Diploma Thesis

Höhere Technische Bundeslehranstalt Leonding

Abteilung für Informatik

Matching Personal Profiles and Spare Time Activities Based on Big Data Analysis

Submitted by: **Markus Geilehner, 5AHIF**
Simon Gutenbrunner, 5AHIF
Martin Singer, 5AHIF
Date: **April 4, 2018**
Supervisor: **Peter Bauer**

Declaration of Academic Honesty

Hereby, I declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

This paper has neither been previously submitted to another authority nor has it been published yet.

Leonding, April 4, 2018

Markus Geilehner, Simon Gutenbrunner, Martin Singer

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorgelegte Diplomarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Gedanken, die aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Leonding, am 4. April 2018

Markus Geilehner, Simon Gutenbrunner, Martin Singer

Abstract

Currently, searching for spare time activities is a task which has to poll a big bunch of different data sources and while browsing these the result a user will get is probably not fitting for him/her. This is the reason why we created *Ennui* as an iOS and Android application for young people to support them in finding opportunities to spend their free time. The goal was to build an application which makes it easy to find spare time activities like events, offers or playable games and which gives recommendations based on a users behaviour.

This thesis describes:

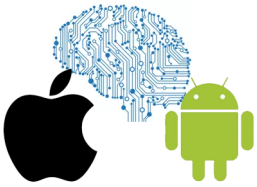
- iOS Application - Built in XCode 9 with Swift 4
- Android Application - Built in Android Studio with JAVA
- Three big data algorithms - To get recommendations based on the users behaviour
- Test Environment - To test these big data algorithms and finding the best one for our situation.

The above mentioned big data algorithms are

- C4.5
- CART
- CHAID

Moreover to understand the mentioned algorithms the authors have taken a dive into machine learning, testing results and how development in iOS and Android has been done.

The final result contains the two applications with recommendations based on a users behaviour determined by the C4.5 algorithm.



Zusammenfassung

Wenn derzeit jemand versucht Freizeitaktivitäten zu finden, muss er / sie viele verschiedene Quellen durchsuchen und möglicherweise ist das Ergebnis nicht einmal passend für den Benutzer. Dies ist der Grund wieso wir "Ennui" als iOS und Android Applikation erstellt haben, um jungen Leuten die Aufgabe, Aktivitäten in deren Freizeit zu finden, abzunehmen. Das Ziel dieser Anwendung war es, das Finden von Events, Angeboten oder Spielen möglichst einfach zu gestalten und basierend auf dem User ihm / ihr auch Vorschläge für solche zu liefern.

Diese Diplomarbeit beschreibt:

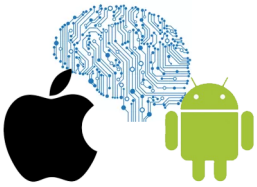
- iOS Applikation - Entwickelt in XCode 9 mit Swift 4
- Android Applikation - Entwickelt in Android Studio mit JAVA
- Drei Big Data Algorithmen - Um empfehlen auf Basis des Benutzers zu ermitteln
- Test Environment - Um die drei Big Data Algorithmen zu testen und um den besten dieser zu finden.

Die oben genannten Algorithmen sind

- C4.5
- CART
- CHAID

Um die genannten Algorithmen zu verstehen sind die Autoren noch auf Machine Learning eingegangen, weiters in die Test Ergebnisse und wie entwickeln in iOS und Android funktioniert.

Das letztendliche Ergebnis beinhaltet die beiden Applikationen mit Empfehlungen basierend auf dem User Verhalten ermittelt durch den C4.5 Algorithmus.



Acknowledgments

We would like to express our deepest gratitude to Prof. Peter Bauer who provided helpful comments and suggestions. We would also like to express our gratitude to our families for their moral support and warm encouragements. We also want to give thanks specifically to Spotify for their support in providing music for us while writing.

Contents

Chapter 1

Introduction

1.1 Initial Situation

This thesis builds upon an already existing school project developed by the authors of this thesis, which is called *Ennui*. Ennui is a web application with the goal of finding spare time activities for the user. These spare time activities are presented to the user in three categories: events, offers and games, where events represent activities which have a specific time and location, such as the *Linzer Pflasterspektakel* or the *Oktoberfest*. Unlike events, offers have no time restriction, but their location is still specific. Note that an offer still can have opening hours. Popular examples for offers include cinemas, bowling centers or public baths. Games are not restricted by any time or location and can generally be executed anywhere and anytime, however it has to be considered that some games require certain items. Popular examples for games are *Völkerball* or *Schnopsn*. For a more detailed description of events, offers and games see section 1.3

Unfortunately Ennui is only available as a website, and although it provides multiple options for restricting the large amount of events presented, the event results are very general for all users.

1.2 Overview

The thesis tries to build up on that project and extends it in several ways. First of all it extends Ennui with two mobile clients, one for Android and one for iOS. Additionally each one of the authors developed a machine learning algorithm to determine which future events fit the best to a certain user, based on his past event preferences. Also a test environment was developed to determine the best algorithm. The chosen algorithm then got implemented in the already established Ennui backend. Goal of the thesis is on one hand the completion and successful implementation of the mobile apps for Android and iOS and on the other hand writing three fully functioning machine learning and a test environment to determine the best fitting one for the Ennui environment.

1.3 Basic Terminology

1.3.1 Event

An *Event* represents an occurrence in the future at a defined location, starting- and ending time. Some popular examples are concerts, fairs, or a sports game.

An Event holds the following properties:

- **Name:** serves as an unique identifier of an event
- **Description:** holds all of the additional information necessary for the user
- **Category:** holds one or more values from the following list:
  - Party
  - Music
  - Art
  - Games
  - Food
  - Comedy
  - Literature
  - Health
  - Shopping
  - Home-Garden
  - Sports
  - Theater
  - Others
- **Location:** holds information about longitude, latitude and name of the location if existent
- **Start time:** timestamp of the official start time
- **End Time:** timestamp of the official end time
- **Host:** individual or company which hosts the event
- **Cover Url:** link to the events cover image
- **Website Url:** link to the website of the event if existent

1.3.2 Offer

An *Offer* represents an establishment that usually offers spare time activities for the user. The difference between an Offer and an Event is, that the Offer is not limited to a duration and therefore does not have a start/end point. Nevertheless they still may have opening hours. Popular examples of Offers are Bowling Centers, Pool Halls and Cinemas.

An offer holds the following properties:

- **Name:** serves as an unique identifier of an offer
- **Description:** holds all of the additional information necessary for the user
- **Category:** holds exactly one value from the following list:
  - climbing
  - bowling
  - cinema
  - cart
  - billiard
  - lasertag
  - segway
  - museum
  - paintball
  - casino
  - trampolin
  - open air pool
  - indoor pool
  - thermal bath
  - high rope course
  - ice skating
  - escape the room
  - skiing
  - water skiing
  - swimming
- **Location:** holds information about longitude, latitude and name of the location if existent
- **Opening Hours:** holds the duration during which the offer is useable for each day of the week
- **Rating:** represents the offers rating on Google Maps
- **Reviews:** holds the last 5 reviews left by users via Google Maps

1.3.3 Game

A *Game* represents an activity for one or more users. Just like the Offer a Game is not dependent to a specific duration but in addition to that it is also not limited to a specific location. Nevertheless there still might be additional requirements/recommendations for ideal conditions. A Game holds the following properties:

- **Name:** serves as an unique identifier of a game
- **Category:** holds one of the favorite values:
  - drinking games
  - card games
  - dice games
  - board games
  - outdoor games
  - ball games
- **Description:** text to describe the game as short but as precise as possible
- **Instructions:** text to describe the process of playing the game and listing its components and/or additional requirements for location, time, etc. if existing
- **Minimum Player Count:** integer which represents the minimum player count needed for the game
- **Maximum Player Count:** integer which represents the maximum player count possible for the game

1.3.4 Machine Learning

Machine learning in computer science means that a program or an algorithm "learns" to perform a specific task without being explicitly programmed to do so. Machine learning is an important topic in artificial intelligence and data science.

1.3.5 AlgorithmDTO

For the event attributes provided by Ennui, we considered the following to be relevant for calculating the decision tree. The reason why a attribute is chosen will be given at each attribute. Relevant information for calculating the best fitting spare time activities were the following fields.

EventID

The Event-ID was used as a return parameter. After calculating the best fitting spare time activities, the IDs of events with the highest confidence factor were returned to be displayed by the GUI.

Day

The weekday was provided for finding preferences concerning the date in general. If the user for example starts to work later every Thursday, events that happen on Wednesday will be much more attractive for him then events on Mondays.

Start Time

The start time attribute was used for similar reasons as the day attribute. With this information, the algorithms can filter for a time frame. To make this more clear: If a user of Ennui for example works starting from 8 a.m. to 5 p.m. the user will not be able to attend events in this time frame. Start time is a relevant information to prevent recommending events for this time.

Location

To find preferences towards locations this attribute was implemented. Users often have their go-to cities when planning spare time activities, this may be caused by different reasons, for example the good pubic transport connection or friends living nearby.

Operator

The operator attribute shows the company or person, that organised the event. As many companies have specified in special spare time activity types like music concerts or parties, users may want to attend events from the same operator they have already made good experiences with.

Category

The category tag is provided as an information from the Facebook API and is also used in the Ennui frontend, where the user is able to filter for categories to refine his search for events. This attribute can be one of the following values:

- Party: Events in clubs and festivals
- Music: Concerts for all music genres
- Art: Art exhibitions, ballet and many more events
- Literature: Readings, book clubs, ...
- Comedy: Comedy shows from all sorts of artists.
- Food: Everything food-related, food-festivals, cooking classes, ...
- Games: Short adventures like 'Exit the Room'
- Health: Everything health-related, workshops, classes, ...

- Shopping: Special shopping events, for example christmas fairs
- Home: Events and exhibitions about furniture, home design, ...
- Sport: Sport-related events: football games, marathons, triathlons,
- Theatre: Theatres, musicals, operas
- Others: Events that could not be classified for one of the further categories.

Part of the week

To generalize the `day` attribute, the `part of the week` attribute was generated by the developer. This just splits the week into two types of days: weekend and workdays. Speaking from a normal working week, the split happens at Friday 2 p.m.. From this point to Sunday all events get the `weekend` tag, the rest of the week is tagged with `workday`. This is needed for two reasons. First to give the algorithms a hint for making the split at weekend and workday and second, to shrink tree sizes dramatically. This can be explained with the fact that `part of the week` only has two attributes and `day` seven. Most of the time, users only differ their activities between workdays and weekends. With `part of the week` this split only leads to two leafs, instead of seven leafs with the day attribute.

Additional tag

With the `additional-tag` attribute, the `category` attribute gets more refined. This is especially necessary for more general categories like `Sports` or `Music`. If we take "Sports" as an example, without any other knowledge, a 70 year old frail man, that just likes going to the football-stadium to cheer for his favourite football-club, would get recommendations for other sport events like a triathlon.

Each category has a different number of additional tags, that just specifies the section itself. To continue with the first example, in the `Sports` section we have to differ between the forms of sport and active/passive sports. The last split is the most important one, not everyone that likes watching live sports (=passive) also wants to do active sports, which means doing the sport himself.

Different to the other attributes, the `additional-tag` attribute is not provided by Facebook. It gets generated by scanning the title, as well as the description of the event for keywords. If a activity with the category `sports` for example has the words `Fussball` or `Kicker` in her title or description, we can nearly guarantee that the event is an football-related one. As the biggest distinction provided, `additional-tag` is a key attribute for splitting the decision trees.

1.4 Related Work and Projects

This section includes a survey of other work around the area of this thesis. The following list of related work includes:

- C4.5 Programs for Machine Learning C4.5: Programs for Machine Learning was written in 1993 by Ross Quinlan [Qui14]. It covers the functionality of the C4.5 algorithm and its implementation in C.
- Classification and Regression Trees Classification and Regression Trees was written in 1984 by Leo Breiman, Jerome Friedman, Charles J. Stone and R.A. Olshen [BFOS83]. This book introduces the concept of the CART algorithm and decision tree classifiers.
- An Exploratory Technique for Investigating Large Quantities of Categorical Data An Exploratory Technique for Investigating Large Quantities of Categorical Data was written by G.V. Krass in 1980 [Kas80] and covers the functionality of the CHAID algorithm. This work is was published in Vol. 29, No. 2 of *The Journal of the Royal Statistical Society, Series C (Applied Statistics)* [otRSS18], which is a journal about statistical applications and problems.

1.5 Structure of this Thesis

This thesis is structured in nine chapters, including this one. *Chapter 2* serves to give an insight about machine learning and to give the user a profound knowledge for an easier understanding of the following chapter. Chapter 2 also gives a brief overview about different machine learning algorithms. The following *Chapter 3* builds up on the second one knowledge wise. It gives a detailed insight on the thee developed machine learning algorithms called C4.5, CART and CHAID and explains their features and the implementation into the backend. In *Chapter 4* the implementation of the test environment, a web application for applying, testing and comparing the three big data algorithms is explained. The chapter starts with a short introduction of the used frameworks. Afterwards the overall structure of the test environment is described and the database gets explained. Finally a somewhat elaborate list of features including code examples, where necessary, is given. For finding the best fitting algorithm to implement into the backend of Ennui, a study of comparing them was realized which will be covered in *Chapter 5*. This study consisted of a two-step procedure. Firstly, all three algorithms were tested in the test environment, as explained in *Chapter 4*. With the information gain of this step, the algorithms got refined and implemented into the running system Ennui. Tested with real event data and real users, the best algorithm was found. *Chapter 6* and *7* cover the implementations of the mobile clients. They cover an explanation of each GUI element, explain all the features and give an insight about the code-wise implementation. The last 2 chapters serve as a wrap-up of the thesis. *Chapter 8* lists all the technologies used for development and *Chapter 9* summarizes the whole thesis.

Chapter 2

Machine Learning

2.1 Introduction

This chapter is a small dive into the matter of machine learning. It is full with basic information to understand how the algorithms in chapter three are working. After a start with some general stuff about machine learning, a description of types of machine learning and common machine learning algorithms where some algorithms are explained shortly follows. Last but not least Weka is explained (see 5.1).

2.2 Classification

Classification can be described with assigning observed objects to one category of a set of categories. This categorization is based on a collection of already categorized objects called a training set.

training set In statistical classification a training set is a list of objects. This objects have to be of the same type as the to-be classified objects, with the only difference that the elements of the training set already are classified. A classifier uses elements similar to a to-be classified object in its training set as a reference to classify this certain object. More about this topic can be found in 5.2.1.

test set to determine how well a training set was adapted, a test set is used. It contains a list of non-classified objects, which simulate the to-be classified objects of the runtime.

For example a classification of an email could be determining whether it is `spam` or `genuine`. For more information about statistical classification see [Wik18c].

2.2.1 Over- and Underfitting

The biggest cause for a weak classification accuracy of a classifier is overfitting or underfitting. This section covers both problems and gives ideas on how to counteract those problems. To see the source of this information see [Bro16].

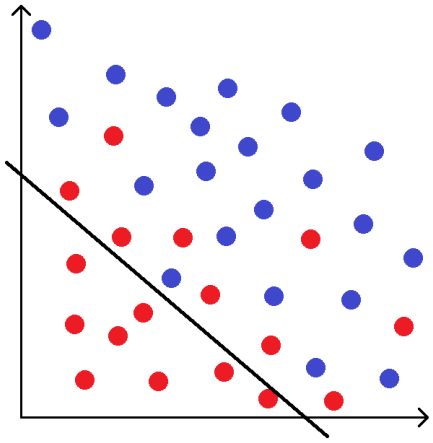


Figure 2.1: An underfitting clasification

Underfitting

Underfitting is when a classifier fails to give a clear and distinctive classification on the data in the training set. This can be caused by not considering enough attributes from the training set, so including more attributes decreases the chance of an underfitting algorithm. All in all underfitting can be described as failing to interpret the training set and therefore fails to do a clear classification.

Overfitting

Unlike an underfitting classification, an overfitting one performs a very distinctive classification on the training set. In fact an overfit means that the training set is adapted so well that an algorithm fails to generalize new data. This can be caused by considering too many attributes for the classification. All in all overfitting can be described as relying too much on the training set and therefore not being able to conclude new classifications.

2.2.2 Classifiers

When talking about statistical classification a classifier is an algorithm that implements such classification. It categorizes objects based on a training set consisting of already classified objects of the same object type. So in the case of this diploma thesis, it includes 3 tree based classifiers, classifying events by being either recommended to the user or not.

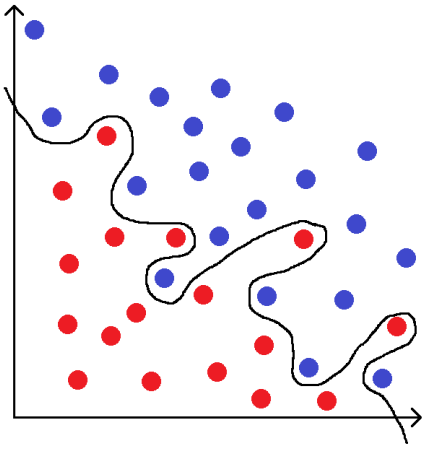


Figure 2.2: An overfitting classification

2.2.3 Clustering Classifiers

One popular type of classifiers are clustering classifiers. The goal of a clustering classifier is to assign multiple objects in a given set of data points to certain groups. Such groups are called clusters. Ideally data points of the same cluster share similar traits or attribute values whereas data points of different clusters ideally have highly distinguishable traits / attribute values. The topic of clustering based classifiers obviously can be extended by a lot but since none of the algorithms covered in this thesis are clustering-based this will not be deepened. The information of this section is based on [Wik18b] Popular classifiers based on clustering include:

- k-means
- Mean-Shift Clustering
- DBSCAN (=**D**ensity-**B**ased **S**patial **C**lustering of **A**pplications with **N**oise)
- Expectation-Maximization Clustering using Gaussian Mixture Models
- Agglomerative Hierachical Clustering

2.2.4 Decision Tree Classifiers

What is a Decision Tree?

Another type of classifiers are decision trees. Decision trees are used for classifying objects. In a decision tree each split represents a question about a to-be classified object and each terminal node a certain classification of the object. For example a decision tree can be used for determining how likely a bank client will pay back his debts. A

decision tree for this example would contain questions about the age of the client, the total amount of his dept or the martial status of the client. Depending on the attributes of the client he ends up in one of the terminal nodes after going through the questions which indicate the reliability of the solvency of the client. Popular decision-tree based classifiers include:

- ID3
- C4.5
- CART
- CHAID
- MARS

Trees in Programming Theory

In programming theory a tree is an abstract data structure that consists of multiple elements called nodes. The data set which is held by the tree is distributed between the nodes of a tree, with one node usually holding one specific value. The main characteristics of a tree is that each node of the tree has zero or more child nodes with each child node being the root of a separate subtree. In a tree there can only be one node that does not have a parent which is called a trees root. The parent-child relation between the nodes is one directional and follows a clear level based hierarchy. This hierarchy results in each node being associated with a certain level. The level of a node indicates its distance to the root node, meaning the root node has a level of 1, its direct descendants are level 2, their direct descendants level 3 and so on. This level hierarchy also means that a node can not be a parent of one of its ancestors and therefor makes it impossible for the hierarchy chain to form a circle. Another attribute of a node is its rank which defines the amount of nodes that branch off of it. If the maximum rank of each node of a tree does not exceed two, the tree is considered a binary tree. If a node has no children it is considered a terminal node / leaf of the tree. A visualisation of a tree can be seen in figure 2.3. For more information about trees see [Wik18d] and [Wik18e]

Traversing Trees

With this rather unique arrangement of items in a tree there are several ways of stepping through its items. There are several ways in which a tree can be traversed. The following methods of traversing can only be applied on binary trees which include the following:

Depth-First Search In depth-first search all descendants of a certain node have to be traversed first before traversing to its sibling. Depth-first search can be done in the following orders:

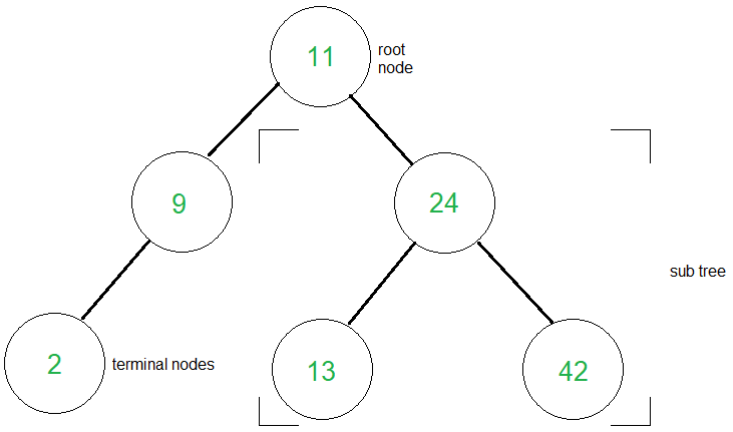


Figure 2.3: Binary Tree Example

- **Pre-order:** in pre-order the data of the root is displayed first. Then the subtrees of a node are recursively traversed from left to right. A visualization can be seen in figure 2.4
- **In-order:** with in-order traversing the left subtree is traversed recursively first. If the node has no more left subtree the data of the node is displayed. After displaying the data the right sub-tree is traversed recursively. A visualization can be seen in figure 2.5
- **Post-order:** when traversing trees in post-order the left subtree is traversed first. When a node does not have a left sub-tree it then traverses its right subtree. If the node does not have any non traversed subtree the data of the node is displayed. A visualization can be seen in figure 2.6

Breath-First Search In breath-first search the nodes of a tree are traversed in a level order, where the value of each node of a level is outputted first before going to the next level.

How to Build a Decision Tree

A decision tree is generated from a so called training set. A very simple way for building a decision tree out of a training set would be to split up the elements of the training set into all their possible values, where each level of the tree represents one attribute. The only attribute which is not split into its possible values is the one we want the objects to be classified by. So to come back to the bank client example if one wants to know whether a client is rather male or female the tree splits all the attributes of the client except for the gender. When determining the solvency of the client we split up all the

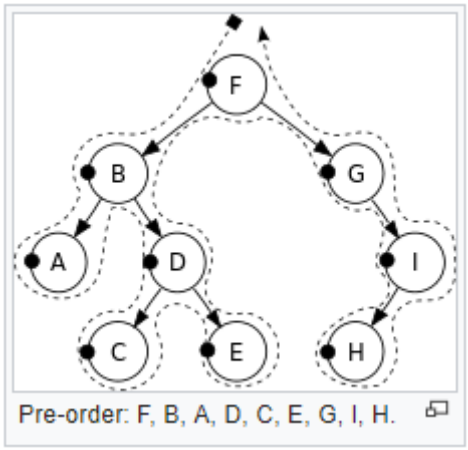


Figure 2.4: Pre-order traversal

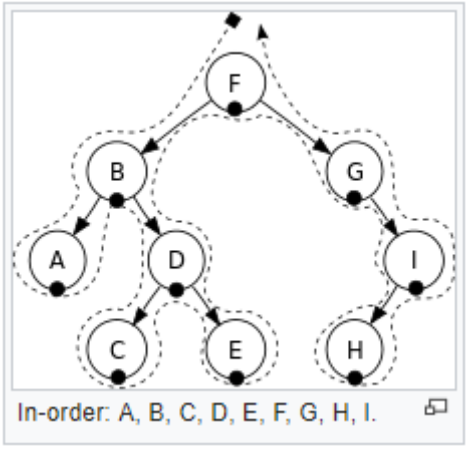


Figure 2.5: In-order traversal

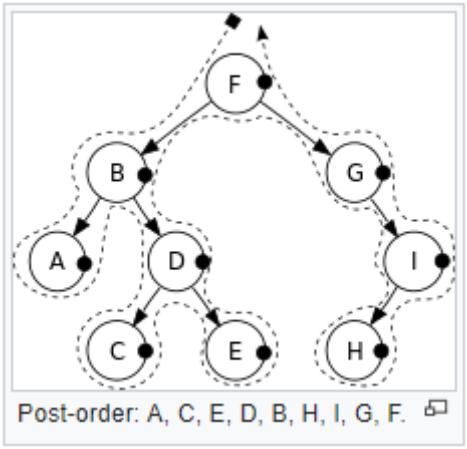


Figure 2.6: Post-order traversal

attributes of the training data except whether the client payed back his dept or not. Each terminal node represents one certain combination of attributes which means every object of the training set can be associated with one terminal node of the decision tree. With each terminal node representing a certain subset of the training set, it is possible to calculate a confidence level for each terminal node. This confidence level indicates how many percent of the subset meet with a certain value of the classified attribute.

How to Use a Decision Tree

After the decision tree is generated we can use it to classify new objects. For that we go down the questions of the decision tree, so we end up at one terminal node of the tree. We can now assume the confidence level of the node we calculated when building the tree to classify the object.

2.3 Regression

Next to the classification there is *Regression*, used for predicting continuous responses, which means predicting for instance the trend in market prices or weather change in the next few days.

2.4 Types Of Machine Learning

2.4.1 General

In machine learning one has to differ between three kinds of learning methods, namely *Supervised Learning*, *Unsupervised Learning* and *Reinforcement Learning*. Supervised and unsupervised learning are the biggest of them and the difference between those types

is that in *Supervised Learning* the goal is to predict the future while in *Unsupervised Learning* the aim is to understand the existing data. In *Reinforcement Learning* (see [Mar17]) it is not directly about understanding data or predicting something, it is about learning what action a software agent should or can do and what action would be best for a specific situation. More about those three types is explained in the following sections.

2.4.2 Supervised Learning

To be able to predict the future we need existing data. In *Supervised Learning* data is already existing and each data point consists of an input and an output, expressed in mathematics the formula for each data point would be:

$$f(x) = y$$

where x is the input and y the output ('label') of the input x. Based on the known data and after training, a model should be able to predict the label y of a new input x'. Training in this case means that the model gets fed with already existing sample data where for some x the corresponding y is already given, i.e. the data is labelled.

As an example imagine a bunch of photos with bodies on it and a model has to recognize or learn which pixel x is which body part y (see figure 2.7). After training this model it is able to predict which pixel x is which body part y on a new photo. One possible thought of the model can be to see that the pixel x is near the right corner ergo the model can label it as 'the right leg' which is the output y.

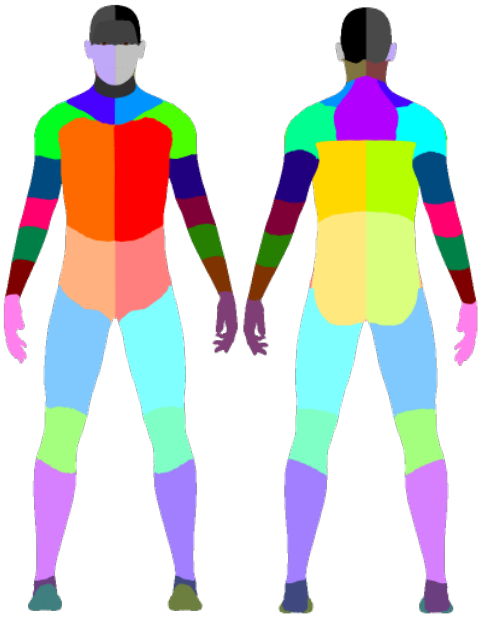


Figure 2.7: Body Parts (Source: [A. 16])

2.4.3 Unsupervised Learning

As already mentioned in 2.4.1 *Unsupervised Learning* is about understanding existing data or in other words explore hidden structures in data. Different to the Supervised Learning explained before the data which has to be analysed is not labelled and there is no data to fall back on. Unsupervised Learning is used in segmentations like image or market segmentation, the goal is to group the data by similarities which is equal to a cluster analysis. Clustering can also be used to find new groups in data for example in an image to detect new structures. Another example for the usage of clustering is when we want to train a model with data, but the data is to big and each data point has over thousands of attributes, then we have to compress the data but we do not want to lose important information. Therefore the goal is to minimize the data, remove irrelevant data and summarize the important ones which are similar.

2.4.4 Reinforcement Learning

Next to the other two methods of learning *Reinforcement Learning* is not about understanding or predicting data, it is used to learn which action in a given context is a good or a bad idea. Imagine an agent or a robot which has to fulfil a specific task, the task is to put out a blaze and the agent has two options, option one is to run into the fire and try to blow it out or option two, going to the tap, fill a bucket with water and empty it over the fire. (see figure 2.8) When taking the right decision, the agent gets a reward, the *reinforcement signal* or a penalty when the decision is wrong.

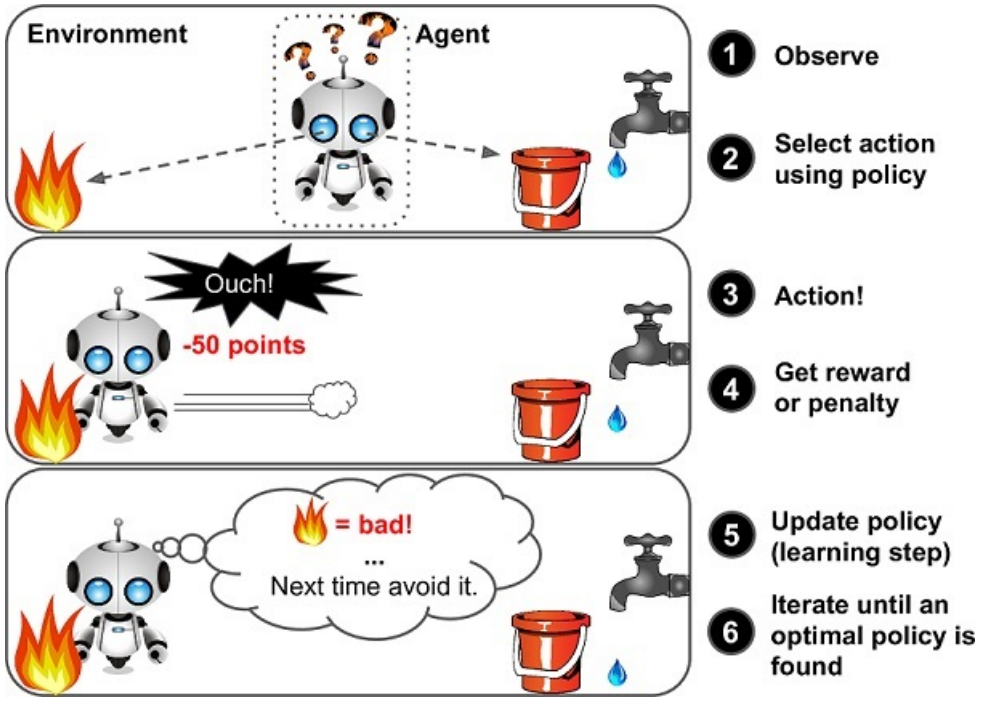


Figure 2.8: Roboter Task [Mar17]

2.5 Common Machine Learning Algorithms

2.5.1 General

In the following sections the most common machine learning algorithms will be explained shortly. Those algorithms can be applied on almost any data problem. For an abstract overview read more on [Sun17a] which is also the source of some information written later.

2.5.2 Linear Regression

It is used to estimate real values for example cost of a house or total sales based on continuous variables. We establish a relationship between independent and depended variables by fitting a best line. This line is called regression line and represented in a mathematical equation it would look like the following one.

$$Y = a * X + b$$

Where

- Y is the dependent variable

- a the slope
- X the independent variable and
- b the intercept.

These coefficients a and b are derived based on minimizing the sum of squared difference of distance between data points and regression line.

To understand how this process works, imagine asking a child to estimate the weights of all the other children around without asking them. The child will visually analyse each one with their height and width and then it will say how heavy each one is. Combining those visible parameters like height and width is *Linear Regression*.

In the following figure (2.9) the best fit line or the regression line is displayed. Now it is possible to calculate the weight for an arbitrarily given height. The function for this would be

$$f(x) = 0.2811x \cdot 13.9$$

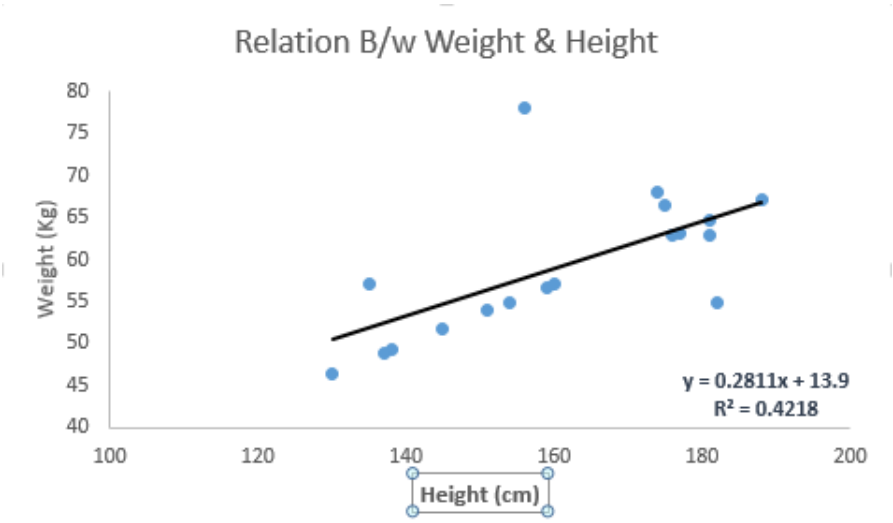


Figure 2.9: Linear Regression For Finding The Weight By A Given Height Of A Person (Source: [Sun17a])

Furthermore for linear regression there are two types, the *Simple Linear Regression* and *Multiple Linear Regression*. The first one is characterized by one independent variable whereas the second one has multiple independent variables ($v > 1$) as the name suggests.

2.5.3 Logistic Regression

First of all *Logistic Regression* is an algorithm for classification and not for regression. It is a type of regression that predicts the probability of occurrence of an event by

fitting data to a logistic function. The logistic regression makes use of several predictor variables that may be either numerical or categorical. Since, it predicts the probability, its output value lies between zero (0) and one (1). And that is the big difference to the linear regression, there the output can exceed the range between zero (0) and one (1) (see figure 2.10). Read more about the difference at [Mac17].

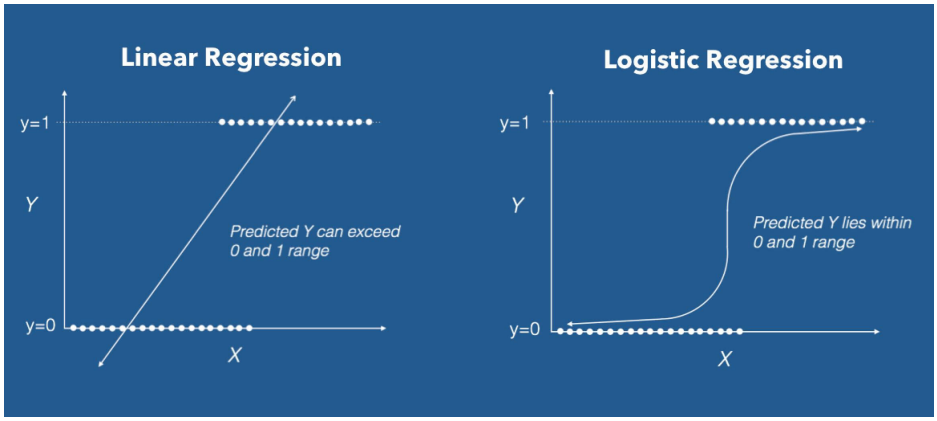


Figure 2.10: Linear Versus Logical Regression [Mac17]

Some good examples where logistic regression is used are:

- Spam Detection - Predicting if an email is spam or not
- Marketing - Predicting if a given user will buy an insurance product or not
- Health - Predicting if a given person will get a heart attack or not

To get the odds if an event is happening or not, or an email is spam or not the simplified calculation is to divide "the probability of event occurrence" by "the probability of not event occurrence".

$$\text{ODDS} = \frac{p}{1-p}$$

Afterwards we take a logarithm of the result

$$ln(\text{ODDS}) = ln(\frac{p}{1-p})$$

and the outcome is modelled as a linear combination of the predictor variables.

$$Y = ln(\frac{p}{1-p}) = \beta_0 + \beta_1 \cdot X_1 + \beta_2 \cdot X_2 + ... + \beta_n \cdot X_n$$

Above, p is the probability of presence of the characteristic of interest and β_n are the coefficients. It chooses parameters that maximize the likelihood of observing the sample values rather than that minimize the sum of squared errors.

2.5.4 SVM - Support Vector Machines

Support Vector Machines can be used for both classification and regression problems, but is mostly used for classification. In Support Vector Machines each data item is plotted as a point in an n-dimensional space, where n is the number of features for the given problem, with each value of a feature being a value of a particular coordinate. After plotting these points, we are able to perform the classification by finding the hyper-plane that differentiates the two classes. Note: There can also be more possibilities for placing a hyper plane.

For example in figure 2.11 the $y - axis$ stands for the length of a persons hair in centimetres and the $x - axis$ for height in centimetres. Imagine the green points as male and the blues ones as females. Through these parameters we can identify if the person is either a male or a female. When placing the hyper plane through the diagram we have classified those points. Also visible in figure 2.11 there are three ways (as an example) to place the hyper plane (of course there can be infinite), the best one is the green one because it is the farthest of the nearest vector point.

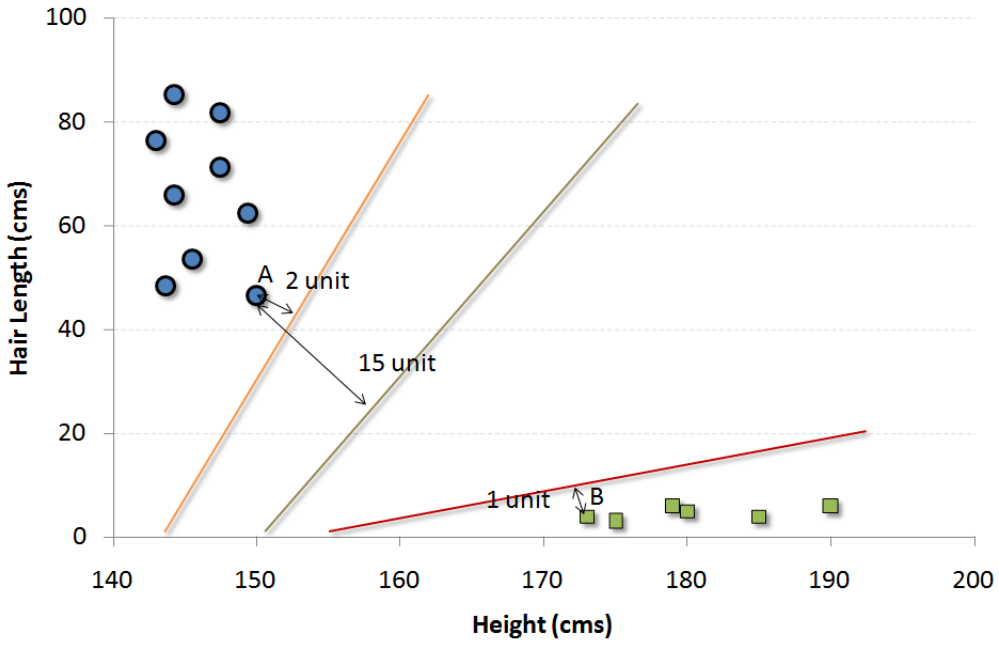


Figure 2.11: Support Vector Machines Hyper Plane

Read more about *SVM* and how to implement it at [Sun17b].

Table 2.1: Weather Sample Data

Weather	**Playable (Yes/No)**
Sunny	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Overcast	Yes
Rainy	No
Rainy	No
Sunny	Yes
Rainy	Yes
Sunny	No
Overcast	Yes
Overcast	Yes
Rainy	No

2.5.5 Naive Bayes

Naive Bayes is a classification technique based on the *Bayes Theorem* [Ric18] which is pretty useful for very large data sets. It assumes the independence between predictors. That means that the *Naive Bayes Classifier* assumes that any feature in a class is independent or unrelated to other features in this class.

The before mentioned *Bayes Theorem* is a way for calculating the posterior probability $P(c|x)$ from $P(c)$ and $P(x|c)$. This is possible with the following equation.

$$P(c|x) = \frac{P(x|c) \cdot P(c)}{P(x)}$$

where

- $P(c|x)$ is the posterior probability of class (target) given predictor (attribute)
- $P(c)$ is the prior probability of class
- $P(x|c)$ is the likelihood which is the probability of predictor given class
- $P(x)$ is the prior probability of predictor

Lets assume we want to classify if players of a soccer match will play or not based on the following weather data. Firstly we have to convert the data in the table above to a frequency table which will look like the following, to work easier with the data provided. As it is visible in the table above the total number of data provided is 14. We can now calculate the probability for each class which is calculated in the next table. Now we have prepared all the data and we can start with the calculation of the posterior

Table 2.2: Converted Sample Data Frequency Table

Weather	**Playable No**	**Playable Yes**
Overcast	0	4
Rainy	3	2
Sunny	2	3
Total	5	9

Table 2.3: Sample Data with Probability

Weather	**Playable No**	**Playable Yes**	**Probability**
Overcast	0	4	$\frac{4}{14} = 0.29$
Rainy	3	2	$\frac{5}{14} = 0.36$
Sunny	2	3	$\frac{5}{14} = 0.36$
Total	5	9	
	$\frac{5}{14} = 0.36$	$\frac{9}{14} = 0.64$	

probability by using the before mentioned *Bayes Theorem* equation. The class with the highest posterior probability is the outcome of the prediction. Lets calculate if the players will play when the weather is sunny.

$$P(\text{Yes}|\text{Sunny}) = \frac{P(\text{Sunny}|\text{Yes}) \cdot P(\text{Yes})}{P(\text{Sunny})}$$

The probability of $P(\text{Sunny}|\text{Yes})$ is $\frac{3}{9} = 0.33$ and $P(\text{Yes})$ is $\frac{9}{14} = 0.64$. Now applied to the equation the result would be:

$$P(\text{Yes}|\text{Sunny}) = \frac{0.33 \cdot 0.64}{0.36} = 0.60$$

which has higher probability. *Naive Bayes* uses a similar method to predict the probability of different classes based on various attributes.

2.5.6 kNN k - Nearest Neighbours

Again *kNN* can also be used for classification and regression, but is mostly used for classification problems. It stores all available cases and classifies new cases by a majority vote of its k neighbours. The case being assigned to the class is most common amongst its k nearest neighbours measured by a few distance functions like Euclidean, Manhattan, Minkowski and Hamming. While the first three are used for continuous functions the last one *Hamming* is used for categorical variables. For better understanding see figure 2.12.

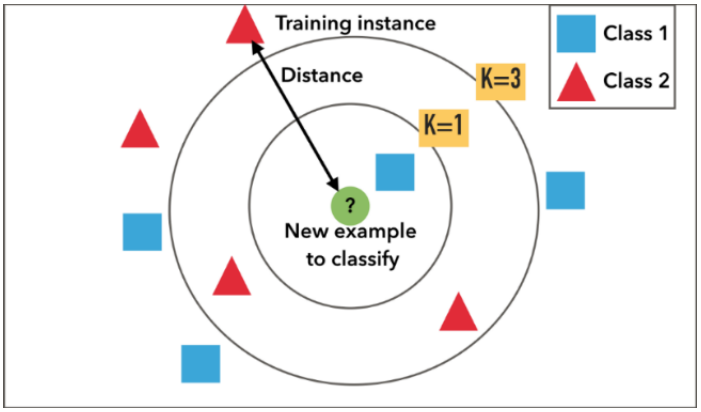


Figure 2.12: kNN (k - Nearest Neighbours) [Bro17]

The test object (green one) should be classified either to class one or two. If the result of the distance calculation is $k = 3$, then the green object will be assigned to the second class because in the inner circle there are two of the red triangles and only one blue square. If $k = 5$ for instance the sample object would be assigned to the blue squared because there are more blue squares than red triangles. As now perceptible the output of a *kNN algorithm* is a class membership.

2.5.7 K-Means

It is a type of *unsupervised algorithm* (see 2.4.3) which solves the clustering problem. Its procedure is a simple way to classify a data set through a certain number of clusters k. The data points inside a cluster are homogeneous and heterogeneous to peer groups. In simple terms *K-Means* decipher how many clusters are present in a given data set by looking on it's shape. For an example see figure 2.13, the black circles are the clusters which have been deciphered out of a bunch of data points.

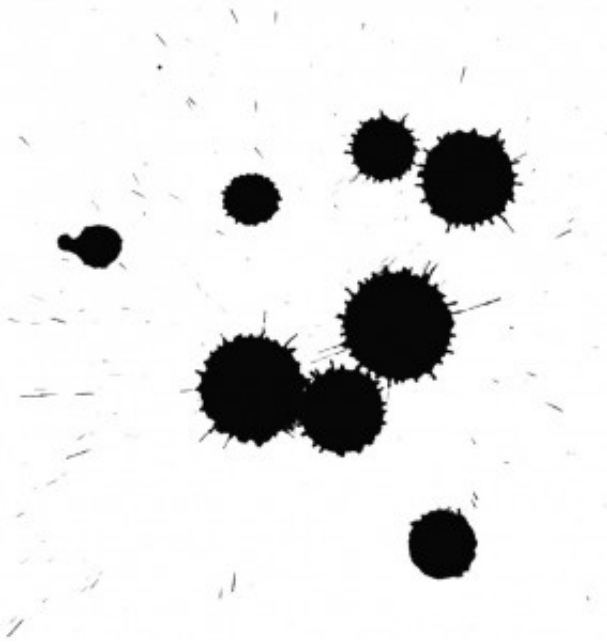


Figure 2.13: kNN (K-Means)

How *K-Means* forms a cluster or more, visible in figure 2.13:

1. It picks k data points for each cluster (also known as centroids)
2. Each data point forms a cluster with the closest centroids
3. Then *K-Means* finds the centroid of each cluster based on existing cluster members.
4. After the algorithm has new centroids, step two (2) and three (3) are repeated to find the closest distance from each data point with the new centroid. The result will be new k-clusters. This proceeder has to be repeated until convergence appears or the centroids are not changing anymore.

2.5.8 Random Forest

The *Random Forest* algorithm is a collection of decision trees (see 2.2.4), this collection is called *Forest*. The *Random Forest* is built out of k *CART* (explained later at section 3.2) algorithm models. These k models are initiated with different trainings data and variables. An object gets classified by 'votes', that means each decision tree (or CART algorithm) is classifying the object and the final class of the object is the most frequent one. For a simplified visualization see figure 2.14.

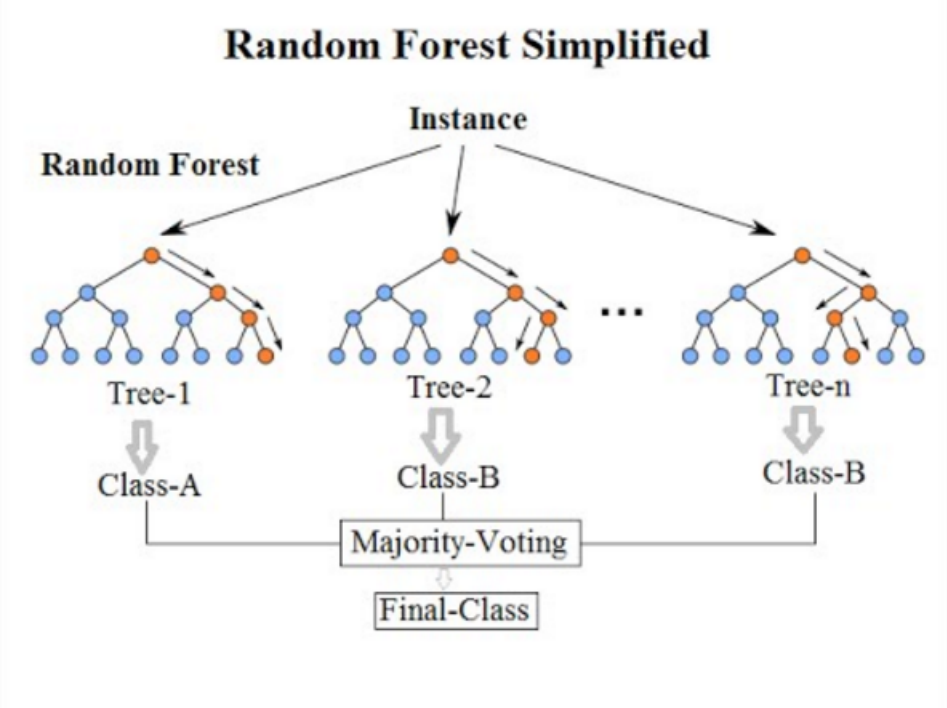


Figure 2.14: Random Forest Simplified by [Koe17]

2.5.9 Dimensionality Reduction Algorithms

In industry and other areas data is produced daily. For instance sensors are storing and producing data continuously for an analysis of how something is or has been done. While capturing data a lot of redundancy can occur. As an illustrative example imagine a motorbike rider in racing competition. The rider gets captured by GPS, various sensors and videos, all of the data recorded can be similar to each other but is not exactly the same and each sensor is important for the analysis. Now to the analysis, imagine an analyst who has to find out what the strategy of the rider is, the variables / dimensions would be immense and it would be nearly impossible to find out the strategy of the rider. This is the problem which should be solved by *Dimensionality Reduction Algorithms*.

A good example for such a reduction is figure 2.15. There are 2 dimensions x_1 and x_2 which are a measurement of objects in centimetres (x_1) and inches (x_2). When using both dimensions in machine learning, they would convey similar information and introduce a lot of noise in a system ergo the better way would be to use only one dimension. In figure 2.15 the data has been converted from 2D into 1D (z_1) which made it easier to explain.

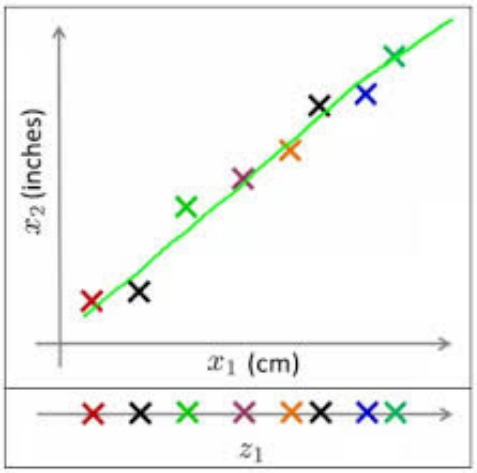


Figure 2.15: Dimension Reduction Example

In similar ways n dimensions can be reduced to k dimensions in a data set ($k < n$). These k dimensions can be directly identified, filtered or can be for instance a combination of average dimensions or new ones that represent existing multiple dimensions well.

Dimensionality Reduction Algorithms are not only for faster classifying because of smaller data but also for

- reducing storage space because of compressing data
- faster computations
- and so on.

Read more about *Dimensionality Reduction Algorithms* at [Sun15].

2.5.10 Classifier Boosting Algorithms

Boosting algorithms are used when we have plenty of data to make a prediction.These boosting algorithms are able to reduce bias and variance and it combines multiple weak predictors to build a super strong predictor in supervised learning (see 2.4.2).

The author will explain these boosting algorithms with *AdaBoost* one of many boosting techniques. The easiest classification problem where we need to assign every observation to a given set of class is the one with binary classes. To understand the concept of AdaBoost imagine two classes zero's (0) and one's (1). Each number is an observation. There are only two features available x-axis and y-axis. For instance (3,4) is in figure 2.16 a one (1). Each observation has to be classified and we have to find optimal classification boundary using AdaBoost.

Firstly, we have to visualize the data to see whether the data has a linear classifier boundary or not. As visible in figure 2.16, no such boundary exists to separate the before mentioned classes.

0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0

Figure 2.16: AdaBoost Data Visualization

Secondly, we have to make the first decision stump. *Decision stump* is a unit depth tree which decides just one most significant cut on features. In figure 2.17 it highlights the first three lines assuming the classes are all zero's and the other lines are containing all one's. However the first decision stump is not correct because there are 9 one's and 18 zero's being wrong classified.

0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0
0	0	0	1	1	1	0	0	0

Figure 2.17: AdaBoost First Decision Stump Visualized

After the wrong classification, step three would be to add more weight to the mis-classified observations (if we know them). In the next step we have to make sure that these mis-classified observations will be correct next time. In figure 2.18 these observations are bold.

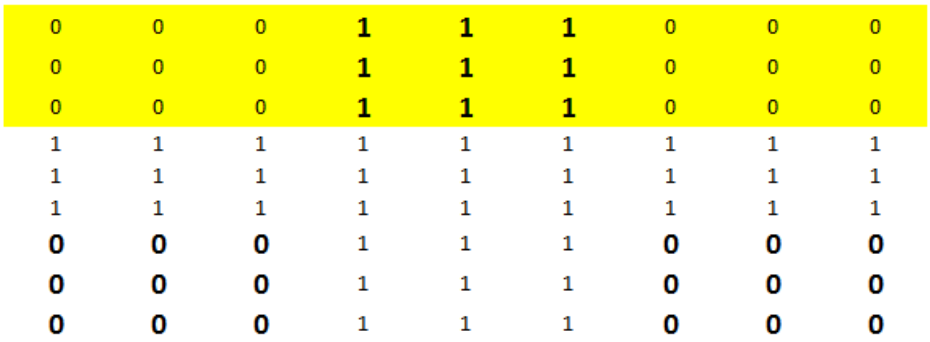


Figure 2.18: AdaBoost Adding Weight To Mis-Classified Observations

Last but not least the before explained steps have to be repeated to get the final classifier, while repeating, it is important to have a focus on the before mis-classified observations. At the end we take a weighted mean of all the boundaries discovered which is visible in figure 2.19. Note: it can be like in figure 2.19, but it can also be different.

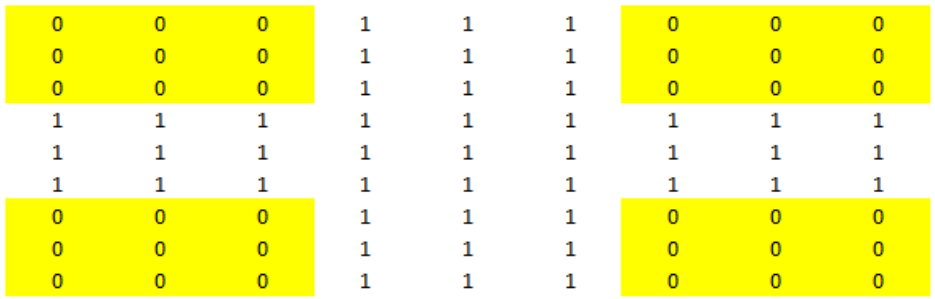


Figure 2.19: AdaBoost Final Classification

Read more about boosting algorithms at [Tav15]

2.6 Weka

2.6.1 Introduction

Weka [EHW16] is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a dataset or called from own Java code. Weka contains tools for data-preprocessing, classification, regression, clustering,

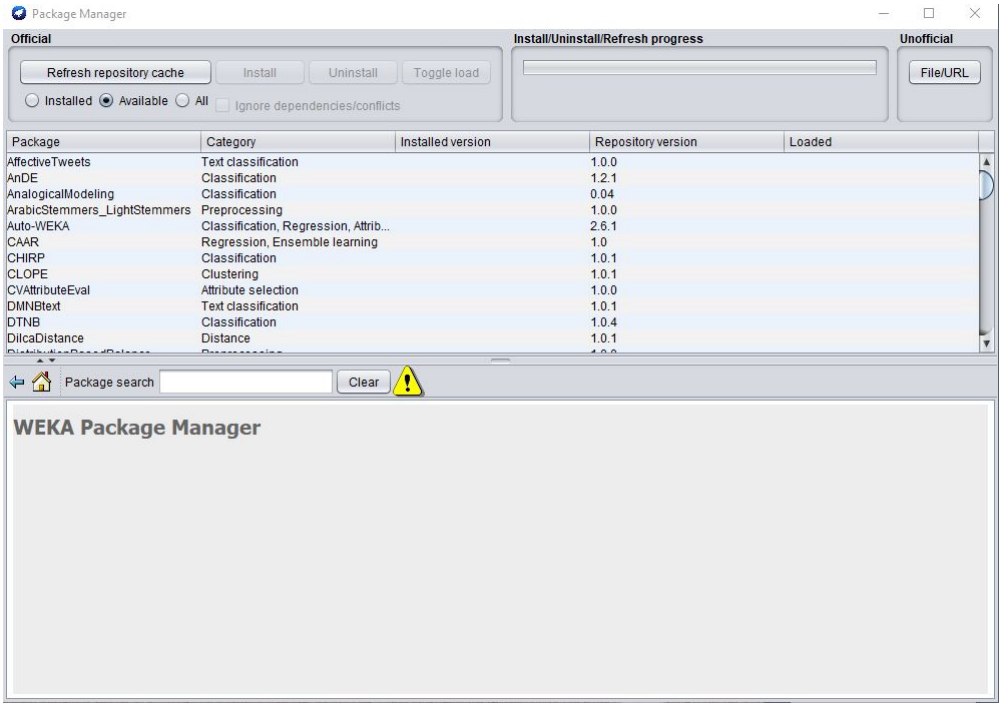


Figure 2.20: Screenshot of the Weka package manager.

association rules and visualization. It is also well-suited for developing new machine learning schemes.

The name *Weka* comes from the eponymous, flightless bird, found only on the islands of New Zealand.

2.6.2 Used Features

Package-Manager

With the package manager of Weka, classifiers that are not part of the main weka jar can be added, updated or deleted. The packages can be chosen from a Weka repository.

Preprocess

In the Preprocess-Section (see 2.21), the hard coded training set can be watched in detail. Every attribute can be shown with the distribution of values. This distribution can also be displayed in a graph, with the outcome of `Recommend` or `Don't Recommend`. Another option is to filter the trainings-set for attribute values, attributes and many more options provided by Weka.

The training set can be imported through the data explorer, via an website url in the JSON format, or directly via JDBC from the database, selecting tables or fields through

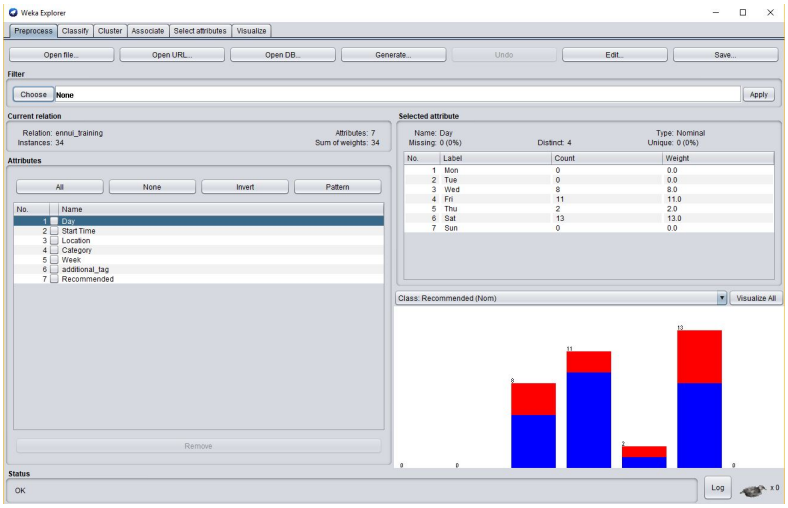


Figure 2.21: Screenshot of the Weka-Preprocess section.

a database query. Generating an entire training set is also possible. The loaded training set can also be edited in Weka and saved after.

Classify

In the *Classify* section of Weka (see 2.22), the user is able to test his training set with all classifiers provided by the Java library plus algorithms that he has downloaded with the package manager.

With double clicking the selected classifier, many options can be changed, for example if the tree should be pruned or not. These options are different for each classifier, representing all parameters defined in the class of the Java library.

Under *Test options* the options for classifier output and test set can be set. In Weka there are four options for test data:

- Use training set: The training set to train the classifier is also used for testing it.
- Supplied test set: With this option, the user is able to choose his own coded test set from a file explorer. For C4.5 and CHAID, this option was used. With providing the same test set, a comparison could be made.
- Cross-validation: The explanation for this option was taken from StackOverflow [Sta18]. The user defines how many folds he wants to perform. For each fold, the training set is split into a training set for producing a classifier and a test set for testing it. If we take an example of a training set with 100 lines of data, and a fold of 10, the following will happen: Weka produces 10 equal sized sets. Each set is divided into two groups: 90 data lines are used for training and 10 for testing. It then produces a classifier from the 90 lines and tests it with the 10 lines for test 1.

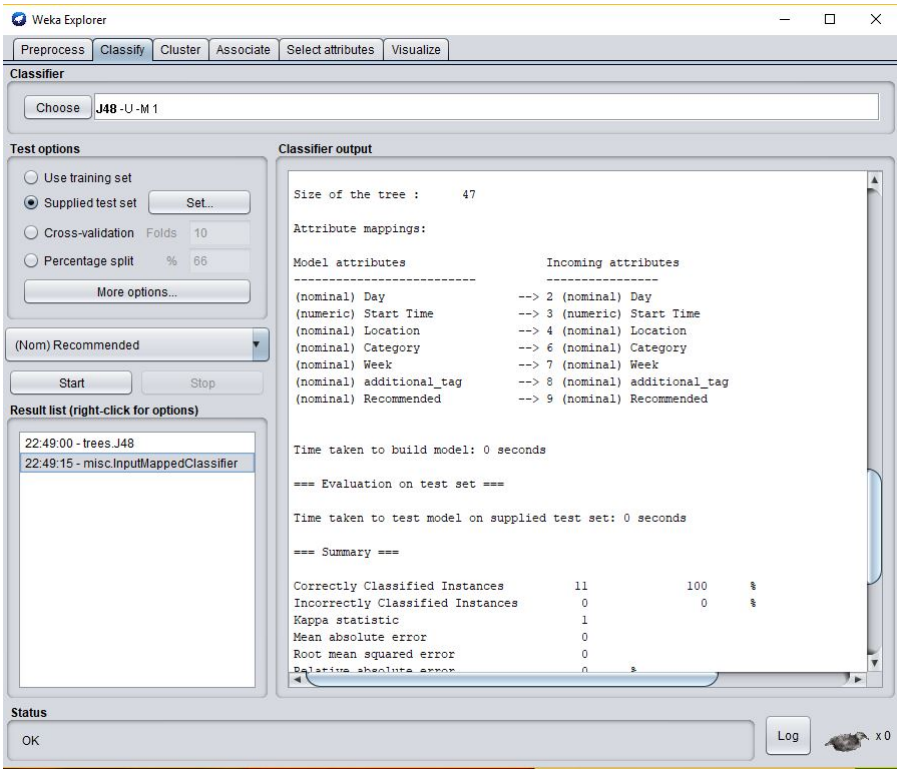


Figure 2.22: Screenshot of the Weka-Classify section.

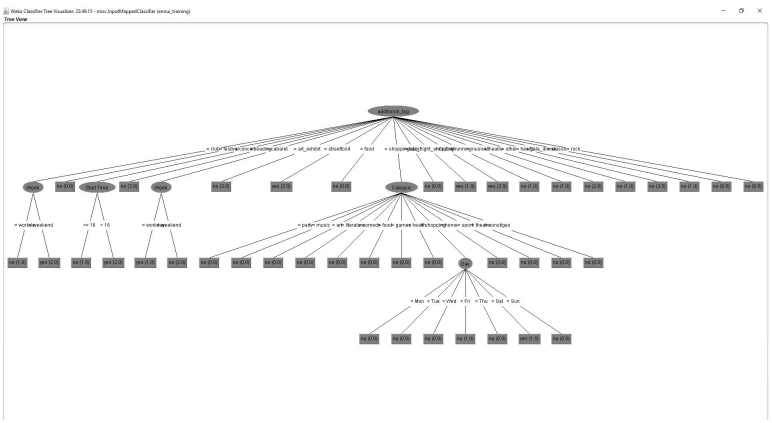


Figure 2.23: Screenshot of a tree visualization by Weka.

It does the same thing for set 2 to 10 and produces 9 more classifiers. After that, it averages the performance of the 10 classifiers produced from 10 equal sized sets and prints it out in the classifier output.

- Percentage split: This provides the option to enter a percentage. The percentage indicates how many percent of the training set will be used for training. The test set gets generated from the remaining data points.

In the classifier output, many information is provided.

- Run information: Information about the used classifier, the name of the training set, number of instances, a list of the attributes and the selected test mode.
- Classifier model: The created tree in textual form, number of leaves, size of the tree and the time needed for building the classifier.
- Evaluation on test set: A summary of the test results, including correctly and incorrectly classified instances and a confusion matrix. The confusion matrix shows where errors occurred, and what the classifier expected as a result.

After building the classifier, the tree can be visualized by Weka as seen in 2.23.

2.6.3 Training Set

ARFF stands for *Attribute-Relation File Format* and is an ASCII text file, that describes a list of instances sharing a set of attributes. ARFF files were developed by the *Machine Learning Project at the Department of Computer Science of The University of Waikato for use with the Weka machine learning software* [oW17].

```
@relation ennui_training
@attribute 'Day' { 'Mon','Tue','Wed','Fri','Thu','Sat','Sun'}
@attribute 'Start Time' numeric
@attribute 'Location' {'Linz'}
@attribute 'Veranstalter' { 'der Hafen','A1 Musikpark','tips-Arena',
@attribute 'Category' { 'party','music','art','literature','comedy',
@attribute 'Week' {'workday','weekend'}
@attribute 'additional_tag' {'club','festival','concert','reading','
@attribute 'Recommended' { 'no', 'yes'}
@data
'Sat',20,'Linz','der Hafen','party','weekend','club','yes'
'Fri',20,'Linz','A1 Musikpark','party','weekend','club','yes'
'Wed',20,'Linz','der Hafen','party','workday','club','no'
```

Figure 2.24: Training set

The file has two distinct sections, starting with the header information, followed by the data information (see Figure 5.1)

Header

The header of the ARFF file contains the name of the relation, a list of attributes (the columns for the data) and their types.

Attribute declarations take the form of an ordered sequence of `@attribute` statements. Each attribute in the data set has its own `@attribute` statement which uniquely defines the name of that attribute and the data type of it. The order the attributes are declared indicates the column position in the data section of the file. For example, if an attribute is the third one declared then Weka expects that all that attributes values will be found in the third comma delimited column. This can also be seen in figure 5.1.

The format for the @attribute statement is:

```
1  @attribute <attribute-name> <datatype>
```

The `<datatype>` can be any of the four types supported by Weka:

- `numeric` : double or integer numbers
- `<nominal-specifications>` : Nominal values are defined by providing an `<nominal-specification>` listing the possible values `{<nominal-name1>, <nominal-name2>, <nominal-name3>, ...}`
- `string` : string values
- `date[<date-format>]`: the `<date-format>` is an optional string specifying how date values should be parsed and printed

Data

The data section starts with the `@data` annotation, followed by the instance data. Each instance is represented on a single line, with carriage returns denoting the end of the instance.

Attribute values for each instance are delimited by commas. They must appear in the order that they were declared in the header section (i.e. the data corresponding to the n^{th} `@attribute` declaration is always the n^{th} field of the attribute).

Missing values are represented by a single question mark.

Chapter 3

Methods for Matching Personal Data with Events

3.1 C4.5-Algorithm

3.1.1 Introduction

C4.5 is an algorithm used to generate a decision tree developed by Ross Quinlan. C4.5 is an extension of Quinlan's earlier algorithm called ID3 [Qui86]. The primary source for this section was taken from [Qui14]. So all information presented in this section is taken from there if not cited otherwise.

3.1.2 Constructing Decision Trees with C4.5

The original idea of creating an initial decision tree from a set of training data goes back to the work of Hoveland and Hunt in the late 1950s, published in their book *Experiments in Induction* in 1966 [HMS66], that describes extensive experiments with several implementation of concept learning systems.

Divide and Conquer

The skeleton of Hunt's method for constructing a decision tree from a set T of training cases leads to three possibilities. Classes are denoted by C_1, C_2, ...

- T contains one or more cases, all belonging to a single Class C_j: The decision tree for T is a leaf identifying class C_j
- T contains no cases: The decision tree is again a leaf, but the class of the leaf must be determined from information other than T. For example, this can be done with background knowledge of the domain, such as the overall most used class in other cases. C4.5 uses the most frequent class at the parent of this node.

Category	DayOfWeek	StartTime	Location	Decision
Party	2	19	Linz	Don't Recommend
Party	3	21	Wien	Don't Recommend
Party	4	21	Linz	Don't Recommend
Party	5	22	Wien	Recommend
Party	6	19	Linz	Recommend
Comedy	1	18	Linz	Recommend
Comedy	3	19	Wien	Recommend
Comedy	6	20	Wien	Recommend
Comedy	6	18	Linz	Recommend
Music	5	16	Wien	Don't Recommend
Music	7	20	Wien	Don't Recommend
Music	1	19	Linz	Recommend
Music	3	20	Linz	Recommend
Music	5	19	Linz	Recommend

Figure 3.1: A small training set

- T contains cases that belong to a mixture of classes: In this situation, the idea is to refine T into subsets of cases that are, or seem to be heading towards, single-class collection of cases. A test set T is chosen, based on a single attribute, that has one or more mutually exclusive outcomes $O_1, O_2, ..., O_n$. T is partitioned into subsets $T_1, T_2, ..., T_n$, where T_i contains all the cases in T that have outcome O_i of the chosen test. The decision tree for T consists of a decision node identifying the test and one branch for each possible outcome. The same tree-building machinery is applied recursively to each subset of training cases.

An Illustration

The successive division of the set of training cases proceeds until all the subsets consist of cases belonging to a single class. This is illustrated in Figure 3.1, a small training set consisting of four attributes and two classes. To simplify the following discussion the cases have been grouped on the first attribute `category`. Also to mention is the fact, that `DayOfWeek` in this case is given in numbers, starting with `Monday = 1`, `Tuesday = 2`, to `Sunday = 7`.

Since these cases do not all belong to the same class, the divide-and-conquer algorithm attempts to split them into subsets. For this example, the test is grouped on the `category` attribute with three outcomes, `category = party`, `category = comedy` and `category = music`. The middle group contains only cases of class `Recommend` but the first and third subsets still have mixed classes. If the first subset is further divided by a test on `DayOfWeek`, with outcomes $DayOfWeek < 5$ and $DayOfWeek \geq 5$, and the third subset by a test on `location`, with outcomes `location = Linz` and `location = Wien`, each of the subsets would now contain cases from a single class. The final divisions of the subsets and the corresponding decision tree are shown in figure 3.2.

```
Partition of cases:
  category = Party:
  |   DayOfWeek < 5
  |   |   Category        DayOfWeek       StartTime         Location      Decision
  |   |   Party               5               22              Wien        Recommend
  |   |   Party               6               19              Linz        Recommend
  |   DayOfWeek ≥ 5
  |   |   Category        DayOfWeek       StartTime         Location      Decision
  |   |   Party               2               19              Linz      Don't Recommend
  |   |   Party               3               21              Wien      Don't Recommend
  |   |   Party               4               21              Linz      Don't Recommend
  category = Comedy
  |   Category        DayOfWeek       StartTime         Location      Decision
  |   Comedy              1               18              Linz        Recommend
  |   Comedy              3               19              Wien        Recommend
  |   Comedy              6               20              Wien        Recommend
  |   Comedy              6               18              Linz        Recommend
  category = Music
  |   Location = Wien
  |   |   Category        DayOfWeek       StartTime         Location      Decision
  |   |   Music               5               16              Wien      Don't Recommend
  |   |   Music               7               20              Wien      Don't Recommend
  |   Location = Linz
  |   |   Category        DayOfWeek       StartTime         Location      Decision
  |   |   Music               1               19              Linz        Recommend
  |   |   Music               3               20              Linz        Recommend
  |   |   Music               5               19              Linz        Recommend

Corresponding decision tree:
    category = Party
    |   DayOfWeek < 5: Recommend
    |   DayOfWeek ≥ 5: Don't Recommend
    category = Comedy: Recommend
    category = Music
    |   location = Wien: Don't Recommend
    |   location = Linz: Recommend
```

Figure 3.2: Final partition of cases and corresponding decision tree

Gain Criterion

The original ID3 on which C4.5 is based uses a criterion called *gain*, defined below. The information theory that is the base of this criterion can be explained as follows: The information conveyed by a message depends on its probability and can be measured in bits as minus the logarithm to base 2 of that probability. So, if there are eight equally probable messages for example, the information conveyed by any one of them is $-\log_2(1/8)$ or 3 bits.

Again we have a possible test with n outcomes that splits the set T of trainings cases into subsets $T_1, T_2, ..., T_n$. If this test is to be evaluated without exploring subsequent divisions of the $T_i s$, as explained in the section above, the only information available for guidance is the distribution of classes in T and its subsets. Because class distribution will be frequently referred in the following chapter it will be notated as following: If S is any set of cases, let freq(C_i, S) stand for the number of cases in S that belong to class C_i. Also, the standard notation will be used, in which $|S|$ denotes the number of cases in set S. Hunt's original concept learning systems experiments considered several rubrics under which a test might be assessed. Most of these were based on class frequency criteria. For example, his first experiment, CLS1 was restricted to problems with two classes, positive and negative, and preferred tests with an outcome whose associated case subset contained

- only positive cases; or, failing that
- only negative cases; or, failing these,
- the largest number of positive cases

Even though his programs used simple criteria of this kind, Hunt suggested that an approach based on information theory might have advantages [HMS66].

Looking at the example of selecting one case at random from a set S of cases and announcing that it belongs to some class C_j, the message has a probability

$$p = \frac{\text{freq}(C_j, S)}{|S|}$$

and so the information it conveys is

$$\log_2(\frac{\text{freq}(C_j, S)}{|S|})\text{bits}$$

To find the expected information from such a message, concerning class membership, the classes in proportion to their frequencies in S get summed, giving

$$\text{info}(S) = -\sum_{j=1}^{k} \frac{\text{freq}(C_j, S)}{|S|} \times \log_2(\frac{\text{freq}(C_j, S)}{|S|})\text{bits}.$$

When applied to the set of trainings cases, $info(T)$ measures the average amount of information needed to identify the class of a case in T. (This quantity is also known as the entropy of the set S.)

Now consider a similar measurement after T has been partitioned in accordance with the n outcomes of a test X. The expected information requirement can be found as the weighted sum over the subsets, as

$$\text{info}_x(S) = -\sum_{i=1}^{n} \frac{|T_i|}{|T|} \times \text{info}(T_i).$$

where $|T_i|$ stands for the number of elements in a part of the whole set T. This formula will be explained later in the illustration. The quantity

$$\text{gain}(X) = \text{info}(T) - \text{info}_x(T)$$

measures the information that is gained by partitioning T in accordance with the test X. The gain criterion, then selects a test to maximize this information gain (which is also known as the mutual information between the text X and the class).

As a concrete illustration, consider again the training set of figure 3.1. There are two classes, nine cases belonging to `Recommend` and five to `Don't Recommend`, so the information that is needed to identify

$$\text{info}(T) = -9/14 \times \log_2(9/14) - 5/14 \times \log_2(5/14) = 0.940\text{bits}.$$

(This represents the average information needed to identify the class of a case in T.) After using the attribute category to divide T into three subsets, the result is given by

$$\text{info}_x(T) = 5/14 \times (-2/5 \times \log_2(2/5) - 3/5 \times \log_2(3/5))$$

$$+4/14 \times (-4/4 \times \log_2(4/4) - 0/4 \times \log_2(0/4))$$

$$+5/14 \times (-3/5 \times \log_2(3/5) - 2/5 \times \log_2(2/5)) = 0.694\text{bits}.$$

In the calculation above the information requirement is calculated after T is divided. In our case we have three subsets, the first being the elements T_i of T that have the category `Party`, the second consists of elements that have the category `Comedy` and the third of events that have the category `Music`. For each line, the number of elements in T_i is divided through the number of elements in the set T. This fraction then gets multiplied with the `info(T)` formula defined above, starting first with the elements that have the class `Recommend` and after that calculating the elements that have the class `Don't Recommend`. This then results in a information requirement of `0.694 bits`. The information gain by this division is therefore $0.940 - 0.694 = 0.246$ bits. Now suppose

that, instead of diving T on the attribute `category`, we had partitioned it on the attribute `location`. This would have given two subsets, one with three `Recommend` and three `Don't Recommend` cases, the other with six `Recommend` and two `Don't Recommend` cases. The similar computation is

$$\text{info}_x(T) = 6/14 \times (-3/6 \times \log_2(3/6) - 3/6 \times \log_2(3/6))$$

$$+8/14 \times (-6/8 \times \log_2(6/8) - 2/8 \times \log_2(2/8)) = 0.892 bits$$

for a gain of 0.048 bits, which is less than the gain resulting from the previous test. The gain criterion would then prefer the test on `category` over the latter test on `location`.

Gain Ratio Criterion

In ID3 the selection of a test was first made on the basis of the gain criterion. This approach gave good results but has a serious deficiency - it has a strong bias in favor of tests with many outcomes. This can be seen by considering a hypothetical medical diagnosis task in which one of the attributes contains a patient identification. Since every such identification is intended to be unique, partitioning any set of training cases on the values of this attribute will lead to a large number subsets, each containing just one case. Since all of these one-case subsets necessarily contain cases of a single class, $\text{info}_X(T) = 0$, so information gain from using this attribute to partition the set of training cases is maximal. From the point of view of prediction, however, such a division is quite useless.

This issue in the gain criterion can be rectified by a kind of normalization in which the apparent gain attributable to tests with many outcomes is adjusted. Consider the information content of a message pertaining to a case that indicates not the class to which the case belongs, but the outcome of the test. By analogy with the definition of $\text{info}(S)$, we have

$$\text{split info}(X) = -\sum_{i=1}^{n} \frac{T_i}{T} \times \log_2(\frac{T_i}{T})$$

This represents the potential information generated by dividing T into n subsets, whereas the information gain measures the information relevant to classification that arises from the same division. Then,

$$\text{gain Ratio}(X) = \text{gain}(X)/\text{splitInfo}(X)$$

expresses the proportion of information generated by the split that is useful, i.e., that appears helpful for classification. If the split is near-trivial, split information will be small and this ratio will be unstable. To avoid this, the gain ratio criterion selects a test to maximize the ratio above, subject to the constraint that the information gain must be large - at least as great as the average gain over all tests examined. It is apparent that the patient identification attribute will not be ranked highly by this criterion. If

there are k classes, as before, the numerator (information gain) is at most $\log_2(k)$. The denominator, on the other hand, is $\log_2(n)$ where n is the number of training cases, since every case has a unique outcome. It seems reasonable to presume that the number of training cases is much larger than the number of classes, so the ratio would have a small value.

To continue the previous illustration, the split on the `category` attribute produces three subsets containing five, four and five cases respectively. The split information is calculated as

$$-5/14 \times \log_2(5/14) - 4/14 \times \log_2(4/14) - 5/14 \times \log_2(5/14)$$

or 1.577 bits. For this test, whose gain is 0.246 (as before), the gain ratio is $0.246/1.577 = 0.156$. The gain ratio criterion typically gives a consistently better choice of the right split than the gain criterion. It even appears advantageous when all tests are binary but differ in the proportions of cases associated with the two outcomes. However, Mingers [Min89] compares several test selection criteria and while he finds that gain ratio leads to smaller trees, he expresses reservations about its tendency to favor unbalanced splits in which one subset T_i is much smaller than the others.

Possible Tests Considered

A criterion for evaluating tests provides a mechanism for ranking a set of proposed tests so that the most favorable-seeming test can be chosen. This presumes, of course, that there is some way of generating possible tests. Most classifier-building systems define a form for possible tests and then examine all tests of this form. Conventionally, a test involves just one attribute, because this makes the tree easier to understand and sidesteps the combinatorial explosion that results if multiple attributes can appear in a single test. C4.5 contains mechanisms for proposing three types of tests:

- *Standard* test on a discrete attribute, with one outcome and branch for each possible value of that attribute
- A more complex test, based on a discrete attribute, in which the possible values are allocated to a variable number of groups with one outcome for each group rather than each value.
- If attribute A has continuous numeric values, a binary test with outcomes $A \leq Z$ and $A > Z$, based on comparing the value of A against a threshold value Z.

All these tests are evaluated in the same way, looking at the gain ratio (or, optionally, the gain) arising from the division of the trainings cases that they produce. It has also proved useful to introduce a further constraint: For any division, at least two of the subsets T_i must contain a reasonable number of cases. This restriction, which avoids near-trivial splits of the training cases, comes into play only when the set T is small.

3.1.3 Implementation

In Weka, explained in *Chapter 2 : Machine Learning* 2.6 a Java implementation of the C4.5 algorithm can be found. Weka also provides a large amount of tools, which can be used to analyse the algorithm in his functionality and results, watching the generated tree classifier from own training data graphically and testing the algorithm with own test data. After testing the algorithm in the environment of Weka and changing parameters to guarantee the best fit for Ennui without producing an overfit, the algorithm was implemented into the test environment as follows:

```
1   public List<Long> applyC45(List<AlgorithmDTO> events, String filepath)
        throws Exception {
2       BufferedReader reader = new BufferedReader(
3               new FileReader(filepath));
4       Instances data = new Instances(reader);
5       reader.close();
6       data.setClassIndex(data.numAttributes() - 1);
7       String[] options = {"-U"};
8       J48 tree = new J48();       // new instance of tree
9       tree.setOptions(options);  // set the options
10      tree.setMinNumObj(1);
11      tree.buildClassifier(data); // build classifier
12      Instances dataRaw= prepareData(events);
13      for (int i = 0; i < dataRaw.numInstances(); i++) {
14          double clsLabel = tree.classifyInstance(dataRaw.instance(i));
15          double[] confidenceLevel =
                tree.distributionForInstance(dataRaw.instance(i));
16          if(clsLabel ==0){
17              events.get(i).setRecommended("no");
18              events.get(i).setConfidenceLvl(confidenceLevel[0]);
19          }
20          if(clsLabel ==1){
21              events.get(i).setRecommended("yes");
22              events.get(i).setConfidenceLvl(confidenceLevel[1]);
23          }
24
25      }
26      return events.stream().filter(a ->
            a.getRecommended().equals("yes")).map(AlgorithmDTO::getId)
27      .collect(Collectors.toList());
```

In the lines 2-5, the training data gets read from the hard-coded training set. In the following line 6 the class index gets set on the last attribute. The class index is the attribute used for the decision in each terminal leaf, in the case of Ennui `recommended` with the outcomes `no` and `yes`. After that, from line 7-11 the tree-based classifier gets built with the options provided. These options got tested before in the environment of Weka and have worked the best for Ennui. In line 12, the `prepareData()` method gets called, preparing the events of Ennui for the classification. After that, a `for` loop is

executed that classifies each event and sets the confidence level of it. Lastly, the ids of the events that are recommended by the algorithm get returned.

3.2 Cart-Algorithm

3.2.1 Introduction

CART stands for *Classification and Regression Tree* and differently to the C4.5 algorithm the CART algorithm results in a binary tree. It was developed in the early 1980's by Breiman, Friedman, Olshen and Stone [BFOS83]. The CART algorithm helps to classify objects though a tree based model. It is one of the classical algorithms for this functionality.

This section starts with an explanation about the *GINI-Index* how it works and examples for it, afterwards purity is explained and through this knowledge the author is able to explain how the *CART-Algorithm* works. Last but not least the author shows how the algorithm was implemented in JAVA.

3.2.2 Gini Index

The Gini-Index or also called the *Gini coefficient* has been invented by Corrado Gini (also mentioned in [BFOS83]). It is a measure of statistical dispersion in a frequency distribution.

The CART algorithm uses it to declare the condition on what data has to be split and which decision rule or condition is fitting best for the data object to classify. The better this rule is the smaller is the tree because the leafs are purer.

Lets consider an object with different attributes for example a data object which looks like this in JAVA.

```
1 public class DataObject {
2     private long id;
3     private String category;
4     private String name;
5     private Date start_time;
6     private Date end_time;
7 }
```

We want to know on which attribute we should split and which would fit best for splitting. To do so there is the Gini-Index which helps us calculating a percentage value of each field. The calculated value can be between zero (0) and one (1). If the result is zero the attribute is absolutely pure and we do not have to split on this attribute. The higher the result is the impurer the node is and we have to split on it. Given an attribute t which can get N different values v $v_1, v_2, ..., v_n$. The relative frequency of a

Table 3.1: Sample Data Frequency

category	**absolute frequency (c)**	**relative frequency (n)**	**Percentage Visualized**
Club	30	0.56	56%
Concert	8	0.15	15%
Reading	1	0.02	2%
Festival	13	0.24	24%
Classic	2	0.04	4%
Overall	54	1	100%

value v_i is denoted by $h_i(1 <= i <= N)$. Now the Gini-Index for t is defined as follows.

$$GINI(t) = 1 - \sum_{i=1}^{N}(h_i)^2$$

Note: The impurity level or the Gini-Index is not able to reach one (1). To reach one there must be an infinite count of possible values in one attribute where all values have the same percentage h.

To clarify this concept we take test data which describes the attribute *category*. As an illustrative example the author has written imaginary data in the following table. How the test data has taken place is explained in chapter five (5). In the table we have listed all possible nominal values the attribute *category* can have. The *absolute frequency* is the amount how often the attribute occurs in the test data with the value listed. In the third column the author just calculated the percentage of the absolute frequency. The *Gini-Index* for this attribute is now calculated with this equation:

$$GINI = 1 - \sum_{i}^{N}(h_i)^2$$

h_i is the relative frequency in the table above. It has to be a decimal value between zero and one and not the percentage visualized which is also in the table above. In the example of the author the result is:

$$GINI = 1 - (0.56^2 + 0.15^2 + 0.02^2 + 0.24^2 + 0.04^2) = 0.609739$$

The result is the level of impurity reached in this case impurity is by 60 % . Note: this calculation happens for each attribute in an object.

Lets assume we have calculated the *Gini-Index* for each attribute in the object and we decided splitting by the attribute *category*, because the impurity level on this attribute is the highest.

The attribute *category* can be more than one nominal value, so we have to decide on which value, of the values the attribute *category* can be, we have to split or is fitting best for splitting.

Table 3.2: Sample Data Left Node t_L

category	**absolute frequency (c)**	**relative frequency (n)**	**Percentage Visualized**
Concert	8	0.33	33.3%
Reading	1	0.042	0.42%
Festival	13	0.542	54.2%
Classic	2	0.083	0.83%
Overall	24	1	100%

Table 3.3: Sample Data Right Node t_R

category	**absolute frequency (c)**	**relative frequency (n)**	**Percentage Visualized**
Club	30	1	100%
Overall	30	1	100%

The equation to do this is:

$$GINI(s,t) = GINI(t) - P_L \cdot GINI(t_L) - P_R \cdot GINI(t_R)$$

where

s = the split which has to be calculated

t = the node or attribute in the example it is *category*

$GINI(t)$ = The Gini-Index of node t

P_L = The proportion of instances for the left subset, or how many instances are in the left subset divided by the total number of instances in the node t

$GINI(t_L)$ = The Gini-Index of the left node after split s

P_R = The proportion of instances for the right subset, or how many instances are in the right subset divided by the total number of instances in the node t

$GINI(t_R)$ = The Gini-Index of the right node after split s

We want to split by the nominal value *club* in the attribute *category* ergo the imaginary decision rule would be: Is the category club or not? According to this rule the two subsets or the nodes after splitting by this rule are in the following tables. The first one is for t_L and the second one for t_R. In the first table the author removed the category *club* because *club* can not be in the left branch and in the second table the author removed everything which has not the category *club* because in the right branch there should be the *club* category.

To get P_L and P_R we have to sum the instances of each subset and divide it by the total instances of the node. According to this P_L is:

$$P_L = \frac{24}{54}$$

and P_R is:

$$P_R = \frac{30}{54}$$

Table 3.4: Song Samples

Song-Id	Specific Genre
0	Hardstyle
1	Hardcore
2	Lounge
3	Old School Rap
4	Underground Rap
5	Frenchcore

After applying the data to the equation

$$GINI(s,t) = GINI(t) - P_L \cdot GINI(t_L) - P_R \cdot GINI(t_R)$$

The result is

$$GINI(s,t) = 0.61 - \frac{24}{54} \cdot 0.587 - \frac{30}{54} \cdot 0 = 0.34894$$

After calculating this computation for each value in the attribute we can decide on which one we want to split and which one is best for splitting. Note: The bigger the impurity level is the better is it for splitting.

3.2.3 Purity

Purity describes the level of how effective the split was and how good the classification worked. For example there is an object called *Song* and those songs should be classified in a music genre. In the following table there are six example songs listed. There are many music genres and those genres also have subgenres. In this example the author wants to classify the above listed songs to the main genre. Note, the algorithm does not know the genre in the second column, this column is just for visualization. After analysing the songs the algorithm classified them to a main genre. And there it is unnecessary if the genre is *Old School Rap* or *Underground Rap* both are a form of Rap (Only if our goal is to classify the main genre not the sub genre). And that is purity. If the tree would split again after knowing the main genre for example 'is underground rap or not' would only cost performance and does not take affect in the end result if there should only the main genre be known. For visualization of the tree see figure 3.3

3.2.4 How CART Works

Roughly said CART takes all the data and selects the greatest separation from the object through the before mentioned GINI-Index. Let's say this object

$$O = t1$$

where $t1$ stands for an attribute inside the object, is at the beginning or rather at the first decision rule or condition of the tree. Assuming X as the split condition, for example

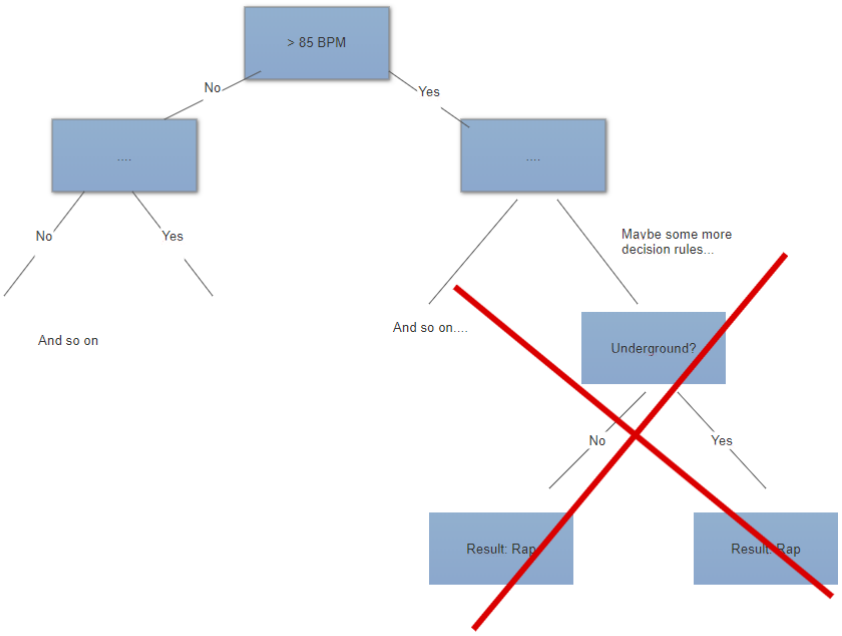


Figure 3.3: Purity Music Genre Example

'Is the person taller than 190cm?', than if

$$t_1 < X$$

the object is sent to the left side otherwise when

$$t_1 > X$$

it is sent to the right side of the tree. Because CART is based on a Binary Decision Tree (see 2.2.4) there are only two possibilities where the object can go to. Those sending and splitting continues till the end. 'Till the end' could also be infinite so the algorithm has to set a point where to stop splitting. Most of the time this stopping point is reached when a split happens where the purity (see 3.2.3) does not change any more or is too small that it would make sense. CART itself is building its tree all the way out and afterwards the pruning happens where the algorithm decides which nodes have and will be deleted if the before mentioned purity change is to small.

3.2.5 Implementation

To be able to implement the CART algorithm the Java-Library from Weka-Tools was used. In the Ennui backend the implementation of the CART algorithm out of the Weka-Library is written in the following steps. The method head of the algorithm implementation looks like the following:

```
1 public List<Long> applyCART(List<AlgorithmDTO> events, Instances data) throws
      Exception {
2     }
```

The return type of this function is a *List of Longs* and those numbers are going to be the recommended events based on data delivered via the first parameter `List<AlgorithmDTO> events` (For more details of how this data object is built see 1.3.5). The second parameter *Instances data* is used to build the classifier (see 2.2.2). To finally build the classifier the following code is needed.

```
1 data.setClassIndex(data.numAttributes() - 1);
2         String[] options = {"-U"};
3         SimpleCart tree = new SimpleCart();     // new instance of tree
4         tree.setOptions(options);  // set the options
5         tree.setMinNumObj(1);
6         tree.setUsePrune(false);
7         tree.buildClassifier(data);
```

To prepare data for the classifier to classify the data the author has written the following lines.

```
1 Instances dataRaw= prepareData(events);
2         List<AlgorithmDTO> classifiedEvents =
              classifyEventData(tree,dataRaw,events.toArray(new
              AlgorithmDTO[events.size()]));
```

The events which are delivered via the first parameter of the method head have to be prepared to run through the classifier. For this the method `prepareData()` is called, for more information about this method see 4.5.6. After preparing the event data for classification, it is finally possible to classify the data by calling the method `classifyEventData()`.

```
1  private List<AlgorithmDTO> classifyEventData(AbstractClassifier tree,
       Instances dataRaw, AlgorithmDTO[] dtos) throws Exception {
2          for (int i = 0; i < dataRaw.numInstances(); i++) {
3              double clsLabel = tree.classifyInstance(dataRaw.instance(i));
4              double[] confidenceLevel =
                   tree.distributionForInstance(dataRaw.instance(i));
5              if(clsLabel == 0){
6                  dtos[i].setRecommended("no");
7                  dtos[i].setConfidenceLvl(confidenceLevel[0]);
8              }
9              if(clsLabel == 1){
10                 dtos[i].setRecommended("yes");
11                 dtos[i].setConfidenceLvl(confidenceLevel[1]);
12             }
13         }
14         return Arrays.asList(dtos);
15     }
```

This method simply loops through the data and classifies the events. After classifying those events by calling `tree.classifyInstance()` we get a `boolean` whether the event is recommended or not. For getting the confidence level (see 2.2.4) also a simple method `tree.distributionForInstance()` is called. To finally return the ids of events which are recommended the last line in the now completed method is added.

```
1  public List<Long> applyCART(List<AlgorithmDTO> events, Instances data) throws
       Exception {
2          data.setClassIndex(data.numAttributes() - 1);
3          String[] options = {"-U"};
4          SimpleCart tree = new SimpleCart();     // new instance of tree
5          tree.setOptions(options);  // set the options
6          tree.setMinNumObj(1);
7          tree.setUsePrune(false);
8          tree.buildClassifier(data); // build classifier
9          Instances dataRaw= prepareData(events);
10         List<AlgorithmDTO> classifiedEvents =
               classifyEventData(tree,dataRaw,events.toArray(new
               AlgorithmDTO[events.size()]));
11         return classifiedEvents.stream().filter(a ->
               a.getRecommended().equals("yes")).map(AlgorithmDTO::getId)
12         .collect(Collectors.toList());
13     }
```

The last line is just an iteration over the classified events, filtering them by the recom-

mendation state and mapping them to a `List<Long>`.

3.3 CHAID-Algorithm

3.3.1 Introduction

CHAID (=**C**hi-**S**quare **A**utomatic **I**nteraction **D**etection) is a big data sorting algorithm, that uses decision trees for determining event recommendations for users [Wik17]. It was invented in 1980 by Gordon V. Krass [Kas80]. Its main trait is that it uses *Chi squared distribution tests of independence* and adjusted significance testing (see section 3.3.3) when building the decision tree for determining the attribute for splitting a node. To improve the quality of the decision tree's separation a node is allowed to have more than two child nodes, which means the tree generated via CHAID is, like the C4.5 not binary which differentiates it from the CART algorithm. This section serves to give an overview of the CHAID algorithm and all its features. It starts with general information about the algorithm and also gives an insight in how the algorithm is used for segmenting a set of objects based on a training set. (see 2.2.4). Then it will cover how thees principles were used for the specific case of this diploma thesis, namely matching personal profiles with spare time activities. The Java implementation into the Back-End will also be covered in this section. For the source of this section see [noa], [Wik17] and [efr17]

3.3.2 Chi-Squared Distribution Tests of Independence

The CHAID algorithm uses a variety of different statistical operations during its running time, with the *Chi-Squared Distribution Tests of Independence* (see [Wik18a]) being the most used one. Since it is referenced in the following chapter a couple of times and its mechanics are mandatory for understanding the complete process of the CHAID algorithm, it will be explained in the following section.

Chi-squared distribution tests of independence are used to determine whether there is a significant relationship between two *categorical* or *ordinal* variables. Any variables whose values can be any value of an interval (for example height or age) have to be converted into ordinal variables first.

categorical A categorical variable is a variable with a defined set of possible values that do not follow a specific order. Popular examples of a categorical variable would be color, gender or genres of music.

ordinal A ordinal variable is a variable with a defined set of possible values that do follow a specific order. Popular examples of a ordinal variables are level of education or variables that represent a scale with values like *low, medium* and *high*.

A chi-squared test of independence calculates a chi-squared value which indicates how significant the relation between the two compared variables is. It is calculated as

Table 3.5: Chi-Squared Test Example

	Male	**Female**	**Total**
Blue	190	25	**215**
Pink	10	225	**235**
Total	**200**	**250**	**450**

follows: Assume a set of observed data. Let O be a two dimensional matrix of observed values where one dimension represents an observed amount for each possible value of the first variable and the other dimension the observed amount for each possible value of the second variable. Let E be a matrix of the same size in each dimension as O and a value E_{ij} in E be the respective expected amount of a value O_{ij} in O. The expected amount of a value is calculated with the arithmetical average which is the total amount of values observed at index i multiplied by the total amount of values observed at index j and divided by the size of the observed data set. Let X and Y be the total amount of possible values for the two compared variables.

$$\chi^2 = \sum_{i=1}^{X} \sum_{j=1}^{Y} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

To visualize this calculation assume the following example. The following chi-square test calculates whether there is a significant relation between the gender and favorite color of students, or in other words test if the two variables are dependent on each other. A survey in a school returned the following result:

Applying the formula for chi-squared tests on this observed data set, the formula for the chi-square value would look like this:

$$\chi^2 = \frac{(190 - 95.6)^2}{95.6} + \frac{(25 - 119.4)^2}{119.4} + \frac{(10 - 104.4)^2}{104.4} + \frac{(225 - 130.6)^2}{130.6}$$

So basically all this formula does is squaring the difference between observed and expected for each combination of values of the dependent attribute and the values used to perform the chi-square test, divides it by the respective expected amount and adds those values together.

Calculating a p-value

A possible extension to chi-squared distribution tests of independence is calculating a so called *p-value*. This p-value indicates the probability of getting the observed chi-squared value of the test statistics of something even larger. This p-value is calculated by forming an integral over the distribution-function of the chi-square test starting at the calculated chi-value as seen in figure 3.4. The only problem with this is that the distribution function of the chi-square test depends on the amount of *Degrees of Freedom* F of the test as seen in figure 3.5. Degrees of freedom are an indicator on how many variable factors are affecting the range of the outcome. The amount of calculated freedom

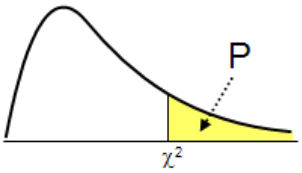


Figure 3.4: p-value under the distribution function (source: [Med18])

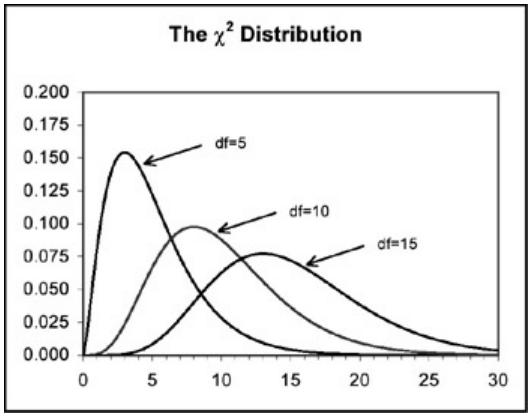


Figure 3.5: Distribution Functions with different degrees of freedom (source: [Kar03])

is calculated like this: Let X and Y again be the total amount of possible values of the two compared variables.

$$F = (X - 1) \times (Y - 1)$$

which in our example with the students would be

$$F = (2 - 1) \times (2 - 1) = 1$$

After the amount of degrees of freedom is calculated the integral can be formed over the right distribution-function and therefore, the p-value can be calculated.

3.3.3 General

This part covers the general steps of the CHAID algorithm. As already mentioned, CHAID uses decision trees for *classifying* objects. To generate this decision tree the algorithm needs a training set which contains already classified objects.

classify Classifying objects means identifying to which of a set of groups a new observed object belongs based on a so called training set which consists of objects whose

group is already known. Popular examples include determining whether a bank client is solvent or not or whether an e-mail is spam or not.

Based on the training set, the algorithm knows the structure of a to-be classified object, meaning it knows what attributes the object has and what possible values they can have. It then builds a tree (see 2.2.4) where each node represents a split by a certain attribute. For the procedure of the CHAID algorithm the following information is given:

- A training set T consisting of classified objects O. This training set is provided by the backend.
- An object type O having the following fields
  - A one or more attributes X with each attribute having a certain amount of possible values p. The attributes and all their possible values are provided by the training set. Additionally the attributes are referenced by X_1, X_2, ..., X_n where n is the actual number of attributes the object has.
  - A classifier attribute (also called dependent attribute) Y which indicates to which element of groups the object belongs. For the training set this field already has a value.
- A root node R of the decision tree, which holds the training set T and also the list of attributes X.

The procedure of the algorithm goes as follows:

Step 1: Merging the Attribute Values

The first step of the algorithm is to merge very similar values p for each attribute X of the object O. This is done because a decision tree with effectively less possible values for each attribute becomes significantly smaller and therefore, results in an increased performance. The process of merging multiple branches of a decision tree into one is called *pruning* a tree.

The way CHAID implements this is by comparing all values p of a certain attribute X_i pairwise. The two values compared are denoted as p_1 and p_2. The algorithm then performs a chi-squared test of independence(see section 3.3.2) using X_i as first variable (with only considering p_1 and p_2 as possible values) and Y as the second for calculating a p-value associated with the certain combination of values of X_i . As already stated in section 3.3.2 the p-value serves as an indicator on how similar p_1 and p_2 are in relation to Y. The larger the p-value is, the less significant is the chi-squared test and therefore the less different are the values. So all the algorithm now has to do is to search for the pair p_1 and p_2 with the highest p-value associated and merge them. This process of merging is repeated until the most similar pair of values is considered not similar enough for a merge, meaning that the p-value is smaller than a user specified limit, called the

alpha level. This repetition of merging steps entails that a value can be merged with an already merged value. Once the highest p-value is lower than the alpha-level the merging process is stopped. This merging step is applied to all attributes X.

Step 2: Determining the Best Split

After each attribute of X is pruned, the node R is ready for splitting. Even though we can choose an arbitrary attribute to split the node, we can also calculate which attribute has the most distinguishable values. This can be archived by calculating a so called *adjusted p-value* for each attribute. The p-value for a whole attribute X_i is calculated by performing a chi-squared test of independence using X_i as the first variable and Y as the second. The adjusted version of this p-value is calculated using *bonferroni adjustments*.

Bonferroni Adjustments Bonferroni Adjustments are used for calculating an adjusted version of a p-value. This is done by multiplying the p-value of an attribute with a so called Bonferroni multiplier B. This multiplier is needed because in mathematical statistics, performing several simultaneous tests on a single data set increases the chance of obtaining false-positive results. Assuming that an attribute X originally has I categories and is reduced to r categories after the merging step and v is an iterator variable for the sum, the formula for calculating B looks like the following:

$$B = \sum_{v=0}^{r-1} (-1)^v \frac{(r-v)^I}{v!(r-v)!}$$

where $v!$ is the faculty of v. Assuming an attribute X_i which originally had 4 possible values which got merged to 2 values by the first step of the CHAID algorithm. Calculating B with this conditions for X_i would look like the following:

$$B = \sum_{v=0}^{1} (-1)^v \frac{(2-v)^4}{v!(2-v)!} = -9$$

The attribute having the smallest *adjusted p-value* is the one used for the split. If the smallest p-value is less or equal a user specified limit, called *alpha level*, or there are no attributes left, the node will not be split and becomes terminal.

Step 3: Splitting a Node

After the best split for the node R is determined, it is ready for splitting. For each value of the chosen attribute X_i a new child node C_i is created, meaning each child node C_i of R represents one or more possible values of X_i. As it is with R each child node C_i also needs a list of attributes and a training set to continue the procedure of the algorithm. For the training set, each element of T whose value of the attribute X_i equals the one that the child node C_i represents is adopted to the training set of C_i.

The list of attributes of each child node, equals the list of attributes of their parent R with the attribute X_i which was determined in step 2 excluded from it. After the split is performed the algorithm is repeated at step 1 for each child node C_i with each child node becoming the root node R of its own sub-tree.

3.3.4 Implementation

This section covers the Java implementation of the CHAID algorithm, for the specific task of our diploma thesis, namely finding the best event recommendation for a user based on his preferences. Meaning the algorithm classifies the events by their recommendation state which is either recommended or not, or more specifically true or false. As a training set a set of already classified events is used, meaning it is already known to the algorithm whether an event of the training set is recommended for the user or not.

Data Structure

For the implementation multiple Java objects were created and used throughout the programming process. Since the most important ones will be referenced multiple times during this text and understanding their structure is crucial for a full understanding of the implementation.

AlgorithmDTO Roughly speaking, an *AlgorithmDTO* represents an Event. More specifically it describes a to-be classified object for the CHAID algorithm. Additionally since the objects in the training set have the same structure as the to-be classified objects AlgorithmDTOs are used for the code-wise implementation of the training set as well, however it has to be considered that for the training set only the fields relevant for segmentation are filled. (An example of a relevant field would be the event-category or the day of the week, an example of a non relevan field would be the name or the id of an event). For a more detailed description of the AlgorithmDTO see 1.3.5.

ChaidAttribute *ChaidAttributes* represent the variables/attributes of an Event which will be used for splitting the decision tree. A ChaidAttribute not only contains a list of all possible values for the attribute. It also registers for each possible values how many of the events, which have that specific value, are recommended and how many are not. This list of possible values is generated by collecting all values of a specific attribute from the training set. Popular examples for ChaidAttributes are the category, the day, the location and the additional tag attributes from the AlgorithmDTO (see 1.3.5).

ChaidNode A *ChaidNode* is a specific node of a decision tree created by the CHAID algorithm. It holds:

- the training set, which is implemented via a list of AlgorithmDTOs
- a list of ChaidAttributes, which represents the possible attributes to split the node.

- a confidence level, which indicates the probability of a new AlgorithmDTO being recommended, or not.

Setting Up

To build a user specific decision tree, the algorithm has to interpret the training set first, and then transfer the data into a list of Java AlgorithmDTOs (see 3.3.4). The Algorithm then creates a root ChaidNode (see 3.3.4). A sample training set would look like the following:

```
@relation ennui_training
@attribute 'Day' { 'Mon','Tue','Wed','Fri','Thu','Sat','Sun'}
@attribute 'Start Time' numeric
@attribute 'Location' {'Linz'}
@attribute 'Veranstalter' { 'der Hafen','A1
    Musikpark','tips-Arena','Lentos','Posthof','Stadtplatz','Plus-City','Gugl','Volkstheater'}
@attribute 'Category' {
    'party','music','art','literature','comedy','food','games','health','shopping','home','sport'
@attribute 'Week' {'workday','weekend'}
@attribute 'additional_tag'
    {'club','festival','concert','reading','cabaret','art_exhibit','streetfood','food','shopping'
@attribute 'Recommended' { 'no', 'yes'}
@data
'Sat',20,'Linz','der Hafen','party','weekend','club','yes'
'Fri',20,'Linz','A1 Musikpark','party','weekend','club','yes'
'Wed',20,'Linz','der Hafen','party','workday','club','no'
'Sat',20,'Linz','tips-Arena','music','weekend','concert','yes'
'Fri',20,'Linz','tips-Arena','music','weekend','concert','yes'
'Sat',14,'Linz','tips-Arena','music','weekend','concert','no'
'Sat',20,'Linz','tips-Arena','music','weekend','classic','no'
'Sat',20,'Linz','tips-Arena','music','weekend','rock','no'
'Sat',18,'Linz','Lentos','art','weekend','art_exhibit','no'
'Fri',19,'Linz','Lentos','art','weekend','art_exhibit','no'
'Wed',18,'Linz','Lentos','art','workday','art_exhibit','no'
'Sat',18,'Linz','Lentos','literature','weekend','reading','no'
'Fri',19,'Linz','Lentos','literature','weekend','reading','no'
'Wed',18,'Linz','Lentos','literature','workday','reading','no'
'Sat',20,'Linz','Posthof','comedy','weekend','cabaret','no'
'Fri',20,'Linz','Posthof','comedy','weekend','cabaret','no'
'Wed',19,'Linz','Posthof','comedy','workday','cabaret','yes'
'Sat',15,'Linz','Stadtplatz','food','weekend','streetfood','yes'
'Fri',16,'Linz','Stadtplatz','food','weekend','gala_dinner','no'
'Wed',18,'Linz','Stadtplatz','food','workday','streetfood','yes'
'Sat',15,'Linz','Stadtplatz','health','weekend','health','no'
'Fri',16,'Linz','Stadtplatz','health','weekend','health','no'
```

Both the list of events(*AlgorithmDTOs*) and the list of attributes are provided by the training set, which is read by a `BufferedReader`. Since those lines of the training set

which contain information about the possible values for the attributes start with an *"@"* as seen in the snippet above it is very simple to extract this data from the file. The code snippet below shows how collecting the possible values for the attributes day, location, category, week and additionalTag is implemented code-wise:

```
if(line.contains("@")) {
   if(line.contains("Day")) {
      days = collectDataFromLine(line);
    }
    if(line.contains("Location")) {
        locations = collectDataFromLine(line);
    }
    if(line.contains("Category")) {
      categories = collectDataFromLine(line);
    }
    if(line.contains("Week")) {
      week = collectDataFromLine(line);
    }
      if(line.contains("Additional Tag")) {
         additionalTags = collectDataFromLine(line);
      }
}
```

The function *collectDataFromLine* returns a list of strings and uses the Java Stream API for a cleaner code. It looks like the following:

```
private List<String> collectDataFromLine(String line) {
   return Arrays.stream(line.split("\\{")[1].split("'"))
      .filter(s -> !(s.contains(",") || s.contains(" ") || s.contains("}") ||
          s.equals("")))
      .collect(Collectors.toList());
}
```

For collecting the Events from the trainings-set the algorithm looks for lines that do not start with an `@` as seen in the source code implementation below:

```
if(!line.contains("@")) {
   AlgorithmDTO newEvent = new AlgorithmDTO();
   line = line.replace("'", "");
   newEvent.setWeekday(line.split(",")[0]);
   newEvent.setStart_hour(Integer.valueOf(line.split(",")[1]));
   newEvent.setCity(line.split(",")[2]);
   newEvent.setCategory(line.split(",")[4]);
   newEvent.setWeek(line.split(",")[5]);
   newEvent.setAdditional_tag(line.split(",")[6]);
   newEvent.setRecommended(line.split(",")[7]);
   trainingsData.add(newEvent);
}
```

Merging Attribute Values

After the set up is finished and the root node is created, the algorithm is ready to start its procedure. The following code snippet shows the java implementation of the `mergeLeastSignificantPredictors` function, which is implemented in the ChaidAttribute-Class.

```
1  public void mergeLeastSignificantPreditors(List<AttributeValue> predictors) {
2     AttributeValue maxP1 = null;
3     AttributeValue maxP2 = null;
4
5     double maxPval = 0;
6     for(int i1 = 0; i1 < predictors.size(); i1++){
7        for(int i2 = i1; i2 < predictors.size(); i2++) {
8
9           AttributeValue p1 = predictors.get(i1);
10          AttributeValue p2 = predictors.get(i2);
11
12          if(!p1.equals(p2)) {
13             double pValue = p1.compareChiSquare(p2);
14             if((maxP1 == null && maxP2 == null) || pValue > maxPval) {
15                maxP1 = p1;
16                maxP2 = p2;
17                maxPval = pValue;
18             }
19          }
20       }
21    }
22    if(maxPval >= ALPHA_LEVEL && predictors.size() > 2) {
23       maxP1.merge(maxP2);
24       predictors.remove(maxP2);
25       mergeLeastSignificantPreditors(predictors);
26    }
27 }
```

This method takes a list of values and compares all of them with each other and performs chi-square tests to find the pair with the highest associated p-value, which can be seen from lines 6 to 21. As the lines 22 to 26 already suggests, this method is recursive, meaning it calls itself over and over again, until the highest p-value is lower than a user specified alpha level or the amount of values left is 2. The most important line of the snippet above is line 13 which performs the chi-square comparison. A more detailed insight in the `compareChiSquare` function is given in the snippet below:

```
1  public double compareChiSquare(AttributeValue p2) {
2     double chiSquareScore = 0;
3     int degreesOfFreedom = 1;
4     int[] clickRateSums = new int[2];
5     for(int i = 0; i < clickRateSums.length; i++) {
```

```
6       clickRateSums[i] = this.getDataPerRecommendation()[i] +
            p2.getDataPerRecommendation()[i];
7       if(clickRateSums[i] == 0) {
8          degreesOfFreedom--;
9          return 1;
10      }
11   }
12   for(int i = 0; i < this.getDataPerRecommendation().length; i++) {
13      double expected = (clickRateSums[i]*this.getSum())/(this.getSum() +
            p2.getSum());
14      double actual = this.getDataPerRecommendation()[i];
15
16      if(clickRateSums[i] != 0) {
17         chiSquareScore+= chiSquareCheck(actual, expected);
18      }
19   }
20   for(int i = 0; i < p2.getDataPerRecommendation().length; i++) {
21      double expected = clickRateSums[i]*p2.getSum()/(this.getSum() +
            p2.getSum());
22      double actual = p2.getDataPerRecommendation()[i];
23
24      if(clickRateSums[i] != 0) {
25         chiSquareScore+= chiSquareCheck(actual, expected);
26      }
27   }
28   return PValueCalculator.pochisq(chiSquareScore, degreesOfFreedom);
29 }
```

This code snippet performs a chi-square distribution test of independence as described in section 3.3.2, where the *this*-Object represents the first variable and `p2` the second. The first for-loop from line 5 to 11 calculates the total of recommended / not recommended events among the two attribute values. The last two loops from line 12 to 27 calculate the chi-square value by adding up the fractions of the formula, which is shown in section 3.3.2. Lastly the function returns a p-value calculated from the chi-square score and the amount of degrees of freedom which were counted throughout the procedure of evaluating the chi-square score.

Splitting a Node

As already mentioned in section 3.3.3 the best split is determined by calculating an adjusted version of the p-value of each attribute and finding the highest one. The code snipped below shows the code-wise implementation. This code snippet shows a `for-loop` iterating over all attributes of the root node. The prune method in line 2 was already described in the section above. Line 3 performs a p-value calculation with taking account of the Bonferroni-Correction. The rest of the code serves to find the attribute with the highest adjusted corresponding p-value and saves it.

```
for(ChaidAttr p : this.prunedChaidAttrs) {
   p.prune();
   double adjPVal = p.getAdjustedPValue();
   if(minPVal == -1 || adjPVal < minPVal) {
      minPVal = adjPVal;
      this.splitBy = p;
   }
}
```

After the best split is determined the algorithm can split up the chosen attribute as seen in the following code snippet. As seen in this excerpt the program creates a new child node for each value of the attribute chosen for the split. Also for each child node the split function is recursively.

```
List<ChaidAttr> newChaidAttrs = new ArrayList<>();
for(ChaidAttr p : this.originalChaidAttrs) {
   if(p.getAttrClass() != this.splitBy.getAttrClass()) {
      newChaidAttrs.add(p);
   }
}
for(AttributeValue v : this.splitBy.getValues()) {
   List<AlgorithmDTO> newTrainingsData = new ArrayList<>();
   for(AlgorithmDTO e : this.trainingsData) {
      if(v.test(e)) {
         newTrainingsData.add(e);
      }
   }
   Chaid child = new Chaid(newTrainingsData, newChaidAttrs, this.level+1);
   child.setLabel(v);
   if(child.prunedChaidAttrs.size() >= 2) {
      child.split();
   }
   else {
      child.isTerminal = true;
   }
   children.add(child);
}
```

Chapter 4

Test Environment

4.1 General

The test environment is a web application for applying, testing and comparing the three big data algorithms. The environment is divided into a frontend section, written in HTML and JavaScript and the backend section, written in Java.

This chapter starts with a short introduction of the used frameworks. After this the overall structure of the test environment is described and the database gets explained. Finally a somewhat elaborate list of features including code examples, where necessary, is given.

4.2 Used Frameworks

4.2.1 Facebook SDK

To crawl real events for testing, the Facebook SDK [Fac18] was required to provide the login feature. With the login-generated user token, the app uses the crawl algorithm taken from *Ennui* to get Facebook events.

4.2.2 jqWidgets

The Javascpript/HTML5-Framework jqWidgets [jQW17] was used for generating filterable and well designed tables. jqWidgets also provides pop-up windows used for adding or editing events. The choice to use jqWidgets was made because of the previous use in the summer internship. The following components were used:

- jqx.dataAdapter
- jqxGrid
- jqxWindow
- ThemeBuilder

jqx.dataAdapter

The jqxDataAdapter represents a jQuery plug-in which simplifies data binding and data operations and supports binding to local and remote data. It was used for data binding from the JSON-Response and preparing it to be shown in the jqxGrid.

The following code example shows how the `dataAdapter` maps items from a JSON-Array to an object prepared to be shown in the `jqxGrid`. The path of the field gets provided via the `map` key and is only necessary if the datafield is nested in the object structure.

```
1  function initTable() {
2    $.getJSON(
3      "http://192.168.137.1:8080/events/nearby?latitude=" +
4        usrLat +
5        "&longitude=" +
6        usrLong +
7        "&country=AT&radius=400&start_time=0&end_time=0",
8      function(data) {
9        eventData = data;
10       var eventSource = {
11         datatype: "json",
12         datafields: [
13           { name: "id", type: "int" },
14           { name: "name", type: "string" },
15           { name: "owner", map: "owner>name", type: "string" },
16           { name: "description", type: "string" },
17           { name: "dummyStatus", type: "boolean" },
18           { name: "city", map: "place>location>city", type: "string" },
19           { name: "country", map: "place>location>country", type: "string" }
20         ],
21         id: "id",
22         localdata: data
23       };
24       var eventAdapter = new $.jqx.dataAdapter(eventSource);
```

jqxGrid

The jqxGrid is a jQuery widget that displays tabular data. It offers support for interacting with data, including paging, grouping, sorting, filtering and editing.

The grid itself uses the data provided by the jqxDataAdapter called `eventAdapter`, as explained above. With the `selectionmode` key, the developer can define how many rows the user is able to select at once. With `singlerow` as the value, only one row can be selected.

With the key `filterable` set to true, the user is able to filter fields in the table.

The filtering itself is textual, it shows events that contain the string-expression provided in the text field.

With the key `sortable` set to true, the user is able to sort the string values provided in the columns either ascending or descending.

```
            $("#eventGrid").jqxGrid({
          width: "80%",
          source: eventAdapter,
          height: "80%",
          selectionmode: "singlerow",
          showfilterrow: true,
          filterable: true,
          sortable: true,
          columnsresize: true,
          rowsheight: 36,
          columns: [
            { text: "Name", datafield: "name" },
            { text: "Country", datafield: "country" },
            { text: "City", datafield: "city" },
            { text: "Owner", datafield: "owner" },
            { text: "DummyStatus", datafield: "dummyStatus" },
            { text: "Description", datafield: "description" },
            { text: "Edit", cellsrenderer: editRenderer },
            { text: "Delete", cellsrenderer: deleteRenderer }
          ]
        });
```

With adding the `cellsrenderer` key to a column definition, a javascript method can be provided that defines the design and functionality of the column.

```
function editRenderer(row, col, val, element, properties, rowData) {
      return (
        '<input class="center" type="button" onclick = editEventPopUp(' +
        rowData.id +
        ') value="Edit"/>'
      );
    }
    function deleteRenderer(row, col, val, element, properties, rowData) {
      return (
        '<input class="center" type="button" onclick = deleteEvent(' +
        rowData.id +
        ') value="Delete"/>'
      );
    }
```

jqxWindow

The *jqxWindow* was used to provide the pop-ups for editing and adding data via CSV. To implement it, a `<div></div>` element in the HTML document has to be created. The window is not shown until the user clicks the button. When clicking the button, a

pop-up window opens, with the elements coded in the div.

```
$("#popupWindowCSV").jqxWindow({
    width: parseInt(winWidth / 100 * 60),
    height: parseInt(winHeight / 100 * 60),
    minWidth: 200,
    minHeight: 400,
    maxWidth: null,
    maxHeight: null,
    resizable: true,
    draggable: true,
    isModal: true,
    autoOpen: false,
    showCloseButton: true
  });
```

Theme Builder

The *Theme Builder* is a tool for creating new themes for jqWidgets. After choosing color, widget design and templates, a `.css` file is generated, which supplies all settings made in the theme builder.

4.2.3 Spring Boot

The Java framework *Spring Boot* [Spr18] was used to embed Apache Tomcat and to have a valid *Rest Service Component* with request mapping. The choice to use Spring Boot was made because it is also used in the backend of Ennui.

4.3 Overall Structure

4.3.1 Backend

The backend of the test environment is a Spring Boot application written in Java. Codewise, parts of the backend of Ennui were reused, for example the *Crawling Machine*. The Spring Boot application is divided in the following components:

- Users : Everything user-related, like the *Login* feature.
- Algorithms : The implementations of the algorithms.
- Events : Everything event-related (adding, editing, deleting, getting the events from the database)

All components have their own:

Rest-Controller: The frontend is able to communicate with the backend over the *Rest-Controller* class. This happens with a *HTTP-Request* to an URL, for example: `http://192.168.137.1:8080/algorithms/applyC45`

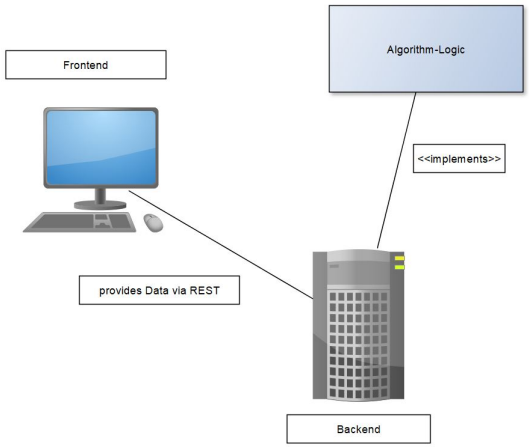


Figure 4.1: Structure of Test-Environment

For the web application there are two forms of *HTTP-Requests* implemented:

- HTTP-GET: The *HTTP-GET* method is used to retrieve information, for example to retrieve all Events in the Database
- HTTP-POST: The *HTTP-POST* method is used to send data to the server, for example a new event to add into the database.

To make the backend recognize that a call is sent from the frontend, request mapping has to be done. This happens with the annotation `@RequestMapping('path')` by *Spring*.

Once a Rest-Controller method is called via request mapping, Code is executed in the service class.

DTO: The *DTO*, which stands for *Data Transfer Object* defines the attributes of each component. All the informations provided by POST or GET requests, get converted to DTOs, to work with them in the service class.

Service: The service class gets called by the Rest-Controller to execute the requests.

4.3.2 Frontend

The frontend of the test environment is an HTML5/Javascript web application, with jqWidgets as a framework for generating the user interface. In opposite to the backend, the frontend is not based on components of Ennui, because the main purpose of the test environment is to be simple and functional, instead of a well designed, but complex user interface with unnecessary features for testing.

4.4 Database

For saving all the crawled events, the Facebook pages where the data mining algorithm gets the events from and the user information *ObjectDB* was used [Obj18]. ObjectDB is an object database for Java. It supports the two standard Java APIs, *JPA (Java Persistence API)* and *JDO (Java Data Objects)*. In case of the test environment, JPA was used for communicating with the database.

4.4.1 Java Persistence API (JPA)

The following explanation of the Java Persistence API will be backed up with information taken from a JPA tutorial [Vog18].

General

The Java Persistence API is one possible approach to *Object-relational mapping (ORM)*. This means that the developer can map, store, update and retrieve data from relational databases to Java objects and vice versa.

JPA permits the developer to work directly with objects rather than with SQL statements. The mapping between Java objects and database tables is defined via persistence metadata. The JPA provider will use the persistence metadata information to perform the correct database operations.

JPA metadata is typically defined via annotations in the Java class. Alternatively, the metadate can be defined via XML or a combination of both. A XML configuration overwrites the annotations.

Entity

A class which should be persisted in a database must be annotated with `@Entity`. JPA creates a database table for every entity. Persisted instances of the class will be represented as one row in the table.

```
1 @Id
2 @GeneratedValue
3 private long id;
```

All entity classes must define a primary key value. JPA allows to auto-generate the primary via the `@GeneratedValue` annotation. This can be seen in the example above.

Persistence of fields

The fields of the entity will be saved in the database. JPA can use either the instance variables (fields) or the corresponding getters and setters to access the fields. Mixing both methods is not allowed. JPA persists per default all fields of an Entity, if fields should not be saved they must be marked with the `@Transient` annotation.

By default each field is mapped to a column with the name of the field. The default name can be changed via the `@Column (name="newColumnName")` annotation.

Relationship Mapping

JPA allows to define relationships between classes. Classes can have one to one, one to many, many to one, and many to many relationships with other classes.

A relationship can be bidirectional or unidirectional. In a bidirectional relationship both classes store a reference to each other while in an unidirectional only one class has a reference to the other class. In the test environment, only unidirectional relationships are used.

```
1 @OneToMany(fetch = FetchType.EAGER)
2 private Object owner;
3 private String description;
4 @OneToMany(fetch = FetchType.EAGER)
5 private Object place;
6 @OneToMany(fetch = FetchType.EAGER)
7 private List<String> categories;
```

A one to many relationship is for example annotated with `@OneToMany`. In this annotation, the `FetchType` can also be defined. In the test environment, the type `FetchType.EAGER` is used. This defines that an attribute from another class is automatically loaded when a entity gets fetched from the database.

Entity Manager

The entity manager provides the operations from and to the database, for Example finding objects, persisting them, or removing objects from the database.

Entities which are managed by an `EntityManager` will automatically propagate these changes to the database (if this happens within a commit statement)

In the following code example, a database operation with the entity manager will be shown.

```
1  public void updateEvent(EventDTO eventDTO){
2        EntityManager em = getEntityManager();
3        EventDTO old = em.find(EventDTO.class,eventDTO.getId());
4        em.getTransaction().begin();
5        old.setName(eventDTO.getName());
6        old.setPlace(eventDTO.getPlace());
7        old.setOwner(eventDTO.getOwner());
8        old.setDescription(eventDTO.getDescription());
9        if (eventDTO.getStart_time() != 0){
10           old.setStartTimeAsLong(eventDTO.getStart_time());
11       }
12       if (eventDTO.getEnd_time() != 0){
13           old.setEnd_timeAsLong(eventDTO.getEnd_time());
```

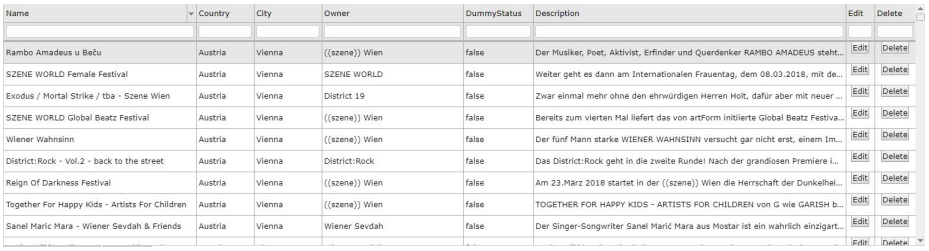


Figure 4.2: The event list shown in a table

```
14          }
15          em.getTransaction().commit();
16          em.close();
17      }
```

Persistence units

The `EntityManager` is created by the `EntityManagerFactory` which is configured by the persistence unit. The persitence is described via the `persistence.xml` file. A set of entities which are logical connected will be grouped via a persistence unit. The `persistence.xml` file defines the connection data to the database, for example the driver, the user and the password.

4.5 Code/Feature Explanation

4.5.1 Event List

As seen in Figure 4.2, the events crawled via the Facebook API get shown in a table generated by jqWidgets. The table itself is filterable and sortable. The following information is shown:

- Name: The name of the event
- Country: The country in which the event happens.
- City: The city, the event is located in
- Owner: The name of the operator creating the event
- DummyStatus: Shows if the event is real or added by the developer
- Description: Shows the description for the event.

The EventList is a key feature of the test environment, letting the user watch all Events that he can apply his algorithm on and editing or deleting them.

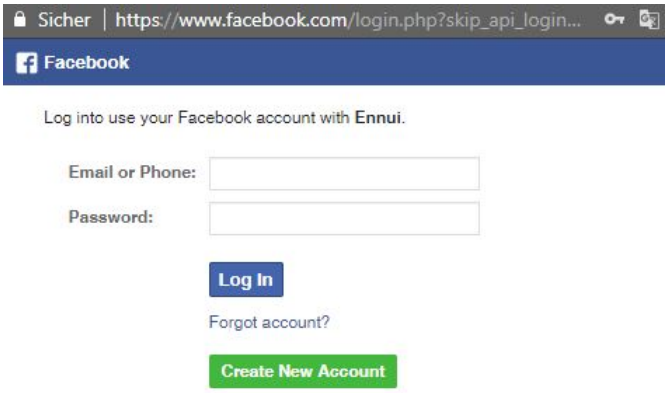


Figure 4.3: The Login-Feature provided by the FacebookAPI

4.5.2 Facebook Login

As seen in figure 4.3, the Facebook login provided by the Facebook API is used for registration and login purposes. With the user logging in via Facebook, his name, email address, gender, age_range (displaying the age of the user) and most important the generated `accessToken` are getting sent to Ennui. With the access-token the crawling algorithm is able to access the pages liked by the user. These pages get added to the page list in the database and get crawled when the next scheduled crawling task starts. The next algorithm shows the Facebook login and how the login token is sent to the backend.

```
1  function logInWithFacebook() {
2    FB.login(function(response) {
3      if (response.authResponse) {
4        accessToken = response.authResponse.accessToken;
5        FB.api(
6          "/me?fields=first_name,last_name,name,email,gender,age_range",
7          function(response) {
8            name = response.name;
9            $("#userInfo").text("Erfolgreich eingeloggt als: " + name);
10           jQuery.ajax({
11             headers: {
12               Accept: "application/json",
13               "Content-Type": "application/json"
```

```
            },
            type: "POST",
            url: "http://192.168.137.1:8080/users/login",
            data: JSON.stringify({ fbToken: accessToken }),
            dataType: "json",
            success: function success(response) {
              alert("Login successful");
            }
          });
        }
      );
    } else {
      alert("User cancelled login or did not fully authorize.");
    }
  });
  return false;
}
```

The function starts with sending a call to Facebook, requesting the needed data. If the login succeeds, the data is sent as a response and can be handled by the application. Moving on, a HTTP-POST is made towards the backend, providing the `accessToken` received. If the user is not already registered, he gets inserted into the database with his personal data mentioned and his page likes get saved into the `pagesToCrawl` list.

4.5.3 Add Event

To add own events, `Add Event` was developed as a feature. The user is able to enter all relevant data, consisting of the name of the event, start time and end time, location with country, zip code, city, street, the operator that created the event and a event description. The last option for the user is to choose if he wants the event to be a *dummy event*, which is an event especially created to test the algorithm, or just a normal event. The user interface for this feature can be seen at figure 4.4.

The following code snippet shows how the event data gets sent to the backend.

```
function addEvent() {
  var date = new Date($("#inputStartTimeDate").val());
  var stime = date.getTime() / 1000 +
    parseInt($("#inputStartTimeTime").val().split(":")[0]) * 60 *
    60 + parseInt($("#inputStartTimeTime").val().split(":")[1]) *60;
  var edate = new Date($("#inputEndTimeDate").val());
  var etime = edate.getTime() / 1000 +
    parseInt($("#inputEndTimeTime").val().split(":")[0]) * 60 *
    60 + parseInt($("#inputEndTimeTime").val().split(":")[1]) * 60;
  var requestBody = {
    fbToken: accessToken,
    name: $("#inputName").val(),
    owner: $("#inputOwner").val(),
    description: $("#inputDescription").val(),
```

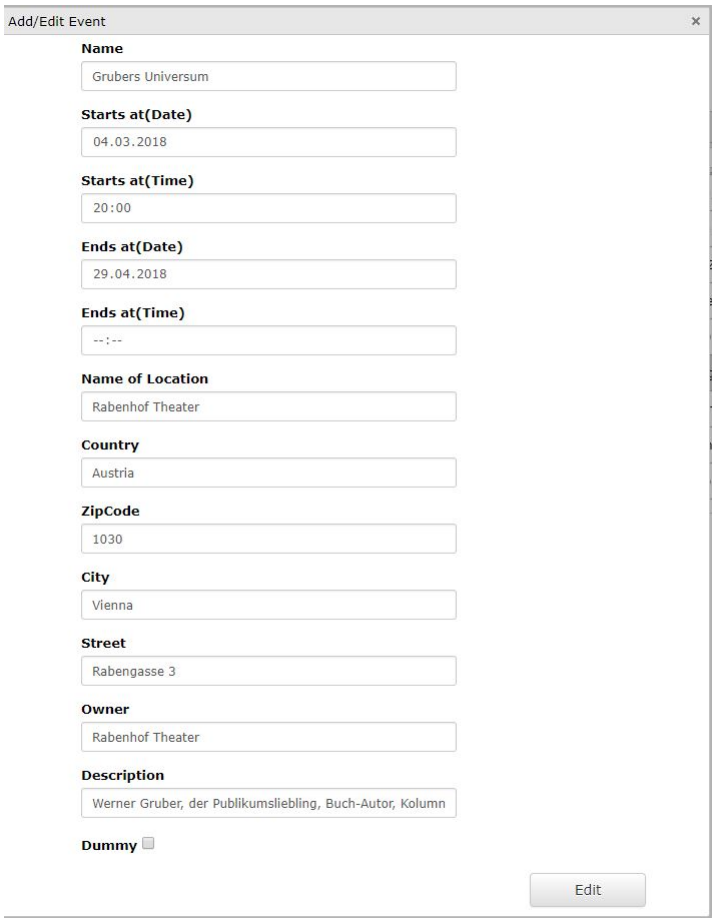


Figure 4.4: Add/Edit Event

```
15      categories: [""],
16      website: "#",
17      country: $("#inputCountry").val(),
18      start_time: stime,
19      end_time: etime,
20      city: $("#inputCity").val(),
21      postal: $("#inputZipCode").val(),
22      street: $("#inputStreet").val(),
23      place_name: $("#inputPlaceName").val(),
24      longitude: usrLong,
25      latitude: usrLat,
26      dummyStatus: document.getElementById("inputDummy").checked
27    };
28    post("http://192.168.137.1:8080/events/add", requestBody, function(response)
          {
29      alert("Successfully added Event!");
30    });
31  }
```

After collecting the data from the input fields and converting them into the right format (lines 2 to 27), an `HTTP-POST` call (line 28) gets sent to the backend. After getting a response, an alert is created to let the user know that his created event has been successfully added to the database of the test environment (line 29).

4.5.4 Edit Event

Different to Ennui, all events can be edited in the test environment. The user finds the same layout as for the `Add Event` feature (see figure 4.4), but with the filled event data. Afterwards the user is able to make changes to all fields and submit them.

```
1   function editEvent(oldCategories, oldId) {
2     var date = new Date($("#inputStartTimeDate").val());
3     var stime = date.getTime() / 1000 +
4       parseInt($("#inputStartTimeTime").val().split(":")[0]) * 60 *
5       60 + parseInt($("#inputStartTimeTime").val().split(":")[1]) * 60;
6     var edate = new Date($("#inputEndTimeDate").val());
7     var etime = edate.getTime() / 1000 +
8       parseInt($("#inputEndTimeTime").val().split(":")[0]) * 60 *
9       60 + parseInt($("#inputEndTimeTime").val().split(":")[1]) * 60;
10    var requestBody = {
11      fbToken: accessToken,
12      id: oldId,
13      name: $("#inputName").val(),
14      owner: $("#inputOwner").val(),
15      description: $("#inputDescription").val(),
16      categories: [oldCategories],
17      website: "#",
18      country: $("#inputCountry").val(),
```

```
19      start_time: stime,
20      end_time: etime,
21      city: $("#inputCity").val(),
22      postal: $("#inputZipCode").val(),
23      street: $("#inputStreet").val(),
24      place_name: $("#inputPlaceName").val(),
25      longitude: usrLong,
26      latitude: usrLat,
27      dummyStatus: document.getElementById("inputDummy").checked
28    };
29    post("http://192.168.137.1:8080/events/edit", requestBody, function(
30      response
31    ) {
32      alert("Successfully edited Event!");
33    });
34  }
```

As seen in the code above from line 2 to 28, this feature also collects the data from the Input-Fields and converts them into the right format. Then a HTTP-POST call gets sent to the backend (line 29). After getting a response, an alert is created to let the user know that his edited event has been successfully updated in the database of the test environment (line 30 to 34).

Another interesting addition in comparison to the Add-Event feature is the fact, that the Edit-Event can be called for every event through the event table. This is realized with the `cellsrenderer` attribute provided by the jqxGrid. In this attribute, the design and function of the field is saved. In the case of Edit-Event, a button is created that has an `onClick` method that hands the rowId to the pop-up-preparation-method called `editEventPopUp()`. There, all information needed for the form gets fetched and filled in the right fields. The following function is the `cellsrenderer` for the *Edit* button in the event list.

```
1  function editRenderer(row, col, val, element, properties, rowData) {
2        return (
3          '<input class="center" type="button" onclick = editEventPopUp(' +
4          rowData.id +
5          ') value="Edit"/>'
6        );
7      }
```

4.5.5 Add Event as CSV

Because there are many dummy events needed to test the algorithms, a solution for adding several events at once was needed. This was realized with this feature. CSV was chosen as the input because it can easily be created via Microsoft Excel for example and is also the format for data input in Weka (see 2.6.3). The user interface for this feature can be seen in figure 4.5.

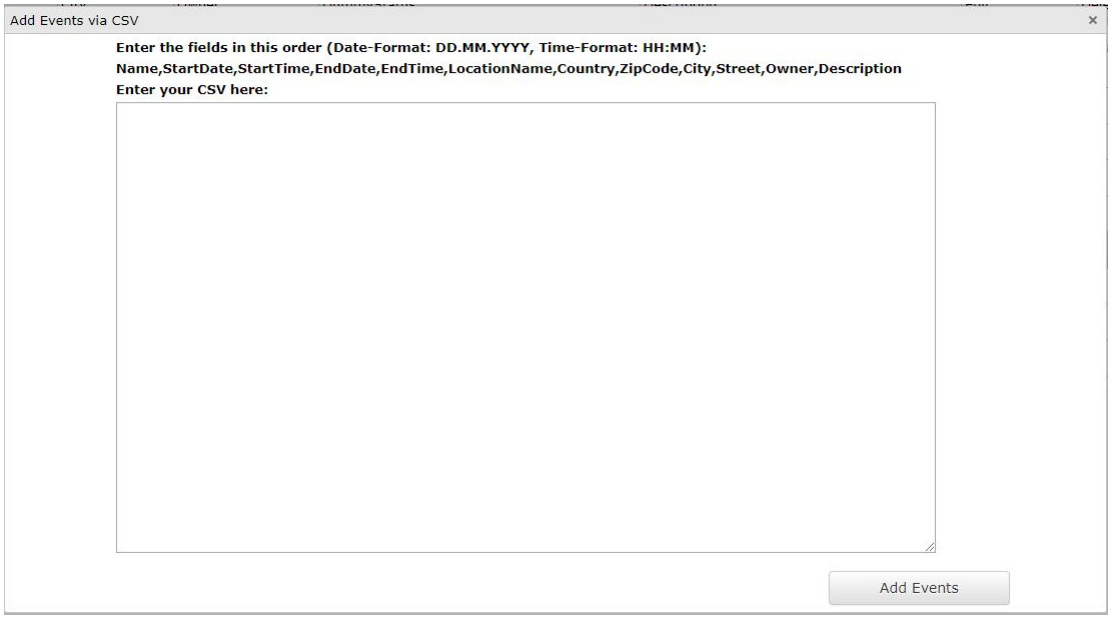


Figure 4.5: Add Events via CSV

```
1     $("#submitCSVButton").on("click", function() {
2   var allText = $("#textAreaCSV").val();
3   var lines = new Array();
4   var allTextLines = allText.split(/\r\n|\n/);
5   for (var i = 0; i < allTextLines.length; i++) {
6     var data = allTextLines[i].split(",");
7     var row = [];
8     for (var j = 0; j < data.length; j++) {
9       row.push(data[j]);
10    }
11    lines.push(row);
12  }
13  $.each(lines, function(index, data) {
14    var dateConvert = data[1].split(".");
15    var date = new Date(
16      parseInt(dateConvert[2]),
17      parseInt(dateConvert[1]) - 1,
18      parseInt(dateConvert[0])
19    );
20    var stime =
21      date.getTime() / 1000 +
22      parseInt(data[2].split(":")[0]) * 60 * 60 +
23      parseInt(data[2].split(":")[1]) * 60;
24    dateConvert = data[3].split(".");
```

```
25        var edate = new Date(
26          parseInt(dateConvert[2]),
27          parseInt(dateConvert[1]) - 1,
28          parseInt(dateConvert[0])
29        );
30        var etime =
31          edate.getTime() / 1000 +
32          parseInt(data[4].split(":")[0]) * 60 * 60 +
33          parseInt(data[4].split(":")[1]) * 60;
34        var requestBody = {
35          fbToken: accessToken,
36          name: data[0],
37          owner: data[10],
38          description: data[11],
39          categories: [""],
40          website: "#",
41          country: data[6],
42          start_time: stime,
43          end_time: etime,
44          city: data[8],
45          postal: data[7],
46          street: data[9],
47          longitude: usrLong,
48          latitude: usrLat,
49          dummyStatus: true
50        };
51        post("http://192.168.137.1:8080/events/add", requestBody, function(
52          response
53        ) {
54          alert("Successfully added Event " + index + "!");
55        });
56      });
57    });
```

Reading a csv file and converting it to an array is taken from [Sta17]. The solution found worked with the `.split()` method provided by JavaScript (lines 4 to 12). The resulting array generated through the split is then saved into the object that gets posted to the backend as a response body (line 51).

Every event then gets added to the database separately, if there is a problem with the event data, the user gets a message that the event with a specific id could not be posted to the backend.

4.5.6 Apply Algorithm

For all three algorithms (C4.5, CART, CHAID), this feature has had to be implemented. The user is able to filter the event list displayed in the table for many attributes, listed in section 4.5.1. If the user then presses one of the three `Apply <Name of Algorithm>`

buttons, the IDs of all events of the filtered list get posted to the backend.

In the backend, all events that have an ID of the list that got sent to the backend get fetched from the database. After that, the fetched `EventDTOs` have to be converted to `AlgorithmDTO`. This happens in the code below:

```
List<AlgorithmDTO> data = new ArrayList<>();
        for(EventDTO event : events){
            AlgorithmDTO add = new AlgorithmDTO();
            add.setStart_time_converted(new
                java.util.Date((long)event.getStart_time()*1000));
            add.setCategory(event.getCategories().get(0));
            if (event.getPlace() instanceof HashMap){
                HashMap location = (HashMap) ((HashMap)
                    event.getPlace()).get("location");
                add.setCity(location.get("city").toString());
                add.setOperator((String)
                    ((HashMap)event.getPlace()).get("name"));
            }
            add.setName(event.getName());
            add.setId(event.getId());
            Calendar calendar = GregorianCalendar.getInstance(); // creates a
                new calendar instance
            calendar.setTime(add.getStart_time_converted()); // assigns
                calendar to given date
            add.setStart_hour(calendar.get(Calendar.HOUR_OF_DAY)); // gets hour
                in 24h format
            int day = calendar.get(Calendar.DAY_OF_WEEK);
            String dayStr;
            String weekStr;
```

First, fields that stay the same in the `AlgorithmDTO`, like the category, the location, the operator, the id, and the name of event get assigned to the new object. After that, a instance of `Calendar` gets created with the `Date` of the event. From the `Calendar` the hour on which the event starts can be extracted and the day of the week when the event happens.

```
switch(day){
                case 1:
                    dayStr="Sun";
                    weekStr="weekend";
                    break;
                case 2:
                    dayStr="Mon";
                    weekStr="workday";
                    break;
                case 3:
                    dayStr="Tue";
                    weekStr="workday";
```

```
13                  break;
14              case 4:
15                  dayStr="Wed";
16                  weekStr="workday";
17                  break;
18              case 5:
19                  dayStr="Thu";
20                  weekStr="workday";
21                  break;
22              case 6:
23                  dayStr="Fri";
24                  if (add.getStart_hour() >= 14){
25                      weekStr="weekend";
26                  }
27                  else{
28                      weekStr="workday";
29                  }
30                  break;
31              case 7:
32                  dayStr="Sat";
33                  weekStr="weekend";
34                  break;
35              default:
36                  dayStr="undefined";
37                  weekStr ="undefined";
38                  break;
39
40          }
```

With the extracted day of the week, a `switch` statement is made. In the statement, the `day` and `part of the week` attribute get assigned their values. On Friday, the split between `workday` and `weekend` happens at 2 p.m. which equals 14 in the 24 hours format.

```
1           String description ="";
2           if (event.getDescription() != null) {
3               description = event.getDescription().toLowerCase();
4           }
5           String tag = "";
6           switch(add.getCategory()){
7               case "party - nightlife":
8                   add.setCategory("party");
9                   tag="club";
10                  break;
11              case "music":
12                  tag="concert";
13                  break;
14              case "art - culture":
15                  add.setCategory("art");
16                  tag="art_exhibit";
```

```
17                  break;
18              case "literatur":
19                  add.setCategory("literature");
20                  tag="reading";
21                  break;
22              case "comedy":
23                  tag="cabaret";
24                  break;
25              case "food":
26                  if (description.contains("streetfood")){
27                      tag="streetfood";
28                  }
29                  else if(description.contains("dinner")){
30                      tag="gala_dinner";
31                  }
32                  else{
33                      tag="food";
34                  }
35                  break;
36              case "health":
37                  tag="health";
38                  break;
39              case "shopping":
40                  tag="shopping";
41                  break;
42              case "home - garden":
43                  add.setCategory("home");
44                  tag="shopping";
45                  break;
46              case "sport":
47                  if (description.contains("fussball") ||
                        description.contains("kicker") ||
                        description.contains("soccer"){
48                      tag="football";
49                  }
50                  else if (description.contains("marathon") ||
                        description.contains("laufen")){
51                      tag="running";
52                  }
53                  else{
54                      tag="sport";
55                  }
56                  break;
57              case "theatre":
58                  if (description.contains("musical")){
59                      tag="musical";
60                  }
61                  else{
62                      tag="theatre";
```

```
63                  }
64                  break;
65              default:
66                  tag="other";
67                  break;
68          }
69          add.setAdditional_tag(tag);
70          add.setWeek(weekStr);
71          add.setWeekday(dayStr);
72          data.add(add);
73      }
74      return data;
```

Lastly, the additional tag attribute gets generated out of the description of the event. Because it would not be reasonable to pass the description as an attribute to the algorithms, the relevant information has to be filtered with searching for `String` values that are typical for the tag. This has to be done in categories that are more general, like `sport` for example. For other categories, only one tag is available, that just gives additional information to the category. In some cases there is also the need to change the category names provided by Facebook to shorter names, for example at line 8 of the code above. After that, the generated values get assigned to the new object and the object gets added to a `List<AlgorithmDTO>`. After the conversion of all events, the backend sends the `AlgorithmDTOs` to the algorithms for classifying them.

After classification, the IDs of the events that are recommended by the algorithm get sent to the frontend of the test environment. The events with the IDs get fetched from the local array and displayed in a table. The code for displaying the recommended events in the table is shown below. The structure of the table is equal to the event list.

```
1  function initAlgorithmTable(ids) {
2      var data = new Array();
3      $.each(ids, function(index, value) {
4        var index = eventData.findIndex(function(x) {
5          return x.id == value;
6        });
7        data.push(eventData[index]);
8      });
9      var algorithmSource = {
10       datatype: "json",
11       datafields: [
12         { name: "id", type: "int" },
13         { name: "name", type: "string" },
14         { name: "owner", type: "string" },
15         { name: "description", type: "string" },
16         { name: "dummyStatus", type: "boolean" },
17         { name: "city", map: "place>location>city", type: "string" },
18         { name: "country", map: "place>location>country", type: "string" }
19       ],
```

```
20        id: "id",
21        localdata: data
22      };
23      var algorithmAdapter = new $.jqx.dataAdapter(algorithmSource);
24
25      $("#algorithmGrid").jqxGrid({
26        width: "80%",
27        source: algorithmAdapter,
28        height: "80%",
29        selectionmode: "singlerow",
30        showfilterrow: true,
31        filterable: true,
32        sortable: true,
33        columnsresize: true,
34        rowsheight: 36,
35        columns: [
36          { text: "Name", datafield: "name" },
37          { text: "Country", datafield: "country" },
38          { text: "City", datafield: "city" },
39          { text: "Owner", datafield: "owner" },
40          { text: "DummyStatus", datafield: "dummyStatus" },
41          { text: "Description", datafield: "description" }
42        ]
43      });
44    }
```

Chapter 5

Simulation Results

5.1 Introduction

For finding the best fitting algorithm to implement into the Ennui backend, a two step test was made. All three algorithms were tested in the test environment, as well as on the running system.

The main differences between the two steps was the number of events and the different test set.

The first step was the implementation of each algorithm into the test environment. This includes preparing the provided event data for parsing it into the algorithm. To see which fields were relevant for calculating the best fitting spare time activities, look at section 1.3.5.

```
@relation ennui_training
@attribute 'Day' { 'Mon','Tue','Wed','Fri','Thu','Sat','Sun'}
@attribute 'Start Time' numeric
@attribute 'Location' {'Linz'}
@attribute 'Veranstalter' { 'der Hafen','A1 Musikpark','tips-Arena',
@attribute 'Category' { 'party','music','art','literature','comedy',
@attribute 'Week' {'workday','weekend'}
@attribute 'additional_tag' {'club','festival','concert','reading','
@attribute 'Recommended' { 'no', 'yes'}
@data
'Sat',20,'Linz','der Hafen','party','weekend','club','yes'
'Fri',20,'Linz','A1 Musikpark','party','weekend','club','yes'
'Wed',20,'Linz','der Hafen','party','workday','club','no'
```

Figure 5.1: Training set

5.2 Step 1 - Test Environment

5.2.1 Explanation of the Used Training Set

A training set is needed to give the machine learning algorithms data to learn from. For more general information see 2.2.1. With providing a hard-coded training set, the interesting part was watching different approaches of the algorithms to build trees from the given data.

In the case of Ennui, the last field has the name `recommended`, and takes two values, `yes` or `no`. With this attribute the developer tells the algorithm which events he should recommend, based on simulated habits of a test user. Learning from this decisions, based on different event data, the algorithm is now able to build a decision tree, where all leaf nodes have either `yes` or `no` as a value. For the use on the running system, the training set is generated from user clicks. In the test environment the training set with the simulated habits is only created for testing and comparing the algorithms, this would make no sense in a production system, because there would be no adaption to the habits of the user.

After experimenting with Weka (see 2.6), the writer came to the conclusion, that with 3 instances of data for the same category, machine learning algorithms can make valid decisions on the recommendation without the risk of overfitting. With this information, a training set with 36 instances of training data with the last attribute being either `yes` or `no` has been built.

Looking at figure 5.1, it can be seen how the training set is constructed for making the algorithm learn how to decide. As mentioned before, the test person likes to go to events with the category `party`, but only on weekends. To represent his affinity with events of this section, two events happening on different places at the weekend are recommended. The third one, is an exact copy of the first line of data, the only difference is the day of the week being Wednesday, a workday. With these three instances, the algorithm is now able to spot that the `week` attribute is the relevant attribute for him to split on for the category `party`.

5.2.2 Criteria for Evaluation

With writing the training set, the developer wanted to point out the simulated habits of a test person. To examine if the algorithms have recognized these habits by the training data, a test set was created. The test set consisted of events, that were different to the events of the training set, but represented the same habits that the algorithm should recognize with the training data provided. The list below shows these habits:

- The affinity for going to parties in clubs on weekends
- Visiting concerts, but only in the evening
- Watching comedy shows, but only on workdays, not on weekends
- The affinity for street-food festivals, not gala dinners

- Going shopping, but only on Saturdays or in the evening (= *Late Night shopping*)
- Watching football games but not taking part actively in any sports competitions

The testing works as follows: The algorithm builds a classifier out of the training data and then looks at the test data. The test data consists of events and the decision of the developer if the event should be recommended or not. The algorithm only gets the event data without the decision made by the developer and classifies it. After classifying the event, he compares his decision with the decision hard-coded in the test set. If the classifier decides equal to the test set, all habits have been successfully recognized. The problem with testing the habits only with a test set is, that the classifier could also make the right decision based on a false assumption. This can easily be explained with an example: The test set contains an event with the category `party` on Friday. The user just likes to go partying, so the event should be recommended because of the category, but the classifier recommends the event because it is on a Friday. If another event with the category `party` would happen on Monday, the classifier would not recommend it, because it is not on a Friday. For this reason it is also necessary to watch the tree structure.

The structure of the trees will be compared based on the correlation with the habits defined above. Another criteria will be the size of the tree. This is simply relevant for performance reasons.

C4.5

```
additional_tag = club
|   Week = workday: no (1.0)
|   Week = weekend: yes (2.0)
additional_tag = festival: no (0.0)
additional_tag = concert
|   Start Time <= 16: no (1.0)
|   Start Time > 16: yes (2.0)
additional_tag = reading: no (3.0)
additional_tag = cabaret
|   Week = workday: yes (1.0)
|   Week = weekend: no (2.0)
additional_tag = art_exhibit: no (3.0)
additional_tag = streetfood: yes (2.0)
additional_tag = food: no (0.0)
additional_tag = shopping
|   Category = party: no (0.0)
|   Category = music: no (0.0)
|   Category = art: no (0.0)
|   Category = literature: no (0.0)
|   Category = comedy: no (0.0)
|   Category = food: no (0.0)
|   Category = games: no (0.0)
```

```
23 |   Category = health: no (0.0)
24 |   Category = shopping
25 |   |   Day = Mon: no (0.0)
26 |   |   Day = Tue: no (0.0)
27 |   |   Day = Wed: no (0.0)
28 |   |   Day = Fri: no (1.0)
29 |   |   Day = Thu: no (0.0)
30 |   |   Day = Sat: yes (1.0)
31 |   |   Day = Sun: no (0.0)
32 |   Category = home: no (3.0)
33 |   Category = sport: no (0.0)
34 |   Category = theatre: no (0.0)
35 |   Category = sonstiges: no (0.0)
36 additional_tag = sport: no (0.0)
37 additional_tag = late_night_shopping: yes (1.0)
38 additional_tag = football: yes (2.0)
39 additional_tag = running: no (1.0)
40 additional_tag = musical: no (1.0)
41 additional_tag = theatre: no (2.0)
42 additional_tag = other: no (1.0)
43 additional_tag = health: no (3.0)
44 additional_tag = gala_dinner: no (1.0)
45 additional_tag = classic: no (0.0)
46 additional_tag = rock: no (0.0)
```

Looking at the tree generated by C4.5 it can be seen, that the size is in comparison to the other ones much bigger. This is caused by the light pruning parameters, which lead to a very accurate but big tree. Another reason for the big tree size is the first split happening on the `additional-tag` attribute because this is the attribute with the most values. The additional splits are situational, on the `week` attribute, `start time`, as well as the `category` attribute, caused by the fact that the additional tag `shopping` is used in the shopping category and the home category, representing exhibitions to shop for home accessoires/furniture.

As clearly seen in the tree, the affinity for visiting clubs on weekends is represented by split between workday and weekend under the tag `club`. The algorithm also has found out that the user only likes going to concerts at the evening, and realizes this decision with a split on `start time` with the split value being 16, which equals 4 p.m. Moving on to the comedy shows, the algorithm does choose the `week` attribute again to show that the user only likes attending to these on workdays. The affinity for street food is represented by the `additional-tag streetfood` as a leaf node with the decision yes. The user also likes going shopping on Saturday, as well as attending Late Night Shoppings, this is also found in the tree. Last, watching football games but not taking part actively in sports is also handled by the `additional-tag` attribute.

With all the criteria fulfilled, it can be seen that the tree may be bigger than the others, but is precise in his prediction based on the training data. The size of the tree may be also relevant in the following performance test.

CART

```
additional_tag=(streetfood)|(late_night_shopping)|(football)|(club)|(concert)
|  Start Time < 14.5: no(1.0/0.0)
|  Start Time >= 14.5
|  |  Day=(Fri)|(Thu)|(Sat)|(Mon)|(Tue)|(Sun): yes(7.0/0.0)
|  |  Day!=(Fri)|(Thu)|(Sat)|(Mon)|(Tue)|(Sun)
|  |  |  Start Time < 19.0: yes(2.0/0.0)
|  |  |  Start Time >= 19.0: no(1.0/0.0)
additional_tag!=(streetfood)|(late_night_shopping)|(football)|(club)|(concert)
|  Category=(shopping)|(comedy)
|  |  Day=(Wed)|(Sat)|(Mon)|(Tue)|(Thu)|(Sun)
|  |  |  Day=(Wed): yes(1.0/0.0)
|  |  |  Day!=(Wed)
|  |  |  |  Start Time < 18.0: yes(1.0/0.0)
|  |  |  |  Start Time >= 18.0: no(1.0/0.0)
|  |  Day!=(Wed)|(Sat)|(Mon)|(Tue)|(Thu)|(Sun): no(2.0/0.0)
|  Category!=(shopping)|(comedy): no(18.0/0.0)
```

Looking at the tree generated by the CART algorithm, we already can see the speciality of it. As more detailed explained in section 3.2 the tree only consists of binary splits, meaning that a parent node never is able two have more than two child nodes. This circumstance makes the algorithm not a perfect fit for the needed solution, caused by the fact that all of the attributes provided, except `start time` have more than two attributes.

This can also be seen in the criteria not being fulfilled. Because of the limited capabilities with only being able to split into two nodes, the algorithm has to work with negations, as seen in the first split on the `additional-tag` attribute. Looking at the first point at the criteria list, the algorithm makes the decision that the user likes attending club events after the `start time` of 14.5 which equals 2:30 pm, on Friday, Thursday, Saturday, Monday, Tuesday and Sunday. This is way too generic and does not represent the wanted result, being a split on the `week` attribute. While the next point, attending concerts but only in the evening is solved strange, but correct, the following one, being watching comedy shows only on weekdays is false.

With the other points also being only partly solved, or completely false, CART has only fulfilled 16,67 percent of the criteria. This already disqualifies the algorithm for being too inaccurate in his classification.

CHAID

```
- Root [splitBy: Week]
   - Workday [splitBy: Category]
      - Party, Art, Literature: 0%
      - Comedy:  100%
      - food:  100%
      - health:  0%
      - shopping: 100%
```

```
      - home:  0%
      - sport:   100%
      - theatre:  0%
   - Weekend [splitBy Category]
      - party [splitBy: Day]
         - fri:     50%
         - sat:     50%
      - music [splitBy: additionalTag]
         - concert:  66%
         - festival: 0%
      - art: 0%
      - liter.:  0%
      - comedy:  0%
      - food:  50%
      - health:  0%
      - shopping: 50%
      - home:  0%
      - sport [splitBy: Day]
         -fri: 0%
         -sat: 100%
      - theatre: 0%
      - sonstige: 0%
```

Looking at the last tree, generated by the CHAID algorithm, a big difference can be spot immediately. Instead of showing `yes` or `no` as the result in a leaf node, the algorithm shows the confidence factor for the decision `yes`, with this factor being above 50 percent the event is recommended. This is really beneficial for filtering for the best fitting events, because there are more distinctions.

Watching the tree, it can be seen that the algorithm fulfils the first point of the criteria, with the split on the `week` attribute, but the confidence factors for recommending the events on weekends are only 50 percent, which does not qualify the event for a recommendation. Moving on, concerts, food and shopping are not correctly classified. This is reasoned by the pruning executed from the algorithm, that makes the tree faster to process, but imprecise.

With adding the point of recognizing the fact that the user likes comedy shows only on workdays, the algorithm only fulfils 33,33 percent of the criteria, making him imprecise.

5.2.3 Comparison of Results

After implementing all three algorithms into the backend of the test environment, the algorithms were tested with the exact same conditions.

In the first test run (see Table 5.1), the writer used a pool of 1638 real events in Austria, crawled via the *Facebook API*. Watching the results of this run several similarities in the results of the C4.5 and CART algorithm can be spotted. This is caused by similar pruning parameters, as well as both using the Weka library for the implementation.

Table 5.1: TestResults of Step 1

Test Results			
	C4.5	CHAID	CART
Events provided	1638	1638	1638
Events recommended	676	355	668
Time needed	114 ms	147 ms	110 ms
Gradations of confidence level	2	5	2

Table 5.2: TestResults 2 of Step 1

Test Results			
	C4.5	CHAID	CART
Events provided	10328	10328	10328
Events recommended	3240	2885	3075
Time needed	510 ms	458 ms	509 ms
Gradations of confidence level	2	5	2

Both algorithms are not specially focused on pruning, which leads to bigger trees, providing less diversity in the confidence level gradations.

To make this point clear, the confidence level is the percent-ratio of `yes` in the exit leaf node. If there is for example a leaf node with 8 events that are recommended and 2 events that are not recommended, the leaf node would have a confidence level of 80 percent.

The best case for a tree is to have a confidence level of 100 percent for a decision in every leaf node. This is the case with a not or only slightly pruned tree, like C4.5 or CART.

CHAID is instead an algorithm specialised in pruning, making the confidence level more diverse, in our result with 5 gradations of confidence levels.

This is positive in the fact that events can be separated easier and less events get recommended, but in our case we have the problem of partly wrong confidence levels, leading to not recommending party events to the user, for example.

To prove this fact, the CHAID algorithm has only half the amount of recommended events, leading to a distinction of the three algorithms in a frame of 19,6 percent of the total event pool.

Looking at the times needed for calculating the recommended events, it can be seen that all algorithms have responded in a timeframe of 100-150 ms. This is totally acceptable and no disqualifier for any of them.

In the second test run (see Table 5.2), the writer used a bigger pool of 10328 real events all around the World, crawled via the Facebook API. Comparing the results to the first testing, it can be seen that the difference between all three algorithms gets

smaller, the distinction of recommended events is shrinked to a frame of 3,4 percent of the total event pool.

Performance-wise all three algorithms respond in a timeframe of 450-510 ms, compared to the first testing, we can see that a event pool having 10 times the size of the first pool takes about 3 to 5 times of the time to be classified. While the CHAID algorithm was the slowest algorithm in classifying the smaller amount of events, he is the fastest in processing the bigger amount. This can be explained by the pruning process. With the smaller decision tree provided by CHAID, the calculation per event takes a much shorter period of time.

5.2.4 Conclusion

After testing all three algorithms in the test environment and analysing the generated trees, the writer came to the conclusion, that C4.5 is the best algorithm with using a hard-coded training set. This is reasoned by having a similar performance and results as the other algorithms, but providing the most precise decision tree.

While the other two algorithms both have had similar or better performance and results, they disqualified with a too imprecise tree, fulfilling under 50 percent of the criteria. Nevertheless, all three algorithms will be additionally tested in Step 2, with the environment being the live system of Ennui.

5.3 Step 2 - Implementation into the Running System

5.3.1 Introduction

For the second step of the study all three algorithms were implemented into the backend of Ennui. The main difference to the first test step was the training set. While the set in step 1 was a hardcoded file on the filesystem, in step 2 it gets generated from user preferences via an algorithm developed in the backend. Another difference is the range of events the algorithm gets. While there were no restrictions to the location of the event in the test environment (unless it was filtered in the table), there are only events provided in the range of 25 kilometres in the running system. With this restriction, the event-pool size shrinks clearly, this helps in several situations. Firstly, the time for classifying is minimized with only providing events that are in the range of the user, secondly only events that are in the users reach are recommended.

5.3.2 Creation of Training Set

To make a training set based on user preferences, the clicks and favorizations of events for each user get saved into the database of Ennui. For every event the user has clicked or favorized, a database entry is created. This entry consists of the following fields:

- ID In every table in a database, each entry has to be identified via an ID. This field has no other meaning.

- UserID: With the *UserID*, the user that clicked the event gets recognized
- EventID: With the *EventID*, the clicked event gets recognized. Further, all needed information for the algorithm (name of the event, start time, category, ...) can be found with the ID of the event.
- Count: *Count* shows how often the event got clicked by the user. A click increments the counter by one, a favorize of the event by 5.

For each click that is tracked, an event gets added to the training set of the user. With this implementation there is one problem that occurs. With only adding the events the user has clicked to the training set, the events the user is not interested in are not covered. But for a complete training set these entries are as essential as the events the user does like. This is caused by the fact, that the algorithms need a set with a good coverage of nearly all available options to provide a valid classification. To solve this problem, an algorithm was developed that first clusters the events the user clicked and then negates them and adds them to the set. This will be explained in the following section.

Negating Existing Events

With the click or setting an event favoured, the user shows his interest in it. With these actions, he defines the section of events in the training set that represent his preferences, leading to the events that get recommended for him. The section that shows events he is not interested in is not represented, so it has to be generated through an algorithm. The goal is to group similar events to so called *clusters*, to minimize the need of generating events. First, all events get grouped on the `additional tag` attribute. This is reasoned by the fact, that this attribute provides the biggest segmentation for the events, with having the most options. The set of events clicked by the user gets scanned for the `additional tag` and saves all used tags in a collection.

```
1 for(AlgorithmDTO event: events) {
2 if (tag.indexOf(event.getAdditional_tag().toLowerCase()) != -1){
3                 if
                      (!coveredTags.contains(event.getAdditional_tag().toLowerCase())){
4                     coveredTags.add(event.getAdditional_tag().toLowerCase());
5                 }
6             }
7         }
```

All events with the same tag now get compared on two attributes: `start time` and `part of the week`. These two attributes were selected, because firstly, `start time` is one of the most relevant information for an event, representing the daytime on that an user wants to to certain activities. Secondly, `part of the week` was chosen because it also provides a generalization of the `day` attribute, which helps making the number of generated events as small as possible. The last attribute not covered is `category`. But

since `additional tag` is just a more refined split of the `category` attribute, this also does not have to be minded.

Negations get generated for a whole event cluster and only if all events of a tag are in the same cluster. If for example all events of a tag are on the weekend, but one event is on workdays, no negation will be generated. This is because it can not be assured, that the user not also likes events of this tag on workday, showing his interest with clicking on it. If all events of the tag are in the same cluster, a negated event is generated. If we continue the example used before, an event of the cluster gets copied, providing the exact same information except the `part of the week` attribute and the class value. Now the algorithm is able to recognize the affinity of the user to only one part of the week and splits on the attribute.

Another important step is to generate events for tags that are not covered through the user clicks. If a user does not click on a single event of a tag, it has to be assumed that the user is not interested in events of it. To provide this information to the algorithms, events get generated. The negated events have different, generated values from the provided options, except for the `additional tag` field and the class value. With this procedure the algorithm will not find any similarities between the generated events and the clicked events, and will split on the `additional tag` field. Because all not covered tags have the class value `no`, this will lead to only recommending events that are covered by user clicks.

```
1   for (int i = 0; i< coveredTags.size(); i ++){
2           String actualTag = coveredTags.get(i);
3           ArrayList<AlgorithmDTO> taggedEvents = new ArrayList<>();
4           events.stream().filter(o ->
                o.getAdditional_tag().equals(actualTag)).forEach(
5               o -> {
6                  taggedEvents.add(o);
7               });
8           AlgorithmDTO reverse = taggedEvents.get(0);
9           reverse = getPartOfWeek(reverse,taggedEvents,"weekend","workday");
10          if (reverse.getRecommended().equals("no") ){
11              double[] instanceValue1 = new double[dataRaw.numAttributes()];
12              instanceValue1[0] = day.indexOf(reverse.getWeekday());
13              instanceValue1[1] = reverse.getStart_hour();
14              instanceValue1[2] =
                    category.indexOf(reverse.getCategory().toLowerCase());
15              instanceValue1[3] = week.indexOf(reverse.getWeek());
16              instanceValue1[4] = tag.indexOf(reverse.getAdditional_tag());
17              instanceValue1[5] = classVal.indexOf("no");
18              dataRaw.add(new DenseInstance(1.0, instanceValue1));
19          }
20          reverse = taggedEvents.get(0);
21          reverse = getTime(reverse,taggedEvents,15);
22          if (reverse.getRecommended().equals("no") ){
```

```
23                double[] instanceValue1 = new double[dataRaw.numAttributes()];
24                instanceValue1[0] = day.indexOf(reverse.getWeekday());
25                instanceValue1[1] = reverse.getStart_hour();
26                instanceValue1[2] =
                      category.indexOf(reverse.getCategory().toLowerCase());
27                instanceValue1[3] = week.indexOf(reverse.getWeek());
28                instanceValue1[4] = tag.indexOf(reverse.getAdditional_tag());
29                instanceValue1[5] = classVal.indexOf("no");
30                dataRaw.add(new DenseInstance(1.0, instanceValue1));
31            }
32            taggedEvents.clear();
33        }
34 tag.removeAll(coveredTags);
35        for (int i = 0; i < tag.size(); i ++){
36            double[] instanceValue1 = new double[dataRaw.numAttributes()];
37            instanceValue1[0] = i%day.size();
38            instanceValue1[1] = 24-i;
39            instanceValue1[2] = i%category.size();
40            instanceValue1[3] = i%2;
41            instanceValue1[4] = oldTags.indexOf(tag.get(i));
42            instanceValue1[5] = classVal.indexOf("no");
43            dataRaw.add(new DenseInstance(1.0, instanceValue1));
44        }
45
46 public AlgorithmDTO getPartOfWeek(AlgorithmDTO ret, List<AlgorithmDTO>
       taggedEvents, String first, String second){
47        ret.setRecommended("yes");
48        if (taggedEvents.stream().filter(o ->
              o.getWeek().equals(first)).findFirst().isPresent()){
49            if (taggedEvents.stream().filter(o ->
                  o.getWeek().equals(second)).findFirst().isPresent() == false){
50                ret.setWeekday("Mon");
51                ret.setWeek("workday");
52                ret.setRecommended("no");
53            }
54        }
55        else if (taggedEvents.stream().filter(o ->
              o.getWeek().equals(second)).findFirst().isPresent()){
56            if (taggedEvents.stream().filter(o ->
                  o.getWeek().equals(first)).findFirst().isPresent() == false){
57                ret.setWeekday("Sat");
58                ret.setWeek("weekend");
59                ret.setRecommended("no");
60            }
61        }
62        return ret;
63    }
64
65
```

```
66      public AlgorithmDTO getTime(AlgorithmDTO ret, List<AlgorithmDTO>
            taggedEvents, int startHour){
67          ret.setRecommended("yes");
68          if (taggedEvents.stream().filter(o -> o.getStart_hour() >=
                startHour).findFirst().isPresent()){
69              if (taggedEvents.stream().filter(o -> o.getStart_hour() <
                    startHour).findFirst().isPresent() == false){
70                  ret.setStart_hour(startHour - 1);
71                  ret.setRecommended("no");
72              }
73          }
74          else if (taggedEvents.stream().filter(o -> o.getStart_hour() <
                startHour).findFirst().isPresent()){
75              if (taggedEvents.stream().filter(o -> o.getStart_hour() >=
                    startHour).findFirst().isPresent() == false){
76                  ret.setStart_hour(startHour);
77                  ret.setRecommended("no");
78              }
79          }
80          return ret;
81      }
```

With this implementation, a nearly complete coverage can be guaranteed. This coverage is needed for building valid decision trees.

5.3.3 Comparison of Generated Trees

Because the training set was not hardcoded any more, the preferences of the user could not be simulated that easily. This also leads to a shrink of the size of all three trees, due to the fact that the training set is bigger and more general. For comparing all three trees a new account had to be created. With this account, many events got clicked/favored leading to a training set of 149 entries. These entries consisted mostly of events from the categories `party`, `music`, `comedy` and `sports`. In the following sections, each tree that got generated will be compared with the preferences in categories of the created user.

C4.5

```
1  additional_tag = club: yes (37.0)
2  additional_tag = festival: no (1.0)
3  additional_tag = concert: yes (31.0)
4  additional_tag = reading: no (1.0)
5  additional_tag = cabaret
6  |   Day = Mon: no (0.0)
7  |   Day = Tue: no (0.0)
8  |   Day = Wed: yes (10.0)
9  |   Day = Fri: yes (10.0)
10 |   Day = Thu: no (0.0)
```

```
|   Day = Sat: no (0.0)
|   Day = Sun
|   |   Start Time <= 17: no (1.0)
|   |   Start Time > 17: yes (10.0)
additional_tag = art_exhibit: no (1.0)
additional_tag = streetfood: no (1.0)
additional_tag = food: no (1.0)
additional_tag = shopping: no (1.0)
additional_tag = sport: yes (32.0/1.0)
additional_tag = late_night: no (1.0)
additional_tag = football: no (1.0)
additional_tag = running: no (1.0)
additional_tag = musical: no (1.0)
additional_tag = theatre: no (1.0)
additional_tag = other
|   Day = Mon: no (2.0)
|   Day = Tue: no (0.0)
|   Day = Wed: no (0.0)
|   Day = Fri: yes (1.0)
|   Day = Thu: no (0.0)
|   Day = Sat: no (0.0)
|   Day = Sun: no (0.0)
additional_tag = health: no (1.0)
additional_tag = gala_dinner: no (1.0)
additional_tag = classic: no (1.0)
```

Watching the decision tree generated by C4.5 it can be seen that it is the biggest one of all three, as in step 1. This is due to the fact that the algorithm splits first on the `additional tag` attribute, leading to more leafs. All preferences in categories are represented by the tree. Additionally, the algorithm has found preferred days for visiting comedy events, Wednesday, Friday and Sunday.

CART

```
additional_tag=(club)|(concert)|(sport)|(cabaret)
|  Start Time < 15.5
|  |  Day=(Mon)|(Wed)|(Fri)|(Thu)|(Sat): yes(20.0/0.0)
|  |  Day!=(Mon)|(Wed)|(Fri)|(Thu)|(Sat): no(2.0/0.0)
|  Start Time >= 15.5: yes(109.0/0.0)
additional_tag!=(club)|(concert)|(sport)|(cabaret)
|  Day=(Fri)
|  |  Category=(art): yes(1.0/0.0)
|  |  Category!=(art): no(2.0/0.0)
|  Day!=(Fri): no(15.0/0.0)
```

Looking at the decision tree generated by CART, differences to the other trees can be found easily. This can be explained by the tree being a binary decision tree, performing

Table 5.3: TestResults of Step 2

Test Results			
	C4.5	CHAID	CART
Events provided	238	238	238
Events recommended	20	67	89
Time needed	230 ms	340 ms	305 ms
Gradations of confidence level	6	4	1

only binary splits. Because attributes like `additional tag` have more than two options, the algorithm has to work with negations. The algorithm represents all preferences of the user in categories, but also recommends art events, only added to the training set by a accidentally made click of the user.

CHAID

```
- Root [splitBy: Week]
   - Workday [splitBy: Day]
      Mon: yes
      Tue: yes
      Wed: yes
      Thu: no
   - Weekend [splitBy: Day]
      Fri: yes
      Sat: no
      Sun: yes
```

Looking at the last tree generated by CHAID, it can be seen that the algorithm has not recognized the preferences of the user in categories and has split on the `part of the week` attribute. The algorithm also for inexplicable reasons splits on the `day` attribute after the first split. The same result would be achieved with a split only on the `day` attribute, making the first split superfluous.

5.3.4 Comparison of Results

As in the first step of the study, all three algorithms also get compared on their performance, gradations of confidence levels and the number of recommended events. The main difference to the first step in this case is the number of events. In the running system Ennui, the backend filters the events for the location in a radius of 25 km to the user (the radius range can be changed by the user). With only providing events nearby the user, the amount of events provided shrinks remarkable. In the case of the study, there were 238 events in a radius of 25 km to the user.

The following points were relevant for the evaluation seen in table 5.3:

- The number of events provided: The number of events provided are the same for each algorithm, to have the same conditions for testing.
- The events recommended by each algorithm: To get the best recommendations out of the algorithms, only the highest confidence level is considered as recommended. With this procedure, algorithms that have more confidence level gradations also most likely have the least events in the highest confidence level, because the events get divided more often. In the case of Ennui, a tinier but quality-wise better pool of recommendations is the goal, because only 10 get shown on the application.
- The time needed for the request: This is really important for performance reasons. In difference to the Test Environment, the request consists of:
  - Applying the filter applied by the user onto the database.
  - Fetching the filtered events from the events
  - Applying the algorithm to the events fetched
  - Returning all events fetched and the best 10 recommendations

  Due to the fact that all three request process the same amount of data, the time that the algorithms need for classifying the events can still be compared.
- The gradations of the confidence levels: This is relevant for the amount of recommended events. With more gradation levels, less events will be in the highest level, representing the best fitting events, that will be shown in Ennui.

As clearly seen in table 5.3, C4.5 leads in all points provided by the table, recommending the least events with the most gradation of the confidence level, which is good in the case of Ennui as explained above. Performance-wise the algorithm was also the fastest of all three. Looking at the other two algorithms, it can be seen, that the CHAID algorithm also provides good results, but with not recognizing the user preferences and additionally making an superfluous split, the decision tree generated by CHAID is not suitable for the use on the running system. Performance-wise CHAID is the worst of the three algorithms. This can be explained due to the fact that the algorithm was self-developed and not executed through a library as the other two algorithms. The last algorithm CART, has a major problem if we look at table 5.3 with only having one gradation at the confidence level, meaning that the confidence level can only be 0 percent or 100 percent. This leads to a really big pool of recommended events, that does not have the quality of events provided by other algorithms.

5.4 Overall Conclusion

With being the best performing algorithm overall in both steps of the study, C4.5 was chosen as the best fitting algorithm for Ennui. Having the biggest tree size of all three, C4.5 was best in representing the user preferences provided by the training set, without

producing an overfit. The problem with CART was that the algorithm produced a binary decision tree. The events of Ennui have many attributes that have more than two outcomes, so the classifier had to work with negations and struggled most of the time to represent the simulated habits of the test user. Another reason for the bad overall result of CART could be the pruning, which made the tree smaller in size, but also more inaccurate, without having an extreme performance boost in comparison to the other algorithms. While the CHAID algorithm really worked well with the hard coded training set provided in the first step of the study, he struggled with the training data generated from user clicks, producing a classifier with superfluous splits and not representing the habits of the user on the running system. In contrast to C4.5 and CART, that got developed with the help of the Weka Java library, CHAID was completely self developed by Simon Gutenbrunner. As a help, the training set used for testing the algorithms in step 1 was used for the development process, leading to really good results with using it. With a training set of a much bigger size and consisting of events the user has clicked, the algorithm had problems finding the preferences represented in them.

Chapter 6

Android Implementation

6.1 Overview

This Chapter is about the Android implementation of our project. The Application serves as a Front-End which means it displays the data from our Back-End server, demonstrates the algorithms (see chapter 3) and provides ways for the user to interact with the data . This Chapter starts with a glossary/explanation of words and terms used for developing android applications which will also be used in the following chapter. It then continues with listing and explaining all the features the application has. For some features the most significant source code will be highlighted and explained. The chapter closes with describing the Frameworks and how they were used in the project.

6.2 Android Glossary

6.2.1 Activity

An android application is built out of one or multiple *activities*. Like the name suggests an activity represents a certain thing a user can do, such as logging into an application, changing some settings or viewing a users profile. They are depicted to the user as full windowed pages. In the android implementation of this project, each feature is implemented via an activity. For a list of all features see section 6.3. (source: [Goo18a])

6.2.2 Fragment

Fragments are modular parts of an android applications which are embedded in activities. They are reusable meaning they can be used in multiple activities a multiple amount of times. They can also be created/destroyed during the apps runtime. A popular example of fragment usage would be when providing multiple layouts of the user interface for different kinds of devices like smartphones and tablets. In this case, each layout is implemented as a fragment and the application then decides which one to display depending on the users device. (source: [Goo18c])

6.2.3 AsyncTask

An *AsyncTask* is a tool in android programming that allows the application to do multiple tasks at the same time. For example when the application tries to load some online resources the rest of the program would have to wait until the request is finished, meaning for the user it would appear as it stopped working since it would not react to his inputs. When using AsyncTasks for requesting online resources however, the application is still capable of doing something else during the request, like reacting on user inputs. (source: [Goo18b])

6.3 Code Feature Explenation

This section explains all parts of the UI(Buttons, etc), and how the activities are connected with each other.

6.3.1 Event Fragment

The *EventFragmen*t is the UI's way of displaying most of an event(see section 1.3.1) data and providing an option for the user to interact with it. It consists of a label for the event-organizer, the events cover image, the name of the specific event, its start and end time together with its location.Clicking on the cover image leads to the respective detail page(see section 6.3.5). There is also a button to favor/unfavor an event. Favorizing an Event is only possible if the user is logged in already. When done so the application will send a REST-Request (POST) to the backend server, to tell the server to mark this event as favored. If the request was successful the event will be marked as favored, meaning upon completion the outlined heart will turn completely black and will appear in the list of favored events accessible in the user overview (see section 6.3.19). For the graphical difference between an unfavored and favored event see figure 6.1. The program implements this feature via an Event Listener and the REST-Requests are sent via AsyncTasks. To unfavor an event a logged in user has to click on the black heart icon which indicates that the event is favored. Upon clicking it the application will, similar to the favoring process, send a REST-Request (POST) to the Back-End server telling it to mark the event as unfavored again. On success the once black heart icon is again only displayed as an outline and the event will no longer appear in the list of favored events in the user overview (see seciont 6.3.19). The implimentation of the source code for favoring/unfavoring an Event looks like the following:

```
1  //on favorize-button click
2  favBtn.setOnClickListener(new View.OnClickListener() {
3
4     @Override
5      public void onClick(View v) {
6
7        //check if user is logged in
8        if( AccessToken.getCurrentAccessToken() != null) {
```

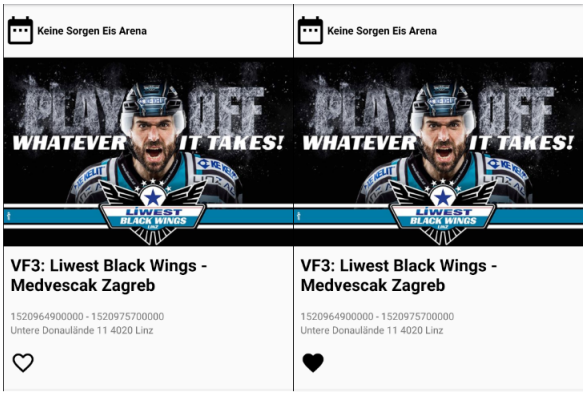


Figure 6.1: Event Fragment unfavored/favored

```
9
10          //unfavor if event is favored
11          if(isFavored) {
12             RemoveFavTask removeFavParser = new RemoveFavTask();
13                removeFavParser.execute(
14                   "https://ennui.htl-leonding.ac.at:8443
15                      /events/unfavorize/"+event.getId()
16                );
17          }
18
19          //favor if event is not favored
20          else {
21             AddFavTask addFavParser = new AddFavTask();
22                addFavParser.execute(
23                   "https://ennui.htl-leonding.ac.at:8443
24                      /events/favorize/"+event.getId()
25                );
26          }
27       }
28    }
29 });
```

6.3.2 Offer Fragment

The *Offer Fragment* is how the UI displays the most important data of a specific offer (see section 1.3.2) in a compact way. It consists of the Offers Name, its address and if the Offer is currently open or closed. By clicking on in the user gets directed to the respective detail page (see section 6.3.11).

6.3.3 Game Fragment

The *Game Fragment* is the UI's way of compactly displaying data of a respective game (see section 1.3.3) and providing a way of some basic interactions. It consists of an image to give a basic impression of the game. Above there is a label which shows the game's category. There is also one for the game's name, one for the recommended/required player count and a button to favor or alternatively unfavor a game if it is already favored. By clicking on the image the user is directed to the game's respective detail page (see section 1.3.3). When a logged in user clicks on the outlined heart icon which represents that the spicific Game is unfavored, the application sends a REST-Request (POST) to the server to tag it as favored. If the request was successful the favored Games favorize-button will turn from a back outlined heart into a complete black one. Additionally the favored Game will from now on appear in the list of favored Games accessible via the user overview activity (see section 6.3.19). For the graphical difference between an unfavored game and a favored one see figure 6.2. The program implements this feature via an Event Listener and the REST-Requests are sent via `AsyncTasks`. To unfavor a Game a logged in user has to click on an already favored favorize-button, which is displayed as a black heart. Upon clicking it the application will send a REST-Request (POST) to the server to tag the game as unfavored again. If this process is successful the the favorize-button of the unfavored game will turn into an outlined heart again and the game will no longer appear on the list of favored games accessible via the user overview activity (see section 6.3.19). The source code looks for favoring/unfavoring a game looks as the following:

```
1  //on favorize-button click
2  favBtn.setOnClickListener(new View.OnClickListener() {
3
4     @Override
5     public void onClick(View v) {
6
7        //check if user is logged in
8           if( AccessToken.getCurrentAccessToken() != null) {
9
10          //unfavor if game is already favored
11             if(isFavored) {
12                GameFragment.RemoveFavTask removeFavParser = new
                      GameFragment.RemoveFavTask();
13                removeFavParser.execute(
14                  "https://ennui.htl-leonding.ac.at:8443/games/unfavorize/"+
                        game.getId()
15                );
16             }
17
18             //favor if game is unfavored
19             else {
20                GameFragment.AddFavTask addFavParser = new
                      GameFragment.AddFavTask();
```

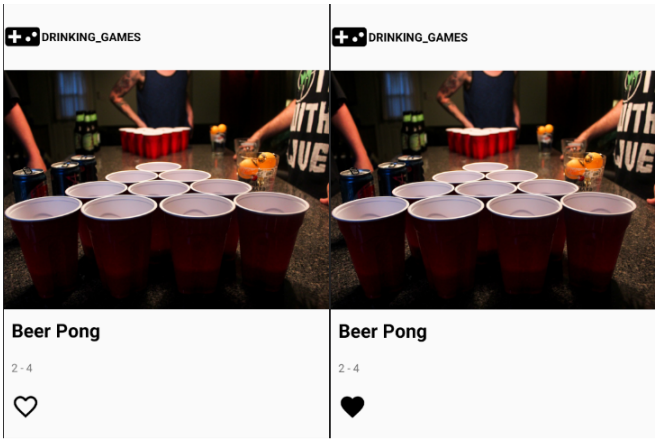


Figure 6.2: Game Fragment unfavored/favored

```
21                addFavParser.execute(
22                  "https://ennui.htl-leonding.ac.at:8443/games/favorize/"+
                        game.getId()
23                );
24            }
25        }
26    }
27 });
```

6.3.4 Event List

Upon application start or by clicking the respective link in the navigation drawer(6.3.21), the application shows the *Event List* activity. It consists of two sets of event fragments (6.3.1). The first set lists recommended events evaluated by a big data sorting algorithm, the second one lists all available events found in a user determined radius.

6.3.5 Event Details

The *Event Details* activity contains all values displayed in the event fragment(see section 6.3.1), a detailed description about the event and a Button which leads to the Event Location of the respective event. Via right-swipe the user can navigate back to the event list (see section 6.3.4).

6.3.6 Event Location

The Activity consists of a full page map created via Google Maps API(see section 6.4.2) with a marker pinned to the events location.

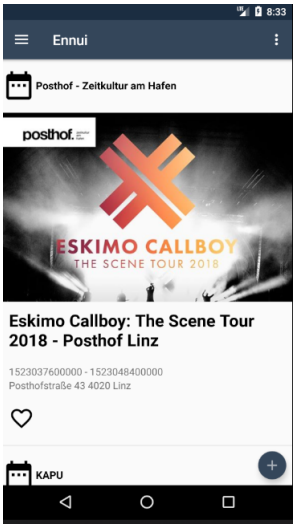


Figure 6.3: Event List

Figure 6.4: Event Details

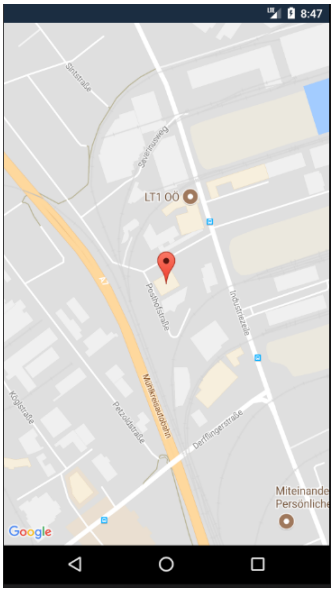


Figure 6.5: Event Location

6.3.7 Add Event

This activity allows the user to submit a new events. The form contains text fields for the events name, its image url, an optional link for a website, and description. There is also an input field for the events location which auto completes a location entered with data from Google Maps. There are also two date pickers for start and end date and two time pickers for start and end time. Furthermore there are radio buttons to assign a category to the event. All of the inputs except the website are mandatory and the event has to start before it ends once the user has clicked the submit button on the bottom of the form. The event then has to be approved by an admin which can not be done on the mobile client. By swiping right the user cancels the submitting process and gets directed back to the event list(see section 6.3.4).

Upon submitting the data in the Add Event Form the client checks if all required input fields are filled and if so, he sends a REST-Request (POST) to the Back-End which leads to queuing it to a list of Events waiting for being approved or denied by an Ennui-administrator. The REST-Request is done by an `AsyncTask` (see section 6.2.3). The code snippet below shows input validation.

```
1 if(allFieldsFilled) {
2    Date start = new Date();
3     Date end = new Date();
4     try {
5       SimpleDateFormat format = new SimpleDateFormat("d/M/yyyy h:m");
6         format.setTimeZone(TimeZone.getTimeZone("CEST"));
```

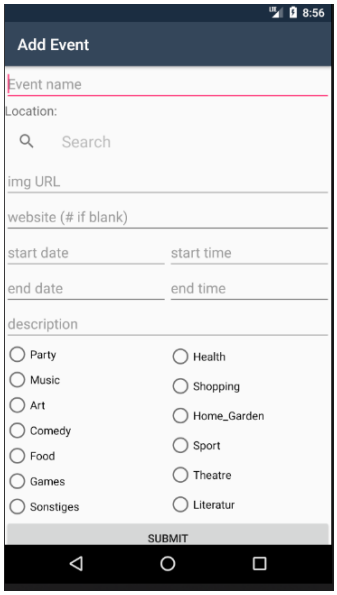


Figure 6.6: Add Event Form

```
7           start = format.parse(startDate.getText() + " " + startTime.getText());
8         end = format.parse(endDate.getText() + " " + endTime.getText());
9       }
10    catch (Exception e) {
11      Toast.makeText(AddEvent.this, "date parse error",
            Toast.LENGTH_SHORT).show();
12    }
13
14      if(start.compareTo(end) == 1) {
15      Toast.makeText(
16      AddEvent.this,
17      "End date must be greater than start date!",
18      Toast.LENGTH_SHORT)
19      .show();
20    }
21    else {
22      AddEvent.AddEventTask task = new AddEvent.AddEventTask();
23        task.execute("https://ennui.htl-leonding.ac.at:8443/events/add");
24    }
25  }
```

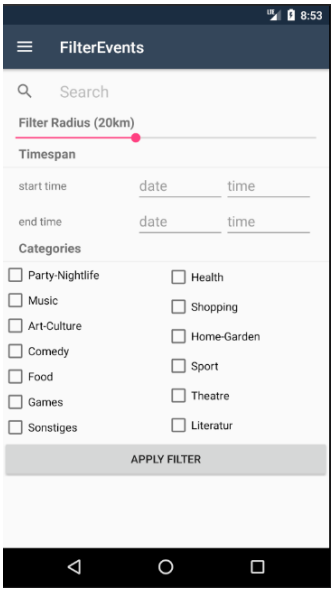


Figure 6.7: Filter Events Form

6.3.8 Filter Events

In this activity the user can filter the events displayed on the event list(see section 6.3.4) and narrow down the shown elements by different specifiers. The user can filter the Events by name, radius, start- and end-time, and categories. Once clicked on the Apply Filter button the user returns to the Event List activity with the new set of Events. The user can cancel the filtering process and go back to the Event List by swiping right. Upon clicking on the apply button the application redirects the user to the Event List (see section 6.3.4) but also parses filter parameters adjusted by the user. The application then only displays Events who match the criteria given by those parameters. The application implements this with building the request path for fetching the events from the filter parameters.

```
1  FilterEventOptions filterOptions = (FilterEventOptions)
       getIntent().getSerializableExtra("filterOptions");
2
3  String requestPath = "https://ennui.htl-leonding.ac.at:8443/events?";
4  requestPath += "country=" + filterOptions.getCountryCode() + "&";
5  requestPath += "latitude=" + filterOptions.getLatitude() + "&";
6  requestPath += "longitude=" + filterOptions.getLongitude() + "&";
7  requestPath += "radius=" + filterOptions.getRadius() + "&";
8  requestPath += "startTime=" + filterOptions.getStartTime() + "&";
9  requestPath += "endTime=" + filterOptions.getEndTime() + "&";
10
11 if(filterOptions.getCategories() !="") {
```

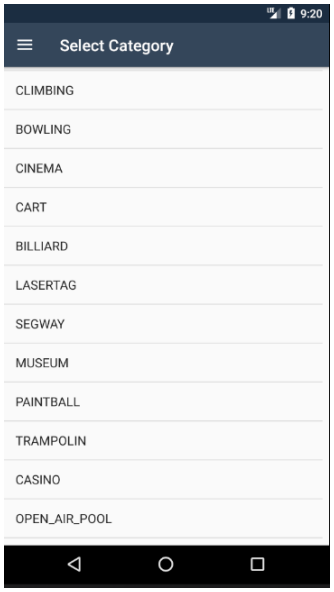


Figure 6.8: Offer Categories

```
12      requestPath += "categories=" + filterOptions.getCategories() + "&";
13  }
14
15  EventFetchTask task = new EventFetchTask();
16  task.execute(requestPath);
```

6.3.9 Offer Categories

By clicking on the offers link in the navigation drawer (see section 6.3.21) the Application shows this activity, which lists all of the offers categories. Upon clicking one it redirects the user to the offer list(see section 6.3.10) activity that consists all offers of the selected category.

6.3.10 Offer List

The *Offer List* activity consists of a set of offer fragments (see section 6.3.2) all with the same category which was selected in the offer categories (see section 6.3.9) activity. This activity lists all of this Fragments underneath each other. Via right-swipe the user can navigate back to the Offer Categories. Unlike on the Event List, there is no button on the bottom right that allows to submit new Elements, since submitting new offers is not an implemented feature.

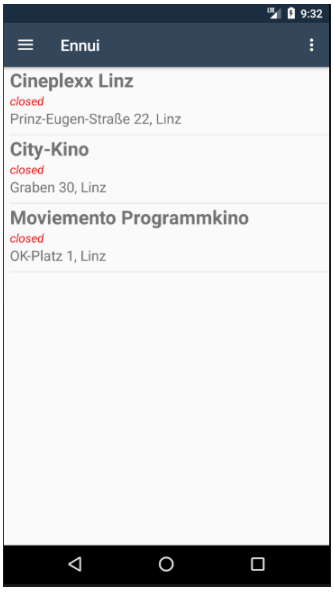


Figure 6.9: Offer List

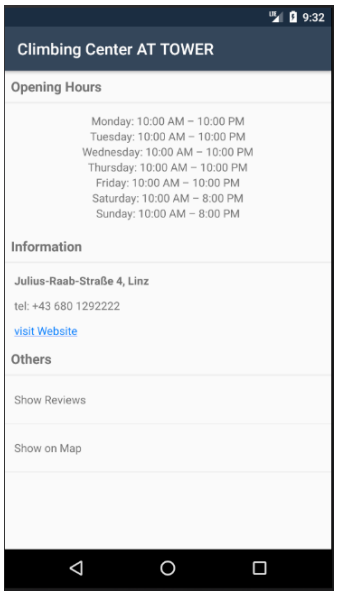


Figure 6.10: Offer Details

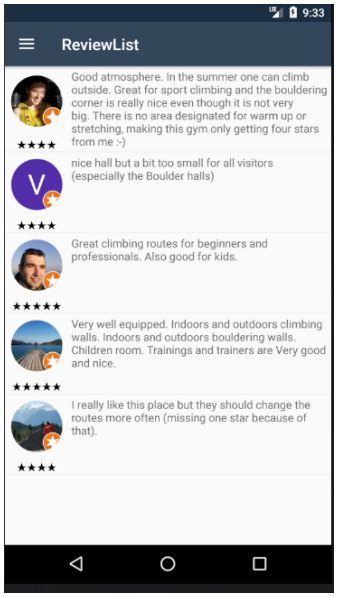


Figure 6.11: Offer Comments

6.3.11 Offer Details

The *Offer Details* activity shows all of an offers data, including those not displayed in the offer fragment (see section 6.3.2) which are an address of the offer, all opening hours, a button which opens the offer comments (see section 6.3.12) and one which directs the user to the offer location (see section 6.3.14) activity. By swiping right the user gets directed back to the offer list (see section 6.3.10).

6.3.12 Offer Comments

The *Offer Comments* activity shows reviews for the respective offer, rated by Google users. It consists of a set of comment fragments (see section 6.3.13) which are displayed underneath each other. By swiping right the user gets directed back to the Offer Details page.

6.3.13 Comment Fragment

A *Comment Fragment* holds and displays all the data of a review. It consists of a profile image of the Google user who made the review, a rating displayed by 1-5 stars and an optional additional text.

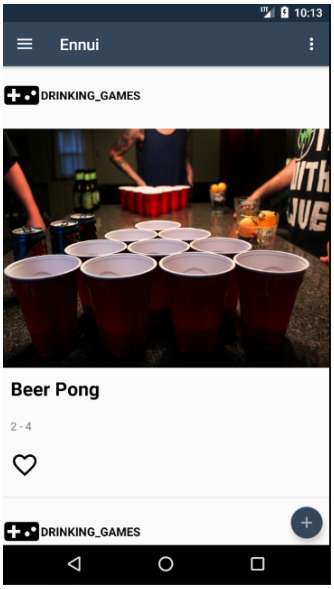


Figure 6.12: Game List

6.3.14 Offer Location

The activity consists of a full page map created via Google Maps API (see section 6.4.2) with a marker pinned to the offers location.

6.3.15 Game List

The *Game List* activity consists of a set of game fragments (see section 6.3.3) which are all displayed underneath each other. On the bottom right there is a button which allows the user to submit a new game via the add game form (see section 6.3.17). On the top right there are also menu options which redirects the user to the filter games form (see section 6.3.18).

6.3.16 Game Details

This activity shows all of a games data displayed in the game fragment (see section 6.3.3) and additional info like the games description and instructions on how to play. By swiping right the user gets redirected back to the game list.

6.3.17 Add Game

On this page the user can submit new games (1.3.3). The activity consist of 6 input fields for the games name, a detailed description, instructions on how to play, an url for the image and the minimum and maximum recommended/required player count. There

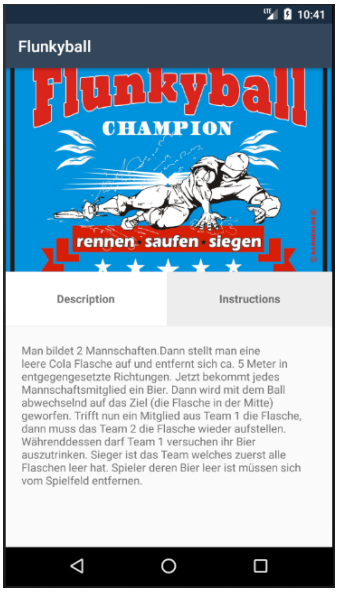


Figure 6.13: Game Details

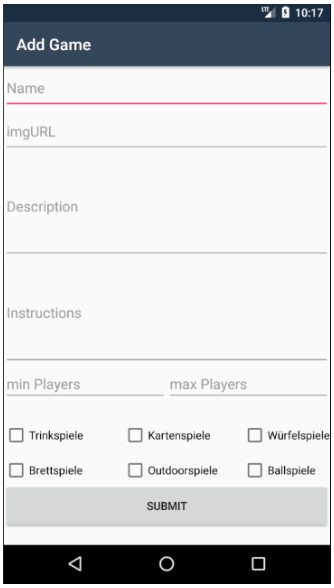


Figure 6.14: Add Game Form

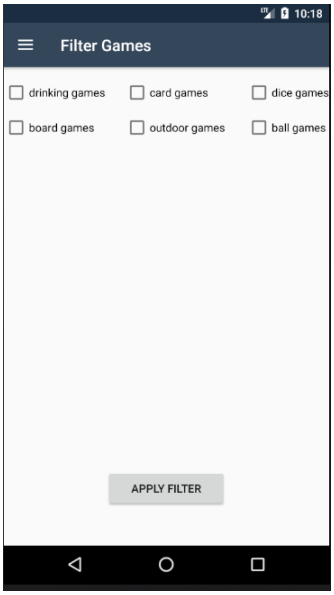


Figure 6.15: Filter Games Form

are also multiple checkboxes, one for each category. Once clicked on the submit button all fields have to be filled and at least one category has to be selected. Also the minimal player count can not be higher than the maximal player count. The submission then has to be approved or declined by an administrator which can not be done on the mobile client. The user can cancel the process of submitting by swiping right and going back to the game list (see section 6.3.15). When submitting the typed in data the client checks if all required input fields are filled in correctly and if so, sends a REST-Request (POST) to the server, queuing it into a list of Games waiting to be approved or declined by an Ennui-administrator.

6.3.18 Filter Games

Via this Activity the user can narrow down the amount of games (see section 1.3.3) presented in the game list (see section 6.3.15). It presents a dynamic amount of checkboxes, one for each category. After clicking the Apply Filter button the user gets redirected back to the Game List with only Games of the selected categories shown in it. When swiping right the user cancels the filtering process and gets directed back to the Game List.

6.3.19 User Overview

This page illustrates what Facebook user is currently logged into the application. The activity holds an image of the users profile picture, the users name and buttons redirecting the user to a list of his favorite Events and one redirecting to his favorite games.

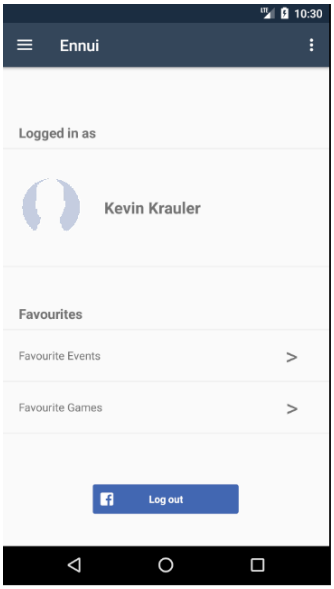


Figure 6.16: User Overview

6.3.20 Login

This page is displayed when the user tries to access the user overview (see section 6.3.19). It lists the benefits gained when connecting a Facebook account to the Application, and also has a button to connect with Facebook on the bottom of the page. When logged in additional features become available for the user to use, including adding, filtering and favorizing or unfavorizeing events or games. This feature is used in the login page activity (see section 6.3.20). In the source code this process is part of the Facebook SDK (see section 6.4.1 and is implemented via a callback manager as seen in the snippet below.

```
this.cbManager = (CallbackManager) CallbackManager.Factory.create();
loginBtn.registerCallback(cbManager, new FacebookCallback<LoginResult>() {
   @Override
      public void onSuccess(LoginResult loginResult) {
      Intent detailsIntent = new Intent(UserActivity.this, UserActivity.class);
         startActivity(detailsIntent);
         }

         @Override
        public void onCancel() {
            Toast.makeText(UserActivity.this, "login canceled",
                Toast.LENGTH_LONG).show();
        }

```

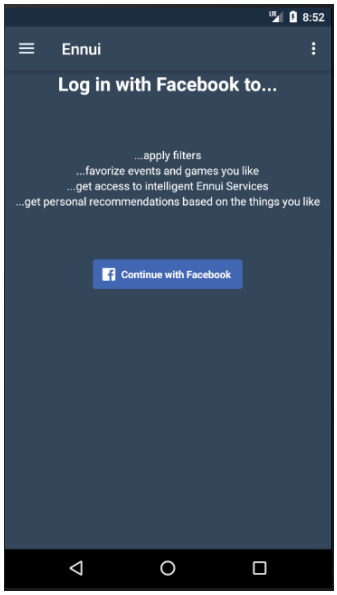


Figure 6.17: User Login

```
14          @Override
15          public void onError(FacebookException error) {
16          Toast.makeText(UserActivity.this, "error", Toast.LENGTH_LONG).show();
17         }
18       });
```

6.3.21 Navigation Drawer

The *navigation drawer* is accessed by clicking on the menu button on the top left corner of the application. it has 4 links including, one redirecting to the event list (see section 6.3.4), one to the offer list (see section 6.3.10), one to the game list (see section 6.3.15) and one to the user overview (see section 6.3.19).

6.4 Frameworks

6.4.1 Facebook SDK

The *Facebook SDK* is used to enable the users to log in without caring about security flaws which come with a self developed solution. Additional advantages of the Facebook SDK are the simplicity of including a user management system, predetermining a users preferences. Unfortunately there might be users who does not use Facebook and therefor can not log into the application since we do not provide generating independent user accoutns.

6.4.2 Google Maps API

The *Google Maps API* is used for showing an events/offers Location on the Google Maps Application. It is also used for auto completing the location input in the add event activity(see section 6.3.7).

Chapter 7

iOS Implementation

7.1 Introduction

The iOS application is used as Front-End or in other words to visualize our project and see how the algorithms are working "live" in it. This chapter is about how the iOS application is implemented and how it can be used. This chapter starts with a glossary explaining the most important words concerning iOS-Development. This is followed by a section describing the general project structure. After this section the features and the code of them will be explained. The chapter closes with a section on problems and their respective solutions and the used frameworks.

7.2 Supported Platforms

This software was developed using XCode 8 and Swift 3. Later on a upon release of Swift development was switched to Swift 4. Available for Devices with iOS 10.0 and later.

7.3 iOS Glossary

7.3.1 Group

A Group is a collection of classes dedicated to a specific topic. The concept of Swift-Groups is pretty similar to JAVA "packages" and CSharp namespaces. On a file system level every group has its own directory containing files.

7.3.2 Storyboard and Views

The iOS Storyboard (See Figure 7.1) is basically a collection of ViewControllers and their connection between each other represented visually in XCode. Those ViewControllers like the UITableViewController (7.3.6) are consisting out of views to create an interface.

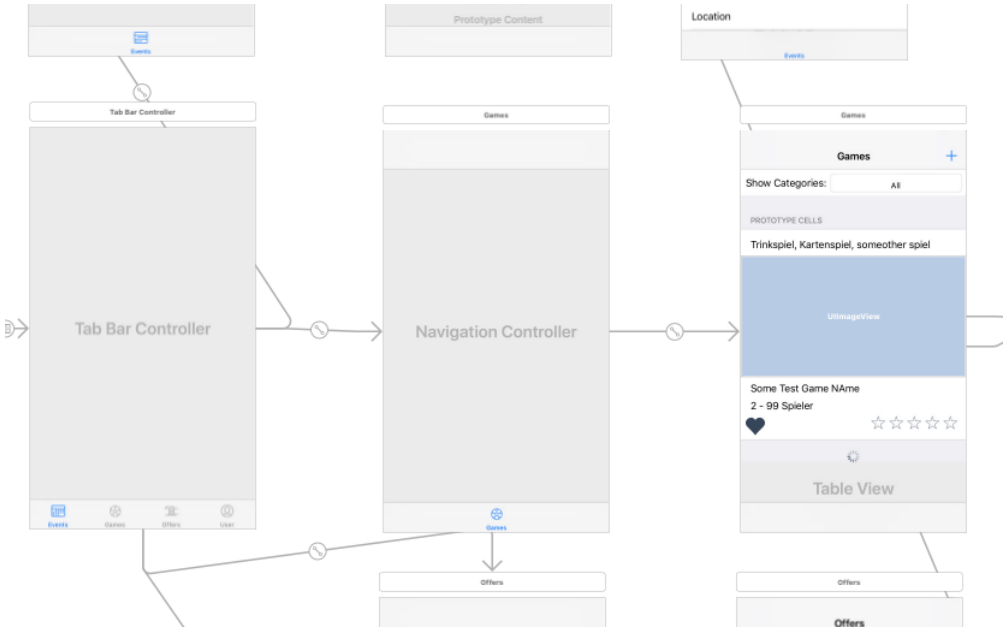


Figure 7.1: Storyboard-Views

7.3.3 nil

Referring to another language for example JAVA or C-Sharp there is `nil` called `null`. A value 'A' can either have data inside, empty data, or no data which is then called nil. For better visualising lets think about a toilet paper roll hanging on a toiler paper holder. If the toilet paper on this holder is full it is not zero if it is empty it is zero and if there no toilet paper on the toilet paper holder it is nil. To check if a value is nil before assigning it to another value or something swift is able to assign values as optional see chapter 7.3.5.

7.3.4 Dictionary

A dictionary in swift is equal to the dictionary in C-Sharp or to the Map in JAVA. The dictionary is made up of key value pairs where each value has their own key and each key their value. To read data out of it, the dictionary has to be called by the key.

7.3.5 Optionals

When parsing data between via HTTP-Requests it is possible that some properties of a specific object are empty or null. Without checking for these null-values every application would get an exception and crashes. To skip the checkback if value 'A' is nil (See nil 7.3.3) Swift invented optionals. To mark a value as optional, a question mark '?' can

be written before it to check if it is either nil and missing or not. If the value is missing the code is skipped. For example parsing out of a dictionary:

```
1    dto.value = dictionary.value(forKey: "icon") as? String;
```

Next to the question mark there's the exclamation mark '!'. If this operator is written before a value or instead of the question mark in the example above it means that the value which gets assigned to another is not nil but a value is inside and the application neither checks if the value is nil or assignable.

7.3.6 UITableViewController

The view element called `UITableViewController` is a `ViewController` with its task to display a table. This table is built out of sections and cells which are representing the data (see 7.3.7). The before mentioned section is basically a headline followed by cells.

7.3.7 UITableViewCell

The `UITableViewCell` is the child of `UITableViewController` and it is used to display data. It can contain various other views or elements and it belongs to a specific section in the table.

7.3.8 Segue

In Swift a Segue is a connection between two ViewControllers. Next to referring to another view which should be shown or the previous when tapping back, it can contain data.

7.3.9 User Defaults

The User-Defaults existing on each iOS-Device is a key value based database. This database is used to load user settings for the application. As it is key value based a *JSON* can be saved into it. There a many ways to do so, but the author used the following one. The example is about how to save the *UserDto* into the defaults.

```
1  let loginUserDto = backendService.parseLoginJSON(json: nsdict);
2  let userDefaults = UserDefaults.standard;
3  let encoder = JSONEncoder()
4  if let encoded = try? encoder.encode(loginUserDto){
5     userDefaults.set(encoded, forKey: "userDto");
6  }
7  userDefaults.synchronize();
```

After parsing the object to a *JSON*, the *UserDefaults* have to be initiated. To save the data the method `set()` and finally a `synchronize()` has to be called onto the *UserDefaults*. After synchronizing the data with the *UserDefaults* it can be read out with the key where the object is saved to.

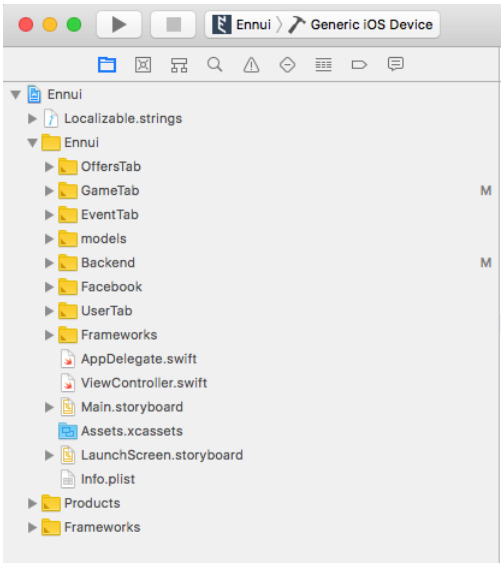


Figure 7.2: XCode8 Project-Structure

```
let decoder = JSONDecoder()
if let dtoData = userDefaults.data(forKey: "userDto"), let usr = try?
    decoder.decode(UserLoginDto.self, from: dtoData){
   self.userDto = usr;
}
```

Now a decoder is used to decode and map the existing entry in the database to an object. Read more about User Default in the official Apple documentation [App17a].

7.4 Project Structure

In figure 7.2 the project structure is visible. It is separated in eight Groups (see 7.3.1) where the following four are the most UI related groups.

- EventTab
- OffersTab
- GameTab
- UserTab

7.5 Code/Feature Explanation

7.5.1 Login

While the login view (see Figure 7.3) is visible a user can choose between two options. The first option is to login with Facebook (using Facebook-SDK see 7.7.1) and the second one is to proceed without logging in. When a user is logged in, more features are available than proceeding without it. If a user is already logged in, this view will be skipped when re-opening the app. This behaviour is described in the following listings. The first method entered by the app is `viewDidLoad()` in the `ViewController` class.

```
override func viewDidLoad() {
   super.viewDidLoad()

   let fbService = FacebookService();
   if(FBSDKAccessToken.current() != nil){
      doBackendLogin()
   }
   else{
      let cg = CGRect(x: view.frame.width/2 - 150, y: view.frame.height/2 + 50,
          width: 300, height: 50)
      fbService.createFacebookButton(view: self,delegateClass: self, cg: cg);
   }
}
```

The Facebook-SDK saves whether a user is logged in, and based on that, it can be asked if a user is already logged in or not via the `{FBSDKAccessToken`. If this token is nil (7.3.3) a login in our Backend is needed which is handled by following method:

```
func doBackendLogin(){
   let backendService = BackendService()
   backendService.doPost(path: "/users/login", dict: [:],token:
       FBSDKAccessToken.current().tokenString, callback: { (nsdict) in
      let loginUserDto = backendService.parseLoginJSON(json: nsdict);
      let userDefaults = UserDefaults.standard;
      let encoder = JSONEncoder()
      if let encoded = try? encoder.encode(loginUserDto){
         userDefaults.set(encoded, forKey: "userDto");
      }
      userDefaults.synchronize();
      //Observer Pattern
      NotificationCenter.default.post(name: NSNotification.Name(rawValue:
          "userDataRecieved"), object: nil);
   })
   segueLoginToMain();
}
```

An HTTP-POST Request is sent to the Ennui backend and it returns a JSON encoded

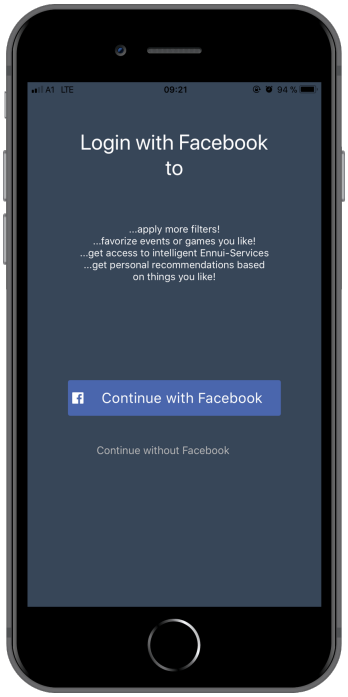


Figure 7.3: Login View

user profile determined through the access token. If the user profile is valid a segue (7.3.8) to the Main App is initiated.

7.5.2 Event List

After the login window the first view which can be seen is the event list in the tab "Events". Basically it is a list where nearby Events are displayed in cells. As it can be seen in Figure 7.4 there are two sections in the `UITableViewController` (see 7.3.6). In the first section "Recommended Events" the recommendations for the user are displayed and in the second one all events which are nearby the user's current location are displayed. The two sections are a static array of Strings.

```
1 let sections = ["Recommended Events","Events"];
```

To get and list all events an array of two arrays containing EventDTO's was used. The two arrays in one array is to decide whether the event is in the 'recommended events' section or in the 'events' section.

```
1 var eventItems:[[EventDto]] = [[],[]];
```

When the user enters the app the location gets tracked and with this information where

the user currently is, the application is able to send a *GET-Request* to the Ennui backend to get a list of events and if the user is logged in also a list of recommended events. This happens in a method called *getEventsFromServer()* where a part of the code looks like the following snippet:

```
backendService.doGet(path: path +
    String(eventFilter.getFilterAsRequestParameters()),token: token, callback:
    {
  (nsdict) in
   let eventHolder: Holder<EventDto> = backendService.parseHolder(json: nsdict)
   if eventHolder.success != nil && eventHolder.success {
      if(eventHolder.recommendedResults != nil){
         self.eventItems[0] = eventHolder.recommendedResults
      }
      if(eventHolder.result != nil){
         self.eventItems[1] = eventHolder.result
      }
   }
   DispatchQueue.main.async {
      self.tableView.reloadData();
      self.eventLoadingBar.stopAnimating();
   }
})
```

With the method of a written *BackendService* the author is able to perform a *GET-Request* with the parameters for a possible filtering (see section 7.5.5) and the API-Endpoint path. If the user is logged in the token is also copied into the request (As mentioned before this is to get the recommended events for the user).

After parsing the delivered *JSON* from the backend the application has to check if the call was successful or not. If the result has been successfully delivered the application can save the events and recommended ones into the before mentioned array of *eventItems*. Through the `DispatchQueue` the application reloads the UI and displays the whole event table.

All table cells are reusable so it can happen that more cells look like the same and display the same data. To prevent this it is important to override the following function.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
    IndexPath) -> UITableViewCell {
   let cell = tableView.dequeueReusableCell(withIdentifier: "eventCell", for:
       indexPath) as! EventCell;
   let dto = eventItems[indexPath.section][indexPath.row];
   cell.eventDto = dto;
   cell.eventNameLabel.text = dto.name;
   cell.locationLabel.text = dto.placeName;
   cell.timespanLabel.text = dto.getTimeSpanLabel();
   .
   .
```

```
10      .
11  }
```

By setting the data to the cell it is important to reset and set all the data otherwise problems can occur for instance the before mentioned one.

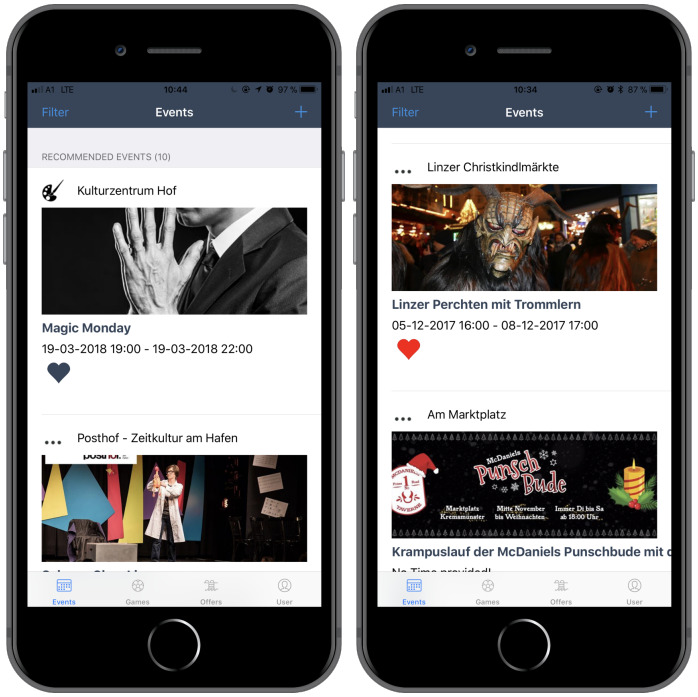


Figure 7.4: Event List

7.5.3 Favorize Events

When clicking on the blue heart in the event list, the event gets favorized and the heart turns red. (see Figure 7.4) Automatically a Push-Notification gets registered two hours before the favorized event will start. How it works: When clicking on the heart a method called `favorizeButton()` is called where the event either gets favoured or unfavoured.

```
1  // When the event is already favoured:
2  backendService.doPost(path: "/events/unfavorize/" + String(sender.tag), dict:
       [:],token: FBSDKAccessToken.current().tokenString, callback: {
3     (nsdict) in
4     // print(nsdict.value(forKey: "success") as! String);
5  })
6  removeNotification(eventName: sender.eventDto?.name)
7
8  // When the event has to be favoured:
```

```
backendService.doPost(path: "/events/favorize/" + String(sender.tag), dict:
    [:],token: FBSDKAccessToken.current().tokenString, callback: {
   (nsdict) in
   //print(nsdict.value(forKey: "success") as! String);
})
alert(message: NSLocalizedString("Event has been added to 'Favored Events'",
    comment: ""), title: NSLocalizedString("Favored!", comment: ""));
addNotificationForEvent(eventName: sender.eventDto?.name ?? "Event",
    startTime: sender.eventDto?.starttime)
```

As written in the code above the Notification when favouring an event gets registered or removed. Again everything is handled via HTTP-POST Request's to our Backend Server.

7.5.4 Event Detail View

When clicking on the event itself the detail view opens (see Figure 7.5) As headline the name of the event will be displayed. In front of the cover the date, a link to the website and the owner can be seen. Important for every event is a description for it which is placed under the header. When a user now wants to visit an event he can check out the location in the location tab.

This *ViewController* is reusable and is created just once. To prevent displaying wrong data the function `viewWillAppear` inherited from `UITableViewController` is used.

```
override func viewWillAppear(_ animated: Bool) {
   self.navigationItem.title = eventDto?.name;
   if(cover == nil && eventDto?.coverUrl != nil){
      downloadImageAndSetCover(url: (eventDto?.coverUrl)!);
   }
   else{
      self.coverImage.image = self.cover;
   }
   self.eventNameLabel.text = self.eventDto?.name;
   if let owner = self.eventDto?.ownerName{
      self.byOwnerLabel.text = "by " + owner;
   }
   else{
      self.byOwnerLabel.text = ""
   }
   .
   .
   .
}
```

The whole class `EventDetailViewController` is based on a `UITableViewController` as before mentioned. But this Table is a static one and can be built in the storyboard UI in X-Code. It has no dynamical cells like the Event-List explained in section 7.5.2.

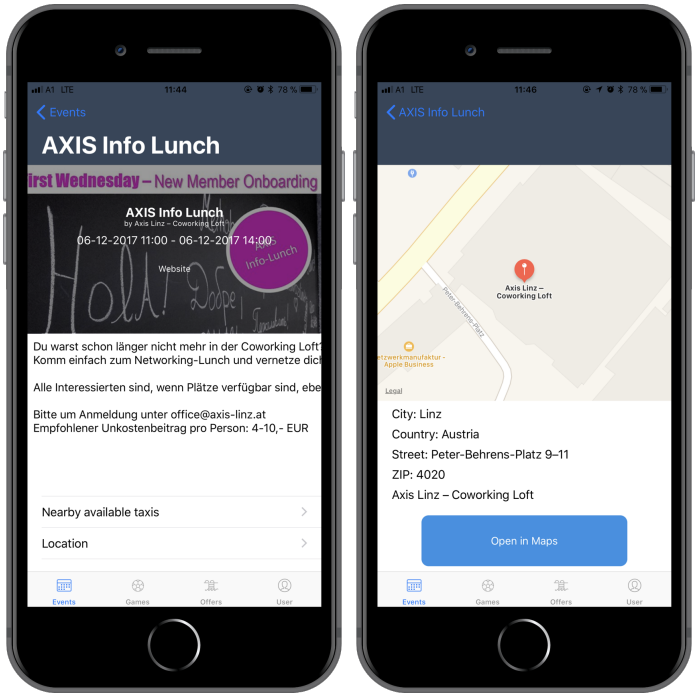


Figure 7.5: Event Detail View

7.5.5 Event Filter

When clicking on the "Filter" (see Figure 7.4) the filter for events will be opened. (see Figure 7.6) In the first section called "Localisation" the user is able to enter a different place, while typing an auto complete box appears with suggestions (see Google-Maps-SDK 7.7.2). Next to the place it can be chosen in which radius "Ennui" should search for events. When a user wants to find events on a specific date he can also select it. Not every event is interesting for everyone, so one or more categories can be selected too. After clicking back or swiping from left to right, the filter will be applied and the new result will be shown.

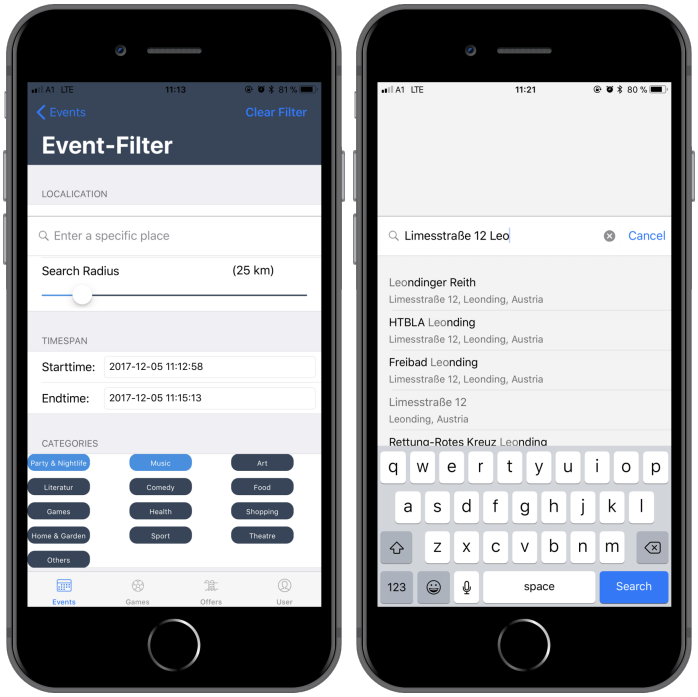


Figure 7.6: Filter View

7.5.6 Add Event

When clicking on the add icon ("+") (see Figure 7.4) a personal event can be added.(see Figure 7.7) To see how the event looks like when it was added, the "Preview"-Button is available.

In the method `addEventButtonAction` the event will be sent to the Ennui backend after checking the required fields.

```
1 @IBAction func addEventButtonAction(_ sender: UIButton) {
2     let backend = BackendService()
3     var addEvent = true
4
5     addEvent = checkRequiredField(msg: NSLocalizedString("Event-Name is
          required!", comment: ""), field: eventNameField) ? addEvent : false
6     addEvent = checkRequiredField(msg: NSLocalizedString("Event-Hoster is
          required", comment: ""), field: eventHosterField) ? addEvent : false
7     addEvent = checkRequiredView(msg: NSLocalizedString("Event-Description is
          required!", comment: ""), field: eventDescriptionView) ? addEvent : false
8     addEvent = checkRequiredField(msg: NSLocalizedString("You have to select
          categories!", comment: ""), field: eventCategoriesField) ? addEvent :
          false
9     addEvent = checkRequiredField(msg: NSLocalizedString("Event-Starttime is
          required!", comment: ""), field: eventStartField) ? addEvent : false
```

```
10      addEvent = checkRequiredField(msg: NSLocalizedString("Event-Endtime is
            required!", comment: ""), field: eventEndField) ? addEvent : false
11      .
12      .
13      .
14  }
```

To finally POST the event to the backend the application has to provide the data in a JSON-Body out of a SWIFT - Dictionary.

```
1   let dateFormatter = DateFormatter()
2   dateFormatter.dateFormat = "EEE, dd MMM yyyy HH:mm:ss zzz"
3   body["name"] = eventNameField.text
4   body["owner"] = eventHosterField.text
5   body["description"] = eventDescriptionView.text
6   body["category"] = backend.convertCategoryNameToCategory(name:
        self.selectedCategory).rawValue
7   body["coverUrl"] = eventImageField.text
8   body["ticketUri"] = eventWebsiteField.text
9   body["country"] = country
10  body["starttime"] = dateFormatter.string(from: NSCalendar.current.date(from:
        startDateComp)!)
11  body["endtime"] = dateFormatter.string(from: NSCalendar.current.date(from:
        endDateComp)!)
12  body["city"] = city
13  body["zip"] = postal
14  body["longitude"] = long
15  body["latitude"] = lat
16  body["street"] = street
17  body["placeName"] = placeName
18  backend.doPost(path: "/events/add", dict: body,token:
        FBSDKAccessToken.current().tokenString, callback: {
19     (nsdict) in
20     let eHolder:Holder<EventDto> = backend.parseHolder(json: nsdict)
21     if eHolder.success {
22        backend.alert(view: self, message: NSLocalizedString("EVENT_APPROVEMENT",
              comment: ""), title: NSLocalizedString("EVENT_ADDED", comment: ""))
23     }
24     else{
25        backend.alert(view: self, message: NSLocalizedString("Something went
              wrong!", comment: ""), title: "Error!")
26     }
27     self.navigationController?.popViewController(animated: true)
28  })
```

The UI Elements and values are getting set to the dictionary 'body' and after it the *BackendService* gets called with the `doPost()` method. When receiving data back from the backend the application is able to tell the user which has submitted the event before

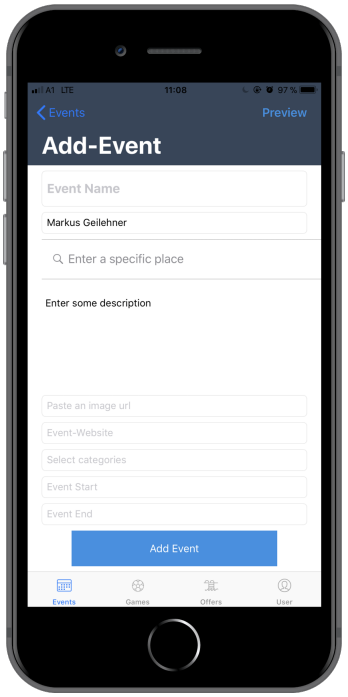


Figure 7.7: Add Event View

if it was successful or not. If it was successful the event has to be reviewed by "Ennui-Moderators" which have to unlock the event for all users. This system is for preventing spam events and other irrelevant content.

7.5.7 Offer List

When changing the Offers-Tab the first view which can be seen is the list of offer categories which are available to search for (see Figure 7.8). Those categories are basically cells in a `UITableViewController` and are dynamically loaded from the *BackendService* as follows.

```
1 let backend = BackendService()
2 backend.doGet(path: "/offers/categories", token: "", callback: {
3    (jsonstring) in
4    let jsonDecoder = JSONDecoder()
5    let jsonData = jsonstring.data(using: .utf8)!
6    if let acts:[String] = try? jsonDecoder.decode([String].self, from:
         jsonData){
```

```
7       for act in acts{
8          self.activities.append(backend.convertOfferCategoryToName(cat: act))
9       }
10      DispatchQueue.main.async {
11         self.tableView.reloadData()
12      }
13   }
14 })
```

The author has refactored the code above for visualized representation (The JSON-Parsing is used to be in an outsourced method). After decoding the retrieved *JSON* the activities are getting set and the UI gets again reloaded via the *DispatchQueue*.

When selecting one of the available offer categories the offers for a specific category will be listed (see Figure 7.8). In the code behind it is again a simple request to the Ennui backend.

```
1  func getOffers(){
2     let backend = BackendService()
3     backend.doGet(path: "/offers?category=" +
          backend.convertNameToOfferCategory(cat: activity) + "&latitude=" +
          String(self.userLocation.coordinate.latitude) + "&longitude=" +
          String(self.userLocation.coordinate.longitude), token: "") {
          (jsonstring) in
4        let offerHolder: Holder<OfferDto> = backend.parseHolder(json: jsonstring)
5        if offerHolder.success != nil && offerHolder.success{
6           self.offerItems = offerHolder.result
7           DispatchQueue.main.async {
8              self.tableView.reloadData()
9              self.activityIN.stopAnimating()
10          }
11       }
12
13    }
14 }
```

As in the request explained before, the author made again use of the *BackendService* and the `doGet()` method. After parsing the JSON and checking if the data is correct the it is set to the table and the UI reloads.

As it is by the events, offers also have a detail view which can be accessed by clicking on one of them (see Figure 7.8). As in every detail view specific information about the offer will be displayed (see Figure 7.8). Probably a user can not decide which offer would be best for him or her, therefore each offer has reviews based on Google (see Figure 7.8).

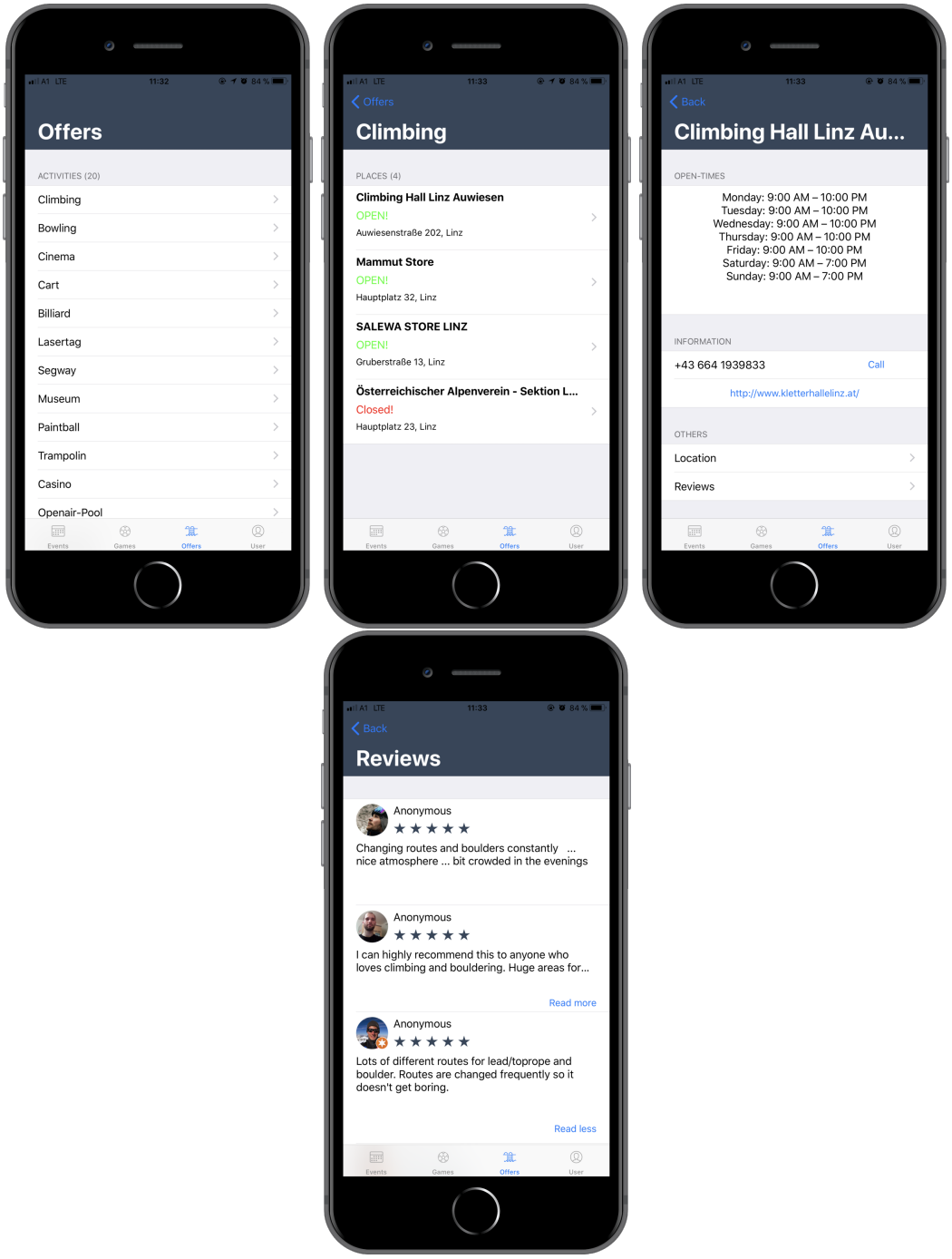


Figure 7.8: Aviable Offer Categories - List Of Offers Nearby - Detail View - Detail Reviews

7.5.8 Game List

When selecting the Game-Tab a list of games is loaded (see Figure 7.9). As it is in the other tabs, the games are not location based and basically all of them are loaded into the `UITableViewController`.

```
backend.doGet(path: path,token: token, callback: {
   (arr) in
   let gameHolder:Holder<GameDto> = backend.parseHolder(json: arr)
   if gameHolder.success != nil && gameHolder.success{
      self.gameItems = gameHolder.result
      self.setGameItemsToDisplay()
      DispatchQueue.main.async {
         self.loadingBr.stopAnimating()
         self.tableView.reloadData()
      }
   }
})
```

With the *BackendService* a `doGet()` is performed again to get all available and activated games in database. As we already now, the items are getting parsed and than the UI reloads after setting them to the table.

To filter the games a category can be chosen on the top of the view and those will be displayed. The 'Filter-System' is differently to the system behind the *EventTab*. It is much simpler and there only a variable is set to check which games have to be displayed. When clicking on the selectable filter a `PickerView` gets displayed, depending on what the user selects the games are filtered.

```
func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent
    component: Int) {
   categorySelection.text = categories[row]
   selectedCategory = categories[row]
   self.setGameItemsToDisplay()
   DispatchQueue.main.async {
      self.tableView.reloadData()
   }
}
```

With the method `setGameItemsToDisplay()` the correct game items according to the current filter are getting displayed.

```
func setGameItemsToDisplay(){
   gameItemsToDisplay = []
   if selectedCategory == categories[0]{
      gameItemsToDisplay = gameItems
   }
   else{
      let backend = BackendService()
```

```
8         for ele in gameItems{
9            if ele.categories
10              .contains(backend
11                 .convertReadableCategoryToGameCategory(cat: selectedCategory)){
12                 gameItemsToDisplay.append(ele)
13           }
14        }
15     }
16  }
```

If the filter is not set the `gameItems` currently in the global initialized array are used to display. Otherwise those games are iterated through and the filter checks if the game should be displayed or not.

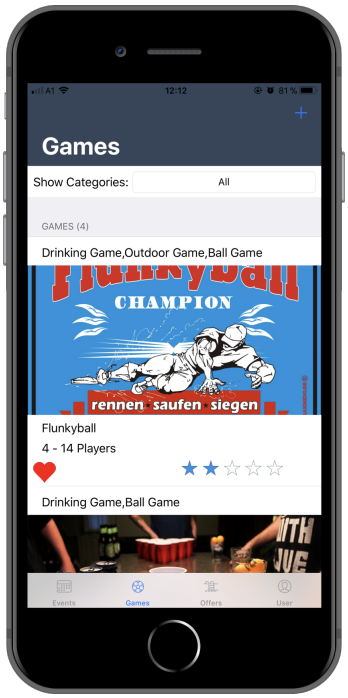


Figure 7.9: Game List

7.5.9 Favorize Games

The system behind favouring games is pretty similar to the system behind favouring the events. When the user clicks the heart displayed in each game-cell the hear turns red and the game gets favoured. As already known favouring games and events is only possible when logged in. So the event for favouring is added to the cell like this.

```
1  if(FBSDKAccessToken.current() != nil){
```

```
    cell.favorizeButton.isHidden = false;
    cell.favorizeButton.tag = dto.id;
    cell.favorizeButton.removeTarget(nil, action: nil, for: .allEvents);
    cell.favorizeButton.addTarget(self, action: #selector(favorizeGameHandler),
        for: .touchUpInside)
}
else{
    cell.favorizeButton.isHidden = true;
}
```

The target or in other words the action behind the button gets removed first and after it added again, that is because a target can have more events and that is not what the author wants to have in one favour button. This can also be a problem because the cells where the buttons are int, are reusable as the author already mentioned by explaining the event section. The `favorizeGameHandler` method is basically a post request to remove or add the game to the list of the users favoured games. Internally happens another interesting thing. After those requests which are sent to the backend the method `removeOrAddGameToFavoredGames(gameDto: getGameInList(id: sender.tag)!)` is called. Because the application has to refresh the games in the list if they are favoured by the user or not it would be inefficient to send again a request to the backend.

```
func removeOrAddGameToFavoredGames(gameDto: GameDto){
    if(userDto?.favouriteGames != nil){
        var i = -1;
        for (idx,dto) in (self.userDto?.favouriteGames.enumerated())! {
            if dto.id == gameDto.id{
                i = idx;
            }
        }
        if i >= 0{
            self.userDto?.favouriteGames.remove(at: i);
        }
        else{
            self.userDto?.favouriteGames.append(gameDto)
        }
    }
}
```

In this function the favourite games of a user are iterated through and are checked if they are still a favour of the user or not. According to the state of each game they are getting added to the favour list again or not.

7.5.10 Rate Games

Games can also be rated by clicking on one of the five stars displayed in the Game-Cell (7.3.7). When clicking again on one of the five stars, the rating will be changed for the user. For this system a library by Neumerzhitckii Evgenii was used [Neu18]

The rating UI is basically a `ViewController` with additional features (Read more by [Neu18]). As already known cells are reusable in Swift so we have to reset and set it's state.

```
if dto.ratedByUser{
            cell.ratingStars.rating = Double(dto.rating)
            cell.ratingStars.settings.filledColor = UIColor.init(red:
                41.0/255.0, green: 145.0/255.0, blue: 228.0/255.0, alpha: 1.0)
        }
```

The code snippet above checks if the user has already rated the game, if so the rating stars are set. To make the rating stars 'rateable' the library provided an event called `didFinishTouchingCosmos` where the author has handled the rating.

```
if FBSDKAccessToken.current() != nil{
   cell.ratingStars.settings.updateOnTouch = true
   cell.ratingStars.didFinishTouchingCosmos = {
      rating in
      let backend = BackendService()
      self.gameItemsToDisplay[indexPath.row].ratedByUser = true
      self.gameItemsToDisplay[indexPath.row].rating = Int(rating)
      backend.doPost(path: "/games/rate",
         dict: ["userId": self.userDto?.id ?? -1,
            "gameId": dto.id,
            "rating": Int(rating)],token:
                FBSDKAccessToken.current().tokenString, callback: {
            (nsdict) in
            //print(nsdict.value(forKey: "success") as! String)
      })
      DispatchQueue.main.async {
         self.tableView.reloadData()
      }
   }
}
else{
   cell.ratingStars.settings.updateOnTouch = false
}
```

The rating system is only for users which are logged in so if they are not this event gets disabled. Otherwise a backend call is sent with the user's id, gameId and the rating.

7.5.11 Add Games

Users also can add their own game and provide them for other users browsing through the App. This can be achieved by clicking on the "+" Button in the GameTab. Then the formula for a game appears and the user can fill it out (see Figure 7.10). The system behind is equal to the system behind adding an event (see 7.5.6).

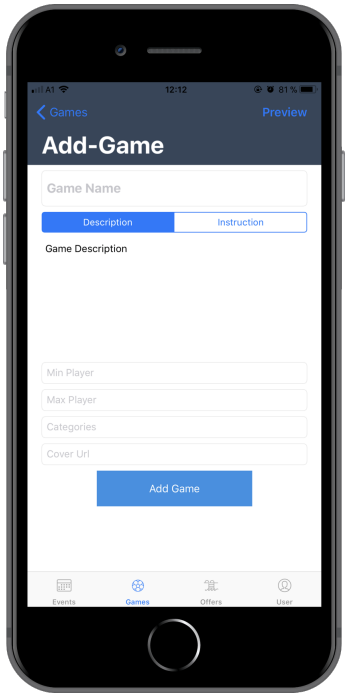


Figure 7.10: Add Game

7.5.12 User

Last but not least, there is the User-Tab (see Figure 7.11). In this view the currently logged in user, both favored events and games (see Figure 7.11) as well the logout button is displayed. The user is able to select his favourite events or favourite games, both are displayed in a new view and are not loaded new each time the user clicks on one of them. The reason behind not sending an *HTTP-Request* to the backend server is because it is more efficient loading the favourites out of the *User-Defaults* (see 7.3.9). How it works, in the "main view" of the *User-Tab* a segue (see 7.3.8) is initiated when the user clicks either on 'Favourite Events' or 'Favourite Games' with the events/games stored in the *User Defaults*.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
   if(FBSDKAccessToken.current() != nil){
      if let dest = segue.destination as? UserFavoredEventsController{
         dest.eventItems = self.userDto.favouriteEvents;
         dest.userDto = userDto;
      }
      else if let dest = segue.destination as? UserFavoredGamesController{
         dest.gameItems = self.userDto.favouriteGames
         dest.userDto = self.userDto
      }
```

```
11      }
12  }
```

In the above snippet of the prepare override function for changing the view with a segue, the first distinction is to distinguish if the initiated segue is leading to the `UserFavoredEventsController` or `UserFavoredGamesController`. Based on the leading controller the events or games are loaded for it.

The 'favourites view' is inherited from a simple `UITableViewController`, which makes it possible to perform a swipe on each cell. When the user now swipes from right to left a delete option appears to de-favour the game or event.

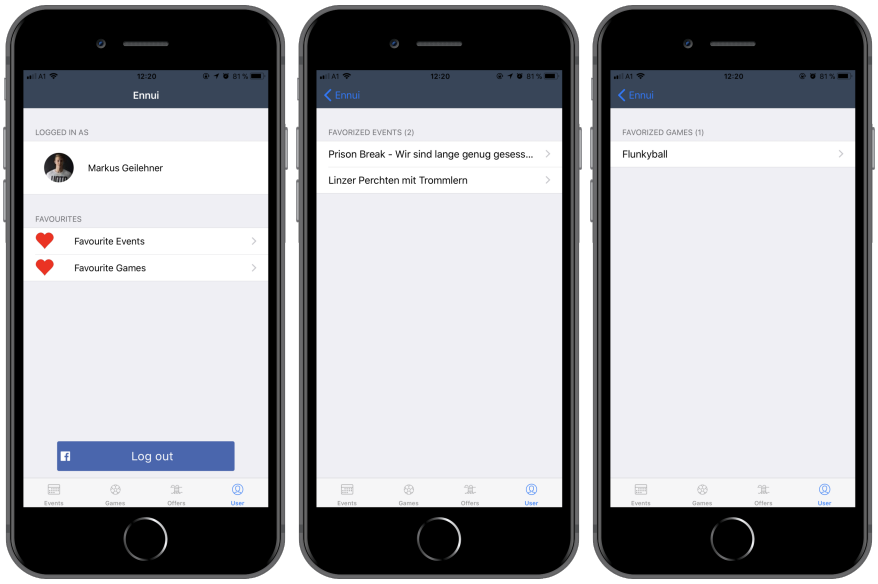


Figure 7.11: User Tab - Favored Events - Favored Games

7.5.13 Multi Language

In the Project-Hierarchy there's a *Localizable.strings* file where the location based strings are handled. Currently there's a German and an English version of it. To add new strings the Syntax like `"KEY"="VALUE";` has to be followed. To use the location based strings or something similar just write the following line:

```
1  NSLocalizedString("KEY-VALUE", comment: "")
```

The key value represents the key as the name says. The return value is a string with the value saved to the key in the language of the users phone.

7.6 Special Implementation Issues

7.6.1 From Swift 3 to Swift 4

In the beginning the author has written the application in Swift 3 later on Apple officially released Swift 4 with new features like the JSONEncoder. As already mentioned in Swift 3 there is no simple way where it is possible to convert JSON-Data in one line except installing some libraries. The way the author did it before was creating an object and converting the JSON-Data into a dictionary (See dictionary at 7.3.4) and parsing each value as optional (explanation of optionals 7.3.5) out of it. The old way to parse those data looked like this:

```
if let dict = ele as? NSDictionary{
   let dto = TaxiDto()
   dto.address_components = dict.value(forKey: "address_components");
   dto.international_phone_number = dict.value(forKey:
       "international_phone_number") as? String;
   dto.icon = dict.value(forKey: "icon") as? String;
   dto.name = dict.value(forKey: "name") as? String;
   dto.rating = dict.value(forKey: "rating") as? Int;
   taxiArray.append(dto);
}
```

Finally Swift 4 had the JSONEncoder and JSONDecoder implemented which made the way converting data much easier and faster. The above code looks like this in Swift 4:

```
if let jsonString = ele as? String{
   let jsonData = jsonString.data(using: .utf8)!
   let decoder = JSONDecoder()
   if let taxi = try? decoder.decode(TaxiDto.self, from: jsonData){
      taxiArray.append(taxi);
   }
}
```

7.6.2 Localization

Those events shown by "Ennui" are not limited to a local area it always lists events close to the users current location. Therefore the "Ennui" mobile app can be used worldwide and the target group for this app is also worldwide. This is the reason a localization has been implemented. After researching how Apple was used to this topic the author found some tutorial on the internet. By reading through the following tutorial it was easy to handle this feature in the iOS Application [Miz16].

7.6.3 Storyboard Refreshing

After implementing a library for the rating system (Read more about it in 7.5.10) the storyboard kept refreshing and building continuous which made working nearly impossible. The answer for this problem was found on stackoverflow [MrT15].

7.6.4 Blank Event Images

Some images were blank when browsing through the event tab. Later on the author found out that Facebook is changing the cover URLs of some events. The fix was quite simple, instead of using the URL getting from the Ennui backend, the application now makes an API call to Facebook to get the newest cover URL which is loaded afterwards.

7.7 Tools And Frameworks

7.7.1 Facebook SDK

The Facebook-SDK is used as user system in the application to verify the user and to get data from them. The advantages of using Facebook are the following points:

- Facebook is a secure system
- The Ennui backend itself has not to care much about user security and token generation because this is done by Facebook.

Disadvantages of using Facebook as login system:

- Not everyone on earth is using Facebook so they cannot login if they do not have an existing account on Facebook.

Read more about the Facebook SDK [Fac17]

7.7.2 Google Maps SDK

To get a location based auto complete and the offers, the Google Maps SDK is used. Read more about the Google Maps SDK [Goo17a]

Chapter 8

Explanation of Technology

8.1 Visual Studio Code

Visual Studio Code is a source code editor which runs on desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript, TypeScript and Node.js and has extensions for other languages (such as C++, C-Sharp, Java, Python, PHP, Go) and runtimes (such as .NET and Unity). Read more at [Mic17].

8.2 IntelliJ IDEA

IntelliJ IDEA is a capable and ergonomic IDE for JVM. After IntelliJ IDEA's indexed the source code, it offers experience by giving relevant suggestions in every context like instant and clever code completion, on-the-fly code analysis and reliable refactoring tools. Also a wide variety of supported languages and frameworks are at hand. Read more at [Jet17].

8.3 XCode 9

XCode is an environment for building apps for Mac, iPhone, iPad, Apple Watch and AppleTV. Some integrated features are for instance source control, an emulator for apple devices and since the new update the IDE is much faster than ever before. Read more at [App17b].

8.4 Android Studio

Android Studio is Androids official IDE. It is purpose built for Android to accelerate the development and helps building applications for Android devices. It offer tools custom-tailored for Android developers, including code editing, debugging, testing, and profiling tools. Read more at [Goo17b].

8.5 Weka

Weka [EHW16] is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a dataset or called from own Java code. Weka contains tools for data-preprocessing, classification, regression, clustering, association rules and visualization. It is also well-suited for developing new machine learning schemes. Read more at [oW17].

8.6 jQWidgets

jQWidgets provides a comprehensive solution for building web sites and mobile apps. It is built entirely on open standards and technologies like HTML5, CSS, JavaScript and jQuery. jQWidgets enables responsive web development and helps creating apps and websites. Read more at [jQW17].

8.7 Java

Java is a programming language which mostly is used for backend systems. The applications built with it are running on every system assumed Java is installed. Read more at [Ora17].

8.8 Spring Boot

Spring Boot is the starting point for building all Spring-based applications. It is a framework extension for Java and helps building REST, WebSockets and more. It has a security included and supports SQL and NoSQL. For server based systems, spring provides *Tomcat*, *Jetty* and *Undertow* as runtime support. Read more at [Piv17].

8.9 Swift

Swift is a general-purpose programming language built using a modern approach to safety, performance, and software design patterns. The goal of the Swift project is to create the best available language for uses ranging from systems programming, to mobile and desktop apps, scaling up to cloud services. Most importantly, Swift is designed to make writing and maintaining correct programs easier for the developer. Read more at [Inc17].

Chapter 9

Summary

9.1 General

This chapter serves as a conclusion of the diploma thesis. It summarizes all of the chapters above, with a focus on the results archived. The chapter starts with a short recap of the machine learning chapters and especially the three machine learning algorithms that where developed for this thesis. Afterwards the conclusions of the *Simulation Results* and the *Test Environment* will be presented. Finally this chapter covers the achievements of the mobile clients for Android and iOS.

9.2 Machine Learning and Methods for Matching

Chapter 2: Machine Learning serves as an introduction to the topic of Machine learning. It covers a variety of topics starting from the basic terminology of some technical terms, up to a detailed description of various types of classifiers. It also manages to tackle some of the problems faced when developing the algorithms explained in *Chapter 3: Methods for Matching* like over- and underfitting. Talking about the algorithms of *Chapter 3*, namely C4.5, CART and CHAID the results of each algorithm are outlined in the following.

C4.5 Aside from some general information section 3.1 not only contains a step-by-step instruction of the algorithms functionality, but also a documentation of a fully working Java implementation of the algorithm.

CART In section 3.2 the author presents a detailed and clear overview of the algorithms procedure. It also explains the most important statistical operations of the algorithm and presents source code of the authors Java implementation of the CART algorithm.

CHAID Section 3.3 covers a step-by-step overview of the algorithms procedure. The author also explains some of the most important statistical operations needed for

the CHAID algorithm and gives insight in the source code of his Java implementation.

9.3 Test Environment

Chapter 4 covers the implementation of the test environment, a web application for applying and testing the three algorithms explained in the section above. The requirement of a simple and functional environment to test the algorithms has been met.

9.4 Simulation Results

Chapter 5 covers a realized study for comparing the three algorithms explained in 9.2. In a two-step procedure the best fitting algorithm for Ennui was found. The first step consisted of tests in the test environment (explained in the section above) with hard-coded training data based on simulated habits of a test user. This tests were best executed by C4.5. The second step consisted of a test on the running system Ennui with real users and real event data. The training set got generated from user clicks, representing the habits of each user. This step also showed that C4.5 fits best to Ennui.

9.5 Mobile Clients

Chapters 6 and *7* cover an overview and also the corresponding authors implementation of a native mobile client for Ennui. Both native applications for Ennui are able to meet all the requirements given and present the data in a layout typical for their operating system. A list of these requirements includes

- A Facebook login
- Visualization of events, games and offers and their respective details
- Adding own events
- Filtering events
- Rating offers
- Adding games
- Filtering games
- Rating games

9.6 Overall Conclusion

Since the mobile clients are both fully working, each algorithm is having a complete Java implementation and the test environment being able to find the best suitable algorithm for the Ennui backend, it shows that all of these tasks can be conducted within the time frame of eight months with a team of three developers and about 180 working hours per person.

Bibliography

[A. 16] A. Shafaei, J. J. Little. CRV 16 | Real-Time Human Motion Capture with Multiple Depth Cameras, 2016. URL: http://www.cs.ubc.ca/~shafaei/homepage/projects/crv16.php.

[App17a] Apple Inc. UserDefaults - Foundation | Apple Developer Documentation, 2017. URL: https://developer.apple.com/documentation/foundation/userdefaults.

[App17b] Apple Inc. Xcode - Apple Developer, 2017. URL: https://developer.apple.com/xcode/.

[BFOS83] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth Publishing, Belmont, Calif, June 1983.

[Bro16] Jason Brownlee. Overfitting and Underfitting With Machine Learning Algorithms, March 2016. URL: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/.

[Bro17] Adi Bronshtein. A Quick Introduction to K-Nearest Neighbors Algorithm, April 2017. URL: https://medium.com/@adi.bronshtein/a-quick-introduction-to-k-nearest-neighbors-algorithm-62214cea29c7.

[efr17] efrique. Some questions about Chi-Squared Tests of Independence • r/AskStatistics, 2017. URL: https://www.reddit.com/r/AskStatistics/comments/78ijg0/some_questions_about_chisquared_tests_of/.

[EHW16] Frank Eibe, Mark A. Hall, and Ian H. Witten. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, 2016.

[Fac17] Facebook. Facebook iOS SDK - Documentation, 2017. URL: https://developers.facebook.com/docs/ios/.

[Fac18] Facebook. jQuery - Web-SDKs - Dokumentation, 2018. URL: https://developers.facebook.com/docs/javascript/howto/jquery.

[Goo17a] Google. Google Maps SDK for iOS, 2017. URL: https://developers.google.com/maps/documentation/ios-sdk/?hl=de.

[Goo17b] Google Inc. Download Android Studio and SDK Tools | Android Studio, 2017. URL: https://developer.android.com/studio/index.html.

[Goo18a] Google. Activity | Android Developers, 2018. URL: https://developer.android.com/reference/android/app/Activity.html.

[Goo18b] Google. AsyncTask | Android Developers, 2018. URL: https://developer.android.com/reference/android/os/AsyncTask.html.

[Goo18c] Google. Fragments | Android Developers, 2018. URL: https://developer.android.com/guide/components/fragments.html.

[HMS66] Earl B. Hunt, Janet Marin, and Philip J. Stone. *Experiments in induction*. Academic Press, 1966. Google-Books-ID: sQoLAAAAMAAJ.

[Inc17] Apple Inc. Swift.org, 2017. URL: https://swift.org.

[Jet17] JetBrains. IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains, 2017. URL: https://www.jetbrains.com/idea/.

[jQW17] jQWidgets. jQWidgets - JavaScript UI Widgets framework, 2017. URL: https://www.jqwidgets.com/.

[Kar03] Stephen T. Karris. Values of the Chi-squared distribution table, 2003. URL: https://www.medcalc.org/manual/chi-square-table.php.

[Kas80] G. V. Kass. An Exploratory Technique for Investigating Large Quantities of Categorical Data. *Applied Statistics*, 29(2):119, 1980. URL: http://www.jstor.org/stable/10.2307/2986296?origin=crossref, doi:10.2307/2986296.

[Koe17] William Koehrsen. Random Forest Simple Explanation, December 2017. URL: https://medium.com/@williamkoehrsen/random-forest-simple-explanation-377895a60d2d.

[Mac17] Machine Learning Plus. Logistic Regression - A Complete Tutorial with Examples in R, September 2017. URL: https://www.machinelearningplus.com/machine-learning/logistic-regression-tutorial-examples-r/.

[Mar17] Maruti Techlabs. How businesses can leverage reinforcement learning?, April 2017. URL: https://www.marutitech.com/businesses-reinforcement-learning/.

[Med18] Software Medcalc. 11.8: The Chi-Square Distribution | Engineering360, 2018. URL: https://www.globalspec.com/reference/69568/203279/11-8-the-chi-square-distribution.

[Mic17] Microsoft. Visual Studio Code - Code Editing. Redefined, 2017. URL: http://code.visualstudio.com/.

[Min89] John Mingers. *An Empirical Comparison of Selection Measures for Decision-Tree Induction*. 1989.

[Miz16] Taka Mizutori. iOS Localization Tutorial, July 2016. URL: https://medium.com/lean-localization/ios-localization-tutorial-938231f9f881.

[MrT15] MrTJ. ios - Xcode keeps building storyboard after each keystroke - Stack Overflow, February 2015. URL: https://stackoverflow.com/questions/28476030/xcode-keeps-building-storyboard-after-each-keystroke/28500914.

[Neu18] Evgenii Neumerzhitckii. Cosmos: A star rating control for iOS/tvOS written in Swift, January 2018. original-date: 2014-11-28T04:23:24Z. URL: https://github.com/evgenyneu/Cosmos.

[noa] CHAID and Exhaustive CHAID Algorithms. URL: ftp://ftp.software.ibm.com/software/analytics/spss/support/Stats/Docs/Statistics/Algorithms/13.0/TREE-CHAID.pdf.

[Obj18] ObjectDB. ObjectDB - Fast Object Database for Java with JPA/JDO support, 2018. URL: https://www.objectdb.com/.

[Ora17] Oracle. Was ist Java?, 2017. URL: https://java.com/de/about/whatis_java.jsp?bucket_value=desktop-chrome65-windows10-64bit&in_query=no.

[otRSS18] Journal of the Royal Statistical Society. Journal of the Royal Statistical Society: Series C (Applied Statistics) :, 2018. URL: https://rss.onlinelibrary.wiley.com/hub/journal/14679876/overview.

[oW17] University of Waikato. Attribute-Relation File Format (ARFF), 2017. URL: https://www.cs.waikato.ac.nz/ml/weka/arff.html.

[Piv17] Pivotal. spring.io, 2017. URL: https://spring.io/.

[Qui86] J. Ross Quinlan. *Induction of Decision Trees*. 1986.

[Qui14] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Elsevier, June 2014. Google-Books-ID: b3ujBQAAQBAJ.

[Ric18] Richard Routledge. Bayes's theorem | Definition & Example, February 2018. URL: https://www.britannica.com/topic/Bayess-theorem.

[Spr18] SpringBoot. Spring Boot, 2018. URL: https://projects.spring.io/spring-boot/.

[Sta17] StackOverflow. jquery - How to read data From *.CSV file using javascript? - Stack Overflow, 2017. URL: https://stackoverflow.com/questions/7431268/how-to-read-data-from-csv-file-using-javascript.

[Sta18] StackOverflow. Cross Validation in Weka - Stack Overflow, 2018. URL: https://stackoverflow.com/questions/10437677/cross-validation-in-weka.

[Sun15] Sunil Ray. Beginners Guide To Learn Dimension Reduction Techniques, July 2015. URL: https://www.analyticsvidhya.com/blog/2015/07/dimension-reduction-methods/.

[Sun17a] Sunil Ray. Essentials of Machine Learning Algorithms (with Python and R Codes), September 2017. URL: https://www.analyticsvidhya.com/blog/2017/09/common-machine-learning-algorithms/.

[Sun17b] Sunil Ray. Understanding Support Vector Machine algorithm from examples (along with code), September 2017. URL: https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/.

[Tav15] Tavish Srivastava. Getting smart with Machine Learning - AdaBoost and Gradient Boost, May 2015. URL: https://www.analyticsvidhya.com/blog/2015/05/boosting-algorithms-simplified/.

[Vog18] Vogella. Java persistence API - Tutorial, 2018. URL: http://www.vogella.com/tutorials/JavaPersistenceAPI/article.html.

[Wik17] Wikipedia. Chi-square automatic interaction detection, August 2017. Page Version ID: 798187503. URL: https://en.wikipedia.org/w/index.php?title=Chi-square_automatic_interaction_detection&oldid=798187503.

[Wik18a] Wikipedia. Chi-squared test, April 2018. Page Version ID: 833536956. URL: https://en.wikipedia.org/w/index.php?title=Chi-squared_test&oldid=833536956.

[Wik18b] Wikipedia. Cluster analysis, March 2018. Page Version ID: 830532702. URL: https://en.wikipedia.org/w/index.php?title=Cluster_analysis&oldid=830532702.

[Wik18c] Wikipedia. Statistical classification, March 2018. Page Version ID: 831678129. URL: https://en.wikipedia.org/w/index.php?title=Statistical_classification&oldid=831678129.

[Wik18d] Wikipedia. Tree structure, February 2018. Page Version ID: 825245394. URL: https://en.wikipedia.org/w/index.php?title=Tree_structure&oldid=825245394.

[Wik18e] Wikipedia. Tree traversal, March 2018. Page Version ID: 831268480. URL: https://en.wikipedia.org/w/index.php?title=Tree_traversal&oldid=831268480.

List of Figures

List of Tables