patient input does not match the possible features of a disease, the disease is not excluded, but rather the score of the disease is lowered accordingly.
- The categories *color, formation, amount of lesions, border, age* and *gender* can only yield to a single observation of the patient.
- Values of the categories *color, amount of lesions, formation* and *border* have **likelihoods** that describe the possibility that the features can be observed with a disease. *age* and *site* have a number of possible values that can occur with a disease, there is no explicit probability necessary. *gender* has the occurrence ratio of the values as possible disease input.
- Due to the importance of the location of a lesion for the diagnosis, the partial score of the category *site* should be up-weighted by a factor of 3.0.
- Features can have three different **likelihood** values: “yes”, “improbable” and “no”. They have respective numerical probabilities of 1.0, 0.03 and 0.0005. The value “improbable” can be considered as a “yes” value with lower probability.
- A disease can have an overall **frequency** of “rare” (0.00001), “less common” (0.001), “common” (0.01) or “very common” (0.1).

4.2 Disease Metadata

When we analysed the disease data and the ranking methods of Dermtrainer, we concluded that most parts of the system could be built in a abstract way. The structures that need to be domain specific are those that describe the diseases.

In order to describe a disease, respectively the symptoms of a patient, *categories* and manifestations of those have to be defined. Additionally, dependencies between categories and features have to be modeled.

To describe the *likelihood* a feature occurs in a disease, concrete values have to be defined.

Furthermore, to depict the epidemiology of a disease, values for the overall *frequency* have to be specified.

In order to specify this information, such that it is both human- and machine-readable, JSON[2] is an effective format which allows the interchange of large amount of structured data. Additionally, formats constructed with JSON can be extended easily and without breaking changes that would invalidate existing structures. Therefore, the disease metadata is formalized using JSON Schema[3] files, that also allow to validate given JSON files.

Alternatives for the representation of the metadata that we considered, but decided against, are:

- XML[4]: Allows the creation of large, structured configuration files. Provides with XML Schema Definition[5] a similar formalization support as JSON. XML files are potentially less human-readable compared to JSON files.
- YAML[6]: YAML provides similar if not better human-readability than JSON. The lack of an official formalization syntax is a big disadvantage of YAML.

Sample input JSON structures with data from the running example can be seen in section A.1.

4.2.1 Categories

The formalization of a category with its respective features can be described by the context-free grammar given in listing 4.1. User input for feature categories is then an array of categories defined by this grammar.

The following elements are defined for each category:

- *name*: The name of the category, respectively the feature.
- *values*: The features of a category. A value can be a simple string, or an object with a *name* and a *dependent* or *parts* attribute.
- *type*: Defines the type of input a category has for the disease data. Possible values are:
  - *contain*: this category takes an array of features per disease as input.

[2]http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
[3]https://tools.ietf.org/html/draft-handrews-json-schema-01
[4]https://www.w3.org/TR/2008/REC-xml-20081126/
[5]https://www.w3.org/TR/2004/REC-xmlschema-1-20041028/
[6]http://yaml.org/spec/1.2/spec.pdf

Listing 4.1: Context-free grammar for a category in EBNF notation

```
category → "{"
  "\"name\":" varName ","
  "\"values\":" "[" varValues "],"
  "\"type\":" varType
  [ "," patient ]
  [ "," conflict ]
  [ "," weight ]
  [ "," least ]
  "}"
varValues → value { "," value }
value → depValue | partValue | varName
depValue → "{\"name\":" varName
            ", \"dependent\": [" categories "]}"
partValue → "{\"name\":" varName
            ", \"parts\": [" varParts "]}"
varParts → varPart { "," varPart }
varPart → varName | partValue
varType → "\"likelihood\"" | "\"contain\"" | "\"ratio\""
patient → "\"patient\" : {\"exclusive\":" boolean "}"
conflict → "\"noHardConflict\":" boolean
weight → "\"weight\":" posDec
least → "\"atLeast\":" posInt
categories → category { "," category }
varName → "\"" alpha { alpha | "_" } "\""
alpha → "a" | "b" | ... | "z" | "A" | "B" ... | "Z"
posDec → "0." nums nonZeroNum | posInt "." nums nonZeroNum
posInt → nonZeroNum [ nums ]
nums → { num }
num → "0" | "1" | ... | "9"
nonZeroNum → "1" | ... | "9"
boolean → "true" | "false"
```

  - *likelihood*: this category takes a likelihood per feature per disease as input.
  - *ratio*: this category takes a ratio of the features per disease as input.

- *patient*: Object that holds options for the patient input of the resulting system. In this thesis the only option that can be given is *exclusive*, which is a boolean value that states, whether the patient input is an array of features or a single feature.
- *noHardConflict*: Upon a mismatch of a feature the default behavior of the system is to exclude the mismatching disease for the patient. With this option set, a

mismatch of a feature in this category will only lead to a downweighting of the disease, rather than excluding it.

- *weight*: A multiplicative weight applied to the sub score of a category during the ranking of the diseases.
- *atLeast*: Positive integer value that describes a constraint. The constraint depends on the given *type* of the category:
  - *contain*: a minimum of *atLeast* features of the category have to be given in the disease input.
  - *likelihood*: a minimum of *atLeast* features of the category have to have the likelihood "yes" in the disease input.
  - *ratio*: this property is ignored.

Furthermore, if a feature value of a category is an object, it has one the following properties besides *name*:

- *dependent*: Property of a feature that holds categories that are dependent on the containing feature. This means values for these dependent categories may only be given in a disease if the dependency feature is potentially present in the disease. The attribute of being "potentially present" is defined differently depending on the type of the dependency category:
  - *contain*: the dependency feature has to be present in the disease.
  - *likelihood*: the dependency feature has to have a *likelihood* other than "no".
  - *ratio*: the dependency constraint is ignored.
- *parts*: Property of a feature that describes parts of this feature in a transitive relationship. Parts can be simple strings or feature objects containing parts themselves.

4.2.2 Frequency

The input for the *frequency* (listing 4.2) is a simple mapping of names to a positive number below or equal to 1 where at least one property must be given. The property names of this object are then the values used to define the overall frequency of diseases.

4.2.3 Likelihood

Input for the *likelihood* is defined analogously to the *frequency* input, with the exception, that the properties "no" and "yes" are mandatory. These are fixed values that are essential for describing disease data. Other values are optional and can be simple value mappings, with positive numbers below or equal to 1, or complex objects.

A *likelihood* object has the following properties:

Listing 4.2: Context-free grammar for frequency input in EBNF notation

```
frequency → "{" variables "}"
variables → variable { "," variables }
variable → varName ":" posDecBelowEqualOne
varName → "\"" alpha { alpha } "\""
alpha → "a" | "b" | ... | "z" | "A" | "B" ... | "Z"
posDecBelowEqualOne → "0." comNum | "1.0"
comNum → num { num }
num → "0" | "1" | ... | "9"
```

- *value*: positive numbers below or equal to 1
- *yesGroup*: boolean property that states whether this likelihood should be grouped with the "yes" value, when grouping of likelihood values is performed[7].

The corresponding context-free grammar can be seen in listing 4.3.

Listing 4.3: Context-free grammar for likelihood input in EBNF notation

```
likelihood → "{"
  "\"yes\":" posDecBelowEqualOne ","
  "\"no\":" posDecBelowEqualOne
  variables
  "}"
variables →  { "," variable }
variable → varName ":" posDecBelowEqualOne | varYesGroup
varYesGroup →  "{\" value \":" varName
                ", \"yesGroup\":" boolean "}"
varName → "\"" alpha { alpha } "\""
alpha → "a" | "b" | ... | "z" | "A" | "B" ... | "Z"
posDecBelowEqualOne → "0."comNum | "1.0"
comNum → num { num }
num → "0" | "1" | ... | "9"
boolean → "true" | "false"
```

4.3 Disease Format

In order to use the decision support system, the disease knowledge base has to be filled beforehand. Therefore, a structured way of capturing and validating diseases is needed

[7]This grouping of likelihood values is only done during the scoring in the TFIDF based method.

that is also accepted by health professionals.

Experience from the Dermtrainer project shows that the acquisition of disease data with a customized, highly structured tool, with branching menus and automatic validation, is poorly accepted by physicians.

Identified problems with this approach were the inability to quickly change features of diseases, and the fact that only one disease could be edited, respectively viewed at a time. Physicians reported that this contradicts their workflow. They wanted to view all diseases and their features in a structured list without the need to browse branching menus.

Therefore, we decided on using Excel spreadsheets[8] for capturing disease data. The usage of spreadsheets for the description of disease data yields several advantages for the user:

- All major operating systems provide tools for creating and editing Excel spreadsheets.
- Spreadsheets let the health professional check and edit whole disease or category groups efficiently.
- Allows grouping of related categories in different sheets.
- Existing disease spreadsheets can be easily adapted to changed or new metadata. This is especially useful, when experimenting with the framework.
- Spreadsheets can be considered as generally common in the medical environment.

The disease spreadsheet has to comply to the following structure:

- The file may have as many sheets as the user needs for grouping categories.
- Completely empty sheets are ignored.
- All input is case-insensitive.
- Each sheet needs to have the first row filled with the category names the application can expect from each column. This row is called “header” from here on.
- The first column of each sheet has to be named “ID” and provide unique ids for each disease.
- Lists of values are separated by a “,” character.
- Mandatory columns that are not dependent on the medical domain, are the following:
  - “NAME”: name of the disease

[8]For compatibility reasons we use Excel 97/2000/XP .xls spreadsheets

  - "FREQUENCY": overall occurrence frequency of the disease. Expects a *frequency* as input.
  - "CONFLICT_EXCLUSION": a list of *category* names for which the disease should have non-excluding conflicts.

- The input of features and the header representation is dependent on the *type* of the category and whether the category is *dependent* on another feature:
  - *likelihood*: Each feature of the category needs to have a *likelihood* value and therefore an entry in the disease spreadsheet. To prevent non-unique entries, the headers of such features are structured as: "*<CATEGORY NAME>_<FEATURE NAME>*".
  - *contain* and *ratio*: The *name* of the *category* is put in the header for the category.
  - *contain* categories take a list of features as cell value.
  - *ratio* categories take a ratio in the form of $x_{ratio} : y_{ratio}$ for feature x to y, whereas the order of features is the same as the order in the *category* input. x_{ratio} and y_{ratio} have to be positive numbers.
  - If a category is *dependent* on a feature, all headers concerning this category have a prefix describing the dependency that is defined as: "*<DEPENDENCY CATEGORY NAME>_<DEPENDENCY FEATURE NAME>_*".
- Null values are expressed with "DNA" (does not apply).

Of course, due to the missing validation upon input in simple spreadsheets, the spreadsheet has to be validated before the data can be used for diagnoses. Besides syntactic correctness, the following constraints have to be met:

- Header entries have to be successively.
- Each id (disease) has to be provided in each sheet that is not empty exactly once.
- Each row containing a disease has to be complete, such that for each header there is a valid value supplied[9].
- If the category has an *atLeast* constraint, the number of features that can appear with a disease have to be at least *atLeast*.
- If the feature, a category *depends* on, is not present in the disease, "DNA" has to be put in all corresponding cells of the dependent category. Furthermore, a potential *atLeast* constraint of the dependent category is overridden.

[9]Note that a cell can still be empty, due to an empty list being a valid value for certain features.

- If the feature, a category *depends* on, is present in the disease, all corresponding cells of the dependent category have to contain valid values other than “DNA”.
- “ID”, “NAME” and “FREQUENCY” are mandatory fields and have to be provided in the spreadsheet. Other fields are optional as long as they are not provided in the header of a sheet.
- Headers, other than“ID”, must not be provided in multiple sheets.

A sample input file with categories of the running example can be seen in figure A.1.

CHAPTER 5

Medical Decision Support Framework

Usually when reading about a "framework" in the domain of computer science, one expects a collection of software defined, generic functionality that can be used and adapted by additional code written by the user. This implies that a user needs at least basic knowledge on how to write code in order to use a software framework. The overall goal of this thesis is to enable physicians to use this framework without the assistance of a computer scientist.

Since we assume that the majority of physicians do not have this kind of knowledge, we decided to adapt the term "framework" in this thesis: Instead of using custom code as primary mean of customization, this framework is defined as a code generator, only needing the structures defined in section 4.2, in addition to some trivial information to be configured properly[1].

In order to provide a robust and extensible code generator, yeoman[2] is a useful collection of best practice open source tools for scaffolding web applications:

- yeoman generators are, at their core, Node.js[3] modules. This makes a generator easy to extend due to having access to the Node.js ecosystem via its package manager npm[4].

[1]Of course, the code generated by this framework can also be used as proper software framework by computer scientists.

[2]`http://yeoman.io/`

[3]`https://nodejs.org/`

[4]`https://www.npmjs.com/`

- yeoman uses *adapters* as an abstraction layer for user interaction. This makes it particular easy to integrate a yeoman generator into another application and for instance, provide a custom graphical user interface[5].
- yeoman offers comprehensive documentation and tutorials for setting up a generator[6].

We considered Grunt[7] and Gulp[8] as alternatives for yeoman, which are tools for automating recurring tasks during development. Both tools would allow for to create a automated code generator, but user input, file access and the overall workflow of the generator would have to be implemented, in comparison to yeoman which delivers these functionalities out of the box.

Therefore, this chapter describes the yeoman generator and the data structures used to create the decision support system.

5.1 Framework Structure

Yeoman generators are built on the concept of sub generators. Specific sub generators can be called separately from the user. Sub generators themselves can be composed with other sub generators. The generator presented in this thesis comprises three sub generators, as can be seen schematically in figure 5.1:

1. *app*: The default sub generator that started when no sub generator is selected explicitly by the user. This sub generator collects the information needed from the user for the *client* and the *server* sub generator and then starts both.
2. *client*: Generates the client side of the decision support system.
3. *server*: Generates the server that contains the business logic of the decision support system.

5.2 Generation Workflow

A yeoman generator uses a run loop which is a queue system that supports different priorities for tasks. Since the proposed generator is composed of three generators, we do not run tasks sequentially, but use the predefined priority groups that are supported by yeoman:

1. *initializing* - initialization of the generator

[5]By default yeoman generators are operated via the command line.
[6]yeoman even provides a “generator-generator” that scaffolds the basic structures for a new generator.
[7]`https://gruntjs.com/`
[8]`https://gulpjs.com/`

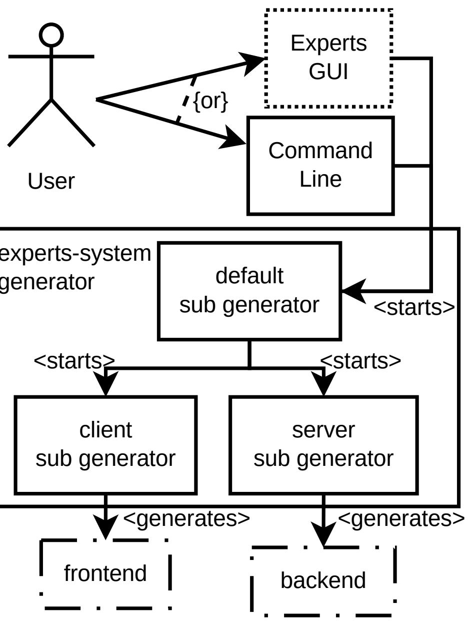


Figure 5.1: Schematic structure of the framework. Dashed rectangles represent the generated parts of the framework. Doted rectangles represent parts that are out of scope for this thesis, but could enhance the usability of the framework.

2. *prompting* - asking the user for input

3. *configuring* - configuring the project and saving the given input

4. *default* - all methods without an explicit priority

5. *writing* - writing files to the disk

6. *conflicts* - handling of conflicts - *Not needed in this generator.*

7. *install* - installing of dependencies for the generated code - *Not needed in this generator.*

8. *end* - cleanup - *Not needed in this generator.*

In the following sections, the tasks executed in the priority groups that are used in this generator are described in detail.

5.2.1 Initializing

First, after starting the generator or a specific sub generator, the previously saved configuration is loaded from the disk[9] if the generator is started in an already existing project. This step is skipped if the sub generator is called from another sub generator, since the calling generator provides the initializing configuration as parameters.

5.2.2 Prompting

The user is prompted options that configure the decision support system. Options, in the order they are prompted, are as follows:

- **Project name:** The name of this decision support system. This will only be displayed in the frontend. *Default:* "ExpertSystem"
- **Input format:** The format for the *frequency*, *category* and *likelihood* input. Formats to choose from are *JSON* and *DSL*. In the scope of this thesis only JSON is supported, the option for DSL is deactivated. However, preparatory actions have been carried out, such that adding a DSL support can be integrated.
- **Frequency input:** JSON input for the Frequency object, expected to be structured as described in section 4.2.2.
- **Likelihood input:** JSON input for the Likelihood object, expected to be structured as described in section 4.2.3.
- **Category input:** JSON input for the Category objects, expected to be structured as described in section 4.2.1.
- **Java package name:** Advanced option for specifying the package name of the backend Java application. *Default:* "com.mycompany.myexpertsystem"
- **Score:** The method that should be used to calculate scores for the disease ranking. Options are "TFIDF" for the TFIDF based or "Probabilistic" for the probability-based algorithm.

JSON input provided by the user is validated with the JSON Schemas described in the respective sections. Non-compliant Input is not excepted and an error message is shown to the user.

If the generator is started in an already existing project, only the options **Frequency input**, **Likelihood input**, **Category input** and **Score** are prompted. This allows the user to change disease metadata and the score algorithm of an existing decision support system.

JSON input objects for the running example can be seen in the appendix in section A.1.

[9]Yeoman stores configuration properties that are explicitly marked to be saved in a generator in the project directory in a *.yo-rc.json* file.

5.2.3 Configuring

First, the JSON input objects are parsed. Folders and package structures are set up. Finally, files that should be written to disk are specified and properly configured with the given configuration.

5.2.4 Default

The configuration is saved to the *.yo-rc.json* file of the project.

5.2.5 Writing

Files are written to disk according to the configuration. Static files are only copied, template files are adopted to the specified domain information using the Embedded JavaScript (EJS)[10] templating syntax, as it is a simple way of templating with pure JavaScript and is included in yeoman generators by default.

[10] `http://ejs.co/`

CHAPTER 6

Generated System

The next goal is to provide a web application that can be deployed by the generator defined in chapter 5. This chapter describes the overall structure and main components of this generic decision support system.

Since the system needs to be adapted to the specific domain, we need to specify constant domain specific information in a deployed system. This is possible by providing EJS templates in files that need to contain domain specific information.

For the overall structure of the system, we decided for a basic three tier architecture as can be seen in figure 6.1. This allows for the *client* and the *server* to be deployed and started separately which correlates with the structure of the generator. The main

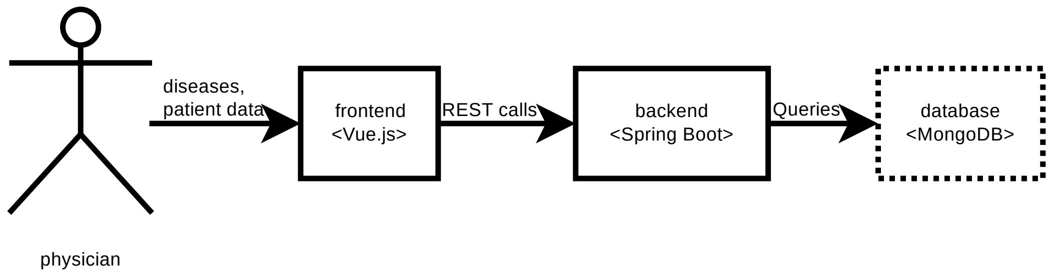


Figure 6.1: Schematic structure of a system generated by the framework. Dotted components represent dependencies that need to be provided by the user.

components of the system are as follows:

- A Vue.js[1] web application as GUI, facing the user.

[1]`https://vuejs.org/`

- The server exposes REST endpoints for the web application to call. The implementation is done in Java 8, using the Spring Framework[2].
- For persisting disease data, a MongoDB[3] instance is used. NOTE: The database has to be provided by the user, exposed on the default port `27017`.

6.1 Server

As mentioned above, the server is a Java 8 application based on the Spring Framework. To minimize the configurational effort of Spring, Spring Boot[4] is used.
Spring Boot provides an embedded Tomcat, automatically configures Spring and third-party libraries and does not require XML configuration. Spring boot also offers "starter" dependencies that bundle preconfigured parts of the Spring Framework for specific application areas. In this application the starters `web`, for setting up REST endpoints and `data-mongodb` to abstract the MongoDB database connection are used.

6.1.1 Structure

Structurally we opted for a best practice layered architecture (figure 6.2) with `controller`, `service` and `persistence` layer.

The `rest` package provides controller classes that serve the REST endpoints. Input is passed directly to the corresponding service classes in the `service` package with the exception of disease spreadsheets. Spreadsheets are validated for completeness of diseases and parsed into `rest.model` entities.

Validation of user input is done using annotation based Bean Validation[5] on `rest.model` classes. The validation is triggered before a `rest.model` entity enters the `service` layer.
Since we introduced disease category specific constraints in section 4.3, we implemented custom Bean Validation field annotations and corresponding validators:

- `@AtLeast`: Takes an integer x as parameter. If the annotated field is a java.util.Collection and holds feature category enum values, the validator checks that at least x values can be observed with the disease[6]. This annotation can only be applied to fields of the disease model.
- `@DependentOn`: Models a dependency that a category has on a feature. Takes a string s as parameter that has to be in the form of

[2]`https://spring.io/`
[3]`https://www.mongodb.com/`
[4]`https://spring.io/projects/spring-boot`
[5]`https://beanvalidation.org/`
[6]This depends on the type of the category the annotated field belongs to. Please refer to section 4.2.1 for detailed description.

"*<category_field_name>.<dependent_feature_name>*", all in lower case. Therefore, the field of the category containing the dependency and the feature that represents the actual dependency are specified.

The annotated field has to be a java.util.Collection that holds feature category enum values, or a simple category enum field.

Input is valid if the annotated field only holds values when the dependency feature occurs with the disease.

Additionally, the annotation can be given an *atLeast* integer parameter that models the same constraint like the `@AtLeast` annotation, but in conjunction with the dependency.

This annotation can be used in disease and patient models, but the *atLeast* parameter is ignored for patient models.

The business logic of the server is provided by the `service` package. Before processing the input data, entities are mapped from `rest.model` to `core.profile` entities to achieve a clear separation between controller and service. Saving new disease data triggers the `renewDiseaseBase` method that reloads all disease data within the `PatientService`. This `PatientService` expects the existence of a `ScoreFactory` Bean that is used to create scores and rank diseases accordingly. This usage of dependency injection makes it particular easy to change the implementation of the actual scoring algorithm.

The core package comprises enum constants and profiles used to describe patient and disease data. Additionally, the interface `ScoreFactory` and its implementations are included in this package.

A `ScoreFactory` provides the following methods used by the system:

- `calculateScores` takes a patient profile and calculates scores for all diseases that this factory holds. A score holds the *rank* and the *similarity* metrics, as well as information about the disease.
- `bestOf` takes a list of scores and selects the "best". This can be as simple as sorting the scores for the highest values and returning the first x scores.
- `getScoreName` returns the name of the score.
- `setDisesaes` method to update the disease base of the score factory.

For each *category* defined by the user during the setup, an `enum` is created by the framework in the `core.featurecategory` package. Both *frequency* and *likelihood* enums are created in the `core.enumeration` package. This package also holds the *category* enum which lists all generated categories. The generated enums for the running example can be seen in figures A.2 - A.5.
Other classes that are derived from templates that depend on the generated *category*, *frequency* and *likelihood* enums are as follows:

- `rest.model.DiseaseModel`: Model class for diseases. Category fields are dependent on the type of the category.
- `rest.model.PatientModel`: Model class for patient data. Category fields are dependent on the type and the patient options of the category.
- `core.profile.DiseaseProfile`: Service class for diseases. Analogous to the model class.
- `core.profile.PatientProfile`: Service class for patient data. Analogous to the model class.
- `rest.file.Attributes`: Class that holds string constants, naming all possible header values for disease spreadsheets.
- `core.score.*.*MethodFactory`: Implementations for the respective score factories. Weights of categories are added.
- `core.score.*.*Computations`: Implementations for the actual score computations per category for the respective scores. All categories are statically referenced and their scores calculated. Additionally, the configuration whether a category has hard conflicts is set.

All other files only get `package` and `import` statements set.

6.1.2 REST endpoints

A REST API is served by the application to communicate with the frontend which serves the following endpoints:

- POST */api/diseases/xls*: Expects a multipart request containing a .xls file that is structured as described in section 4.3. Validates and saves the diseases provided in the file and updates the disease base of the system.
- GET */api/diseases*: Returns an array of all disease profiles.
- POST */api/patient*: Expects a patient profile and returns an array of ordered diseases.

A Swagger API documentation[7] of the running example can be seen in listing A.5.

[7] `https://swagger.io/solutions/api-documentation/`

6.2 Persistence

As mentioned before, the system uses a MongoDB to store the disease data. We decided for a document-oriented database because of the following reasons:

- Conceptually a disease can be seen as a document which contains a number of key value pairs. Since there can be dependencies defined between categories, keys may have no values. If the system would use a relational database system we therefore would have a lot of unnecessary `null` values. Diseases are queried only completely, so no indexing or query optimization is necessary.
- The framework presented in this thesis is designed, so that the user can experiment with it and change the disease metadata in an easy way. Therefore, when the user changes the structure of the system, either by using the framework or manually, no migration scripts have to be written for the database in order keep already saved data. This allows for quick iterations during the development process.

6.3 Client

The frontend of the system uses Vue.js[8] and is based upon the free version of CoreUI[9]. Vue.js is a versatile JavaScript framework for constructing user interfaces. At its core Vue.js only focuses on the view layer, but can be extended by an adoptable ecosystem of libraries. Since scaffolding and configuring a web application from scratch can be a tedious task, we decided to use a template as a baseline to work with.
In CoreUI basic development, as well as production environment are preconfigured. It features a custom CSS theme and fully integrated Bootstrap 4[10] for the styling of the application. Furthermore, basic routing is configured and the overall structure is set up, such that new pages and features can be implemented without large changes.

The web application is designed to be as generic as possible, what results in `config.js` being the only file holding user defined information adapted by the framework. In this file categories, features and their dependencies are defined which are used to render the user interface. Due to the domain of this application being a decision support system, we decided that the user interface should be divided in two sections: one for providing and altering the disease base (figures 6.5) and one for diagnosing patients (figure 6.3 and 6.4).

A small, custom-built Express[11] based web-server serves the application. Besides hosting the application, this server proxies all requests to the actual server, except PUT requests on */feature* which are used to save feature figures directly on the web-server. We decided against storing feature figures in the database, because this would weaken the separation of the tiers.

[8]`https://vuejs.org/`
[9]`https://coreui.io/`
[10]`https://getbootstrap.com/`
[11]`http://expressjs.com/`

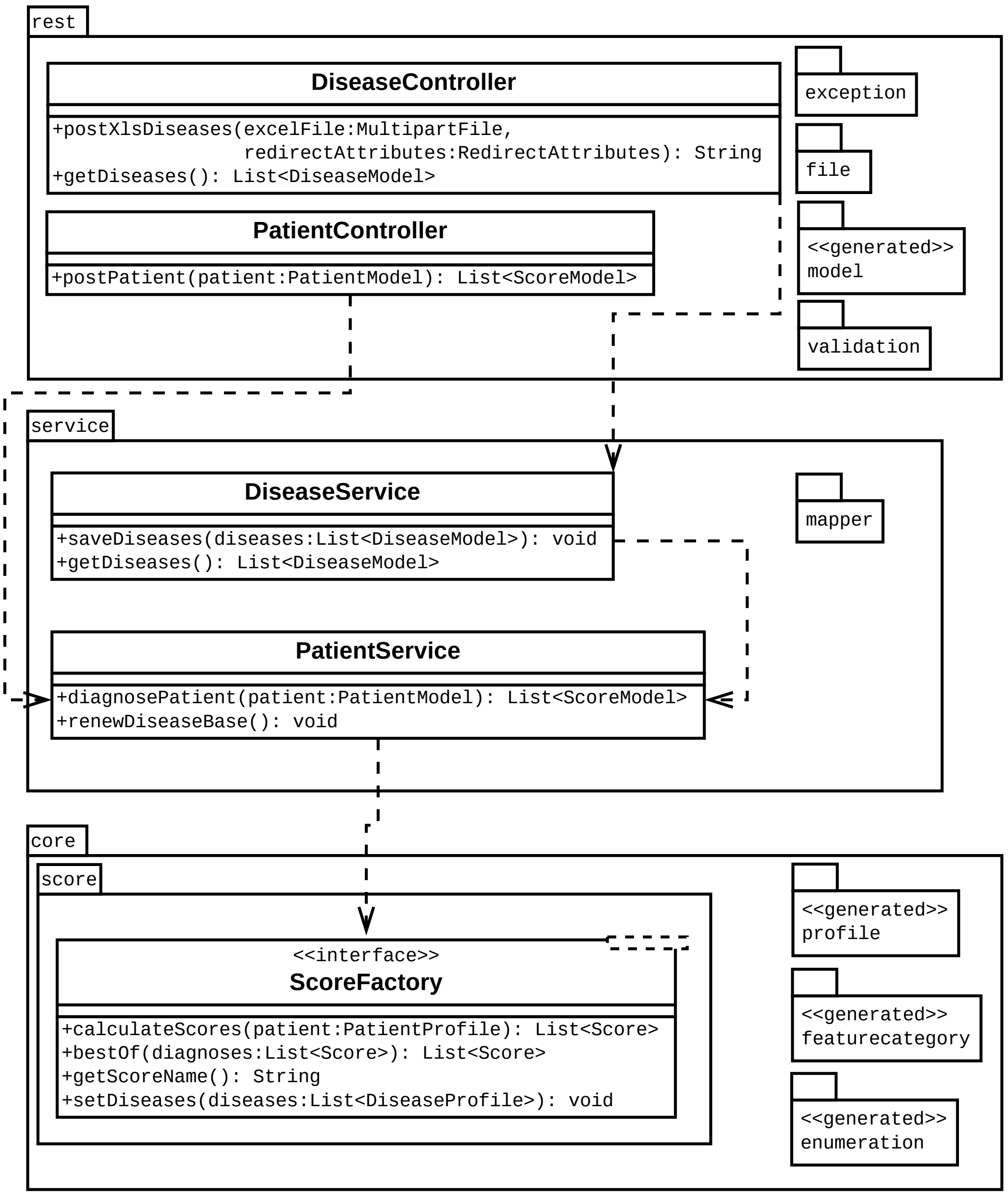

rest
DiseaseController
+postXlsDiseases(excelFile:MultipartFile,
redirectAttributes:RedirectAttributes): String
+getDiseases(): List<DiseaseModel>
PatientController
+postPatient(patient:PatientModel): List<ScoreModel>
exception
file
<<generated>>
model
validation
service
DiseaseService
+saveDiseases(diseases:List<DiseaseModel>): void
+getDiseases(): List<DiseaseModel>
PatientService
+diagnosePatient(patient:PatientModel): List<ScoreModel>
+renewDiseaseBase(): void
mapper
core
score
<<interface>>
ScoreFactory
+calculateScores(patient:PatientProfile): List<Score>
+bestOf(diagnoses:List<Score>): List<Score>
+getScoreName(): String
+setDiseases(diseases:List<DiseaseProfile>): void
<<generated>>
profile
<<generated>>
featurecategory
<<generated>>
enumeration

Figure 6.2: Class structure of the server application. The core package does not represent a separate layer, but rather the core structures used to compute diagnoses. Due to the persistence layer being provided by the Spring Framework, it is not illustrated here.

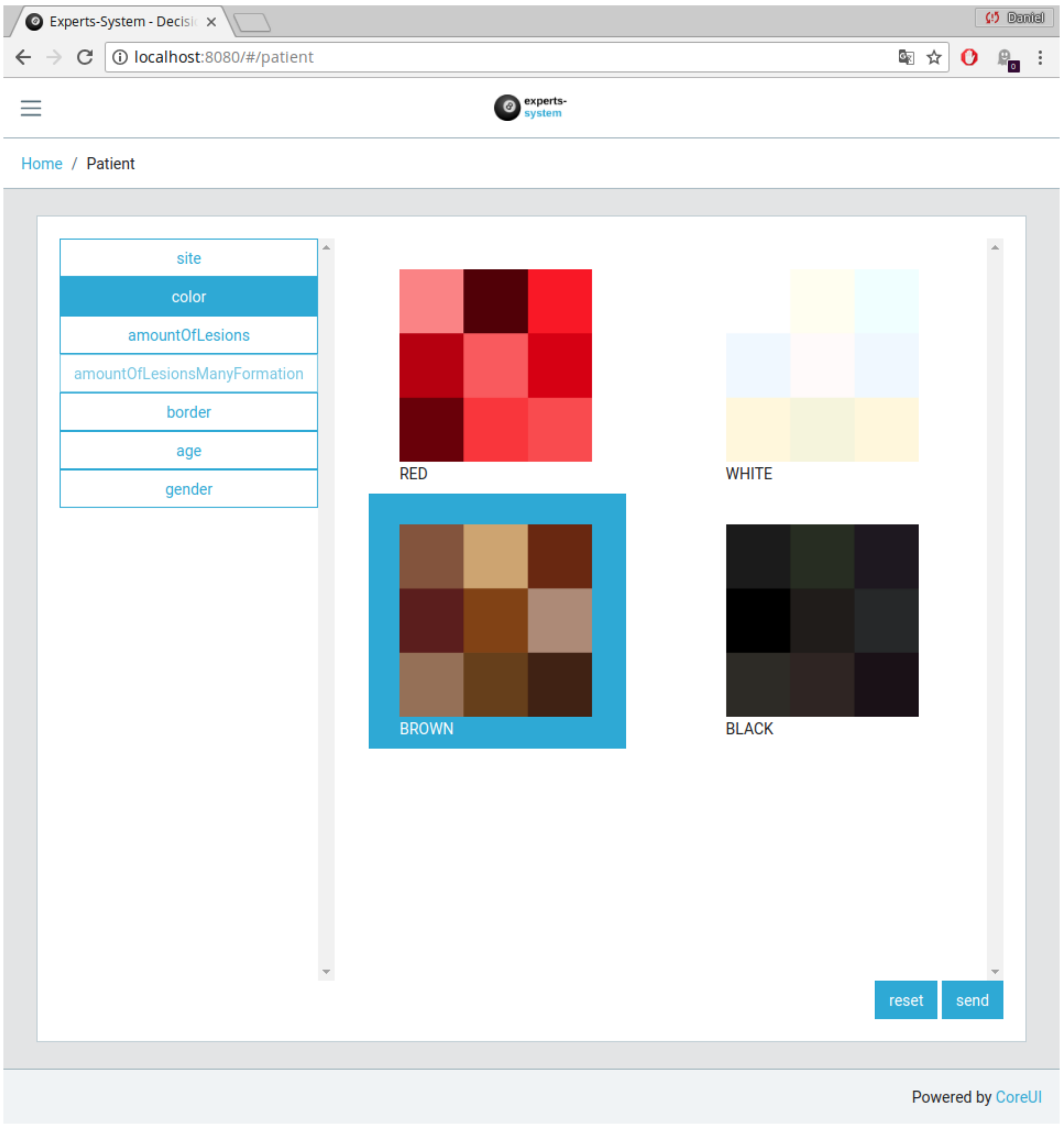
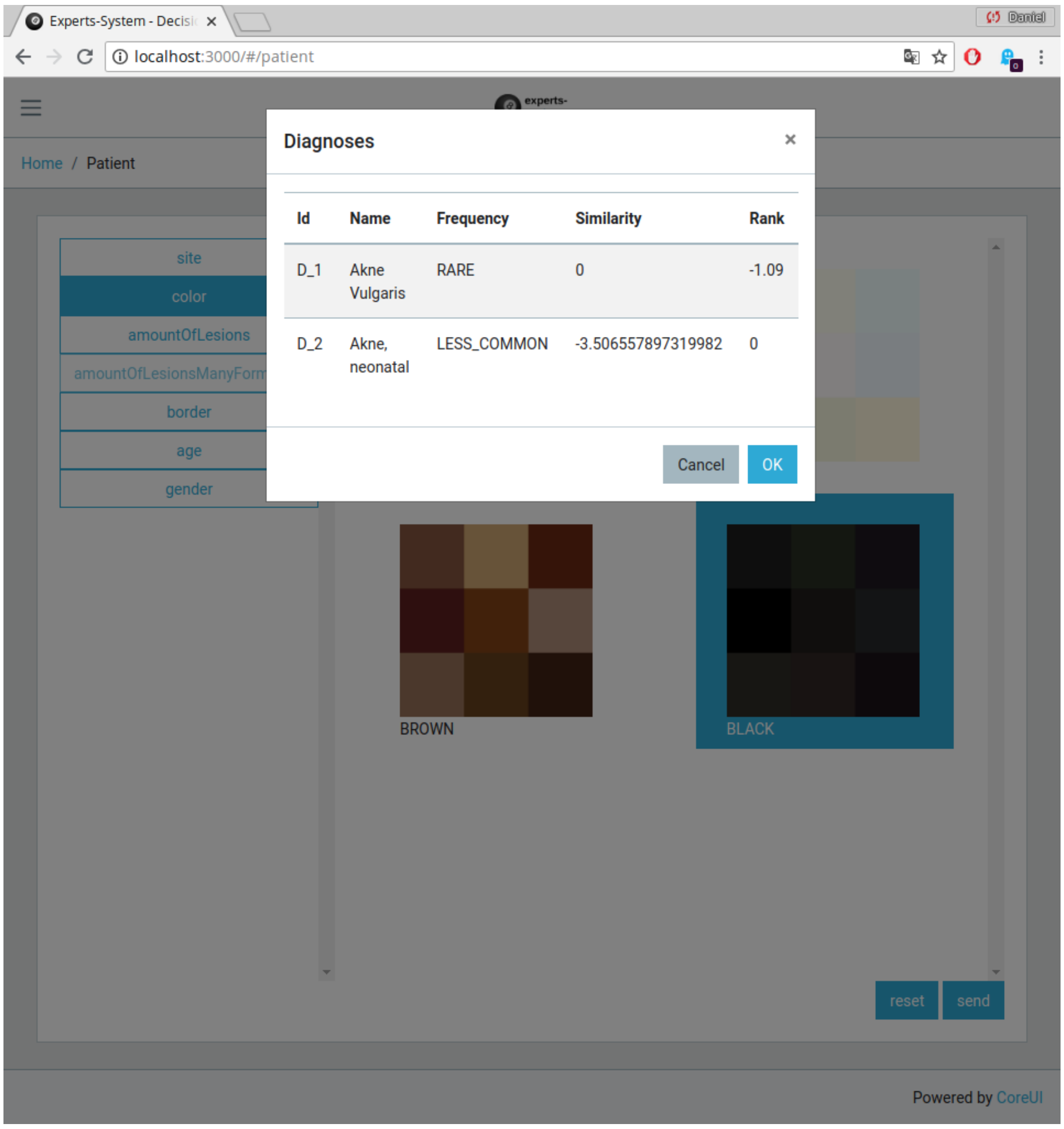


Figure 6.3: Interface for entering patient findings. The side bar on the left side represent the possible categories, on the right side are the corresponding features with graphical representation. Greyed out categories cannot be chosen, due to the dependency not fulfilled (In this case *amountOfLesions* has to have the value "MANY" to unlock the dependent category). Selected features have a blue background.



Diagnoses

Id	Name	Frequency	Similarity	Rank
D_1	Akne Vulgaris	RARE	0	-1.09
D_2	Akne, neonatal	LESS_COMMON	-3.506557897319982	0

Figure 6.4: Popup that shows the user the diseases that match the patient input according to the used score.

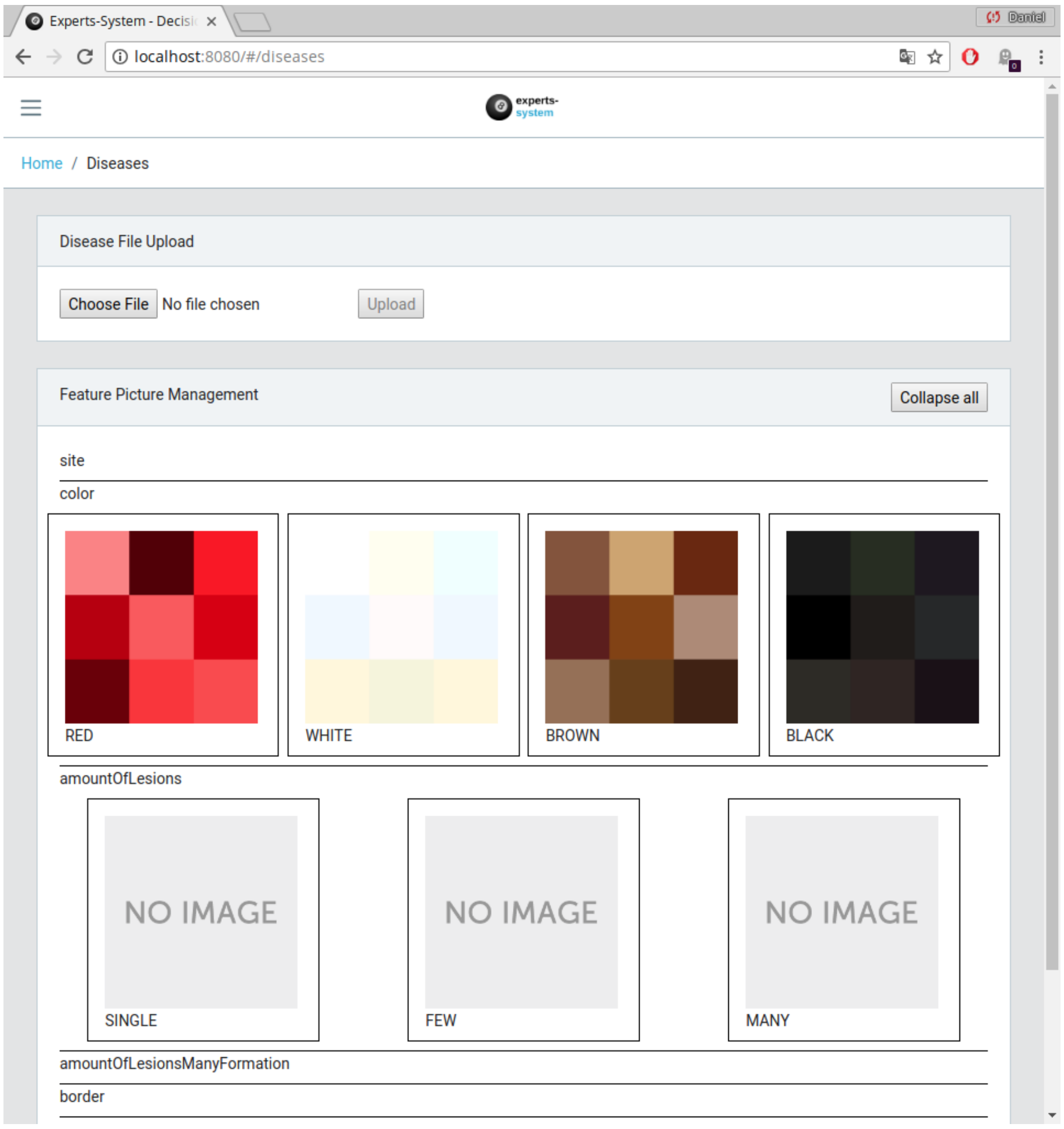


Figure 6.5: Interface for submitting disease data to the server and setting the figures for features. The feature figures for the color categories have been set beforehand and are not part of the framework. When generating a new system, all features will show the “NO IMAGE” figure.

CHAPTER 7

Evaluation

In the following chapter, we evaluate the correctness of the framework and analyse its usability.

7.1 Evaluation of correctness

This section evaluates the correctness of a generated system in conjunction with the configuration flexibility the framework provides. For this we attempt to configure the framework with the disease metadata found in Dermtrainer and compare the diagnoses of the generated system to the diagnoses from Dermtrainer.

Test data[1] is taken from the original development project of Dermtrainer and includes a disease knowledge base of 620 diseases and 422 patient cases diagnosed by dermatologists, respectively medical students. Metrics used for comparing scores are the number of "good", "best" and "correct" diagnoses that are among the first six diseases. These terms were assigned to diseases per patient case by a dermatologist and can be described by the following definitions:

- "good": A disease that matches the symptoms of the patient "good". The disease could be a possible diagnosis a dermatologist might make.
- "best": A disease that matches the symptoms of the patient "best". The disease most likely could be a possible diagnosis a dermatologist might make.
- "correct": The correct diagnosis in a particular case.

The results (table 7.1) show that both, the systems created with the framework and Dermtrainer, achieve comparable overall diagnostic quality.

[1]Unfortunately, due to copyright reasons we cannot include the test data in this thesis.

System	**“correct”**	**“best”**	**“good”**
Framework with score based on TFIDF	373	487	180
Framework with Probabilistic score	354	435	133
Dermtrainer	387	457	164

Table 7.1: The accumulated numbers of “correct”, “best” and “good” diseases a system diagnosed in the test data. The framework based systems are set up equivalently, only the score implementations differ.

This proves that the framework presented in this thesis can be used to construct decision support systems that deliver comparable results when compared to an already existing system for dermatology that has been specifically constructed and tuned for the domain. Additionally, it shows that the configuration options of the framework are flexible enough, such that the domain of an existing decision support system can be modeled.

It is worth mentioning, that although the number of “correct” diagnoses is lower for the system using the TFIDF score, it has considerably higher numbers for “best” and “good” diagnoses than Dermtrainer.

7.2 Analysis of usability

This framework is intended to be used by physicians directly, which might not have the specific knowledge necessary to operate and configure the generator from the command line without assistance. Since, as stated before, it is unrealistic to assume that physicians will devote their time to this project for a proper evaluation, we will analyse the proposed interface for configuring the framework regarding the required skills.

For this we want to consider two use cases for the framework and analyse these, whether they are suitable for a layperson in the field of computer science:

1. The user is left alone with the framework and has this thesis as reference work.
2. The user gets an instance of the framework that has already been set up with exemplary category, frequency and likelihood JSON files. An exemplary disease spreadsheet containing the example categories is available. Someone that is familiar with the framework and the input structures explains these to the user.

Although it was our initial goal, the prototype of the framework presented in this thesis cannot be operated without profound knowledge in the field of computer science, which renders the first use case unrealistic. A layperson can not be expected to operate the generator and configure/start the application via the command line, let alone provide the necessary database or other dependencies.

However, when the user is given an already set up system with the corresponding JSON structures, it basically leaves the user to adapt or add properties in these JSON structures and the disease spreadsheet to effectively change the system. Due to the already existing JSON files, the user can grasp the meaning of single properties easier. Basically, in this use case all steps that need more knowledge in the field of computer science than we can expect from a layperson, are taken over by a computer scientist. This way the user only has to focus on the domain specific elements.

During the development of Dermtrainer, the introduction of new categories by the dermatologist had been stopped due to the high implementation effort at some point. This reduced the amount of experiments that could be conducted with different category configurations. If the dermatologists would have gotten the preconfigured framework as described in the second use case, they would have had virtually no limitations on how often disease metadata could be modified.

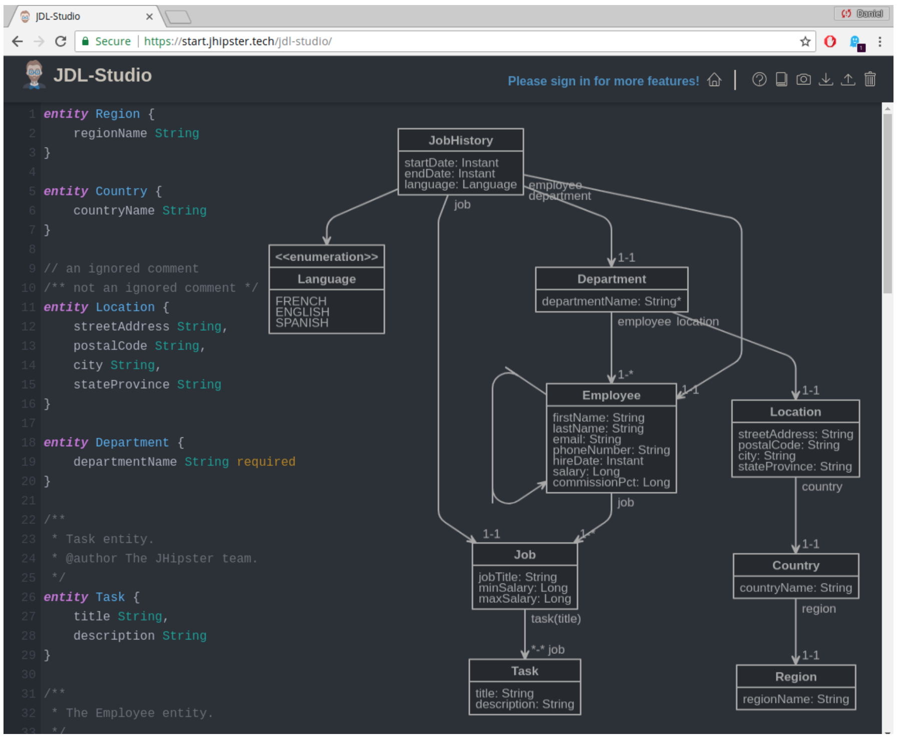
Having reduced the complexity of configuring the framework in the second use case, the skills required by a user reduce to having a basic knowledge of abstract concepts and computer science related terms like `object`, `array` and `string`.

The use of a graphical JSON editor like Visual JSON Editor[2] or JSON Formatter[3] could alleviate some of those requirements further, because it reduces the effort of changing JSON structures to entering data into simple input forms input. Due to the inconclusive definition of some properties (e.g. a feature value of a category can be a simple string or a more complex object) a straight-forward generation of a input form from the JSON Schema is not possible.

For the first use case to potentially become realistic, we identified the following improvements for the framework:

- A custom graphical user interface that is adapted to the needs of health professionals and meets the following requirements:
  - A diagram of the created entities with drag and drop functionalities should be rendered for easy dependency and entity management.
  - The input should be editable directly as text, for quick, precise changes.
  - Validation of the input should be done by the GUI as well.
  - A custom DSL language should be established that formalizes all input characteristics.
- Plausible example configurations for categories, frequency and likelihood input that demonstrates all possible types and optional properties for entities based in existing medical domains.
- A manual that describes all properties and options for entities and the framework.

[2]`https://github.com/rsuter/VisualJsonEditor`
[3]`https://jsonformatter.org/json-editor`



```
entity Region {
    regionName String
}

entity Country {
    countryName String
}

// an ignored comment
/** not an ignored comment */
entity Location {
    streetAddress String,
    postalCode String,
    city String,
    stateProvince String
}

entity Department {
    departmentName String required
}

/**
 * Task entity.
 * @author The JHipster team.
 */
entity Task {
    title String,
    description String
}

/**
 * The Employee entity.
 */
```

Figure 7.1: Web editor for JDL documents with diagram rendering [31]. The structure of this editor could be used as a model for the development of a GUI for the decision support system framework.

As a model for these improvements, we propose the JHipster Domain Language (JDL) Studio[4] from JHipster which can be seen in figure 7.1. Although it misses the essential drag and drop feature of entities, so that it can be used without using a domain specific language at all, we think that this is a solid design starting point for the development of a GUI. Additionally, it includes entity examples and a comprehensive description of the language and the editor.

[4]https://start.jhipster.tech/jdl-studio/

CHAPTER 8

Conclusion

This thesis gave a contribution to generic frameworks for medical decision support systems and the description of disease metadata.

A generic framework was developed that allows for the setup of a medical decision support without the need to write additional code. Input structures were defined that describe disease metadata such that features of diseases, their occurrence probability and the epidemiology of diseases can be specified. Two generic methods for the computation of the similarity of disease and patient data were presented.

To evaluate the framework, we replicated the existing dermatology decision support system Dermtrainer, using only the presented framework and its structures. The domain of Dermtrainer could be completely mapped and the resulting system achieved comparable results to Dermtrainer.

Although the framework was found to be not usable for a laymen in the field of computer science on their own, we presented a use case in which the framework could be used by a laymen successfully.

This offers a range of possibilities for future work with this framework. A GUI should be implemented for the usage of the framework, so that a user does not have to use the CLI. Also exemplary configurations and a proper manual could help to make the framework easier to use.

The framework proved to be flexible enough to depict the dermatology domain of Dermtrainer, therefore, experimenting with other medical domains is a potential area. As for the decision support system itself, new methods of calculating the similarity between disease and patient data should be explored and implemented to further increase the diagnostic quality.

APPENDIX A

Running Example

A.1 Disease Metadata

Listing A.1: Category input for the running example

```
[
  {
    "name": "site",
    "values": [
      "head",
      {
        "name": "torso",
        "parts": [
          "breast",
          "stomach"
        ]
      },
      {
        "name": "arms",
        "parts": [
          {
            "name": "hands",
            "parts": [
              "fingers"
            ]
          }
        ]
      },
      {
        "name": "legs",
        "parts": [
          {
            "name": "feet",
            "parts": [
              "toes"
            ]
          },
          "knees"
        ]
      }
    ],
    "type": "contain",
    "weight": 3
  },
  {
    "name": "color",
    "values": [
      "red",
      "white",
      "brown",
      "black"
    ],
    "type": "likelihood",
    "atLeast": 1,
    "patient": {
      "exclusive": true
    }
  },
  {
    "name": "amountOfLesions",
    "values": [
      "single",
      "few",
      {
        "name": "many",
        "dependent": [
          {
            "name": "formation",
            "values": [
              "scattered",
              "cumulative"
            ],
            "type": "likelihood",
            "atLeast": 1,
            "patient": {
              "exclusive": true
            }
          }
        ]
      }
    ],
    "type": "likelihood",
    "atLeast": 1,
    "patient": {
      "exclusive": true
    }
  },
  {
    "name": "border",
    "values": [
      "delimited",
      "irregular"
    ],
    "patient": {
      "exclusive": true
    },
    "type": "likelihood"
  },
  {
    "name": "age",
    "noHardConflict": true,
    "values": [
      "infant",
      "child",
      "adult",
      "elder"
    ],
    "patient": {
      "exclusive": true
    },
    "type": "contain",
    "atLeast": 1
  },
  {
    "name": "gender",
    "values": [
      "male",
      "female"
    ],
    "patient": {
      "exclusive": true
    },
    "type": "ratio"
  }
]
```

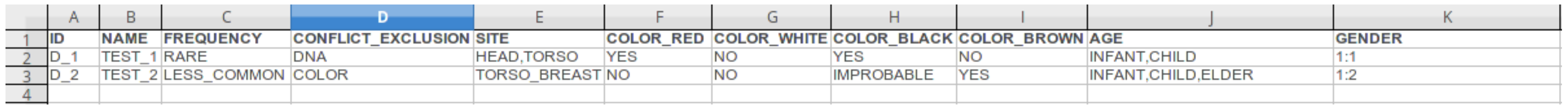
Listing A.2: Likelihood input for the running example

```
{
        "yes": 1.0,
        "no": 0.0005,
        "improbable": {
                "value": 0.03,
                "yesGroup": true
        }
}
```

Listing A.3: Frequency input for the running example

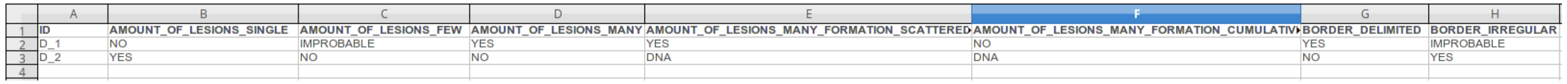
```
{
        "rare": 0.00001,
        "less_common": 0.001,
        "common": 0.01,
        "very_common": 0.1
}
```

A.2 Disease Format



	A	B	C	D	E	F	G	H	I	J	K
1	ID	NAME	FREQUENCY	CONFLICT_EXCLUSION	SITE	COLOR_RED	COLOR_WHITE	COLOR_BLACK	COLOR_BROWN	AGE	GENDER
2	D_1	TEST_1	RARE	DNA	HEAD,TORSO	YES	NO	YES	NO	INFANT,CHILD	1:1
3	D_2	TEST_2	LESS_COMMON	COLOR	TORSO_BREAST	NO	NO	IMPROBABLE	YES	INFANT,CHILD,ELDER	1:2
4											

(a) Sheet 1 with mandatory category input.



	A	B	C	D	E	F	G	H
1	ID	AMOUNT_OF_LESIONS_SINGLE	AMOUNT_OF_LESIONS_FEW	AMOUNT_OF_LESIONS_MANY	AMOUNT_OF_LESIONS_MANY_FORMATION_SCATTERED	AMOUNT_OF_LESIONS_MANY_FORMATION_CUMULATIV	BORDER_DELIMITED	BORDER_IRREGULAR
2	D_1	NO	IMPROBABLE	YES	YES	NO	YES	IMPROBABLE
3	D_2	YES	NO	NO	DNA	DNA	NO	YES
4								

(b) Sheet 2.

Figure A.1: Exemplary disease input spreadsheet for the running example with two sheets.

A.3 Client

Listing A.4: Configuration of the categories for the frontend of the running example

```
{
  "site": {
    "values": [
      "HEAD",
      {
        "name": "TORSO",
        "dependent": []
      },
      {
        "name": "ARMS",
        "dependent": []
      },
      {
        "name": "LEGS",
        "dependent": []
      }
    ],
    "exclusive": false
  },
  "color": {
    "values": [
      "RED",
      "WHITE",
      "BROWN",
      "BLACK"
    ],
    "exclusive": true
  },
  "amountOfLesions": {
    "values": [
      "SINGLE",
      "FEW",
      {
        "name": "MANY",
        "dependent": [
          "amountOfLesionsManyFormation"
        ]
      }
    ],
    "exclusive": true
  },
  "amountOfLesionsManyFormation": {
    "values": [
      "SCATTERED",
      "CUMULATIVE"
    ],
    "exclusive": true
  },
  "border": {
    "values": [
      "DELIMITED",
      "IRREGULAR"
    ],
    "exclusive": true
  },
  "age": {
    "values": [
      "INFANT",
      "CHILD",
      "ADULT",
      "ELDER"
    ],
    "exclusive": true
  },
  "gender": {
    "values": [
      "MALE",
      "FEMALE"
    ],
    "exclusive": true
  }
}
```

A.4 Server

Listing A.5: Swagger API documentation for the running example

```
{
  "swagger": "2.0",
  "info": {
    "description": "Api Documentation",
    "version": "1.0",
    "title": "Api Documentation",
    "license": {
      "name": "Apache 2.0",
      "url": "http://www.apache.org/licenses/LICENSE-2.0"
    }
  },
  "host": "localhost:8090",
  "basePath": "/api",
  "tags": [
    {
      "name": "disease-controller",
      "description": "Disease Controller"
    },
    {
      "name": "patient-controller",
      "description": "Patient Controller"
    }
  ],
  "paths": {
    "/diseases": {
      "get": {
        "tags": [
          "disease-controller"
        ],
        "summary": "getDiseases",
        "operationId": "getDiseasesUsingGET",
        "produces": [
          "*/*"
        ],
        "responses": {
          "200": {
            "description": "OK",
            "schema": {
              "type": "array",
              "items": {
                "$ref": "#/definitions/DiseaseModel"
              }
            }
          },
          "401": {
            "description": "Unauthorized"
          },
          "403": {
            "description": "Forbidden"
          },
          "404": {
            "description": "Not Found"
          }
        }
      }
    },
    "/diseases/xls": {
      "post": {
        "tags": [
          "disease-controller"
        ],
        "summary": "postXlsDiseases",
        "operationId": "postXlsDiseasesUsingPOST",
        "consumes": [
          "multipart/form-data"
        ],
        "produces": [
          "*/*"
        ],
        "parameters": [
          {
            "name": "excelFile",
            "in": "formData",
            "description": "excelFile",
            "required": true,
            "type": "file"
          }
        ],
        "responses": {
          "200": {
            "description": "OK",
            "schema": {
              "type": "string"
            }
          },
          "201": {
            "description": "Created"
          },
          "401": {
            "description": "Unauthorized"
          },
          "403": {
            "description": "Forbidden"
          },
          "404": {
            "description": "Not Found"
          }
        }
      }
    },
    "/patient": {
      "post": {
        "tags": [
          "patient-controller"
        ],
        "summary": "postPatient",
        "operationId": "postPatientUsingPOST",
        "consumes": [
          "application/json"
        ],
        "produces": [
          "*/*"
        ],
        "parameters": [
          {
            "in": "body",
            "name": "model",
            "description": "model",
            "required": true,
            "schema": {
              "$ref": "#/definitions/PatientModel"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "OK",
            "schema": {
              "type": "array",
              "items": {
                "$ref": "#/definitions/ScoreModel"
              }
            }
          },
          "201": {
            "description": "Created"
          },
          "401": {
            "description": "Unauthorized"
          },
          "403": {
            "description": "Forbidden"
          },
          "404": {
            "description": "Not Found"
          }
        }
      }
    }
  },
  "definitions": {
    "DiseaseMetaModel": {
      "type": "object",
      "properties": {
        "diseaseName": {
          "type": "string"
        },
        "id": {
          "type": "string"
        },
        "overallFrequency": {
          "type": "string",
          "enum": [
            "RARE",
            "LESS_COMMON",
            "COMMON",
            "VERY_COMMON"
          ]
        },
        "softConflictCategories": {
          "type": "array",
          "items": {
            "type": "string",
            "enum": [
              "GENDER",
              "AGE",
              "BORDER",
              "AMOUNT_OF_LESIONS",
              "AMOUNT_OF_LESIONS_MANY_FORMATION",
              "COLOR",
              "SITE"
            ]
          }
        }
      },
      "title": "DiseaseMetaModel"
    },
    "DiseaseModel": {
      "type": "object",
      "properties": {
        "age": {
          "type": "array",
          "items": {
            "type": "string",
            "enum": [
              "INFANT",
              "CHILD",
              "ADULT",
              "ELDER"
            ]
          }
        },
        "amountOfLesions": {
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "amountOfLesionsManyFormation": {
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "border": {
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "color": {
```

```
          "type": "object",
          "additionalProperties": {
            "type": "string"
          }
        },
        "diseaseName": {
          "type": "string"
        },
        "gender": {
          "type": "object",
          "additionalProperties": {
            "type": "number",
            "format": "double"
          }
        },
        "id": {
          "type": "string"
        },
        "overallFrequency": {
          "type": "string",
          "enum": [
            "RARE",
            "LESS_COMMON",
            "COMMON",
            "VERY_COMMON"
          ]
        },
        "site": {
          "type": "array",
          "items": {
            "type": "string",
            "enum": [
              "HEAD",
              "TORSO",
              "TORSO_BREAST",
              "TORSO_STOMACH",
              "ARMS",
              "ARMS_HANDS",
              "ARMS_HANDS_FINGERS",
              "LEGS",
              "LEGS_FEET",
              "LEGS_FEET_TOES",
              "LEGS_KNEES"
            ]
          }
        },
        "softConflictCategories": {
          "type": "array",
          "items": {
            "type": "string",
            "enum": [
              "GENDER",
              "AGE",
              "BORDER",
              "AMOUNT_OF_LESIONS",
              "AMOUNT_OF_LESIONS_MANY_FORMATION",
              "COLOR",
              "SITE"
            ]
          }
        }
      },
      "title": "DiseaseModel"
    },
    "PatientModel": {
      "type": "object",
      "properties": {
        "age": {
          "type": "string",
          "enum": [
            "INFANT",
            "CHILD",
            "ADULT",
            "ELDER"
          ]
        },
        "amountOfLesions": {
          "type": "string",
          "enum": [
            "SINGLE",
            "FEW",
            "MANY"
          ]
        },
        "amountOfLesionsManyFormation": {
          "type": "string",
          "enum": [
            "SCATTERED",
            "CUMULATIVE"
          ]
        },
        "border": {
          "type": "string",
          "enum": [
            "DELIMITED",
            "IRREGULAR"
          ]
        },
        "color": {
          "type": "string",
          "enum": [
            "RED",
            "WHITE",
            "BROWN",
            "BLACK"
          ]
        },
        "gender": {
          "type": "string",
          "enum": [
            "MALE",
            "FEMALE"
          ]
        },
        "site": {
          "type": "array",
          "items": {
            "type": "string",
            "enum": [
              "HEAD",
              "TORSO",
              "TORSO_BREAST",
              "TORSO_STOMACH",
              "ARMS",
              "ARMS_HANDS",
              "ARMS_HANDS_FINGERS",
              "LEGS",
              "LEGS_FEET",
              "LEGS_FEET_TOES",
              "LEGS_KNEES"
            ]
          }
        }
      },
      "title": "PatientModel"
    },
    "ScoreModel": {
      "type": "object",
      "properties": {
        "disease": {
          "$ref": "#/definitions/DiseaseMetaModel"
        },
        "explanation": {
          "type": "number",
          "format": "double"
        },
        "rank": {
          "type": "number",
          "format": "double"
        },
        "similarity": {
          "type": "number",
          "format": "double"
        }
      },
      "title": "ScoreModel"
    },
    "View": {
      "type": "object",
      "properties": {
        "contentType": {
          "type": "string"
        }
      },
      "title": "View"
    }
  }
}
```

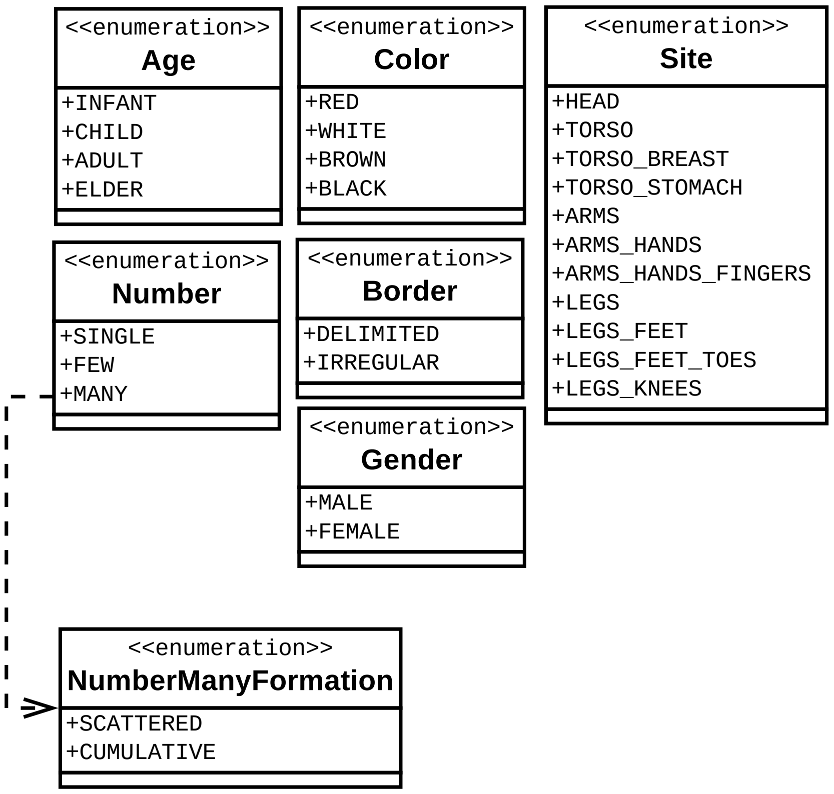


Figure A.2: Generated category enums for the running example. The dashed arrow represents the dependency the category *NumberManyFormation* has on the value “MANY” of the category *Number*. This dependency is also expressed through the name of the depending category.

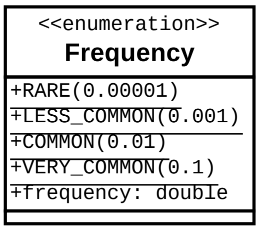


Figure A.3: Generated frequency enum for the running example.

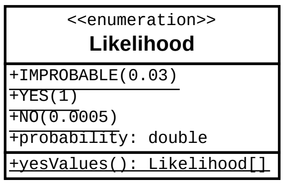


Figure A.4: Generated likelihood enum for the running example.

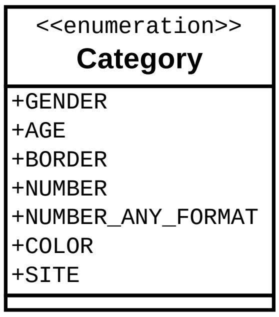


Figure A.5: Generated category enum for the running example.

List of Figures

List of Tables

Listings

Bibliography

[1] F. M. Amiri, A. Khadivar, and A. Dolatkhah, “A fuzzy expert system for response determining diagnosis and management movement impairments syndrome,” *International Journal of Business Information Systems*, vol. 24, no. 1, pp. 31–50, 2017.

[2] W. W. Melek and A. Sadeghian, “A theoretic framework for intelligent expert systems in medical encounter evaluation,” *Expert Systems*, vol. 26, pp. 82–99, 2009.

[3] K. Abudahab, D.-l. Xu, Y.-w. Chen, and A. T. Rules, “Generic Expert System and its Application in Knowledge Modelling and Inference,” in *Generic Expert System and Its Application in Knowledge Modelling and Inference*, pp. 1367 – 1372, 2013.

[4] S. S. Sikchi, M. S. Ali, and S. S. Sikchi, “Generic Medical Fuzzy Expert System for Diagnosis of Cardiac Diseases,” *International Journal of Computer Applications*, vol. 66, no. 13, pp. 35–44, 2013.

[5] H. A. Mohammadi Motlagh, B. Minaei Bidgoli, and A. A. Parvizi Fard, “Design and implementation of a web-based fuzzy expert system for diagnosing depressive disorder,” *Applied Intelligence*, vol. 48, pp. 1302–1313, may 2018.

[6] A. Soltani, T. Battikh, I. Jabri, and N. Lakhoua, “A new expert system based on fuzzy logic and image processing algorithms for early glaucoma diagnosis,” *Biomedical Signal Processing and Control*, vol. 40, pp. 366–377, 2018.

[7] A. A. S. Asl and M. H. F. Zarandi, “A type-2 fuzzy expert system for diagnosis of leukemia,” *Advances in Intelligent Systems and Computing*, vol. 648, pp. 52–60, 2018.

[8] M. N. Desai, V. Dahiya, and A. K. Singh, “Proposed Model for an Expert System for Diagnosing Degenerative Diseases – Using Digital Image Processing with Neural Network,” in *Information and Communication Technology for Intelligent Systems (ICTIS 2017) - Volume 1* (S. C. Satapathy and A. Joshi, eds.), (Cham), pp. 68–73, Springer International Publishing, 2018.

[9] S. B. Chaudhuri and M. Rahman, “Design of a Medical Expert System (MES) Based on Rough Set Theory for Detection of Cardiovascular Diseases,” *Advances in Intelligent Systems and Computing*, vol. 563, pp. 325–332, 2018.

[10] O. M. Alade, O. Y. Sowunmi, S. Misra, R. Maskeliūnas, and R. Damaševičius, "A Neural Network Based Expert System for the Diagnosis of Diabetes Mellitus," *Advances in Intelligent Systems and Computing*, vol. 724, pp. 14–22, 2018.

[11] E. Avuçlu and F. Ba¸sçiftçi, "Computer Methods and Programs in Biomedicine An expert system design to diagnose cancer by using a new method reduced rule base," *Computer Methods and Programs in Biomedicine*, vol. 157, pp. 113–120, 2018.

[12] C. P. C. Munaiseche, D. R. Kaparang, and P. T. D. Rompas, "An Expert System for Diagnosing Eye Diseases using Forward Chaining Method," in *IOP Conference Series: Materials Science and Engineering*, vol. 306, 2018.

[13] F. Ricci, L. Rokach, and B. Shapira, "Introduction to Recommender Systems Handbook," in *Recommender Systems Handbook*, pp. 1–35, Springer US, 2011.

[14] Z. Gantner, S. Rendle, C. Freudenthaler, and L. Schmidt-Thieme, "MyMediaLite : A Free Recommender System Library," in *Proceedings of the fifth ACM conference on Recommender systems*, (New York), pp. 305–308, ACM, 2011.

[15] S. Vargas, "Novelty and diversity enhancement and evaluation in recommender systems and information retrieval," in *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, (New York), pp. 1281–1281, ACM, 2014.

[16] G. Guo, J. Zhang, Z. Sun, and N. Yorke-smith, "LibRec : A Java Library for Recommender Systems," in *UMAP Workshops*, vol. 2, 2015.

[17] S. Inzunza and R. Juárez-Ramirez, "A Comprehensive Context-Aware Recommender System Framework," in *Studies in Systems, Decision and Control*, pp. 1–24, Springer, 2018.

[18] T. Hussein, T. Linder, W. Gaulke, and J. Ziegler, "Hybreed : A software framework for developing context-aware hybrid recommender systems," *User Modeling and User-Adapted Interaction*, vol. 24, no. 1, pp. 121–174, 2014.

[19] X. Ma, H. Li, J. Ma, Q. Jiang, S. Gao, N. Xi, and D. Lu, "APPLET : a privacy-preserving framework for location-aware recommender system," *Science China Information Sciences*, vol. 60, no. 9, 2017.

[20] W. Wang, H. Yin, S. Sadiq, L. Chen, M. Xie, and X. Zhou, "SPORE : A Sequential Personalized Spatial Item Recommender System," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 954–965, 2016.

[21] J. H. Chen, T. Podchiyska, and R. B. Altman, "OrderRex : clinical order decision support and outcome predictions by data-mining electronic medical records," *Journal of the American Medical Informatics Association*, vol. 23, no. 2, pp. 339–348, 2016.

[22] F. Gräßer, H. Malberg, S. Zaunseder, S. Beckert, and S. Abraham, "Neighborhood-based Collaborative Filtering for therapy decision support," in *CEUR Workshop Proceedings*, pp. 22–26, 2017.

[23] Q. Zhang, G. Zhang, J. Lu, and D. Wu, "A Framework of Hybrid Recommender System for Personalized Clinical Prescription," in *2015 10th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*, pp. 189–195, 2015.

[24] Y. Bao and X. Jiang, "An Intelligent Medicine Recommender System Framework," in *2016 IEEE 11th Conference on Industrial Electronics and Applications (ICIEA)*, pp. 1383–1388, 2016.

[25] N. Michalopoulos, G. E. Raptis, M. Mamalakis, C. Katsini, and A. Vigotsky, "A Personalised Monitoring and Recommendation Framework for Kinetic Dysfunctions : The Trendelenburg Gait," in *ACM International Conference Proceeding Series*, p. Article number a8, 2016.

[26] E. Riedl, "Dermtrainer – A novel decision support system for training and diagnosis in dermatology." url: `https://iktderzukunft.at/en/projects/dermtrainer.php` [Accessed on 2018-07-26].

[27] A. Masood and A. A. Al-Jumaily, "Computer aided diagnostic support system for skin cancer: A review of techniques and algorithms," *International Journal of Biomedical Imaging*, vol. 2013, pp. 1–22, 2013.

[28] S. Robertson, "Understanding inverse document frequency : on theoretical arguments for IDF," *Journal of Documentation*, vol. 60, no. 5, pp. 503–520, 2004.

[29] C. D. Manning, P. Raghavan, and H. Schütze, *Scoring, term weighting, and the vector space model*, pp. 100–123. Cambridge University Press, 2008.

[30] T. Roelleke and J. Wang, "TF-IDF Uncovered : A Study of Theories and Probabilities," in *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, (Singapore), pp. 435–442, 2008.

[31] JHipster, "JDL-Studio." url: `https://start.jhipster.tech/jdl--studio/` [Accessed on 2018-08-02].