CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Multi-objective optimization by Machine Learning

Master's thesis in Computer science and engineering

Hampus Hagstrand

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Multi-objective optimization by Machine Learning

Hampus Hagstrand

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Multi-objective optimization by Machine Learning

Hampus Hagstrand

Supervisor: Carl-Johan Seger
Examiner: Jean-Philippe Bernardy

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Multi-objective optimization by Machine Learning

Hampus Hagstrand
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Solving multi-objective optimization with machine learning can significantly improve various fields, such as multi-junction traffic management or stock portfolio optimization. These are problems that can have a large amount of relevant and irrelevant data. This thesis targets one such problem area, focusing on multi-objective optimization in trot horse harness racing, specifically the V75.

A large part of the project was data-related, such as data collection, preprocessing, and engineering. The predicting part is divided into two parts single race prediction and system predictions. The single-race prediction utilizes the large amount of data collected to train a neural network to predict the percentage of the horse finishing behind the winner. The system prediction uses the result from the neural network to pick a system. During this process, a greedy algorithm selects more horses in the races that the machine learning deems close and fewer that it deems one-sided.

The performance evaluation showed that the single race predicting performed on par with the more advanced baseline and showed clear signs of finding a pattern between the data and the finishing result. The system prediction found some accuracy but did not surpass the odds baseline.

Keywords: Machine Learning, Artificial Intelligence, Multi-objective Optimization, Horse Racing, V75

Acknowledgements

Firstly, I would like to begin by expressing my sincere gratitude to Carl-Johan Seger, who has been an outstanding supervisor. His guidance, expertise, and patience were instrumental in completing this thesis.

Secondly, I would like to thank Jean-Philippe Bernardy for his insightful feedback on the half-time report. His critiques significantly improved the quality of my work.

Thirdly, I want to extend my appreciation to Magnus Edvardsson, Per Klevmarken, and Anton Hålldén. They introduced me to the world of V75, a sport that formed the foundation of this thesis. Their wisdom, expertise, and knowledge about the game played a crucial role in shaping this work.

Lastly, I would also like to acknowledge that Grammarly was used for spell and grammar checking, ensuring the readability and correctness of this thesis.

Hampus Hagstrand, Gothenburg, 2023-06-27

Contents

List of Figures

1

Introduction

This thesis aims at working on optimization under constraints problems given a large amount of both relevant and irrelevant data. An example of a problem like this would be traffic management. A city has numerous junctions (events), and each intersection has different traffic volume, location, speed, and much more (data). Different traffic patterns and control strategies can be used at each junction to affect its performance (actions). However, the city has a restricted budget and can not do the best actions at each intersection (limiting factor). Another example of this type of problem is portfolio optimization. In this case, there are multiple investment opportunities (events). Each stock/company has different attributes, such as price, P/E, employees, and much more(data), and the limiting factor is budget. A final example, and one that we will use as a driving example in this thesis, is V75 betting, where deciding what horses to pick in each race is the events, previous races are the data, and a budget is the limiting factor.

A more general explanation of this type of problem is that there are multiple events independent of each other and several possible actions at each event. Every event has the same data points, but the data itself differ. The only commonality between the events is their effect on the limiting factors such as a budget. Supervised machine learning(ML) can be used to determine the most optimal actions at each event, disregarding the limiting factor. Because of this, the ML needs to return a metric that quantifies how crucial a specific action is so that at a later stage, an algorithm can determine if the action is necessary enough to perform and affect the limiting factor.

There is a wide range of this type of problem that varies in importance. Solving these problems can be essential for multiple reasons. Solving the traffic management problem can significantly impact and improve the safety and satisfaction of the residents in that city. Another aspect is resource allocation; if the city can maintain the same level of performance at each intersection with fewer resources, the city can spend that money elsewhere. Solving these problems using machine learning can also unravel undiscovered connections between data and the outcome. Perhaps the car's color or the vehicle registration year will have an unexpected impact.

This thesis will use horse racing and, more specifically, trotting and V75 as the problem. Each race is considered an event, and the data is, for example, the horse age, carriage type, shoe, win percentage, and much more. The limiting factor will be an artificial budget, and the aim is to retain as much money as possible playing

on V75. The following section will discuss the background of horse racing, trotting, and V75.

Data availability is the primary reason for using trotting and V75 as the optimization under constraints problem, since there are over 100000 races stored in an archive easily accessible.

1.1 Horse Racing

Horseracing has been an established sport for over three centuries and was first invented in England, but today it is common in many countries, including, Sweden[1]. It is usually heavily associated with betting, and because of this, many people have attempted to find the perfect system to predict the winner[2]. Since it is still thriving today, one can assume that no such method is widely known or exist at all.

With the advancement of technology and analysis, the sport has evolved and now presents its players with more data. This is data related to the horses such as speed, carriage, trainer, age, genes, and much more[3]. Below is an image of all the different data points for each horse you can see on the Swedish betting site ATG.se. One can argue that this is more than you effectively use, but it is hard for a human to determine the relevant parameters[4]. Outside variables, such as start type, race distance, track type, weather, etc., must also be considered when predicting the winner. Different horses are good at various scenarios, but the same can also be stated about the "drivers". To make it even more complicated, in some games, you have to predict multiple races and predict most of them correctly to get any return[5]. As one can see, it is more complex to find the potential winners than it first appears, and since it involves both humans and animals, a perfect system is likely impossible to find.

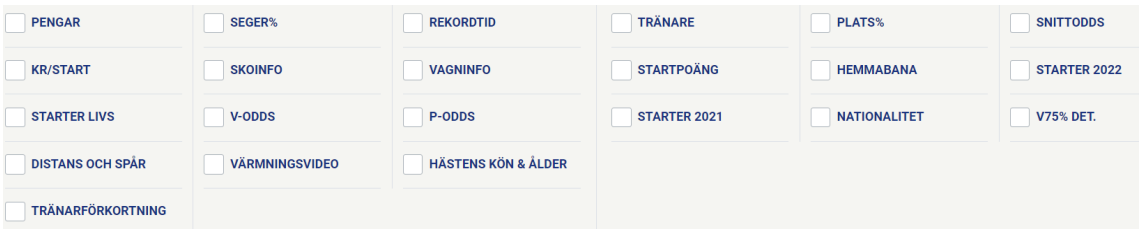

Figure 1.1: All options presented by ATG.se

1.1.1 Trotting and V75

This thesis will focus on trotting with harness racing, specifically, V75[5]. Trotting is when the horse is only allowed to trot and not gallop, and harness racing is when the "driver" is sitting in a carriage behind the horse, as seen in Figure 1.2. The goal is very straightforward, as in most races, the first to pass the finish line is considered the winner.

Figure 1.2

In Swedish trotting, there are two start types and four different distances. The exact distances can vary from track to track but are usually 1640, 2140, 2640, or 3140 meters. One English mile (1609 meters) can also be used on rare occasions.

In Sweden, there are two different starting types called autostart and voltstart. In autostart, a start car is used that gathers the horses behind it and then steadily accelerates up to the start line, where it quickly accelerates away from the horses signaling the start of the race. Voltstart is more complex than autostart; here, the horses are divided into two to three groups, each group of horses running in a loop. The horses start on a start command that goes "klart-ett-två-kör" when the command finishes, the horses should be running so that they all are in the correct track position. With this starting type, redoes are common. Both start types can be seen in figures 1.3 and 1.4.

V75 is a game mode where the player must guess the winner of seven races and get at least five correct to receive any winnings. Of course, the argument can be made that the game mode is unimportant and only the individual races matter, but that is not entirely true. Although the races do not affect one another, the number of horses the player pick in each race affects the system's cost. In V75, the cost is calculated by multiplying the number of horses the player has in each race with each other, and this results in the number of rows the system consists of. In V75, each row costs 0.5 KR, so the number of rows is multiplied by 0.5 to calculate the total cost. So, for example, if the player chooses four horses in each race, the cost would be $4^7 \times 0.5 = 8192$ KR, but if they were too fewer horses in some of the races and more in the others the cost could stay the same but more horses are picked for example, $1 \times 1 \times 3 \times 5 \times 9 \times 10 \times 10 \times 0.5 = 6750$ KR which is both cheaper and ten more horses. This makes it clear that the player should not evenly distribute the horses between the races, and they need to find races where they only pick a small

number of horses, preferably only one, to be able to afford to pick more horses in the more uncertain races.

The payout works a bit differently. As mentioned, there is only a payout when the player gets 5,6 or 7 right, but the exact cash payout usually differs from each system. The payout is determined by the number of winning payers and the amount of money in the pot. The pot could vary from just a few million to over 20 million Swedish crowns. This fluctuation is because if there were no winners in the private V75, that pot would be combined with the next one. This means that some weekends are better to play on than others. But the player also wants to be one of the few winners since every player who got seven right shares the pot, the fewer, the better.

Figure 1.3: Autostart

Figure 1.4: Voltstart

2

Background

2.1 Machine learning

Machine learning (ML) is a sub-field of Artificial intelligence (AI) that study the development of models that are able to learn from data and make decisions from it. The ML can improve its performance by analyzing more data during the training process. This ability enables it to discover patterns and trends that were otherwise hidden and makes ML a versatile and essential tool in numerous fields.

Seeing as ML is a broad area but can be divided into many subareas. However, the two most common are supervised and unsupervised learning. Supervised learning is when the ML learns from labeled data and tries to map the input data (features) to the output. Each input feature is associated with one or more outputs, and during the training, the algorithm uses the labeled data and finds the connections between the features and the outputs. Unsupervised learning is when the ML is trained on unlabeled data.

In addition to supervised and unsupervised learning, there is reinforcement learning and semi-supervised learning. Reinforcement learning is when the ML model learns to make decisions by taking available actions to maximize a goal and minimize a penalty. This learning type is especially popular when machine learning models are tasked with, for example, learning how to walk. Semi-supervised learning is, as the name suggests, a combination of supervised and unsupervised learning. This means the model is trained on partly labeled and unlabeled data.

All machine learning models also have some measurement of how well they are performing. For example, this measurement can be accuracy, whether the model made the right decision or not. This measurement is often used when the model solves a classification problem with a limited number of prediction options. However, pure accuracy is seldom good when the model needs to make a prediction for a regression problem, such as a number with unlimited possibilities. With these types of problems, metrics such as mean absolute error (MAE) are much more suitable since it gives an average of how far the model was off, so the lower the MAE, the better.

ML dose also comes with some common problems; the most common ones are over and underfitting. Overfitting is when models learn their training data well and perform poorly on unseen data. Underfitting is the opposite and occurs when the model doesn't know enough from the training data and performs poorly even on that. Bellow in Figure 2.1 is an illustration of overfitting and underfitting.

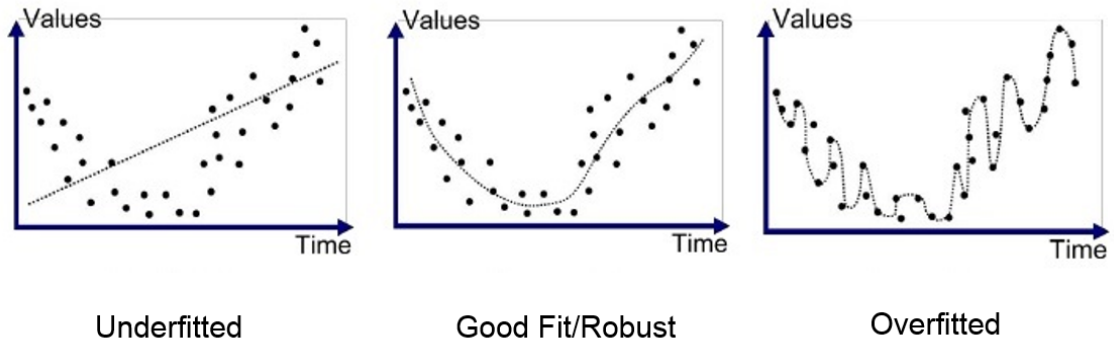

Figure 2.1

To easily observe over and underfitting, it is common practice to divide the available data into two parts, training and validation. The training data set is only used during training, while the validation data set is only used to evaluate the model's actual performance. Employing this practice means over, and underfitting will be easily detected and shows a more realistic model performance.

Machine learning also has multiple sub-fields, such as K-nearest neighbors, decision trees, long short-term memory, and artificial neural networks, and many more. In the following section, neural networks will be explained in more detail.

2.1.1 Neural Networks

Artificial neural networks (ANNs) are models that are inspired by the structure of the neural network in the brain. ANNs consists of interconnected layers, each tasked with applying a specific operation on the input data. During the training phase, the model learns to match input data to the output data, which involves tuning the weights and biases of the model. This tuning aims to minimize a loss metric calculated by the given loss function. This loss function indicates the difference between the actual output and the model's predicted values.

ANNs have two fundamental components: an input layer and an output layer. These layers indicate the beginning and the end of the network. The input layer is the first layer and takes the training data the network will learn from in its rawest form. The number of nodes in the input layer typically corresponds to the number of features in the training data. The input layer performs no actions or modifications on the data and is tasked with passing the data to the next layer. The output layer is the final layer, and the final predictions are generated here. The number of nodes in this layer corresponds to how many values need to be predicted.

ANNs can also contain hidden layers. Hidden layers are neurons between the input and output layers. Hidden layers help the neural network find non-linear patterns between the input features and the output predictions. In Figure 2.2, an overview of the ANN architecture can be seen. ANNs have multiple use cases, such as image recognition, natural language processing, game playing, and speech recognition.

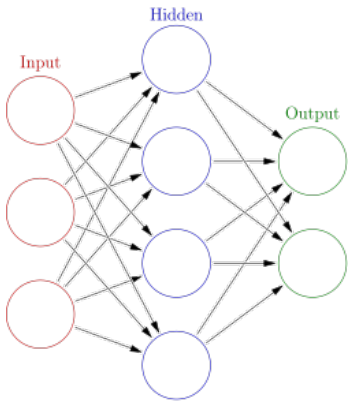

Figure 2.2: Overview of the ANN architecture

2.2 Previous work

Using AI to solve optimization problems is not a new idea, and it has even been used to predict sports outcomes. There are also a few attempts at predicting horse races, which have even been reported in mainstream media[6]. For example, using machine learning, Ndiaye and Koffi demonstrated how to process the horse racing data and how to use that to predict the winner[7], [8]. They attempted two machine learning approaches, "lightGBM" and Deep learning, and found success with them, turning a profit by betting on the winner and betting if the horse ends top 3. Cambell attempted a similar approach but saw less success and experienced significant overfitting during his process[9]. However, in his discussion, he mentions, "One shall study the domain knowledge before data analysis. My lesson is I didn't understand the horse racing terms." He also warns that preprocessing probably takes longer than model building[3]. There are a few other works that are similar to the ones discussed above; however, a few of them chose to specify a specific location, such as Hong Kong [10] or Poland [11], and some try to predict the top two [11]. What they all have in common is that they only focus on one race and focus only on predicting one potential winner or, in some cases, the top 2. However, in this thesis, the focus will be to try to predict the winner in multiple races and choose numerous potential winners in races deemed close.

2.2.1 Sport

As discussed, predicting sports outcomes is not a new idea. However, it is still a complicated problem to solve. For example, there is a yearly competition to develop a machine-learning algorithm that tries to predict the outcome of march madness,

the National Collegiate Athletic Association men's and women's college basketball tournaments[12]. This ML competition has over 1000 competitors and has had some very talented and educated winners. Since it is a yearly competition with different winners, it is not a problem someone has fully solved. Microsoft has also been working on predicting the football world cup using Bing[13]. In 2014 Bing did manage to predict all 16 of the games in the knockout stage. However, in the group stage, Bing only had a success rate of 60% and also failed to predict the winner in the final, so it is far from perfect. It is also worth noting that the 2014 word cup had very few upset wins, and it was usually the statistically favored team that won. It is also worth mentioning that ML predictors developed by large companies such as Google or Microsoft are typically kept secret since there is money to be made in sports betting. However, these were two result sports where either team A or B won. There are many more outcomes in horse racing, and it is, therefore, harder to predict.

The article "The Future of Sports Betting: AI-Powered Predictive Analytics" discusses the use of AI in sports but also reflects on AI in sports ethically[14]. The article mentions that "the main concern is the potential for AI to be used to manipulate outcomes in favor of certain bettors or teams. This could lead to unfair advantages and an uneven playing field, which would be detrimental to the integrity of sports betting." They also discuss that AI can find patterns between the data and the outcome that humans would not be able to identify, giving an unfair advantage to better who have access to this technology. However, the points discussed in the article are valid; therefore, no actual bets will be used during this thesis.

On the other hand, some work indicates that Artificial intelligence and machine learning do not have unfair advantages compared to human predictions. In the paper "Human Decision Making and Artificial Intelligence: A Comparison in the Domain of Sports Prediction," the authors compare the accuracy of human and AI predictions in the 2015 rugby world cup[15]. The paper found that the AI had an accuracy of 89.58% and the humans had 85.52%, and it drew the conclusion "that for rugby, over the limited period of a specific tournament, the evidence was not strong enough to suggest that a human agent is superior in terms of accuracy when predicting match outcomes compared to a machine learning approach."

3

Methods

At a high level, the project consists of five primary parts, all working with some form of data. The parts are web scraper, Preprocessor, Validator, Machine Learning, and lastly, Evaluator. Figure 3.1 shows that the data begins as a web archive, and with the web scraper, the archive is turned into raw data. The Preprocessor then processes that raw data. After this step, the Validator can check the data to confirm its accuracy. The data is now ready to train a machine-learning model to generate predictions. Lastly, the predictions can be evaluated to determine the performance and accuracy of the machine learning models.

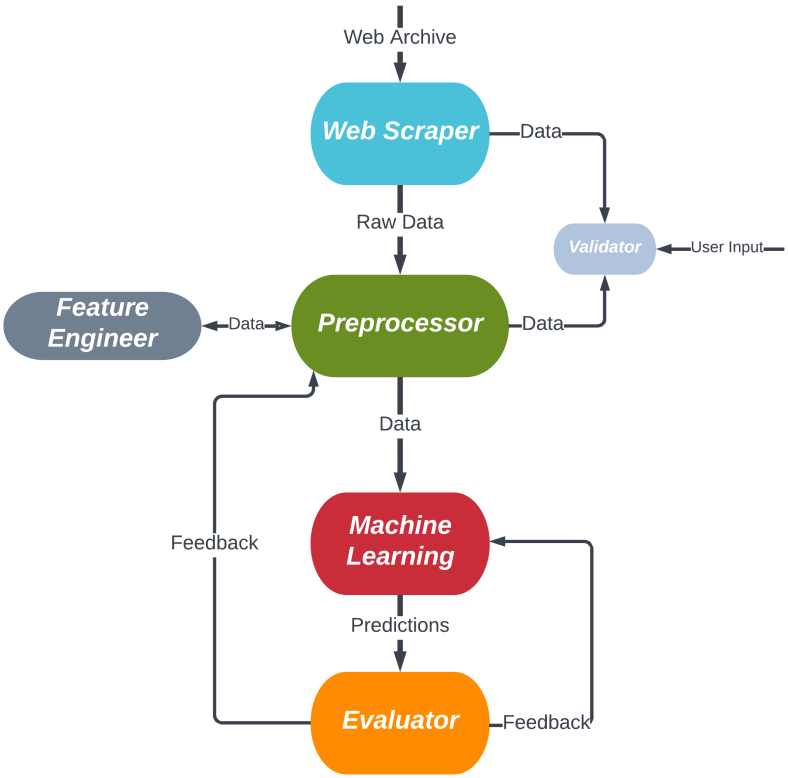

Figure 3.1: An overview of the whole process

3.1 Data

The project's data-related task is divided into two sections, scraping and preprocessing. Each section has its sub-parts and will be explained in detail in this section. Initially, it was planned for the preprocessing to be a subclass of the scraping, which was valid for early versions. However, after significant reconstruction and functionality addition, it grew and was separated from the scraping class. Additionally, both are developed in such a way that they are modular, and new functionality can easily be added, or existing mechanics can be tuned without requiring much reconstruction or disrupting the functionality of other parts.

3.1.1 Scraping

The Scraper is designed to collect data from the web containing information about horse racing. It starts by gathering race Ids from a race archive. The program then uses these ids to access the specific races and collect all relevant data about the race and the horses and stores it in a CSV file. CSV or comma-separated values is a commonly used format for artificial intelligence, machine learning, and general data storage. It is a text file that separates values with a comma[16]. The Scraper class uses several different libraries to streamline its process. It uses the request library to handle posts and get requests against websites. Json[17] is then used to extract the information from the scrape data the request returned, and lastly, Pandas library[18] is used to store and save the data as a CSV. The Scrapper also used some custom classes to represent the data more intuitively. Primarily there are two classes, the race class and the horse class. As the names of the classes suggest, the race class represents the race-related attributes, and the horse class the horse specifics ones. Both classes can be seen in the code below with all their values; note that the race class has a list of horse objects. The following section will explain the scaping process from start to finish, step by step, at a reasonably low level.

```
class race:
    id: int = None
    date: int = None
    track: int = None
    distance: int = None
    start_type: bool = None
    horses = []
```

```
class Horse:
    id: int = None
    name: str = None
    money: int = None
    money_per_start: int = None
    distance: int = None
    track: int = None
    win_percanteage: int = None
    shoes_front: bool = None
    shoes_back: bool = None
    shoes_change: bool = None
    carriage: bool = None
    carriage_change: bool = None
    age: int = None
    home_track: str = None
    at_home: bool = None
    top3_percentage: int = None
    place: int = None
    gallop: bool = None
    kmTime: float = None
    starts_life: int = None
    driver_id: int = None
    trainer_id: int = None
    starts_2023: int = None
    third_2023: int = None
    second_2023: int = None
    first_2023: int = None
    starts_2022: int = None
    third_2022: int = None
    second_2022: int = None
    first_2022: int = None
    average_odds: float = None
    V75_odds: float = None
    top3_odds: float = None
    points: int = None
    record_time: float = None
```

3.1.1.1 Process

This Section will follow a single race throughout the scraping process. An overview of the process can be seen in Figure 3.4. As mentioned in the previous Section, an archive is first scraped to gather race ids. A race id is a string structured as YYYY-MM-DD_XX_ZZ. The date represents the day the race took place, XX represents the track id, and ZZ represents the race number. This id is unique for each race, and every ID is saved, so no archive scraping is done twice.

The program can now perform an additional scraping with the id. The result of this scrape is data relevant to that specific race; however, it is in the form of a TXT file. That TXT file is now processed into a more suitable format and added to a CSV file. The whole process is now done again. For a comprehensive explanation of the implementation, see Appendix A.1.1.

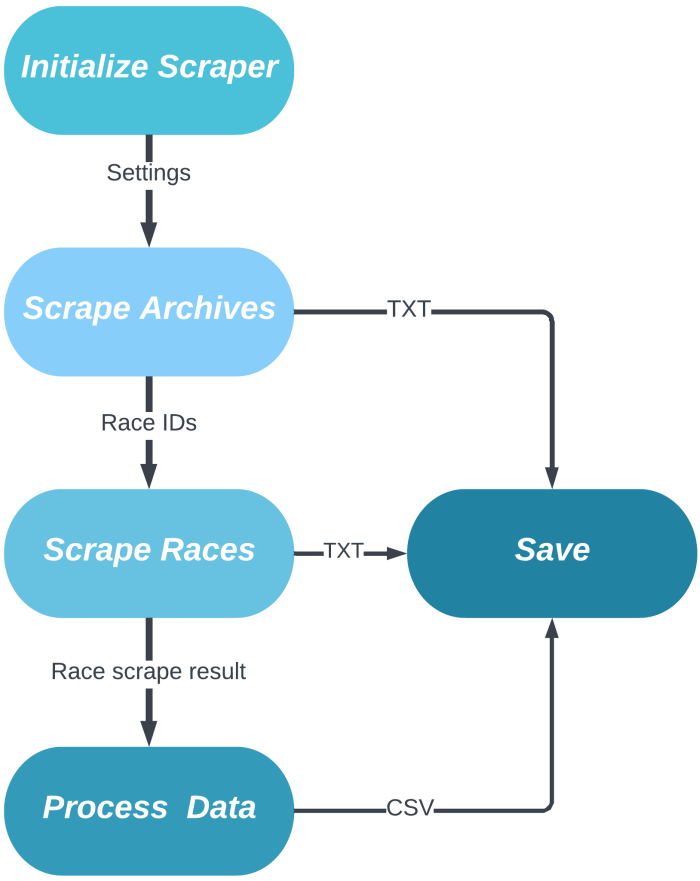

Figure 3.2: An overview of the scraping process

3.1.2 Preprocessing

The goal of the Preprocessor is to take the raw data from the scraper and convert it to something more suitable for machine learning. More specifically, the Preprocessor class has multiple assignments that are divided correctly to enable effortless tweaking and simplifying the process of adding and removing new tasks. The different sub-processes include the calculation of lap percentage, removal of unnecessary data, removal of data only available after the race, and the normalization of every column. The following section will motivate and explain these sub-processes in more detail.

In addition to the data-related tasks, the Preprocessor also handles loading and saving the processed data. This ability makes it so the same data does not need to be processed twice, significantly reducing the execution time.

To aid in these tasks, the class uses three libraries. Pandas library helps handle the data, the math library assists in calculating the lap times, and the train test split library divides the data[19].

```
    def _preprocess(self):
        # Gets the lap percentages in the form of a data frame
        df_y = self._get_lap_percentages()

        # Process the race data
        df_x = self._process_x()

        # Normlizes all race data
        df_x = self._normalize_all_columns(df_x)

        # Saves both the race data and the lap percentages
        self.save(df_x, df_y)

    def _process_x(self):
        df_x = self.df

        # Removes the data that is considered unnececary
        df_x = self._remove_unnececary_data(df_x)

        # Removes data that is only available after the race
        df_x = self._remove_cheating_data(df_x)

        # Removes data that was given to declude
        df_x = self._remove_columns(df_x, self.columns_to_declude)

        return df_x
```

3.1.2.1 Process

In Figure 3.3, the high-level data flow of the Preprocessor is shown. For an unprocessed race, it begins with the calculation of lap times. The lap times determine the finishing order of the horses and can also be used to calculate the lap percentages, which indicate how close a race was. The lap times are then stored in a separate CSV file. Several data points are now removed that are deemed unnecessary or cheating. For example, race id is removed since it has no impact on race. Similarly, the finishing time and if the horse galloped is also removed since it is a fact known only after the race is completed, and thus effectively would be cheating. Lastly, all the data points for the race are normalized and then saved in a separate CSV file. A more in-depth explanation of the implementation process can be found in Appendix A.1.2.

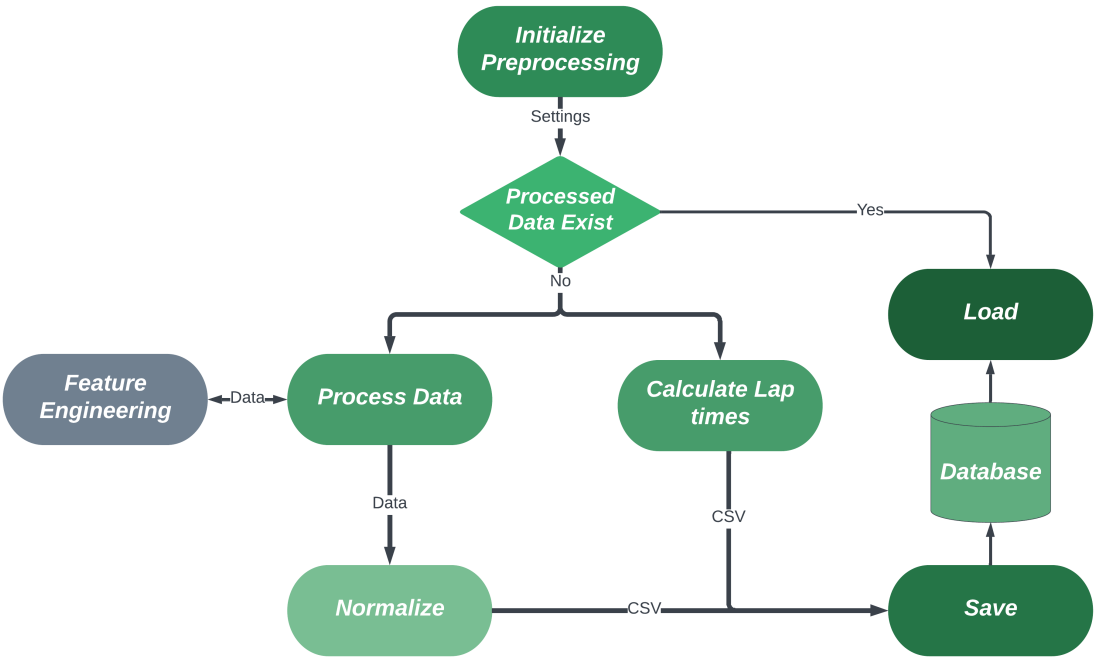

Figure 3.3: An overview of the preprocessing process

3.2 Validator

The Validator class is a relatively simple program but very important. As the name suggests, it is used to validate that the scraper and Preprocessor work correctly and catch any inconsistencies. It is not fully automated and is more of a tool to speed up validation and requires user input to determine if a data point is correct. The Validator is constructed in a modular and adaptive way, so if any changes are made to either the scraper or Preprocessor, the Validator does not require any modification.

3.2.1 Process

The Validator can be given the races it should evaluate or get random races from a CSV. How it gets races are determined in the get races section in Figure 3.4. Once it has the races, it moves on to validate them. It presents the user with each data point in the race and asks whether that is correct. When all data points and races are checked, the results are saved in a TXT file. Appendix A.1.3 provides a more detailed explanation of the process.

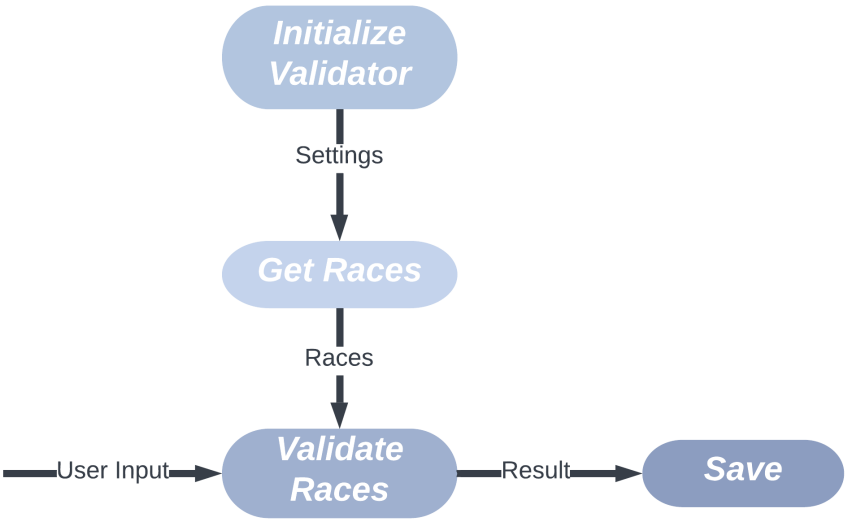

Figure 3.4: An overview of the Validator process

3.3 Evaluation

The evaluation part of the project is crucial and essential to determine how well a model is performing. The evaluation can be divided into two parts. One that measures different metrics and calculates new ones. The other part is about creating baselines that are used to compare with the machine-learning models. Both are equally important in determining the performance of a model.

3.3.1 Evaluator

The purpose of the Evaluator is to take predictions and correct answers, compare them and, from that comparison, calculate metrics and graphs that showcase the accuracy of the predictions. The predictions are given in a list, so the Evaluator can not determine if the predictions came from a neural network model or a trivial baseline. This approach is by design and makes it so the Evaluator can assess an advanced model and a baseline identically. The Evaluator also handles storing these different metrics as well as the different tables and graphs created. This feature eases comparing models and baselines against each other. The Evaluator is also

structured so that it is effortless to add additional metrics, which has been crucial since the Evaluator has evolved throughout the process.

3.3.1.1 Process

The race evaluation starts with translating the predicted and correct lap percentages to the finishing positions for a more straightforward comparison. This comparison is during the Evaluate Races process in Figure 3.5. During this process, "totals" are also calculated, such as how many actual horses finished 5th, how many were correctly predicted 5th, and so on. The process now moves on to calculating the metrics. Here the totals from the previous section are used to calculate numerous percentages, such as accuracy. These metrics are then saved to disk. In addition, some are used to generate graphs that are also saved. For additional information regarding the implementation, see Appendix A.1.4.

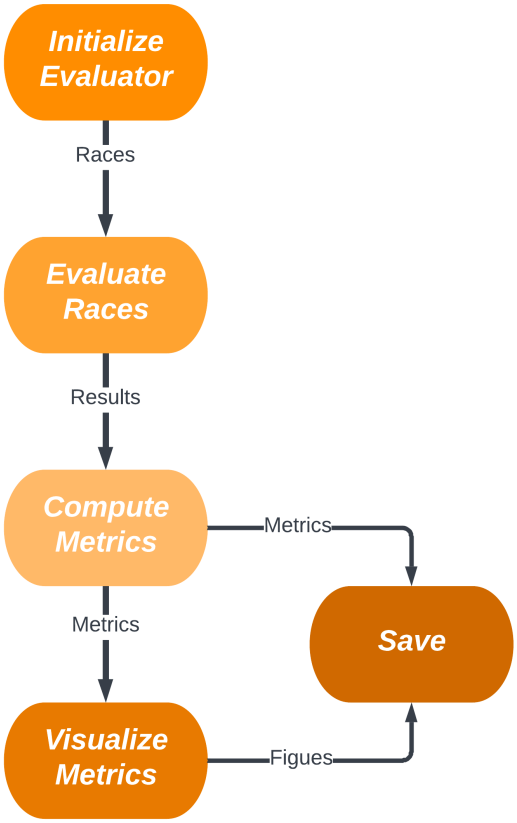

Figure 3.5: An overview of the Evaluator process

3.3.2 Baselines

To determine how the developed machine-learning approaches performed, they were compared to trivial baselines. Since baselines have vastly different accuracy, multiple ones were used in varying degrees of complexity. The overall structure of each baseline is similar and also shares some similarities with the machine learning models. None of the baselines used any form of learning. However, the baselines have a prediction method similar to the machine learning model that takes race data, and depending on the baseline, it uses that data to determine the race's finishing order and return it. The reason for these similarities is that it makes it easier to evaluate the baseline with the same Evaluator that the machine learning models use.

3.3.2.1 Random Baseline

As the name of the baseline suggests, this model ignores all the given data and returns a random finishing order. This is the least complex and worse performing baseline. This model was primarily used initially as a sanity check and a pathfinder for future baselines. The code for this baseline can be seen below and gives an idea of how future baselines will be structured.

```
class RandomBaseLine:

    @staticmethod
    def predict(self, race_data, horses):
        result = []
        for i in range(0,len(race_data)):

            result += [random.sample(list(range(0,horses)), horses)]

        return result
```

3.3.2.2 Starting positions baseline

This baseline determines the predicted finishing order based on each horse's starting position and aims to demonstrate a connection between the data and the result. Seeing as some starting positions are better than others. The baseline is still relatively simple and will predict the horse with the first track position as the winner and the rest in descending order. It is, however, worth noting that the best track positions do not necessarily go from the first to the last. It is usually best to have one of the first track positions; however, some horses have trouble running directly next to or behind the car, meaning that a middle position can be one of the worst depending on the horse. Also, in the case of a volt start, multiple horses can have the same track position, and in that case, the horse with the lowest number is prioritized, seeing as that horse starts in the front row.

```
    def predict(race_data, horses):
        track_position = race_data.filter(regex = "_track")

        for  data in track_position:
            order = np.sort(data)
            for i in range(0, horses):

                index = data.tolist().index(order[i])
                race_result[index] = i
                data[index] = replacer
                replacer -= 1
            result += [race_result]

        return result
```

3.3.2.3 Win percentage baseline

This baseline uses the overall win percentage of each horse to determine the finishing order, and it returns the horses in the finishing order of highest win percentage to lowest win percentage. Unlike previous baselines, this baseline effectively uses previous race results to determine the finishing order, presumably leading to a significant increase in accuracy and performance.

```
class WinPercentageBaseLine(BaseLine):

    @staticmethod
    def predict(race_data, horses):
        result = []
        win_percentage = race_data.filter(regex = "_win_percentage")
        for  data in win_percentage:
            race_result = [0]*horses
            order = np.sort(data)[::-1]
            replacer = -2
            for i in range(0, horses):
                index = data.tolist().index(order[i])
                race_result[index] = i
                data[index] = replacer
                replacer -= 1
            result += [race_result]


        return result
```

3.4 Machine learning models

3.4.1 Tensorflow and Keras

TensorFlow[20] and, more specifically, Keras[21] will be used for the first implementation of the machine learning model. Both are open-source deep-learning libraries compatible with Python. They have a user-friendly and straightforward interface for building, training, tuning, and evaluating neural networks. They are excellent for first-time implementation and should give a good first indication if the model can find any patterns between the input and output.

In more detail, a Keras machine learning model comprises interconnected layers, each tasked with applying a specific operation on the input data. During the training phase, the model learns to match input data to the output data, which involves tuning the weights and biases of the model. This tuning aims to minimize a loss metric calculated by the given loss function. This loss function indicates the difference between the actual output and the model's predicted values.

Here is a high-level overview of A typical Keras model consisting of a description of the different layers in the neural network. The architecture includes the input layer, which needs to have the input shape of the input data, and the output layer, which must know the form of the output data. The architecture can also include multiple hidden layers which perform specific operations on the data from the previous layers.

The model now needs to be compiled, which includes defining the optimizer, the loss function, and the evaluation metrics. The task of the optimizer is to tune the model's parameters (weights and biases). The loss function, as mentioned previously, measures the difference between the model prediction and the actual output values. Furthermore, the evaluation metrics such as accuracy and precision are used to help assess the model's performance.

Training the model now commences and is usually the most time-consuming step in the process. The model receives input and output training data and starts iterating over it. The loss metric is computed during each iteration, and the model parameters are updated. The process is repeated for a number of so-called epochs, which is a complete pass through the dataset. After the training, the model can be evaluated using a test dataset. The test dataset includes data the model has not encountered before, which indicates the model's actual performance and can uncover over and under-fitting. Depending on the evaluation results, the model's different hyperparameters can be fine-tuned. These are parameters such as learning rate, batch size, layer configuration, and much more. This process includes a lot of trial and error.

The following sections will describe the actual model used in this thesis. As mentioned before, the exact values of the hyperparameters are subject to trial and error, so precise values will be presented in the results section.

3.4.1.1 Model

This model aims to get the input and output layer working properly and have the machine learning model compile correctly, it also hopes to find improvement between each training epoch and subsequently perform better than the random baseline. However, the model is still simple enough that the training and testing are relatively fast to ease the tuning phase. Actual hyperparameter tuning will also be performed on this model. Below, the code can be seen.

The model consists of 3 layers, the input and output layers which are separated to leave room for a dense layer between them. The dense layer has 128,256,512 or 1024 neurons and has either Tanh, ReLU, or Sigmoid as activation functions. The optimizer is Adam, with a learning rate of 0.001, 0.0001, or 0.00001 and a loss function of mean absolute error (MAE). What hyperparameter performance best will be presented in the result section. To summarise this model, the goal is to find what activation function works best, what number of neurons gives the best performance in the single hidden dense layer, and what learning rate should improve accuracy.

The input data for this model are all seen in Section 3.1. The total dataset will be 10000, where 8000 is used for training and 2000 for validation.

The data processing will be divided into two different experiments. In experiment one, no normalization is done, and the data is in its rawest form. In experiment two, the data is normalized. Hyperparameter tuning will be performed in both experiments.

3.4.1.2 Hyper parameter tunner

The model discussed in the previous sections requires significant tuning and testing. For example, the intermediate model has three options for learning rate, three for activation function, and at least four for the number of neurons. If all combinations are to be tested, that would be 36 different models, which will take quite a lot of time depending on the data set size. If this testing were done manually, it would require tedious work and be suboptimal in searching efficiently. However, Keras Tuner is an open-source flexible, and powerful Python library that can manage the hyperparameter optimization process for Kears models.

Keras tuner works by using different search algorithms, such as random search, hyperband, and Bayesian optimization, to discover the best-performing hyperparameters. The library starts with defining all possible hyperparameter values and then iteratively evaluates the different combinations. The library uses a user-defined objective function to determine what combination works best. The object function can, for example, be the validation accuracy or loss. At the end of the process, the tuner returns the best-performing model according to the objective function. Below is the code for the intermediate model.

```
def build_simpel_model_tuner(hp):
    model = Sequential()
    activation = hp.Choice(
    "activation", values=["tanh", "sigmoid"])
    model.add(Dense(x_train.shape[1]+1,activation=activation,
    input_dim = x_train.shape[1]))
    model.add(Dense(units=hp.Choice("units",
    values=[128,256,512,1024]),
    activation=activation))

    model.add(Dense(15))
    model.compile(
        optimizer=Adam(
            learning_rate=0.0001
        ),
        loss='mae',
        )

    return model

earlyStopping = EarlyStopping(
monitor='val_loss', patience=80,
verbose=1, mode='min', restore_best_weights=True,
min_delta=0.001)

reduce_lr_loss = ReduceLROnPlateau(
monitor='val_loss', factor=0.1, patience=40,
erbose=1, min_delta=0.001, mode='min')

tuner = RandomSearch(
    build_simpel_model_tuner,
    project_name = "experiment_1_simple_model",
    objective="val_loss",
)

tuner.search(x_train,y_train, epochs =
10000,validation_data=(x_test, y_test), batch_size=200,
callbacks=[earlyStopping, reduce_lr_loss])
```

3.5 Multi-Event Decision Making

With a machine learning model that can predict individual races see Section 4.1, the focus changed to predict systems of races, such as V75, as explained in Section 1.1.1. To achieve this, two picking algorithms were developed, ranging in complexity.

Although the algorithms vary, they do have a lot in common. Such as, they all require at least three input variables, a budget, the price, and a list. The budget is the maximum expense a system is allowed to incur. The cost represents what one row in the system costs. Lastly, the list contains races, and each race has the predicted percentage of a horse and its unique id. Each algorithm also returns the same type of structure. For each race given, the algorithm returns a list indicating by id what horse to pick. So, for example, in V75, the cost would be 0.5kr, and the list would contain seven races. The return structure would be a list of seven lists containing IDs for what horse to pick in each race. However, the algorithm would be able to handle other game modes, such as V64 or V86.

Since the picking algorithms only predict one system and do not handle dividing the races into systems, some preprocessing must be performed. The SystempredictionManager handles this, which takes two lists of races, one with the horse ids and the predicted percentage for each horse and the other with the horse ids and the actual percentages. Both lists are in the same order meaning that the first race in the predicted list is the same as the correct percentage list. The lists are now divided into systems of a given size, for example, 7. With the division complete, picks for each system in the prediction list are calculated using a given picking algorithm.

The following sections will list and explain the picking algorithm that the SystempredictionManager use.

3.5.1 Greedy naive odds baseline

The greedy naive odds baseline is similar to the greedy picking algorithm; however, instead of using the predicted percentage, the baseline will use the odds. So this baseline first picks one horse in each race that the odds show will win. After that, it finds the horse who is closest to the winner according to the odds in each race, and in the race that is the closes, it chooses that horse. This process is repeated until the budget is reached.

Presumably, this baseline will perform very well; however, using this approach will lead to a minimal payout as was described in Section 1.1. So although this baseline is highly likely to beat the ML in pure numbers, it might not mean it surpasses it in money payout. However, seeing as this thises will only focus on the number of correct systems, this will not be observed in the results.

3.5.2 Greedy naive picking algorithm

The greedy naive picker is a simple, straightforward algorithm. It begins by picking the best horse in each race, and since the ML model predicts the percentage each

horse finishes behind the winner, it is the one with the lowest percentage. It then looks at each race, finds the second-best horse, and calculates the distance between the 1st and 2nd. It now finds the horse with the smallest distance to the winner and calculates the system's total cost to add that horse. If the cost is under the budget, the horse is added to the picks and removed from the list of available horses to pick. However, if the total cost exceeds the budget, the horse is not added to the picks but is still removed from the list of available horses. This process is repeated until there are no more available horses to choose from.

4

Results

This chapter will present the results and performance of both single-race and system predictions. Additionally, some analyses and observations will be discussed.

4.1 Single race predictions

Single race prediction is not the primary objective of this thesis and is merely a stepping stone for the primary purpose of picking a system of races. However, it is a fundamental process, and selecting a system would be impossible without it. In the following sections, the results of different race predictions will be presented, both results of different types of machine learning models and also various kinds of data processing.

4.1.1 Baselines

4.1.1.1 Random Baseline

The random baseline, as expected, performed very poorly, with an overall accuracy of 6.8%; however, that is not a very good metric. The overall accuracy includes all positions, but it is predicting the first place that is the most important. So in the placement accuracy figure, we see the accuracy for each position. We also, in this figure, see some volatility toward the later positions. For example, at position 11, we have an accuracy of 8.4%, while at 14 and 15, it is zero. These accuracy levels can be explained by the fact that there are few races with this many horses. Therefore, one horse at the 14 and 15 finishing positions represents a more significant percentage. Since the ultimate goal is to predict multiple races and, in some cases, pick numerous horses in a single race, it would be interesting to see if the winner was included if you take the top x best-predicted horse where x can be 1 to the number of horses in the race. The accuracy when x is one is already known since they only take the predicted winner, so 6.6 %. However, in Figure 4.2, we see the accuracy if we include, for example, the two best-predicted horses. Additionally, a metric was calculated to show how far off the predicted finishing position was from the actual finishing position on average. So, for example, if the actual placement was first and the horse was predicted to finish fourth, that would be a distance of three. Figure 4.3 shows the average distance the prediction was off at each actual placement. The random baseline has an average distance of 4.53.

The random baseline does not perform well. However, it indicates what a lousy performance looks like and what a good performance might be. This baseline can be seen as the floor of expected performance.

The average distance and top x accuracy figure also show some interesting patterns. In the average distance, a U shape can also be observed. This shape is simply because at the position when prediction, position seven can predict over and under, while at number one, only lower predictions can be made. The top x accuracy Figure is linear, which is to be expected with random picking.

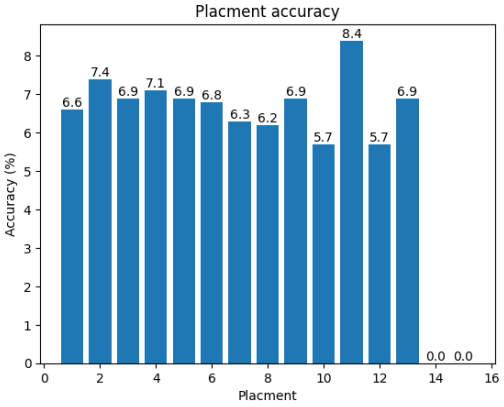

Figure 4.1

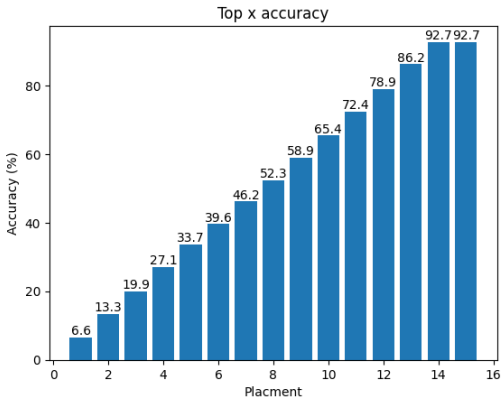

Figure 4.2

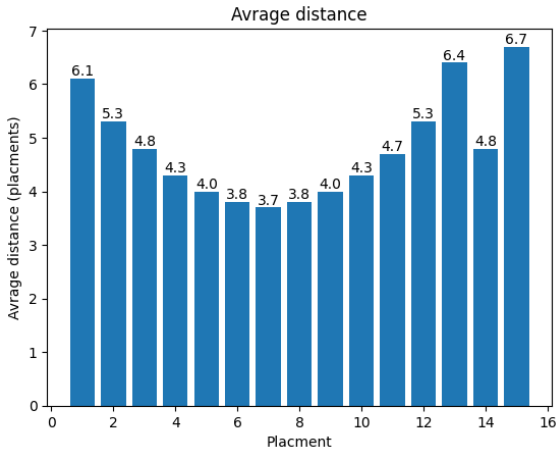

Figure 4.3

4.1.2 Starting track

The starting position baseline performance is better than the random baseline, indicating a correlation between the horse's starting and finishing positions. It has an overall accuracy of 8.33% and an average distance of 3.78, both better than random. A few interesting observations can be seen in Figures 4.4, 4.5, and 4.6. In Figure 4.4, we see that the starting position is better at indicating some finishing positions than others, such as 1,2,8,10, and 12. The U can still be observed in the average distance position but is more flattened, and its center has moved to the left, closer to the top positions. Lastly, the top X accuracy is starting to show some nonlinear patterns indicating that the winner is favored towards the top.

This baseline is still pretty poor, but it does indicate what finding a pattern between the data and the finishing result might look like. Although this pattern appears to be bad, it is still better than the random baseline.

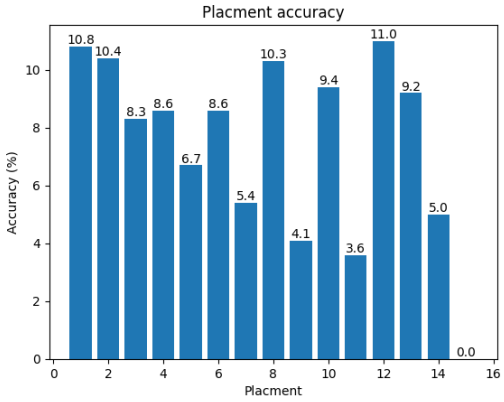

Figure 4.4

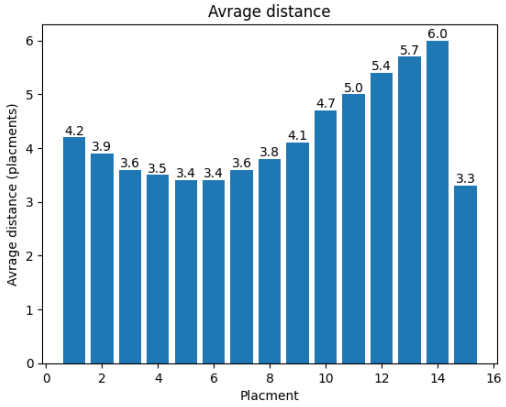

Figure 4.5

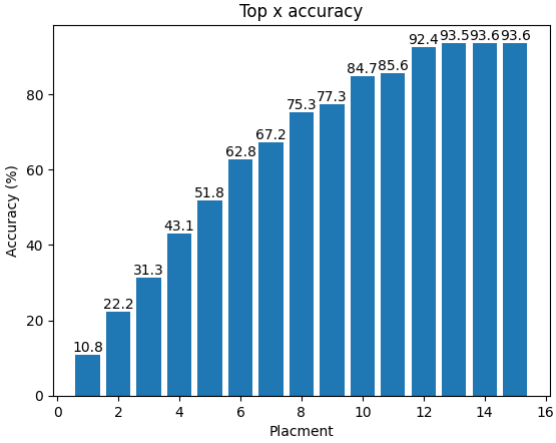

Figure 4.6

4.1.3 Win percentage

The win percentage baseline shows considerable improvements showing a strong correlation between the horse's win percentage and its finishing position. It has an overall accuracy of 12.39% and an average placement distance of 2.99. However, the most significant improvement can be seen in the placement accuracy Figure 4.7, where we can see that finding the winner has an accuracy of 24.3%, which is a massive improvement from the other baselines. However, considering it is the win percentage that is used in this baseline, it is not very surprising. Other observations that can be made in Figures 4.8 and 4.9 are that the U in the average distance graph is considerably flatter towards the top positions, and the top x Figure is not linear anymore but rather logarithmic.

This baseline demonstrates an acceptable level of performance, mainly when predicting the winner. Additionally, it highlights the pattern in average distance Figures and top X accuracy that should be aimed for.

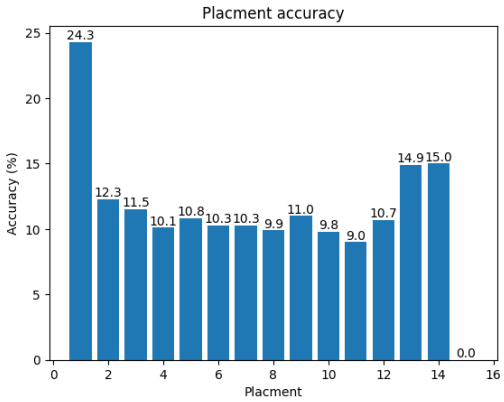

Figure 4.7

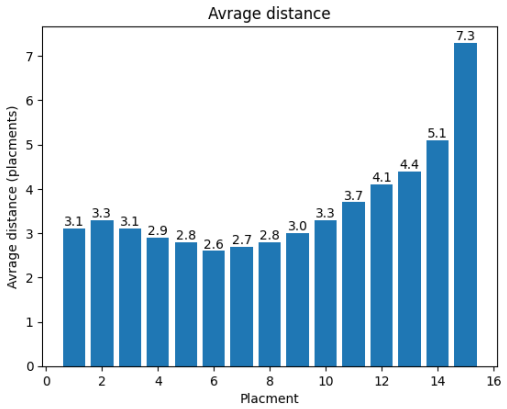

Figure 4.8

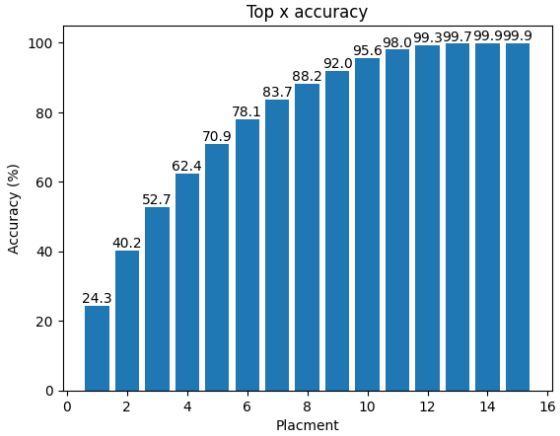

Figure 4.9

4.1.4 Experiment 1

The best-performing hyperparameters for the model in Experiment 1 described in section 3.4.1.1 were the activation function of "tanh," 512 neurons in the single hidden layer, and a learning rate of 0.0001. Below is the exact implementation of this model:

```
model = Sequential(name=name)
model.add(Dense(input_dim+1,activation="tanh",input_dim))
model.add(Dense(units=512,activation="tanh"))
model.add(Dense(15))
model.compile(optimizer=Adam(learning_rate=0.0001))
```

The model had a mae of 2.53 at its best, and the progression during the training process can be seen in Figure 4.10. However, the mae does not give a good indication of how this translates into actual placement prediction, so with the help of the Evaluator described in Section 3.3.1, we see that we have an overall correct accuracy of 11.34 %. In Figure 4.11, the accuracy for each position is shown, and here it can be observed that predicting the winner has a significantly higher percentage of 16.9% than the overall percentage. It is also clear that the model has found some form of a pattern since it performs considerably better than the random baseline.

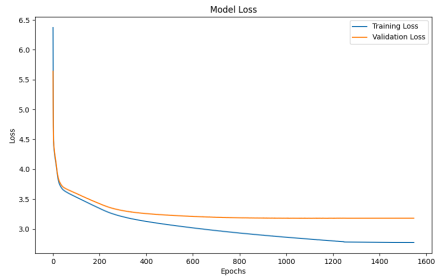

Figure 4.10

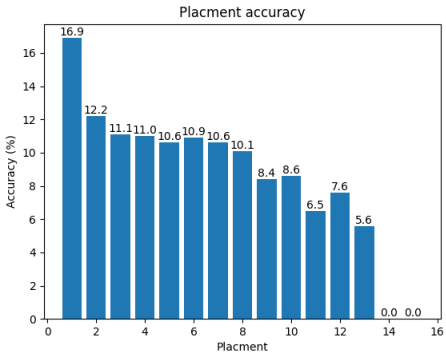

Figure 4.11

The top x accuracy graph 4.12 shows a logarithmic improvement similar to the win percentage, and the Average distance graph 4.13 shows a somewhat flattened U shape compared to the random baseline.

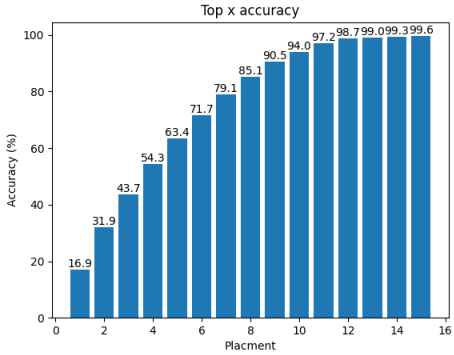

Figure 4.12

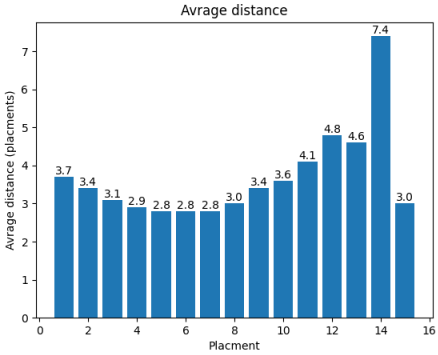

Figure 4.13

4.1.5 Experiment 2

In experiment 2, the data is now also normalized, meaning that every value is between 0 and 1. Normalizing makes it so that each feature is treated equally. Before, higher values had a more significant effect than smaller values. The model that performed best now during the tuning process is very similar to the previous one, except for having 1024 neurons in its hidden layer instead of 512. Another difference is that this model trained significantly faster than the previous one, which can be observed by comparing figures 4.14 and 4.10 where we see that the previous model needed more than 1500 epoch to stop learning while this model only needed less than 700.

With normalization enabled, we have an overall accuracy of 13.42%, which is an improvement but not a large one. However, once again, this metric only shows part of the picture. If we look at the placement figure 4.15, we see that the accuracy for the winner is significantly higher, with an improvement of 7.5 %. The same can also be observed in predicting 2nd place, which increased by 3%. The rest of the placement grew by roughly 1%.

A similar trend can be seen on the average distance in Figure 4.16 where placements 1 and 2 had the most improvement with the exception of placement 14, which is probably an outlier due to a few races having 14 horses. In the top x accuracy, we see a significant improvement, as seen in Figure 4.17. After including the top 3 horses, this model has almost exact distances as the previous model had at 4.

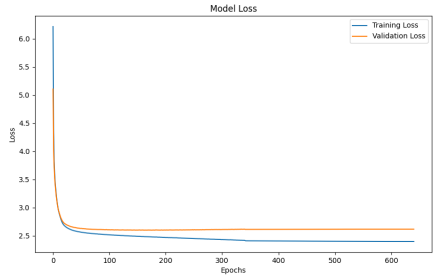

Figure 4.14

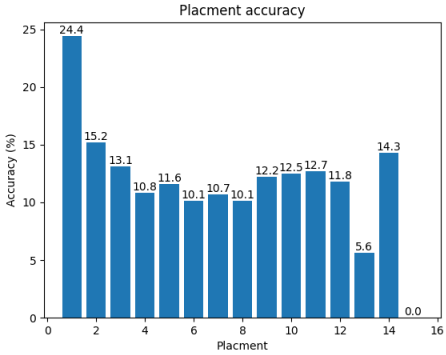

Figure 4.15

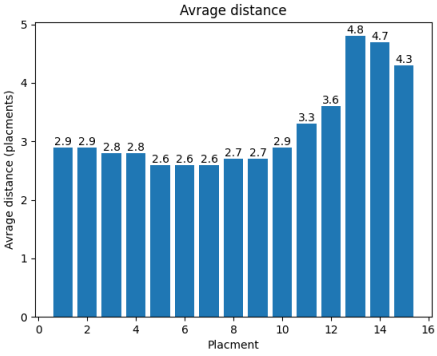

Figure 4.16

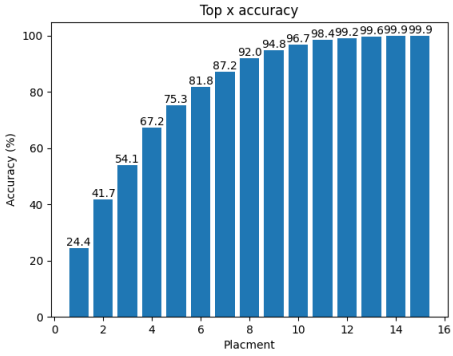

Figure 4.17

4.2 System race predictions

Only the best-performing machine learning model from each experiment was used for system prediction. However, multiple-picking algorithms, such as greedy, were tested with each model. The algorithms were also tested when they were forced to only select one horse in x races where x can be 1 through 4. This allows the picker to pick more in the uncertain races and forces it only to choose one horse in the races if it thinks a clear winner is present.

To simulate a V75, the algorithms were tested with three budgets: 124, 1024, and 8192. The algorithms predict the outcomes of seven random races. Section 1.1.1 explains that a V75 only results in a payout if at least five races are correctly predicted. If the algorithm gets less than five correctly, it is considered a complete miss. The pickers were be tested on the test dataset, which is roughly 2000 races, which means it is 285 systems with a size of 7.

4.2.1 Experiment 1

4.2.1.1 Greedy Algorithm

The greedy algorithm with the data in its rawest form and a budget of only 124 kr only had 11 systems, where it got five right. This amount corresponds to 3.9 % of

the 285 systems it tried to predict. The algorithm guessed, on average, 2.5 of the seven races correctly. The algorithm accuracy slightly increased when forced to pick only one horse in 3 races. Then it got five right in 4.5 % of the races and found the winner in 2.6 of the seven races.

With a budget of 1024 kr and not being forced to pick one horse in any race, the greedy algorithm got three systems with six rights and 37 systems with five rights. In percentages, this results in 0.1% and 13%. The number of found winners on average in each system has also increased to 3.16. However, when forced to pick one horse in 4 races, the algorithm gets one system with seven correct guesses (0.1%), four systems with six (1.4%), and 35 systems with five (12%). And the found winner average in each system increased to 3.6.

Lastly, with a budget of 8192 kr, the algorithm got three systems with seven (1%), 22 with 6 (7.7%), and 90 with 5 (31.6%) and found the winner on average 3.9 times in each system. It also so no improvement when forced to pick one horse in 1-4 races.

4.2.2 Experiment 2

4.2.2.1 Greedy Algorithm

With normalization enabled and a budget of 124 kr, the greedy algorithm gets one system with seven right (0.35%), 13 with 6 (4.6%), and 60 with 5 (21.1%), significantly better than experiment 1. The same can also be seen in the average number of races correct in each system, where this experiment gets 3.4, which is better than experiment one with a budget of 1024. However, it does not see any improvement when forced to only pick one horse in 1-4 races.

With the budget increase to 1024 kr, the same trend can be seen where this experiment gets seven systems with seven correct picks (2.5%), 39 with six (13.7%), and 115 (40.4%) with five, which is significantly better than even the highest budget of experiment one. Now the average of correct choices in each system is up to 4.2. However, similar to the lowest budget, no improvement was seen when forced to pick one horse.

For the highest budget of 8192 kr, experiment 2 gets 23 with seven right (8.1%), 90 with 6 (31.6%), 186 with 5 (65.3%), and 4.9 correct picks on average in each system. This result is incomparable to any previous one; however, once again no improvement when forced to pick one horse.

4.2.3 Baselines

4.2.3.1 greedy odds baseline

The greedy odds baseline did unsurprisingly well, always taking the horse with the best odds until the budget was reached. With a budget of 128, it got 15 systems with seven correct races (5.3%), 81 with six correct (28.4%), and 179 with five (62.8%). Also, forcing the baseline to pick only one horse in two races did see an increase in both the number of seven right systems and five, with 21 (7.3%) and 181(63.5%). There was a slight decrease in the number of six correct systems to 78 (27.3%).

With a budget of 1048, these are increased to 51 (17.9%), 145(50.8%), and 231(81.1%) of seven, six, and five rights. With this budget, there was no increase in performance when forcing the baseline to pick one horse in one to three races.

Lastly, with budget 8192, the baseline gets 99(34.7%) with seven, 205(72%) with six, and 258(90.5%) with five. As with the previous title, no improvement was seen when forced only to pick one horse in one to three races.

What needs to be remembered, however, is that this baseline might have picked a lot of the correct horses; however, they are the horse with the lowest payout. This baseline may give less payout than a single seven-correct system with multiple upset wins.

5

Conclusion

5.1 Discussion

A few conclusions can be drawn from the presented results in Section 4. A significant one is that the accuracy from both the single race prediction and system prediction indicates that the machine learning approach has found a connection between the input data and the finishing results since both perform significantly better than the random baselines.

Each increasing step with the budget showed an unsurprising rise in overall accuracy and the number of systems with five, six, and seven correct races. However, Figure 5.1 shows that despite the increase in the budget, the algorithm's performance did not have a proportional increase in any of the two experiments. The conclusion that can be drawn from this is that the algorithm efficiency will presumably decrease as the budget increases.

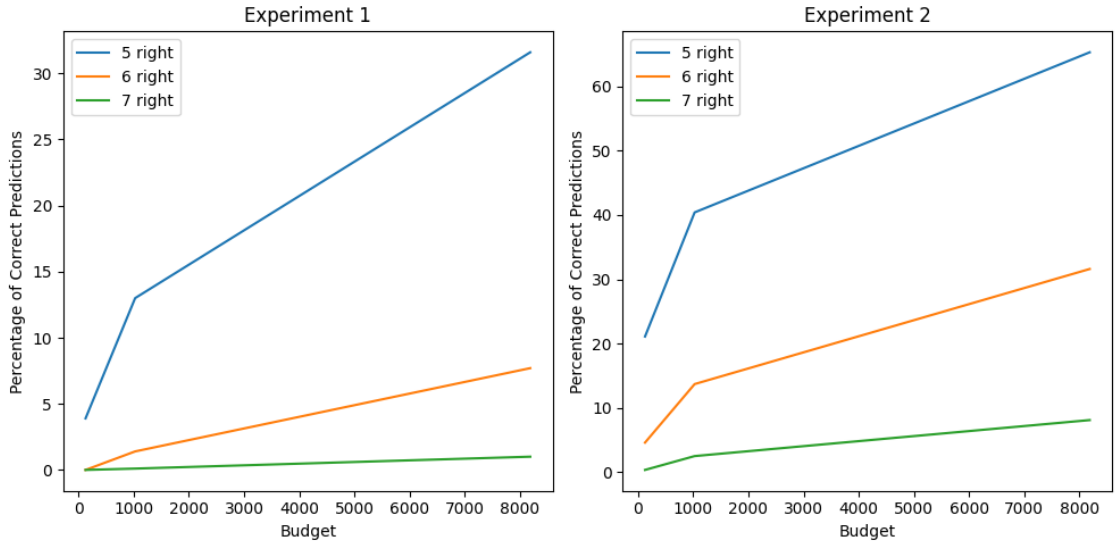

Figure 5.1

Another significant conclusion that can be drawn is that normalizing the data had a considerable performance boost. The top two graphs in Figure 5.2 show the accuracy between the two experiments, and the two bottom shows how much the accuracy

increased between the two budgets. Experiment 2 considerably outperforms Experiment 1. It even does this with a one-level lower budget in every case, which is extremely impressive considering the jump between budgets 2 and 3. The disproportion between the accuracy and the budget can again be seen in these figures, strengthening the presumption that efficiency decreases as the budget increase.

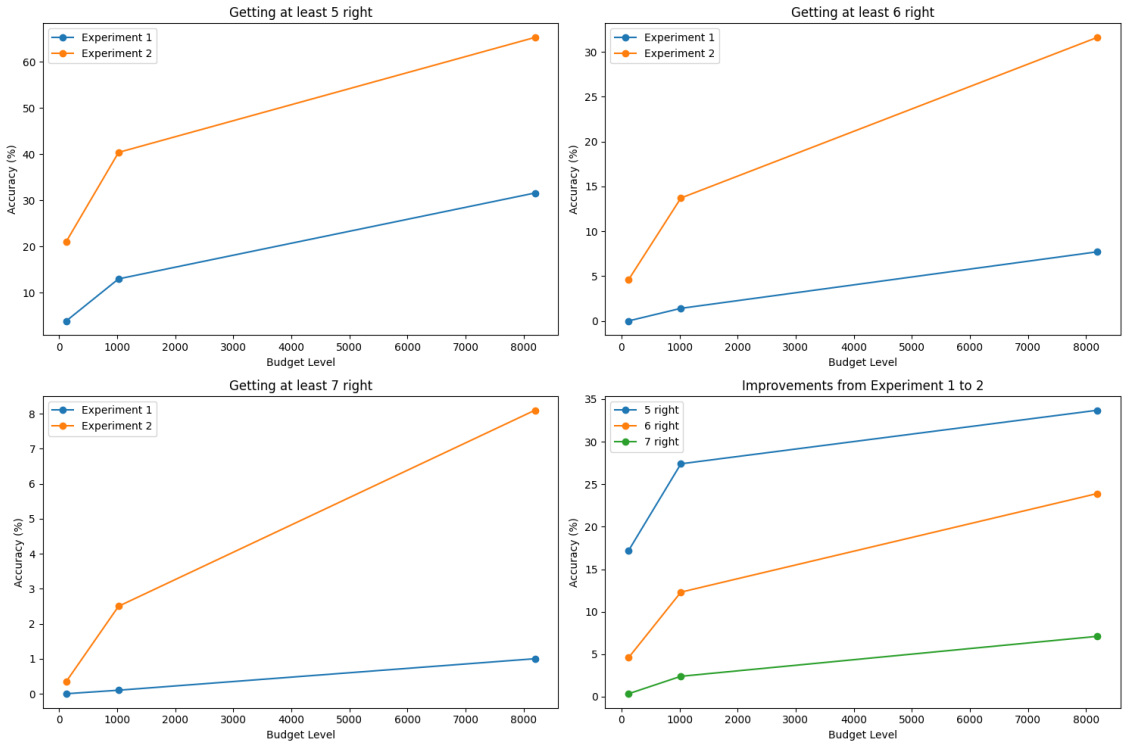

Figure 5.2

Forcing the algorithm to only choose one horse in one to four races showed no measurable improvement in any case, which is interesting, as that is a common strategy among human players. This result suggests that it is too uncertain only to pick one horse if the data does not support it, even if it allows for more horses in other races.

However, the result shows that the machine learning model has found a pattern between the data and the finishing result, indicating that solving Multi-objective optimization by Machine Learning is possible; however, a significant part of the project has been spent on data collection and processing showing that data quality and quantity are paramount. For this approach to be applicable to other problems, such as stock market or traffic management, sufficient high-quality data must be available.

5.2 Future work

One of the most obvious things to do is to use the existing V75 systems instead of combining seven random races when evaluating the system picker. This implementation should be straightforward but requires changes to the scraper and a new process that connects the related races.

Doing this should give a better indication of how well the system actually performs. It should also be an excellent way to determine if combining seven random is a good way to evaluate the system pickers by analyzing if their performance is similar. However, there are a few drawbacks with only using actual V75 races; since there is roughly one every week, there are only 52 every year, so having 200, as with the random one, would require scraping roughly four years back. There is also the issue of not having any of these V75 races in the training data set. In the random system picker, this was solved by only taking from the test data set; however, this is not possible, seeing how small the test set is. One solution is to remove the V75 races that will be used to test the system picker from the train data set; however, there is the issue of testing on older races than those used for training. These drawbacks presumably mean that a combination of random and actual V75 races is optimal.

One way to address many issues with only using V75 is to include more systems than V75, such as V64, V65, and V86. Including these will increase the number of systems to test; however, there is the issue that these systems have different costs and payouts at different numbers of correct races. So presumably, some distinctions need to be made in the system evaluation to separate them.

With the addition of using real systems to evaluate, new metrics can also be calculated. One of the most interesting ones would be to measure the payout and profit/loss. This metric gives a significantly better indication of how well the picker performs since payout can vary greatly depending on what horses won. For example, a system with five rights can have a bigger payout than a system with seven correct, or a system with seven rights can have a payout 10000 times more than a five rights system.

With these new metrics, the most optimal budget can be calculated. This would be the budget that gives the most profit or minor loss.

Further future work could be additional feature engineering. Currently, feature engineering is rather basic and only calculates the average position in the last five races and similar data. However, more advanced data could be calculated, such as previous performance on the same track, starting position, driver, and much more. Performance could be defined by average speed, position, or win percentage. For instance, new data points could include average position on the current track, win percentage on the current track, and average time on the current track. With these new features, some feature selection could also be performed to only include the beneficial features, not those that dilute the data.

Something also prevalent throughout this process was that increasing the size of the training data set increased performance. So finding the optimal size of the

training data set where the performance increases subside would be interesting but time-consuming.

Lastly, more picking algorithms could be tested as well. The current greedy one is very simple and the most obvious one. For example, a similar one could check if taking the best horse or the second and third best horse is cheaper, and this would increase the number of good horses it takes. Presumably, most additional picking algorithms will be more advanced variants of the basic greedy one.

5.3 Conclusion

To conclude, this thises explores the possibility of Multi-objective optimization by Machine Learning and, more specifically, analyzes this by using horse racing. Somewhat encouraging results were found since the machine learning model discovered a connection between the data and the finishing position. This performance resulted in an accuracy rate notably better than the random baseline and slightly better than the more advanced baselines. As for system prediction, the result was still good but did not beat the odds baseline. However, more research analyzing what races the predictor got right is important, and adding the profit metric.

The strategy of only choosing one horse, commonly employed by human players, did not yield better results suggesting that the ML strategy differs from the traditional human ones.

Future work was also discussed, suggesting further system improvements and addressing challenges and limitations. Work such as expanding feature engineering and selection and testing new picking algorithms.

In summary, this study opens up new opportunities for applying machine learning to multi-objective optimization problems and lays the basis for future exploration.

Bibliography

[1] B. R., *The Jockey Club and Its Founders: In Three Periods*. Smith, Elder, 1891. [Online]. Available: https://books.google.no/books?id=SRBDAAAAIAAJ.

[2] Brett, Kate McKay, *How to bet on the ponies*, https://www.artofmanliness.com/living/games-tricks/how-to-bet-on-horses/, Accessed: 2022-12-12, 2022.

[3] https://www.atg.se/spel/2022-12-17/V75/romme, Accessed: 2022-12-12, 2022.

[4] N. Kühl, M. Goutier, L. Baier, C. Wolff, and D. Martin, *Human vs. supervised machine learning: Who learns patterns faster?* 2020. DOI: 10.48550/ARXIV.2012.03661. [Online]. Available: https://arxiv.org/abs/2012.03661.

[5] https://www.atg.se/V75/om-v75, Accessed: 2022-12-12, 2022.

[6] T. Clark, "The use of artificial intelligence in horse racing: Predictive analytics for better performance," *365 Retail*, Mar. 2023. [Online]. Available: https://365retail.co.uk/the-use-of-artificial-intelligence-in-horse-racing-predictive-analytics-for-better-performance/.

[7] I. Ndiaye and K. Cornelis, "Horse racing prediction: A machine learning approach (part 1)," *CodeWorksParis*, May 2021. [Online]. Available: https://medium.com/codeworksparis/horse-racing-prediction-a-machine-learning-approach-part-1-44ed7fca869e.

[8] I. Ndiaye and K. Cornelis, "Horse racing prediction: A machine learning approach," *CodeWorksParis*, May 2021. [Online]. Available: https://medium.com/codeworksparis/horse-racing-prediction-a-machine-learning-approach-part-2-e9f5eb9a92e9.

[9] A. Campbell, "Use machine learning to predict horse racing," *Towards Data Science*, Jun. 2020. [Online]. Available: https://towardsdatascience.com/use-machine-learning-to-predict-horse-racing-4f1111fb6ced.

[10] W.-C. Chung, C.-Y. Chang, and C.-C. Ko, "A svm-based committee machine for prediction of hong kong horse racing," in *2017 10th International Conference on Ubi-media Computing and Workshops (Ubi-Media)*, 2017, pp. 1–4. DOI: 10.1109/UMEDIA.2017.8074091.

[11] P. Borowski, M. Chlebus, *et al.*, *Machine learning in the prediction of flat horse racing results in Poland*. University of Warsaw, Faculty of Economic Sciences, 2021.

[12] Kaggle, *March machine learning mania 2022 - mens*, https://www.kaggle.com/competitions/mens-march-mania-2022, Accessed: 2022-12-12, 2022.

[13] Taylor Hatmaker, *Bings prediction technology is 13-0 with its world cup predictions*, https://www.dailydot.com/debug/bing-world-cup-perfect-record/, Accessed: 2022-12-12, 2022.

[14] IndustryTrends, "The future of sports betting: Ai-powered predictive analytics," *Analytics Insight*, Dec. 2022. [Online]. Available: https://www.analyticsinsight.net/the-future-of-sports-betting-ai-powered-predictive-analytics/.

[15] A. Pretorius and D. A. Parry, "Human decision making and artificial intelligence: A comparison in the domain of sports prediction," in *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, ser. SAICSIT '16, Johannesburg, South Africa: Association for Computing Machinery, 2016, ISBN: 9781450348058. DOI: 10.1145/2987491.2987493. [Online]. Available: https://doi.org/10.1145/2987491.2987493.

[16] L. of Congress. "CSV, Comma Separated Values (RFC 4180)," Digital Preservation. (), [Online]. Available: https://www.loc.gov/preservation/digital/formats/fdd/fdd000323.shtml.

[17] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgo, "Foundations of json schema," in *Proceedings of the 25th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2016, pp. 263–273.

[18] pandas development team, *pandas: Data analysis and manipulation library.* [Online]. Available: https://pandas.pydata.org/.

[19] scikit-learn developers, *train_test_split - scikit-learn.* [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html.

[20] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[21] F. Chollet *et al.* "Keras." (2015), [Online]. Available: https://github.com/fchollet/keras.

A

Appendix 1

A.1 Implementation

A.1.1 Scraper

The scraping class is initialized with only a single value. This value represents the desired number of races that should be acquired and is subsequently used in naming the resulting CSV file. Because of this naming convention, the scraper can first check if an existing file already has this name and, if so, return that file without doing any actual scrapping. This feature saves significant time; however, only the name is checked and not any of the content, meaning that if modifications have been made to the scraper class since creating the file, they will not be represented.

When no previous file exists, the scraper invokes the method "scrape and save". This method oversees the scraping and storing of race data and determines when to stop, and it also does some output to indicate how far along the scraping process is. The code for the method can be seen below. More specifically, the method has a while loop that scrapes, saves, and keeps track of the total number of scraped races. The while loop stops when the total number of races reaches the desired size. The method scrapes an archive page. An archive page can be seen as a list of race IDs in chronological order from new to old, and each page consists of 50 IDs. These IDs are then extracted with the method "Get race ids" and are then used to scrape the individual races in the method "get races," which will be explained in detail later, but in essence, it returns races in a list of race objects. Lastly, these races are saved with the save race function, and the page number is incremented. This feature is crucial since it means the method saves continually throughout the process, and depending on the desired size, the execution time can be multiple hours. Saving like this mitigates the impact of unforeseeable events, such as power or internet outages and outliers in the data that cause errors. Now the different parts of the while loop will be explained in more detail, starting with the scraping of archives.

```
    def _scrape_and_save(self):

        while self.size <= self.goal_size:
            print(str(self.size) + "/" + str(self.goal_size))
            archive = self._scrape_archive_page(page)
```

```
            race_ids = self._get_race_ids(archive)

            races = self._get_races(race_ids)

            self._save_races(races, page)

            page += 1
```

The scrape archive page method uses the request library to generate a post request acquiring the archive page at the given page number. The page and its content are then saved in a text file before being returned. This ensures that no unnecessary post requests are ever performed, and if the same page were to be asked for again, instead of making a request, it would be loaded and returned.

The returned archive page contains much information, and most of it is not of interest to the scraper, and the page is also in a non-desirable format. These problems are what the "get race id" method addresses. This method loops through all the races in the archive page, extracts each race's id using the Json library, and returns them.

These ids are, as previously mentioned, then used in the "get races" methods. In this method, the ids are iterated through and used to scrape the corresponding race to that id. This race scrape contains information about the race and all the participating horses. Similar to the archive, each race scrape is saved, so it is only to be performed once. Another similarity is that the scraped race contains some clutter and is in a non-desirable format. Furthermore, the json library is used again in the "scrape to race" method to turn the scrape into a list of race objects and return them to the previously mentioned "scrape and save" method that then saves them.

The "save races" method is relatively straightforward. It takes the given race objects and turns them into a data frame using the Panda library, and the library is then used again to concatenate the new race data frame with the previously saved races.

After all the desired races have been scraped, extracted, and saved, they are sent to the preprocessing class, which will be explained in detail in the next section

A.1.2 Preprocessor

The DataPreprocsssing class takes four initiating parameters:

- df
- Unprocessed data in raw form
- A name used to save and load the processed data
- A path that states where the processed data should be stored
- A boolean value which indicates whether the data should be normalized

Before any preprocessing, the program checks whether a processed file with the same name already exists. If that is the case, the program assumes that no processing is necessary, loads the located files, and returns them. If no file is found, the program proceeds with preprocessing the given data in df with the method preprocess, which is seen in the code above. This method begins with calculating the lap time of each horse in every race. This calculation is done by taking the horse km_time in seconds and multiplying it with the distance the horse will run in kílometesrs. The distance is not always the same for every horse in the same race. These lap times are then used to calculate the percentage of each horse finishing behind the winner. Doing this conversion makes each race more comparable to the other and makes it easier to determine if a race is close. Both methods are computationally heavy since they iterate through every horse in every race. The calculated lap percentages are then saved as a CSV file named "Y_*name*".

The program then moves on to processing the race-related data. This is the data that should be available before the race and not after. Because of this constraint, the program first removes all data considered cheating, such as finishing order, speed, and whether the horse galloped, essentially the data recorded during or after the race. Then, all data deemed unnecessary is removed. Unnecessary data can, for example, be the page number used during the scraping. A data point like this should not impact the actual race result and would only dilute the dataset. Lastly, a list of custom features/columns is removed. This method was added to ease the implementation of the data points tester and will be discussed in Section 2.5. The complete data frame of x is then returned and saved accordingly.

A.1.3 Validator

The validator takes a data frame of races as an instance variable and has three primary methods and a few smaller helper methods. The crucial methods are "validate_specific", "validate_random", and "_validate_race". Validates_specific takes a list of integers as a parameter representing the indexes of races in the data frames. These are the races that should be validated, and the method now iterates through the integers, gets the race from the data frame, and calls validate_race. Validate random calls validate specific with a list of random integers and are used to remove any bias in the races chosen to be validated. Validate_race takes race as a parameter and is tasked to validate it. The validated race code can be seen below and begins with creating and printing a comment stating the race's id and at what track. The method now starts iterating through every data point in the given race and asking for input on wheater the data is correct. Depending on the input, the comment is updated accordingly, and when all data points have been checked, the comment is returned. The comment is then saved at a given path.

```
def _validate_race(self, race):
        comment = race.id + " at track: " + race.track_id

        for column in race.columns:
            datapoint =  race.iloc[index]
```

```
            index += 1

            print ("in race " + str(id)
            + " the " + column + " is " + str(datapoint))
            input = self._get_input()

            if input == "y":
                comment = comment + " " + column + " correct"
                continue

            elif input == "n":
                comment = comment +  " " + column + " incorrect"
                continue

            elif input == "s":
                print("skiping datapoint")
                continue

            elif input == "sr":
                break

        return comment
```

A.1.4 Evaluator

The Evaluator class has no instance variables, and all necessary inputs are given in the three primary methods. The three methods are Evaluate_predictions, Show, and Save. The Evaluate predictions method takes two parameters, the predicted and correct answers, and is tasked with determining how good the predictions are compared to the correct answers. The method does this by first iterating through every race and every prediction checking if the prediction was correct. In addition to getting the overall correct number of predictions, the correct prediction for each position is also saved. However, the current stored values are only a count and are almost meaningless, so the method now calculates the overall percentages, both the total and by position. The show method can now be called to print and show a bar graph of the different metrics. The save method can also be called with a path, and as the name suggests, it saves both the metrics and the graphs.