University of Pristina "Hasan Prishtina"

Topic: Application of the Random Forest classification algorithm in predicting the outcome of the video game Dota 2

Mentors: Prof. Dr. Eng. Lule Ahmedi

Student: Labinot Vila

Pristina, December 2019

Abstract

IntelliDota is a project implemented using the Scala and Flutter programming languages, the combination of which brings a smart, stable, fast and helpful application, through which we can analyze data sources, visualize and apply different metrics and algorithms on them. This project is divided into two parts, this document is about the IntelliDota Classification part.

IntelliDota Classification represents the first part of the IntelliDota project in which a self-created data set is used in which the Random Forest algorithm is applied. By the words self-created data set we mean that the data set is not taken ready-made, but is created with the help of the Steam Public API whose format is Json, filtered based on certain columns and prepared for the algorithm.

After creating the algorithm, we save the trained model so that it can be reused whenever we want to perform a new row classification. This is done via an isolated Docker container.

Since this application contains two parts, we have created a *backend* application in which we provide a programming interface for the services of the IntelliDota Clustering part. We have created these services using the Scala programming language, specifically the Play framework.

Scala is known as a powerful programming language, while Play is a framework that provides the MVC pattern for web programming. Along with Play, the Spark framework is also used, which is the main way we implement and solve our problems. Spark offers data set manipulation, 'distributed' programming, and a large number of machine learning algorithms. Scala is written in the Spark programming language, so it combines wonderfully.

In order to have uninterrupted access to the endpoints, the project has been converted to a container and hosted in the Cloud via Google Cloud Platform. The conversion to container is done via Docker. The connection to the API is done via https://serverfinal3-qm4ka2ucaq-ew.a.run.app.

ACKNOWLEDGMENTS

First, I would like to thank my family throughout these years for their support and motivation, appreciating their dedication and difficulties, and then Prof. Dr. Lule Ahmedi for the contribution, seriousness, and encouragement she has offered to me and every student who has had the honor of being taught by her throughout these years.

Also, a special thank you goes to Prime LLC, specifically to mentor Endrit Bytyqi for the willingness and skills that he has never hesitated to share.

List of tables

List of figures

Table of contents

1 entry

Just as human curiosity never dies, so does the advancement of technology. Although it is not wrong to say that technology has almost reached its peak, there are still unexplored paths that we can say represent a technological world in itself. Among them, and among the most popular and sought after by people is artificial *intelligence* .

1.1 Motivation – Why artificial intelligence?

Artificial intelligence, also referred to as machine *intelligence* It is about stimulating the intelligence acquired and developed by machines [1].

This science is otherwise defined as the field of study of intelligent agents. (eng. *intelligent agents*) *:* a device that understands its environment and takes actions that maximize the likelihood of achieving its goal.

We say that an agent is rational if it does the right thing, that is, every action in certain circumstances is correct. But how can we define what is correct and what is not? If the agent goes through a sequence of actions and states that satisfy us, we say that the agent has had a good performance [2].

So, machines are trying to adopt human understanding skills. and problem solving (eng. *learning and problem solving*) *.*

The goal of an intelligent agent can be simple, such as playing a game of GO, or complex, such as performing mathematical operations.

The basic principle of artificial intelligence is the use of algorithms. Algorithms are a set of instructions that a computer can execute. A complex algorithm is built as a set of simple algorithms (Fig 1).

```
1. Nëse kemi një 'sulm', luaj në atë hapësirë, përndryshe
2. Nëse kemi një 'nyje' nga e cila mund të krijohen dy 'sulme',
       luaj aty, përndryshe
3. Luaj në qendër nëse është bosh, përndryshe
4. Luaj në qoshen tjetër nëse kundërshtari ka luajtur në një
       qoshe, përndryshe
5. Luaj në një qoshe boshe, përndryshe
6. Luaj në çfarëdo hapësire boshe.
```

Fig 1: Algorithm of a tick-tack-toe game

Some algorithms are capable of learning from a set of data, as in our case, where the algorithm learns a strategy or a good way. (*rule of thumb*) which it applies to new data, and some other algorithms can write other algorithms themselves.

Some of *the* learning algorithms, such as *nearest - neighbor* , *decision trees ,* or *Bayesian* networks can theoretically approximate any data to a given function (if they have infinite memory and time). Artificial intelligence has been stimulated since ancient times, and among the most popular approaches are:

- Symbolism, also known as formal logic or implication: if the person has a cold, then they have the flu.

- Interference Bayesian (*Bayesian interference*) *:* if a person has a cold, then there is a probability that they also have the flu.

- *Support* Vector Machine or *Nearest-neighbor:* after reviewing the data of people who have a cold, including age, symptoms, and other factors, and these factors belong to the current patient, we say that the patient has the flu.

A machine is said to be intelligent if it passes the Turing test, a test that was originally designed to provide a satisfactory definition of intelligence. The test is passed if it is impossible to tell whether the answers come from a human or a machine after a series of human questions [3].

1.2 Problem description

It is known that today we find applications of artificial intelligence in almost every area of life, especially in the field of computing, starting from recommender systems, prediction systems, and many others.

Considering these applications, we have attempted to build an assistant for the popular video game Dota 2 as it would not only help people during the game, but would also enrich the community with a unique data set (since we are creating one ourselves through the relevant

calls) and source code that presents the progress of the project and an example of how to achieve something similar both in terms of artificial intelligence, and in terms of making calls and filtering the Json structure.

It should be noted that there are many datasets related to this topic on the internet, but due to the techniques and methodologies we want to implement, we have chosen to create one ourselves.

problem can be described in several stages, the most important of which are:

- Data set creation – by successfully completing this step, we are given the opportunity to apply metrics, comparisons, and even artificial intelligence algorithms to the relevant data source. The main issue is how to fulfill the requirements and filter them, in order to obtain a healthy data set.

- Applying the algorithm – we create a trained model, whether local or not, with which we can apply the classification in real time. Locally, this is easily done as there is no need for additional measures to save the model, while if we want to publish on a host server , we must save the model as an image.

- Choosing a programming language – to achieve the above requirements, we need a programming language that offers multiple power and methods for data processing and preprocessing, as well as options for artificial intelligence.

- Creating endpoints – gives calls made from outside a specific address where to refer to get the desired results.

So, we say that our application has successfully completed its work if it is able to classify and cluster the data we give it as input in real time, quickly and in an organized manner.

The intelligent part of the application can be implemented in many ways, starting from the Python programming language with the relevant libraries that enable the creation of an application similar to what we want to do, to writing an algorithm on our own. But since it was not considered reasonable to 'reinvent the wheel', we have decided to use the Scala programming language together with Spark, which offers a powerful arsenal for manipulating data and artificial intelligence. A more detailed description of why we chose this language with the relevant libraries will be explained later.

As for the data creation part, there are other ways (such as using a programming language to create the calls) but not methodologies, which means that the data set we have created can only be obtained through the official Steam website through the relevant calls and filters.

2 Artificial Intelligence – Classification as Supervised Learning

In general, there are different forms of machine learning, but three more specifically represent almost every form of learning. We have:

- Unsupervised learning – the agent learns ways to solve or compute a given problem based on input even though there is no feedback from any given source (i.e., the algorithm is neither rewarded nor punished for the chosen action). The most common and widespread technique is clustering, which makes potentially accurate groupings from input data. For example, a taxi gradually develops the concept of a 'good traffic day' or 'bad traffic day' without ever being given any previous examples from a 'teacher'.

- Reinforcement learning – the agent learns from a series of output actions, whether they are punishing or rewarding actions. For example, a tip at the end of a taxi ride tells the algorithm that it has acted well, otherwise it has acted badly. It is up to the algorithm to decide which parts of itself have made it perform the action well.

- Supervised learning – the agent learns from input-output pairs, and learns a function through which it creates a scheme that maps future input values to output values.

This application will use supervised learning, a technique that will be described below with the corresponding algorithm!

The purpose of supervised learning is described as follows (Fig 2) where x and y can be any values and the function h is the hypothesis.

```
Dhënë një bashkësi çiftesh me N elemente trajnuese:
(x1, y1), (x2, y2), ..., (xN, yN) ku secila dalje
yJ (j = 1, 2, ... N) është gjeneruar nga një funksion
i panjohur y = f(x), atëherë gjej një funksion h që
përafron funksionin f.
```

Fig 2: Supervised learning expressed mathematically

To measure the accuracy of the hypothesis, we provide a test set of data different from the training data. We say that the hypothesis has generalized well if it has accurately predicted the value *y* for the new type of data [4]. When the output *y* is an interval of finite values, we say that the learning problem is classification and is called binary or Boolean if the interval has only two values. Whereas when *y* is a continuous number we say that the learning problem is regression. Technically, in regression we try to find a close value, because the probability of correctly guessing the value is 0.

2.1 Supervised learning algorithms

There are many supervised learning algorithms, however, although they work on the same principle, they all have advantages, disadvantages, and cases where one algorithm is more suitable than another. There is no algorithm that works best on all possible problems. Among the main parameters that make one algorithm have an advantage over another algorithm are:

- Variability in data – whether the input data has high variability (discrete, continuous, continuously increasing data).

- Repetition in the data – if the input data has large repetitions, this results in a high correlation matrix which means that some algorithms such as Linear Regression or Logistic Regression will perform poorly.

- The presence of independence and non-linearity – if each attribute of the input data has an independent impact on the result, then linear algorithms generally perform better than others.

Below we will list some of the most common and suitable algorithms regardless of the data set.

2.1.1 Decision trees

Decision Trees *are* used to classify an instance by traversing it from the top of the tree to a leaf node, which returns the result of the classified instance. The goal is to create a model that predicts the value of a variable depending on other input values.

So, a tree is built by dividing the set of input attributes into certain groups, this process, called recursive partitioning. (eng. *recursive splitting*) , is repeated recursively on each subset obtained. This process ends when the obtained attribute is unique, which means that further splittings do not affect the final value [5].

Despite its shortcomings and advantages, based on the massive use and documentation of this algorithm, we can easily say that it is among the most powerful algorithms currently in use.

An example of how Decision Trees work is described below (Fig 3).

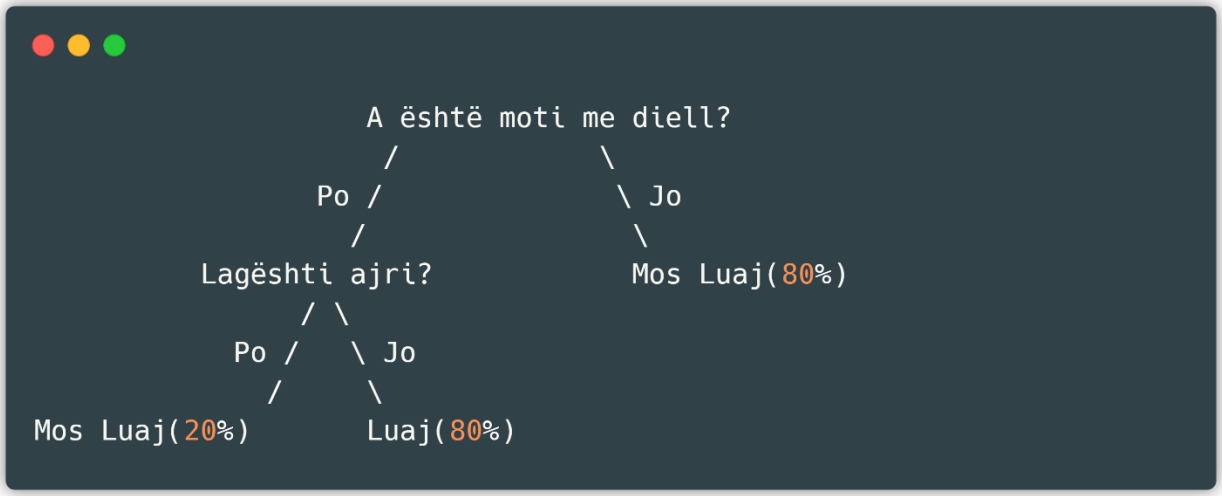


Fig 3: Example of Decision Trees

Advantages and disadvantages of *the Decision Trees approach*

+ They are capable of generating understandable rules.

+ They are capable of performing classification without large calculations.

+ Perform both classification and regression.

- They are weak when it comes to predicting the value of a continuous attribute.

- If we have little training data, then the margin of error is large.

- To train data, you need a lot of computing power.

2.1.2 Naive Bayes

It belongs to the family of probabilistic classifiers. (eng. *probabilistic classification*) where Bayes' theorem is used as a basis (Fig 4) under the assumption that no attribute is dependent on the other, hence the name *naïve* [6].

```
P(A/B) = P(B/A)P(A) / P(B) ku:

P(A/B) - probabiliteti i ngjarjes A kur ka ndodhur ngjarja B
P(B/A) - probabiliteti i ngjarjes B kur ka ndodhur ngjarja A
P(B) - probabiliteti i ngjarjes B
```

Fig 4: Bayes' Theorem

While this is best explained with an example, if cancer is age-dependent, then Bayesian theorem can be used to weight the age-dependent nature of cancer in predicting whether a given person has cancer or not. This method is only applicable to discrete data.

2.1.3 Random Forest

It is an ensemble algorithm , which means that it combines one or more algorithms of the same or different type to classify the object, so it is the same as if the algorithm went through *SVM, Naïve Bayes* or *Decision Tree* and finally voted on which algorithm to take as the basis. This type of algorithm creates a set of trees randomly selected from the training data, collects the votes from different sets of trees voted as decision-makers and assigns the final class of the object [7]. A summary of how Random Forest works is described below (Fig 5).

```
Supozojmë se të dhënat trajnuese janë: [X1, X2, X3, X4] me atribute
korrosponduese [L1, L2, L3, L4]. Atëhere Random Forest mund të krijojë
tri pemë që marrin si vlera trajnuese nënbashkësitë e të dhënave trajnuese, pra

[X1, X2, X3], [X1, X2, X4], [X2, X3, X4].

Në fund predikimi bazohet në votat më të shumta prej pemëve përkatëse.
```

Fig 5: Summary of Random Forest

Such algorithms, called ensemble algorithms, are much more accurate than other types of algorithms because groups of trees protect each other from individual errors, unless all groups of trees fail in the same way. For example, when some groups of trees are wrong, most of the others are right, so in this way the trees are able to all move in the right direction together.

2.1.4 Support Vector Machine

Support Vector Machines in machine learning are supervised models that analyze data used for classification and regression. Given a set of training data, each belonging to a category, *the SVM* can build a model that associates future examples with the corresponding categories, making this choice in a non-probabilistic manner.

Specifically, an *SVM* is a representation of points in space, organized so that each point in space is separated by a space that tends to be as large as possible. Then, the new predicted examples follow the same pattern and the category they belong to depends on the space they fall into. As an example, we have the section below where the green points represent the first group G1 while the yellow points represent the second group G2. Any new data that, as stated in the definition above, belongs to one of these groups, belongs to the group whose attributes are most similar (Fig 6).

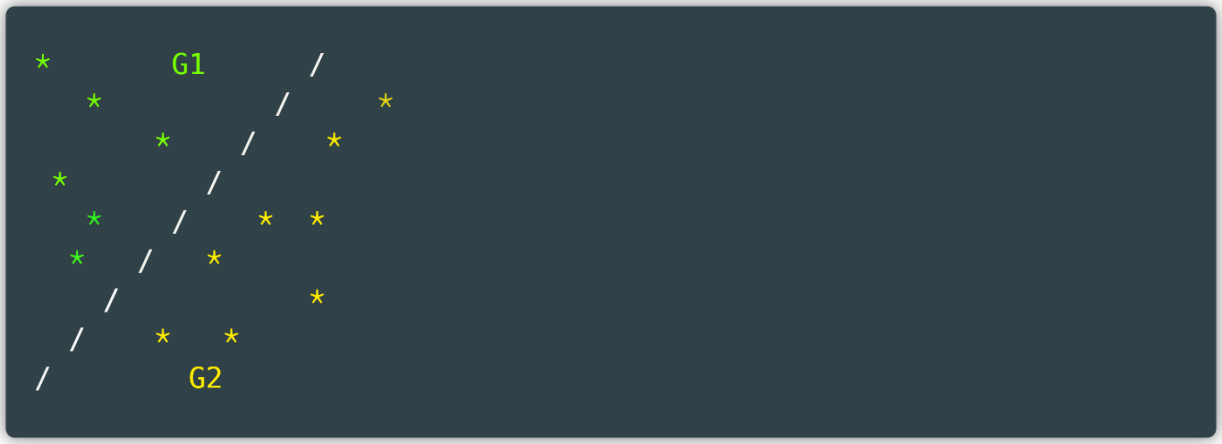


Fig 6: Dividing line in SVM

2.1.5 Algorithm chosen for application

Considering most of the algorithms, their operation and specific application cases, we have chosen the Random Forest Classification algorithm. The reasons why we have chosen these algorithms are given below:

- By setting the depth of the tree, we are not subject to overfitting *because* the model is not trained so much that it is too dynamic, nor is it too dynamic to have low accuracy (if the data set has a significant number of rows).

- Our data set consists of numerical values (as will be seen below) and the algorithm in question is accurate in prediction as it is judged by a number of decision trees and not just one strain.

- Of all the tests of other algorithms on our data set, this algorithm has proven to be significantly more accurate (by about 5% difference).

So this algorithm will be used in the following.

3 Introduction to the project

Having described artificial intelligence and the main classification algorithms as representatives of supervised machine learning, we can now move on to describing the solution provided to the problem, as well as the results obtained.

We are starting with a description of the software tools that were used, so that their use can be understood later, depending on the case.

- *IntelliJ* – build tools (eng. *Integrated Development Kit* , (IDEA) in which the application is written, built (eng. *build*) AND TESTED (eng. *test*). This tool is written in the Java programming language and was chosen among many others for its high organization, the many possibilities it offers, and the ready-made packages for both Scala and Play.

- *Git* – the platform on which the work progress is coordinated. Since this platform dominates the work market, it has been seen as reasonable to practice this way of working even more. It works through the command line interface (*Command Line Interface, CLI*).

- *Github* – the platform where *Git* puts source files. This platform generally collects all public projects written by every programmer in the world and offers them to researchers so that they can access both source code and applications that are built as utilities, such as the tool used below *Ngrok.*

- *Postman* – the tool through which HTTP requests are made requirements (eng. *requests*) such as *GET* and *POST* with the corresponding attributes. Thanks to this tool, we were able to stimulate requests locally.

- *Ngrok* – the tool through which the possible accesses of the endpoints are published. This was needed because there was no need to copy the project to each working environment but the services in which the project is running locally are exposed to a public URI through which the second part of the project is implemented. For example, the endpoint, which is accessed via the link localhost:9000/getColumns?kind=steam is exposed at the link http://338ab558.ngrok.io/getColumns?kind=steam and this connection can be called from any source.

- *Sublime Text* – a tool used to examine large JSON data sets, from which columns were selected for sorting. Also, a useful tool for checking the validity of JSON structures (via CTRL + ALT + J).

- *Excel* – also a tool for viewing CSV format data , to which various filters and controls can be applied.

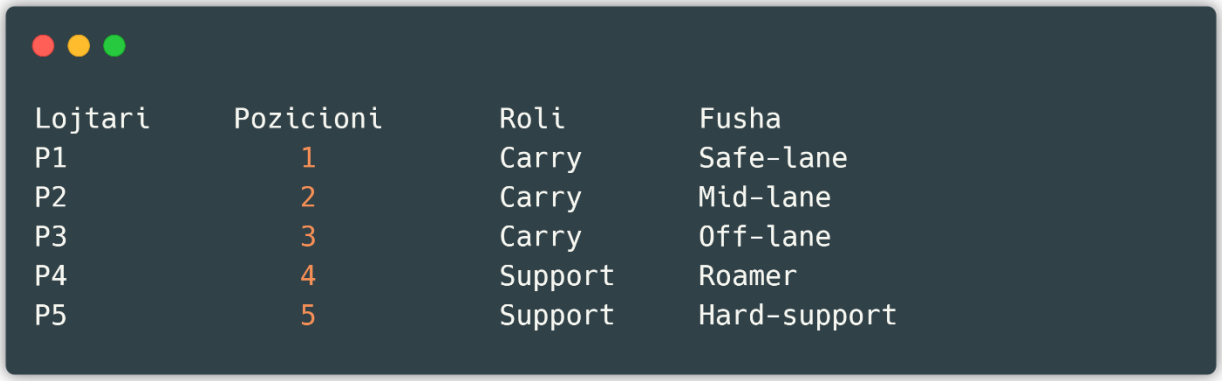
- *Powershell* – the tool used to manage the Docker image. Unlike other CLIs (Command Line Interfaces), this tool is more functional, feature-rich, and has a more user-friendly interface.

- *Docker* – the tool used to create the image so we can publish our services on the internet accessible to everyone.

IntelliDota is an application built in Scala and Flutter that attempts to *cluster and* classify *the* results of a video game , in our case the video game *Dota 2. We will call the part of the application that performs the classification* IntelliDota Classification and the part of the application that performs the clustering IntelliDota Clustering. Before we continue further with the application, we will give a brief description of what this video game is and what we want to preach.

3.1 Computer game Dota 2

Dota 2 stands for Defense of the Ancients , where two teams of five players attempt to destroy the enemy's main base. A team of five players has a higher-level division, namely carries *and* supports . Essentially, each *support* supports a *carry* (Fig 7) .



Lojtari	Pozicioni	Roli	Fusha
P1	1	Carry	Safe-lane
P2	2	Carry	Mid-lane
P3	3	Carry	Off-lane
P4	4	Support	Roamer
P5	5	Support	Hard-support

Fig 7: First look at the Dota 2 video game lineup

Let's mention some of the characteristics of each of these roles. Position 1 is known for kills, assists, high levels of money, experience and activity in the game, but not always. Position 2 is known for tower pushing, killing and creating space for position 1. Position 3 is known for its ability to stay first in line, does not cause high damage, but the entire enemy attack falls on it. And finally there are positions 4 and 5, where both are supports but position 5 is known for healing, while position 4 is more of a light support. Each player, regardless of the role, has hundreds of statistics, but among the most important and important are *leaver_status, gold_per_min, leaver_status, xp_per_min, deaths, tower_damage* , etc. The full list can be found in the following chapter in which the postPredict, where the function is described correctly and a description is given of whether the attribute is derived or not.

From a high perspective, the team is organized in such a formation (Fig 8):

Fig 8: Team organization

So, if this data is collected, analyzed, and metrics are applied to it, we can get different results either about the game, or even the players who play this game.

With different algorithms, we can clearly see what is being played the most, what is being liked and how these attributes along with many others affect the outcome of the game. Among all these aspects, we have decided to preach the attribute of the victory of the radiants (eng. *radiant_win)* , which is nothing more than one of the above columns that identify a specific game, with the only difference being that this column (as well as several others) does not belong to a specific player, but stands as an attribute that identifies the game as a whole.

From what has been said, we can observe that in these games that we want to analyze, we have two types of attributes:

- Attributes that belong to players, for example the attributes *gold_per_min* , *xp_per_min* etc. or

- Attributes that belong to the game, such as the duration of the game, whose attribute is named *duration* , the *radiant_win attribute* that we currently want to predict, *the teamfights attribute* , and many others.

By combining these two attributes, we can create a data source with sound data on which we can apply intelligent algorithms. We have implemented one with the help of the Steam Public Application Programming Interface. (eng. *Steam Public Application Programming Interface, known as Steam Public API)* .

Valve (the company that developed this website) provides these *APIs* so that developers can use the data in interesting and useful ways. That is, developers are allowed to execute requests and perform various queries. At the moment, the current list is limited to just a few video games, but this list is expected to expand quickly [8].

4 Data collection for classification and pre-processing

4.1 First view of the Json structure from which we want to retrieve data

As a data source link for classification, the official Valve website, i.e. Steam via the Steam Public API *(https://steamapi.xpaw.me*) , was used, as mentioned above. [9]Before accessing this page, you must have a Steam account *(https://steampowered.com*) in order to obtain a *Steam Key and* a *domain* name . (eng. *domain name*) from which we are allowed to retrieve data from the API PUbLIC offered by them [10]. A pair of them looks like this, and is set up so that the user using it has a certain limit on *fetching* . of the data (expanded below, Fig 9).

Fig 9: Key-domain pair

Upon receiving these credentials we can make a simple request via the link below (Fig 10).

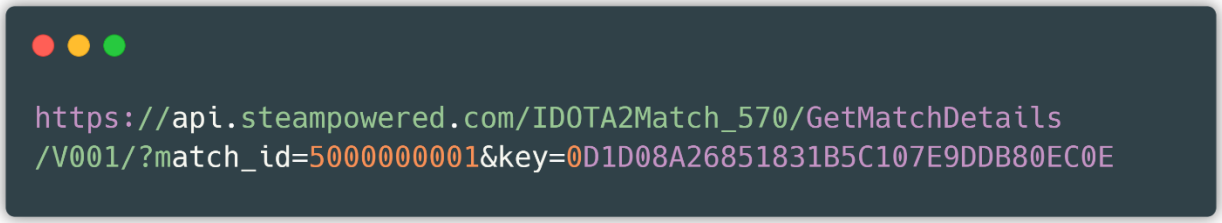


Fig 10: Link to a specific game

where as match id the numerical value of a game with an identification number is given corresponding, which in the above case is 5000000001, and key is the key generated above.

The data format represented by the above link is Json (Javascript Object Notation) and this allows us to store and transport data [11]. A valid match format is found from the number 1 to the number 6 billion, so a match id would be valid from the value match id 1 up to match id 6000000000.

It is known that to collect so much data, it takes years and years, considering that a video game takes about 45 minutes. And for such a long period of time, even the data structure, which in our case is Json, changes significantly, as it has always been attempted to create a structure that is more descriptive and understandable than the previous one. What we want to say is that the Json structure with match id 1 is not the same as the Json structure with match id 5 billion . For this reason, we have chosen an interval in which the Json structure does not change and

possesses enough data that we can create a proper data set. The interval we have chosen is (Fig 11):

```
val START_ID    = 5000000000L
val END_ID      = 5999999999L
val FEEDS       = 50000
```

Fig 11: Selected interval for data

Where to start id represents the start of the interval we want to get the data from and end id represents the end of the interval, during this interval, due to automation we have decided that the value of the data number be dynamic, so in our case FEEDS, which is 50,000.

Depending on the number of FEEDS, our data set will have that much data. We will refer to it as the word data set. our data source which we are creating ourselves.

4.2 Pre-processing of collected data

While retrieving the data, we encountered game ids that were inappropriate or corrupt in the sense that they damaged the prediction algorithm or did not show real data. Such attributes were set by Steam itself. when certain circumstances occur in the game (Fig 12).

```
Mos e proceso strukturën Json në rrethana të tilla:
1. Struktura përmban një atribut të quajtur 'error'.
2. Struktura nuk përmban atributin 'radiant_win',
    atribut ky që duam të predikojmë.
3. Atributi 'duration' është 0, do të thotë që loja nuk
    ka filluar fare.
```

Fig 12: Conditions for bypassing the video game Dota 2

By excluding these cases, we have ensured that each Json structure is valid and has descriptive data, so that the algorithm is as accurate as possible.

This was the only case in which there was a need for external intervention, otherwise this data would contain no null value , a non-descriptive value. or any other type that would damage our data set in any way.

It should also be noted that all data in this data set are of continuous form, for this reason it was more than necessary to apply the removal of outliers , *while* the attributes *leaver_status* and *level* are not affected by this procedure, since respectively one attribute is binary while the other is discreet .

Since there is a limit of about 11,000 consecutive calls in the above link that is intended to prevent abuse and various attacks, we have divided the request sending process into 5 parts, where each part contains 10,000 consecutive calls resulting in almost a quarter of the final data set. And then we have merged these 5 parts to create a complete data set, which we are now operating on (Fig 13Fig 12).

```
DS1         DS2         DS3         DS4         DS5
10.000  +   10.000  +   10.000  +   10.000  +   10.000

= DS (50.000 rreshta)
```

Fig 13: Data set fragments with corresponding rows

This complete data set contains 13 columns and 50,000 rows. How this procedure was carried out will be shown later after we enter the part of implementing the concept through the respective programming language.

4.3 Generic metrics for the derived dataset

To gain more insight into the dataset, the connection, and the data in general, we apply some of the metrics we created. It should be noted that all of these metrics are being performed on a sample. with 1000 rows because the processing on the data set full would take more time.

We start with the final getSamplefrom which we see a sample of our data set (Fig 14), from which we see that we have continuous numerical values in most of the attributes, except for *leaver_status* and *level* which can be considered discrete values, one specifically binary.

Since we only had prior knowledge about this game and how it works, the selection was not the most difficult part, where in a JSON structure with about 65 attributes, we selected attributes contained in this JSON file and attributes that we extracted ourselves, and it turned out to be a successful operation.

```
[{
    "gold_per_min": 6355,   "level": 150,
    "leaver_status": 0,     "xp_per_min": 9056,
    "radiant_score": 63,    "gold_spent": 241485,
    "deaths": 61,           "denies": 40,
    "hero_damage": 267225,  "tower_damage": 10333,
    "last_hits": 974,       "hero_healing": 2316,
    "duration": 2991,       "radiant_win": 1
},
{   "gold_per_min": 3300,   "level": 149,
    "leaver_status": 0,     "xp_per_min": 3810,
    "radiant_score": 46,    "gold_spent": 151995,
    "deaths": 48,           "denies": 49,
    "hero_damage": 219657,  "tower_damage": 25377,
    "last_hits": 1791,      "hero_healing": 375,
    "duration": 3306,       "radiant_win": 1
}{...}]
```


Fig 14: Sample data set

We see each column and its corresponding type through the getSchemaand see that all columns are of type number (Fig 15). So, we are allowed numeric metrics such as comparisons, numeric filtering, and others. Let's not forget that in the *Match class* we have associated the number type with each attribute.


```
[{  "column": "gold_per_min",   "type": "IntegerType"   },
 {  "column": "level",          "type": "IntegerType"   },
 {  "column": "leaver_status",  "type": "IntegerType"   },
 {  "column": "xp_per_min",     "type": "IntegerType"   },
 {  "column": "radiant_score",  "type": "IntegerType"   },
 {  "column": "gold_spent",     "type": "IntegerType"   },
 {  "column": "deaths",         "type": "IntegerType"   },
 {  "column": "denies",         "type": "IntegerType"   },
 {  "column": "hero_damage",    "type": "IntegerType"   },
 {  "column": "tower_damage",   "type": "IntegerType"   },
 {  "column": "last_hits",      "type": "IntegerType"   },
 {  "column": "hero_healing",   "type": "IntegerType"   },
 {  "column": "duration",       "type": "IntegerType"   },
 {  "column": "radiant_win",    "type": "IntegerType"   }]
```


Fig 15: Data set columns with corresponding types

If we apply grouping and counting to *leaver_status* , via getGroupAndCount(parameter implied to be set *to leaver_status*) we see that the split is not a true split, so about 80% of the data is in *leaver_status* 0 and about 20% are in *leaver_status* 1 (Fig 16).


```
[{  "leaver_status": 1, "count": 162    },
 {  "leaver_status": 0, "count": 838    }]
```


Fig 16: Population of leaver_status in data set

If we apply the next metric, which is an internal call in which we want to check how the *leaver_status relationship stands* with the column we want to predicate, so Tab 8: getSchema endpoint

getDoubleGroup, we get (Fig 17):


```
[   { "leaver_status": 1,   "radiant_win": 0,   "count": 100    },
    { "leaver_status": 1,   "radiant_win": 1,   "count": 62     },
    { "leaver_status": 0,   "radiant_win": 0,   "count": 368    },
    { "leaver_status": 0,   "radiant_win": 1,   "count": 470    }]
```


Fig 17: Report leaver_status with radiant_win in the data set

So, when *leaver_status* is not zero it means that the game is not finished as a team and therefore white people have lost almost 65% of their games.

However, that the column selection was good can be verified through the correlation matrix, getCorrelationMatrix, a matrix which represents the relationship of the columns to each other.

So we have both correlations and inverse correlations, where a full report of this matrix is attached to *Github* . Let's now look at the other attributes, just to get a preview of what data we're dealing with.

- Function: group by column *xp_per_min* into 3 partitions

Result: we obtain a relatively good division where 828 elements belong to the interval 0 ~ 472, 113 elements belong to the interval 472 ~ 4755, while about 60 elements belong to the interval 4755 onwards (Fig 18).


```
[{ "bucket": "1.0", "count": "828", "border": "472.0"   },
 { "bucket": "2.0", "count": "113", "border": "4755.0"  },
 { "bucket": "3.0", "count": "59",  "border": "9038.0"  }]
```


Fig 18: getGroupAndCount in 3 divisions according to xp_per_min

- Function: grouping by *level column* into 4 divisions
  The result: once again proves that our data set is healthy, because it is easily understood by ordinary players of this game that the dominant part of the games ends with all players at maximum *level* , which is confirmed by this diagram (Fig 19).


```
[{ "bucket": "1.0", "count": "17",  "border": "4.0"     },
 { "bucket": "2.0", "count": "26",  "border": "42.0"    },
 { "bucket": "3.0", "count": "235", "border": "80.0"    },
 { "bucket": "4.0", "count": "722", "border": "118.0"   }]
```


Fig 19: getGroupAndCount in 4 divisions by level

- Function: group by column *hero_healing* into 3 partitions
  Result: a level 0 division is automatically created, because there is a large population. Otherwise, this division lets us understand that there are two main groups, those who heal and those who do not heal (Fig 20).


```
[{  "bucket": "1.0",    "count": "973", "border": "0.0"     },
 {  "bucket": "2.0",    "count": "22",  "border": "20812.0" },
 {  "bucket": "3.0",    "count": "4",   "border": "41624.0" },
 {  "bucket": "4.0",    "count": "1",   "border": "62436.0" }]
```


Fig 20: getGroupAndCount in 3 divisions by hero_healing

- Function: group by column *denies* into 4 partitions

Result: a division into 5 groups is created, because level 0 is also considered. It can be seen that the main interval lies between the numbers 0 – 100. While a significant part is found in the first three groups. Also, these data can be said to be discrete (Fig 21).


```
[{  "bucket": "1.0",    "count": "215", "border": "0.0"     },
 {  "bucket": "2.0",    "count": "488", "border": "31.0"    },
 {  "bucket": "3.0",    "count": "268", "border": "62.0"    },
 {  "bucket": "4.0",    "count": "28",  "border": "93.0"    },
 {  "bucket": "5.0",    "count": "1",   "border": "124.0"   }]
```


Fig 21: getGroupAndCount in 4 divisions by denial

The groupings performed above are not good practice since the number of categorizations is very small and sound data cannot be extracted, but it was performed in this way for presentation (space) reasons; otherwise, if we want to extract analytical data about the relevant columns, it is much more reasonable to do so in tens.

So, in total we have selected 14 attributes out of several dozen (Fig 22).

```
Atributet e zgjedhura nga struktura Json:
- radiant_score, duration dhe radiant_win.

Atributet e derivuara nga struktura Json:
- gold_per_min, level, leaver_status, xp_per_min,
    gold_spent, deaths, denies, hero_damage,
    tower_damage, last_hits dhe hero_healing.
```

Fig 22: Attributes selected from the structure

Structure attributes are those attributes that did not need further processing before being included in the data set, unlike extracted structure attributes, which we have composed of small pieces.

It should be noted that composite attributes are still attributes of the Json structure, but not ones that we can access directly. Below we describe what we mean by direct access.

The structure of a Json schema is shown below (Fig 23). It can be seen that the main attribute is *result* , which is considered the parent for every other attribute. The children of this attribute

can be categorized into two groups: attributes with sub-attributes such as *players* which consists of 10 objects, each object with properties relevant to a specific player.

```
{       "result": {
        "players": [], //10 lojtarë
        "radiant_win": true,
        "duration": 1469,
        "start_time": 1567328090}}
```

Fig 23: Structure of a Json object

We have aggregated these objects with the corresponding keys and created data for our new data set. Unlike this type of attributes, we also have attributes whose values can be taken directly, without further aggregation or processing. Operating with these attributes is much easier than with attributes that need to be derived or processed, but this number is in the ratio of 2/8 to the attributes that need to be processed, which is a small number.

We have performed the maneuvers and manipulations with the current data set because Scala with its libraries offers very good opportunities to pass data from the Json schema to our Match class, an adaptation that is accomplished with minimal and concise code, so in a way such a way of working has been imposed on us. Let's see how we have realized and organized the data and code in the current project in Scala.

4.4 Creating a Dataset – The Scala Way

Before we continue with the explanation of the project, we need to briefly expand on the Gson concept.

Gson [12]is a Java programming language library that, among many other functions, allows us to convert Java objects into the corresponding JSON structure representation. Since Java runs on the Java Virtual Machine *(JVM)* , it means that Scala also can use this same library [13].

In our case, we used Gson to pass the data from the JSON structure to a Scala *case class.* A *case class* allows us to define the model or structure of our dataset, where a row of the dataset is a new object of this class. Gson was enabled for us after we added the dependency to the project composition as *gson* via the *com.google.code.gson link* , whose version is 2.8.5.

To better explain the above statement, let's use an example. Let's say we have a JSON schema with a single attribute called attribute whose value is 10 and we create a *case class* named Class with an attribute called attribute and use the *fromJSON method* then we get a new object of the class Class with the attribute variable having the value 10. This is a very good and useful practice to create classes automatically, and a long series of these classes results in a data set.

Our model, called Match (Fig 24) consists of 14 attributes, and each of these attributes must also appear in the JSON structure obtained through the links mentioned above (via the Steam API). Each attribute belongs to the number type (Integer , *Int)* .

```
case class Match
(
    gold_per_min: Int,      level: Int,
    leaver_status: Int,     xp_per_min: Int,
    radiant_score: Int,     gold_spent: Int,
    deaths: Int,            denies: Int,
    hero_damage: Int,       tower_damage: Int,
    last_hits: Int,         hero_healing: Int,
    duration: Int,          radiant_win: Int
)
```

Fig 24: Match class model

As mentioned above, this class only serves to create a new object for each record brought through the connections made to the main class at certain intervals. The created object of the Match class is then stored in a sequence. The sequence represents a Scala collection that, by importing a certain function of the Spark library, is allowed to be converted to a data set and vice versa. Since the data set is immutable, we need to create the complete data set in a sequence and then convert it. During this procedure, we track the amount of sequence objects so that if it reaches a certain number, it stops, as the size of the desired data set has been reached. This number in our code is the constant *FEEDS* . We get the Spark functions from *the spark-core* and *spark-sql dependencies* from *org.apache.spark* , whose version is 2.4.4 in this project.

This way of describing is high-level, while below (Fig 25) describes exactly the flow of the program, the variables used and the main logic of creating a valid record. We start by creating a correct connection in order to perform the request. This connection is created through three strings: *steam api, gameId* and *steam key* , where *steam api* and *steam key* are constants while *gameId* varies depending on the following interval, which starts with *start id* and ends with *end id* .

```
var matches = Seq[Match]() // sekuencë e klasës Matches

for (gameId <- START_ID to END_ID) {
    val game = Fetcher.fetchGames
            (steam_api + gameId + steam_key, args)

    if (game != null) {
        matches = matches :+ game
        foundGames += 1

        println(foundGames+". Analyzing game ["+gameId+"]")
    }

    if (foundGames == FEEDS) break
}
```

Fig 25: Main application ring

Args represents a string to which we have given values through *program* [14]*arguments* . of IntelliJ , the software tool with which the application was processed), and this string contains all the obtained values which are gold_per_min, level, leaver_status, xp_per_min, gold_spent, deaths, denies, hero_damage, tower_damage, last_hits and hero_healing (as above)

In addition to this information, the object with the corresponding method *Fetcher.fetchGames is also visible* (Fig 26), static *method* This helps us to collect data. In this method, the request is *defined* . which provides access to the Steam Public API, checks whether the request is valid, and if so , *returns* a new instance of the Match class with all the attributes specified above retrieved from the Json structure in which the request was made. The ability for requests was made possible for us after adding *requests* from *com.lihaoyi* , the latest version of which is 0.1.8.

This object is then added to the sequence called *matches* and the process is repeated until a sequence of 500,000 data items is formulated.

```
def fetchGames(api: String, args: Array[String]): Match = {
    val responseAsJSON = requests.get(api)
    if (responseAsJSON.statusCode != 200) return null

    val preparedGames = Derivator.prepareGame
      (responseAsJSON.get("players").getAsJsonArray, args)

    gson.fromJson(responseAsJSON, classOf[Match])
}
```

Fig 26: Structure of the fetchGames method

After this process, all that remains is to convert it to a *Spark Dataframe* and save the file in a cleaned *-version* locally (Fig 27).

```
val gamesDF = matches.toDF

gamesDF.write.format("csv").option("header", true)
    .mode("overwrite")
    .save(System.getenv("fetched_steam_data"))
```

Fig 27: Converting and saving the data set to the local machine

Where a new folder is created in the *path* corresponding, which is also defined through *IntelliJ* - environment variables . This definition facilitates teamwork, where the working team, instead of changing the values of static variables in the local environment, only changes the environment variables.

data set on our local machine , on which we can apply various metrics and even smart algorithms. A sample of data retrieval and analysis looks like this (Fig 28).

```
8. Analyzing game [5000000018]
9. Analyzing game [5000000019]
10. Analyzing game [5000000020]
```

Fig 28: Data acquisition view

4.5 Sorting algorithm in Scala

With the data set created, we are now able to make a prediction on the *radiant_win attribute* .

As a first step, normally the data set is imported via the Spark library (Fig 29).

```
var dataframe = spark.read
    .option("header", true)
    .option("inferSchema", true)
    .csv(System.getenv("fetched_steam_data"))
```

Fig 29: Reading the created data set

Then you need to rename the attribute *radiant_win* to *label* . *since the way the algorithm works is that it must have a single* attribute as input, called *features* while the attribute that is predicated should be called *a label* .

In addition to this quality, a data set with continuous data (eng. *continual*) it is always reasonable to use a *scaler* [15] along with a method for removing outliers , so that the algorithm is not dominated by a single attribute.

As a new feature added to Spark are pipelines . *But* what exactly does this feature offer us? Instead of applying these three steps separately, we can put them in one stage and apply the algorithm to the main stage rather than a specific step (Fig 30.

```
val assembler = new VectorAssembler()
    .setInputCols(args).setOutputCol("non-scaled")
val scaler = new StandardScaler()
    .setInputCol("non-scaled").setOutputCol("features")
    .setWithStd(true).setWithMean(true)
val algorithm = new RandomForestClassifier()
    .setLabelCol("label").setFeaturesCol("features")
    .setNumTrees(10)
val pipeline = new Pipeline()
    .setStages(Array(assembler, scaler, algorithm))
```

Fig 30: Order of phases according to implementation

We see that we have three main functions:

- *vecAssembler_id* – stands for *VectorAssembler* and serves as a string of columns to aggregate into a single attribute, called in our case without scaling (*non-scaled).* So, as input columns (*inputColumns)* it accepts the columns we want to train and as output column (*outputColumn)* it is the newly created *non-scaled column* .

- *stdScal* – stands for *standardScaler* and is used to return all numbers relatively in a certain range, in our case between 0 and 1. So, each column has only transformed values in the range 0 – 1. It contains two static methods which are *withMean* and *withStd* . The first prepares the data by taking the average value before scaling it, while the second sets the standard deviation. However, the column weights do not change despite these mutations.

- *rfc* – stands for *Random Forest Classifier* (described above), provides the relevant methods by which we can train and apply metrics to new data.

So, if we do a brief summary about the steps of a data set which passes to the prediction; the attributes that serve in the prediction are merged and renamed as *non-scaled,* this attribute is then scaled and renamed to *features,* and finally the *Random Forest Classifier algorithm is applied* , which yields a data set which contains one more column than the previous ones called *prediction,* whose value is what we want.

Phases not only simplify the procedure, but also make the code more concise, readable, and manageable. A visualization of the above looks like this (Fig 31).

```
args                                => Array("non-scaled")
Array("non-scaled") =>    scale     => Array("features")
Array("features")   =>    classify  => Array("label")
```

Fig 31: Simple visualization of phases

The accuracy of our algorithm with a 3/10 data split currently reaches 87%, as seen below (Fig 32):

```
Random forest classifier accuracy: 87.29641693811075.
```

Fig 32: Accuracy of the Random Forest Classifier algorithm

As an evaluation metric, we used the Spark library's *accuracy metric,* which is implemented through the *MulticlassClassificationEvaluator class* , which uses two main methods that serve to obtain the current predictions and compare them with the real ones. It returns the probability of accuracy, i.e. a number from 0 to 1.

We then store the locally trained model in this way (Fig 33).

```
val model = pipeline.fit(train)
model.write.overwrite.save
    (System.getenv("classified_model"))
```

Fig 33: Saving the trained model to the local machine

The big advantage of this action is that there is no need to train the model for each data we want to predict, but the algorithm loads the trained algorithm and applies the specified action to it, which is performed in a few milliseconds.

Otherwise, if we didn't store the trained algorithm, we would have to train it for each prediction, which for our dataset takes approximately 5-6 minutes. This would result in a terrible experience for anyone trying to apply metrics to it.

4.6 Removal of the isolated

As we mentioned above, due to continuous values, the possibility of *outliers appearing* is high, so it is best to remove data with these properties before applying the *Random Forest algorithm.*

For this part we have used the interquartile range method . This method is applied to each column of our data set, and the method works in such a way that first the data are arranged in ascending order and the medians of the first half and the second half are taken. Let's say that Q1 is half of the first median while Q3 is half of the second median. We find the difference between these two values, IQR = Q3 – Q1, this value is called the interquartile range (from which it gets its name) and we add or subtract the values Q3 and Q1, respectively, so, A = Q3 + 1.5 * IQR and B = Q1 – 1.5 * IQR where 1.5 is an increasing or decreasing value of the interval we want to get. We remove each value in the data set smaller than B and larger than A (Fig 34).

```
5   9   2   1   3   7   6   5   8   // vargu i dhënë
1   2   3   5   5   6   7   8   9   // rradhitja rritëse
                |                   // mediana
   (2 + 3) / 2     (7 + 8) / 2      // Q1 dhe Q3
IQR = Q3 - Q1 = 7.5 - 2.5 = 5
A = Q1 - 1.5 * IQR = 2.5 - 1.5 * 5 = -5
B = Q3 + 1.5 * IQR = 7.5 + 1.5 * 5 = 15
```

Fig 34: Example of finding an interquartile range

5 classification and storage as a trained model

5.1 Application structure

As we mentioned above, the architecture in which Play is built is MVC, i.e. *Model, View* and *Controller.* In our case, due to the integration of the application with Flutter, we have disconnected the *View component* , while the *Model component* is not used since we do not need a connection to the database, since we are storing the data locally. Something like the *model component* can be considered the *Match class* mentioned above, which represents a structure in which JSON information is passed .

So, we currently use a controller and *a* package called *utilities* through which we execute multiple methods (Fig 35).

```
controllers ->
    MainController
utilities   ->
    Dataset
    Pre
    Statistics
```

Fig 35: Project structure in Scala

MainController represents the main controller, which routes each request to the correct path. So, each method with *routing* The relevant one is managed by this controller.

Pre (Fig 36) also represents an object on which global methods for Spark are executed. *and for uploading steam* and Kaggle datasets .

```
object Pre {
  def spark(appName: String, master: String) = {}

  def dataframe(spark: SparkSession, path: String) = {}
}
```

Fig 36: Presentation of the Pre object

The Dataset represents a Scala object that looks like this (Fig 37) in which all the methods with the parameters of the corresponding controller *MainController are organized.* Each controller path therefore executes one of the above methods.

```
object Dataset {
  def getStages(path: String)
  def getColumns(dataframe: DataFrame)
  def getSample(dataframe: DataFrame, percentage: Double)
  def getCorrelationMatrix(dataframe: DataFrame)
  def getPredictedModel(path: String)
  def getStats(dataframe: DataFrame)
  def getRawStats(spark: SparkSession, path: String)

  def predict(spark: SparkSession,
              dataframe: DataFrame, s: Seq[Int])
}
```

Fig 37: Dataset object view

And finally *Statistics* contains the methods of the *MainController controller,* specifically *getGroupByAndCount* and *getBinary* methods . Since the complexity of this method is greater, it was necessary to create a new class for functionality.

5.2 Real-time classification implementation

Recall that during our program, we have preserved the classified model. What does this mean?

It is known that the data set must be trained, and then tested with test data, in our case, we had an accuracy of about 87% with training data. But for a normal application, the training data only serves as *testing* . to show whether our algorithm is worth testing with real data or not.

Our algorithm has proven very successful with training data from the same source, but let's try testing it with a string data of a completely different nature.

In our application we also provide a final call called getStages, a call that helps us remember the parameters and functions we have used in our algorithm, in other words, it enables the extension of the planned stages depending on the algorithm (Fig 38).

```
{ "vecAssembler_4d7620a503d8": [
      { "name": "inputCols",    "value": "[Ljava.lang]" },
      { "name": "outputCol",    "value": "non-scaled"   }],
  "stdScal_3eb70c027ef3": [
      { "name": "inputCol",     "value": "non-scaled"   },
      { "name": "outputCol",    "value": "features"     },
      { "name": "withMean",     "value": "true"         },
      { "name": "withStd",      "value": "true"         }],
  "rfc_ba39ad829e83": [
      { "name": "featuresCol",  "value": "features"     },
      { "name": "labelCol",     "value": "label"        },
      { "name": "numTrees",     "value": "10"           }]}
```


Fig 38: Phases represented as JSON objects – data set 1

where you can see the names of the classes (for example *vecAssembler_id*) and the methods used with the corresponding values (for example *outputCol* with the value *non-scaled*).

And if we compare it with the original function, everything makes sense, since the above names completely correspond to the methods written above. As an example, we are illustrating another case, where the K-Means algorithm with certain parameters was used (Fig 39).


```
{   "vecAssembler_025011705475": [
        {   "name": "inputCols",    "value": "[Ljava.lang]" },
        {   "name": "outputCol",    "value": "features"     }],
    "kmeans_dda0042efb74": [
        {   "name": "k",            "value": "6"            },
        {   "name": "maxIter",      "value": "25"           }]}
```


Fig 39: Phases represented as JSON objects - data set 2

Where based on this case, we can say that the *vecAssembler class* and the *kmeans class* with their static methods with their respective methods (for example *name* with the value 6) were used. This is done as follows (Fig 40).

```
val assembler = new VectorAssembler()
    .setInputCols(elements)
    .setOutputCol("features")
val kmeans = new KMeans()
    .setK(6)
    .setMaxIter(25)
```

Fig 40: K-Means in Scala

We move on to the preaching part where we need to create a request for our application, the response of which will show us the result in JSON format (Fig 41).

```
localhost:9000/postPredict?
gold_per_min=2585&  level=139&  leaver_status=0&
xp_per_min=3658&    radiant_score=34&   gold_spent=89220&
deaths=65&  denies=21&  hero_damage=170629& tower_damage=2732&
last_hits=901&  hero_healing=10700& duration=2549
```

Fig 41: POST sermon form

It can be seen that the attributes gold_per_min, level, leaver_status, xp_per_min, radiant_score, gold_spent, deaths, denies, hero_damage, tower_damage, last_hits, hero_healing and duration have been assigned the corresponding numerical values. It is known that the number of attributes in the connection is the same as the number of attributes accepted by the endpoint. We execute the call and we get the corresponding response (Fig 42).

```
"probability": {
    "type": 1,
    "values": [
        0.9838030841661979,
        0.016196915833802155
    ]
},
"prediction": 0.0
```

Fig 42: Result of POST request for sermon

key is returned . with a type *that* represents the type of algorithm, which in our case is classification, a list of values representing the probability, and the prediction itself.

So, the algorithm classified the input values from us as 0 with a probability of 0.983 which corresponds to 98%, a very high accuracy.

What happens behind the scenes when the call is made via the link above is that a Spark RDD is created – which represents another form of more primitive collection than data sets (Fig 43).

```
val RDD = spark.sparkContext.makeRDD(list)
```

Fig 43: Creating RDD from list of input attributes

and then simulate the creation of a data set with the input columns that are provided through the input part *(frontend)* from the RDD (Fig 44)

```
val df = spark.createDataFrame(RDD, StructType(columns))
```

Fig 44: Creating a data set from an RDD

Now, *df* represents our data set with one row. With this data set created, we can apply it to the previously trained model (Fig 45).

```
getPredictedModel(sys.props.get("classified_model").get)
            .transform(df)
```

Fig 45: Trained model and real-time classification

Where *getPredictedModel* returns the trained model in the specified path called *classified_model* and through the *transform method* , real-time classification is performed and the response is returned in JSON format.

6 Preparing the image for displaying services

6.1 What is Docker?

Before we continue, we need to describe what Docker is and how it works. Docker is a tool that helps in creating, developing and publishing web applications. It does this by packaging our software in packages called containers. Containers are isolated from each other, which means that they have complete independence. The motivation for Docker, among many other reasons, came from the fact that to execute a program from two different systems it is much easier to create a Docker image that can be executed from the other side than to copy the source code. The other and main reason is that to publish a system on the web, which in this case was the publication of APIs created by the backend, Docker is the most suitable tool, especially when the application is distributed in several parts, as in our case, the data sets part, the trained models for classification and clustering and others.

Below we describe the main steps we took to create an image like the one mentioned above.

6.2 Container creation and browsing

The first step we need to take is to install Docker from the official website. Whenever we create a Docker image, there must be a text file called Dockerfile in the main project directory. In our case, we have created this file in the project's root *directory* .

We need to configure the Dockerfile so that its execution will enable the creation of our image. The content of our Dockerfile is as follows (Fig 46):

```
1  FROM aa8y/sbt:0.13.18
2
3  USER root
4
5  RUN apk update && apk add --no-cache libc6-compat
6
7  COPY ./target/universal/scalatry-1.0.zip .
8
9  RUN unzip scalatry-1.0.zip
10
11 COPY ./main_route .
12
13 CMD scalatry-1.0/bin/scalatry -Dhttp.port=${PORT}
```

Fig 46: First view of Dockerfile

where line number 1 represents the main image and the environment in which the application will be executed. Since our project is built in Scala and has SBT as the base for building (eng.

build), then we use the same image with the same version as the project, i.e. 0.13.18. Next, we assign the user as *root* (step 3) in order to have administrative access. It should be noted that this image is not an official image from Docker but created by a specific person or persons, which in our case is the developer *aa8y* . So, we are creating an image from another image that we are using as a development environment in our image.

We update our environment and add *libc6-compat* which enables libraries to have no conflicts and ensures that the application does not have errors during execution (step 5).

We copy the path ./target/universal/scalatry-1.0.zip to the base directory of our image that we have not yet created (step 7). The above path is provided by *the sbt shell* via the *dist command* which creates a compressed application ready for publication, i.e. with all the libraries and dependencies ready and packaged in it.

After copying the project folder to the image's root directory, we decompress it (step 9). This step must be performed in order to have the folder open for execution.

Finally, we copy our local *main_route folder* and place it in the base directory of the Docker image (step 11). Recall that *main_route* represents the folder in which all the datasets needed for the application are located along with the trained models, so the path ./main_route/fetched_steam_data represents the dataset created as above.

Step 13 represents an abstract action, which means that after executing the steps above, you should visit the path scalatry-1.0/bin/ and execute the scalatry BIN file. The suffix -Dhttp.port represents an action that forces the application to run on port ${PORT}. This port comes from *GCP (Google Cloud Platform*), since as a rule of GCP it is that the port on which the web page will be published is dynamic, and not assigned by the application developers (in our case us).

A summary of all the steps above is described below (Fig 47)

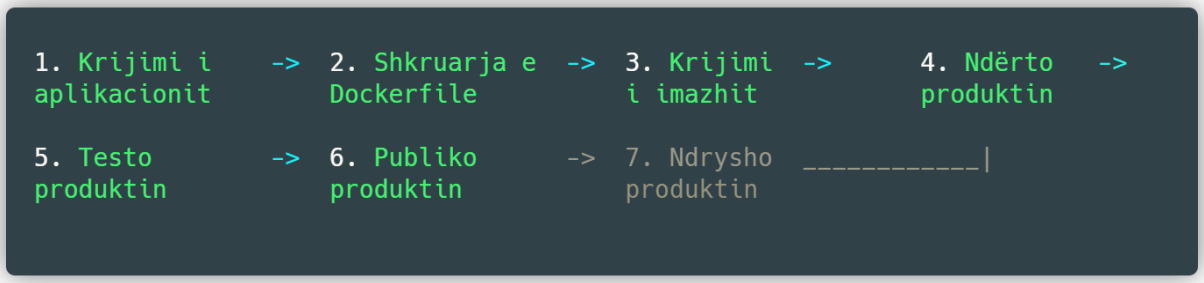


Fig 47: Summary of image creation steps

With these steps completed, our image is ready to be created. To proceed further, we open a *CLI* (*Command Line Interface*) in order to build the image and use the commands (Fig 48)

```
cd Documents/Github/ScalaTry/
docker build -t intellidota .
```

Fig 48: Commands for building Docker image

The first command allows you to change to the directory where the project is located, while the second command builds the image. The *-t prefix* stands for tag *and* allows the image created in this case to be recognized as intellidota, while the dot indicates that the path to use to create the image is where we are currently. After executing that command, we get the following results (Fig 49):

```
PS C:\Users\Labinot\Documents\Github\intellidota> docker build -t intellidota .
      Sending build context to Docker daemon  235.3MB
Step 1/7 : FROM aa8y/sbt:0.13.18
 ---> 5760094a84cb
Step 2/7 : USER root
 ---> Using cache
 ---> 66fc68e29875
Step 3/7 : RUN apk update && apk add --no-cache libc6-compat
 ---> Using cache
 ---> a89b58f78e6e
Step 4/7 : COPY ./target/universal/scalatry-1.0.zip .
 ---> Using cache
 ---> d9b718abb4c1
Step 5/7 : RUN unzip scalatry-1.0.zip
 ---> Using cache
 ---> bd79cc99cad8
Step 6/7 : COPY ./main_route .
 ---> Using cache
 ---> fba5375c8e10
Step 7/7 : CMD scalatry-1.0/bin/scalatry -Dhttp.port=${PORT}
 ---> Using cache
 ---> a3a823eff3c5
Successfully built a3a823eff3c5
Successfully tagged intellidota:latest
SECURITY WARNING: You are building a Docker image from Windows against a non-Windows Docker host. All files and
directories added to build context will have '-rwxr-xr-x' permissions. It is recommended to double check and
reset permissions for sensitive files and directories.
PS C:\Users\Labinot\Documents\Github\intellidota>
```

Fig 49: First look at building the image in Docker

And if we want to display the images created so far, we use the *docker images command* . After creating the image, we run it using the command: *docker run -it intellidota* where the prefix *-it* stands for interface *and* we get the results as follows (Fig 50):

```
PS C:\Users\Labinot\Documents\Github\intellidota> docker run -it intellidota
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/src/app/scalatry-1.0/lib/ch.qos.logback.logback-classic-
1.2.3.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/src/app/scalatry-1.0/lib/org.slf4j.slf4j-log4j12-
1.7.16.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [ch.qos.logback.classic.util.ContextSelectorStaticBinder]
[warn] o.a.h.u.NativeCodeLoader - Unable to load native-hadoop library for your platform... using builtin-java
classes where applicable
[info] play.api.Play - Application started (Prod) (no global state)
[info] p.c.s.AkkaHttpServer - Listening for HTTP on /0.0.0.0:1234
```

Fig 50: Successful image build

So, the server is set up at 0.0.0.0 with port 1234, so visiting *localhost:1234* takes us to our application. With this result, even publishing to any major Cloud platform will not present us with any problem. Finally, let's look at the contents of our container.

We start with the command *docker container ps --all* . This allows us to list all containers along with their respective identification numbers and the result looks like this (Fig 51):

```
PS C:\Users\Labinot> docker container ps --all
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS               NAMES
c26ebae69ab5        intellidota         "/bin/sh -c 'scalatr…"   26 seconds ago      Up 25 seconds
                     vigorous_shockley
```

Fig 51: Container listing

Which means that our container has the ID number c26ebae69ab5 and is named intellidota. Once we have this information, we run the command *docker exec -it c26ebae69ab5 bash* to jump into the container's contents. The word *exec* stands for *executor* .

We are now inside our container. Let's start with the PWD (Point Working Directory) command to see the path we are currently in (Fig 52):

```
bash-4.4# pwd
/usr/src/app
```

Fig 52: The directory where we are working in the container

Let's see what's inside the current directory as follows (Fig 53):

```
bash-4.4# ls
classified_model    fetched_steam_data  logs
scalatry-1.0.zip    clustered_model     kaggle_data         scalatry-1.0
```

Fig 53: Container contents

So, we have the trained models, the compressed and decompressed project (scalatry-1.0.zip and scalatry-1.0 respectively). If we go back to the base directory, i.e. root, we see that we have (Fig 54):

```
bash-4.4# cd ~
bash-4.4# ls
bin      lib      libexec  local    sbin     share    src
```

Fig 54: Container base directory

Based on this result, we see that a Unix operating system is used within the aa8y/sbt:0.13.18 image, since we have not used such an image directly.

This is another reason why Docker is currently one of the most widely used tools, as it offers flexibility and is easy to use and manage.

7 Conclusion

The project is divided into two parts, the *frontend* and the *backend* , which is where the most difficulties came from, as the visualization and description of the main functions had to be done in full coordination. However, the experience of creating such a platform in which the *frontend* and *backend are coordinated* has been special and will come into great use in the future, as nowadays it is rare to work with *a frontend* integrated into the corresponding *framework* .

Although similar projects exist in the industry, none are like this. Among the most prominent projects that we have been inspired by are Opendota and Dotabuff, which integrate graphics with real-time statistics and provide a nice interface in which improvisation can be easily achieved, however, neither integrates machine learning into them, be it clustering or classification, or any other type of learning.

The first part of the project, called IntelliDota Classification, starts with the creation of our data set from the Steam Public API. By creating the data set, we intend to send GET requests more than 50,000 times until we get the necessary Json documents, so that our data set has an initial look, although during these GET requests, some exceptions are made. So, we only get data sets that are descriptive and do not contain unexpected values. A plus during this procedure is that we do not deal with empty values since we only filter them when retrieving the data set. So, we do not need any specific pre-processing.

The Random Forest algorithm is then developed on the created data set, in which an accuracy of about 87% was obtained. We used Random Forest with a depth of 10 units in order to avoid *overfitting* . We saved the trained model using a Docker image so that it can be reused later, whenever necessary, without the need for retraining.

Then we continued with the creation of web services through the Scala programming language, specifically the Play framework. Play represents an MVC (model, view, controller) model and is a powerful asset in the development of web pages. As web services, we have created several, the main ones being the ability to classify and cluster in real time, not forgetting the possibilities for graphical analysis or visualization.

We have combined our entire project into a Docker image. We have done this so that the project can be executed anywhere, regardless of the environment in which it is located. Images are another new way of hosting that has been used a lot recently, and Docker is the main tool to create images. The image is built when a Dockerfile is created within the project structure that is used as an instruction set for the Docker tool. Our Docker image includes the Steam and Kaggle datasets along with the training models for each. Our image is isolated, which is where the main advantage comes from.

Meanwhile, our project cannot currently be expanded further but can be used as a basic application for interesting statistics. However, we will continue to work with machine learning and extract deeper analyses both for video games and for concrete cases. In video games, a further step can be taken, such as extracting analyses of what were the main reasons that a team won. The level of machine learning that needs to be developed for this application is not

more advanced than the level we are currently using, but the analytical capabilities need to be increased, since everything is again extracted from graphs.

Other important elements for both the project and the future have been the use of the Json structure, i.e. manipulating attributes via Scala, the use of Gson as an intermediary library between Json and Scala, and retrieving data via Scala dependencies, such as *lihaoyi ,* through which we have implemented the requests. It is worth mentioning once again the *ngrok application* through which we were able to publish our endpoints so that the second part, the *frontend , can be implemented* .

All parts of this application are public, starting from the created dataset, to Scala, the connection between them with the REST model, and the classification and clustering algorithm so that developers can be helped in any way they can.

8 Annexes

The source code and data set are listed below:

- Backend: https://github.com/LabinotVila/IntelliDota
- Trained models: https://github.com/noraibrahimi/IntelliDotaIC
- Frontend: https://github.com/noraibrahimi/IntelliDota-mobile
- Steam dataset: https://www.kaggle.com/labinotvila/dota-2-steam-api-fetched-dataset

9 Endpoint list

9.1 index

host / index

Parameters:	
Description:	Returns the content of the start page, which is the list of endpoints provided by us.

Tab 1: Endpoint index

9.2 getColumns

host / getColumns

Parameters:	kind	– dataset type [steam / Kaggle]
Description:	returns a list of columns of the corresponding dataset	
Example:	/getColumns? kind = steam	

Tab 2: getColumns endpoint

9.3 getSample

host / getSample

Parameters:	kind	– dataset type [steam / Kaggle]
	percentage	– percentage of the monster [0 to 100]
Description:	returns a sample of the corresponding dataset, depending on the percentage	
Example:	/getSample? kind =steam& percentage =10	

Tab 3: getSample endpoint

9.4 getStages

host / getStages		
Parameters:	kind	– dataset type [steam / Kaggle]
Description:	returns an array of the stages the data has gone through	
Example:	/getStages? kind =kaggle	

Tab 4: getStages endpoint

9.5 getCorrelationMatrix

host / getCorrelationMatrix		
Parameters:	kind	– dataset type [steam / Kaggle]
Description:	returns a string of numbers indicating the relationship between columns of the corresponding dataset	
Example:	/getCorrelationMatrix? kind =kaggle	

Tab 5: Endpoint: getCorrelationMatrix

9.6 getGroupAndCount

host / getGroupAndCount		
Parameters:	kind	– dataset type [steam / Kaggle]
	attribute	- grouping occurs according to this attribute
	partitions	– number of partitions of numeric data
Description:	returns a range of groups and the corresponding interval that the data belongs to	
Example:	/getGroupAndCount? attribute =xp_per_min& partitions =3	

Tab 6: Endpoint getGroupAndCount

9.7 getStages

host / getStats		
Parameters:	kind	– dataset type [steam / Kaggle]
Description:	returns the number of rows and columns of the corresponding dataset	
Example:	/ getStats? kind = steam	

Tab 7: getStages endpoint

9.8 getSchema

host / getSchema		
Parameters:	kind	– dataset type [steam / Kaggle]
Description:	returns the columns and corresponding type of the specified dataset	
Example:	/ getSchema? kind =steam	

Tab 8: getSchema endpoint

9.9 getDoubleGroup

host / getDoubleGroup		
Parameters:	kind	– dataset type [steam / Kaggle]
	col1	- first column
	col2	- second column
Description:	returns grouping and counting by corresponding columns	
Example:	/ getDoubleGroup? kind =steam& col1 =leaver_status& col2 =radiant_win	

Tab 9: getDoubleGroup endpoint

9.10 postPredict

host / postPredict			
Parameters:	gold_per_min	– money per minute	- derived
	level	– levels	- derived
	leaver_status	– match end status – derived	
	xp_per_min	– experience per minute	- derived
	radiant_score	- whites' points	- direct
	gold_spent	– money spent	- derived
	deaths	– deaths during the game	- derived
	denials	– denials of murders	- derived
	hero_damage	- damage to opponents	- derived
	tower_damage	- damage to towers	- derived
	last_hits	- final hit for killing	- derived
	hero_healing	- team assist	- derived
	duration	- length of the game	- live
Description: people won	Returns two new attributes in the data set that indicate whether white or not and the exact percentage of the answer given.		
Example:	/postPredict?		
	gold_per_min =2585& level =139& leaver_status =0& xp_per_min =3658		
	& radiant_score =34& gold_spent =89220& deaths =65& denials =21		
	& hero_damage =170629& tower_damage =2732& last_hits =901		
	& hero_healing =10700& duration =2549		

Tab 10: PostPredict endpoint

10 REFERENCE

[1] P. Norvig and SJ Russell, Artificial Intelligence: A Modern Approach, Harlow, United Kingdom: Prentice Hall, 2009.

[2] P. Norvig and SJ Russell, «Artificial Intelligence: A Modern Approach,» in *Artificial Intelligence: A Modern Approach* , Harlow, United Kingdom, Prentice Hall, 2019, p. 37.

[3] P. Norvig and SJ Russell, «Artificial Intelligence: A Modern Approach,» in *Artificial Intelligence: A Modern Approach* , Harlow, United Kingdom, Prentice Hall, 2009, p. 2.

[4] PRSJ Norvig, «Artificial Intelligence: A Modern Approach,» in *Artificial Intelligence: A Modern Approach* , Harlow, United Kingdom, Prentice Hall, 2009, p. 695.

[5] JR Quinlan, «Wikipedia,» University of Sydney, 1975. [Online]. Available: https://en.wikipedia.org/wiki/Decision_tree_learning. [Accessed 6 October 2019].

[6] Bayes, «Wikipedia,» [Online]. Available: https://en.wikipedia.org/wiki/Naive_Bayes_classifier. [Accessed 6 October 2019].

[7] Ho and . L. Breiman, «Wikipedia,» 1995. [Online]. Available: https://en.wikipedia.org/wiki/Random_forest. [Accessed 11 September 2019].

[8] Valve, «Steam,» 8 October 2016. [Online]. Available: https://steamcommunity.com/dev. [Accessed 4 November 2019].

[9] neilatsyracuse, «Reddit,» Reddit, 2018. [Online]. Available: https://www.reddit.com/r/DotA2/comments/7yzlu4/dota2_api/. [Accessed 2 October 2019].

[10] S. Steam Developers, «Steam,» 12 September 2003. [Online]. Available: https://steamcommunity.com/dev. [Accessed 5 October 2019].

[11] «w3school,» w3, 2019. [Online]. Available: https://www.w3schools.com/js/js_JSON_intro.asp. [Accessed 1 October 2019].

[12] «Github,» Google, 22 May 2008. [Online]. Available: https://github.com/google/gson. [Accessed 11 October 2019].

[13] Oracle, «Wikipedia,» OpenJDK, 8 May 2007. [Online]. Available: https://en.wikipedia.org/wiki/Java_virtual_machine. [Accessed 28 September 2019].

[14] "Jetbrains," JetBrains, 1 January 2001. [Online]. Available: https://www.jetbrains.com/help/idea/2016.1/intellij-idea-help.pdf. [Accessed 1 October 2019].

[15] M. Zaharia, “Apache Spark,” 9 September 2019. [Online]. Available: https://spark.apache.org/docs/latest/ml-features. [Accessed 16 October 2019].

[16] P. Bugnion, PR Nicolas and A. Kozlov, Scala: Applied Machine Learning, Packt Publishing, 2017.

[17] Prime, «Prime,» Prime, November 2016. [Online]. Available: https://goprime.io/. [Accessed June 2019].

[18] Databricks, The Data Scientist's Guide to Apache Spark™, Berkley: Apache Spark, 2019.