Faculty of Fundamental Problems of Technology
Wrocław University of Science and Technology

Analysis of cyclists' training data using Big Data methods

Mateusz Bulanda - Gorol
Index number: 245089

Master's thesis written
under the guidance of
prof. dr hab. inż. Marek Klonowski

Politechnika
Wrocławska

Wrocław 2022

To my Dad,
who passed on my passion and shaped my character.

Your son, Mateusz

Contents

1 Introduction **1**
1.1 Main idea 1

2 Data analysis in cycling **3**
2.1 Problem statement 3
2.1.1 Power meter in cycling as a training factor 3
2.1.2 Important parameters 4
2.2 Description of the data 10

3 Data **13**
3.1 Data characterization 13
3.2 Creating DataFrame 16

4 System's implementation **19**
4.1 Packages 19
4.2 Data set 21
4.3 Machine Learning Methods 26
4.4 Clustering 26
4.5 Clustering 28
4.6 Machine Learning Models 33

5 Summary of the results obtained **37**
5.1 Clustering 37
5.2 Other Machine Learning Methods 38

6 Conclusion **41**

Bibliografia **46**

A Appendix **47**
A.1 Adding month column 47
A.2 Crating DataFrame base on activities metadata for one athlete . 47

A.3 Calculating FTP per month 47
A.4 Calculating Power Zones 48
A.5 Time spent in every power zones for every activity 49
A.6 Calculating TSS 50
A.7 Crating DataFrame base on activities metadata for all athletes . 50
A.8 Dreating DataFrame contains monthly data for all of cyclists . . 51
A.9 Calculating change of FTP 53
A.10 Calculating sumarized change of FTP 53
A.11 Calculating Balance 54
A.12 Calculating the average FTP for every athlete 54

Introduction

1.1 Main idea

This work brings together two worlds, the world of cycling and data analysis. Nowadays, they are increasingly meeting and sometimes producing interesting results. This work follows this trend. This chapter discusses its structure.

The next chapter 2 briefly describes the history of innovations in cycling that have led to the generation of very large amounts of data in the sport. Hand in hand with this development is the development of their analysis and understanding, which positively affects the effectiveness of training. Of course, a lot of metrics have been developed, described in the following section, which help make training more accurate, predictable, processed, and therefore repeatable. However, despite these data and knowledge, the process is still inaccurate and extremely difficult to optimize traditionally. This is where modern data science comes to the fore, in particular the analysis methods associated with Big Data. It seems that they can bring cycling training to a next level, making it more accurate and better optimized. This work attempts to be the first step in that direction. Its goal is to build a model that, based on a rider's training data, can predict how his form will behave.

To achieve this goal, in the third chapter 3, a basic analysis of the riders' training data, derived from the Golden Cheetah Open Data collection [28], was performed. Then the process of creating the dataset that was to be finally used to model the rider's change in form was described. At this stage, it was known that the parameter that best depicts the rider's training status is FTP, so it was based on this parameter that data preparation proceeded. When the data were ready for processing, an attempt was made to cluster them in terms of trying to divide the riders into those who train effectively and those who record poor training results.

In the next part of the work 4, it was shown how the whole system was implemented. First, libraries and packages used at all stages of the project

were shown. Later, it was shown how a dataset was created from tens of thousands of files corresponding to individual activities. Later, the implementation of the K-Mean algorithm for clustering of data was shown. The results of this clustering were presented graphically. However, it turned out to be disappointing and gave the first indication that there might have been some problems in the earlier process. Next, four machine learning models were built to predict the change in the FTP parameter after each training month. The final item was a discussion of the implementation of four machine learning models - Decision Tree Regressor, Random Forest Regressor, Gradient Boosting Regressor, and Sequential Model. Each of them was unable to learn the prepared data and obtained a high error in its prediction.

Next, the results of clustering and prediction using machine learning models 5 were discussed. Large discrepancies were discussed, and the entire process was analyzed to find the source of the error. Several hypotheses have been made on the subject.

The last part 6 summarizes the whole work and concludes it. A possible direction for further research and development of this topic was also outlined given the growing interest in the use of Big Data methods in cycling.

Data analysis in cycling

2.1 Problem statement

2.1.1 Power meter in cycling as a training factor

Cycling has been closely related to the development of technology since the beginning of its history. No wonder cyclists have been fighting for years on very demanding routes of several hundred kilometers long, and at the finish line seconds are often decisive. Therefore, every dozen or so years there is a "technological revolution", which provides cycling with a new technology that can raise the level of the riders who use it. Not surprisingly, it very quickly becomes commonplace among top cyclists who cannot afford to fall behind their rivals. Ambitious amateurs, too, are following in the footsteps of the pros by adopting this type of technology. What is not to be done out of passion? Thus, at the end of the twentieth century, for example, carbon fiber appeared in cycling and began to be used to build ever lighter, stiffer, and more aerodynamic bicycles. Another revolution took place at the turn of the century. At that time, the first power measurements of cyclists appeared. At first, such devices were used only by the top riders, but very soon they became common among professionals. They completely changed the way they trained. They meant that large amounts of data were generated during each activity. From now on, it was possible to plan and conduct training much more precisely and measurably assess the level of the athlete. Since then, the sporting level of cycling has reached a higher level [20]. In recent years, a new generation of riders has entered the professional peloton, including riders such as Tadej Pogacar, Wout van Aert, and Mathieu van der Poel. They are characterized by remarkable results for their young age. Notably, just a decade or so ago there were no such young riders who were regularly successful. What these riders have in common is that they are the first generation who have been training with power measurements since childhood. This has made their sporting development very well managed and allowed for spectacular results.

History shows that power measurement brings a new quality to cycling training. It allows the cyclist to accurately record and monitor the intensity of effort and the length of effort at a certain intensity. On this basis, the effect of training in terms of increased form can be clearly determined. Of course, over the years since the advent of power measurement, training knowledge has developed and many parameters have emerged that are extremely helpful in training.

2.1.2 Important parameters

Fuctional Treshold Power (FTP)

The best-known and most widely used parameter by amateurs is FTP. Hunter Allen and Andrew Coggan in their book, 'Training and Racing with a Power Meter', write that *FTP is the highest power that a rider can maintain in a quasi-steady state for approximately one hour without fatiguing. When power exceeds FTP, fatigue will occur much sooner, while power below FTP can be maintained considerably longer* [11]. An extremely important point in endurance sports is the boundary between aerobic and anaerobic effort. FTP is the power that corresponds to this point. Thus, below FTP, in the aerobic zone, an athlete can sustain an effort for a long time, but when he increases the generated power above FTP and thus enters the anaerobic zone, he will be able to sustain such an effort for several, in the case of well-trained cyclists for several minutes. This is a direct result of the dependence of the concentration of lactate in the blood on the intensity of the effort (in our case expressed by the power generated by the cyclist).

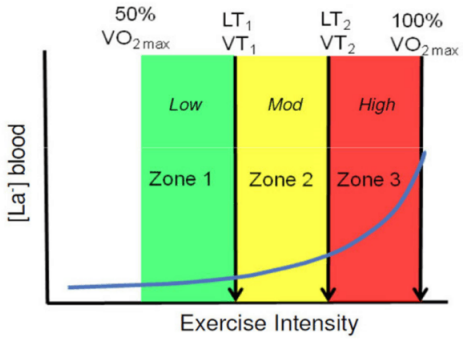


Figure 2.1: Dependence of blood lactate concentration and workout intensity. Source: [33].

Below FTP, the concentration increases linearly with increasing intensity,

but above FTP, the relationship becomes exponential, and the body can no longer cope with the removal of lactate. The time spent at the FTP point can vary depending on the rider's level of training and can also depend on the phase in the training cycle the cyclist is currently in. The time is said to vary from 40 to 70 minutes. However, it is usually averaged out to 60 minutes. Thus, it can be said that FTP is approximately the maximum power that a cyclist is able to sustain for about an hour.

So, one can see how important FTP is for a cyclist. During a race, which usually lasts several hours, the rider knows where the limit value is, which he should not approach for most of the duration of the competition so that at the crucial moment he still has a reserve of energy to enter the zones of anaerobic effort. However, this parameter is useful not only during the competition, but also during training. Based on this value, training zones are determined 2.2. Subsequently, a training plan is created so that the athlete spends a certain amount of time in specific zones (at specific intensities), depending on the goal of training for a given period. A large part of the training is devoted to improving FTP, due to its key role in the athlete's high performance. This plan is usually laid out by the trainer based on his knowledge and experience and is modified as needed during implementation. There are times when the volume is too high and overtraining of the rider will occur. In this situation, it is necessary to take a step back from high-intensity training and rest. This, of course, causes a decline in form. Similarly, in the other direction, scheduled training may be too weak a stimulus for the rider, and thus his potential will not be fully realized. This is undoubtedly a problem, and there is clearly room here to exploit the potential of using the Big Data tool to optimize cycling training. This is the goal that guided the creation of this work.

Level	Name/Purpose	% of Threshold Power	% of Threshold Heart Rate	RPE	Time
1	Active Recovery	≤ 55%	≤ 68%	< 2	70 -80 years
2	Endurance	56 - 75%	69 - 83%	2 - 3	2.5 hours to 14 days
3	Tempo	76 - 90 %	84 - 94%	3 - 4	2.5 - 8 hours
4	Lactate Threshold	91 - 105%	95 - 105%	4 - 5	10 - 60 minutes
5	VO^2 Max	106 - 120%	> 106%	6 - 7	3 -8 minutes
6	Anaerobic Capacity	121 - 150%	N/A	> 7	30 seconds - 2 minutes
7	Neuromuscular Power	N/A	N/A	MAX	5 -15 seconds

Figure 2.2: Power Zones. Source: [25].

In recent years, many methods have been developed to determine FTP, ranging from laboratory studies to simple home tests. The most popular method, which combines relatively high accuracy and does not involve all of the strength of the athlete, and thus does not expose him to overtraining (combined with training on consecutive days, of course), is the 20-minute test. It consists of performing a warm-up with a five-minute accent of very strong riding (above the assumed FTP), and then a five-minute rest is followed by the 20-minute main part of the test, during which the cyclist maintains a constant, highest possible power for such a time. The average power of the 20-minute section is inserted into the formula:

$$\text{FTP} = 0{,}95 \cdot \text{result of the test.} \tag{2.1}$$

This value is considered sufficiently accurate for most cases. It is important to perform the test according to the same protocol each time.

Time To Exhaustion (TTE)

However, FTP is not the only parameter that affects the form of an athlete. Slightly above, we mentioned that the duration of the threshold effort can vary from 40 to 70 minutes. This, of course, depends on the level of training. Therefore, another important parameter from the point of view of an endurance athlete will be the TTE, that is, the maximum threshold effort time for a given athlete. Since the athlete rides in the vicinity of FTP at key moments of the race, the TTE parameter will determine his endurance in this range. This will be another component of the rider's form and another important element that the rider will work on during training [14].

Mean Maximal Power (MMP) and Critical Power (CP)

MMP is the power one are able to maintain in a certain amount of time. The MMP model contains a lot of information about a particular athlete. It allows to determine how much energy can be released from which energy system in a unit of time. The human body has three energy systems: aerobic, anaerobic, and creatine phosphate. The aerobic system releases energy using oxygen, where fats and carbohydrates are burned. This is a relatively slow process, but at the same time it can be sustained for a long time. The anaerobic system is good at releasing energy fairly quickly. It does this by anaerobically burning the glycogen stored in the muscles. However, it causes

an increase in muscle acidification (a greater amount of H+ ions accumulate in the muscles), which in turn leads to muscle weakness, so this system is suitable for short and intense efforts. The creatine phosphate system is sometimes called a sprinting system. It allows to immediately use the energy stored in the muscles and generate very high power, which is only possible to sustain for several seconds [9]. Of course, all these systems work simultaneously, but depending on the intensity of the effort, one of them will be dominant. Using the MMP for successive time intervals, we can draw a power curve.

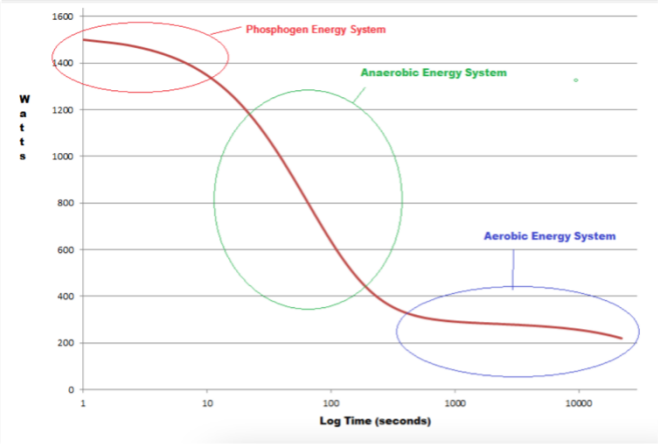


Figure 2.3: Three energetic systems on power curve. Source: [9].

When we look at the part of the graph corresponding to prolonged activities, we see that it clearly flattens out and asymptotically tends to a certain value. We call this value Critical Power and it is embedded in the aerobic system [19]. If we plotted the FTP value on the graph, we would see that they lie close to each other. Moreover, these values are correlated with each other to a certain extent, an increase in one causes an improvement in the other, so it happens that these quantities are sometimes mistakenly equated with each other. However, in reality, they will never be equal.

The Power Curve provides us with very important data about the rider. This curve has a similar shape for each rider, but by looking at this graph, we can make a characterization of the rider. For example, if we see very high values in the area of the creatine phosphate system and at the same time the graph slopes steeply downward in the area of the anaerobic system, we can be almost certain that we are dealing with a sprinter whose riding characteristic is that he can produce very high power in a short time. In another case, we may have a rider whose curve appears to be flattened, the

maximum values are relatively low, and at the same time, the asymptotic flattening takes place for high values of generated power. In this case, we are likely to have a triathlete who will be characterized by high CP values that allow him to ride relatively fast over very long distances. These are just examples of riders' characteristics that serve to better illustrate how much information we can read from the power curve. Of course, there are more types of rider and their characteristics that we can find on this curve. It is worth noting at this point that cycling training is really about increasing any of the areas of the power curve, depending on the fitness the rider is currently working on. In a more mathematical sense, we could say that any training is aimed at increasing the area under this graph, and thus increasing the power curve integral. Perhaps by dividing this graph into fragments and monitoring changes in the field of the separated part, we can somehow assess the cycler's increase or decrease in the form of a particular element of his overall fitness.

VO2max

Another important parameter is the oxygen ceiling or VO2max. This is an indicator of our body's maximum capacity to take in oxygen per minute, sometimes divided by kilograms of body weight. This is one of the most important performance parameters for any athlete, and the high values of this indicator characterize the best cyclists in the world. Professionals will determine it during laboratory tests, but there is also a "home" way, intended mainly for amateurs. It consists of determining the maximum constant power over about 5 minutes. We insert the average power obtained from such an episode into the following formula along with our current body weight. This formula was proposed by the American College of Sports Medicine and should work well for most in estimating Vo2Max. Here is our formula [24]:

$$VO2max[L/min] = 0{,}0108 \cdot \text{test power} + 0{,}007 \cdot \text{body weight}. \tag{2.2}$$

Normalized Power (NP)

The simplest measure of training intensity is the average power, which interpretation does not need to be explained to anyone. Unfortunately, it will not tell us very accurately the effort of our body caused by training. With our help comes NP, which is the weighted average power of workout [17]. To determine this magnitude, we need to calculate the rolling average of the

power for every 30 seconds of activity. Then raise each of the resulting values to the fourth power. Finally, we take the root of the fourth degree from the average of the results obtained [23]. An example would be a constant running at 200W for an hour. In this case, our average power and NP will be very close to each other and will oscillate around 200W. The situation changes when we spin 100W for 20 minutes, for 20 minutes 200W, and for the last 20 minutes 300W. In this case, the average power will also be 200W, but the NP will be about 240W. This kind of exercise will require more effort from our body, which will be reflected in normalized power as a result of the intensity of the ride. As it increases, acidification, glycogen demand, or stress hormone production such as cortisol increases much faster. All of these variables result in a much stronger response of the body to exertion.

Training Stress Score (TSS)

Unit that measures the training load of each unit based on its intensity and duration. It allows us to compare workouts and races with each other. 100 TSS means an hour's ride at the threshold or in the proverbial "deadlift." With an NP of 200W and an FTP threshold of 250W for an hour, we will generate 80 TSS. The NP of the entire workout, the duration of the exercise, and the current FTP threshold are used to calculate the TSS. The aforementioned Andrew Coggan is the creator of this indicator and the formula for it is expressed as follows[16]:

$$TSS = \frac{t \cdot NP \cdot IF}{FTP \cdot 3600} \cdot 100, \tag{2.3}$$

where:

t - time of exercise in seconds,

NP - Normalized Power of activity,

IF - Intensity Factor, which is defined as NP divided by actual FTP,

FTP - Functional Threshold Power.

The TSS factor is extremely important when putting together a short- and long-term training plan, as it allows to get a good idea of what kind of stress an athlete is under. It will be an important factor in considering optimizing cycling training.

2.2 Description of the data

There are many platforms on the market to collect and analyze data generated during cycling training, the most popular of which are Strava [7] and TraingPeaks [8].

However, they do not allow the downloading of user data due to privacy concerns. A large dataset of many cyclists is needed to perform the data analysis. Finding such a collection is not trivial. However, it turned out that the owner and co-creators of GoldenCheetah software decided to make available a dataset containing more than 2 million activities from 6,000 users called GoldenCheetah OpenData. In addition, they have created a Python library to work with these data. The data are grouped by user ID and we can easily retrieve the data of a specific athlete. Of course, not all data are of good quality. First of all, some activities are not cycling, and among those that are, not all of them contain data from the power generated by the cyclist (such a person probably does not use power measurement on the bike). We have to discard this type of data and focus only on those activities that contain data on the power generated by the rider. Each activity is stored in a separate file in the form of second-by-second data of distance, power, heart rate, cadence, and altitude. The file loads by default in pandas. DataFrame format.

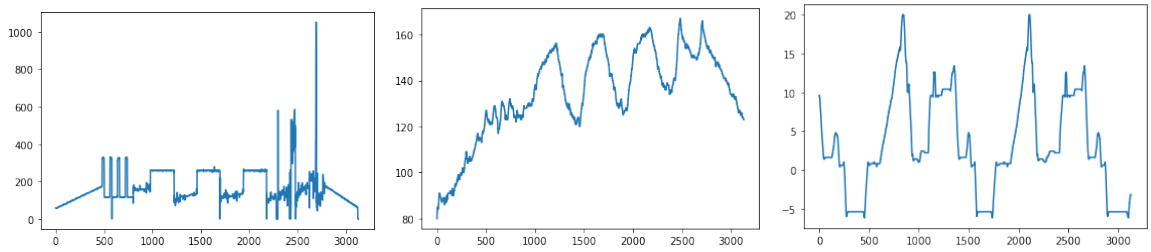


Figure 2.4: Power, heart rate, and altitude over time for random activity.

Additionally, each file has extensive metadata in JSON format. In them, we can find a great many parameters such as TSS, or CP, power distribution, speed distribution, cadence distribution, and average maximum power curve.

There are several types of devices that can measure the power generated by a competitor. They can be built into the rear hub, crank spider, crank arms or pedals. The last two solutions are by far the most popular on the market, mainly because of their relatively affordable price. All devices of this type work in a similar way. They have multiple strain gauges embedded in the

bicycle component on which they are mounted on and measure the stresses on it. The tensile force value is multiplied by the speed at which the element rotates and thus the power value is obtained.

Figure 2.5: Stages [5] power meter on crank arm. Source: [21].

Data

3.1 Data characterization

Let us take a closer look at the data in question. Mark Liversedge, one of the developers of GoldenCheetah software, posted a Python script [27] of his on his blog [28], in which he illustrates the data we are dealing with in a clear way. Using his work, we will perform a basic analysis of the data we will be dealing with based on the charts he created. From our point of view, the most relevant data are the power generated by the rider, because they determine the result of the rider during the competition, characterize his form, and also show the relative effort put into training. So let us take a look at basic power statistics based on a cross section of the entire group of riders to whose data we have access.

Let us look at the distribution of the mean maximum power generated by athletes in 3, 8, 20 and 30 minutes.

One can see that when considering the power from the perspective of a cross section of a large group of athletes, seems to have a normal distribution. We can assume for them with high probability that parameters such as Functional Threshold Power 2.1.2 or Critical Power 2.1.2 will also have a normal distribution. It can be assumed that this is a linear function of independent normal random variables, so it is also normal. Therefore, let us define the Critical Power Ratio as Critical Power divided by the average maximum power of 20 minutes of exercise.

One can see that the above distribution is also a normal distribution, which confirms our thesis. It is worth mentioning at this point that the Critical Power value is estimated and may differ in some cases from the actual figure 2.1.2, however, with a large statistical sample, the accuracy of the estimate is sufficient.

As was said in the first chapter 2.1.2 extremely important for the training athlete is the graph of the dependence of Mean Maximal Power over time. It allows to visualize the growth of the rider's overall form, as well as individual

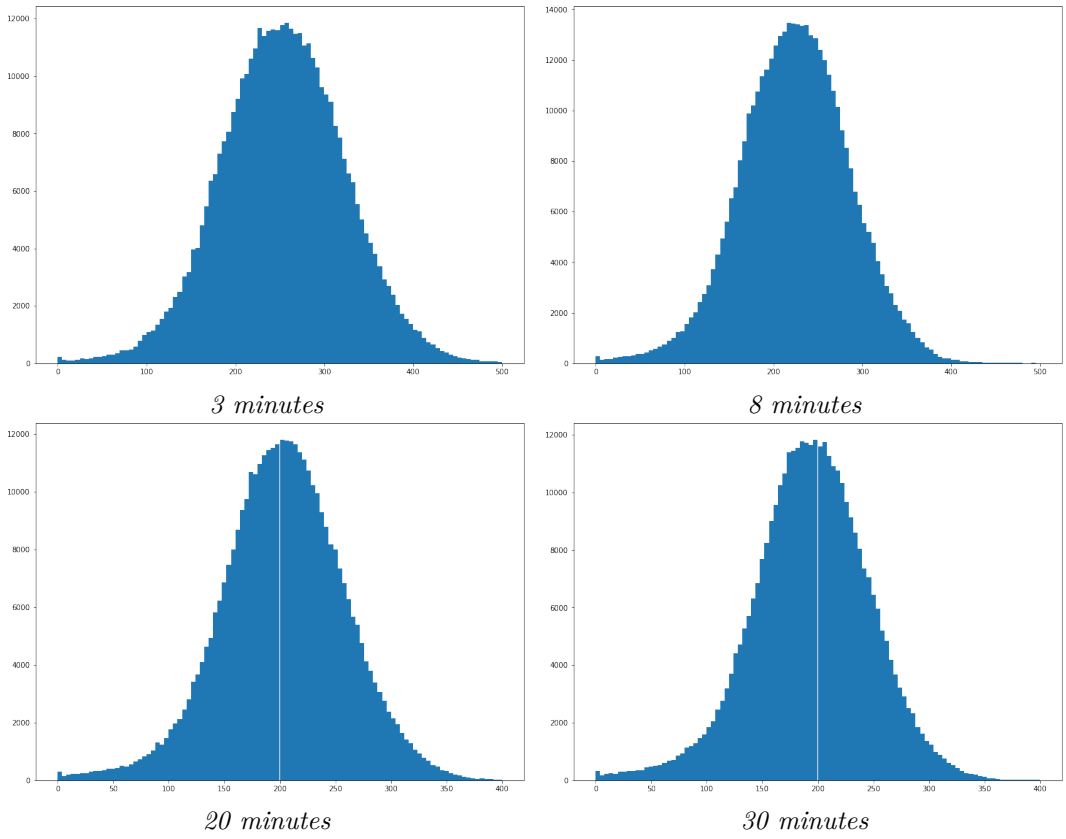


Figure 3.1: Distribution of the mean maximum power generated by athletes during 3, 8, 20 and 30 minutes. Source: [27]

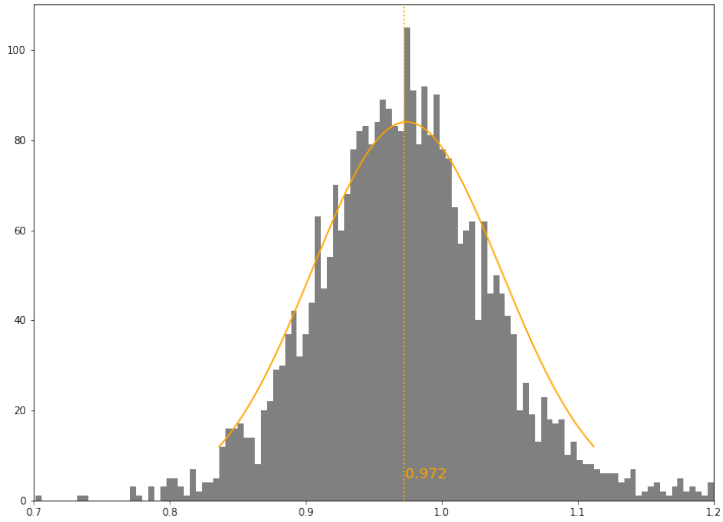


Figure 3.2: Distribution of Critical Power Ratio for athletes. Source: [27]

performances, such as sprinting and time trials. So, let us take a look at what the MMP curves look like for the group of athletes under consideration:

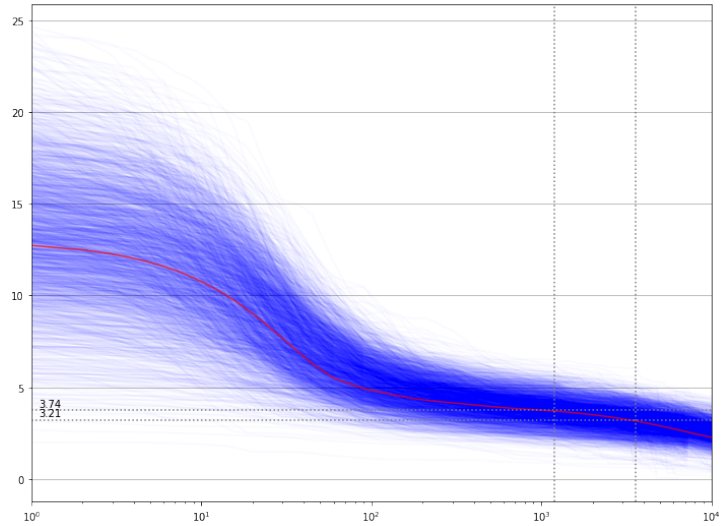


Figure 3.3: Mean Maximal Power Curves for 3000 athletes. Source: [27]

It is clear that the curve for each athlete takes a similar shape, but each is slightly different. This can be seen, especially in the area of short and intense efforts. The conclusion at this point is that each athlete in his training should aim to increase a certain part of this graph 2.1.2.

If we think about cycling training and its effects in the context of preparing a training plan, many variables come to mind on which this effect will depend. So, let us try to look at this problem analytically and see how the form of the rider depends on the training volume, that is, the number of kilometers ridden and the number of activities performed by the cyclist.

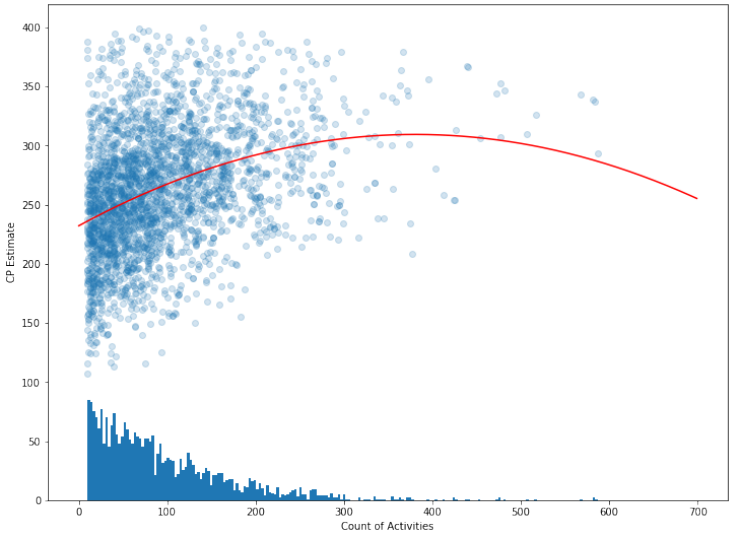


Figure 3.4: Coefficient between estimated CP and count of activities. Source: [27]

In the above relationship, one can find a correlation, albeit a tenuous one, between the number of training sessions and an athlete's Critical Power, but it is not clear whether the higher CP value is due to more activity, or whether the apparent relationship is coincidental and the higher CP value is due to well-planned training that involves regularity, and therefore also results in more activities. However, what about cyclists who do not follow a training plan and do not try to improve their fitness while performing activities frequently? To resolve this, it is not enough just to analyze the value of CP, but to consider how this value changes over time depending on the amount of training.

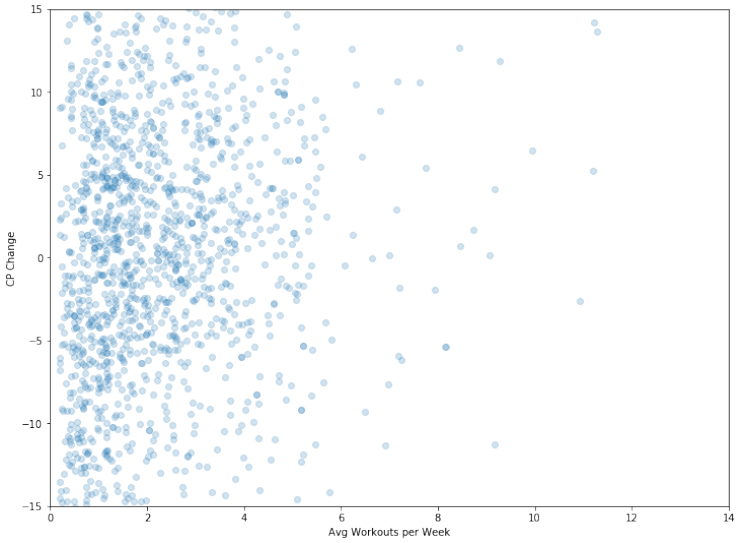


Figure 3.5: Coefficient between change estimated CP and number of workouts per week. Source: [27]

One can see that in this case there is no correlation. Let us see if we can see a correlation between the change in CP and the number of kilometers driven.

We can see that in this case, too, there is no correlation, so we can conclude that training volume alone does not increase the athlete's form. Therefore, training quality is responsible for training effect, and this is what we will discuss in the next chapter.

3.2 Creating DataFrame

DataFrame is the standard and basic data structure used by Pandas for data processing. With the knowledge of cycling training that we want to model with Machine Learning, we need to think about what parameters we will need to predict the changes in the rider's FTP, resulting from the training

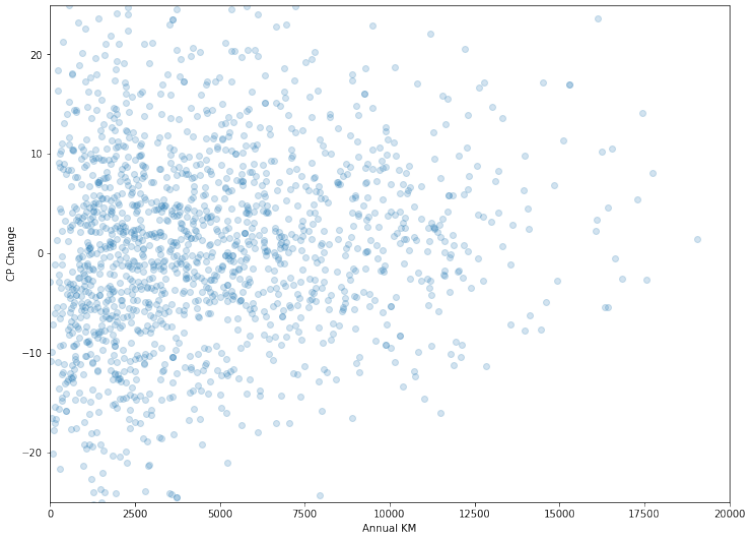


Figure 3.6: Coefficient between change estimated CP and number of kilometers annually. Source: [27]

he has done. This happens because we must first create a large DataFrame in which we collect all the necessary data. It will be in the form of a table, where each row corresponds to a single activity.

The first column collects identification data, that is, the ID of both the athlete and the specific activity. The next column is the date from which we extract the month so that we can measure how the rider's sports level changes over such a time interval. Then we have the weight of the rider, which has a very large impact on the absolute power generated by the rider. The duration of the activity is also important, it determines for us the training volume. In the next column, we have the normalized power, which was mentioned in the previous section 2.1.2. Note that all the parameters mentioned so far were taken directly from the metadata attached to each activity, of which they were only a small part. So, one can see how rich a dataset we are dealing with. With these data collected, it was necessary to determine several more parameters on the basis of them. Among them, the FTP parameter was first counted. According to what is known about it from the previous chapter, it was calculated as 0.95 of the value of the average maximum power of 20 minutes of driving, for each month of 2.1.2. Then, on the basis of this value, the training zones were determined and stored in the following columns as their upper limits. These values are shown in Table 2.2.

With the limits of the power zone determined, more data were added, namely the time the athlete spent in each zone during each workout. It is

worth noting at this point that the limits of the exercise zones change with the change in FTP, so an athlete who is steadily increasing his level of fitness will perhaps spend a similar amount of time in each zone, although if we look at the power, this one will steadily increase. The time spent in each zone will strongly influence the change in the rider's form and will be a key parameter in further analysis.

However, the next-to-last parameter, which is TSS, is also important. This parameter was determined according to the formula 2.3. The value of this parameter will determine the effort that a cyclist puts into training and therefore can illustrate fatigue. This will be of particular importance in the event of possible overtraining.

With the generated main DataFrame, which contains all the activities of interest, we can proceed to create a second DataFrame, which will be the input set for the machine learning model that predicts the change of FTP based on the data in it. First, we need data showing how FTP changes in our dataset to train the machine learning model in question. Traditional conventional training plans are usually based on four-week cycles. After each of these, an FTP test is performed to update the power zones. In the case at hand, a similar approach was adopted and the FTP change was calculated at a monthly interval. As the FTP parameter changed each month, the data that caused the change had to be collected monthly. Therefore, the following columns contain the athlete's weight, the current month, the current FTP, and the ratio of FTP to the athlete's weight. Next, we have the number of activities in a given month and the total activity time in a month. The next columns contain the average normalized power, the total time the athlete spent in each zone, and the sum of all TSS points throughout the month. The resulting dataset, in each row, contains data for one training month for one rider. Having the dataset prepared in this way, one can proceed to build the machine learning model.

System's implementation

4.1 Packages

The Python language was used to handle the data and perform its analysis, and then build a machine learning model to optimize cycling training, along with popular libraries such as:

- NumPy - is used to perform matrix operations. It is the fundamental library for scientific computing. The package consists of multidimensional array objects and a set of routines to process an array and has tools for linear algebra [18].
- Pandas - is one of the most powerful data analysis packages in Python. It is used to read data from files and process them in the form of a DataFrame [30].
- Matplotlib - it is a library for drawing graphs and graphical presentation of data [22].
- OS - allows performing operations for which the operating system is responsible, such as displaying the contents of folders, deleting files, changing permissions, or creating directories [3].
- Shutil - is a supplement to the OS module. Allows for high-level file actions, such as copying or deleting [4].
- Math - contains functions similar to those normally found in a calculator, and some math constants [34].
- Seaborn - a neat and effective library that allows one to quickly create attractive charts in Python. It was built on the basis of the Matplotlib library, and at the same time, enriched with additional types of charts [35].
- Datetime - provides classes for work with a date and time. These classes provide a variety of date, time, and time-frame functions. The date and

time are objects in Python, so when one manipulates them, one is actually manipulating the objects and not a string or timestamps [1].

- dateutil - it is an extension of the standard datetime module [2].
- tqdm - allows one to display the progress bar while the process is running [15].
- IPython - it is an interactive shell for Python with additional syntax elements [32]
- statistics - it is a module that allows to calculate mathematical statistics for numerical data [6]
- TensorFlow - an interface for expressing machine learning algorithms, and an implementation for executing such algorithms. [10]
- scikit-learn - machine learning library for the Python programming language. It includes various classification, regression, and clustering algorithms. It also provides various tools for model fit, data pre-processing, model selection, model evaluation, and many other tools [31].
- XGBoost - it is a collective tree-based machine learning algorithm that uses a gradient-enhancing structure. Artificial neural networks in most cases outperform other frames or algorithms in predicting problems with text, images, and other disordered data [13].

In addition to the standard packages, the GoldenCheetah OpenData library was used. It is used to work remotely with the data described above. Its biggest advantage is that it allows remote connection, and thus does not force us to store the entire database locally. In addition, it has pretty good documentation [29]. Using it, we can access any activity grouped by user ID, or we can analyze all activities of a given user. This is very convenient from our point of view because we focus on specific parameters of the cyclist and their variability. Each of the aforementioned activities is in a separate .csv file, where we can find the time course of distance, power, heart rate, cadence, and elevation, as shown in Fig. 4.1. In addition, each athlete has a JSON file with several metrics that have already been determined based on the collected data. However, it is worth noting that some of them, such as CP, may be questionable because, as a rule, the value of this parameter will vary over time and will depend on the training level of the athlete, while in our data

it is determined of all the data of one athlete. Some of the parameters that we will use for the analysis should be counted for shorter periods to capture their variability.

	secs	km	power	hr	cad	alt
0	0	0.00000	59	80	71	9.6
1	1	0.00205	59	81	75	9.6
2	2	0.00462	59	82	77	9.6
3	3	0.00765	59	83	78	9.4
4	4	0.01111	59	83	78	9.4
...	...	...	...	...	...	...
3126	3126	28.80550	0	123	91	-3.2
3127	3127	28.80560	0	123	0	-3.2
3128	3128	28.80560	0	123	0	-3.2
3129	3129	28.80560	0	123	0	-3.2
3130	3130	28.80560	0	123	0	-3.2

3131 rows × 6 columns

Figure 4.1: Sample activity including distance, power, heart rate, cadence and altitude data.

4.2 Data set

In the previous chapters, it was described what was the concept behind the creation of the dataset. In this chapter, one will look at the practical implementation of this idea and the limitations that arise during it. In the first part of the work, all operations were performed for a single athlete, not to waste time creating functions that count the various parameters for all athletes. Remote data handling also seemed reasonable at this stage. After in-depth analysis of the single-activity data set, it turned out that most of the parameters needed are in the metadata. Furthermore, the Goldencheetach OpenData library allows for very simple and intuitive access to these data [12]. After installing the library, a list of all athletes was created:

```
1    athletes = list(od.remote_athletes())
```

Next, a sample athlete was selected, a list of his activities was created, and both athlete and activity metadata were displayed, as well as graphs of various parameters during the activity:

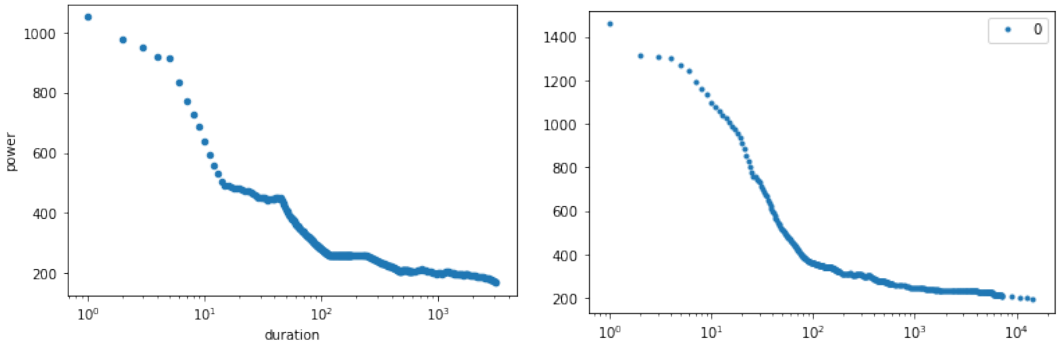
```
    athlete = athletes[448]
    activities = list(athlete.activities())
```

Then all of that rider's activities were filtered so that only cycling activities that have power data remain:

```
bike_activities = list()
for activity in activities:
    if 'power_mmp_secs' in activity.metadata['METRICS']:
        bike_activities.append(activity)
len(bike_activities)
```

and a curve of the average maximum power of the athlete was drawn based on all cycling activities:

```
    for activity in bike_activities:
        df = pd.DataFrame({'duration':activity.metadata['METRICS']['power_mmp_secs'], 'power
    ':activity.metadata['METRICS']['power_mmp']})
        df.plot('duration', 'power', kind='scatter', logx=True)
```



activity number 154 *all activities*

Figure 4.2: Power curves for athlete number 448

After this initial preparation and analysis, the first DataFrame was created based on existing data. To this end, two Pandas Series are created for each activity. The first collects the mean maximum power data, the second contains the remaining data, described in the previous chapter 3.2. The two series are then combined and added as a row to the DataFrame. Finally, a column with the month is added (the implementation of this function is shown in Appendix A.2):

```
    PowerDF = create_dataset(bike_activities)
    PowerDF
```

The process of creating this data set was described in the previous chapter 3.2. Then, functions were created to determine other important parameters.

- FTP per month

The FTP Determination Function looks at the DataFrame that has already been created and calculates the maximum MMP value for each month and then calculates 95% of that value. Then, this value is added to the previously made list as many times as the analyzed athlete has been active in a given month (the implementation of this function is shown in Appendix A.3):

```
PowerDF['ftp'] = calc_ftp(PowerDF)
PowerDF
```

- Power Zones

Power Zones are calculated as a percentage of the FTP value. In this form, they are added to lists, which are then added as subsequent columns to the DataFrame being built (the implementation of this function is shown in Appendix A.4):

```
PowerDF['1_power_zone'], PowerDF['2_power_zone'], PowerDF['3_power_zone'], PowerDF['4_power_zone'], PowerDF['5_power_zone'], PowerDF['6_power_zone'] = calc_power_zones(PowerDF)
PowerDF
```

- Time spent in every power zones for every activity

The function that counts the time spent in each zone analyzes subsequent files with data related to each activity. Power data for each second are classified into the appropriate power zone, and the number of seconds that the competitor spends in a given zone is increased by one. After analyzing all activities, the time spent in each of the zones is added to the lists, which, after analyzing all activities, are added as subsequent columns to the dataset (the implementation of this function is shown in Appendix A.5):

```
PowerDF['time_in_zone1'], PowerDF['time_in_zone2'], PowerDF['time_in_zone3'], PowerDF['time_in_zone4'], PowerDF['time_in_zone5'], PowerDF['time_in_zone6'], PowerDF['time_in_zone7'] = time_in_zones(PowerDF)
PowerDF
```

- TSS

The value of the TSS coefficient is calculated for each activity in the dataset (the implementation of this function is shown in Appendix A.6) according to the formula described in the previous section 2.1.2.

```
PowerDF['tss'] = calc_tss(PowerDF)
PowerDF
```

In this way, the final DataFrame for an one athlete was created. As one can see, the functions counting FTP, Power Zones, Time in Zones, and TSS have

already been written in such a way that they can handle a dataset with more than one athlete. However, this approach proved unsuccessful when trying to build a large dataset with multiple athletes. For many users, remote access to the data turned out to be much too slow. Therefore, the decision was made to download the data that will be used in further analysis, i.e., the data of users whose main sport is cycling. Among the data collected, there were athletes with a lot of activity that took up a lot of space. It turned out that these people had many running and swimming activities, which indicated that we are dealing with triathletes. Unfortunately, the GoldenCheetach OpenData library does not allow to selectively download activities, one needs to download all activities of a given athlete. Therefore, it was necessary to narrow down the area of interesting athletes. Therefore, participants who have a minimum of 200 activities, of which no more than 10 are running and swimming activities, were considered. Initially, this aroused some controversy because one removed all triathletes from the data set, but after further reflection, it comes to the conclusion that both running and swimming affect the athlete's fatigue, and considering them only in terms of cycling activities, false results would be obtained. For triathletes, a separate model would have to be created, taking into account the effort they put into the other two sports to properly optimize their training.

After these restrictions were imposed, 300 athletes were downloaded and the dataset could be built. A similar, though slightly modified, function was used to generate the basic dataset as for the single-athlete dataset (the implementation of this function is shown in Appendix A.7):

```
1    PowerDF = create_dataframe()
2    PowerDF
```

The functions to generate FTP, Power Zones, Time spent in Power Zones, and TSS remained unchanged. After some time, the final DataFrame is ready.

The resulting DataFrame contains 157,416 rows. The next step was to create a second DataFrame that contains data from each training month of each rider. Therefore, data on the number of activities, the training time, the time spent in individual power zones and the TSS were summed for each month. In addition, the normalized power was averaged. The athlete's id, weight, month and current FTP are prescribed for each month. All of these quantities are added to the lists, which at the end become the columns of the new DataFrame (the implementation of this function is shown in Appendix A.8):

activity	athlete	date	weight	duration	np	power_1200s	month	ftp	1_power_zone	...	5_power_zone	6_power_zone	time_in_zone1	time_in_zone2	time_in_zone3	time_in_zone4	time_in_zone5	time_in_zone6	time_in_zone7	tss
4_23.csv	be3b662e-6fda-4c4f-85f8-812a364657f7	2018/02/14 03:54:23 UTC	55.0	13961.0	145.25988	152.0	2	238.45	131.1475	...	286.14	357.675	5701	1612	798	467	218	357	92	143.916548
5_37.csv	be3b662e-6fda-4c4f-85f8-812a364657f7	2018/02/15 01:55:37 UTC	55.0	18214.0	134.91934	165.0	2	238.45	131.1475	...	286.14	357.675	8755	2651	952	661	259	365	144	161.978210
7_08.csv	be3b662e-6fda-4c4f-85f8-812a364657f7	2018/02/16 02:27:08 UTC	55.0	8921.0	161.78067	169.0	2	238.45	131.1475	...	286.14	357.675	3757	991	434	498	243	374	159	114.069528
11_17.csv	be3b662e-6fda-4c4f-85f8-812a364657f7	2018/02/17 00:11:17 UTC	55.0	10115.0	168.34300	202.0	2	238.45	131.1475	...	286.14	357.675	3607	1446	1232	1148	794	379	77	140.042178
9_10.csv	be3b662e-6fda-4c4f-85f8-812a364657f7	2018/02/17 21:29:10 UTC	55.0	3286.0	134.67234	174.0	2	238.45	131.1475	...	286.14	357.675	1393	331	453	310	105	35	4	29.115699
3_07.csv	be3b662e-6fda-4c4f-85f8-812a364657f7	2018/02/17 23:23:07 UTC	55.0	12034.0	169.59874	237.0	2	238.45	131.1475	...	286.14	357.675	4033	1736	953	507	342	458	602	169.105641
5_49.csv	be3b662e-6fda-4c4f-85f8-812a364657f7	2018/02/18 21:35:49 UTC	55.0	15666.0	163.08889	251.0	2	238.45	131.1475	...	286.14	357.675	5918	2419	1546	1195	833	771	164	203.568099

Figure 4.3: Main DataFrame contains all activities.

```
MonthlyDF = all_month_data(PowerDF)
MonthlyDF.head(100)
```

Then four columns were added: absolute change in the FTP value, percent change in the FTP value, change in the FTP value per kilogram of body weight and percent change in the FTP value per kilogram of body weight. They are calculated as the difference between the FTP for the next month and the current month (the implementation of this function is shown in Appendix A.9):

```
MonthlyDF['d_ftp'], MonthlyDF['d_ftp_percent'], MonthlyDF['d_weighted_ftp'], MonthlyDF['
d_weighted_ftp_percent'] = ftp_change(MonthlyDF)
MonthlyDF.head(100)
```

After these operations, it was still necessary to clean up the dataset that could contaminate the learning process. Therefore, one can proceed to delete rows with NaN values (not a number) values and those where the change in the FTP value was very large because it is not possible in real cases. Therefore, all lines where the FTP increased by more than 15% or decreased by more than 30% have been deleted.

```
for i, row in tqdm(FinalDF.iterrows()):
    if(row['time_in_zone1']==0 and row['time_in_zone2']==0 and row['time_in_zone3']==0
and                row['time_in_zone4']==0 and row['time_in_zone5']==0 and row['
time_in_zone6']==0 and              row['time_in_zone7']==0):
            FinalDF = FinalDF.drop(i)
```

```
for i, row in tqdm(FinalDF.iterrows()):
    if(row['d_ftp_percent'] > 0.15 or row['d_ftp_percent'] < -0.3):
        FinalDF = FinalDF.drop(i)
```

The dataset generated this way contains 8192 rows and looks as follows:

eight	month	ftp	weighted_ftp	no_activities	duration	avg_np	time_in_zone1	time_in_zone2	time_in_zone3	time_in_zone4	time_in_zone5	time_in_zone6	time_in_zone7	tss	d_ftp	d_ftp_percent	d_weighted_ftp	d_weighted_ftp_percent
55.0	3	238.45	4.335455	25	392939	149.600000	147356	45403	35812	27714	16746	12446	3589	4462.430228	0.00	0.000000	0.000000	0.000000
55.0	4	241.30	4.387273	23	325440	178.913043	70736	49431	46890	43003	22698	10385	1858	4846.726559	2.85	0.011811	0.051818	0.011811
55.0	5	247.00	4.490909	29	385962	174.137931	96111	64222	53790	37444	19027	10893	2444	5055.405705	5.70	0.023077	0.103636	0.023077
55.0	6	242.25	4.404545	30	402734	168.233333	96742	64041	62602	43589	19607	9950	3265	5462.627962	-4.75	-0.019608	-0.086364	-0.019608
55.0	7	253.65	4.611818	28	375466	169.000000	102148	77232	54253	32138	14716	6186	1604	4523.859755	11.40	0.044944	0.207273	0.044944

Figure 4.4: Final DataFrame contains all monthly data for all athletes.

4.3 Machine Learning Methods

4.4 Clustering

To better show the structure of the discussed data, clustering was performed. The idea behind this was: 'Can one clearly distinguish between athletes who train well and those who train badly?' To verify this, three parameters were calculated for each rider. The first is a complete change to the FTP parameter, which will show whether the athlete's form has increased throughout the training period or not. The second is the `balance`, which was defined as follows: All training months of the athlete were analyzed when in a given month, his FTP increased, the `balance` increased by one, otherwise, it decreased by one. The third parameter is the average FTP per kilogram of body weight during the entire training period. It allows us to estimate whether we are dealing with a generally well-trained athlete or not.

Based on these three parameters, the clustering was performed using the well-known K-Means [26] algorithm from the study. Clustering attempts were made using various combinations of the parameters described above, first using all three and then using two of the three in all combinations.

To properly prepare a machine learning models, which are able to predict change of FTP, it is necessary first to determine which data will be the input to the model and which data the model is supposed to predict. Columns with the following values were set as input data: weight, current FTP, number of activities, sum of the duration of all activities, average normalized power, sum of time spent in each zone, and sum of TSS points. Meanwhile, the modeled value was the change in the FTP parameter after each month of training. Finally, the data were divided into a training set and a test set.

Decision Tree Regressor

To compare the results obtained, four machine learning models were created. The first of these is the Decision Tree Regressor. It is a basic, relatively simple, but universal model that will be a good reference for other models.

Decision Tree has some limitations which can cause overfitting or underfitting. If a shallow tree has a small number of leaves, it cannot learn complex relations. On the other hand, too deep a tree with many leaves split the data into too many small groups and it will make very unreliable predictions for new data [31].

Random Forest Regressor

The second model is the Random Forest Regressor, which creates multiple trees and produces a result by averaging the result from each tree. A big advantage of this model is that it usually works well with the default parameters [31].

Gradient Boosting Regressor

Like the Random Forest Regressor, this model is an ensemble method, which means that it combines the predictions of several models. This makes it handle many problems very well, often surpassing Random Forest Regressor in efficiency. A very important parameter for this model is the number of models that will participate in the prediction. Too few will cause underfitting, while too many models will cause overfitting.

Sequential Model

The last model used is a neural network built on the Sequential Model. This model is the only one that can be called Deep Learning because it uses a deep network with hidden layers. It is more complex than the previous ones and, in theory, it is the most powerful tool for predicting the FTP change of athletes, so it should do it more accurately. However, one needs to parameterize it properly, which is not a trivial task, as one needs to manually set the number of layers, their type and size. In addition, one can add several enhancements that will help train the network properly, such as Dropout, or Batch Normalization. Once all the parameters are correctly selected, the network should work with great efficiency.

4.5 Clustering

To cluster the cyclists as described in the previous chapter, 4.4 were assigned the parameters described there. The first is to completely change the FTP parameter. It has been implemented in such a way that all FTP changes for a given athlete are summed up, and then this value is divided by the number of months, so that athletes training for a long time will not be distinguished. Each of these values is added to the list. When all athletes are included, they are added to the dataset as a new column A.10.

```
sum_change_ftp_athletes = sum_change_ftp(FinalDF)
sum_change_ftp_athletes.head(20)
```

The `balance` was then calculated for each competitor, as described in the previous chapter 4.4. After the `balance` sheet for each athlete has been calculated, this value is added to the list, which is added at the end of the dataset as a new column A.11.

```
balance_change_ftp_athletes = balance_change_ftp(FinalDF)
balance_change_ftp_athletes.head(20)
```

The third parameter is the average FTP value of each rider. The function runs through all the months and sums up the FTP from each of them, and finally divides the sum by the number of months the athlete has trained. Finally, this value is added to the lists. Once all athletes are in the list, they are added to the dataset as a new column A.12.

```
mean_ftp_athletes = mean_ftp(FinalDF)
mean_ftp_athletes.head(20)
```

```
clasterizeDF = pd.concat([mean_ftp_athletes, sum_change_ftp_athletes,
    balance_change_ftp_athletes], axis=1)
```

The first clustering was performed on the basis of all three parameters.

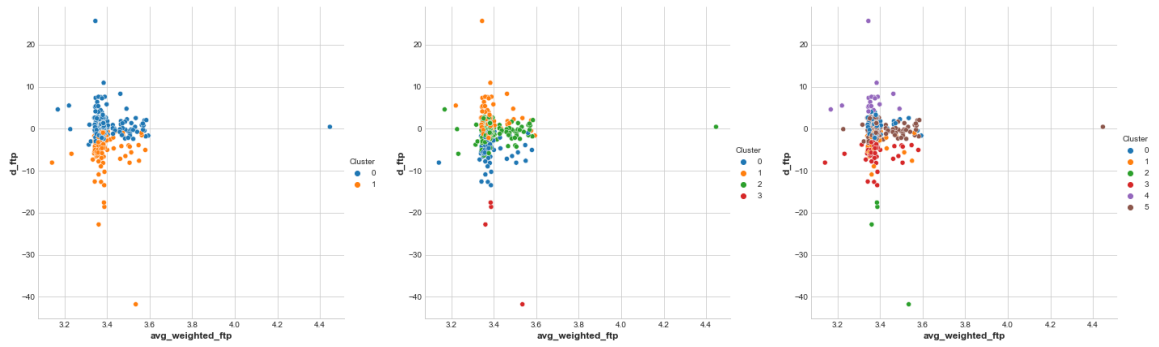
```
X = clasterizeDF.loc[:, ['avg_weighted_ftp', 'balance', 'd_ftp']]
X.head()
```

The division was made into 2, 4, and 6 clusters. The code for all cases is identical and differs in the value of the `n_clusters` parameter in the KMeans function.

```
# Create cluster feature
kmeans = KMeans(n_clusters=2)
X["Cluster"] = kmeans.fit_predict(X)
X["Cluster"] = X["Cluster"].astype("category")
X.head()
```

The clustering results are as follows:

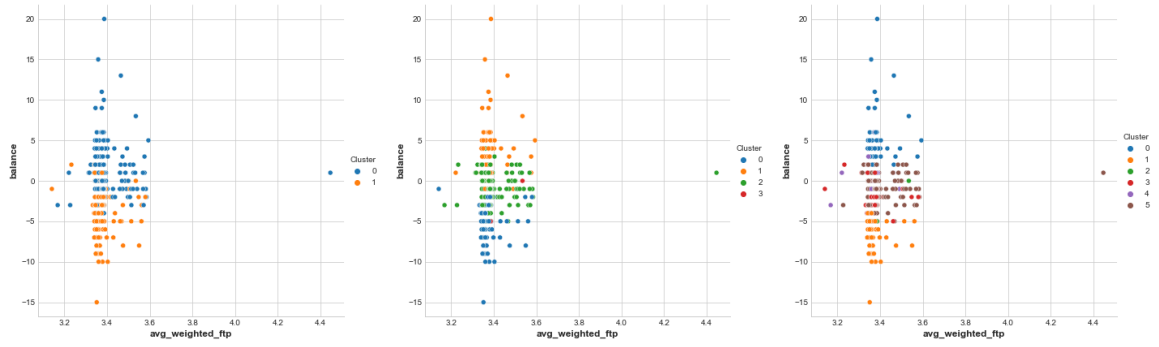
```
sns.relplot(
    x="avg_weighted_ftp", y="d_ftp", hue="Cluster", data=X, height=6,
);
```



2 clusters *4 clusters* *6 clusters*

Figure 4.5: Relation between average weighted FTP and FTP change divided into 2, 4 and 6 clusters.

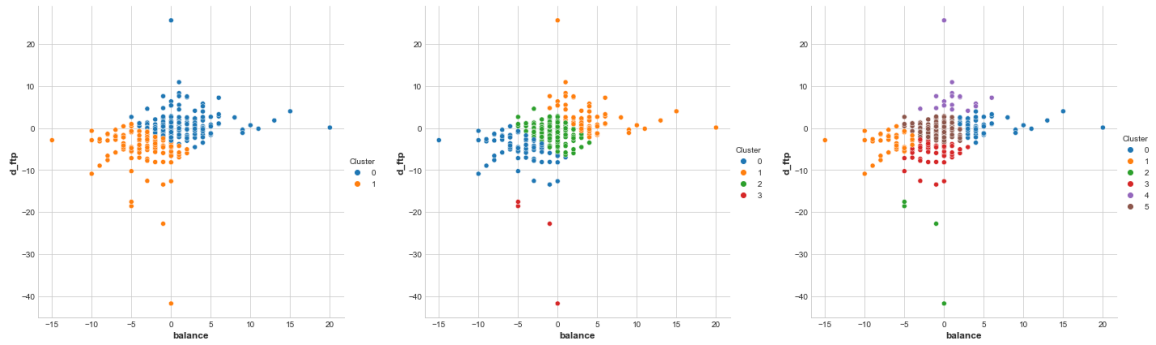
```
sns.relplot(
    x="avg_weighted_ftp", y="balance", hue="Cluster", data=X, height=6,
);
```



2 clusters *4 clusters* *6 clusters*

Figure 4.6: Relationship between average weighted FTP and balance divided into 2, 4 and 6 clusters.

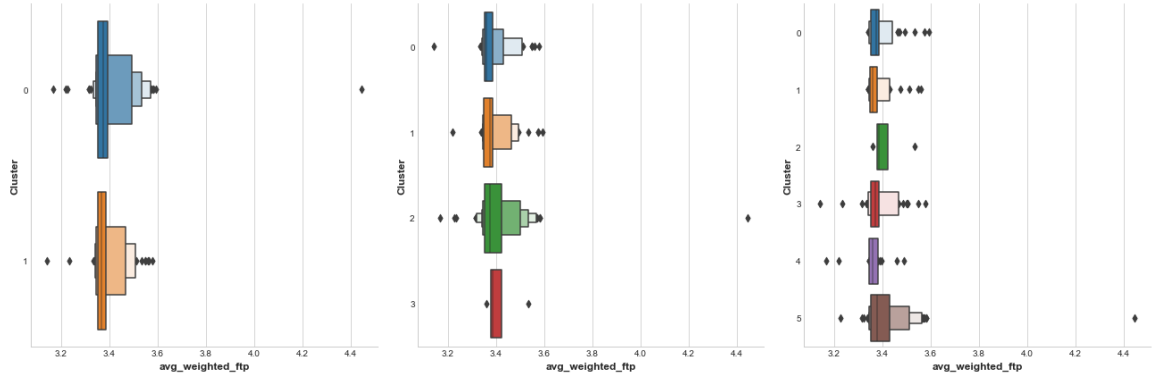
```
sns.relplot(
    x="balance", y="d_ftp", hue="Cluster", data=X, height=6,
);
```



2 clusters *4 clusters* *6 clusters*

Figure 4.7: The relationship between FTP change and balance divided into 2, 4 and 6 clusters.

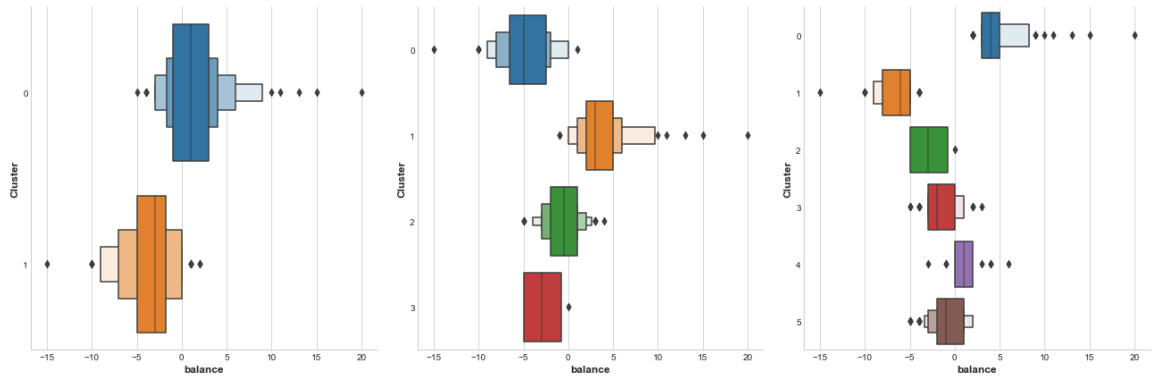
```
X["avg_weighted_ftp"] = clasterizeDF["avg_weighted_ftp"]
sns.catplot(x="avg_weighted_ftp", y="Cluster", data=X, kind="boxen", height=6);
```



2 clusters *4 clusters* *6 clusters*

Figure 4.8: Average weighted FTP distribution in every cluster.

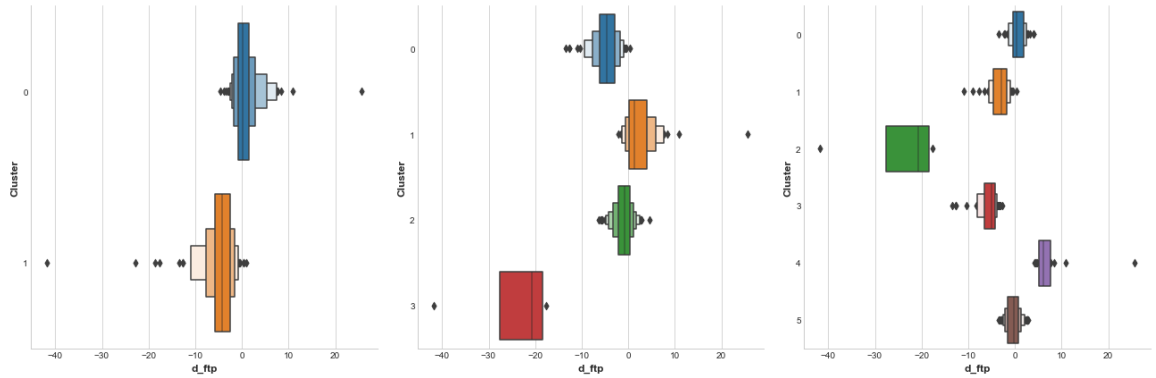
```
X["balance"] = clasterizeDF["balance"]
sns.catplot(x="balance", y="Cluster", data=X, kind="boxen", height=6);
```



2 clusters *4 clusters* *6 clusters*

Figure 4.9: Balance distribution in every cluster.

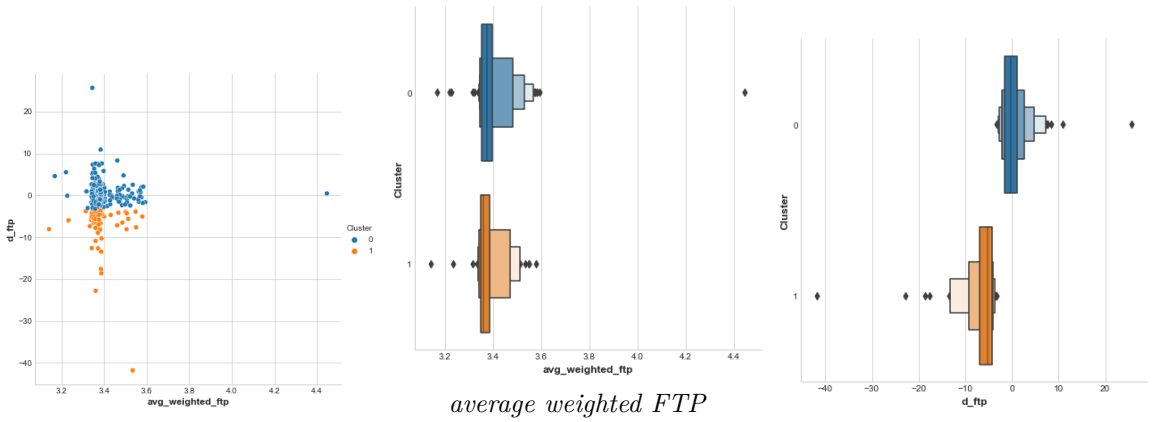
```
X["d_ftp"] = clasterizeDF["d_ftp"]
sns.catplot(x="d_ftp", y="Cluster", data=X, kind="boxen", height=6);
```



2 clusters *4 clusters* *6 clusters*

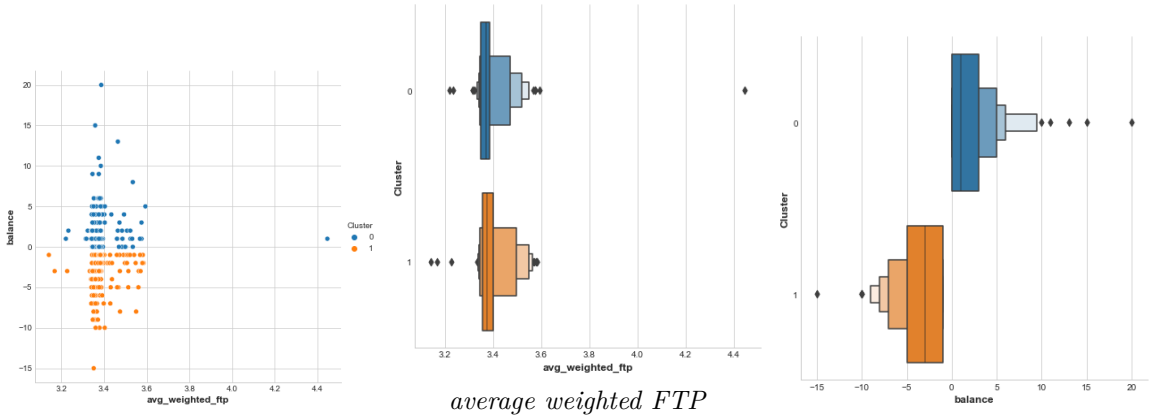
Figure 4.10: FTP change distribution in every cluster.

In the next step, clustering was performed on the basis of two of the three parameters. The implementation was identical to that for the three parameters, so it is unnecessary to copy the same code again. The results of this clustering are as follows:



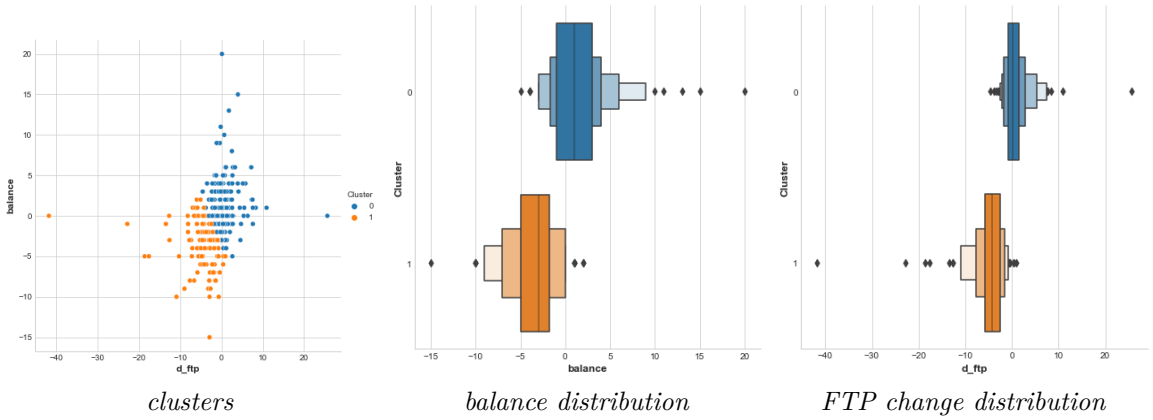
clusters *average weighted FTP distribution* *FTP change distribution*

Figure 4.11: Clustering based on the parameters `avg_weighted_ftp` and `d_ftp`



clusters *average weighted FTP distribution* *balance distribution*

Figure 4.12: Clustering based on `avg_weighted_ftp` and `balance` parameters



clusters *balance distribution* *FTP change distribution*

Figure 4.13: Clustering based on the `balance` parameters and `d_ftp`

4.6 Machine Learning Models

Decision Tree Regressor

The first model that was implemented is the Decision Tree Regressor. The most important parameter that influences the operation of this model is `max_leaf_nodes`. To choose well the value of this parameter, a function has been implemented that calculates the value of the mean absolute error function for the model in question for the given parameter `max_leaf_nodes`. This function is called in a loop that changes `max_leaf_nodes`, and this function returns a mean absolute error. Therefore, it is possible to find the optimal value of `max_leaf_nodes`:

```
def get_mae(max_leaf_nodes, train_X, val_X, train_y, val_y):
    model = DecisionTreeRegressor(max_leaf_nodes=max_leaf_nodes, random_state=0)
    model.fit(train_X, train_y)
    preds_val = model.predict(val_X)
    mae = mean_absolute_error(val_y, preds_val)
    return(mae)
```

```
for max_leaf_nodes in [2,3,4,5,6,7,8,9,10,20,30,40,50,60,70,80,90,100,150,200,500,1000]:
    my_mae = get_mae(max_leaf_nodes, X_train, X_valid, y_train, y_valid)
    print(f'Max leaf nodes: {max_leaf_nodes}  \t\t Mean Absolute Error:  {my_mae}')
```

```
Max leaf nodes: 2               Mean Absolute Error:  14.332704718981114
Max leaf nodes: 3               Mean Absolute Error:  14.177162651429049
Max leaf nodes: 4               Mean Absolute Error:  14.114122538179636
Max leaf nodes: 5               Mean Absolute Error:  14.049905048589977
Max leaf nodes: 6               Mean Absolute Error:  14.024378878793412
Max leaf nodes: 7               Mean Absolute Error:  14.021199728519989
Max leaf nodes: 8               Mean Absolute Error:  14.017666853058824
Max leaf nodes: 9               Mean Absolute Error:  14.07141601366929
Max leaf nodes: 10              Mean Absolute Error:  14.04734604703188
Max leaf nodes: 20              Mean Absolute Error:  14.009778255968127
Max leaf nodes: 30              Mean Absolute Error:  14.064520019440018
Max leaf nodes: 40              Mean Absolute Error:  14.030293724773276
Max leaf nodes: 50              Mean Absolute Error:  14.112943336811322
Max leaf nodes: 60              Mean Absolute Error:  14.23903987072724
Max leaf nodes: 70              Mean Absolute Error:  14.272815503018116
Max leaf nodes: 80              Mean Absolute Error:  14.28934420029861
Max leaf nodes: 90              Mean Absolute Error:  14.503168691337489
Max leaf nodes: 100             Mean Absolute Error:  14.56947573554048
Max leaf nodes: 150             Mean Absolute Error:  14.891086390248203
Max leaf nodes: 200             Mean Absolute Error:  15.11977838151883
Max leaf nodes: 500             Mean Absolute Error:  16.7493371896262
Max leaf nodes: 1000            Mean Absolute Error:  18.093729897974285
```

Figure 4.14: Mean absolute error for different max leaf nodes in Decision Tree Regressor.

The smallest mean absolute error is 14.017666853058824 for a `max_leaf_nodes` equal 8.

Random Forest Regressor

The second model is the random forest regression model. As was said in the previous chapter, this model usually works fine for the default parameters, and it can optimize itself during runtime. The implementation is as follows:

```
forest_model = RandomForestRegressor(random_state=1)
forest_model.fit(X_train, y_train)
rf_preds = forest_model.predict(X_valid)
```

```
    mean_absolute_error(y_valid, rf_preds)
```

The mean absolute error for this model returns 13.649644833197724.

Gradient Boosting Regressor

The most important parameter that influences the operation of this model is `n_estimators`, which determines the number of models that participate in the prediction. To choose it well, a function has been implemented that trains the XGBRegressor model for the given parameter `n_estimators` and returns the value of the mean absolute error function for the test data. This function is called in a loop that passes successive values of the parameter `n_estimators`.

```
    def xgb_mae(n_estimators, train_X, val_X, train_y, val_y):
        xgb_model = XGBRegressor(n_estimators=n_estimators, learning_rate=0.05, n_jobs=4)
        xgb_model.fit(train_X, train_y, early_stopping_rounds=5, eval_set=[(val_X,val_y)],
    varbose=False)
        xgb_pred = xgb_model.predict(val_X)
        xgb_mae = mean_absolute_error(xgb_pred, val_y)
        return xgb_mae
```

```
    for n_estimators in [100,200,300,400,500,600,700,800,900,1000]:
        xgb_mae = get_mae(n_estimators, X_train, X_valid, y_train, y_valid)
        print(f'No estimators: {n_estimators}  \t\t Mean Absolute Error:  {xgb_mae}')
```

The results of this code are as follows:

```
No estimators: 100              Mean Absolute Error:  14.56947573554048
No estimators: 200              Mean Absolute Error:  15.11977838151883
No estimators: 300              Mean Absolute Error:  15.885569092092071
No estimators: 400              Mean Absolute Error:  16.28809131426469
No estimators: 500              Mean Absolute Error:  16.7493371896262
No estimators: 600              Mean Absolute Error:  17.111414883349504
No estimators: 700              Mean Absolute Error:  17.373130283774138
No estimators: 800              Mean Absolute Error:  17.507581023192156
No estimators: 900              Mean Absolute Error:  17.851925174750523
No estimators: 1000             Mean Absolute Error:  18.093729897974285
```

Figure 4.15: Mean absolute error for different `n_estimators` in Gradient Boosting Regressor.

The function `mean_absolute_error` returns the lowest value of 14.56947573554048 for `n_estimators` equal 100.

Sequential Model

There is a Deep Learning model, and in the process of creating it, many more parameters had to be set than in the cases described above. At first, the `early_stopping` function was implemented. It is used to stop training when the loss function for the test set stops decreasing to prevent overfitting:

```
early_stopping = callbacks.EarlyStopping(
    min_delta=0.001, # minimium amount of change to count as an improvement
    patience=30, # how many epochs to wait before stopping
    restore_best_weights=True,
)

```

Then the model itself was implemented. Six dense layers of 4096 neurons have been created in each of these. The activation function in each of them is the ReLU function. Between the dense layers are dropout layers and batch normalization layers. The dropout layers exclude 30% of the workout data to avoid overfitting. Batch normalization layers perform normalization of data that could make training unstable. Also, since 13 training features have been defined, the parameter `input_shape` is set to [13].

```
model = keras.Sequential([
    layers.Dense(4096, activation='relu', input_shape=[13]),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(4096, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(4096, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(4096, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(4096, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(4096, activation='relu'),
    layers.Dropout(0.3),
    layers.BatchNormalization(),
    layers.Dense(1),
])

```

In the model discussed, the algorithm 'adam' was set as optimizer and the loss function, the mean absolute error, was set to be able to compare the results obtained with the results of the models discussed above.

```
model.compile(
    optimizer='adam',
    loss='mae',
)

```

Then the parameter `batch_size` = 256 was set as the one-time number of rows that are supplied to the optimizer function and the parameter epochs = 500, which tells how many times the model will pass through the dataset.

```
history = model.fit(
    X_train, y_train,
    validation_data=(X_valid, y_valid),
    batch_size=256,
    epochs=500,
    callbacks=[early_stopping], # put your callbacks in a list
    verbose=0,  # turn off training log
)

```

The error for the described model is as follows:

```
history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot();
```

```
3 print("Minimum validation loss: {}".format(history_df['val_loss'].min()))
4
```

```
Minimum validation loss: 13.724555015563965
```

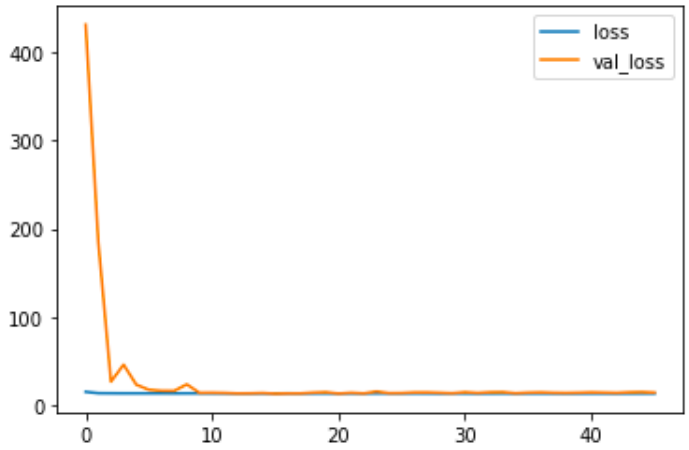


Figure 4.16: Loss functions for Sequential Model.

Summary of the results obtained

5.1 Clustering

Let's take a closer look at the clustering performed by the K-Means algorithm. First, let us analyze the clustering split, using all three parameters: `avg_weighted_ftp`, `balance` and `d_ftp`. If we look at the figures 4.5 and 4.6, that is, the relationship of `avg_weighted_ftp` and `d_ftp`, and of `avg_weighted_ftp` and `balance`, we observe that in the case of splitting into clusters 2 and 4, we can actually separate them. In the case of splitting into six clusters, the division between them blurs. Where this division is visible, we see that it depends solely on the `d_ftp` parameter and we can draw horizontal lines separating the different clusters.

The case is similar when we look at `avg_weighted_ftp` and `balance` in this split. Again, the differences between the clusters can only be seen for the split of the clusters 2 and 4, and it also seems to be independent of `avg_weighted_ftp`.

The relationship between `d_ftp` and `balance` looks slightly different. After eliminating the parameter `avg_weighted_ftp`, one can clearly see the breakdown of not only 2 and 4, but also 6 clusters. Moreover, it can be seen that if we draw lines separating individual clusters, they would be oblique, which indicates a dependence on both parameters. This is complemented by the conclusions from the relationships discussed above, where both parameters were juxtaposed with `avg_weighted_ftp` and both were decisive for the split.

When we look at the distributions `avg_weighted_ftp` 4.8, `balance` 4.9 and `d_ftp` 4.10, we can confirm that the division into clusters does not depend on the parameter `avg_weighted_ftp`, and only between `d_ftp` and `balance`.

In the case of clustering based on two parameters, we can observe analogous, almost identical results as in the case of division into two clusters using three parameters. For clustering based on `avg_weighted_ftp` and `d_ftp` 4.11, we get almost identical dependencies as for the dependencies of these parameters 4.5 when clustering with the three sizes. The same is true for the other

combinations of parameters.

Therefore, it should be repeated that the division into clusters depends only on `d_ftp` and `balance`, and is not related to `avg_weighted_ftp`. However, at this point, some doubts arise because players with a higher average FTP should focus more on the effectiveness of this training in their training and should achieve better results in this matter than players with a lower level of training.

Although it was actually possible to divide the players into two groups due to the effects of their training, this division leaves some doubts and does not seem to be entirely clear.

5.2 Other Machine Learning Methods

Now let us take a closer look at the results obtained from trying to predict player FTP changes with the implemented models. The best The mean absolute error values measured on the test data for each model are as follows:

- Decision Tree Regressor: 14.017666853058824
- Random Forest Regressor: 13.649644833197724
- Gradient Boosting Regressor: 14.56947573554048
- Sequential Model: 13.724555015563965

The mean absolute error alone does not say much. It must be juxtaposed with the spread of the test data. The minimum value in the tests is -73.15 and the maximum is 49.40. Compared to these error magnitude values, which are similar in each of the models, we can see that the error is approximately 9% of the amplitude of the test data. This is a very high value and shows that none of the models is working properly. In addition, some parameters in the Decision Tree Regressor and Gradient Boosting Regressor models were tested for the smallest error, and for the remaining values of the optimized parameters, the error was greater but not much greater (on the same order of magnitude). In addition, the Sequential Model provides the ability to study the error function course for training data. It turns out that this value does not go below 14 4.16. On this basis, it can be concluded that none of the podels have learned the data provided to them, and we are dealing with underfitting. This result is surprising. The first reason that comes to mind is

that the data set is too small. However, it can be quickly thrown out, since the data set contains more than 8000 rows, which seems to be more than enough for either model to learn the data dependency. Besides, the Sequential Model runs on multiple epochs. Therefore, the problem is not too little data. So, it seems that the models behave as if there is no dependency in the data. Let us look at a list of some sample data from the test set and the Sequential Model predictions:

	d_ftp	seq_pred
0	0.00	2.109999
7	22.80	-0.868964
19	14.25	9.323878
21	5.70	7.362654
24	0.95	4.980586
25	-12.35	2.155059
28	-8.55	9.949760
35	-3.80	-2.522422
43	19.95	-2.310605

Figure 5.1: Comparison of some sample data from the test set, and the Sequential model prediction.

The above data show that the model prediction is very different from the actual data. Therefore, we can finally conclude that there is no relationship between the FTP change and the input parameters of the model in the data. This result raises considerable doubts because for several years, tens of thousands of cyclists around the world have empirically confirmed that these parameters affect the change in FTP. Therefore, further questions arise, first of all, as to why the data analyzed in this work do not reflect the reality they were supposed to model.

The data have gone through many transformational processes. Unfavorable performance of any of these processes could cause an error that breaks the dependencies inherent in the data set in question. However, perhaps the initial assumption that a player's FTP can be determined based on the maximum average power of 20 minutes of effort each month is naive. Perhaps many players do not perform their activities with such high intensity. If these assumptions are wrong, the FTP values and the parameters based on them have little to do with reality. Here, it is worth returning for a moment to the

non-obvious clustering results, which indicated some problems with the FTP parameter. Thus, in this case, the problem lies in the interface between the data and its interpretation and use.

Conclusion

The results obtained are not exactly as expected and raise some questions and doubts. They shed new light on the data analyzed and on the data generated during cycling training in general. There is no doubt that parameters such as exercise intensity or time spent in individual power zones have a key impact on how an athlete's sports level will change. It is worth returning to the stage of data aggregation and analyzing subsequent processes to which they are subjected. Perhaps the problem is not that complicated, and it is enough to apply appropriate filters to the data, which will allow to classify to the data set only those players who regularly implement training plans and consciously raise their sports level. Here, further questions arise. Is it then possible to make such a classification without calculating these effects in a rather complicated and lengthy process? Or maybe the problem is not here and the assumptions are correct, but the problem lies somewhere in the implementation of the entire system. It certainly takes a long time to rethink the whole matter in order to find the answers to the above questions. Undoubtedly, if it is possible to solve this issue and actually create a model that can predict how the athlete's form will change as a result of the training stimuli he is subjected to, it will be a breakthrough in cycling training and the next natural step will be to optimize this process so that the growth of the rider's sports level was as high as possible and at the same time did not cause overtraining. The case is undoubtedly open. Probably a relatively small improvement may make the system presented here fully functional, and on its basis it will be possible to build new functionalities improving the quality of cycling training. It should be emphasized that this system is innovative in its application, because the use of data analysis in cycling has not yet reached the limits of such a specific optimization of training effects. It will develop without a doubt in the near future. Of course, the problems discussed will be addressed first so that it can finally become fully functional for any training cyclist and be able to act as an assistant cycling coach. Therefore, the above work should be treated as preliminary results of a larger project that will be developed in the future.

Bibliography

[1] datetime x2014; Basic date and time types x2014; Python 3.10.6 documentation — docs.python.org. https://docs.python.org/3/library/datetime.html. [Accessed 29-Aug-2022].

[2] dateutil - powerful extensions to datetime x2014; dateutil 2.8.2 documentation — dateutil.readthedocs.io. https://dateutil.readthedocs.io/en/stable/. [Accessed 29-Aug-2022].

[3] os x2014; Miscellaneous operating system interfaces x2014; Python 3.10.6 documentation — docs.python.org. https://docs.python.org/3/library/os.html. [Accessed 29-Aug-2022].

[4] shutil x2014; High-level file operations x2014; Python 3.10.6 documentation — docs.python.org. https://docs.python.org/3/library/shutil.html. [Accessed 29-Aug-2022].

[5] Stages Cycling Store — Home page — Stages Cycling Store — stagescycling.eu. https://www.stagescycling.eu/index.php?lang=1&. [Accessed 31-Aug-2022].

[6] statistics x2014; Mathematical statistics functions x2014; Python 3.10.6 documentation — docs.python.org. https://docs.python.org/3/library/statistics.html. [Accessed 29-Aug-2022].

[7] Strava — Run and Cycling Tracking on the Social Network for Athletes — strava.com. https://www.strava.com. [Accessed 27-Aug-2022].

[8] TrainingPeaks — Train With Confidence — trainingpeaks.com. https://www.trainingpeaks.com. [Accessed 27-Aug-2022].

[9] What is critical power?, Jun 2022.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah,

M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[11] H. Allen, A. Coggan. *Training and Racing with a Power Meter*. VeloPress, 2012.

[12] G. Cheetah. Goldencheeta open data python library. https://github.com/GoldenCheetah/OpenData/tree/master/opendata-python, 2019. [Accessed 15-Aug-2022].

[13] T. Chen, C. Guestrin. XGBoost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, strony 785–794, New York, NY, USA, 2016. ACM.

[14] S. Coakley, L. Passfield. Cycling performance is superior for time-to-exhaustion versus time-trial in endurance laboratory tests. *Journal of Sports Sciences*, 36:1–7, 09 2017.

[15] C. da Costa-Luis. tqdm documentation — tqdm.github.io. https://tqdm.github.io. [Accessed 29-Aug-2022].

[16] C. Damian. What is a training stress score [tss] cycling: Trainingpeaks: 2020, Jun 2021.

[17] C. Damian. What is normalized power (np), variability index, Oct 2021.

[18] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Wrze. 2020.

[19] D. W. Hill. The critical power concept. *Sports medicine*, 16(4):237–254, 1993.

[20] C. Howe. The road cyclist's guide to training by.

[21] http://www.facebook.com/tylerbenedict. Power Meters Explained - Every feature, technology function you need to know — bikerumor.com. https://bikerumor.com/power-meters-explained-every-feature-technology-function-you-need-to-know/. [Accessed 31-Aug-2022].

[22] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[23] S. Hurley. Normalized power®: What it is and how to use it - trainerroad blog, Jun 2021.

[24] P. Jaromin. Bikeworld.pl co to jest vo2max i jak go zwiekszyć? / porady, Aug 2022.

[25] A. Kogut. Strefy treningowe - tego jeszcze nie wiesz - way2champ - cycling training plans, Mar 2020.

[26] Y. Li, H. Wu. A clustering method based on k-means algorithm. *Physics Procedia*, 25:1104–1109, 2012. International Conference on Solid State Devices and Materials Science, April 1-2, 2012, Macao.

[27] M. Liversedge. Basic open data notebook, Nov 2018.

[28] M. Liversedge. Getting started with goldencheetah opendata, Nov 2018.

[29] M. Liversedge. Goldencheeta open data python library. https://github.com/GoldenCheetah/OpenData/tree/master/opendata-python, 2018.

[30] T. pandas development team. pandas-dev/pandas: Pandas, Luty 2020.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[32] F. Pérez, B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, Maj 2007.

[33] T. T. Training. Lactate thresholds made simple - by rob bridges, Dec 2019.

[34] G. Van Rossum. *The Python Library Reference, release 3.8.2.* Python Software Foundation, 2020.

[35] M. L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021.

Appendix

A.1 Adding month column

```
def month(df):
return parse(df['date']).month
```

A.2 Crating DataFrame base on activities metadata for one athlete

```
def create_dataset(data):
    criticalPower = pd.DataFrame()
    titles = ['activity','athlete','date','weight','duration','coggan_tss','np']
    for activity in tqdm(data):
        rows = criticalPower
        s2 = pd.Series(activity.metadata['METRICS']['power_mmp'],
          index=activity.metadata['METRICS']['power_mmp_secs'])
        s1 = pd.Series([activity.id,athlete.id,activity.metadata['date'],activity.
metadata['METRICS']['athlete_weight'],activity.metadata['METRICS']['time_riding'],
activity.metadata['METRICS']['coggan_tss'],activity.metadata['METRICS']['coggan_np'
][0]],index=titles)
        row = pd.concat([s1,s2])
        criticalPower = pd.concat([rows, row.to_frame().T], ignore_index=True)
    criticalPower['month'] = criticalPower.apply(lambda row: month(row), axis=1)
    return criticalPower
```

A.3 Calculating FTP per month

```
def calc_ftp(dataFrame):
    x = 0
    mon = dataFrame['month'][0]
    athlete = dataFrame['athlete'][0]
    monftp = []
    ftp = []
    acc = []
    for i, row in tqdm(dataFrame.iterrows()):
        if(mon == row['month'] and athlete == row['athlete']):
            if(np.isnan(row['power_1200s'])):
                acc = acc + [0]
            else:
                acc = acc + [row['power_1200s']]
        else:
            monftp = [max(acc) * 0.95] * x
            ftp = ftp + monftp
            x = 0
            monftp = []
            acc = []
            mon = row['month']
            athlete = row['athlete']
```

```
                if(np.isnan(row['power_1200s'])):
                    acc = acc + [0]
                else:
                    acc = acc + [row['power_1200s']]
            x = x + 1
        monftp = [max(acc) * 0.95] * x
        ftp = ftp + monftp
        return ftp
```

A.4 Calculating Power Zones

```
    def calc_power_zones(dataFrame):
        athlete = ''
        mon = 0
        onePZ = []
        onePZ_acc = 0
        twoPZ = []
        twoPZ_acc = 0
        threePZ = []
        threePZ_acc = 0
        fourPZ = []
        fourPZ_acc = 0
        fivePZ = []
        fivePZ_acc = 0
        sixPZ = []
        sixPZ_acc = 0
        for i, row in tqdm(dataFrame.iterrows()):
            if (i == 0):
                athlete = row['athlete']
                mon = row['month']
                onePZ_acc = 0.55 * row['ftp']
                twoPZ_acc = 0.75 * row['ftp']
                threePZ_acc = 0.9 * row['ftp']
                fourPZ_acc = 1.05 * row['ftp']
                fivePZ_acc = 1.2 * row['ftp']
                sixPZ_acc = 1.5 * row['ftp']
                onePZ = onePZ + [onePZ_acc]
                twoPZ = twoPZ + [twoPZ_acc]
                threePZ = threePZ + [threePZ_acc]
                fourPZ = fourPZ + [fourPZ_acc]
                fivePZ = fivePZ + [fivePZ_acc]
                sixPZ = sixPZ + [sixPZ_acc]
            elif (mon == row['month'] and athlete == row['athlete']):
                onePZ = onePZ + [onePZ_acc]
                twoPZ = twoPZ + [twoPZ_acc]
                threePZ = threePZ + [threePZ_acc]
                fourPZ = fourPZ + [fourPZ_acc]
                fivePZ = fivePZ + [fivePZ_acc]
                sixPZ = sixPZ + [sixPZ_acc]
            else:
                onePZ_acc = 0.55 * row['ftp']
                twoPZ_acc = 0.75 * row['ftp']
                threePZ_acc = 0.9 * row['ftp']
                fourPZ_acc = 1.05 * row['ftp']
                fivePZ_acc = 1.2 * row['ftp']
                sixPZ_acc = 1.5 * row['ftp']
                onePZ = onePZ + [onePZ_acc]
                twoPZ = twoPZ + [twoPZ_acc]
                threePZ = threePZ + [threePZ_acc]
                fourPZ = fourPZ + [fourPZ_acc]
                fivePZ = fivePZ + [fivePZ_acc]
                sixPZ = sixPZ + [sixPZ_acc]
            mon = row['month']
            athlete = row['athlete']
```

```
54          return onePZ, twoPZ, threePZ, fourPZ, fivePZ, sixPZ
```

A.5 Time spent in every power zones for every activity

```
1     def time_in_zones(dataFrame):
2         time_in_zone1 = []
3         time_in_zone2 = []
4         time_in_zone3 = []
5         time_in_zone4 = []
6         time_in_zone5 = []
7         time_in_zone6 = []
8         time_in_zone7 = []
9         z1 = 0
10        z2 = 0
11        z3 = 0
12        z4 = 0
13        z5 = 0
14        z6 = 0
15        time_z1 = 0
16        time_z2 = 0
17        time_z3 = 0
18        time_z4 = 0
19        time_z5 = 0
20        time_z6 = 0
21        time_z7 = 0
22        month = 0
23        athlete_id = dataFrame['athlete'][0]
24        athlete = od.get_local_athlete(athlete_id)
25        #athlete_activities = list(athlete.activities())
26        to_delete = {}
27        for i, row in tqdm(dataFrame.iterrows()):
28            if (i == 0):
29                z1 = row['1_power_zone']
30                z2 = row['2_power_zone']
31                z3 = row['3_power_zone']
32                z4 = row['4_power_zone']
33                z5 = row['5_power_zone']
34                z6 = row['6_power_zone']
35                month = row['month']
36            elif (z1 != row['1_power_zone'] or athlete_id != row['athlete'] or month != row[
    'month']):
37                z1 = row['1_power_zone']
38                z2 = row['2_power_zone']
39                z3 = row['3_power_zone']
40                z4 = row['4_power_zone']
41                z5 = row['5_power_zone']
42                z6 = row['6_power_zone']
43                month = row['month']
44                if (athlete_id != row['athlete']):
45                    athlete_id = row['athlete']
46                    athlete = od.get_local_athlete(athlete_id)
47                    #athlete_activities = list(athlete.activities())
48            activity = athlete.get_activity(row['activity'])
49            #if activity in athlete_activities:
50            try:
51                for i, row in activity.data.iterrows():
52                    if (row['power'] <= z1 and row['power'] > 0):
53                        time_z1 = time_z1 + 1
54                    elif (row['power'] <= z2 and row['power'] > z1):
55                        time_z2 = time_z2 + 1
56                    elif (row['power'] <= z3 and row['power'] > z2):
57                        time_z3 = time_z3 + 1
58                    elif (row['power'] <= z4 and row['power'] > z3):
59                        time_z4 = time_z4 + 1
```

```
                    elif (row['power'] <= z5 and row['power'] > z4):
                        time_z5 = time_z5 + 1
                    elif (row['power'] <= z6 and row['power'] > z5):
                        time_z6 = time_z6 + 1
                    elif (row['power'] > z6):
                        time_z7 = time_z7 + 1
                time_in_zone1 = time_in_zone1 + [time_z1]
                time_in_zone2 = time_in_zone2 + [time_z2]
                time_in_zone3 = time_in_zone3 + [time_z3]
                time_in_zone4 = time_in_zone4 + [time_z4]
                time_in_zone5 = time_in_zone5 + [time_z5]
                time_in_zone6 = time_in_zone6 + [time_z6]
                time_in_zone7 = time_in_zone7 + [time_z7]
                time_z1 = 0
                time_z2 = 0
                time_z3 = 0
                time_z4 = 0
                time_z5 = 0
                time_z6 = 0
                time_z7 = 0
            except:
                to_delete[i] = [row['athlete'], row['activity']]
                time_in_zone1 = time_in_zone1 + [0]
                time_in_zone2 = time_in_zone2 + [0]
                time_in_zone3 = time_in_zone3 + [0]
                time_in_zone4 = time_in_zone4 + [0]
                time_in_zone5 = time_in_zone5 + [0]
                time_in_zone6 = time_in_zone6 + [0]
                time_in_zone7 = time_in_zone7 + [0]
                time_z1 = 0
                time_z2 = 0
                time_z3 = 0
                time_z4 = 0
                time_z5 = 0
                time_z6 = 0
                time_z7 = 0
        print(len(to_delete))
        print(to_delete)
        return time_in_zone1, time_in_zone2, time_in_zone3, time_in_zone4, time_in_zone5,
    time_in_zone6, time_in_zone7
```

A.6 Calculating TSS

```
    def calc_tss(dataFrame):
        tss = []
        for i, row in tqdm(dataFrame.iterrows()):
            tss = tss + [(float(row['duration'])*float(row['np'])*float(row['np'])/float(row
    ['ftp']))/(float(row['ftp'])*3600)*100]
        return tss
```

A.7 Crating DataFrame base on activities metadata for all athletes

```
    def create_dataframe():
        criticalPower = pd.DataFrame()
        activity_id = []
        athlete_id = []
        date = []
        weight = []
        duration = []
        #coggan_tss = []
```

```
        np = []
        power_1200s = []
        for athlete in tqdm(local_athletes):
            activities = list(athlete.activities())
            for activity in activities:
                try:
                    if ('power_mmp_secs' in activity.metadata['METRICS'] and 'athlete_weight
    ' in                 activity.metadata['METRICS'] and 'time_riding' in activity.metadata['
    METRICS']        and 'coggan_np' in activity.metadata['METRICS']):
                        if(len(activity.metadata['METRICS']['power_mmp']) > 284):
                            power_1200s = power_1200s + [activity.metadata['METRICS']['
    power_mmp'][284]]
                        else:
                            power_1200s = power_1200s + [math.nan]
                        activity_id = activity_id + [activity.id]
                        athlete_id = athlete_id + [athlete.id]
                        date = date + [activity.metadata['date']]
                        weight = weight + [activity.metadata['METRICS']['athlete_weight']]
                        duration = duration + [activity.metadata['METRICS']['time_riding']]
                        #coggan_tss = [activity.metadata['METRICS']['coggan_tss']]
                        np = np + [activity.metadata['METRICS']['coggan_np'][0]]
                except:
                    continue
        print(f'len activity: {len(activity_id)}')
        print(f'len athlete: {len(athlete_id)}')
        print(f'len date: {len(date)}')
        print(f'len weight: {len(weight)}')
        print(f'len duration: {len(duration)}')
        #print(f'len coggan_tss: {len(coggan_tss)}')
        print(f'len np: {len(np)}')
        print(f'len power_1200s: {len(power_1200s)}')

        criticalPower['activity'] = activity_id
        criticalPower['athlete'] = athlete_id
        criticalPower['date'] = date
        criticalPower['weight'] = weight
        criticalPower['duration'] = duration
        #criticalPower['coggan_tss'] = coggan_tss
        criticalPower['np'] = np
        criticalPower['power_1200s'] = power_1200s
        try:
            criticalPower['month'] = criticalPower.apply(lambda row: month(row), axis=1)
            return criticalPower
        except:
            return criticalPower
```

A.8 Dreating DataFrame contains monthly data for all of cyclists

```
    def all_month_data(dataFrame):
        monthlyDF = pd.DataFrame()
        athlete_acc = dataFrame['athlete'][0]
        weight_acc = float(dataFrame['weight'][0])
        month_acc = dataFrame['month'][0]
        ftp_acc = dataFrame['ftp'][0]
        no_activities_acc = 0
        duration_acc = 0
        np_acc = 0
        time_in_zone1_acc = 0
        time_in_zone2_acc = 0
        time_in_zone3_acc = 0
        time_in_zone4_acc = 0
        time_in_zone5_acc = 0
        time_in_zone6_acc = 0
        time_in_zone7_acc = 0
```

```
        tss_acc = 0
        athlete = []
        weight = []
        month = []
        ftp = []
        weighted_ftp = []
        no_activities = []
        duration = []
        avg_np = []
        time_in_zone1 = []
        time_in_zone2 = []
        time_in_zone3 = []
        time_in_zone4 = []
        time_in_zone5 = []
        time_in_zone6 = []
        time_in_zone7 = []
        tss = []
        for i, row in tqdm(dataFrame.iterrows()):
            if(month_acc != row['month'] or athlete_acc != row['athlete']):
                athlete = athlete + [athlete_acc]
                weight = weight + [weight_acc]
                month = month + [month_acc]
                ftp = ftp + [ftp_acc]
                weighted_ftp = weighted_ftp + [ftp_acc/weight_acc]
                no_activities = no_activities + [no_activities_acc]
                duration = duration + [duration_acc]
                avg_np = avg_np + [np_acc/no_activities_acc]
                time_in_zone1 = time_in_zone1 + [time_in_zone1_acc]
                time_in_zone2 = time_in_zone2 + [time_in_zone2_acc]
                time_in_zone3 = time_in_zone3 + [time_in_zone3_acc]
                time_in_zone4 = time_in_zone4 + [time_in_zone4_acc]
                time_in_zone5 = time_in_zone5 + [time_in_zone5_acc]
                time_in_zone6 = time_in_zone6 + [time_in_zone6_acc]
                time_in_zone7 = time_in_zone7 + [time_in_zone7_acc]
                tss = tss + [tss_acc]
                no_activities_acc = 0
                duration_acc = 0
                np_acc = 0
                time_in_zone1_acc = 0
                time_in_zone2_acc = 0
                time_in_zone3_acc = 0
                time_in_zone4_acc = 0
                time_in_zone5_acc = 0
                time_in_zone6_acc = 0
                time_in_zone7_acc = 0
                tss_acc = 0
                month_acc = row['month']
                ftp_acc = row['ftp']
                if(athlete_acc != row['athlete']):
                    athlete_acc = row['athlete']
                    weight_acc = float(row['weight'])
            no_activities_acc = no_activities_acc + 1
            duration_acc = duration_acc + int(float(row['duration']))
            np_acc = np_acc + int(float(row['np']))
            time_in_zone1_acc = time_in_zone1_acc + row['time_in_zone1']
            time_in_zone2_acc = time_in_zone2_acc + row['time_in_zone2']
            time_in_zone3_acc = time_in_zone3_acc + row['time_in_zone3']
            time_in_zone4_acc = time_in_zone4_acc + row['time_in_zone4']
            time_in_zone5_acc = time_in_zone5_acc + row['time_in_zone5']
            time_in_zone6_acc = time_in_zone6_acc + row['time_in_zone6']
            time_in_zone7_acc = time_in_zone7_acc + row['time_in_zone7']
            tss_acc = tss_acc + row['tss']
        monthlyDF['athlete'] = athlete
        monthlyDF['weight'] = weight
        monthlyDF['month'] = month
        monthlyDF['ftp'] = ftp
        monthlyDF['weighted_ftp'] = weighted_ftp
        monthlyDF['no_activities'] = no_activities
```

```
        monthlyDF['duration'] = duration
        monthlyDF['avg_np'] = avg_np
        monthlyDF['time_in_zone1'] = time_in_zone1
        monthlyDF['time_in_zone2'] = time_in_zone2
        monthlyDF['time_in_zone3'] = time_in_zone3
        monthlyDF['time_in_zone4'] = time_in_zone4
        monthlyDF['time_in_zone5'] = time_in_zone5
        monthlyDF['time_in_zone6'] = time_in_zone6
        monthlyDF['time_in_zone7'] = time_in_zone7
        monthlyDF['tss'] = tss
        return monthlyDF
```

A.9 Calculating change of FTP

```
    def ftp_change(dataFrame):
        athlete = dataFrame['athlete'][0]
        ftp_acc = dataFrame['ftp'][0]
        ftp_percent_acc = 0
        weighted_ftp_acc = dataFrame['weighted_ftp'][0]
        weighted_ftp_percent_acc = 0
        d_ftp = []
        d_ftp_percent = []
        d_weighted_ftp = []
        d_weighted_ftp_percent = []
        for i, row in tqdm(dataFrame.iterrows()):
            if(i == 0):
                d_ftp = d_ftp + [math.nan]
                d_ftp_percent = d_ftp_percent + [math.nan]
                d_weighted_ftp = d_weighted_ftp + [math.nan]
                d_weighted_ftp_percent = d_weighted_ftp_percent + [math.nan]
                continue
            elif((i+1) == len(dataFrame)):
                d_ftp = d_ftp + [math.nan]
                d_ftp_percent = d_ftp_percent + [math.nan]
                d_weighted_ftp = d_weighted_ftp + [math.nan]
                d_weighted_ftp_percent = d_weighted_ftp_percent + [math.nan]
                break
            elif(athlete == row['athlete']):
                d_ftp_acc = row['ftp'] - ftp_acc
                d_ftp = d_ftp + [d_ftp_acc]
                d_ftp_percent = d_ftp_percent + [d_ftp_acc/row['ftp']]
                d_weighted_ftp_acc = row['weighted_ftp'] - weighted_ftp_acc
                d_weighted_ftp = d_weighted_ftp + [d_weighted_ftp_acc]
                d_weighted_ftp_percent = d_weighted_ftp_percent + [d_weighted_ftp_acc/row['
    weighted_ftp']]
            else:
                d_ftp = d_ftp + [math.nan]
                d_ftp_percent = d_ftp_percent + [math.nan]
                d_weighted_ftp = d_weighted_ftp + [math.nan]
                d_weighted_ftp_percent = d_weighted_ftp_percent + [math.nan]
            athlete = row['athlete']
            ftp_acc = row['ftp']
            weighted_ftp_acc = row['weighted_ftp']
        return d_ftp, d_ftp_percent, d_weighted_ftp, d_weighted_ftp_percent
```

A.10 Calculating sumarized change of FTP

```
def sum_change_ftp(dataFrame):
    athlete = dataFrame['athlete'][0]
    sum_d_ftp = 0
```

```
    no_months = 0
    sum_d_ftp_list = []
    athlts = []
    for i, row in tqdm(dataFrame.iterrows()):
        if(np.isnan(row['d_ftp'])):
            continue
        else:
            if (athlete == row['athlete']):
                sum_d_ftp = sum_d_ftp + row['d_ftp']
                no_months = no_months + 1
            else:
                sum_d_ftp_list = sum_d_ftp_list + [sum_d_ftp/no_months]
                athlts = athlts + [athlete]
                sum_d_ftp = row['d_ftp']
                no_months = 1
        athlete = row['athlete']
    sum_d_ftp_list = sum_d_ftp_list + [sum_d_ftp/no_months]
    athlts = athlts + [athlete]
    sum_d_ftp = row['d_ftp']
    no_months = 1
    claster_athletes = pd.DataFrame(sum_d_ftp_list, index=athlts, columns = ['d_ftp'])
    return claster_athletes
```

A.11 Calculating Balance

```
def balance_change_ftp(dataFrame):
    athlete = dataFrame['athlete'][0]
    balance = 0
    balances = []
    athlts = []
    for i, row in tqdm(dataFrame.iterrows()):
        if(np.isnan(row['d_ftp'])):
            continue
        else:
            if (athlete == row['athlete']):
                if(row['d_ftp'] > 0):
                    balance = balance + 1
                else:
                    balance = balance - 1
            else:
                balances = balances + [balance]
                athlts = athlts + [athlete]
                balance = 0
        athlete = row['athlete']
    balances = balances + [balance]
    athlts = athlts + [athlete]
    balance = 0
    claster_athletes = pd.DataFrame(balances, index=athlts, columns = ['balance'])
    return claster_athletes
```

A.12 Calculating the average FTP for every athlete

```
def mean_ftp(dataFrame):
    athlete = dataFrame['athlete'][0]
    ftp = []
    ftp_acc = []
    avg_ftp = 0
    athlts = []
    for i, row in tqdm(dataFrame.iterrows()):
        if(athlete == row['athlete']):
```

```
            ftp_acc = ftp_acc + [row['weighted_ftp']]
        else:
            ftp = ftp + [mean(ftp_acc)]
            athlts = athlts + [athlete]
        athlete = row['athlete']
    ftp = ftp + [mean(ftp_acc)]
    athlts = athlts + [athlete]
    claster_athletes = pd.DataFrame(ftp, index=athlts, columns = ['avg_weighted_ftp'])
    return claster_athletes
```