Performance Predictor: Machine learning based prediction of user performance in the game 'Plunder Planet'

Bastian Morath

Bachelor Thesis
September 2018

Prof. Dr. Markus Gross

Supervisors:
Rafael Wampfler, Severin Klingler

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

cgl
computer graphics laboratory

Abstract

In this thesis, we developed a machine learning model that predicts the in-game performance of users in the game Plunder Planet by anticipating obstacle crashes. We extracted different features from log files of 19 game sessions of young adolescents that we used to train both classical machine learning models and recurrent neural networks. Finally, the different models were compared. There was no overall superior classifier, but instead there were trade-offs depending on whether the aim was to predict as many obstacle crashes correctly as possible, or rather to focus on being precise in anticipating the crashes. Predicting the in-game performance would enable the automatic adjustment of the level of difficulty to the user's physical and emotional state, allowing for a fast entry into a Dual Flow, a state in which the player is neither over- nor under-challenged, thus offering a better fitness experience to the player.

Zusammenfassung

In dieser Arbeit haben wir Machine Learning Modelle entwickelt, welche die Leistung von Jugendlichen im Spiel Plunder Planet vorhersagen, indem sie Zusammenstösse mit Hindernissen antizipieren. Wir extrahierten verschiedene Features aus den Logfiles von 19 Spielrunden junger Jugendlicher, auf welchen wir sowohl klassische Klassifizierungs-Modelle als auch Recurrent Neural Networks trainierten. Am Schluss haben wir diese Modelle untereinander verglichen. Es gab kein insgesamt überlegenes Modell, sondern Kompromisse zwischen so viele Abstürze wie möglich korrekt vorherzusagen, und Abstürze präzise zu antizipieren. Die Vorhersage der Leistung im Spiel würde die automatische Anpassung des Schwierigkeitslevels an den physischen und emotionalen Zustand des Spielers ermöglichen, was einen schnellen Einstieg in einen sogenannten Dual Flow ermöglicht, einen Zustand, in dem der Spieler weder über- noch unterfordert ist. Dadurch profitiert der Spieler von einem besseren Fitness-Erlebnis.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Z hdk
Zürcher Hochschule der Künste
Game Design

cgl
computer graphics laboratory

Bachelor Thesis

Performance Predictor: Machine learning based prediction of user performance in the game “Plunder Planet”

Abb.: “Plunder Planet” (Source: Subject Area in Game Design, ZHdK)

Project Description

“Plunder Planet” (developed by the Subject Area in Game Design, ZHdK in cooperation with Koboldgames GmbH) is an adaptive computer-based exergame (fitness game) environment providing an individual gameplay/training experience for children and young adolescents. Depending on the player’s fitness level and in-game performance the difficulty and complexity of the game vary. In a previous study detailed information about the user actions as well as their current heart rate was collected. For a subset of the participants we additionally have video data from the play sessions. In this thesis we want to analyse this data set to build predictive models for the user performance. This would allow to automatically adjust the difficulty and complexity level of the game in real-time and could provide important insights into the gameplay.

Tasks

The main task of the thesis is the development of a data-driven model for the future performance of users playing the exergame “Plunder Planet”. The model should be able to predict future heart rate and the ability to sidestep obstacles and to collect crystals in real time. Furthermore, an analysis of the best performing models should allow for insights into how to improve the gaming experience further. The three main tasks of this project are

- Validation and statistical analysis of the log files containing in-game events and heart rate.
- Timestamp synchronization and feature extraction from the video data.
- Creating a predictive model for the user performance based on features extracted from the log files and the video data.

Remarks

A written report and an oral presentation conclude the thesis. The thesis will be overseen by Prof. Markus Gross and supervised by Rafael Wampfler, Dr. Severin Klingler and Dr. Anna Martin-Niedecken (ZHdK). The start date of the thesis is March 5, 2018. The end date of the thesis will be September 5, 2018.

Acknowledgments

I thank my supervisors Rafael Wampfler and Severin Klingler for their enormous support, advice, and valuable input during all the different stages of this thesis. Furthermore, I thank Anna Martin-Niedecken for trusting me with this interesting and cutting-edge project.

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1

2 Related Work 3

3 Dataset 5
3.1 Plunder Planet . . . 5
3.2 Log Files . . . 7

4 Modeling 11
4.1 Feature Extraction . . . 12
4.2 Pre-Processing . . . 15
4.2.1 MinMaxScaler . . . 15
4.2.2 Box-Cox Power Transformation . . . 15
4.3 Feature Selection . . . 16
4.4 Classifiers . . . 16
4.4.1 *k*-Nearest Neighbors . . . 18
4.4.2 Support Vector Machine . . . 18
4.4.3 Random Forest . . . 20
4.4.4 Gaussian Naive Bayes . . . 20
4.4.5 Long Short-Term Memory Network . . . 21

5 Results 23
5.1 Experimental Setup . . . 23
5.1.1 Hyperparameter Tuning . . . 23
5.1.2 Statistical Measures . . . 24

5.1.3 Computing Environment 28
5.2 Window Sizes 28
5.3 Performance Analysis 28
5.3.1 Classical Machine Learning Models 30
5.3.2 Long Short-Term Memory 32

6 Conclusion **35**

7 Future Work **37**

Bibliography **38**

List of Figures

3.1 Plunder Planet in-game design 6
3.2 Plunder Planet FBMC setup 6
3.3 Heart rate artifacts 8
3.4 Heart rate and in-game events 9
3.5 Difficulty levels across all log files 10

4.1 Modeling pipeline 11
4.2 Mean heart rate at obstacle when a crash happens versus no crash happens . . . 14
4.3 Correlations between all initial features when using the entire data set. 17
4.4 Separating Hyperplane of an SVM 19
4.5 Example of a Decision Tree 20
4.6 Structure of an RNN and its unrolled representation 22

5.1 10-fold cross-validation 25
5.2 Youden's J statistic 27
5.3 Performance with varying window sizes 29
5.4 ROC curves of classical models 30
5.5 Leave-one-group-out cross-validation performance 31
5.6 AUC plot of one training round of training the LST 33
5.7 Visualization of our final LSTM network 34

List of Tables

3.1 All measurements that were tracked during a game session of Plunder Planet . . 7

4.1 The different window types 12
4.2 Summary of all features 13

5.1 All hyperparameters that were tuned 24
5.2 Hyperparameter tuning in Neural Networks 25
5.3 Confusion matrix for a two-class classification problem 26
5.4 Performance overview of classical models with 10-fold cross-validation 30
5.5 Performance scores of LSTM network 32

1

Introduction

Digital games combining both physical and psychological fitness and gaming, called *exergames*, emerged in the 1980s. According to Staiano et al. [SC11], exergames promise improvements in the physical state of a player (increase in caloric expenditure, coordination and heart rate), the psychosocial state (social interaction, mood and motivation), and the cognitive state (spatial awareness and attention).

One such exergame is *Plunder Planet*, a dynamically-adaptive exergame developed by Martin-Niedecken and Götz [MNG16, MNG17, MN17, MN18]. The set-up of the game asks the player to navigate a flying pirate ship through a desert filled with obstacles (see Figure 3.1) and to defend him- or herself against giant sandworms by activating a shield. The user gets points awarded by collecting crystals, and points deducted after each collision with an obstacle or a sandworm. Currently, the difficulty level of the game can be set manually by a person observing the user. The goal of this thesis was to create a machine learning (ML) model that predicts the in-game performance of the user. This would enable to automatically adjust the level of difficulty to the user's physical and emotional state, allowing for a fast entry into a so-called *Dual Flow* [SHM07], a state in which the player is neither over- nor under-challenged, thus offering a better fitness experience to the player.

Focus of this Work. Based on log files of users playing the game, we created an ML model that anticipates the user's in-game performance by predicting whether or not the user is going to crash into the next obstacle. The modeling step consisted of analyzing and validating log files, and extracting, pre-processing and selecting features. Different metrics were used to evaluate the performance of our models.

Major contributions:

- Developing a predictor of the in-game performance in the game Plunder Planet.
- Using ML to improve the user's exergame experience.

Thesis Organization. In Chapter 2, we examine work related to ML in classical and video games, student modeling, and Plunder Planet itself. In Chapter 3, we explain the game environment of Plunder Planet in detail, present the measured data as well as key numbers of our log files. We then proceed to explain the modeling pipeline, which consists of feature extraction, pre-processing, and feature selection, along with the different classifiers in Chapter 4. In Chapter 5, we evaluate the models and examine the results. We then conclude and discuss future work in Chapters 6 and 7, respectively.

2

Related Work

Artificial Intelligence in Classical Games. Since the mid-1990s, *Artificial Intelligence* (AI) and ML gained rapid momentum, a major contributor being the computer power that was to double every two years [Moo65, M$^+$75]. In 1997, *Deep Blue* became the first computer to beat the then reigning world champion in chess, Garry Kasparov. In 2011, *Watson*, an AI capable of answering natural language questions, defeated two former champions at *Jeopardy!*, an American television game show. Only five years later, Google's *AlphaGo* defeated *Go* champion Lee Sedol.

Artificial Intelligence in Video Games. Recently, a lot of attention has been paid to video games, which are usually much harder for computers to play since they often are *imperfect information games*: The computer does not know all aspects of the game. This stands in contrast to chess, where the computer knows the position of every chess piece at every point in time. The problem with game bots is that they are often scripted and follow a set of rules. For example, in first-person shooters bots are able to move around in an environment and to shoot enemies, but they often do not learn through the interaction with the player, no matter how many times a player exploits a weakness of the bot. Advances in this area have been made for example in [SBM05] and [MG08], where agents evolve and adapt to the players and the environment in real time. Earlier this year, *OpenAI* came very close to beat top professional *DOTA 2* players.

Exergames. Some of the first commercially available exergame consoles were the *Nintendo Wii*® and the *Microsoft Kinect*®, which to date are two of the most popular consoles of this genre. There are various studies that try to improve physical activity by using technology. For instance, *Bačić* [Bač18] developed an approach that identifies erroneous executed tennis swings, while *Pullen* and *Seffens* [PS18] built an ML solution for gesture detection, which improves yoga skill acquisition through a video exergame.

Student Modeling. Student modeling is concerned with intelligently tutoring students. The fundamental task is to estimate and predict the students' current knowledge. Accurately adapting the difficulty levels to the knowledge of each individual user is crucial for a good *Intelligent*

Tutoring System. *Performance Factors Analysis* [PJCK09] and *Bayesian Knowledge Tracing* [CA94] are two popular methods for estimating student knowledge. Efficient methods for modeling prerequisite skill hierarchies have been successfully applied [KKSG17].

Plunder Planet. Martin-Niedecken and Götz [MNG16] showed that Plunder Planet increased the effectiveness, attractiveness, and motivation of users playing the game. They analyzed the user experience of the players [MN17], which was measured by evaluating the immersion [BC04], presence [WW11] and flow [SHM07] of the users. A non-adaptive and an adaptive version of the game, and two different controller types for the game were analyzed in [MNG17]. A multiplayer version was presented and studied on its potential of improving the social and fitness aspects of the game [MN18].

3

Dataset

In Section 3.1, the game Plunder Planet is explained in more detail. We then proceed to present the log files in Section 3.2, and indicate some key numbers and information along with plots of the data measured during the gaming sessions.

3.1 Plunder Planet

Plunder Planet is a dynamically-adaptive exergame developed by Martin-Niedecken and Götz [MNG16]. According to Sinclair et al. [SHM07], an optimal training experience during an exergame session requires a balance between the game related challenge and player skills (psychological aspect), as well as between the intensity of the required movement input and the player's fitness level (physiological aspect). This condition is called Dual Flow, and it is what Plunder Planet aims its players to enter.

The game targets children and young adolescents. Its goal is to navigate a flying pirate ship through a desert full of obstacles (Figure 3.1) and to activate a shield in order to defend oneself against giant sandworms. The user is rewarded with points by collecting crystals (100 points per crystal), but each collision with an obstacle or a sandworm deduces 50 points. After completing a how-to tutorial, there is a warm-up period of 190 seconds, and only then in-game actions such as sandworms appear. After around 5 and 7.5 minutes, the game decelerates and the user has to repair his or her ship by rapidly pressing two buttons. This intermezzo is designed to relieve from cognitive tasks, and to increase physical user input.

The player can choose between two forms of input devices to navigate. The *Full-Body-Motion Controller* (FBMC) consists of large physical buttons with haptic feedback (Figure 3.2). The other device is a Kinect®sensor. In this thesis, we only focused on the FBMC controller, which challenges the player's cognitive and coordinative abilities [MN17].

Figure 3.1: *Plunder Planet in-game design. The user navigates a pirate ship around obstacles and tries to collect crystals. Source: Zurich University of the Arts, 2017.*

.

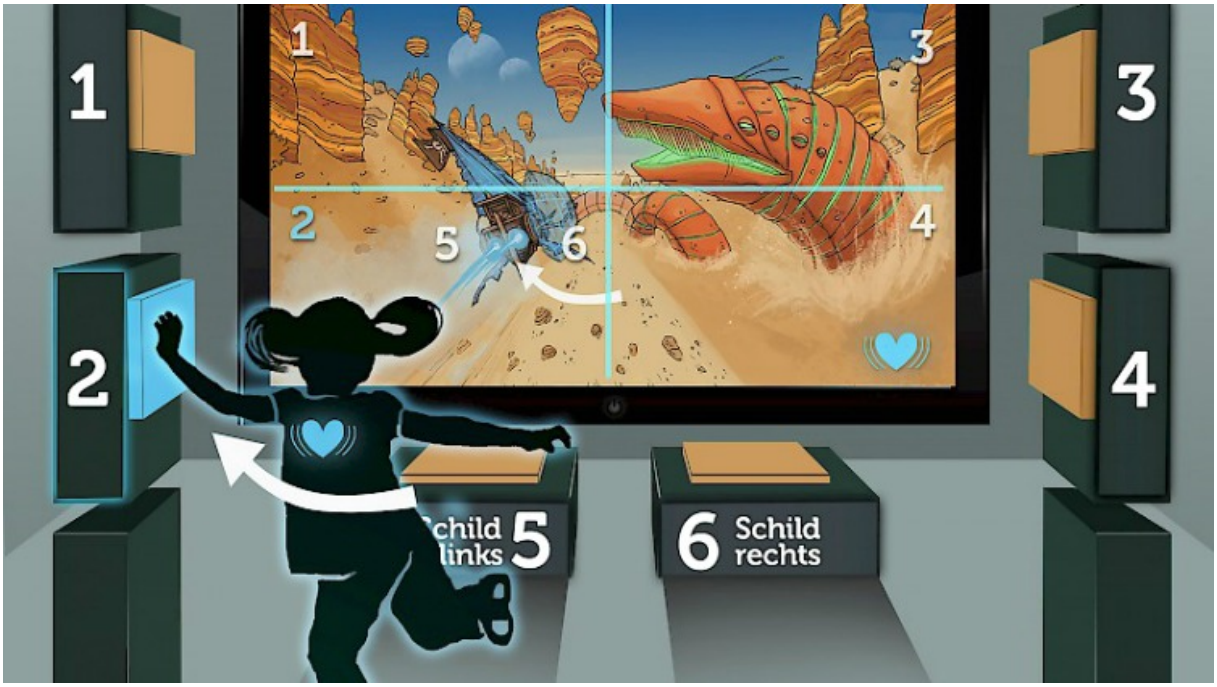


Figure 3.2: *With the FBMC, the player can press buttons labeled one to four to navigate the pirate ship, while the buttons five and six enable the user to defend himself against giant worms. Courtesy of [MNG17].*

.

At the time of writing this thesis, the game mechanics had to be adjusted manually to the user's physical and emotional state via a Trainer-GUI in real-time by a person observing the user. There are two different components of difficulty: a physiological and a psychological one. While the physiological dimension is associated with the frequency of obstacle encounters, the psychological one consists of desert track designs of varying difficulty and different obstacle arrangements.

The aim of this thesis was to develop a model that predicts the obstacle crashes of a user. This would enable to automatically adjust the the two components of difficulty to the user's physical and emotional state.

3.2 Log Files

The data consists of 37 log files. For 19 of these files, heart rate data was recorded, and only these files were relevant to this thesis and will therefore be referred to in the discussion.

One log file corresponds to one game played by a user. In total, there are 11 users with either one or two log files each. The average playing time per user is 621 seconds with a minimum and maximum of 609 and 657 seconds, respectively. We obtained a total of 196 minutes of playing time.

There are 5829 obstacles overall. In 1049 cases (roughly 20 %), the users crashed into an obstacle.

All the relevant data that was measured during a game is listed in Table 3.1.

Table 3.1: *Description of all data that was measured during a game session of Plunder Planet.*

Measurement	Description
Time	Time at which the event happened with up to 10 milliseconds precision.
Logtype	Records whether the user picked up a crystal, passed/ crashed into an obstacle, or whether nothing happened at all.
Score	Current total of points.
Heart rate	Beats per minute, measured with a Polar®H7 sensor.
Difficulty	Difficulty level. Note: Even though the psychological and physiological difficulty would be tracked separately in theory, in our log files they were both represented with the same value.
Obstacle	Encoded the obstacle arrangement (in case there was one).

In the following we are going to discuss the different measurements.

Heart Rate. The heart rate was measured approximately each second. In order to test for artifacts, one approach is to take two subsequent measurements, and every time these two measurements differ by more than 25 % to 35 %, they are considered an artifact. We direct the reader to [BQJB90] for more information on artifact identification. In our study, no log file showed a significant amount of heart rate changes of more than 10 % from one measurement to the next (Figure 3.3). We thus did not have to remove or correct artifacts.

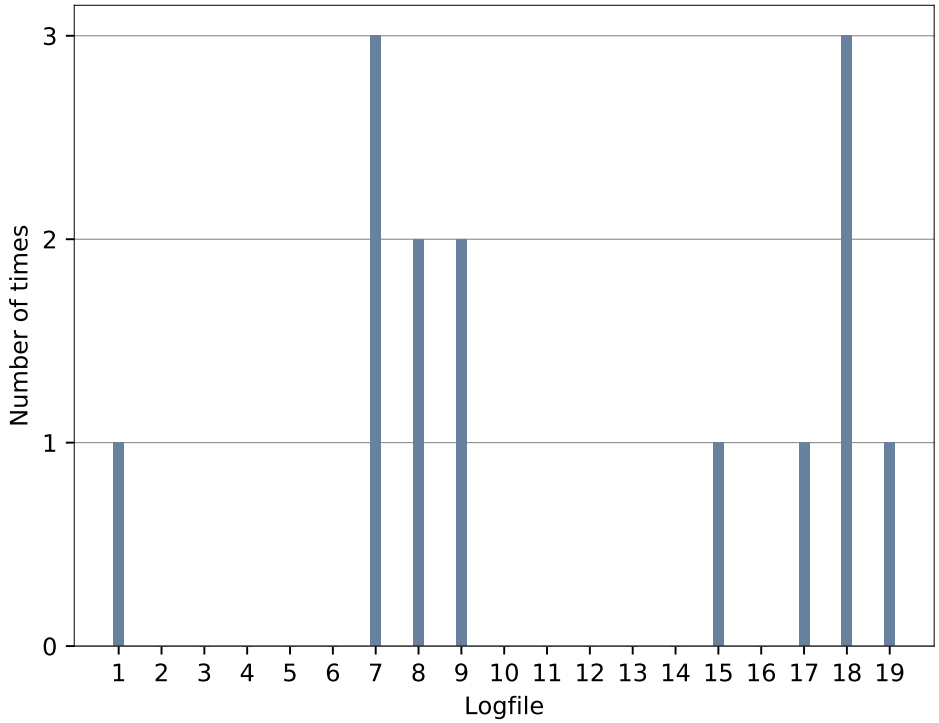


Figure 3.3: *For each log file, the number of times the heart rate changed more than 10 % is shown. No log file shows a significant amount of heart rate changes of more than 10 % from one measurement to the next.*

In-Game Events. In Figure 3.4, the heart rate as well as important in-game events from one specific user are shown. This plot is representative for most of the other log files. A slight tendency of rising heart rate towards the end of a game session is observable. Whenever the user had to repair a ship (by rapidly pressing the two buttons labeled two and three in Figure 3.2), the heart rate of most users went up as expected since this repairing of a ship increased the user's physical input while at the same time offered relief from cognitive tasks (Figure 3.4).

We also observe that apart from the first two to three minutes, when the user got accustomed to the game play at an easier difficulty level, there was usually no significant change in the number of crashes over the course of the game anymore.

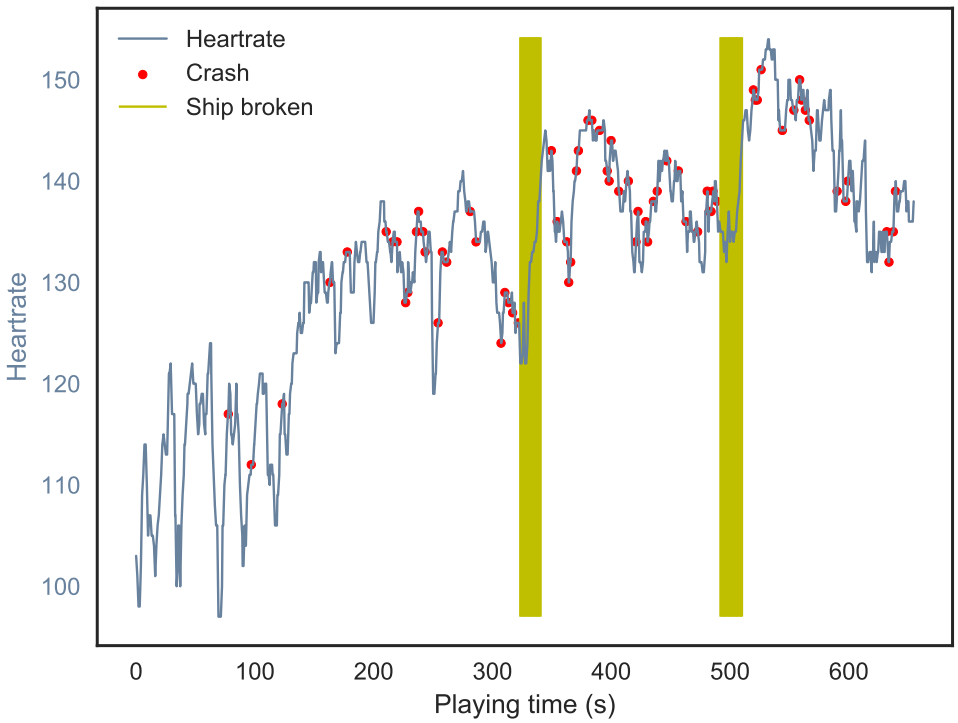


Figure 3.4: *The heart rate of a specific user and the in-game events are indicated. We observe that the heart rate of the user rises whenever a Ship Broken event happens, and note that the amount of crashes virtually does not change anymore after the first two to three minutes.*

Difficulty. Even though the levels of difficulty could theoretically be adjusted in steps of 0.1, our log files only contain three levels: *low* (interval [0.0, 0.3]), *medium* (interval [0.4, 0.6]) and *high* (interval [0.7, 1.0]). Across all log files, the users played 87 % on level *high* and 12 % on level *medium*, while 70 % of level *medium* happened during the first 100 seconds. As stated previously, the psychological and physiological difficulty could be tracked separately in theory, but in our log files they always correspond to the same value. We first assumed that a higher level of difficulty could hint at a higher crash rate of the user. But since the users were on level *high* almost the entire time (Figure 3.5), we could not make use of features including difficulty level information, because there was not enough variation in them.

Crystals. We assumed that in addition to the information whether or not a user crashed into an obstacle, we could make use of the fact whether the user picked up or missed a crystal to get information about the users physical and emotional state. Missing a crystal could indicate that the user was overwhelmed or stressed, which we assumed to translate into a higher crash rate. Unfortunately, the event of a user missing a crystal was not recorded in our log files, only when he picked up a crystal, and since crystals appeared randomly on the track, this information could not be used in the feature extraction step.

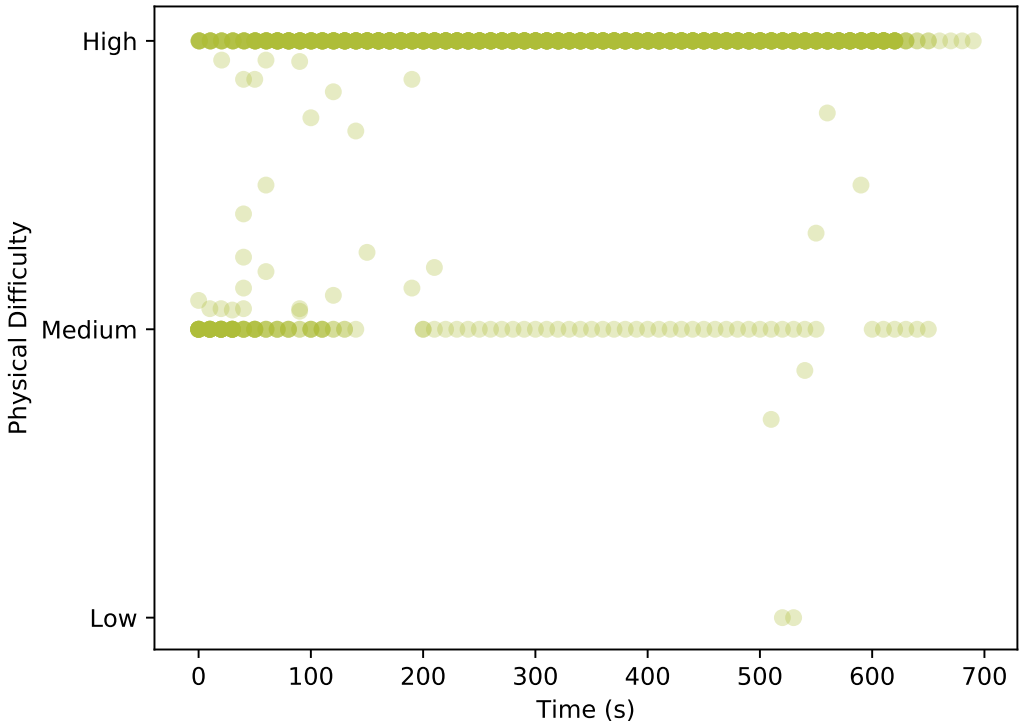


Figure 3.5: *The difficulties across all users after every 10 seconds are shown. Most of the time, the physical difficulty (and thus also the psychological one) is on the highest level.*

4

Modeling

In this chapter, we will explain how we have constructed our model which aims at predicting whether or not the user will crash into an obstacle.

The first step usually taken when building an ML model is to extract the relevant data, called *features*, from the log files, with which the model can be trained. The model can assign a label to a new unseen data point and indicate crash or no crash for each data point. This step is explained in Section 4.1. We will then explain in detail how we pre-processed the features to make them more suitable for the models to work with, which is important for example when a classifier expects its input to follow a certain distribution. This is investigated in Section 4.2. We proceeded to do feature selection in order to improve generalization and interpretability, as demonstrated in Section 4.3. In Section 4.4, we will describe the two different sorts of classifiers we tested: classical ML models and *Recurrent Neural Networks* with *Long Short-Term Memory* (LSTM) units, which try to capture long-term dependencies in our data. The pipeline can be seen in Figure 4.1.

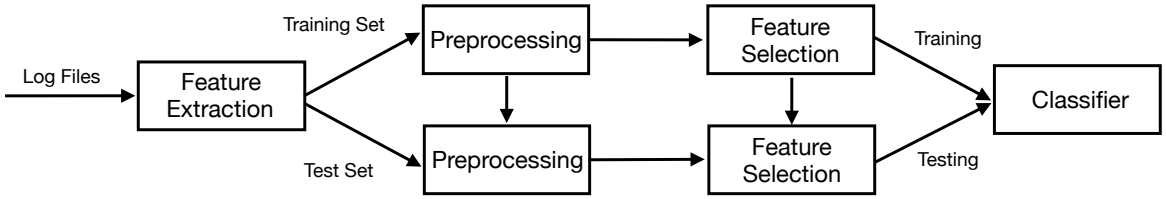


Figure 4.1: *The four steps taken to generate a classifier from user input log files.*

4.1 Feature Extraction

This section explains in detail what type of features were extracted from our log files. This resulted in a feature matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, where n is the number of data points and d is the number of features. The vector $\mathbf{y} \in \{-1, +1\}^n$ encodes the label for each data point, where $y_i = +1$ corresponds to a crash and $y_i = -1$ to no crash at the i^{th} obstacle, and can be easily read out from the log files.

Since we have multiple log files, we extracted one feature matrix $\mathbf{X}_i$ for each file and then concatenated those to our final matrix $\mathbf{X}$:

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \dots \\ \mathbf{X}_k \end{bmatrix},$$

where k is the number of log files.

Sliding Window Feature Extraction. Standard classification models cannot directly be applied to raw time-series data. Instead, we first have to transform it into meaningful features. To accomplish this, we used a sliding window approach. Since we wanted to predict a crash at obstacle i, we generated features by looking at the last couple of seconds before that obstacle appeared and calculated some metric over it.

In particular, we used three different window sizes depending on the feature we calculated (Table 4.1). We tested different window sizes for their performance on predicting crashes, the results of which can be found in Chapter 5.3. The idea behind using different window sizes for different features was that in order to predict a crash, we assumed that it would be much more indicative whether the user crashed into the *last* obstacle rather than whether he had crashed into one 30 seconds beforehand. Thus we figured that features concerning crashes might require a smaller window than for example features related to heart rate, which we expected to need a bigger window in order to show a trend.

Table 4.1: *The different window types we used to extract features.*

Window	Size	Description
crash window	10 s	Used for features concerning crashes.
default window	20 s	Used for features concerning the heart rate and the score.
gradient window	30 s	Used for features concerning gradient changes and linear regression.

In total, 17 features were calculated for each data point (see Table 4.2). In the following paragraphs, we are going to explain the concepts behind the different feature types.

Table 4.2: *Features along with their range, description and window size. The features in bold are the ones that are highly correlated when comparing the correlations of the features across all log files.*

	Range	Description	Window
Heart Rate			
Heart rate statistics	[63..165] *	Using the heart rate, we calculated six features (**mean_hr**, **std_hr**, max_hr, min_hr, **max_minus_min_hr** and max_over_min_hr). * Range of heart rate data.	default window
lin_regression_hr_slope	[-1.24, 1.5]	The slope of the fitting linear regression line over the heart rate.	gradient window
hr_gradient_changes	[0..8]	How many times the heart rate changed from going up to going down or vice versa.	gradient window
Score			
Score statistics	[0..5250] **	Using the score, we calculated five features (**mean_score**, **std_score**, max_score, min_score and **max_minus_min_score**). ** Range of score data.	default window
score_gradient_changes	[0..7]	How many times the score changed from going up to going down or vice versa.	gradient window
Crashes			
%crashes	[0, 1]	The percentage of crashes into obstacles.	crash window
last_obstacle_crash	[0..1]	Whether or not the user crashed into the last obstacle.	no window
Time			
timedelta_to_last_obstacle	[1.0, 2.8]	The time to the last obstacle in seconds.	no window

Heart Rate Features. As we stated previously, the goal of Plunder Planet is to keep the user in a so called *Dual Flow*, a well-balanced adjustment of the physical and psychological aspects of the game. A higher heart rate indicates that he or she is physically or mentally challenged, maybe even overwhelmed, which might affect his or her in-game performance. Users with increased heart rate might crash into obstacles more often. On the other hand, if the heart rate is close to the resting heart rate, the user might not be challenged enough. As a consequence, he or she is likely to perform better in the game, thus lowering the chance of crashing into obstacles.

To test our assumption, we investigated the average heart rate at both obstacle crashes and successful obstacle passings (Figure 4.2). Although the difference is certainly not significant, a slight trend towards a higher heart rate at a crash is observable.

Furthermore, if the heart rate changes strongly and suddenly (indicated by the features *std_hr*, *max_minus_min_hr* and *max_over_min_hr* and), this can hint that the difficulty level of the game has become too high or too low and the user feeling overwhelmed or stressed. An overwhelmed user is expected to crash into an obstacle more often than a relaxed user.

As explained previously, each user first plays a how-to tutorial, during which he or she gets accustomed to the game. In order to account for the different baseline heart rates of the individual users, we normalized it by dividing all the heart rate data of this user by the minimum heart rate of his or her how-to tutorial. The tutorial was later removed from the data set when using the ML model.

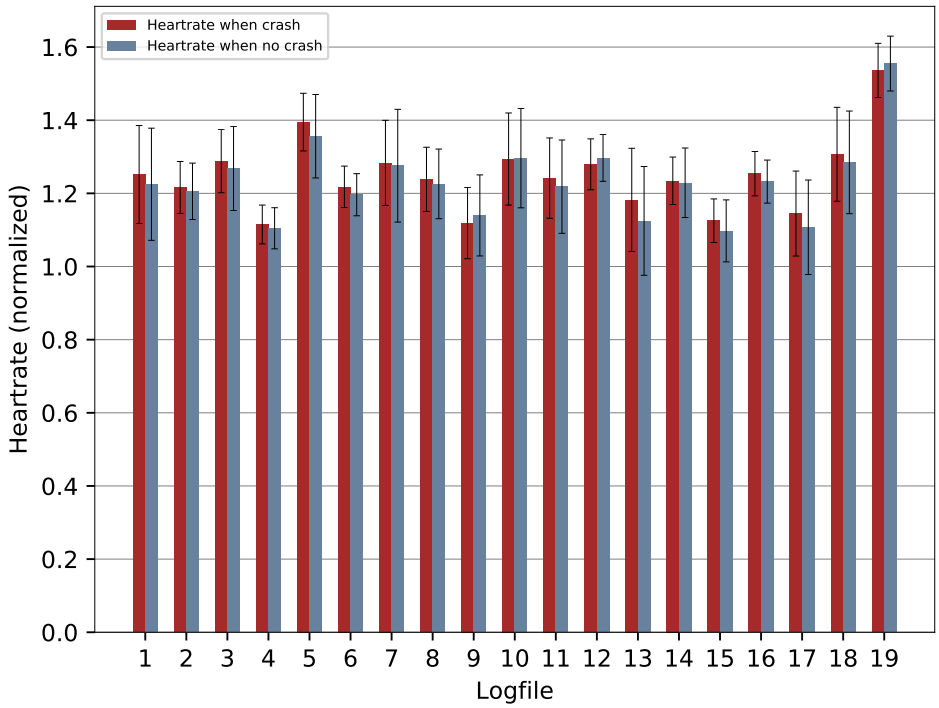


Figure 4.2: *Average heart rate after normalization (discussed in Section 4.1) when a crash happens versus when no crash happens at an obstacle. Although the difference is certainly not significant, usually a higher heart rate at a crash is observable.*

Score Features. The intention of including features that involve the score was that a user who steadily increases his or her score is likely to continue to do so in the future. A rapid decrease of a user's score, resulting from a crash into an obstacle, might indicate that he or she is overwhelmed, and as a result is likely to crash into another obstacle.

Crash Features. We assumed that a user that crashed into the last couple of obstacles, and therefore might show signs of physical or mental overload, has a high chance of crashing in the following obstacle as well. As discussed in 5.2, we tested different window sizes and for the crash window, ten seconds turned out to perform the best. An obstacle appears around every two to three seconds. This supports our assumption that the performance on the last two to three obstacles matters the most when predicting a crash at the next obstacle.

4.2 Pre-Processing

Classifiers often make assumptions on the underlying data. For example, a *Gaussian Naive Bayes* (GNB) model expects the data to follow a normal distribution. If the data did not follow a normal distribution, we applied a *Box-Cox Power Transformation*. Furthermore, it is often necessary to scale the features to a given range. While the feature *%crashes* is constrained to the [0, 1] interval (see Figure 4.2), the feature *lin_regression_hr_slope* is potentially unbounded. Some classifiers such as the *Random Forest* are typically not affected by the scale of the features, others, however, such as the *k-Nearest Neighbors*, which uses a distance metric, are affected.

4.2.1 MinMaxScaler

We can scale each feature to a given range [0, 1] by applying a *MinMaxScaler*. Its robustness to very small standard deviations of features makes it the scaling of choice in this matter, even though zero mean and unit variance are not ensured as would be the case using standardization. In order to avoid introducing potential bias in our model, we fitted and transformed the training data set only, and used the resultant computed parameters to scale the test data set.

The transformation is given by:

$$\mathbf{X}_{scaled} = \frac{\mathbf{X} - \min(\mathbf{X})}{\max(\mathbf{X}) - \min(\mathbf{X})},$$

where $\mathbf{X} \in \mathbb{R}^{n \times d}$ is the feature matrix.

4.2.2 Box-Cox Power Transformation

The GNB classifier assumes its features to be normally distributed. Since not all of our features follow the said distribution naturally, we have to transform them accordingly. One way to

achieve this is to use a Box-Cox Power Transformation as proposed in [BC81]. The transformation is given by:

$$\mathbf{X}'_{:,j;\lambda} = \frac{\mathbf{X}^{\lambda}_{:,j} - 1}{\lambda} \quad \forall j \in \{0, 1, .., d\},$$

where $\mathbf{X} \in \mathbb{R}^{n \times d}$ is the feature matrix and λ is estimated using *Maximum Likelihood Estimation*.

We only applied this transformation for evaluating the GNB classifier.

4.3 Feature Selection

Reducing the number of features through removal of irrelevant or redundant features has the benefit to simplify the model, allow easier interpretations, while at the same time potentially improve the training time of the classifier as well as the performance by enhancing generalization, thus reducing the risk of overfitting.

Even though our data set only consists of 17 features, we applied feature selection in order to remove highly correlated features. For this, we calculated the correlations between all features on the training set and removed the ones that had a correlation factor higher than 0.9. In order to decide which of the two features of a highly correlated feature-pair to keep, we calculated the mean correlation of both features with all other features and kept the one with the lower mean value. These same features were removed in the test set.

The correlation between any two features of the entire data set can be seen in Figure 4.3. It is interesting to see that the mean of both the score and heart rate highly correlates with its respective maximum and minimum values. Furthermore, the standard deviation is highly correlated with the corresponding *max_minus_min* feature. This might be a consequence of the relatively small windows we are using in combination with rather steady, though slightly upward trending heart rate and score values from one obstacle to the next. As a result, both the maximum and minimum over the last couple of seconds usually increase along with the mean.

4.4 Classifiers

In this section, we introduce the different classifiers we used on our data. We give a high-level explanation together with advantages and disadvantages of each of the methods used. [1]

The standard input to a binary supervised learning algorithm is a training set D of some size n. Training examples are pairs of data points $\mathbf{x}_i \in \mathbb{R}^d$ with an associated class label $y_i \in \{-1, +1\}$, i.e. D = $\{(\mathbf{x}_i, y_i) : 1 \leq i \leq n\}$. We want to use the information in the training set to construct a classification rule F:

[1] Throughout this thesis we will refer to the i^{th} data vector as $\mathbf{x}_i$, while x_i refers to the i^{th} feature of the data vector $\mathbf{x}$

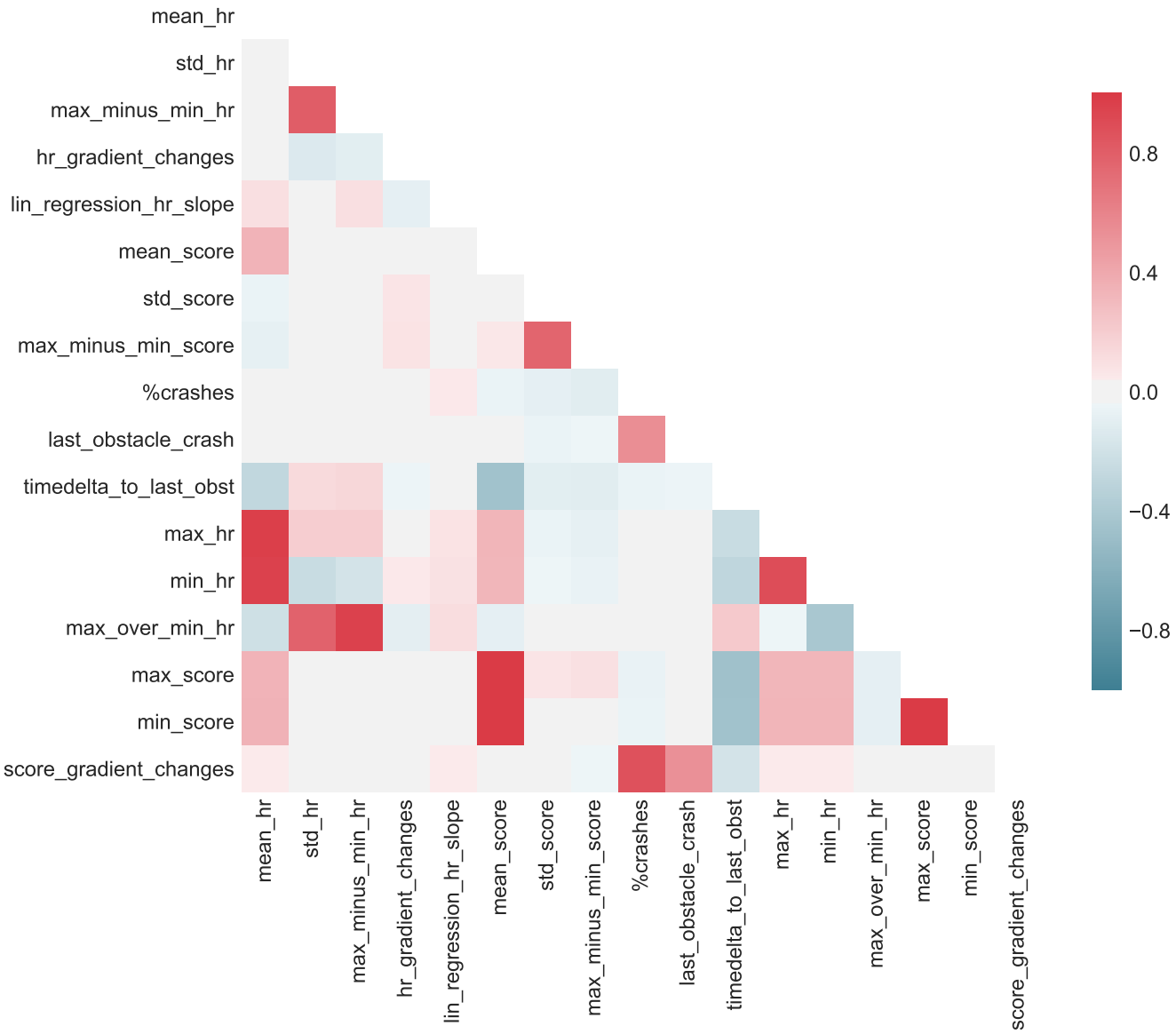


Figure 4.3: *Correlations between all initial features when using the entire data set.*

$$\underbrace{\left(\mathbb{R}^d \times \{-1,+1\}\right)^n}_{\text{Training data}} \rightarrow \underbrace{\left(\mathbb{R}^d \rightarrow \{-1,+1\}\right)^n}_{\text{Classifier } F}$$

This section is mainly based on the books *Pattern Recognition and Machine Learning* [Bis06] and *Machine Learning: A Probabilistic Perspective* [Mur12].

4.4.1 *k*-Nearest Neighbors

One of the simplest classifiers is *k*-Nearest Neighbors. This classifier simply predicts the majority of labels of the *k* nearest neighbors. For a data point $\mathbf{x}$, the classifier predicts:

$$y = \operatorname{sign}\left(\sum_{i=1}^{n} y_i \left[\mathbf{x}_i \text{ among } k \text{ nearest neighbors of } \mathbf{x}\right]\right).$$

The performance of the classifier is highly dependent on the value *k*, which can be found by cross-validation. The higher *k*, the smoother the decision boundary of the classifier. Thus, *k* is a hyperparameter and can be tuned. If *k* is too big, the classification can be overwhelmed by the majority class. This is a big problem if dealing with imbalanced data as we are doing in our study. To circumvent this problem, the neighbors can be weighted by their distance, such that closer neighbors have a bigger impact on the decision of the classifier.

The advantage of the k-Nearest Neighbors is that no training phase is required and multi-class problems can be handled easily. The main disadvantage is that its decision depends on the entire data set and is thus inefficient with a high amount of data.

4.4.2 Support Vector Machine

Classifiers such as a *Linear Classifier* try to find a weight vector $\mathbf{w} \in \mathbb{R}^d$ to compute a weighted sum of features, which is then compared to a threshold to produce a binary output:

$$F(x) = sign(\mathbf{w}^T\mathbf{x} + b) \in \{-1,+1\}.$$

The optimal $\mathbf{w}$ can be found by minimizing the objective:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \ell\left(\mathbf{w}; y_i, \mathbf{x}_i\right),$$

where ℓ is a loss function. An example of a loss function is the *Perceptron loss* $\ell_P\left(\mathbf{w}; y_i, \mathbf{x}_i\right) = \max\left(0, -y_i\mathbf{w}^T\mathbf{x}_i\right)$. The perceptron only updates the weights when the output produced is different from the true label.

The problem with the perceptron loss is that even though it will find a linear separator if the data is indeed linearly separable, it will choose any such $\mathbf{w}$. To have a hyperplane with a margin as

big as possible to its closest data points would be favorable. This is because the training data is only a subset of the entire data set. Even if the real data set was more spread out, the hyperplane should classify correctly, which makes the classification more robust.

The *Support Vector Machine* (SVM) achieves this goal of trying to find an optimal hyperplane that correctly classifies the data by some margin by using the *Hinge loss* $\ell_H(\mathbf{w}; \mathbf{x}, y) = \max\{0, 1 - y\mathbf{w}^T\mathbf{x}\}$, while still encouraging small weights by using an *l2* regularizer. The objective to be minimized thus becomes:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} \underbrace{\max\{0, 1 - y_i\mathbf{w}^T\mathbf{x}_i\}}_{\text{Hinge loss}} + \underbrace{\lambda\|\mathbf{w}\|_2^2}_{\text{L2-regularizer}} .$$

The size of the margin is defined by the so called *Support Vectors*, which are the points closest to the hyperplane on either side (labeled in Figure 4.4).

The advantage of using SVMs lies in their ability to model non-linear decision boundaries, their fair robustness against overfitting due to their regularization parameter, as well as their efficient optimization due to them being defined by a convex optimization problem. The biggest limitation of SVMs lies in the choice of their kernel.

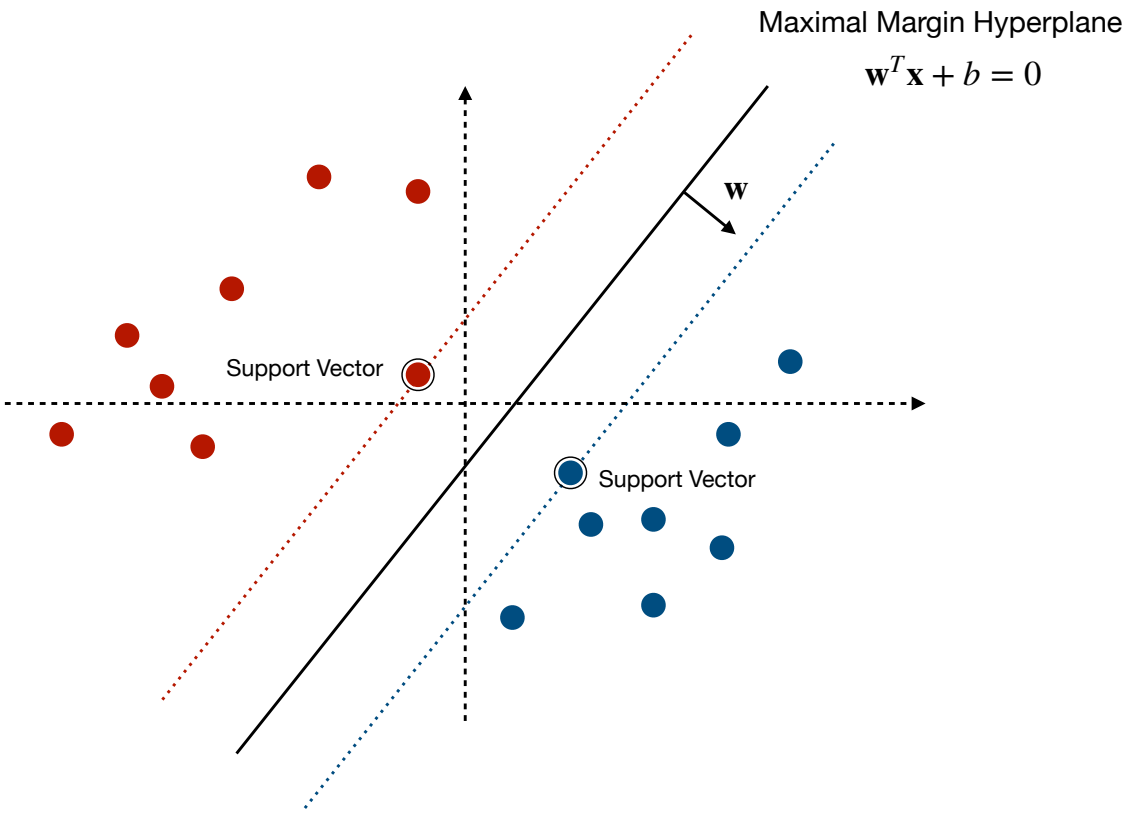


Figure 4.4: *This is an example of a maximal margin hyperplane of an SVM. The red and blue points are the negative and the positive labels, respectively. The separating hyperplane calculated by the SVM is depicted in black.*

4.4.3 Random Forest

A Random Forest classifier is an *ensemble method* that fits and evaluates multiple *Decision Trees* (see below) and averages their scores.

The goal of a Decision Tree is to create a tree-based structure that predicts the labels by learning decision rules inferred from the features. The deeper the tree, the more fine-grained the decision rules. See Figure 4.5 for an example of a decision tree generated on our training data.

On one hand, decision trees are easily interpretable and allow handling of noisy or incomplete data. On the other hand, however, they tend to overfit due to them being able to learn on the training data up to a point of high granularity.

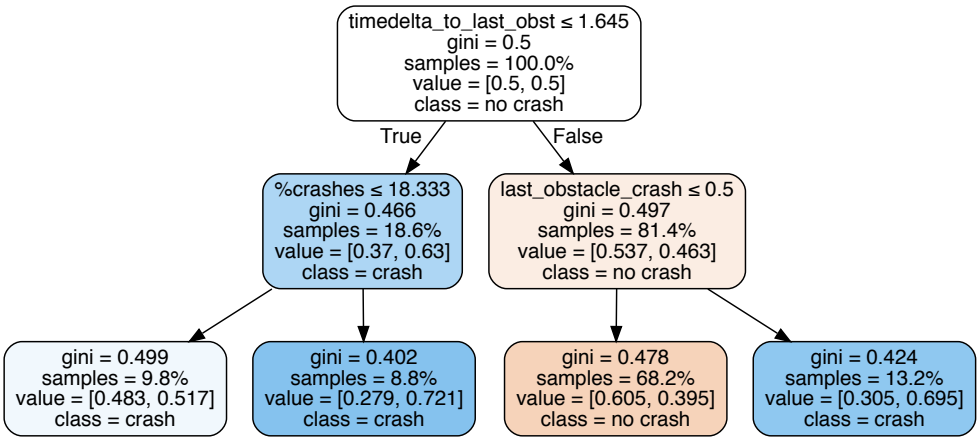


Figure 4.5: *The first three layers of our Decision Tree are shown. Example on how to read the tree: Assume that feature %timedelta_to_last_obst of a data point* $\mathbf{x}$ *is bigger than 1.645, and feature last_obstacle_crash is bigger than 0.5. We arrive at the bottom right node. We then know that 13.2 % of all the training data points end up here, and that data point* $\mathbf{x}$ *gets a positive label assigned with a confidence of 69.5 %.*

4.4.4 Gaussian Naive Bayes

Naive Bayes (NB) models work on the assumption that features are conditionally independent given a label y, that is:

$$P(\mathbf{x}|y) = \prod_{j=1}^{d} P\left(x_j|y\right),$$

where d is the number of features, $\mathbf{x} \in \mathbb{R}^d$ is a data point, and y is the label.

Knowing the distribution $P\left(x_j|y\right)$ of the features, the probability of label y given a data point $\mathbf{x}$ can then be calculated as:

$$P(y|\mathbf{x}) = \frac{P(y)\Pi_{j=1}^{d} P\left(x_j|y\right)}{P(\mathbf{x})}.$$

By noting that $P\left(\mathbf{x}\right)$ is independent of y and can thus be omitted, a classifier can predict the most likely label, that is:

$$\hat{y} = \arg\max_{y} \hat{P}(y) \prod_{j=1}^{d} \hat{P}\left(x_j|y\right),$$

where the probabilities $\hat{P}$ are estimated from the training set.

In case of a *Gaussian* NB model, the features are assumed to be gaussian distributed, that is:

$$\hat{P}\left(x_j|y\right) = \frac{1}{\sqrt{2\pi\hat{\sigma}_y^2}} \exp\left(-\frac{\left(x_j - \hat{\mu}_y\right)^2}{2\hat{\sigma}_y^2}\right),$$

where $\hat{\sigma}_y$ and $\hat{\mu}_y$ are estimated using *Maximum Likelihood Estimation*.

Since GNB makes the assumption of gaussian distributed features, but not all of our features follow one, we applied a Box-Cox transformation beforehand as explained in Section 4.2.

Even though the conditional independence assumptions rarely hold, NB models perform surprisingly well in practice, are easy to implement and do not have any hyperparameters to tune.

4.4.5 Long Short-Term Memory Network

Recurrent Neural Networks (RNNs) are neural networks that have cyclic connections between their nodes (Figure 4.6), which allows them to retain information. This is useful upon working with time-series when we explicitly want to connect previously computed information to a present task. While RNNs are indeed capable of handling long-term dependencies in theory, they do not seem to be able to learn them in practice, as discussed in [Hoc91] and [PMB13].

As a solution, Hochreiter and Schmidhuber [HS97] proposed a variation of RNNs with so called *Long Short-Term Memory* (LSTM) units, which explicitly introduce a memory unit called a cell, and which are capable of learning long-term dependencies.

Feature Matrix. Compared to our two-dimensional feature matrix discussed in Section 4.1, the input to an LSTM layer needs to be three-dimensional. The three dimensions are:

- Samples: One sequence is one sample.
- Time Steps: One time step is one data point of observation in the sample.
- Features: One feature is one observation at a time step.

In our case, the number of samples corresponds to the number of log files, and the number of time steps corresponds to the maximum number of obstacles in a log file. An LSTM network

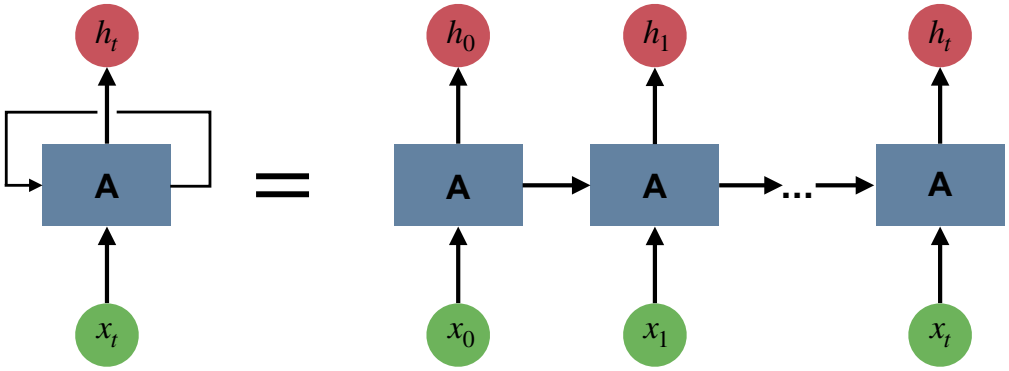


Figure 4.6: *The structure of an RNN and its unrolled representation.* **A** *is part of a neural network,* x_t *is the networks input and* h_t *its output. An RNN can be thought of as multiple copies of a single network, each passing information to its successor. The figure is an adaption of the one shown in [Ola15].*

which is trained with multiple batches expects all its sequences to be of the same length. This can be achieved by cutting all samples to the same minimal length, or by padding all samples with zeros to the length of the longest sample. We opted for the latter. The padding will later be masked out by the LSTM network. This has the advantage that we do not loose any information. After the padding, the two dimensional sequences can be reshaped into a three dimensional matrix.

5

Results

In this chapter, we first describe our experimental setup in Section 5.1, where we explain how we tuned our hyperparameters and our LSTM network, elaborate on all statistical measures we used to evaluate and compare our models, and give details about our computational environment. In Section 5.2, we explain how we tested different window sizes, and in Section 5.3, we discuss our results.

5.1 Experimental Setup

5.1.1 Hyperparameter Tuning

Hyperparameters are the parameters that usually have to be fixed before training and testing the model. Optimizing those is generally a very important step in the model building process. In the following, we list the parameters' range of the classical ML models and explain the method we used to find the optimal one. We then explain how we proceeded to find the best parameters for our LSTM network.

For all classifiers, we have used class weights for handling class imbalance, which penalizes misclassification of the minority class (*crash*) more than the majority class (*no crash*). We have chosen the weights such that they are inversely proportional to the size of the class.

Grid Search. A common approach to do hyperparameter tuning is to use *grid search*, where a list of parameters and the corresponding values are exhaustively searched to get the best model, that is every possible combination of the given list of parameters is evaluated. While it is very simple to use, it has some major drawbacks: It scales exponentially with the number of hyperparameters and often too many dimensions that do not matter are explored, which leads to a bad running time and wasted resources. See [BB12] for an in-depth discussion.

Randomized Search. Bergstra and Bengio [BB12] showed that *randomized search* often is not only more efficient but also finds better models than grid search. As an example, assume having two parameters and aiming to evaluate nine values in total to find the optimal parameter combination. Grid search will effectively only evaluate three values for each parameter because it is constrained to search the best parameters in a grid-like pattern. Random search, on the other hand, can potentially search for nine values for each parameter, thus increasing the chance of finding a better combination than grid search. This is especially beneficial if the effective dimensionality is lower than the number of hyperparameters, for example if only one or two parameters out of many are truly indicative, which is often the case. Usually, random search will sample the important dimensions more often than grid search.

In Table 5.1, the range we tested of each hyperparameter is listed. Note that the Gaussian Naive Bayes classifier does not have any hyperparameters to tune. For all parameters in the Random Forest classifier that we did not tune, we used the following default values: *Gini Impurity* to measure the quality of a split, a minimum number of two samples required to split an internal node, and no maximal depth of the tree.

Table 5.1: *All hyperparameters that were tuned, together with their ranges, are shown for each classifier.*

Classifier	Hyperparameters
k-Nearest Neighbors	$k \in [1, 1000]$, metric $\in$ [minkowski, euclidean, manhattan]
SVM	C $\in [2^{-5}, 2^{5}]$, $\gamma \in [2^{-15}, 2^{3}]$, kernel $\in$ [rbf, sigmoid]
Random Forest	minimum leaf size $\in [1, 50]$, number of trees $\in [1, 128]$
Gaussian Naive Bayes	no tuning parameters

Long Short-Term Memory. Optimizing hyperparameters of neural networks can be very challenging since there are many parameters to tune. Moreover, cross-validation is often not feasible because training a neural network can be computationally expensive.

The general method recommended in [Ben12] is to add layers and units until the network starts to overfit the training set. Like that, the network is ensured to be powerful and wide enough for the task provided. According to [GBD92], we then have a high variance but a low bias. *Dropout*, as discussed in [SHK+14], as well as other regularization methods can be added to reduce overfitting. Table 5.2 summarizes the approach of tuning the main parameters of a neural network ([Ben12]), which we followed.

In order to get the final performance scores of our network, we randomly split the log files into a training set of 14 and a test set of 3 files, fitted the network to the training set and calculated the performance on the test set. We repeated this 20 times and took the average over the individual runs.

5.1.2 Statistical Measures

In this section, we explain all metrics that we used to evaluate our models. We only considered binary classification tasks. The discussion is mainly based on [JCDLT13].

Table 5.2: *Summary of how we optimized the main hyperparameters of our neural network, as suggested in [Ben12].*

Hyperparameter	Approach
Learning rate	Start with high learning rate (close to 1), and if the training criterion diverges, try again with three times smaller learning rate. Repeat until no divergence is observed anymore.
Number of iterations	Can be optimized using early stopping, that is a minimum number of training examples without improvement of a given metric stops the training phase. This is one way of avoiding overfitting.
Hidden units	Units that are too large usually do not affect performance much. Using the same size for all layers generally works fine.
Neuron non-linearity	Since sigmoid units have optimization problems (*Vanishing Gradient Problem*), rectifier and tanh units should be used.

For evaluating the models, we used *10-fold cross-validation*, which means that the model trains on each set of 9 folds and predicts on the one that was left out. See Figure 5.1 for an illustration. There are two approaches to average the scores over the folds: Micro-and macro averaging. In micro averaging, all true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN), as displayed in the confusion matrix in Table 5.3, are summed up and the metric is evaluated accordingly. In macro averaging, in each of the 10 cross-validation rounds the metric is evaluated on the test fold, and at the end the metric is averaged across all rounds. We used the latter approach, as it treats all classes equally, and a standard deviation could be computed, which allows for easier interpretation of the results.

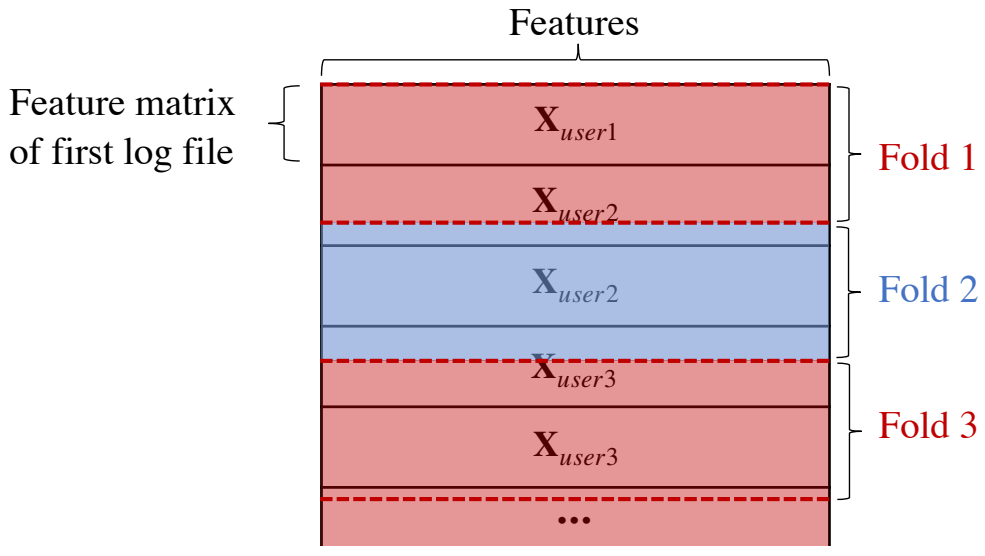


Figure 5.1: *An illustration of part of our feature matrix with 10-fold cross-validation is shown. We generate one feature matrix for each log file. A user has either one or two log files. The training folds are depicted in red, while the test fold is depicted in blue.*

Table 5.3: *The confusion matrix for a two-class classification problem.*

		True value	
		Positive	Negative
Predicted value	Positive	TP	FP
	Negative	FN	TN

One simple metric often used is *accuracy*, but this metric can be misleading when dealing with imbalanced data and we therefore do not make use of it. By always predicting the majority class *no crash*, we would get an accuracy of 0.8, which is higher than the baseline of 0.5, even though the underlying model might not possess a higher predictive power than a model with a lower accuracy.[1]

Below we explain the metrics that we used to evaluate our models.

Recall. Also called *sensitivity* or *true positive rate* (TPR), the *recall* is the fraction of crashes that were predicted as such.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \text{TPR}$$

Precision. The *precision* is the fraction of predicted crashes that were indeed crashes.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Specificity. The *specificity* is the ratio of non-crashes that were correctly predicted as such.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

F_1 Score. The F_1 *score* is the harmonic mean of recall and precision. The problem with looking at either of those only is that it can be very misleading. A high recall value can be achieved by always predicting the positive label, while a perfect precision can be obtained by predicting the negative label only. By considering both the recall and precision, it is a more realistic measure of the classifier's performance.

$$\text{F}_1 = \frac{2}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}$$

[1] This phenomenon is also called *Accuracy Paradox*, see [Zhu07].

AUC. The *receiver operating characteristic* (ROC) shows the TPR as a function of the *false positive rate*, $\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP+TN}}$, at various thresholds. From this, the *area under the curve* (AUC) can be calculated. The benefits of this metric are that it is not affected by class imbalance, and it shows a trade off between TPR and FPR. Using the *Youden's J statistic*, described in the next paragraph, we can find the threshold that optimizes the recall and the FPR. Note that an AUC score of 0.5 corresponds to a random guess.

Youden's J Statistic. Youden's J statistic was proposed in [You50]. It is a metric that summarizes both recall and specificity, and it is closely related to the ROC curve, as it is an index defined for all points on a ROC curve. Informally, it can be described as the vertical distance from the random guess line to the curve, as seen in Figure 5.2.

$$\mathrm{J} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}} + \frac{\mathrm{TN}}{\mathrm{TN}+\mathrm{FP}} - 1 = \mathrm{Recall} + \mathrm{Specificity} - 1$$

Even though our models often obtained a better AUC score than for a random guess, they only got a recall of 0 and precision of 1 since the model always predicted that no crash happens. This was because of the default threshold of 0.5. To solve this problem, we used the maximal Youden's index to select for the optimal decision threshold, which in turn led to a higher F_1 score.

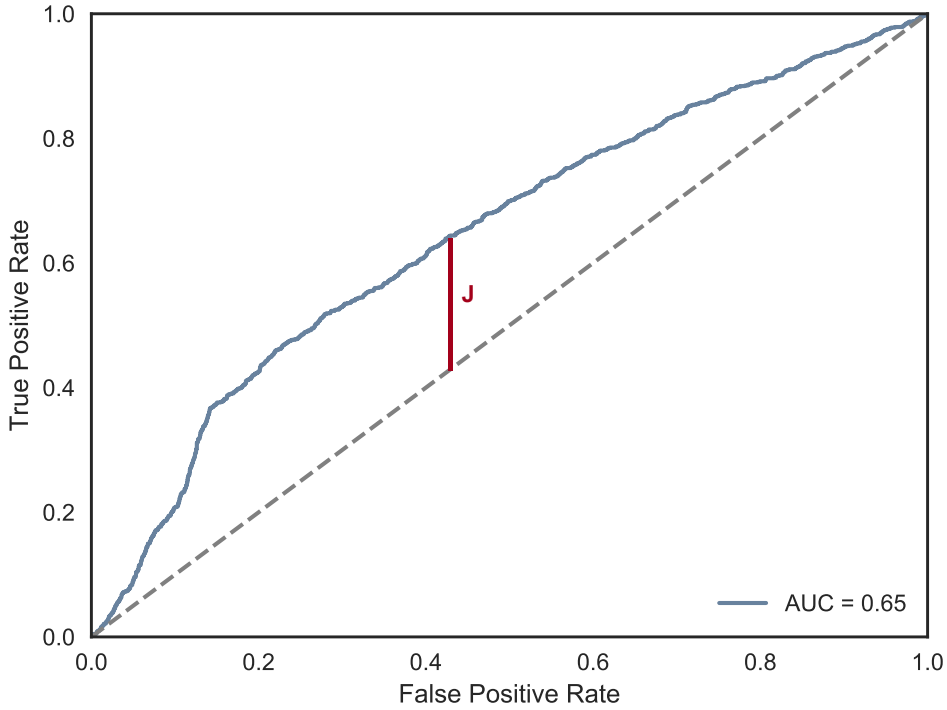


Figure 5.2: *The maximal Youden's index of a ROC curve is illustrated. The random guess is depicted as a dotted grey line. Note that the index is defined for every point on the ROC curve.*

5.1.3 Computing Environment

We used *Python* 3.6 for the implementation. The main libraries we used are *scikit*[2] version 0.19 for implementing the classical ML models, *Keras*[3] version 2.1 for implementing the LSTM network, and *matplotlib*[4] 2.2 to generate plots. Our project can be found on *GitHub*[5].

Hyperparameter tuning and fitting the LSTM networks was done on the *Euler*[6] cluster of *ETH Zurich*. It contains roughly 3,000 high performance nodes with a total of 44,000 cores.

5.2 Window Sizes

As explained in Section 4.1, we used different window sizes to extract our features from the log files. In this section, we show the approach and results of testing the different windows.

We first performed hyperparameter tuning on the *k*-Nearest Neighbors classifier with windows of size *crash window* $= 5$ seconds, *default window* $= 10$ seconds and *gradient window* $= 10$ seconds. As a next step, we used this tuned classifier and plotted the AUC score for different window combinations by fixing one window to 10 seconds and varying the remaining two. See Figure 5.3 for the results when keeping the *default window* fixed.

We observe that there is a clear trend of smaller windows performing better than bigger windows if the *default window* is fixed. The same trend, though slightly less pronounced, can be observed when keeping the *gradient window* constant. Keeping the *crash window* constant while varying the other two windows does not make a significant difference in performance, except when using either a *gradient* or *default window* of 60 seconds. Then the performance is poorer. One might conclude from this that the performance of the user at a point too far in the past is not relevant for his or her current performance. A *gradient window* of 20 or 30 seconds, and a *crash window* of 20 seconds or smaller perform best. This supports our assumption discussed in Section 4.1 that it would be more indicative whether the user crashed into the very recent obstacles rather than whether he had crashed into an obstacle 60 seconds ago.

Taking all these considerations into account, we decided to use window sizes 10, 20 and 30 seconds for the *crash window*, *default window* and *gradient window*, respectively. We then tuned the hyperparameters of all classifiers with those windows.

5.3 Performance Analysis

In this section, we analyze the performance of both the classical ML models and the LSTM network when applying 10-fold cross-validation. Furthermore, we discuss the performance of the classifiers in case a user is left out completely in the training phase.

[2]http://scikit-learn.org/stable. Retrieved 21.08.2018.
[3]https://keras.io. Retrieved 21.08.2018.
[4]https://matplotlib.org. Retrieved 21.08.2018.
[5]https://github.com/bastianmorath. Retrieved on 03.09.2018.
[6]https://scicomp.ethz.ch/wiki/Euler. Retrieved 21.08.2018.

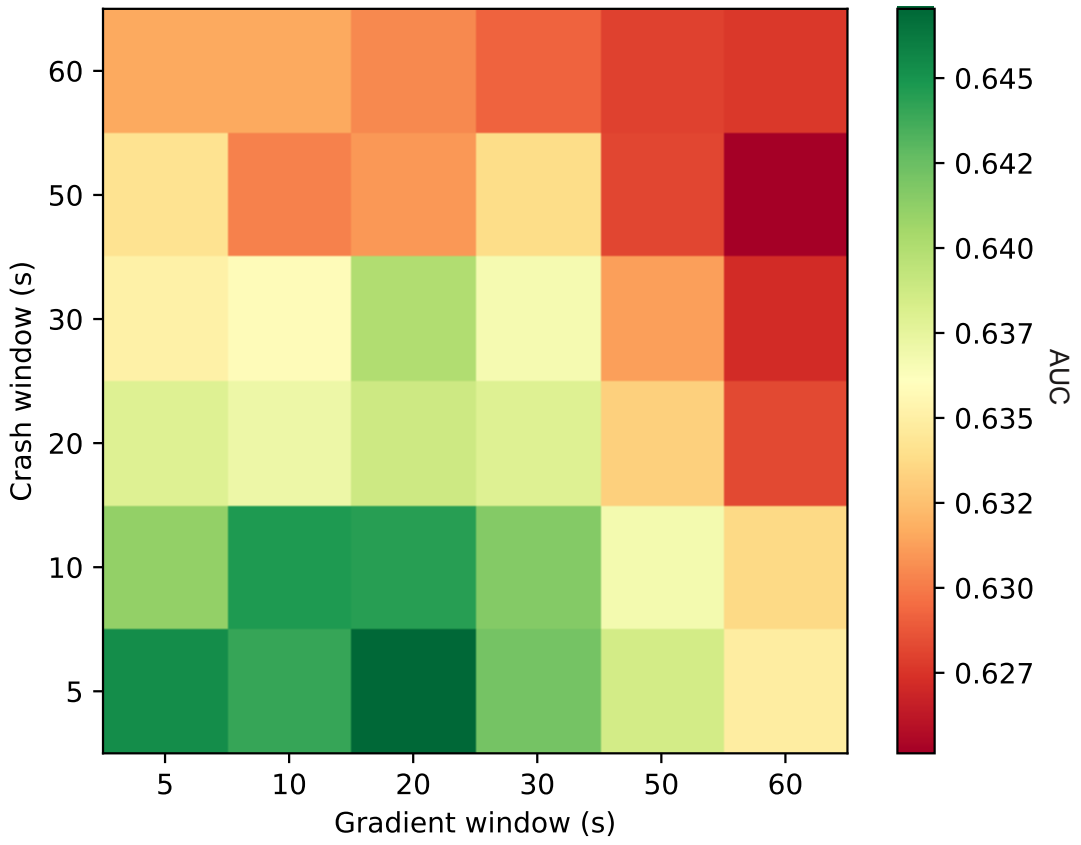


Figure 5.3: *The AUC score for the k-Nearest Neighbors classifier when keeping a constant default window of size 10 seconds and varying the other two windows is shown.*

5.3.1 Classical Machine Learning Models

10-Fold Cross-Validation. Table 5.4 shows the performance of all classical ML models we tested after hyperparameter tuning and when using 10-fold cross-validation. The ROC curves of the classifiers are shown in Figure 5.4. Remember that the recall, precision, as well as the F_1 score were evaluated by calculating the optimal decision threshold with the help of Youden's J index as explained in Section 5.1.2.

Table 5.4: *Performance overview of the classical ML models when using 10-fold cross-validation. The classifiers are listed in decreasing order of their AUC score. The standard deviation is indicated in brackets and was calculated using macro averaging, that is in each of the 10 cross-validation rounds, the metric was evaluated on the test fold, and at the end the metric was averaged across all rounds.*

Classifier	AUC	Recall	Specificity	Precision	F_1
Random Forest	0.648 (±0.07)	0.560 (±0.13)	0.691 (±0.13)	0.297 (±0.08)	0.374 (±0.13)
k-Nearest Neighbors	0.645 (±0.06)	0.631 (±0.20)	0.622 (±0.23)	0.304 (±0.10)	0.378 (±0.08)
SVM	0.641 (±0.06)	0.491 (±0.12)	0.763 (±0.09)	0.322 (±0.07)	0.382 (±0.08)
Naive Bayes	0.634 (±0.07)	0.497 (±0.16)	0.739 (±0.08)	0.309 (±0.08)	0.356 (± 0.09)

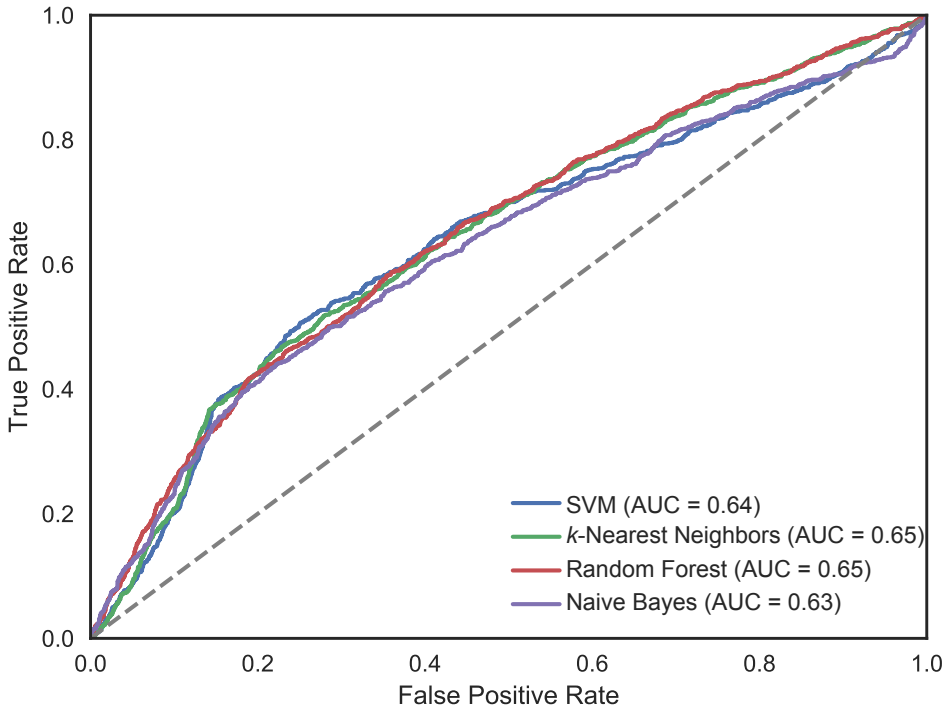


Figure 5.4: *The ROC curves of all classical models are shown. The random guess is depicted as a dotted grey line.*

The first thing to note is that no classifier's AUC score exceeds that of the others significantly. Even though they all perform better than random guessing, they still have a rather poor perfor-

mance. Since there is no classifier of which the ROC curve is strictly above the other classifiers' curves for every threshold, there is no overall superior classifier.

The mean recall is highest for the *k*-Nearest Neighbors classifier, and lowest for the SVM classifier, however there is a relatively big standard deviation across the folds. The precision scores among the classifiers are closer together than the recall scores, with a trend of SVM having the highest precision, and the Random Forest classifier having the lowest.

We therefore suggest to use the *k*-Nearest Neighbors classifier when the focus lies on a high recall, which means that we want to correctly predict as many crashes as possible. On the other hand, if predicting a non-crash as a crash is more severe than falsely predicting a crash as a non-crash, we suggest using SVM, which tends to have the highest precision score.

Our best explanation for the poor performance is that the set of variables measured in our log files was very limited. Basically, all we got was heart rate and score data, along with some information about crashes. Moreover, the fact whether or not the user crashed did not affect the heart rates significantly (see Section 4.1). Thus, heart rate can be considered a rather weak feature in the first place.

Moreover, as discussed in Section 3.2, detailed data on neither the difficulty levels nor the missing of crystals was available, which could be another explanation for the poor performance of our models.

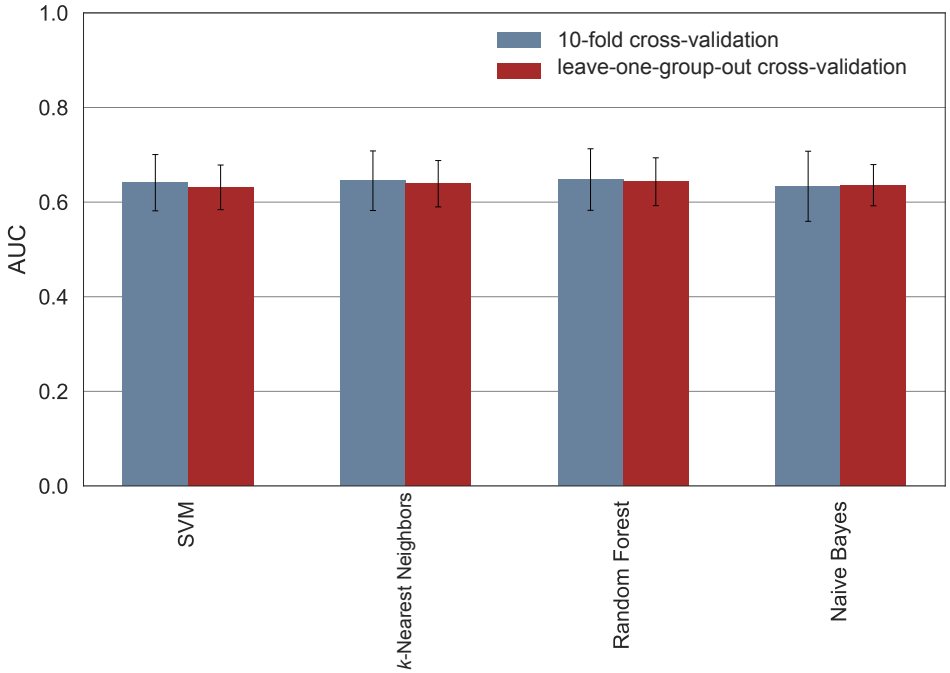


Figure 5.5: *Blue bars show the mean AUC score when applying 10-fold cross-validation, red bars when applying leave-one-group-out cross-validation.*

Leave One Group Out. A problem that can occur with 10-fold cross-validation is that when splitting the feature matrix, for each left-out fold (which usually contains the feature matrix of two or three log files), there are normally at least some of the corresponding user feature matrices present in the training data as well (illustrated in Figure 5.1). The model has thus

already been trained on some of the user data, which can influence the classifiers decision. To test our hypothesis, we evaluated the models with leave-one-group-out cross-validation, where the models are trained on all but one user, and then tested on the left-out user. The results are shown in Figure 5.5. The two different cross-validation methods show virtually no difference in performance. We therefore arrived at the conclusion that our classifiers predict also equally well upon encountering data of a completely new user.

5.3.2 Long Short-Term Memory

As described in Section 5.1.1, we approached the tuning of the neural network by first building a network that overfitted the data and subsequently adding regularization. After getting a network that we considered powerful enough, we added dropout layers with a dropout rate of 20 %. Our final network is depicted in Figure 5.7. It has one LSTM layer with a tanh-activation function and two Dense layers with a relu activation. All layers have 96 units. We use the *adam*-optimizer with a learning rate of $3 * 10^{-4}$, a decay of $4 * 10^{-6}$, and *categorical crossentropy* as a loss function. We trained the network for 130 epochs and accounted for the class imbalance by weighting the loss function accordingly. We have chosen the weights such that they are inversely proportional to the size of the class.

In order to find the best hyperparameters, we trained the network on a subset of the data, and validated it on different, previously unseen data. Figure 5.6 shows the training and validation AUC over the 130 epochs. The fact that the validation AUC is bigger than the training AUC is a result of the dropout only being applied to the training phase, thus weakening the power of the network. When evaluating the validation set, all nodes are added to the network again, which naturally makes the network more powerful. We also observe that the validation score peaks after around 70 epochs and flattens after 120 epochs, which can be an indication that the network is neither over- nor underfitted at this point. We decided to stop training after 130 epochs to prevent over-fitting.

Table 5.5: *The different scores of the final LSTM network are shown. We trained and evaluated the network 20 times and took the average over all rounds. For comparison, the scores of the Random Forest classifier are shown as well.*

	AUC	**Recall**	**Specificity**	**Precision**	**F_1**
LSTM	0.647 ($\pm$0.03)	0.572 ($\pm$0.09)	0.688 ($\pm$0.08)	0.325 ($\pm$0.05)	0.394 ($\pm$0.04)
Random Forest	0.648 ($\pm$0.07)	0.560 ($\pm$0.13)	0.691 ($\pm$0.13)	0.297 ($\pm$0.08)	0.374 ($\pm$0.13)

Our final scores of the LSTM network together with the Random Forest classifier are shown in Table 5.5. We did not obtain a better AUC score than the classical models, which can be an indication that there are no long-term dependencies. The precision score is close to the SVM classifier, while the recall and specificity are very similar to those of the Random Forest classifier. In addition to the remarks in Section 5.3.1, a possible conclusion is that the user's performance at the beginning of the game has no big effect on his or her performance later in the game. We also had to work with a relatively small amount of data (around 6,000 data points), which is relatively little data for training and testing a neural network.

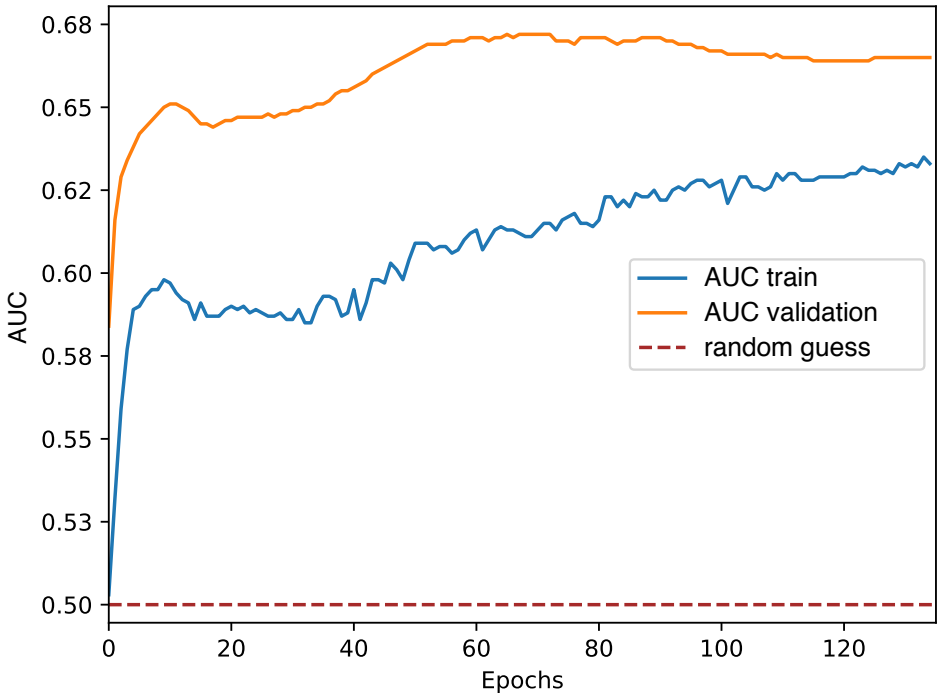


Figure 5.6: *The AUC score of one round of training is shown. The training score increases steadily, while the validation score does not increase anymore after 130 epochs, after which we stopped training to prevent overfitting. We did not include any validation set when calculating the final performance score. Instead we only had one training and one test set per round, as explained previously in Section 5.1.1.*

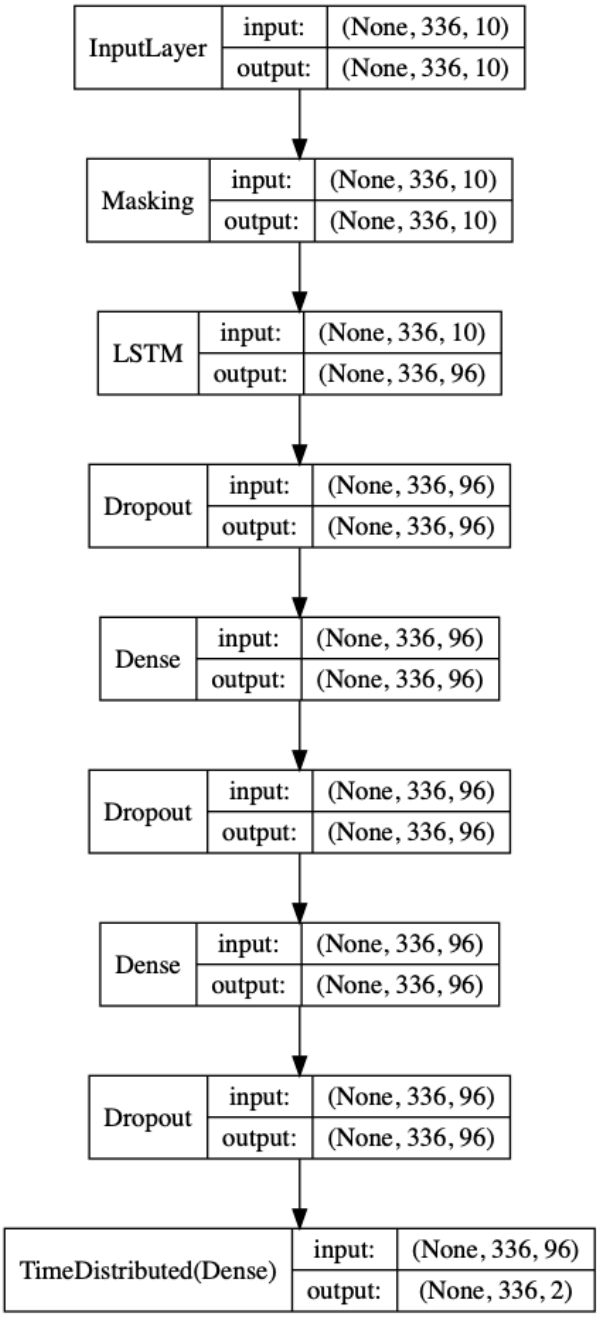


Figure 5.7: *Visualization of our final LSTM network. As explained in Section 4.4.5, the masking removes the padding. The numbers in parenthesis are the input and output dimensions, respectively. A dimension of None means that any size is being accepted.*

6

Conclusion

In this work, we created an ML model that anticipates the user's in-game performance in the game Plunder Planet by predicting whether or not the user is going to crash into the next obstacle. This enables the automatic adjustment of the difficulty level to the user's physical and emotional state, ensuring that the player is neither over- nor under-challenged, thus offering a better fitness experience to the player. We achieved an AUC score of 0.64, which is better than a random guess. We tested both classical ML models (SVM, k-Nearest Neighbor, Random Forest and Naive Bayes classifier), and a recurrent neural network with LSTM units. All models showed virtually the same performance in terms of their AUC score, but different trade-offs between precision and recall were observed. We suggest to use the Random Forest classifier when the focus lies on a high recall, meaning that as many crashes as possible should be predicted correctly. On the other hand, if predicting a non-crash as a crash is more severe than falsely predicting a crash as a non-crash, we suggest using SVM or an LSTM network, since those two tend to yield the highest precision scores.

7

Future Work

We think that our models could be greatly improved by increasing the set of measured variables during a game session. Some ideas are listed below.

Interaction Data. One could focus on extracting more fine-grained difficulty levels, which could improve our model a lot, since we assume that crashing at a low level is a much better indication of an overwhelmed user than crashing at the highest possible level. Moreover, recording data on missing crystals has further potential for improving our model. Furthermore, recording information on the FBMC buttons could enhance our model, for example by tracking which buttons were pressed at what time. If the interval between the previous obstacles and their corresponding button press is becoming smaller and smaller, the user might be too late at pressing the button the next time, and as a result he or she will crash. Another hint that the user is overwhelmed can be when the user not only presses the button too late, but also the wrong button all together.

Biosensor Data. Future work should focus on collecting Biosensor data other than the heart rate. Examples are skin conductance to detect the sweating level of the player, skin temperature, heart rate variability, and applying *electromyography* (EMG) to detect muscle contractions.

Performance Metrics. Instead of predicting obstacle crashes, one could predict the heart rate, for example by using regression. We assume that this allows to predict when a user is just about to be physically overwhelmed, and in order to prevent this, the difficulties of the game could already be lowered at an earlier stage.

Bibliography

[Bač18] Boris Bačić. Towards the next generation of exergames: Flexible and personalised assessment-based identification of tennis swings. *arXiv preprint arXiv:1804.06948*, 2018.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[BC81] GEP Box and DR Cox. An analysis of transformations revisited, rebutted. Technical report, Wisconsin Univ-Madison Mathematics Research Center, 1981.

[BC04] Emily Brown and Paul Cairns. A grounded investigation of game immersion. In *CHI'04 extended abstracts on Human factors in computing systems*, pages 1297–1300. ACM, 2004.

[Ben12] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[BQJB90] Gary G Berntson, Karen S Quigley, Jaye F Jang, and Sarah T Boysen. An approach to artifact identification: Application to heart period data. *Psychophysiology*, 27(5):586–598, 1990.

[CA94] Albert T Corbett and John R Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction*, 4(4):253–278, 1994.

[GBD92] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural computation*, 4(1):1–58, 1992.

[Hoc91] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1), 1991.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[JCDLT13] László A Jeni, Jeffrey F Cohn, and Fernando De La Torre. Facing imbalanced data–recommendations for the use of performance metrics. In *Affective Computing and Intelligent Interaction (ACII), 2013 Humaine Association Conference on*, pages 245–251. IEEE, 2013.

[KKSG17] Tanja Käser, Severin Klingler, Alexander G Schwing, and Markus Gross. Dynamic bayesian networks for student modeling. *IEEE Transactions on Learning Technologies*, 10(4):450–462, 2017.

[M+75] Gordon E Moore et al. Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[MG08] Michelle McPartland and Marcus Gallagher. Learning to be a bot: Reinforcement learning in shooter games. In *AIIDE*, 2008.

[MN17] Anna Lisa Martin-Niedecken. Exploring spatial experiences of children and young adolescents while playing the dual flow-based fitness game" plunder planet. In *Proceedings of the International Conference on Computer-Human Interaction Research and Application (CHIRA'17)*, 2017.

[MN18] Anna Lisa Martin-Niedecken. Designing for bodily interplay: engaging with the adaptive social exertion game plunder planet. In *Proceedings of the 17th ACM Conference on Interaction Design and Children*, pages 19–30. ACM, 2018.

[MNG16] Anna Lisa Martin-Niedecken and Ulrich Götz. Design and evaluation of a dynamically adaptive fitness game environment for children and young adolescents. In *Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts*, pages 205–212. ACM, 2016.

[MNG17] Anna Lisa Martin-Niedecken and Ulrich Götz. Go with the dual flow: Evaluating the psychophysiological adaptive fitness game environment "plunder planet". In *Joint International Conference on Serious Games*, pages 32–43. Springer, 2017.

[Moo65] Gordon E Moore. Cramming more components onto integrated circuits. electronics 38 (8): 114–117. 1965.

[Mur12] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[Ola15] Christopher Olah. Understanding lstm networks, 2015.

[PJCK09] Philip I Pavlik Jr, Hao Cen, and Kenneth R Koedinger. Performance factors analysis–a new alternative to knowledge tracing. *Online Submission*, 2009.

[PMB13] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[PS18] Paula Pullen and William Seffens. Machine learning gesture analysis of yoga for exergame development. *IET Cyber-Physical Systems: Theory & Applications*, 2018.

[SBM05] Kenneth O Stanley, Bobby D Bryant, and Risto Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE transactions on evolutionary computation*, 9(6):653–668, 2005.

[SC11] Amanda E Staiano and Sandra L Calvert. Exergames for physical education courses: Physical, social, and cognitive benefits. *Child development perspectives*, 5(2):93–98, 2011.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[SHM07] Jeff Sinclair, Philip Hingston, and Martin Masek. Considerations for the design of exergames. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 289–295. ACM, 2007.

[WW11] David Weibel and Bartholomäus Wissmath. Immersion in computer games: The role of spatial presence and flow. *International Journal of Computer Games Technology*, 2011:6, 2011.

[You50] William J Youden. Index for rating diagnostic tests. *Cancer*, 3(1):32–35, 1950.

[Zhu07] Xingquan Zhu. *Knowledge Discovery and Data Mining: Challenges and Realities: Challenges and Realities*. Igi Global, 2007.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Performance Predictor: Machine learning based prediction of user performance in the game "Plunder Planet"

Authored by (in block letters):

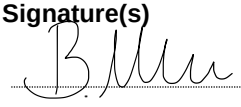
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Morath	Bastian

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 03.09.2018	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.