ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kyral** Jméno: **Tadeáš** Osobní číslo: **466164**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Framework pro rozsáhlé studie strojového učení ve sportech

Název diplomové práce anglicky:

A framework for large scale assessment of machine learning in sports

Pokyny pro vypracování:

Seznam doporučené literatury:

[1] Richter, Chris, Martin O'Reilly, and Eamonn Delahunt. "Machine learning in sports science: challenges and opportunities." (2021): 1-7.
[2] Dubitzky, Werner, et al. "The open international soccer database for machine learning." Machine Learning 108.1 (2019): 9-28.
[3] Treveil, Mark, et al. Introducing MLOps. O'Reilly Media, 2020.
[4] Hubáček, Ondřej, Gustav Šourek, and Filip Železný. "Learning to predict soccer results from relational data with gradient boosted trees." Machine Learning 108.1 (2019): 29-47.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Gustav Šír, Ph.D. katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **08.06.2021** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **19.02.2023**

Ing. Gustav Šír, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání	Podpis studenta

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

A framework for large scale assessment of machine learning in sports

Bc. Tadeáš Kyral

Supervisor: Ing. Gustav Šír, Ph.D.

Study Programme: Open Informatics

Field of Study: Artificial Intelligence

May 20, 2022

Aknowledgements

I would like to thank my supervisor Ing. Gustav Šír, Ph.D. for continuous guidance and support throughout the whole development of the thesis. Also I want to thank my family and friends, Hoang and Dominik, for support during my studies.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on May 19, 2022 .....

Abstract

This thesis seeks to examine the design and implementation of a framework for machine learning, namely, the domain of predictive sports analysis. The mentioned framework's main objective is to enable the creation of respective workflows represented by graphs of individual operations ranging from data transformations to model training. Moreover, these workflows are especially complex given the context of predictive sports analysis, mainly due to the time-dependent characteristics of competitors and the frequent hierarchical composition of heterogeneous predictive models. Additionally, this thesis documents the categorisation of existing sports and models allowing a more seamless transfer between domains and the implementation of new workflows. The advantages of such framework are then displayed by presenting selected case studies describing the libraries' possibilities and its use.
Keywords: sport prediction, framework, machine learning workflows

Abstrakt

Práce pojednává o návrhu a implementaci frameworku pro strojové učení v doméně prediktivní sportovní analýzy. Hlavním cílem framworku je umožnení tvorby příslušných workflows, reprezentujích grafy podléhající operacím, od transformací dat až po trénování modelů. V doméně prediktivní sportovní analýzy jsou tyto workflows poměrně komplexní. To je způsobeno především časově závislou charakteristikou soutežicích a častým hierarchickým skládáním heterogenních prediktivních modelů. Součástí práce je také kategorizace existujících sportů a modelů, umožňující snadnější přenos mezi doménami a implementaci nových workflows. Přínostnost frameworku je pak demonstrována na vybraných případových studiích, které popisují možnosti knihovny a její použití.
Klíčová slova: sportovní predikce, framework, machine learning workflows

Contents

List of Figures

Chapter 1

Introduction

Predictive sports analytics gained a lot of popularity in recent years, mainly due to increased computational power and new models and ideas introduced in the machine learning domain. Most sport prediction workflows are based on the same processes consisting of various data loading, transformation, training, and testing parts. But these processes are typically created by many different people, and consequently their interfaces are not unified for efficient reuse. As a result of this, a problem arises in that it is difficult to combine the existing processes or even just their parts, such as the predictive models. This is then desired in common practice, where we typically have our own data and we want to experiment with some interesting models from other works. Or, on the contrary, when we implement a new model and want to run experiments with different kinds of data. If we then have multiple heterogeneous models, data transformations and processes we would like to reuse from others works on sports predictions, it is often easier to just re-implement these from scratch to be used in our workflow, which becomes quickly intangible.

Perhaps surprisingly, the structure of a typical sports prediction workflow is actually quite complex. Competitors in every sport develop and their performance is dynamically changing throughout the years, months and even weeks. This has to be captured in the machine learning process which reflects itself through cycles present in the respective workflows. Another important characteristic is the common usage of outputs of one model as an input to another model, which reflects itself in hierarchical nesting of the nodes in the workflows. Together with the various data transformations and preprocessing steps necessary around these, this presents a considerable complication w.r.t. to the standard, static, linear machine learning pipelines.

1.1 Goals

The goal of this thesis is to design and implement a framework that will be used to build and run complex machine learning workflows for sports predictions. The framework must provide support for different sport data and models for easier reusability and extensibility. Finally, working examples will be demonstrated.

1.2 Structure

To successfully satisfy these goals, this thesis consists of four main parts. First, we get to know some specifics about this problem, gather requirements, and based on them, look at related work in Chapter 3. Then, in the following Chapter 4 our own solution to the problem will be presented. From this we can move to implementation Chapter 5, where we cover the development process. Lastly, selected use cases will be presented to demonstrate the use of the resulting framework for building complex machine learning workflows.

Chapter 2

Background

First it is necessary to establish the basic terminology in terms of sports machine learning. We describe general machine learning, and then some concrete models used for sport predictions. We also explore the question of how we can make the models better with hyperparameter optimization and lastly the whole concept of machine learning operations and workflows, which is the core of this thesis and the framework.

2.1 Machine Learning

Machine learning is a subset of Artificial Intelligence. The widely used definition is credited to Arthur Samuel: "Field of study that gives computers the ability to learn without being explicitly programmed"[19], even though this statement is not exactly given in his work. This gives us an abstract idea of machine learning. In 1997 Tom Mitchell and McGraw Hill came up with a little more formal definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."[14] Some deeper meaning and explanation will be presented in the next section.

There are three main areas of machine learning - reinforcement, supervised and unsupervised learning [18]. The only relevant area for this thesis is supervised learning. It consists of the training phase where the data are used with labels and the testing phase where there are no labels present.

2.2 Models

For now we can view a computer program mentioned in the previous section as a model with parameters. In general, it can be much more complex, with many nested models, etc. The model can learn and improve from experience or, in other words, it updates its parameters to make better predictions for the mentioned tasks. Updates of parameters based on predictions are made through the performance measure, which gives us feedback on how the model is performing. Inputs to models are features,

and outputs are predictions, most often in terms of the probability distribution over outcomes. We will take a look at some basic models and describe them here.

2.2.1 Standard models

We start with models that are commonly used in machine learning. Here we have a vector $x \in \mathbb{R}^n$ of features and a vector $y \in \mathbb{R}^n$ of labels or target values.

Regression aims to predict some continuous value y based on the inputted vector x [2]. Some basic types are Linear regression and Polynomial regression. Linear regression expects linear relationship between x and y and on the other hand Polynomial regression works with relationship of x and y as polynomial function of the nth degree.

Classification does not assign a continuous value, but a set of numerical values that can have some meaning in the real world such as cat and dog, which are called classes [4]. If we have more than two classes it is called multiclass classification. One of such popular classifiers is called SVM (Support Vector Machine).

Neural network is a composition of simple functions, which are commonly called layers [20]. Inside each layer are neurons that accept input x and give output y based on its weights and bias. The last layer is usually some function, for example SoftMax, that normalizes the output values into a distribution. There are many special classes of neural networks, such as the Graph Neural Networks (GNN) [21].

In addition to these we will also cover models that are more specific to sport machine learning in the following sections.

2.2.2 Statistical models

Statistical models work on the basis that the vector x is drawn from some distribution. In the area of result based predictions, we take as x the number of goals scored by each competitor. We present the data to the model and try to adjust the distribution parameters to maximize the likelihood function [11]. We will cover the Double Poisson model and show its basic workings.

Double Poisson model assumes that the number of goals of the competitors in the match is independent and is modeled by the Poisson distribution (Equation 2.2) with means characterizing the strength of the competitor in attack and the weakness in defense [13]. In more detail, the means are defined as Equation 2.1 where Att is the strength in attack and Def weakness in defense, respectively, for the home or away competitor and H is home advantage.

$$\begin{aligned} log(\lambda_H) &= Att_H - Def_A + H \\ log(\lambda_A) &= Att_A - Def_H \end{aligned} \tag{2.1}$$

These means are then used in the Poission models through which we can state the probability of number of scored goals for each team.

$$P(G_H = x, G_A = y|\lambda_H, \lambda_A) = \frac{\lambda_H^x e^{-\lambda_H}}{x!} \cdot \frac{\lambda_A^y e^{-\lambda_A}}{y!} \tag{2.2}$$

This way we have many variables; it is possible to use only one strength parameter for a competitor that halves their number [11].

Some other model is, for example, Bivariate Poisson, which adds dependency between the goals scored to the Double Poisson model.

2.2.3 Rating systems

Rating systems assign ratings to every competitor, and we commonly use these as a feature to other models, for example, classification [11]. The well-known **Elo rating** assumes that the performance of competitors is normally distributed around their skills with some variance [9]. This describes that competitors do not perform to the same level every match and have good days and bad days. The expected outcomes of the match are numerically expressed between zero and one and are computed using the Equation 2.3 for the home competitor and the away competitor, respectively.

$$\begin{gathered} EH = \frac{1}{1 + c^{(RA-RH)/d}} \\ EA = 1 - EH \end{gathered} \tag{2.3}$$

Actual outcomes are defined as one for a win, half for a draw and zero for loss. If we have calculated the expected outcomes, we can update the ratings to better reflect the competitor skill using Equation 2.4 where R represents the competitor's rating, k is a learning rate, δ is an absolute value of the goal difference, and γ is a metaparameter that influences the change in rating based on the goal difference and finally SH is the actual outcome.

$$\begin{gathered} R_H^t + 1 = R_H^t + k(1+\delta)^\gamma \cdot (SH - EH) \\ R_A^t + 1 = R_A^t + k(1+\delta)^\gamma \cdot (SH - EH) \end{gathered} \tag{2.4}$$

Another rating system is TrueSkill [11], which is similar to Elo in using Normal distribution to model skill but here is also updated the variance, and can also be used for multiple competitors in one match [9]. Then we also have Pi-ratings, Berrar and Steph ratings [11]. Rating systems do not have to have just one rating for each competitor, but can have defensive and offensive or home and away or expected number of goals in the match.

2.3 Hyperparameter optimization

Most models have parameters that can specify the structure or some inner workings, such as the speed of convergence. These parameters say how the learning process should go and are specified before the learning starts, and they obviously affect the

overall quality of the predictions. But they are not strictly given, as they can perform on different types of datasets differently, and for that we use hyperparameter optimization, which is the process of searching for the best configuration of these hyperparameters based on performance.

There are many algorithms for such an optimization, for example, grid search or Bayesian optimization [22]. The first one just searches through all possibilities and picks the one with the best result. The Bayesian optimization is well suited for complicated objective functions that are costly to evaluate or have noise. The idea is based on the Bayes theorem with objective function modeled as posterior probability [6].

2.4 Machine learning operations

Machine learning operations (MLOps) is based on a popular set of software development practices called development operations (DevOps), which aims to make software development and deployment easier and faster through automation. MLOps has similar goals, but unlike classic software, machine learning is more complex mainly due to the use of learning data. This adds practices such as data validation and workflows [5]. In Figure 2.1 we can see the intersection of DevOps, Machine learning and Data engineering which results in MLOps. This simply explains that it consists of already mentioned DevOps regarding deployment, then machine learning concerning models and use of data engineering for data processing. Data engineering can be viewed as a workflow of data that is continuously processed. These processes can be cleansing, feature extraction, and data preprocessing in general [17].

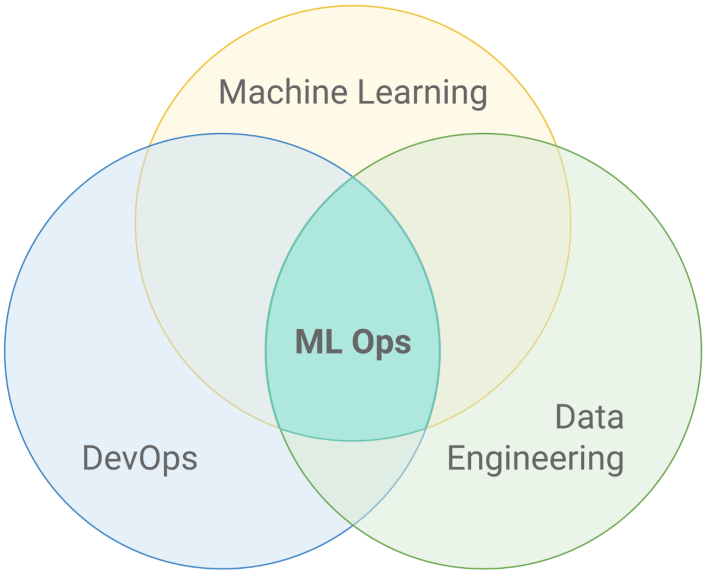

Figure 2.1: MLOps definition diagram [5]

In this thesis, we will not concern ourselves much with deployment, the main topic being data and workflows, which can be viewed as machine learning and data engineering combined.

2.4.1 Workflows

Workflows are a way to automate the process of data extraction, training and testing [8]. We can imagine it as a graph where each vertex represents some functional part and edges specify flow of data between them. In standard machine learning tasks the graph is simple and consists of just a few sequential parts, which mainly are loading data, transformation, extraction of features, training, testing and evaluation. This simple sequential workflow is called a pipeline. However, in general, the workflows can be much more complex. The integral part of workflows is modularity, which means that we can easily change the already mentioned parts, which helps to speed up development.

Chapter 3

Problem analysis

This chapter covers the requirements for the framework. On the basis of them, we can take a look at other frameworks that could satisfy these requirements. But first we will describe the specifics of the data in sports and other specifics emerging from sport related machine learning topics.

3.1 Sport machine learning

Sport machine learning is a little different from normal machine learning, mainly due to the nature of sport data. The main specialty is time dependence, which means that competitors change their performance over time. If we are interested in predicting outcomes of matches of some team, last season or recent matches are much more relevant than how the team was performing five years ago, which actually does not interest us at all. This requires some kind of iterative training and prediction making that enables us to work with time periods. A similar complexity is when we want to use computed outputs from a model as features to another model, but we want both of these learnings to be related to some time period.

3.1.1 Time-dependent learning

We introduced this problem in the previous section and now we will take a look at it in more detail with an example. We stated that we are interested in training and predicting in some iterative manner over time periods. These periods can be, for example, a one-month window, where we take the first month of the dataset, train on it, and then make predictions on the next month, and this continues until we iterate through the whole dataset; this is called rolling strategy. The other strategy is named expanding, which means that the training phase does not consist only of one month, but is expanding, so every new training keeps all the previous months and adds a new month. This enables us to find out how good the model is for continuous prediction with new data coming in every period. We will call these time periods windows and this whole concept window shifting or window shifting strategy, as we are moving these windows based on selected logic.

Then we also briefly discussed the issue of using outputs of some model or models as input to some other model with iteration over time periods. The time periods were already covered in the previous paragraph and we established them as windows. Thus, the iteration is called window shifting. So we have data, two models and for each model one window for training and one for predicting. We also know which model's predictions are used as features to the other one or in other words which model is nested inside the other one. Now we can do one round of training and prediction on the nested model based on the starting state of the windows. This outcome is then used for training and predicting in the outer model, based on the starting state of the windows of this outer model. Then we update all windows to next state and this process keeps repeating. It can also be shown with a simple example of the Elo rating, which we consider as a model and a classifier. First, we make predictions for a month with Elo, so we get ratings for competitors in that month. That we use as features to classifier and that gives us probabilities of the possible results. And then we move on to the next month. Based on the nested nature of the models, we will call this nested learning.

3.1.2 Workflows

Workflows in sport machine learning are usually more complex than what was presented in Section 2.4.1. They are not just simple sequential steps, instead, they can be looked at as a graph with cycles. Cycles are present mainly due to the time-dependent learning mentioned above. In Figure 3.1 such a workflow is depicted using a diagram with data flow and composition.

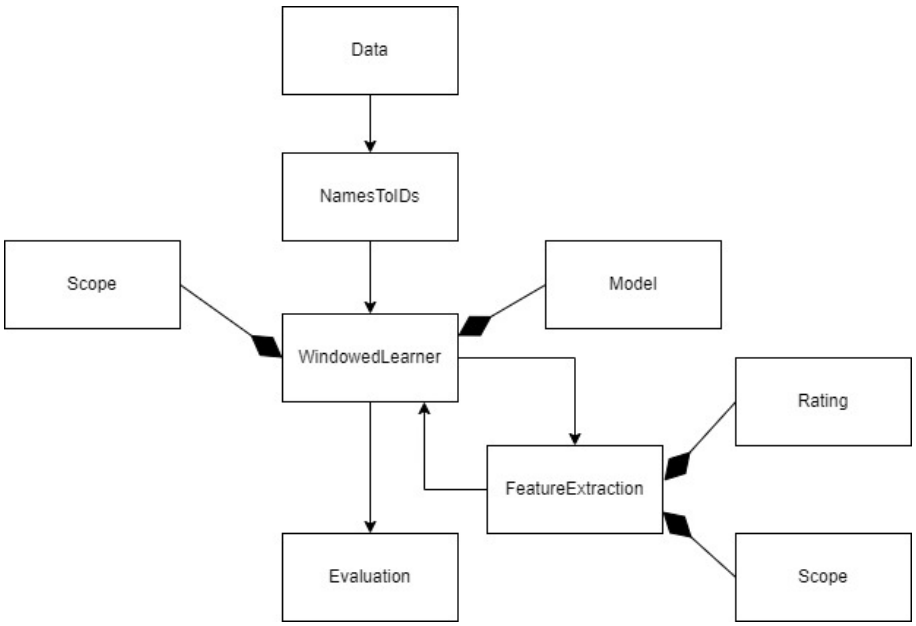

Figure 3.1: A nested workflow diagram

This workflow is similar to the example used in the previous section concerning

nested learning. We load data that are then passed to the transformation, which adds a number identification to every competitor for better manipulation inside the models. Then we have FeatureExtraction that contains some Rating system and scope, which is just an object representing windows shifting strategy. This is inside WindowedLearner which has some model, for example, the regression classifier and scope. The logic is that WindowedLearner passes data to FeatureExtraction, which returns computed features based on the given state of the scope from which WindowedLearner makes predictions based on its scope. This repeats until we iterate through the whole dataset. Finally, we evaluate the performance of the workflow.

3.1.3 Data

Sports data are mostly also complicated as it is difficult to divide sports into strict groups based on their similarity of organization, rules and format. The most obvious division is into matches and races. Matches are played as one competitor against another competitor, competitors can be individuals or teams. This means that match results are given as home and away scores with the names of the two opponents. However, a race involves many competitors and the result is given in most cases as their order. This makes it difficult to use the same models for those two types, even though we can convert race to matches, but that is not very efficient. Then we can also find similarity in the type of sports played in the format of round-robin where each competitor plays every other competitor at least once, which we will call league-based and other group of sports which are based on elimination tournaments. But these are not that strictly given, as many sports based on leagues such as football or hockey have not only competitions as elimination tournaments but also play-offs after the round-robin leagues. This has to be resolved with some preprocessing or selection of nice data as it does not have a general approach. Now we take a look at the relevant features to this thesis.

3.1.3.1 Features

Features are taken from the provided database which is based on data from the Flashcore[1] website. For score-based predictions these are the basic features independent of the type of sport: the date, the name of competition and the season. Match based sports have home and away scores and names for home and away opponent. Races have the ordering and name of the competitor. Some sports, usually elimination tournament based, also have categories, for example ATP (Association of Tennis Professionals), which could be analogous to leagues as in them meet only some group of competitors.

There are, of course, some peculiar cases that do not fit this. For example, boxing is a match but the result is not in the form of a score. But even if the sports have a score, they differ in the quantity of points which is important, as the score or the difference of the score between opponents is input to every model discussed in this thesis.

[1] https://www.flashscore.com/

3.2 Requirements

Requirements were obtained from the needs of the IDA group[2] under the Department of Computers, mainly the part of the group that conducts research on sport predictions. As mentioned in Chapter 1 there is a need for unification to improve and accelerate the development of machine learning workflows. These are the requirements for the framework:

- Ability to create complex workflows with time-dependent learning
- Easily extendable with new models and transformations
- Parameterizable and hyper-parameterizable models and workflows
- Efficient reuse over common sports categories
- Executing parts of workflow in parallel
- Can be deployed to HPC clusters

3.3 Related work

In this section, some of the related frameworks for machine learning operations will be presented and compared with the gathered requirements.

3.3.1 Scikit pipelines

Scikit pipeline is a module from Scikit library for building simple sequential workflows [15]. It does not handle problems that are necessary for this project. Also, as pointed out in the first sentence, it is capable only of creating sequential workflows, which is not enough. Another disadvantage is the narrow scope of Scikit models that must be used.

3.3.2 Ploomber

Very plain and simple framework for constructing workflows [16]. The pipeline is defined through a YAML file, where we characterize for every node the input and output or, in the Ploomber terminology, upstream and product. These have to be serialized, which is not a problem, but there is a problem with importing classes and their methods if used as nodes, which is not very easy to get to work. Another disadvantage is that workflow cycles are not supported, which is the main requirement. A nice feature is the possibility of caching the output of nodes and reusing them in repeated runs.

[2] https://ida.fel.cvut.cz/

3.3.3 Kedro

Kedro belongs to the more popular frameworks. It is used to create machine learning pipelines with good software engineering practices to make the pipelines easily reproducible, modular and well documented [12]. It also contains a nice visualization tool and the overall usability is simple. It supports deployment using many other frameworks, such as Kubeflow [1]. However, it does not support workflows with cycles.

3.3.4 MLflow

Mlflow is mainly a framework for tracking and logging parameters in pipeline runs [7]. It also has project and model components that are meant for easier reproducibility in different environments. Based on that, we can say that this framework has different goals than the previous ones. It certainly does not support any workflow creations on the atomic level with cycles, etc. It is more suited as a secondary tool for the mentioned tracking and it can be easily integrated into a project.

3.3.5 Kubeflow

Widely used MLOps framework of the Kubernetes family of tools meant for Kubernetes containers[3]. Provides a complex software development kit (SDK) for the implementation of pipelines and a user interface for their management and deployment. Everything happens inside the framework, including training, which results in complicated code with a lot of lines that are just for configuration of the framework. It is capable of storing and reusing components for faster building. It is a very powerful framework, which is also its downside given the necessities of this project. Additionally, it does not support the creation of workflows with sport specifics, such as cycles.

3.3.6 Conclusion

Based on the reviewed frameworks and libraries, it can be stated that none of them fully meets the requirements given. Of course, these are not all the projects concerning the studied problem, but these were selected as the ones most commonly used and also as a good representation of the others. Consequently, it is necessary to design our own solution to meet the requirements.

Chapter 4

Design

In this chapter we present the design of the framework with descriptions of how some key parts should work. We start with an architecture that will outline the fundamental components of the project that will be necessary. After that, we look at whether we can find some common elements between sports and group them. There will then be a section on parameterization and how it will be realized. Finally, we discuss usage in terms of server deployment.

4.1 Architecture

The whole framework will be built around workflows and sports. As these are problems that we have already covered, we can use it to analyze which parts will be needed. First, we need some object for loading data from different types of source, which we will call a data loader. Then it is necessary to have an object that holds data and information about the data which will be passed on from one part to another through the workflow, this object will be called data wrapper. In addition, we need feature extractions and other processing of the data, called transformers. Of course, the main part of the framework is training and testing on some model for which we will also need some object that we will name learner. Finally, we evaluate how the model performed. This is a basic description of parts in some workflow. In the following sections, we will characterize these parts in more detail and add some others.

4.1.1 Data Loaders

Data loaders will be used to load data from the source. The source can be a database or a CSV (comma-separated values) file. Every sport will have its own data loader for better usability, as we can pre-specify some parameters for the default database, which is based on data from the Flashscore[1] website.

[1] https://www.flashscore.com/

4.1.2 Data Wrappers

Data wrapper is meant to hold the loaded data and relevant information, such as the type of sport and the names of columns containing features and labels. Each sport will be represented with one wrapper and they will be part of hierarchy based on types of sports such as match, race, league and elimination tournament. Data Wrappers will also be created for specific groups of sports, which are all described in Section 4.2. The idea is to pass the data wrapper from one component of the workflow to another, gradually gaining columns with new data computed from those components.

4.1.3 Transformers

Transformers will be used mainly to extract features from the data or polish the data that are not perfect. Some examples of transformers are converting names of competitors to number identifications, as models do not work with string values, or removing seasons from the data because there are small number of teams competing, which is important for some types of models or feature extractors. There is a need for transformers that keep some inner state, these could be called stateful, then we need also updating transformers that are able to update their state based on some logic.

4.1.4 Scopes

Scopes are for selecting some subset of the data. Mainly used for the window shifting strategy described in Section 3.1.1. Two basic scopes will be enumeration scope for iterating over some list of values and number scope where you need to specify the starting and ending numbers and also size of the step which will be taken every iteration and size of window. The number scope also needs to support the window shifting strategies named rolling and expanding.

4.1.5 Learners

As mentioned at the beginning of this section, the learner will be an object that contains an interface for training and testing. As we have different complexity of learning due to the possible nested learning and window shifting introduced in subsection 3.1.1, it is necessary to have multiple objects for that. Basic learner with scope that only selects data based on scope and performs training and testing without updating the scope. Learner with iterative training, testing and updates of the scope known as window shifting. This one will also support nested learning. Finally, a learner without scope to cover all the possibilities. Any learner will have the possibility to be used only for training or testing.

4.1.6 Blocks

All these components mentioned above are foundation stones from which you can build workflows, but it would be tedious and not effective if it was necessary to write these workflows every time from scratch. Workflows can be very similar, and sometimes the difference is just in a different model or other component. For this case there are components called blocks, which can encapsulate working part of a workflow for reusability. This will make the code more sleak, short and user-friendly.

4.2 Data sport structure

In the previous chapter, we dived into sports and how they have differences among themselves. These distinctions must be taken into account when designing the framework, as it is its main domain. We want to find common elements between sports to create some categorical structure. This allows us to know which sports are interchangable and also we can specify what category of sports can be used as input to models or transformers.

The basic categories are match and race, which were described in Section 3.1.3. Then we can divide sports into individual and team, which would be important for models with more detailed features. These were general characteristics that do not ensure large similarity that would be necessary for learning. For that purpose, it will be useful to create groups containing sports with similar features such as format, rules and mainly mean of goals scored in matches.

As part of the thesis, we identified the following groups:

- Tennis, badminton and beach volleyball
  - Very similar format.
  - Mostly played as best of two and as tournaments with eliminations.
- American football, rugby league and rugby union
  - Similar sports and score.
  - It could appear that Australian football should be included, but the score is much bigger.
- Field hockey and ice hockey
  - Again, very similar in all aspects.

These groups will be represented as different data wrappers, as discussed in the previous section.

Another distinction between sports is mentioned in the description of the groups. Some of them could be described as league based and others as elimination tournament based. This was already mentioned in Secction 3.1.3 and for these cases special data wrappers will be created.

4.3 Platform

There was no requirement for the programming language. The two main choices are Java and Python mainly because of the author's proficiency in those two. Java is more suited for object-oriented programming and the author has more experience in it. On the other hand, almost every relevant sport machine learning project is written in Python and because those projects will be used with this framework, it is much easier to stay in Python. Of course, there is the possibility of calling Python programs from Java, but that would not be efficient, and more importantly, a new project would have to be done in Java as it had to comply to the framework architecture to make use of the framework. From all this, it is decided to use Python as a simpler and more logical solution.

4.4 Parameterization

Parameterization is an essential feature of the whole framework. In the preceding section, we named some basic components and almost all of them need parameterization. Thus, we will describe the basic design in detail here.

It will be realized through a JSON file. This format is easily readible, which is very important as it will be up to the user to understand it and make changes. In the file will be parameters ordered in reflection of the structure of the workflow, that should enable simple editability as you know exactly what you are editing even in complex workflows and also give the user feedback whether the structure is what it should be.

```
{
    "learner": {
        "model": {
            "alpha" : 1,
            "type" : "lgbfs"
        }
    }
}
```

Listing 1: JSON example as parameters file

Another point is hyper-parameterization that has to be somehow connected with the normal parameterization. The best solution for now is to use a different file for that, where you just specify what parameters you want to optimize, but in the same structure. Because the optimizer needs more information than just one value, the structure will be a little bit more complex.

```
{
    "learner": {
        "model": {
            "alpha" : {
                "lb": 0,
                "ub": 1,
                "precision": 0.1
                }
            "type" : "lgbfs"
        }
    }
}
```

Listing 2: JSON example as hyper parameters file

A more detailed description will be further given in Chapter 5.

4.5 Cluster Usage

One of the requirements is to be able to run workflows on a server and use parallelism. In the previous chapter, we covered some frameworks that are used for running workflows. Because they are already implemented for this task, we can use them on top of this framework. The most suitable one is Ploomber as it is easy to use and satisfies the server requirements. This does not mean that the framework will not be runnable without it, nor that the Ploomber will be enforced.

Chapter 5

Implementation

In the previous chapter, we designed our own custom solution that contains a framework to create learning workflows in the sport machine learning environment. In this phase, we are going to implement that solution and describe the individual steps. To best describe the key parts, we will use code snippets taken directly from the custom framework implementation, which will demonstrate the discussed functionality better than an auxiliary textual description. [1]

5.1 Parameterization

For the purpose of parameterization, we created a `Parameterizable` class. As it is very important, we will describe it in a little more detail. The constructor accepts three arguments: *parameters*, *hyper_parameters* and *name*. *Name* variable is string, *parameters* is the dict type or `Parameters` class and *hyper_parameters* is `HyperParameters` class. The general idea is to have default values for *parameters* and *name*, which is achieved through the class variables *name* and *init_parameters*, which all inheriting classes should override if its relevant to them, as seen in Listing 3. These are then set in the *__init__* method as instance variables mainly using the *set_parameters* method. Setting variables to default values can be done using the *reset_state* method, which can be useful in the optimization process.

[1] Note that all the library's code is authors own work

```
name = "scikit_classifier"
init_parameters = {
      name: {
          "model_type":"LR",
          'model_params':{
              "multi_class": "multinomial",
              "solver": "lbfgs"
          }
      }
  }
```

Listing 3: Default parameters of `ScikitClassifier` class

The method *set_parameters* shown in Listing 4 accepts *parameters* and checks if they are relevant to the instance by name. If they are, it iterates over *init_parameters* and takes values from the variable *parameters* based on the key; if not present, it uses the default value.

```
def set_parameters(self, parameters:Parameters):
    """Sets parameters based on input and class variable
    ↪  init_parameters.

    Parameters
    ----------
    parameters : Parameters
        Parameters/dict object containing parameters for given
        class
    """
    if self.name in parameters:
        self.parameters = parameters[self.name]
        if self.init_parameters is not None:
            for param,value in
            ↪  self.init_parameters[self.name].items():
                if param in self.parameters:
                    setattr(self,param,self.parameters[param])
                else:
                    setattr(self,param,value)
    if 'optimization' in self.parameters:
        self.optimization = self.parameters['optimization']
```

Listing 4: Method *set_parameters* in `Parameterizable` class

Sometimes we need to set as a value some class or even method. For this purpose, it is better suited to use as a holder for parameters the dictionary type in Python file as opposed to JSON file, so we do not have to concern ourselves with transforming the dot convention and creating instances, etc. We can simply import the class and

use it in the parameters dictionary as shown in Listing 5. This simplifies everything, even the user experience. And it also preserves the readability of the JSON format, which is very important.

```
params = {
    'load_block': {
        'filter':Equal("Lge", "GER1"),
        'wrapper':{
            'class':FootballWrapper,
            'Football':{
                'football':{
                    'PSQLSchemaName':'isdb',
                    'DB_name':'bet'
                }
            }
        }
    }
}
```

Listing 5: Example of parameters dictionary for loading block

5.2 Optimization

Represented with the `Optimizable` class, which sets the hyper parameters and performs optimization. It is a subclass of `Parameterizable` and the idea is that all classes that should be optimized inherit from this class. If they are optimizable, it means that they do some computation that can be performed better with different parameters. That is why we introduce the *compute* method, which is important as from now on the only method that is necessary to call in order to execute functionality regarding some object in the framework is this one. This makes usability easier, as we can swap classes and the code can stay the same.

Another useful method is *optimize* shown in Listing 6. As you can see, we can set a scoring function, which is used to lead the optimization in the direction we want and the type of optimization framework. For now, there are two frameworks that perform the optimization Optuna[2] and Ray Tune[3]. Optuna is lighter and provides a smaller number of tools for parameterization. The big difference is in setting of hyper parameters, which we will discuss in the next section. The return value is an object from the completed optimization that contains all the information about the best run.

[2]https://optuna.org/
[3]https://docs.ray.io/en/latest/tune/index.html

```python
def optimize(self, hyper_parameters:dict, wrapper:BaseDataWrapper, score_function:Callable=None, type=TypeOfOptimization.OPTUNA, **kwargs):
    """Runs optimization on the class

    Parameters
    ----------
    hyper_parameters : dict
        hyper parameters as a dict
    wrapper : BaseDataWrapper
        wrapper on which optimize
    score_function : Callable, optional
        evaluation function that is used for optimization
    type : TypeOfOptimization, optional
        specifies type of optimization OPTUNA or RAYTUNE
        (default is OPTUNA)
    kwargs
        other parameters passed to selected optimizer
    """
    if hasattr(self,'optimization'):
        if 'score_function' in self.optimization:
            score_function = self.optimization['score_function']
        if 'parameters' in self.parameters:
            kwargs = self.optimization['parameters']
    result = None
    if type == TypeOfOptimization.OPTUNA:
        result = self._optuna_optimize(hyper_parameters, score_function, wrapper, **kwargs)
    elif type == TypeOfOptimization.RAYTUNE:
        result = self._ray_tune_optimize(hyper_parameters, score_function, wrapper, **kwargs)
        return result
```

Listing 6: Method *optimize* in the `Optimizable` class

5.2.1 Hyperparameters

Hyper parameters use the format introduced in Listing 2. Which is an input from the user, but we need different formats for Ray Tune and Optuna. Because Optuna uses a study object and trials, it means that in every iteration it is necessary to suggest the parameter space using the trial and obtain the value. On the other hand, Ray Tune sets the parameter space once at the start, and during iterations we obtain the values directly. So we have two classes; one is `RayTuneHyperParameters` that is initialized with Ray Tune values, and the other `OptunaHyperParameters` that contains the inputted hyper parameter values and also the trial object which is then

used in the object we want to optimize for obtaining the parameters. Unfortunately, Optuna does not support nested hyper parameters which means we are not able to easily optimize parameters based on values of other parameters.

5.3 Data Loaders

Data loaders are responsible for retrieving data mainly from a database, but it can also be from CSV. To select the type of source, we can use the enum class `SourceType`, which has two values for now: PSQL and CSV.

As can be seen in the Figure 5.1 diagram, there are classes `MatchDL` and `RaceDL` that do basic parsing on the data depending on the type. Every sport has its own data loading class that inherits from one of those classes. We can also see that there is `WinterSportDL` that covers almost all winter sports such as alpine skiing or classic skiing. Because this hierarchy was created based on the database used, it contains these groups, but it does not put any constraints and it is still possible to use a database with different schema.

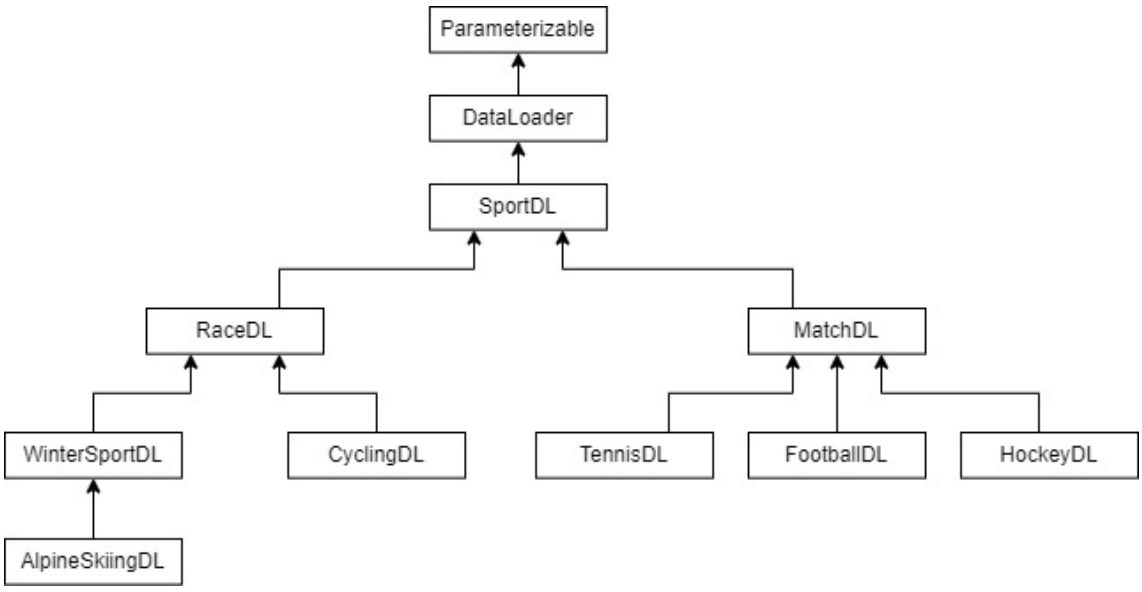

Figure 5.1: Sample of data loaders class hierarchy

5.3.1 SQL builder

This class serves as builder for SQL queries based on the passed name of the schema and the table and the arguments for the where clause. These arguments are ordering as a tuple value and filter as Filter class.

5.3.2 Filter

Filter is a general class for making complex filters with binary operators *AND* and *OR*. We can specify attributes with constraints as equal, greater than, and between. As seen in Listing 7 it is possible to nest filters inside each other as you would normally in SQL query. The advantage is that it can be reused for different databases and sources.

```
And(Equal("Lge", "GER1"), Between("Sea",2003,2013))
```

Listing 7: Filter usage example

5.4 Data Wrapper

As discussed in Section 4.1.2 data wrapper holds data and is passed between different parts of the workflow. The idea is not to use the `DataLoader` class explicitly, but to specify the wrapper that knows which data loader to use. Each wrapper can have one or more data loaders according to the type of wrapper and the type of data it represents. This means that calling the *compute* method calls the data loader to obtain the data. These data are then passed to the `DataHandler` object, which is inside the wrapper and performs basic operations on the data, such as adding columns and rows. Therefore, it is not just a wrapper for the data, but its job is also to load the data through `DataLoader`. The data are in the form of a Pandas library class `DataFrame`[4] and, as mentioned earlier, in class `DataHandler`. The wrapper then adds more specific information about the nature of the data. This leads us to the hierarchy in Figure 5.2, which is similar to that of the data loaders Figure 5.1.

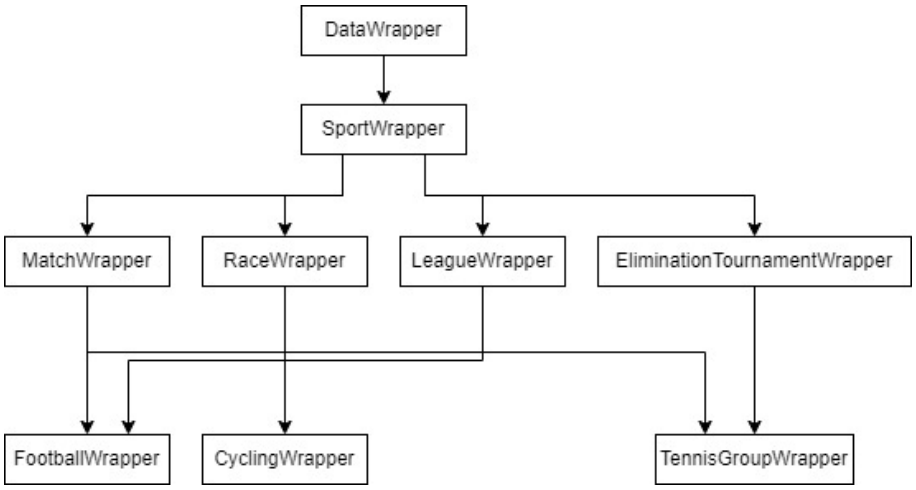

Figure 5.2: Sample of data wrappers class hierarchy

[4]https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

We have basic wrappers such as `MatchWrapper`, `RaceWrapper`, `LeagueWrapper`, `EliminationTournamentWrapper` that specify the names of important columns and other relevant information. Then there are concrete sport wrappers and group wrappers. Group wrappers mirror groups from Section 4.2 and use more data loaders, which means that for each sport, it contains its data loader. The data of all data loaders are then merged into one `DataFrame`.

5.5 Transformers

We have three basic transformer classes: `Transformer`, `StatefulTransformer` and `UpdatingTransformer`. The stateful one has state, and in order to do some transformation we first call the *fit* method that performs fitting of the data and then the *transform* method. If we use updating, there is the *update* method, which is responsible for updating the state.

5.5.1 Feature Extractor

Building on top of the transformers, we have the `FeatureExtractor` class, which, as the name suggests, obtains new columns from the given data and adds them as feature columns. For example, the `NamesToIds` class assigns to each competitor a numerical identification, which is then added to each row in `DataFrame` in the corresponding way. The first part is in the method *fit* and the second part in *extract_features*. As we mentioned earlier, because it has the *fit* method, it is stateful `FeatureExtractor`. We have several classes of `FeatureExtractor`, some others are `DateFromTime` and `ScoreDifference`.

Another group inheriting from the `FeatuerExtractor` class is `StatefulFeatureExtractorModel`, where two parts meet, namely models and feature extractors. This was created to be able to use feature extractors in the learner.

5.6 Learner

Learner represents the training and testing process, where one of them can be omitted. It accepts a scope to select data from the wrapper, `Trainer` and `Testing` classes that hold the model and provides an interface for training and testing, respectively.

5.6.1 Windowed Learner

Windowed learner is a learner with updating which means it iteratively trains and tests on selected data based on the scope that is updated every iteration, as discussed in Section 4.1.5. Other important functionality is nested learning that is done through learners or feature extractions (discussed in the next section), which are passed as an argument and then called before every train and test iteration. As can be seen in Listing 8.

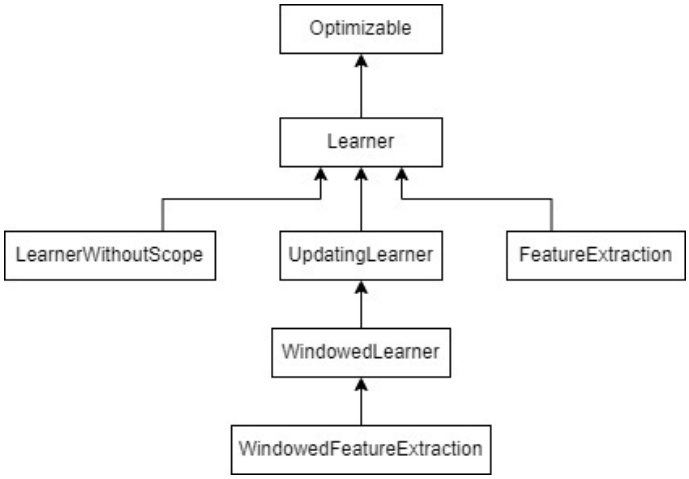

Figure 5.3: Learner class hierarchy

5.6.2 Feature Extraction

Feature extraction is renamed learner for feature extractors, it has the same functionality, but we want to distinguish between them because learner is used for models and it does not make sense to have learner for ratings and other feature extractors, as they usually do not learn anything. There is also a windowed version of this class.

```
def train_test(self, wrapper: BaseDataWrapper):
    outputs = []
    copy = wrapper.deepcopy()
    # iteratively check if still within dataset scope
    while self.scope.holds():
        if self.learners:
            wrappers = []
            for learner in self.learners:
                wrappers.append(learner.compute(copy))
                learner.update()
            wrapper = self.merger.compute(wrappers)
        outputs.append(super().train_test(wrapper))
        self.update()

    if self.tester is not None:
        if len(outputs) == 0:
            return pd.DataFrame()
        features = pd.concat([data for data in outputs])
        return features
    return None
```

Listing 8: Method *train_test* in the `WindowedLearner` class

5.7 Models

Model class has two main methods *fit* and *predict* that do not accept `DataWrapper` object but `DataFrame` that is prepared in the `Trainer` and `Tester` classes mentioned above. This way we can somehow separate models from the framework, which can be useful for feature use. But if it is necessary to also pass information about the data, there is the method *set_parameters_from_wrapper* that takes `DataWrapper` as an argument and can be implemented in concrete models. If we want to implement some model, we just have to inherit from this class and it is pretty straightforward. There are now two exceptions, which are neural models and rating systems.

5.7.1 Neural models

Currently implemented neural models in the framework use the widely known Torch library. This comes with a problem, as it is needed to use their class as a parent, etc. For that case a `TorchModule` class was created, which inherits from Torch's `Module` class and implements its methods. Then we use it in our own classes for neural models through composition. This way it is possible to replace Torch with a different library.

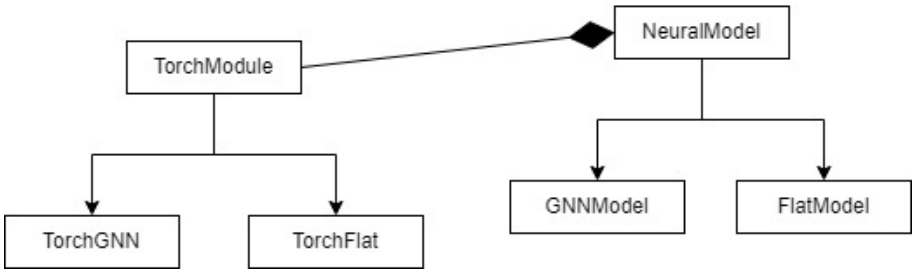

Figure 5.4: Neural model class hierarchy

5.7.2 Ratings

As mentioned in Section 5.5.1 ratings are more of a feature extractor, as they do not train themselves. But we want to use them the same way as models, so a `FeaturesExtractorModel` class was created from which the previously discussed `StatefulFeatureExtractorModel` inherits and consequently also the ratings.

5.8 Scope

We define two main classes of scopes, that is, `EnumScope` and `NumberScope`. The first is for iterating over some list of values in a given column, and the other is for iterating from some starting number to ending number by strides while maintaining

some size of the window or expanding it. As always, these values are set through the parameters. Some parameters can be omitted and set from the data using the *set_parameters_from_wrapper* method.

Usually, there are two scopes used, one for training and one for testing. The testing scope is usually dependent on the training one because we want to perform testing on the data that closely follow the training data. So, for this purpose, a special `TestingNumberScope` was created that accepts scope and its values are set on the basis of it. This mainly applies to hyper-parameter optimization as we want to optimize in synchronization both scope and not have nonsensical values in them.

5.8.1 Data Selector

`DataSelector` class is meant to perform logic of multiple training and testing scopes and their updating, transforming etc. It can be seen as interface for these scopes which results in simpler interaction as we can update both with one method. This method is called *update* and performs updates on the training and testing scopes in synchronized way, so we iterate through all possibilities. It is possible to have different number of training and testing scopes, which enables having distinct granularity of training and testing. The logic of iteration starts on the last scope so if we have two scopes for testing and only one for training the first iteration updates the second scope in testing and the one in training. This can be problem as we can have completely different data in training and testing. However as it goes through all possibilities we can switch the order of the two scopes and it will work as intended.

5.9 Blocks

The idea behind the blocks is to make it easier to implement new workflows with prepared cases of useful code blocks. We will cover the basics of the `Block` class and also some created example blocks. The creation of our own block and usage will be presented in Chapter 6.

The `Block` class as all children of `Optimizable` has *compute* and methods regarding the setting of parameters. Here, we also have the *create_structure* method in which we define the structure of the block with just initialization of classes and passing these objects into other objects if necessary. This method is called in the *set_parameters* method, which does not make much sense, but it is important for the possibility of parameterizing also the content of the *create_structure* method. To actually run the computation of the initialized classes, we call the *compute* method as always. This separation enables us to initialize blocks and retrieve configurable parameters without performing any computation, which of course is valid for all classes in the framework, as all have the *compute* method, which makes it then possible to have this feature in the `Block` as well.

We will now cover two block examples that are pre-prepared for usage. These blocks show the usual motivation behind their creation.

5.9.1 Load Block

Load block serves for creating `DataWrapper` based on the parameters. The wrapper is set through the parameters file and initialized during that process. In this case, the *create_structure* method is empty since this block is simple. It only does the initialization and passing parameters forward. After calling *compute*, we obtain a wrapper object with data. Example parameters can be seen in Listing 5 and even more in use cases in Chapter 6.

5.9.2 Statistical Block

This block is intended for basic statistical learning. This time *create_structure* is more interesting because it contains more logic. We start by initializing training and testing scopes, which are then passed to the `WindowedLearner` class with default Bivariate Poisson model. Then in *compute*, the parameters of the wrapper are set to the scopes, and computation is called on the learner. There are also two attributes called *group_training* and *group_testing* that allow the selection of different scopes. This can be useful if we want to train on the data of some group of sports and then test only on the data of one of the sports. Then we do not need the enumeration scope for selecting sport in the testing phase.

Chapter 6

Use cases

Following implementation, we will demonstrate few selected use cases of the framework. We start with some basic cases to demonstrate the general idea, and then add a little more advanced ones. [1]

6.1 Basic

General structure of a basic workflow is to create parameters, then load data that can go through some transformations, then perform training and testing and finally evaluate the results.

6.1.1 Print parameters

First we take a look at printing parameters which are not a workflow by itself, but it is an important feature for their creation. Every instance of a class that is parameterizable has the method *print_parameters*, if we did not pass any parameters, it will print the default parameters. In this way, it is possible to get the hierarchy of parameters that we can use as a template and change values as needed.[2]

```
LoadBlock().print_parameters()
RatingBlock().print_parameters()
```

Listing 9: Rating workflow

We will show a simple conversion of the output to the input for the `LoadBLock`. In Listing 10 we can see that it starts with the key *load_block*, which is necessary for the `LoadBlock` class to know that these parameters belong to it. Under filter key, some source filtering can be put on the data, for example Listing 7, then we have

[1] As in the implementation in Chapter 5, all code snippets, such as workflows or parameters, are part of the custom project and are used to better explain the usage.

[2] Unfortunately, not all values correspond to the input values needed. It is necessary to replace false with False, true with True, null with None and class string with the actual class.

the wrapper key through which we can specify the type as can be seen under the key *class*. The next item is *Futsal*, which again corresponds to the name of the wrapper used. Then we have parameters for the data loader which is under the *futsal* key as it represents `FutsalDL` class, and there we can define information about the source connection, etc.

```
{
    "load_block": {
        "filter": null,
        "wrapper": {
            "class": "<class
            ↪  'datawrappers.sport.match.FutsalWrapper.FutsalWrapper'>",
            "Futsal": {
                "futsal": {
                    "PSQLSchemaName": "futsal",
                    "streaming": false,
                    "chunksize": 10000,
                    "source_type": "<SourceType.PSQL: 'PSQL'>",
                    "DB_name": "flashscore",
                    "PSQLTableName": "Matches"
                }
            }
        }
    }
}
```

Listing 10: Printed parameters of the rating workflow

Now we must change the class string to the actual class, and depending on our necessities, we can omit some parameters. The resulting dictionary can look like the one in Listing 5 or even just like Listing 11.

```
{
    "load_block": {
        "wrapper": {
            "class": FootballWrapper
        }
    }
}
```

Listing 11: Parameters dictionary input for the rating workflow

6.1.2 Rating workflow

The goal of this workflow is to calculate the ratings and use them in the Scikit classifier to train and predict the result. We start by loading the parameters from the ratingParams file, which contains a dictionary named *params* for this workflow. This dictionary is passed to the constructor of the `Parameters` class, which is just a holder for parameters. Then we use `LoadBlock`, which we pass the parameters and call the *compute* method, which returns a wrapper with data. The type of wrapper is specified through the parameters shown in Listing 5. Next, we are interested in adding identification numbers to competitors, which we will do in `TransformBlock` where we specify to use the `NamesToIds` transformation, which is done by this Listing 12 in parameters.

```
1 'transform_block':{
2     'names_to_ids':True
3 }
```

Listing 12: Parameters for transformation block in rating workflow

The output of `TransformBlock` is a wrapper, but is not the same instance as the input. This is valid for all *compute* methods. In line four, we just instantiate `RatingBlock` with parameters, so we have a reference to it. `RatingBlock` contains the learning and testing with windows shifting strategy. Again, *compute* is called with the wrapper and returns the wrapper with probabilities. Then we use `EvaluationBlock` which computes the ranked probability score (RPS) and the accuracy. Finally, we can use the method *print_parameters* to print the dictionary with the parameters in the console, which can be used to control the set values.

```
1 params = Parameters(params=ratingParams.params)
2 wrap = LoadBlock(parameters=params).compute()
3 transformed_wrap = TransformBlock(parameters=params).compute(wrap)
4 rating = RatingBlock(parameters=params)
5 prob_wrap = rating.compute(transformed_wrap)
6 EvaluationBlock().compute(prob_wrap)
7 rating.print_parameters()
```

Listing 13: Rating workflow

6.1.3 Optimization

Optimization is very important and we will show how to optimize attributes within `RatingBlock`. Specifically, which rating model has better performance if Elo or Berrar. Because it is the rating workflow mentioned above, several first steps are almost the same. In this case, we also have to load the *hyper_parameters* dictionary which is located in the same file as normal parameters. In Listing 14 we can see that

it represents the workflow hierarchy and that we want to optimize the model, which is decided by the values key. The value of this key is a list of dictionaries to optimize between.

```
hyper_parameters = {
        'rating_block': {
            'learner': {
                'feature_extraction':{
                    'model': {
                        'values': [
                            {'class': EloRating},
                            {'class': Berrar}]
                    }
                }
            }
        }
    }
```

Listing 14: Rating hyper parameter dictionary

Optimization itself is defined with the `OptunaMetaOptimizer` class, to which we pass `RatingBlock` and *hyper_parameters*. And, as always, we call *compute* to run the optimization. The result contains the return value discussed in Section 5.2. It is also possible to use the Ray Tune optimizer with the `RayTuneMetaOptimizer` instead of the Optuna one. Or we do not have to use these classes and call the *optimize* method on the `RatingBlock` instance, as shown on the ninth line of Listing 15.

```
1 params = Parameters(params=ratingParams.params)
2 hyper_parameters = ratingParams.hyper_parameters
3 wrap = LoadBlock(parameters=params).compute()
4 wrap = TransformBlock(parameters=params).compute(wrap)
5 rating = RatingBlock(parameters=params)
6 result = OptunaMetaOptimizer(rating, hyper_parameters)
7         .compute(wrap)
8
9 #result = rating.optimize(hyper_parameters, wrap,
  ↪  type=TypeOfOptimization.OPTUNA)
```

Listing 15: Optimization workflow

6.2 Advanced

In the previous section, we covered some workflows that should have helped with a basic understanding of the creation of workflows. Now we can move to a little bit

more advanced use cases including learning with more type of sports and creating our own custom blocks.

6.2.1 Rating and statistical optimization

For the case where we want to optimize between two or more models that learn differently, for example, statistical models and rating systems with classification model, we use the `OptimizationBlock` class. This block contains a variable *block*, which we can parameterize. In Listing 16 we can see how the hyper parameters are defined. We use two categorical values `RatingBlock` and `StatisticalBlock`. It is also possible to optimize nested parameters, which can be seen under the key *params*, where we optimize the alpha parameter of the Bivariate Poisson model.

```
'optimization_block': {
    "block": {
        'values': [
            {"class": RatingBlock},
            {'class': StatisticalBlock,
             'params':{
                'statistical_block':{
                    'learner':{
                        "model": {
                            "bivariate_poisson": {
                                "alpha": {
                                    "lb": 0,
                                    "ub": 0.1,
                                    "precision": 0.001
                                }
                            }
                        }
                    }
                }
             }
            }
        ]
    }
}
```

Listing 16: Hyper-parameter dictionary for the optimization workflow

Also, it is possible to parameterize other parameters that are not in hyper-parameters but are nested in the categorical values. We just define it in the normal parameter dictionary with the same hierarchy as can be seen in the hyper-parameters but, of course, without the *values* key for optimization.

6.2.2 XGBoost workflow

In this example, we will show an XGBoost workflow with computation of several features. The first two steps are the same as always, then instead of `TransformBlock` we use the atomic transformations. They are computed using *compute* and the returned wrappers are then passed to the *compute* method of the `Merger` class, which merges the wrappers. This is useful for making computations parallelizable, and it is not restricted to just transformers, but we can basically parallelize this way any number and any type of computation from which we obtain a wrapper.

```
1  params = Parameters(params=xgboostParams.params)
2  wrapper = LoadBlock(parameters=params).compute()
3
4  wrapperids = NamesToIds(parameters=params).compute(wrapper)
5  wrapperrnd = RoundFeature().compute(wrapper)
6  wrapper = Merger().compute([wrapperrnd,wrapperids])
7
8  wrapper = XGBFeaturesBlock().compute(wrapper)
9  prob = XGBBlock(parameters=params).compute(wrapper)
10 EvaluationBlock().compute(prob)
```

Listing 17: XGBoost workflow

Then `XGBFeaturesBlock` is used, which contains three different feature calculations. These are described in [10], and on the basis of this article they were reimplemented to be used in the framework. The output features are then passed to `XGBBlock` where we use the XGBoost model to predict the results.

6.2.3 Creating custom blocks

Already mentioned blocks and other blocks implemented in the framework were created for some basic repeating use cases, such as making predictions based on ratings. It can be expected that with time new models and new structures of workflows will be added, and there will be a necessity of reusing some part of codes, etc. So, we will now show how to create our own blocks that can be used in different workflows without duplicated code. We start with a short example of the code in Listing 18 and move it to a new custom block. In the actual framework, we can find different example of this problem, but it contains a large chunk of code that is not required for the demonstration of the process.

```
1 wrap = WindowedFeatureExtraction(TeamFeatures(), parameters=params,
  ↪ name='team_features_extraction').compute(wrap)
2 wrap = LeagueFeatures(parameters=params).compute(wrap)
3 wrap = WindowedFeatureExtraction(RankFeatures(), parameters=params,
  ↪ name='rank_features_extraction').compute(wrap)
```

Listing 18: Workflow example for creation of custom block

In this code, we are actually on line eight of Listing 17 and these three feature computations are inside `XGBFeaturesBlock` and we will show how this block could have been created. So, we already have some wrapper with data under the variable *wrap*. We can see that two of the computations are inside `WindowedFeatureExtraction` and we pass them the *name* argument, so it is possible for us to distinguish between them when we define the parameters. First, we create a new class that we will call `CustomBlock` but the name is up to the user. It needs to inherit from class `Block` which will require implementing three methods *create_structure*, *compute* and *reset_state*, which all will be covered later. Also, it is necessary to define class variables *name* and *init_paramaters*, *name* is important for setting parameters as it will be used as a key. *Init_parameters* does not have to be implemented if there is nothing to parameterize and the parameters are only passed to the classes contained in it.

Now, we will populate the *create_structure* method. For this, we need to separate instantiating of classes from computation and also omit passing parameters inside it. In addition, it is necessary to create variables for each class so that we have a reference to them. The resulting content can be seen at line five of Listing 19. After that, we implement *compute* which just calls the *compute* methods on every instance created in the *create_structure*. It can also contain different logic as this is the only method in which we have access to the wrapper, for example, we can pass the wrapper to the scopes that were not set through parameters. Next, we have the *set_parameters* method, where we set parameters for all instances, and also using the super call, we can set parameters for this block. Lastly, we implement the *reset_state* method, which calls this method only on the two feature extractions because it is not necessary to call it on `LeagueFeatures` since this class does not have any state.

```
class CustomBlock(Block):

    name = 'custom_block'

    def create_structure(self):
        self.team_features =
            WindowedFeatureExtraction(TeamFeatures(),
            name='team_features_extraction')
        self.league_features = LeagueFeatures()
        self.rank_features =
            WindowedFeatureExtraction(RankFeatures(),
            name='rank_features_extraction')

    def compute(self, wrapper):
        wrapper = self.team_features.compute(wrapper)
        wrapper = self.league_features.compute(wrapper)
        wrapper = self.rank_features.compute(wrapper)
        return wrapper

    def set_parameters(self, parameters):
        super(XGBFeaturesBlock, self).set_parameters(parameters)
        self.team_features.set_parameters(self.parameters)
        self.league_features.set_parameters(self.parameters)
        self.rank_features.set_parameters(self.parameters)

    def reset_state(self):
        self.team_features.reset_state()
        self.rank_features.reset_state()
```

Listing 19: Examplary implementation of custom block

6.2.4 Rating with different sports

In this use case, we take a look at working with different sports and using them together. For that, the rating workflow presented in Listing 13 will be used. The motivation is to be able to seamlessly switch between different sports and also not use just one sport, but also train on some group of sports and do predictions only for a subset of them. It will be shown using the tennis group containing tennis, badminton and beach volleyball. The basic structure of the workflow is in Listing 20, the first four lines are loadings of different parameter files, and after that we have a well-known rating workflow.

```
1 params = Parameters(params=eloTennisParams.params)
2 #params = Parameters(params=eloTennisGroupParams.params)
3 #params = Parameters(params=eloBadmintonParams.params)
4 #params = Parameters(params=eloBadmintonGroupParams.params)
5
6 wrap = LoadBlock(parameters=params).compute()
7 wrap = TransformBlock(parameters=params).compute(wrap)
8 prob = RatingBlock(parameters=params).compute(wrap)
9 EvaluationBlock().compute(prob)
```

Listing 20: Rating workflow with different sports

So the interesting part is happening in the parameter files, the first one is simple training on the tennis data and also testing on it. Then we have training on the whole group and testing only on tennis, which is done by defining the scopes. In the `RatingBlock` are special variables Listing 21 that decide if we want to use `EnumScope` or not. In this case, we want to compute ratings and also train the Scikit classifier on every sport but we want to make a prediction just for one, which corresponds to the parameters.

```
1 'group_training_extraction': True,
2 'group_testing_extraction': True,
3 'group_training': True
```

Listing 21: RatingBlock variables for group learning

Then it is also necessary to specify what sport we want to test on. This is done as in Listing 22 using parameters for the `EnumScope` with only tennis as the sport to predict.

```
1 "enum_scope": {
2     "col": "Sport",
3     "enum": [
4         "tennis"
5     ]
6 }
```

Listing 22: Enumeration scope with tennis value

The next parameter file uses just badminton data, and the one after that is similar to the thoroughly described tennis group one, but instead of tennis, we predict outcomes of badminton matches.

Of course, it is also possible to use different rating systems or any other thing already mentioned, so we can easily change the whole idea of the workflow just through the parameters.

Chapter 7

Conclusion

The main goal of this thesis was to design and implement a framework for large scale assessment of machine learning in sports. This consisted of a few partial goals presented in Section 1.1. First, we covered some necessary background theory to understand the problem. We defined general machine learning processes, presented machine learning models relevant to this thesis and also went over machine learning operations and workflows which are main topic in this work. Then we dived into the problem itself while describing sport machine learning and how it is different from classic machine learning, mainly due to time dependencies and nesting. We also covered the basic structure of sports data and its features. Then we presented requirements on the framework. We looked into related works, which were mainly frameworks for machine learning operations. None of them met the imposed requirements, therefore we decided to design our own framework.

We started with designing architecture where we came up with basic components from which the framework should consist to work as intended. This was done on the basis of the requirements with the use of an example workflow. We also came up with the hierarchical structure of sports to be used in the framework.

Then we moved on to implementation, which describes the individual components of the framework and how they were implemented. After the implementation, we were able to cover a range of use cases and demonstrated how the framework works.

As required, the framework provides categorization in terms of the hierarchy for sports and also models, which is introduced for easier usage of the framework, which is discussed mainly in sections concerning solution and implementation. It supports multiple types of sports, which is demonstrated in the use cases section. Finally, we created use cases to show how the framework can be used to build workflows.

During the course of the development, we faced countless problems that emerged from the specifics of sport machine learning, but have been successfully dealt with. As discussed, part of this work was the creation of working use cases, for which it was necessary to implement selected models presented in this thesis. Although it was to an extent possible to reuse implementations from some other works, it was by far not as straightforward, and required quite some time and effort. That, in turn, nicely emphasizes the added value of this framework, which spares its future users of that hassle.

7.1 Further extensibility

As this project was concerned with the creation of an extensible framework, there is an expectation of adding new models, transformers, and other functional blocks. However, the framework should already be prepared for it. A very suitable extension would be to incorporate the MLflow framework for tracking runs and experiments, which was not managed as part of this thesis.

Bibliography

[1] Sajid Alam, Lorena Bălan, Nok Lam Chan, Gabriel Comym, Yetunde Dada, Ivan Danov, Lim Hoang, Rashida Kanchwala, Jiri Klein, Antony Milne, Joel Schwarzmann, Merel Theisen, and Susanna Wong. Kedro, 2022.

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

[3] Ekaba Bisong. *Kubeflow and Kubeflow Pipelines*. 2019.

[4] U. Braga-Neto. *Fundamentals of Pattern Recognition and Machine Learning*. Springer International Publishing, 2020.

[5] Cristiano Breuel. Ml ops: Machine learning as an engineering discipline, 2020. <`https://towardsdatascience.com/ml-ops-machine-learning-as-an-`> <`engineering-discipline-b86ca4874a3f`>, accessed on 10. 5. 2022.

[6] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, 2010.

[7] Andrew Chen, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Avesh Singh, Fen Xie, Matei Zaharia, Richard Zang, Juntai Zheng, and Corey Zumar. Developments in mlflow: A system to accelerate the machine learning lifecycle. 2020.

[8] H. Hapke and C. Nelson. *Building Machine Learning Pipelines: Automating Model Life Cycles with TensorFlow*. O'Reilly Media, Incorporated, 2020.

[9] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill(tm): A bayesian skill rating system. In *Advances in Neural Information Processing Systems 20*. MIT Press, 2007.

[10] Ondřej Hubáček, Gustav Šourek, and Filip Železný. Learning to predict soccer results from relational data with gradient boosted trees. *Machine Learning*, 108, 2019.

[11] Ondřej Hubáček, Gustav Šourek, and Filip Železný. Forty years of score-based soccer match outcome prediction: an experimental review. *IMA Journal of Management Mathematics*, 2021.

[12] Kenneth Leung. Building and managing data science pipelines with kedro, 2022. <`https://neptune.ai/blog/data-science-pipelines-with-kedro`>, accessed on 19. 5. 2022.

[13] M. J. Maher. Modelling association football scores. *Statistica Neerlandica*, 1982.

[14] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, New York, 1997.

[15] 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, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2011.

[16] Ploomber. Ploomber framework, 2022.

[17] Vijay Janapa Reddi, Greg Diamos, Pete Warden, Peter Mattson, and David Kanter. Data engineering for everyone. *CoRR*, 2021.

[18] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach.* Pearson, 3 edition, 2009.

[19] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 1959.

[20] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, 2014.

[21] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[22] Tong Yu and Hong Zhu. Hyper-parameter optimization: A review of algorithms and applications. *ArXiv*, 2020.

Appendix A

List of Acronyms

CSV Comma-separated values file format

DevOps Development operations

HPC High performance cluster

JSON JavaScript Object Notation file format

MLOps Machine learning operations

PSQL PostgreSQL database

SDK Software development kit

SQL Structured Query Language

YAML YAML Ain't Markup Language

Appendix B

Project structure

```
/
├── blocks
├── dataloaders
│   ├── filters
│   └── sports
│       ├── match
│       └── race
├── datawrappers
│   ├── groups
│   └── sports
│       ├── match
│       └── race
├── learners
│   ├── testers
│   └── trainers
├── models
│   ├── neural
│   │   └── torch
│   ├── others
│   └── statisticals
├── utils
├── transformers
│   ├── dataset
│   │   └── utils
│   ├── features
│   │   └── ratings
│   └── labels
├── workflows
│       └── advanced
├── enviroment.lock.yml
└── README.md
```