CS:GO Player Skill Prediction

Master Thesis

August 2019

Logitech

Student:
Baran Nama
Computer Science

Supervisors:
Arash Salarian
Prof. Boi Faltings

There are only two ways to live your life.
One is as though nothing is a miracle.
The other is as though everything is a miracle.
— Albert Einstein

To my mother and my 'everything'...

Abstract

Counter-Strike: Global Offensive (CS:GO) is the latest sequel game of the Counter-Strike brand and one of the most popular multiplayer FPS game among amateur players as well as E-Sport professionals. Regardless of its release date (2012), It is still among top-ranked video games in Steam, and numerous professional tournaments have been organized with an astonishing amount of prize pools. The popularity of the game leads to a business market for different kinds of products such as gaming ones like keyboards, mouses, and headsets. Logitech is one of the biggest gaming product manufacturers that tries to exploit this market and understand the dynamics of the competitive environment of CS:GO. In this master thesis, thanks to cooperation between EPFL and Logitech, the methodology of data science has been applied to CS:GO environment in order to scrutinize the relationship between the various statistics of players recorded during the match (such as kill, death, assists) and the skill level of these players. To do so, firstly a distributed data collection and storage system has been implemented to scrape data from several matchmaking servers and store those data reliably and incrementally. Then, a set of essential and diverse features has been extracted from collected raw data. After intense post-processing of featured dataset, a universal label aggregation on dataset has been accommodated by using TrueSkill algorithm to maintain a supervised learning task. The last step of the thesis consists of the analysis of enriched and featured dataset to get meaningful relationships and patterns, especially between player statistics and approximation of the latent skill of players. In the first experiment, a baseline feature set has been utilized by several learning algorithms such as linear regressor, random tree regressor, and neural network, and the outcome of learning models have been summarized. In the second experiment, we have improved the data pipeline by adding additional features, feature selection phase and feature extraction phase, and changed the experiment setup. In the third short experiment, we have considered the temporal dimension of the data as well. The results were compared and conclusion has been made about results, shortcomings of the project, and feature improvements.

Contents

Chapter 1

Introduction

Multiplayer video games have become increasingly popular as the internet speed evolved. People usually form teams with other players and battle against each other, which enhances the game's popularity by creating a competitive sport-like environment. This popularity of the game can push the competition even further that players may team up for professional competitions with a prize. As a example, one of the most popular and competitive games among them is Counter-Strike: Global Offensive (CS:GO). CS:GO is one of the most popular first-person shooter (FPS) game in history and it has been awarded numerous times including ESPORT Game of the year awards [1].
Its huge community and popularity have, of course, a business value that several companies are trying to capitalize. Logitech is one of the companies trying to understand several dynamics of the game and mentality of CS:GO player in order to derive the best possible business benefits. Our aim in this project was to build a predictive model to estimate skill of players using features extracted by analyzing the data recorded in-game during matches. In order to reach our goal, we first build a data infrastructure on Amazon AWS for collection, processing, and aggregation of match data. The data has been scraped from several matchmaking server and yield in an S3 bucket. For aggregation, TrueSkill algorithm has been used to build a global indicator of player skill level. For extraction of relevant features, we developed an analyzer in Golang to process the collected demo files. After that, the whole pipeline of data analysis task has been started, including data cleaning, imputation, feature extraction and selection, data modeling, and learning from data. We will present all the details of each step in the article.

1.1 Counter-Strike: Global Offensive

In order to establish a full understanding, we can start by giving the necessary background information about CS:GO. Wikipedia states that "Counter-Strike: Global Offensive (CS:GO) is a multiplayer first-person shooter video game developed by Hidden Path Entertainment and Valve Corporation. It is the fourth game in the Counter-Strike series and was released for Microsoft Windows, OS X, Xbox 360, and PlayStation 3 on August

21, 2012, while the Linux version was released in 2014.
The game pits two teams against each other: the Terrorists and the Counter-Terrorists. Both sides are tasked with eliminating the other while also completing separate objectives. The Terrorists, depending on the game mode, must either plant the bomb or defend the hostages, while the Counter-Terrorists must either prevent the bomb from being planted, defuse the bomb, or rescue the hostages. There are nine game modes, all of which have distinct characteristics specific to that mode. The game also has matchmaking support that allows players to play on dedicated Valve servers, as well as allowing members of the community to host their own servers with custom maps and game modes. A battle-royale game-mode, "Danger Zone", was introduced in 2018" [2]. The citation from Wikipedia provides nearly all necessary information about the foundation of CS:GO.
We can now give several statistics about it to back the statement about its game popularity. Figure 1.1 illustrates the player number statistics of CS:GO since it has been released. Provided in the figure, there is a constant trend in player number, which indicates that the game remained popular until now by sustaining its player number. Another clue for the popularity of the game has been provided in figure 1.2. Even if seven years have passed since its first release, it dominates the first place in the ranking of top games by concurrent players.

Figure 1.1: CS:GO player statistics (Taken from: https://steamcharts.com/app/730 on April 2019)

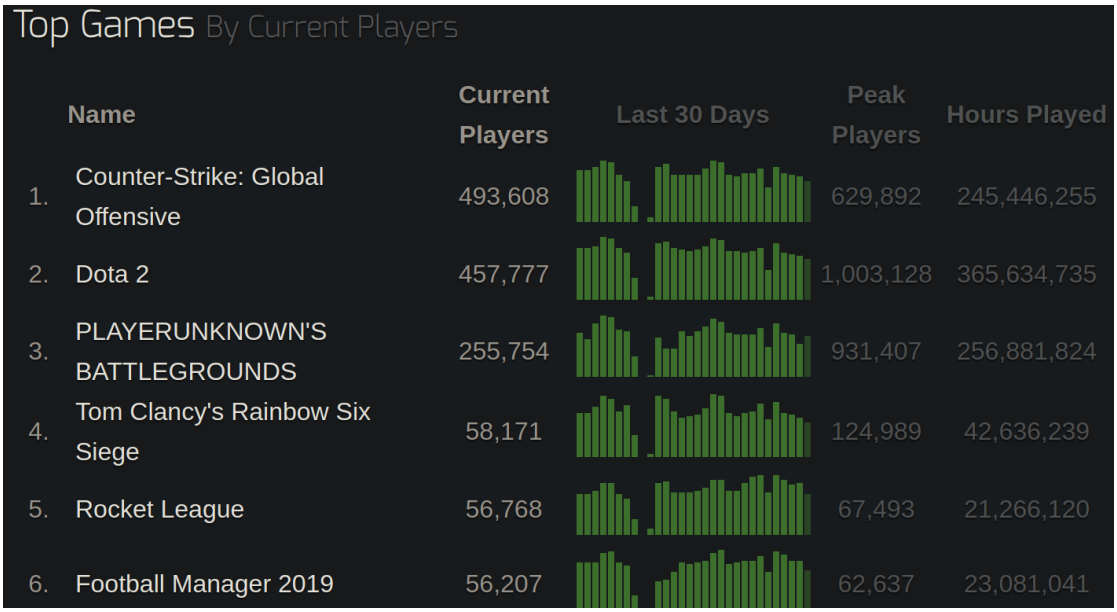


Figure 1.2: Top games by current players (Taken from: https://steamcharts.com/ on April 2019).

This popularity and competition among casual players have got answered by the professional arena of CS:GO as well. According to 99Damage, there are over 2000 professional CS:GO teams seeking to participate in professional tournaments [1]. Moreover, according to E-Sport earning, $74,170,609.45 from 3997 tournaments prize money was awarded to nearly 10000 players between 2012 and 2019 [2].

1.2 General Definitions

In order to construct a better understanding of the project, several definitions need to be provided beforehand.

1.2.1 Demo Files

Demo files are game replays of played matches utilized by Valve's Source game engine. It is a file format to compress a serialization of the sub-sampled data transferred over the network between the server and the players during a match. Usually, they are recorded by a matchmaking server currently the match running on (it can be recorded by a spectator of the match as well). They are usually served as one demo file per map; however, there could be a situation where several demo files per match have recorded if one demo file is not enough to store a long match. The recording of a demo file usually starts before the actual match starts; therefore, it also includes a warm-up period of the match. All information on a demo file has been encoded in a bit-wise manner.

[1] 99Damage is the biggest German website for CS:GO. Link: https://csgo.99damage.de/de/edb/teams
[2] Link: https://www.esportsearnings.com/games

A demo file collects sub-sampled instance of the game on each checkpoint called Tick. A tick is a logical time unit used in a matchmaking server as well as a demo file recording to snapshot a match. On a tick, all client inputs, actions, and interactions with each other and game entities have been sampled and saved to the demo file (game entities are any modifiable objects in the game such as players and weapons). Usual tick rate is 128 ticks per second, which means approximately 7.8ms per tick.
Concatenation of 8 subsequent bits forms a single burst. A burst is another logical structure, and it has two fixed parts consisting of an event list, and a delta change list. Events are basically all occurring actions during a game such as the death of a player, round start or weapon fire. Each event also includes a piece of context information like which player has been killed by whom, which round started or which weapon has been fired by whom. Delta changes are an update transaction of game entity states currently in the game. Essentially, they are describing a list of entities and their properties currently inserted, updated, or removed from the game. Position of a player, ammo count change in the magazine of a weapon or thrown flash bomb could be several examples of delta changes. Because only entity state changes are reported rather than complete status, sequential processing of chained delta changes is a natural need in order to get a complete situation of a game entity on a tick [3, 4].

1.2.2 Matchmaking servers

Matchmaking is a technique of matching online players (as a team or individual) on online play sessions. Therefore, matchmaking servers are mediators seeking to procure the best possible matching among players for conducting an online game session. There are several factors to provide the best available matching among players. Fairness is one of the consideration. Servers usually rank the players and compose team roughly near skill level to conduct a fair match. Less waiting time on lobbies is another consideration by players like low ping values for matchmaking servers. Another consideration is the effective elimination algorithm against cheaters crucial for the reliability and trustfulness of the matchmaking system.
For the case of CS:GO, there are several matchmaking servers other than the official matchmaking system of Valve. These servers claim to offer better player experience by leveraging the relief of issues given above. The most prominent ones are ESEA[3] and FaceIt[4]. These are the matchmaking servers with enhanced lobby and anti-cheating system, improved player support and forum, and better player matching system, which also offer premium service with a monthly subscription. These two servers are the majority of the main data source of our projects. In addition to these servers, HLTV[5] is another website that its database crafted by the community of tournament organizers, players, and fans. It is not a matchmaking server, but a community web page includes all kind of pro and semi-pro tournament matches, player and team statistics, CS:GO community

[3]Link: https://play.esea.net/
[4]Link: https://www.faceit.com/en
[5]Link: https://www.hltv.org/

news, and public forums, which was the last data source for our project. In figure 1.3, an extended comparison of these three servers has been provided.

NAME	PROs	CONs	APR. PLAYER NUM.
Faceit: https://www.faceit.com/en/home	Faceit has a data API which usually meets the expectations and no problem to get data. Good website design, an increasing number of players, and better support. Because it has an API, retrieval of data is easier compared to other servers.	ELO ranking and its change per match are not reported per player easily on data API. Hard to filter important tournaments like the pro or semi-pro using data API. 60 days of demo file availability.	**30k - 50k** player online (CS:GO)
HLTV: https://www.hltv.org/	Comprehensive stats per match on the website. Because it is not a matchmaking server but a publisher of match data, matches have already filtered by only important tournaments involved in pro or semi-pro players. Comprehensive player rating calculation compared to other matchmaking servers. Much more info about pro and semi-pro teams compared to other servers.	No data API and website protected by CloudFlare.Therefore, extensive work needs to bypass protection and scrape web pages. Only focused on CS:GO.	No active players, but appr. **20 -30** matches published per day.
ESEA: https://play.esea.net/	Oldest matchmaking server on CSGO. Several important tournaments (ESEA League, ESEA tournaments) held by ESEA. Several player rank tables which can include wonderkids for CS:GO. ESL Pro league collaboration.	Very strict CloudFlare protection system. Right now, it is unable to be scrapped because all HTML element names are random and page content is dynamic. Only focused on CS:GO. Nearly 30 days demo file availability.	**3K - 5K** online players. Over 5000 players registered on the league.

Figure 1.3: The CS:GO related server comparison with several attributes.

1.2.3 AWS

We are using Amazon Web Server (AWS) to realize data infrastructure. AWS is an on-demand cloud computing facility for everyone including companies and individuals as well, which provides highly scalable and high performance distributed computing building blocks and tools to promote all kinds of the application development process from beginning to end. One of the most prominent cloud service provided by, heavily used in the project as well, is Amazon Elastic Compute Cloud (EC2). EC2 instances are the core computational building blocks on AWS, which consist of virtual computers running on virtual private servers. They are highly scalable and customizable components shaping with computation, speed, and reliability needs.

Another major component of the project is the Amazon Simple Storage Service (S3). S3 is a multi-type object storage bucket serving as a single data warehouse. It aims to sustain scalability, high availability, and low latency with low failure probability and it can administer simple storage for any kinds of applications including internet applications, backup and recovery, disaster recovery, and data archives. Another element used as a database service in the project is the Amazon Relational Database Service (RDS). RDS is highly customizable, scalable, reliable, and distributed relational database implementation that can serve various database engines such as MySQL and PostgreSQL. It offers various significant features including Multi-Availability Zone (AZ) deployment (deployment of synchronous replicas in a different Availability Zone), read replicas (read-only DB replicas circumvent read-heavy database workloads), and performance metrics and monitoring facilities. The last referral is Amazon Elastic Map Reduce(EMR) service. EMR is highly modifiable and scalable map reduce[6] implementation based on Apache Spark and Hadoop frameworks[6] to distribute the data and processing across a resizable cluster of Amazon EC2 instances. Because of distributed handling of data and calculation, they are heavily used in learning tasks and big data processing pipelines [5, 6, 7].

1.2.4 TrueSkill

TrueSkill is a player ranking, and skill assessment algorithm developed on Microsoft labs in 2005 for the use of matchmaking system in online video games on their Xbox Live platform [8]. The algorithm fundamentally tries to estimate the skill level of players in an environment with minimal input features and fast convergence speed to estimate the real (latent) skill of a player (using minimal match record possible) in order to provide a good matching between players. Before diving in the details of TrueSkill, it is crucial to mention older player ranking systems, which TrueSkill is heavily influenced by.
The first legacy ranking system needs to be mentioned in the ELO ranking system. Arpad E. Elo developed the ELO ranking system in 1959, which the algorithm has been adopted by the United States Chess Federation (USCF) in 1960 and World Chess Federation (FIDE) in 1970. It was mainly developed for ranking Chess players, and adopted by other application fields such as video games, association football, American football, basketball, and Major League Baseball [9]. The main working logic is as the following:

1. ELO rating system assumes that every player has a single number representing his / her skill, and assigns an initial skill point to each new player currently registered in a game environment.

2. On each played match, there is a skill point transaction from the loser of the match to the winner of the match. The amount of transferred skill points is determined by a formula and player modeling PDF (Probability Density Function).

[6]Link:https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

3. For calculating the expected outcome of the match, each player skill in a match has been modeled by a Uniform or Logistical PDF with zero mean and a fixed standard variance, which is one of the model parameters. Then the skill difference of these player distribution functions has been taken, and the occurrence probability of this expected outcome has been sampled from selected PDF same with player modeling PDF.

4. Lastly, by considering expected outcome calculated previous level, actual outcome and skill point change per match (K-Factor) (another model parameter can also be though as "Maximum" number of points that a player can gain during a match), the amount of transactional skill point has been calculated and transferred from loser player to winner player.

In the ELO system, the player with higher skill point has been assumed to a better player and expected the outcome of a match favors the better player. When an 'underdog win' occurs (the situation happens when a player with lower skill point won the game), the transactional point between two players is higher comparing other situations like equally skilled players because it is highly probable that the ranking system misestimates skill levels of two players (or more uncertain).
There are several notable limitations in the ELO system. First, there is no measure of certainty for the skill point of a player, which means we do not know how much a player skill can deviate from current expected skill. Second, ELO ranking has several model parameters have to be carefully tuned like the type of PDF in player modeling, K-Factor. Third, K-Factor is a fixed value meaning changes in skill points for a new player is the same for an experienced player. This situation can discourage new players from playing a game and decline the speed of convergence to the real skill level of a newcomer. Fourth, it is not giving very satisfactory results for the multiplayer team-based games because the foundation of the algorithm is mainly suitable for single-player games.
After the ELO, several modifications and improvements have been made to alleviate the issues mentioned above. The Glicko system can be highlighted as one of the most prominent milestones build upon the ELO ranking system. The Glicko system has introduced a new parameter called "rating deviation" (RD) instead of fixed standard deviation used in ELO. By using RD, we can build a confidence interval for player skill, and measure in which degree we are certain about the player skill. A high RD indicates a player is new to the system and several matches are needed to converge latent player skill. When the player begins to play more matches, the RD value is decreasing, which means the ranking system is becoming confident about the estimation of this player skill because of the increasing number of matches. The RD value for a player is increasing as time passes because the system is becoming more uncertain about the estimation of player skill [10].
TrueSkill is a more advanced player ranking system combines the advantageous sides of two previously mentioned ranking systems in a generative probabilistic model utilizing Bayesian inference and factor graphs. Similar to ELO ranking system, TrueSkill adopts

two parameters per player namely mu (μ), which represent player current skill, and sigma (σ), which represents player skill variability like RD parameter in Glitcko. Same as Glitcko, σ is decreasing while a player is playing more matches as the system is becoming more certain about player true skill, and increasing between matches as the system becomes more dubious about the skill. Unlike ELO, there is no fixed K-Factor in TrueSkill, and it is adjusted dynamically depending on the value of σ.
TrueSkill is a mathematically involved algorithm; so, we will try to summarize how TrueSkill works in general, please check the reference for further details [11]. As introduced before, TrueSkill uses factor graphs and message passing algorithm to calculate marginalized posterior skill of the player using a factorized joint PDF of players involved in a match. TrueSkill uses Gaussian distribution with μ value as mean and σ^2 as variance recorded as the result of the previous match as prior observation and tries to get the posterior skill of that player by a using likelihood function (coming from the newly played match). Factor graphs are a type of probabilistic bipartite graph representing a factorization of a probability density function (or joint PDF) and used to model the posterior skill of a player in TrueSkill. It has two types of entities (that is why nominated as a bipartite graph) named variables represented as circles and factors represented as squares. Variable nodes represent marginal values of each variable in a joint probability distribution, and factors represent relations (functions) between these variables, while the edges between factors and variables represent local dependencies. Message passing algorithm (sum-product algorithm) is a tool to calculate any of these marginal values of variables efficiently(without calculating all possible factors between all variables). You can find all details with how the factor graph is utilized for the marginal values of player skills done in the TrueSkill algorithm on the reference[7]
The factor graph in TrueSkill paper is depicted in the figure 1.4. In the first-level nodes, for each player i at time t, prior player skill (s_i) has been modelled by a Gaussian distribution with μ_{t-1} as mean and σ^2_{t-1} as variance ($s_i \sim \mathcal{N}(\mu_{t-1}, \sigma^2_{t-1})$), which has been observed from the result of previous match (posterior skill of a player from a previous match, otherwise predefined default parameters used). After that, in level two nodes, TrueSkill assumes that each player i has contributed the match played at time t proportional to their prior skill s_i, therefore player performance in the match has been modeled as follows:

$$p_i \sim \mathcal{N}(s_i, \beta^2)\,)$$

which β^2 is a model parameter that represents performance variance of a player, and s_i is the center of the player performance distribution. This β parameter can be though as the amount of skill point to guarantee about an 80% chance of winning for one player to another. After modeling player performance for a match, in level three nodes, TrueSkill takes a weighted average of player performance Gaussian distributions to find out each

[7]Link: http://mlg.eng.cam.ac.uk/teaching/4f13/1415/lect0809.pdf

team performance Gaussian:

$$t \sim \sum_i w_i p_i$$

which weights(w_i) are model parameters, and can be adjusted separately by the amount of time a player played in a match. After finding out team performances, in level four nodes, the difference of two team performance Gaussian distributions have been obtained to determine the expected outcome (prior) of the match. This Gaussian distribution has been modified according to real observation (match result) to be compatible with real situation meaning the calculating posterior distribution of team difference according to observation (impossible area (win probability of loser team or loss probability of winner team) of the Gaussian has been truncated). ϵ is the parameter represents draw margin of a match, and team performance difference has been compared to this parameter whether a match has been finalized with a draw or not.
After calculating the truncated posterior distribution of team performance difference, we need to iterate back from down to top to calculate posterior marginal skill of each player. At that point, the message passing algorithm has been used introduced above. However, the problem is that the comparison between a team skill difference and a draw margin of ϵ resulted in a non-gaussian result. Therefore, TrueSkill is approximating previous marginal posterior of the team performance difference with same mean and variance using expectation propagation [12, 13], which is a subclass of approximate message passing algorithms with moment matching [14]. In order to approximate this posterior difference distribution, TrueSkill tries to minimize the Kullback–Leibler divergence [15]. Because the messages in the last step (depicted with number 2 and 5 in the figure 1.4) are approximations of real distributions, the algorithm iterates over all messages the shortest path between any approximated marginal pairs until any of those marginals is not changing anymore. The update rules for each node from bottom to top can be found in original TrueSkill paper with detailed explanation [8]. The plain summary of TrueSkill algorithm can be found in figure 1.5.

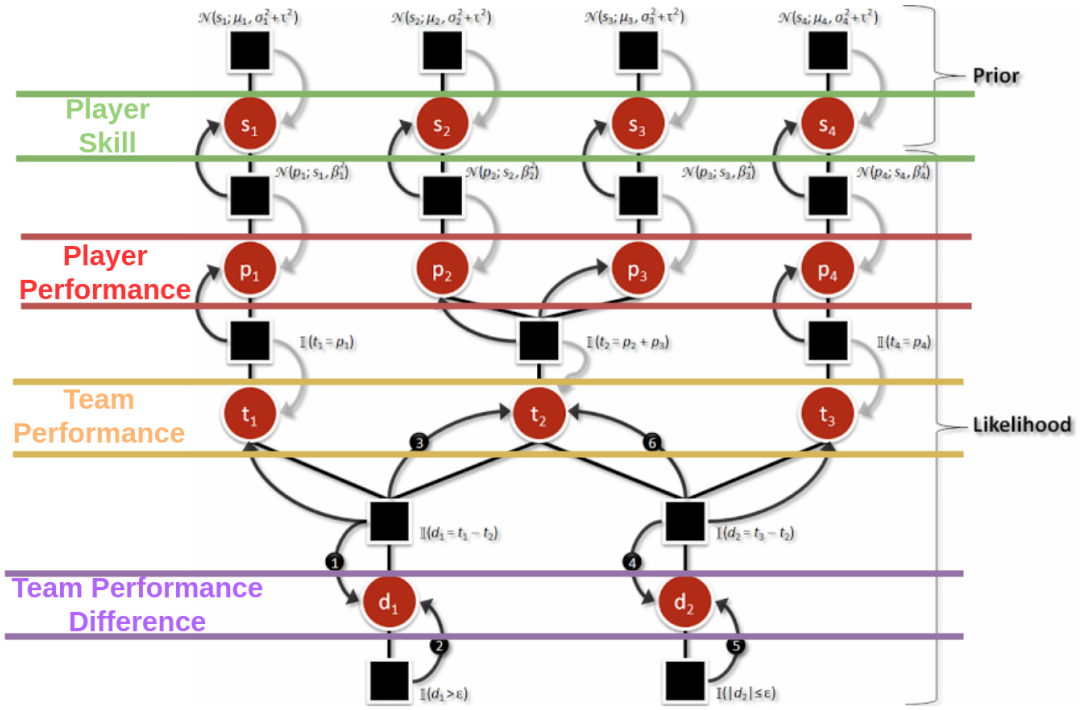


Figure 1.4: Factor graph of TrueSkill.

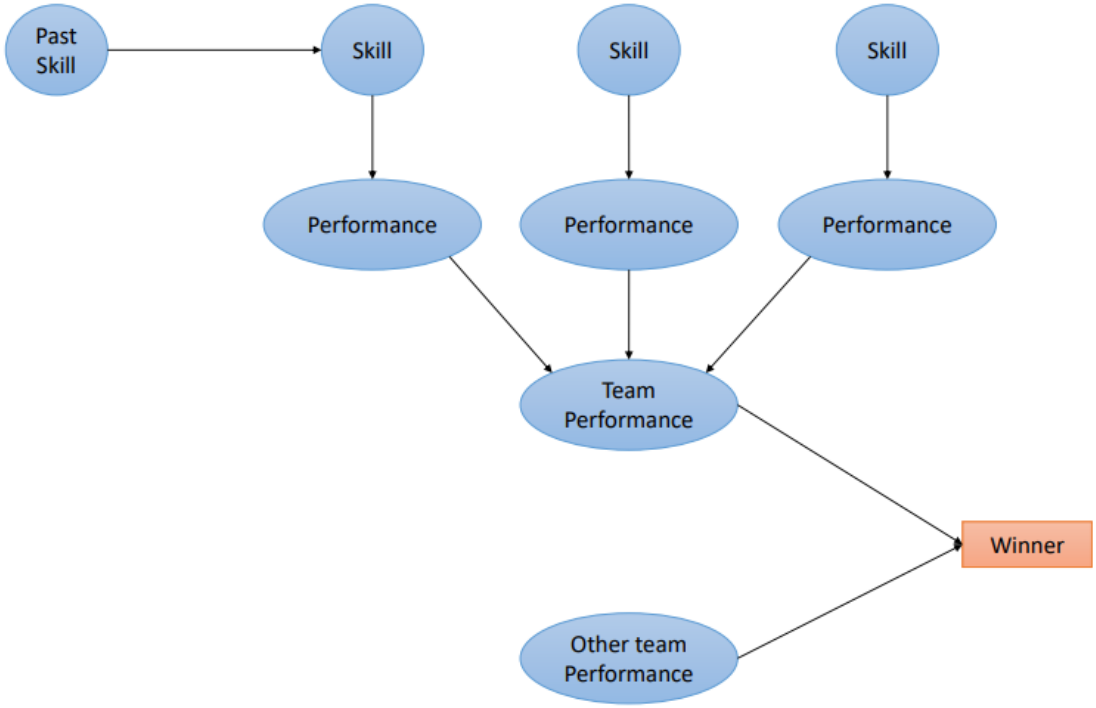


Figure 1.5: Summary of TrueSkill.

Chapter 2

Project aims, scope, and challenges

2.1 Project aim and scope

The project was proposed as an internship project would be executed in September 2018 - February 2019 period. The initial project definition and goals have provided below to clarify what were the pinpoints of the initial project:

> *Professional video gaming is growing to become as important as other sports. Therefore just like any other sport, identifying talents is a big challenge. We hypothesize that it might be possible to identify future stars by analyzing the behavior of gamers using the in-game data. In this project, we will use advanced machine learning methods to build a predictive model for gamers' ranking by analyzing the log files of the real matches. These log files contain very low-level information about the games state as well as gamers' actions. Our goal is to use this model to estimate the ranking of the players after observing only a small number of games. Additionally, we are interested to estimate future potential ranking of a gamer by looking at his past performance.*
> *Objectives*
>
> - *Collect and analyze Counter-Strike:GO game logs.*
> - *Extract features from the logs and builds deep learning models to estimate the ranking of the gamers.*
> - *Develop a predictive model for the future performance of a gamer based on today gaming behavior.*

However, for the scope of the project, we have found that it was hard to handle all project goals at a given period. Therefore, the project plan has been modified to expand project length with another six months of master thesis. In the first half of the project (internship), we were focusing on:

- **Data infrastructure**: We develop a data platform on AWS environment to execute all data-related operations given below.
- **Data collection**: The data has been scraped from several matchmaking servers.

- **Demo file analyzing**: All collected demo files have been uncompressed and analyzed by a self-implemented analyzer.
- **Text file post-processing**: After analyzing demo files, several issues need to be addressed, such as player identification matching between analyzed demo file nicknames and matchmaking servers usernames.
- **Data aggregation(labeling)**: Because there is no global aggregation of player ranks among different matchmaking servers, we have built our player ranks using TrueSkill algorithm.

Even though most parts of defined goals above have been implemented in the first half, several parts have also been completed in the first 2 - 2.5 months of the master project. These parts can be summarized as follows:

- Comprehensive accuracy test of the analyzer and additional data feature extraction ability have been examined.
- Faster implementation of player ranking system by modifying initial architecture and database design of player ranking and feature set have been carried out.
- Implementation of a matching algorithm for pairing usernames collected from matchmaking websites and extracted nicknames from demo file analysis have been finished.
- Execution of unit, integration, and system test of data infrastructure have been finished.

Therefore, we can summarize the high-level project objectives as follows:

- **Understanding game dynamics**: We know that CS:GO is a very enhanced game includes various individual and team-based dynamics and strategic standpoints. Understanding how we can deduce a meaningful summary of this comprehensive game using simple in-game features was one of our main focus points in this project.
- **Predicting player skillset**: One of the most important objectives is understanding how we can understand a player is a good one. This simple looking statement mediates several important subproblems. One of them is that we tried to predict player skillset just by using statistics collected and actions taken by a player in a match. Primarily, we tried to discover a relationship between player quality and match statistics and actions.
- **Discovering skillful newcomers**: As an outcome of the prediction task, we are attempting to discover talented and unnoticed young players. Therefore, Logitech can make benefit from the first mover advantage.

Like project objectives, we can outline the project scope and deliverables as given:

- A scalable and consistent data retrieval, analysis, and storage platform with collected demo and stats files.
- A highly successful and accurate demo file analysis tool.
- A comprehensive data analysis and modeling effort using collected data.

2.2 Project challenges

The project was one of the most longstanding and challenging projects among all supplied ones in the company because of the reasons given below. These listed challenges were the notable ones; however, there is a myriad of sub-challenges emerged by the solution implementation of these challenges mentioned in related chapters.

- **Absence of data**: Initially, the project was in the planing stage; therefore, there was a need for data gathering in an efficient manner (scalability, consistency, and maintainability concerns on data gathering).
- **Absence of easy-to-retrieve data source**: Another big problem was there was no data source one can get all relevant data from it without any extra effort.
- **Implementation of data infrastructure**: Initially, there were several ideas about the data infrastructure; however, design or implementation of a platform to handle all data operations such as data retrieval (scraping), data labeling, data storage, and data analysis were not achieved yet.
- **Difficulty of data labeling**: This project was designed to be a supervised learning project, which needs the outcome of observation (rankings of players at the end of a series of matches) as well. In this case, the outcome of our inputs is a kind of indicator for a player skill set. Even if matchmaking servers have domestic ranking and player skill evaluation system, there is a need for a global evaluation of player skill level holds for all records in our dataset.
- **A need for demo file analysis tool**: The main data (demo files) is gathered from several matchmaking servers and stored in a common S3 bucket. Nevertheless, a finished match content is encoded in each of those files (each demo file) leading to a need for efficient extraction (demo file parsing) and analysis of it. We could not found an analysis tool for demo files that meets our requirements; therefore, we have created our analytic tool. At first glimpse, the solution may seem clear to implement; however, this problem was a particularly challenging one because of subproblems struggled on implementation time will describe in system design chapter.
- **Feature extraction**: It is the fact that CS:GO is a strategic game that includes multiple complex factors on player and team level in order to get a win from a match.

Therefore, we need to keep all tactical and strategical entities on individual and team-based in our horizon by just using a feature identifier to represent them. The aforementioned task is a challenging one to judge, design, and craft new features to summarize better an already played match, which leads to the better judgment of player skill level as well.

- **Team modeling**: Even if we are focusing on individual performance, we already know that in CS:GO, individual performances are heavily affected by team-based behaviors. Therefore, there is a need to add extra features that would summarize team interactions better.

- **Learning**: Of course, like every prediction hurdle, finding out the most accurate and general approach to predict the ranking of a player is an important goal to achieve in the project. This challenge includes several sub-challenges including effective data cleaning and imputation, post-feature extraction, finding out the most effective model to extract knowledge, and effective parameter tuning to boost the model performance.

Chapter 3

System design

Although the data collection and processing was primarily covered in the internship preceding this masters thesis, many parts of the pipeline were revisited and significantly enhanced. Thus, rather than referring to the report of the previous project, the details of the pipeline are described in this chapter. In this part, first, the whole system working process will be presented and then each component separately described in details in addition to encountered issues on each part and their corresponding solutions afterward.

3.1 General schema

In figure 3.1, AWS RDS configuration of our project has been provided. On the first category, three different metadata databases have been deployed to store collected metadata of each demo and stats files by different matchmaking servers. On the second level, the statistics database includes several tables to store relevant match information, rankings of players involved in the match, and statistics of these players during the match.
In figure 3.2, the whole process of data processing has been summarized with an uncomplicated illustration. For an introduction, we can consider the whole architecture from bottom to top:

1. In the first step, demo files with related matchmaking statistics (stats files) have been collected with concurrent execution of pooled workers from different matchmaking web pages and stored in an S3 bucket. Moreover related metadata for demo and stats files such as match date, match map name, and parser version have been inserted to a SQL table (chapter 3.2.1).

2. After collection of data, demo analyzer has fetched and analyzed demo files concurrently, and it outputs a text file that includes player statistics per match. These statistics have been stored in a SQL table for data labeling stage with default ranking values. Text files have been stored in the S3 bucket as well (chapter 3.2.2).

3. All text files have been post-processed for the purpose of matching identities of in-game nicknames with correct usernames (chapter 3.2.3).

4. After post-processing stage, player records have been labeled using TrueSkill algorithm starting from earliest match to latest match (chapter 3.2.4). After labeling data, it is ready to analyze.

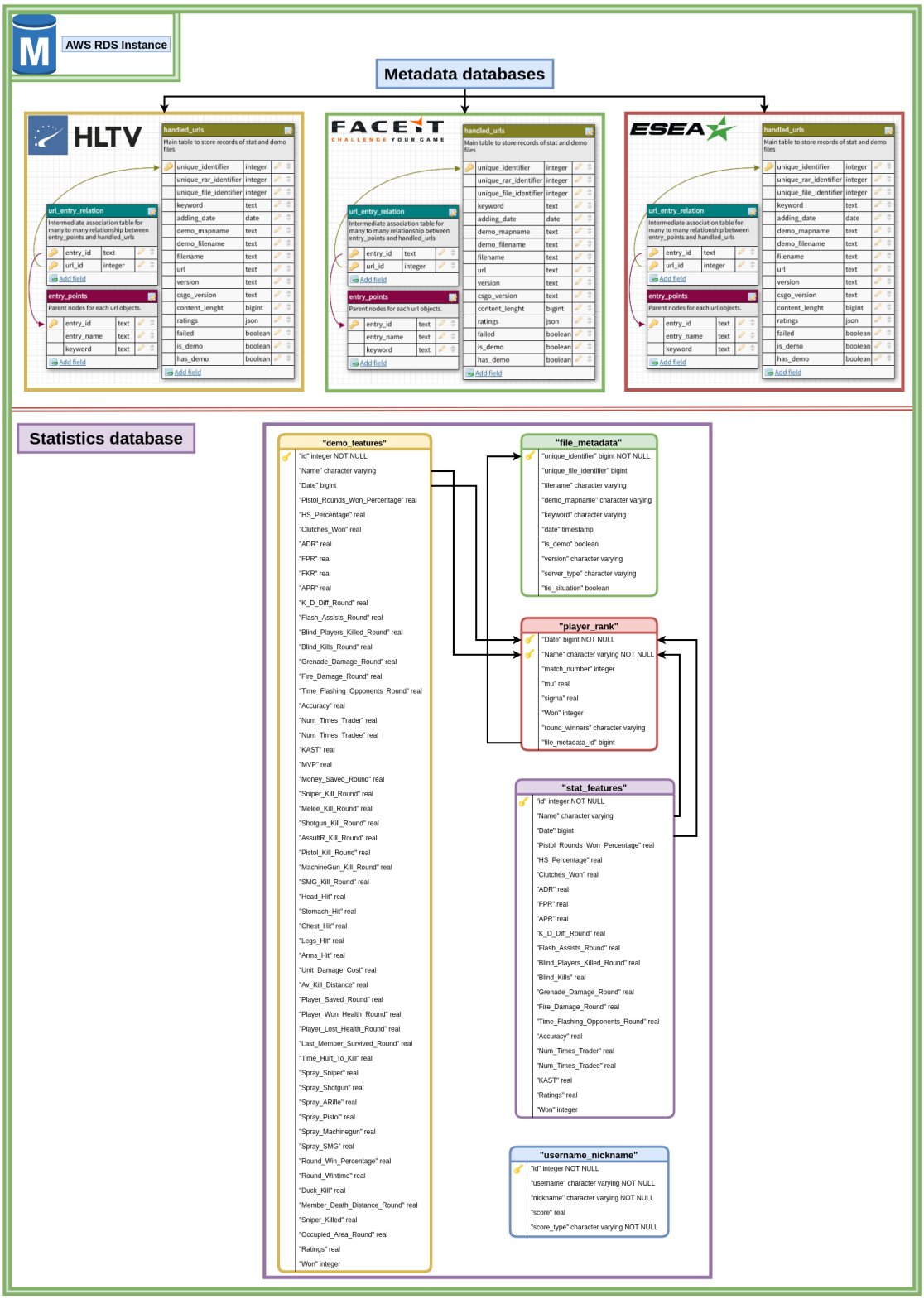


Figure 3.1: DB schema of the project.

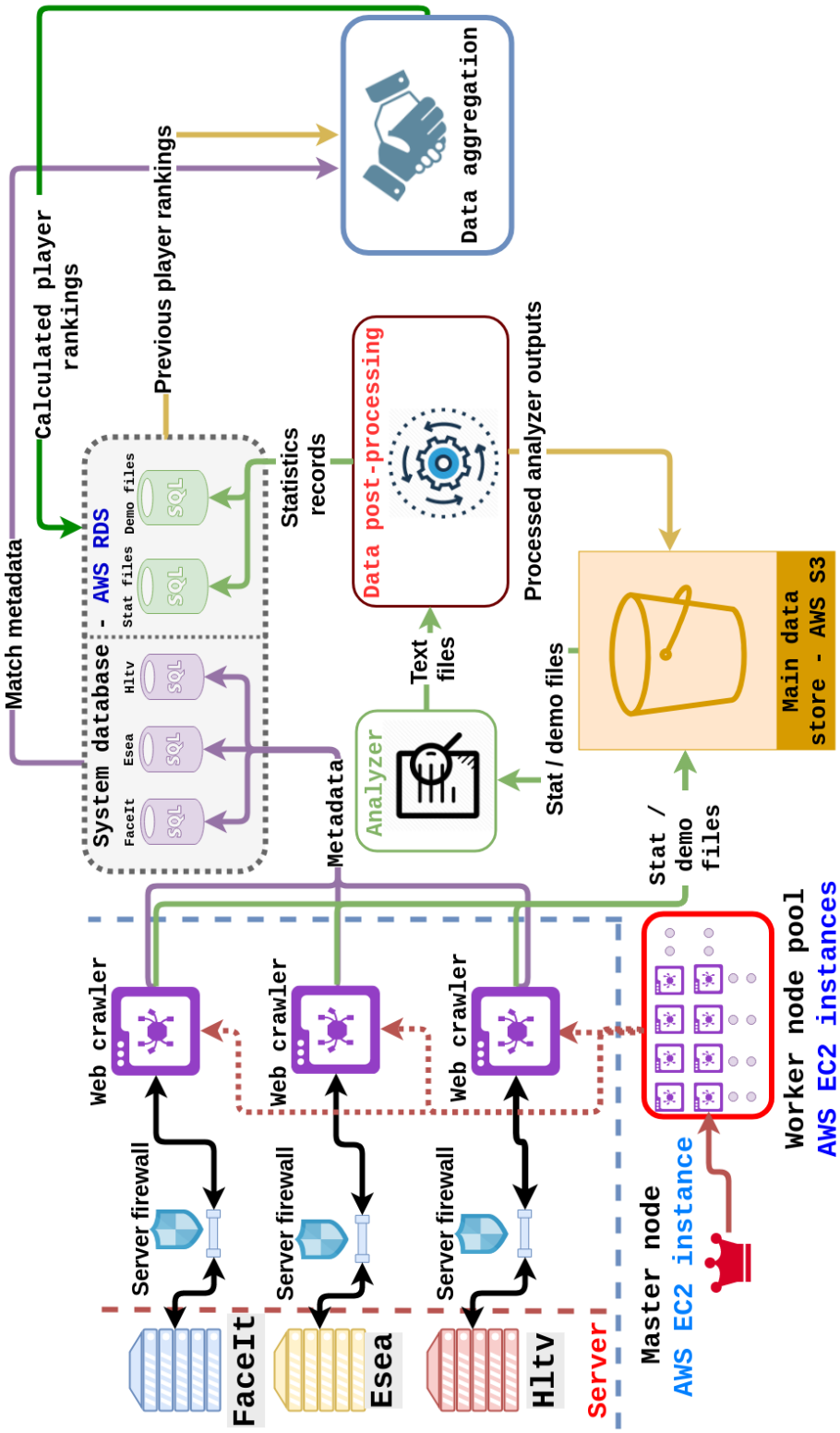


Figure 3.2: General summary of whole system.

3.2 Components

3.2.1 Data collection and storage

As mentioned before, the data was not available to collect at the beginning of the project. Therefore, a need for efficient, scalable, and consistent data collection and storing system emerged initially. Initial challenges for this part are given below:

1. **Absence of easy-to-retrieve data source**: The main objective in this step is to collect as many CS:GO match demo files as possible. However, there is no mainstream data provider that data can be collected from it without any extra effort. At first glance, Steam API would be a good candidate to collect user demo files; however, Steam classified these demo files as the private property of a user, and anyone can only retrieve these demo files with user Steam ID, which is only accessible for the user itself. Therefore, several alternative matchmaking servers have been elected to retrieve data from. These servers have been introduced in previous chapters, namely HLTV, Faceit, and ESEA. However, the emerging problem is that these servers, excluding Faceit, have no Data API to fetch data in bulk. As a consequence, these server websites have been scraped with different scraping approaches for the retrieval of demo files.

2. **Temporary demo files:** All matchmaking servers have stored demo files for a certain amount of time, 60 days at max. Therefore, when scraping old dated demo files from servers, it is highly probable that the related demo file will not be available on the website. An alternative approach is needed for already played matches which demo file does not exist anymore.

3. **Data duplication**: It is noted that the demo file sets in matchmaking servers are not static, which means every day there are a bunch of new demo files added to each server. Consequently, the collection process is executed periodically to accumulate new demo files. Also, already collected demo files should not be fetched again in order to evade duplicate collection. Duplicate collection problem even extends among individual servers, meaning it is highly reasonable that there would be a match collected from different matchmaking servers over and over.

4. **Consistent storage**: This project was not scoped by this master thesis project, meaning it is an ongoing process; the data will be utilized several times with different approaches, stages or subprojects by the company. Therefore, rather than collecting, using, and discarding data; a systematic, consistent and incremental (new demo files are added nearly every months to the system from matchmaking servers) storage is crucial to develop.

After evaluating the problematic side of data collection, it is proper to elaborate the system design and solutions for the problems described above. In figure 3.3, which will be used as a visual guide of data collection system design, you can examine the overall design of the data collection system.

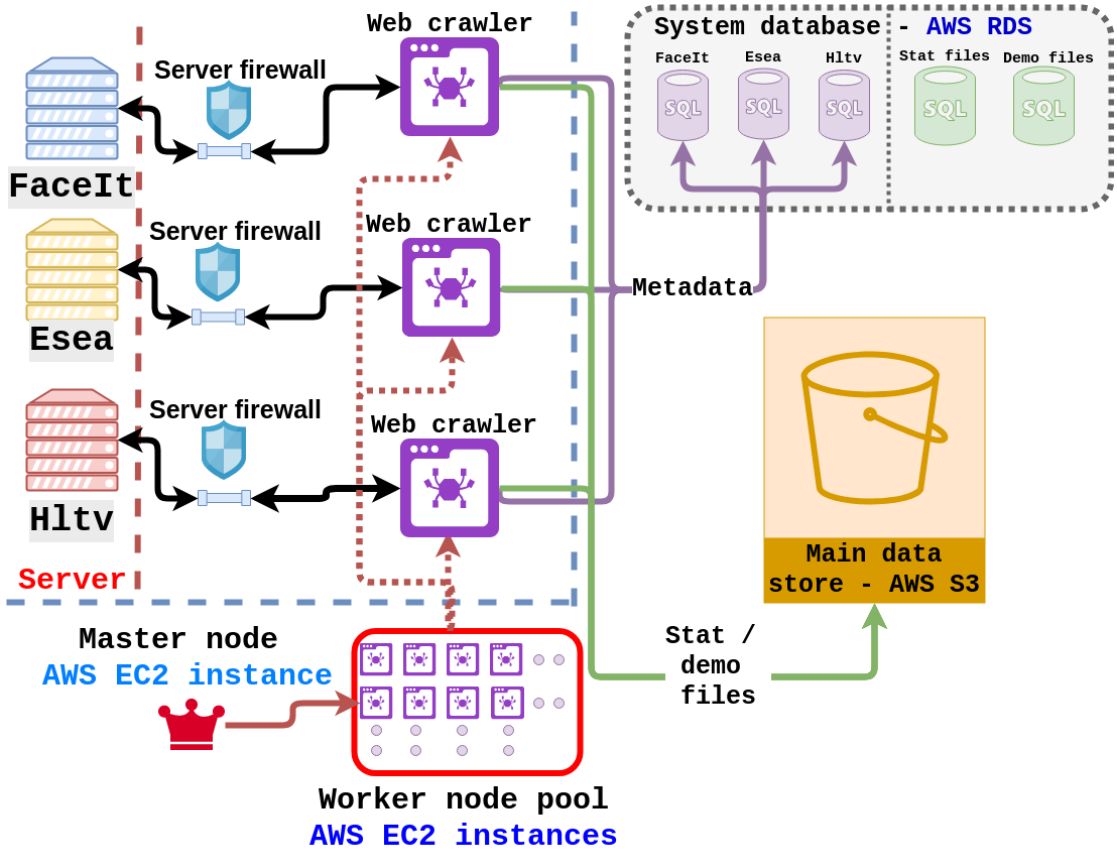


Figure 3.3: System overview of data collection.

The first step is to retrieve relevant data (CS:GO demo files) from related servers. The main difficulty in this step is that all servers, excluding Faceit, have no data API indicating that a unique scraper has to be implemented for each website separately. Moreover, all non-API websites have been protected by Cloudflare [1] anti-bot protection system. To overcome these problems, a unique scraper for each website has been implemented as needed with a different pattern of reaching a match page because usually there is no direct link for each match in a website, meaning several requests per match have been made in scraper implementation. For example, for HLTV, firstly the main list of match page has been scraped, and each match page link has been extracted from the collection. Also, then for each link, a request has been made to crawl match page and get the match statistics page link. After getting a match statistics page link, demo file download link has been extracted from this page finally.

Furthermore, to bypass the anti-bot protection system, a third party python library has been adapted to our project named *cloudflare-scrape* [2]. The library solves JavaScript challenge provided by CloudFlare and yields a token to make further requests to the related web pages. The problematic side in this approach is that Cloudflare is updating

[1]Link: https://www.cloudflare.com/
[2]Link: https://github.com/Anorov/cloudflare-scrape

its bot protection system regularly, which means that for each protection update, there is a need for *cloudflare-scrape* update as well. If relevant update in the bypass system is not provided, the web page trying to crawl will not be available to visit, and Cloudflare protection will respond with a captcha page, which is currently impossible to bypass for this project.
Another big problem leading for anti-bot protection is that IPs of collector workers have been recorded by anti-bot protection system (Cloudflare) and if the request number is above a certain and unknown threshold per worker, determined by website admins for a unit of time, the IP of the worker is banned. After the IP ban, the website will be temporarily unavailable for this IP and no possibility to crawl the website for a given amount of time (usually one day per banned IP). We have found a solution to overcome this problem. On AWS EC2 instances, when an instance has been stopped and restarted again, the public IP, as well as private IP of the instance, has been reassigned [3]. The IP refresh property is an outstanding practice for us because when we get an IP ban for a collector worker by Cloudflare, we can stop and restart the collector worker and continue to scrape with a newly assigned public IP. So that, every time a scraping sub-task finished or a collector worker gets an IP ban, scraper master node is stopping and restarting the worker instance for an IP refresh.
Apart from the advantage of separate workers will have diverse IPs, which lessens the probability of getting an IP ban, there are also several benefits for practicing a pooling of collector workers. The first obvious benefit is the speed of data collection. The script running on a master node will map available jobs to available collector workers; therefore, workers can parallelly scrape web pages, which promotes robustness and execution speed of the data collection system. Another benefit of the pooling method is that a script can generate several pools with separate master processes running in parallel, and each separate pool can focus on different jobs. For illustration, one pool of workers can scrape demo files from HLTV, while other pool of workers can crawl web pages from a different server such as ESEA at the same time.
In terms of storage, there was a necessity for the stable, scalable, and consistent data store, that is why AWS S3 has been picked. All demo files have been saved to S3 bucket categorized by job type (HLTV, Faceit, and ESEA) and specific padding to job type. In order to fast uploading, listing or retrieving demo files, the same S3 bucket has been mounted to each worker using a third party library[4]. On the contrary to demo files, metadata of demo files has been yield in a relational database precisely AWS RDS instance. The main advantage for metadata database is that using metadata, we can check whether a specific demo file has already been downloaded or not, which is very beneficial to circumvent duplicate processing of a demo file.
As an introduced issue, demo files have been served for a limited time frame on match-making servers. To fill the gap caused by missing demo files of old matches, we have accumulated match statistics tables provided by match pages and stored in the S3 bucket

[3]Link: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Stop_Start.html
[4]Link: https://github.com/kahing/goofys

as a text file. Because these match statistics tables are typical to matchmaking servers and they include limited information for the match compared to demo file analysis, only a different subset of features among all available features for a demo file could be collected, which leads to the synthetically heterogeneous feature set problem will describe in details. After examining issues and related cures, now we can summarize the process of data collection. First, master node instantiates a specified number of workers and given jobs as separate pools (Number of workers, type, and the number of jobs are the master script parameters which can be tuned). Note that, all workers are identical and have been created from pre-tuned AWS AMI, which boosts the robustness of implementation and reduces the probability of a bug. Then, each available job will be assigned to each available worker randomly selected from the worker pool, and the master will periodically probe whether any assigned job has been done. When a job is completed with success, it is done for it; otherwise, the failed job will be appended at the end of the job list to retry it. It is crucial to note that because workers are collecting and handling files concurrently, each collected demo file has been saved in a separate directory, which reduces the probability of concurrency-related errors stems from race condition or common resource use. When all free jobs have been scheduled and processed, the master script will shut down all workers and output failed job list.

3.2.2 Demo file analysis

We are mainly collecting finished match context in a demo file format explained in the introduction chapter. Therefore, parsing these demo files and analyzing parser results to extract useful features are two crucial tasks to build a data set. However, there are several tough challenges for handling demo files described below:

- **Different types of demo files**: Demo files are recorded while the match is playing by a matchmaking server. Even if there is a common consensus for event naming and typing, and file format, the pattern, and meaning of events can differentiate from one server to another. Therefore, it is crucial to develop a demo file analyzer that can handle different types of demo files as possible.

- **Missing demo file header**: Another fundamental problem for demo files is that some demo files do not even include a demo file header. A header is a useful data partition in a demo file to fetch match metadata such as match map, tick-rate of a server (tick-rate is very beneficial value to calculate an event duration), and the number of frames.

- **Missing events**: The most crucial problem existed in demo files is that frequently, several important events are missing or malformed. For example, by default, roundStart event indicates a start of a round with additional parameters such as which round is starting, and what is the tick number. It is very probable that a round can start (usually first round) without emitting any roundStart event. Therefore, additional fuzzy logic has to be developed to find out whether a round has

been started without the existence of roundStart event by probing other indicators such as the end of the previous round or existence of any previous round. When we consider all possible events in a typical demo file possible to be missed, the complexity of the problem is getting huge.

- **Invalid events**: Apart from missing events, there could be several invalid events in a demo file as well. There could be a kill event with a disconnected player or exploding flash bomb event thrown by a spectator. It is crucial to filter those invalid events with illogical arguments.

- **Participant issues**: In a match, apart from players, there are other participants as well, such as coach of a team, server bots, and spectators. There is a need to ignore non-player participants not to get the wrong player list and correctly classify valid events. Another problem is that players can be disconnected and reconnected at any time in a match. At this point, it is crucial to save the disconnected player state and recover when the player gets reconnected.

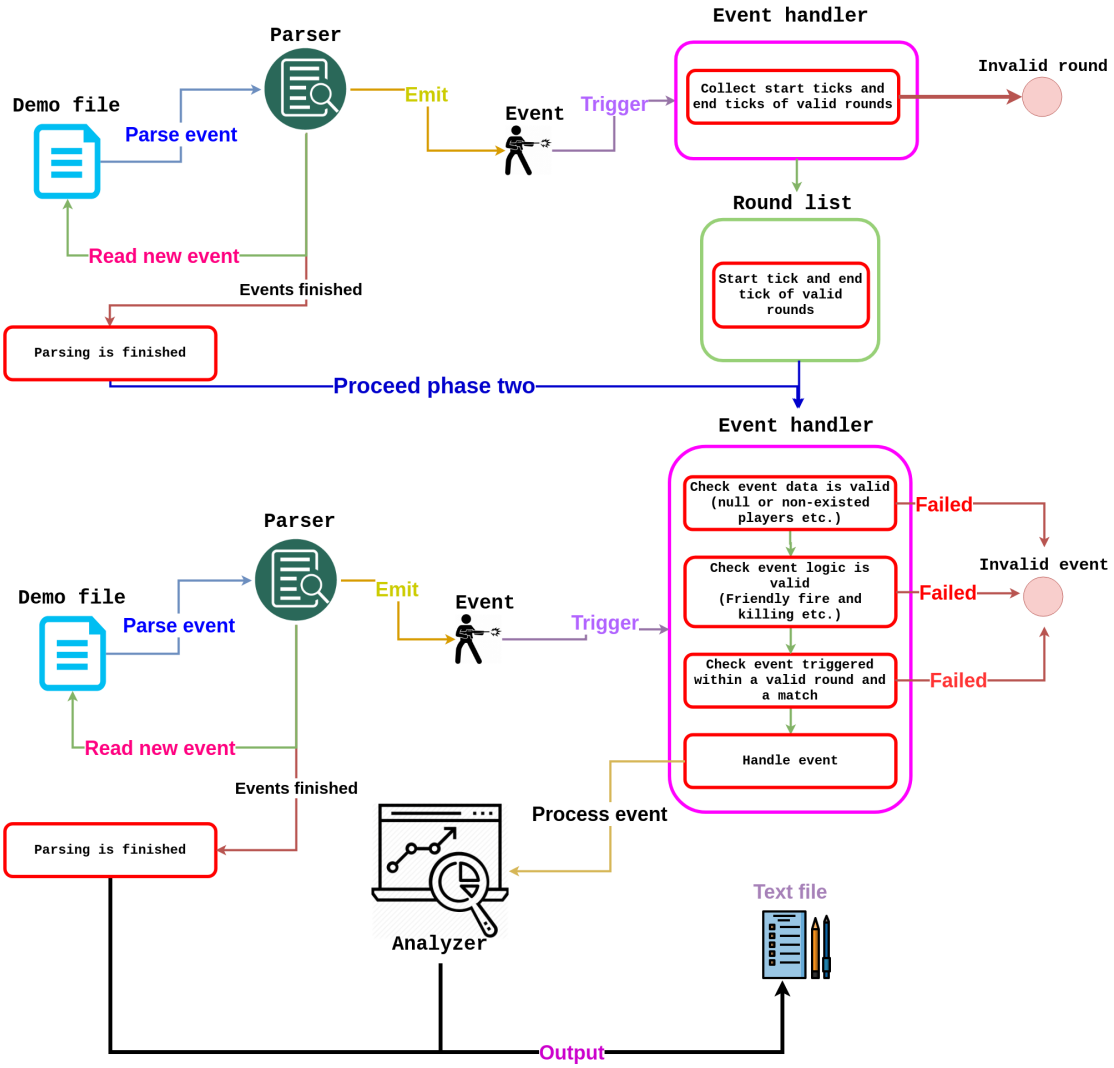


Figure 3.4: System overview of demo analyzer.

In figure 3.4, another visual guide to scrutinize how the demo analyzer works. In the system, there are two primary components, specifically parser and analyzer. The parser parses given low-level demo file and broadcasts a signal for each event encountered. On the other side, the analyzer has subscribed these event signals, and every time an event signal is emitted, the registered event handler takes care of the emitted event. By sequentially processing events, the analyzer state is altering, and the outcome of events is yield to the analyzer.

Firstly, it is significant to pay attention that demo file parsing is a linear and sequential process, which means parser cannot get information from non-parsing part of a demo file

unless it advances ticks and parses the related part. Because we cannot get information from future events located in later ticks and check whether an event is a valid one by considering the future state of a match, we need to find a way to recover from the buggy analyzing stage if it occurs. As can be deduced from discussed issues, it is highly probable that we can process different types of events from an invalid round (round with no roundStart event or no roundEnd event, rounds with an invalid round number), which will lead to a wrong state in analyzer needs to recover from.
Because we cannot know which round is valid beforehand while processing events occurring in this round, we need to find a way to consider only valid events and extract correct features. A very straightforward solution for the problem is that ever time analyzer starts to analyze a critical and buggy section (like roundStart event, or matchStart event), it can serialize (save) its state to create a recovering checkpoint. When the analyzed part shows up an invalid section, it can recover to its latest saved checkpoint and continue to analyze remaining events while ignoring the buggy section. However, there are severe problems for this approach in terms of our application. The first issue is that because of an excessive number of possibly buggy sections; the analyzer would need to save its state many times leading an excessive amount of memory use and a hard to track checkpoint set. Second, there is a significant overhead for saving and recovering a state of analyzer mainly sourcing from the underlying parser component, which means this overhead may reduce the execution speed of the analyzing stage.
Therefore, it is clear that we needed to find a better way to correctly analyze demo files (here a correct analysis of a demo file means deducing correct statistics from a demo file that suits statistics published in a matchmaking web page). For this reason, we decided to parse the same demo file twice, which enables pre-detection of valid rounds and increases the accuracy of the analyzer.
In the first parsing stage, the main aim is to find out which rounds are valid and therefore needs to be considered by the analyzer. In this step, no event handlers other than match and round event handlers (like roundStart, roundEnd, and matchStart) have registered to receive event notifications emitted by the parser to speed up this step (Several trivial player event handlers have been registered as well to deduce the validity of a round like playerHurt event). After finding out which rounds are valid by checking several fuzzy logic indicators (like whether anyone was hurt, or killed, a round started and ended with correct event type and order), start and end tick of each valid round have been yield to a list to proceed next parsing session.
In the second parsing stage, all event handlers including player event handlers (like killEvent, deathEvent) are registered to subscribe emitted events by the underlying parser. In this step, a complete analyzing of demo files is executed by using the correct round list collected from the previous parsing stage. All arguments of a player event are checked for each emitted player event to avoid invalid processing of events (like whether a player in an event actually exists and connected, or involved players are in proper team sides). After finishing the second parsing stage, the analyzer outputs collected statistics categorized by features to a text file stored in the S3 bucket. The output statistics of players are stored

in the database as well for the stage of data labeling.
When we examine the advantages of our approach over the save-restore paradigm, there are several outstanding benefits emerging. First and the most obvious benefit is that memory consumption of our approach is far less compared to saving the state of parser and analyzer every time a buggy part of a demo file starts. This feature is an important one because we are analyzing demo files concurrently, meaning much more total memory usage compared to single execution. Second, our approach is far more simple and easy to implement because there is no need to determine which parts of a demo file can be buggy, which complicates the implementation of the former approach or which recover point is convenient. Third, it is much more accurate compared to the first approach, because a separate parsing session (first stage) is only focusing on determining correct boundaries and validity of events by finding out which rounds are valid while second parsing stage filters wrongly structured player events. Our approach may be a bit slower compared to the first approach because of two parsing stages per demo file; however, it is important to note that there is a significant execution overhead in the first approach for saving and restoring analyzer/parser state as well.
For the second experiment, several drastic improvements have been made on the analyzer architecture for extracting more advanced features. Most of the features adding on the second experiment need a pre or post-conditioning or status saving checkpoints to help the extraction of the related feature. For illustration, to assessing spray amount of a weapon just before a player killed someone needs a positional saving of weapon cursor just n seconds before killing him/her. However, as we already mentioned that the events in demo files had been encoded sequentially, and we cannot know which event will happen on which tick and therefore exact time to save this pre or post checkpoints status. To overcome this problem, we have added a scheduler to schedule these pre and post checkpoints as if they are needed to extract related feature. On the first parsing session, scheduler creates a checkpoint task and adds the checkpoint queue if the currently processing event needs pre or post handling with the execution tick. On the second parsing session, the scheduler checks the checkpoint queue on each tick ends whether there is a checkpoint task that needs to be resolved at the current tick.
The implementation of demo file analyzer has been published on a public repository[5].

3.2.3 Demo text file post-processing

After analyzing demo files in the system, we have one text file per match that includes player statistics collected during a match (one row per player in a match). At this stage, there are several appealing issues need to be fixed. The first issue is the identification of players. In a match, a player can use any type of string as an in-game nickname (including special characters and emojis) and alternate between different nicknames on a series of matches, which causes a critical problem that we called as player identity matching problem. In matchmaking websites, player names have been served with

[5]Link: https://github.com/quancore/demoanalyzer-go

constant usernames chosen by users while registering to the websites. However, in-game nicknames usually differ from these website usernames creating identification problem of a player in a match. This problem also causes undesirable effects on our player ranking system because we are using matchmaking server usernames while storing player statistics and rankings in our database as unique identifiers(in a better way, Steam IDs of players can be used for matching and storing player information in database, however, because of the confidentiality of Steam IDs, matchmaking servers are not providing these IDs on their websites). Therefore, a need for a specific approach for matching two different player identifiers has emerged in this step. The solution algorithm for the problem has been represented in figure 3.5. The solution mainly depends on previously mentioned stat files (a form of text file includes player statistics scraped from matchmaking web pages. It is in the same format (one row per player) with a text demo file, but they may differ regarding feature sets.) to retrieve correct usernames published on matchmaking web pages. At the first stage, in-game nicknames in a text demo file (text files produced after analysis of demo files) have been cleared by removing non-alphanumeric characters such as comma, dot, or emojis. After that, all nicknames and usernames have been lowercasing to maximize string matching probability of possible pairs. Then, all possible pairs between two sets of identifiers have been tested and for each pair Damerau–Levenshtein similarity [16] has been calculated. All nicknames coming from the text demo file has been matched with the best candidate usernames from the stats file if the similarity metrics is above a predefined empirical threshold. For nicknames and usernames which cannot conduct any matching, they are firstly categorized by using the outcome of the match separately. After that, Cosine similarity for each possible pair between nicknames and usernames has been calculated separately for winner and loser team. All nicknames have been matched with best-scored usernames if the similarity value is above an empirical (and different) threshold. At this stage, usually, all nicknames have been matched with usernames; otherwise, remaining nicknames have been discarded from text demo file and not recorded to the database.

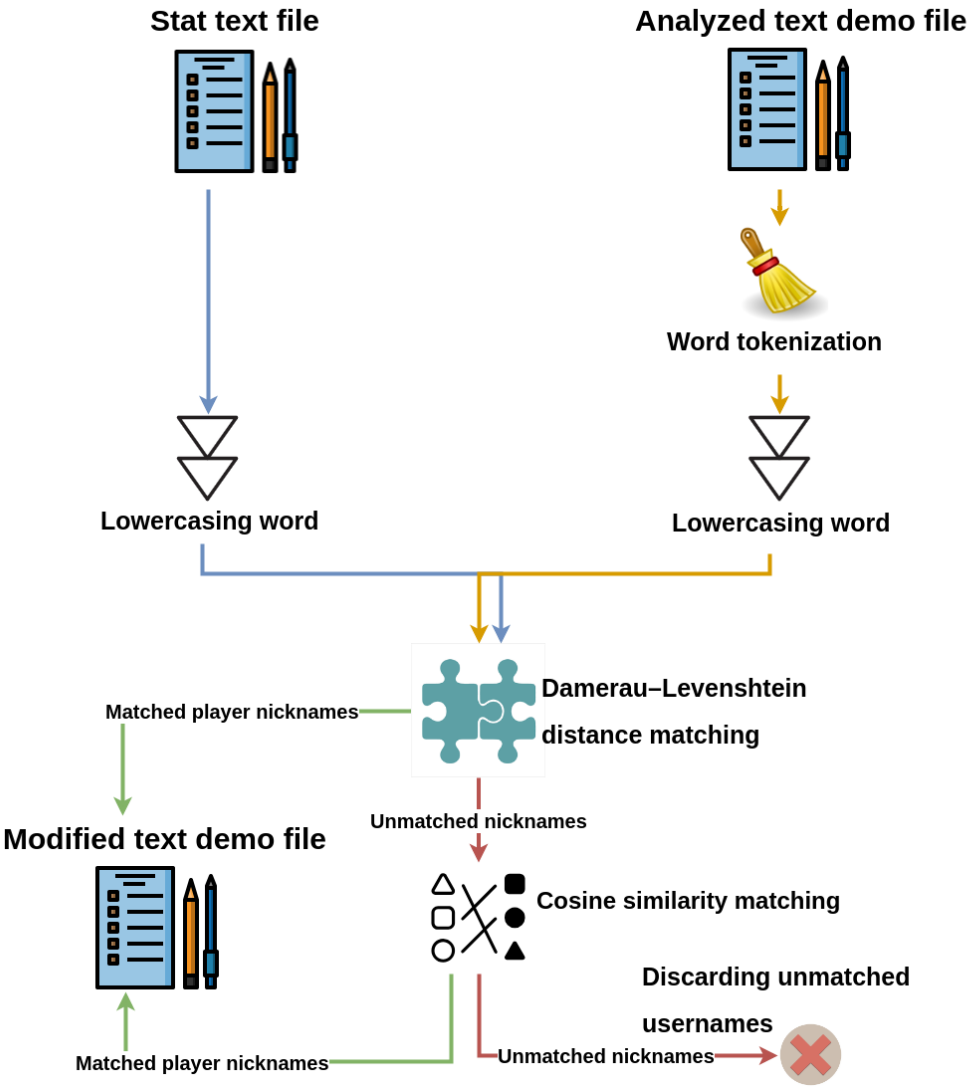


Figure 3.5: System overview of demo text file post-processing.

3.2.4 Data labelling

The project has been initiated by a supervised learning task, which needs a set of outcome for a set of observations, namely, data labels. However, because of the nature of data and issues described above, data (demo files and stats files) were coming from different matchmaking servers meaning there is no common metric to evaluate all player skill levels initially. Therefore, we decided to use TrueSkill algorithm described in the introduction chapter to estimate each player latent skill, which will be used as data labels.
Therefore, our learning task becomes a pipeline trying to model a complex relation between player statistics collected during a match (all collected features will be described in the next chapters) and TrueSkill labels built for each player separately for each time point. However, there are several practical consideration while we were integrating

TrueSkill on our project. The first prominent consideration is that we need to model player skills "close enough" to player latent skills, which means TrueSkill should converge real unobserved skill of players to craft correct data labels. For instance, if a player has few matches recorded in the system, TrueSkill cannot sure about the convergence of real skill for that player. Moreover, the primary consideration in this problem is to find a proper least possible threshold value for the minimum number of matches per player while guaranteeing a good convergence of real skill of this player leading a trade-off need to be considered. The trade-off in this context is the amount of data available after filtering with the mentioned threshold and the goodness of convergence to the real skill of players. Intuitively, when the number of observed matches for a player is increased, TrueSkill will be more convinced about the prediction of the real skill of the player. However, if we filter all players with a high threshold value, we will lose a significant amount of data to train our player model, where data is in a limited situation for this project. Therefore, it is essential to achieving a stable threshold value to maintain the balance between available data and goodness of convergence.
For the purpose, we started to scrutinize the original paper published for TrueSkill algorithm [8]. Despite there is a chapter about the behavior of algorithm convergence, the chapter is not so informative that it is not providing enough clue about a rough calculation about this possible threshold value. For further investigation, FAQ chapter on Microsoft Lab page [17] has been used, which includes various beneficial information about the algorithm convergence performance. The rough calculation for the threshold using provided context in the page as follows:

1. Firstly, we should provide a methodology to procure the following formula. Imagine that a matchmaking system has n players has to be ranked. In order to uniquely encode the ranking of a player (from 1 to n), the algorithm needs to $\log_2 n$ bits per player. Now imagine that two players are playing a head-to-head match. The ranking algorithm will need $2 * \log_2 n$ bits to represent rankings of these two players, which can be generalized as $n * \log_2 n$ for n players. Now, let us check how many bits we gain per match to compensate for the information needed for ranking a player. For two players(or teams) case, we are gaining one bit information per player(or team) (whether win or lose) on each match, so in order to compensate the information required for ranking n teams in total therefore, in a match with k teams, information gain per match would be $\log_2 k!$ bits (there are $k!$ different possibility of ranking k teams per match). Then, the number of matches needed to converge the true skill of a player is given the following equation:

$$\frac{(k * m * \log_2 n)}{(\log_2 k!)}$$

   where k is the number of teams involved in a match, m number of team members per team, and n number of total players in the system (representing number of unique ranking places as well).

2. If we further investigate the formula, the author of the page [11] indicates that information gain per match may be higher than one bit depending on the draw probability of a match (depicted as a graph in the referenced page [11]). On the contrary, the author signifies that not all the matches are equally informative (providing at least one bit) to gain knowledge about skill because of the variance of player performance during several matches. Even worse, some matches can lead to information loss up to %75 (the case of a lower-ranking player (or team) won the match). Therefore, they observed 2 or 3 times greater number of matches needed per player to build the best approximation in practice. However, we do not reflect this empirical constant in our calculation because it increases the minimum amount of needed match per player (threshold value) excessively, which filters a huge amount of data.

3. After this examination, we calculated a set of possible threshold values $t \in ([0, 100])$ and the minimum amount matches (calculated using formula 1) needed to converge player ranks remaining after filtering the players having the number of matches below this threshold. Figure 3.6 shows the results we get, and it indicates that the best matching for these two metrics corresponds to 99 matches per player. However, this threshold value would lead to an excessive amount of players being filtered, meaning significant loss on the dataset. Consequently, we decided to find more optimal threshold value giving a good approximation for player rankings (not the best) while retaining the big proportion of data.

4. As indicated before σ value is a good indicator for the certainty of latent player skill for TrueSkill. Then, we decided to check the sigma(σ) value of several players having the highest number of matches in the system. After an inquiry of the changing trend for σ, we decided that 30 matches will be a good compensation point between the goodness of convergence and the available amount of data, which means all players having less than 30 matches will be discharged and earlier matches than 30th match will be discharged for the players having more than 30 matches in the system.

Before finishing the data labeling chapter, the last thing to mention is the level of granularity regarding when we are updating rankings of the player. As a naive approach, we can update player rankings after a match ends. However, there is another approach that we can update team member rankings by the result of the rounds instead of the match ends. Each round can be seen as a tiny matching of two teams, and for each round, one team tries to beat the opponent team. The advantage of using round results is; obviously, there are at least 16 rounds per match, which means we can compare two teams much more frequently using round results compared to match results. As described in the TrueSkill chapter, when a player plays more matches, TrueSkill algorithm becomes more confident about the skill level; therefore, round results may seem more suitable for updating player skills. In the first experiment, we have used results of matches to update player skills, and in the second experiment, we calculated player skills based on round

results of demo files (stats files does not include round information other than end game statistics).

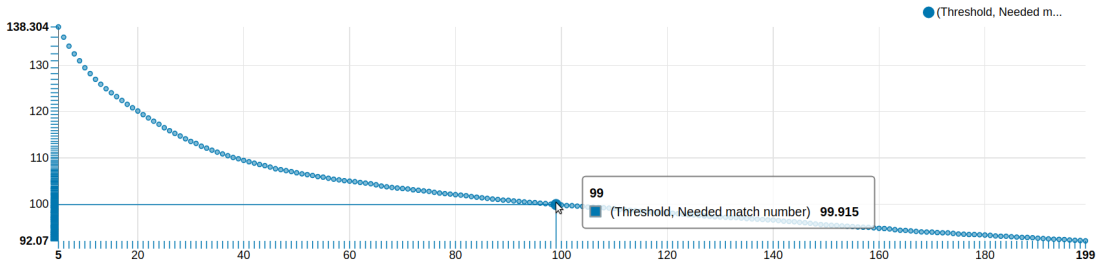


Figure 3.6: Possible threshold values - Number of matches needed after filtering.

The second issue is building an algorithm to correctly estimate player rankings while respecting the date of matches for a player. Our approach is as follow (depicted in the figure 3.7):

1. Because TrueSkill uses previous μ and σ values recorded in the previous match as prior observation for the current match, all matches in the system have been sorted by date using metadata of demo files. While we are sorting the matches, we made a logical assumption that one player can belong to one match at a time. The problem in here is that HLTV is packing several demo files (different subsequent matches between two teams) in a same compressed file and providing the same timestamp for all demo files, which violates the fact of being in only one match at a time. Therefore, we shifted timestamps of subsequent matches after the first match when recording metadata of demo files to the database to alleviate the temporal issue. At the end of the stage, we have uniquely dated and sorted demo files in ascending order ready to process.

2. After sorting demo files, we process demo files starting from earliest to latest one. The processing stage (in the figure 3.7) consists of retrieving latest μ and σ already calculated from the database (The records in the database have been inserted during demo file analyzing and post-processing stage with default μ and σ), calculating new μ and σ for each player involved in the match, and storing these newly calculated values in the database.

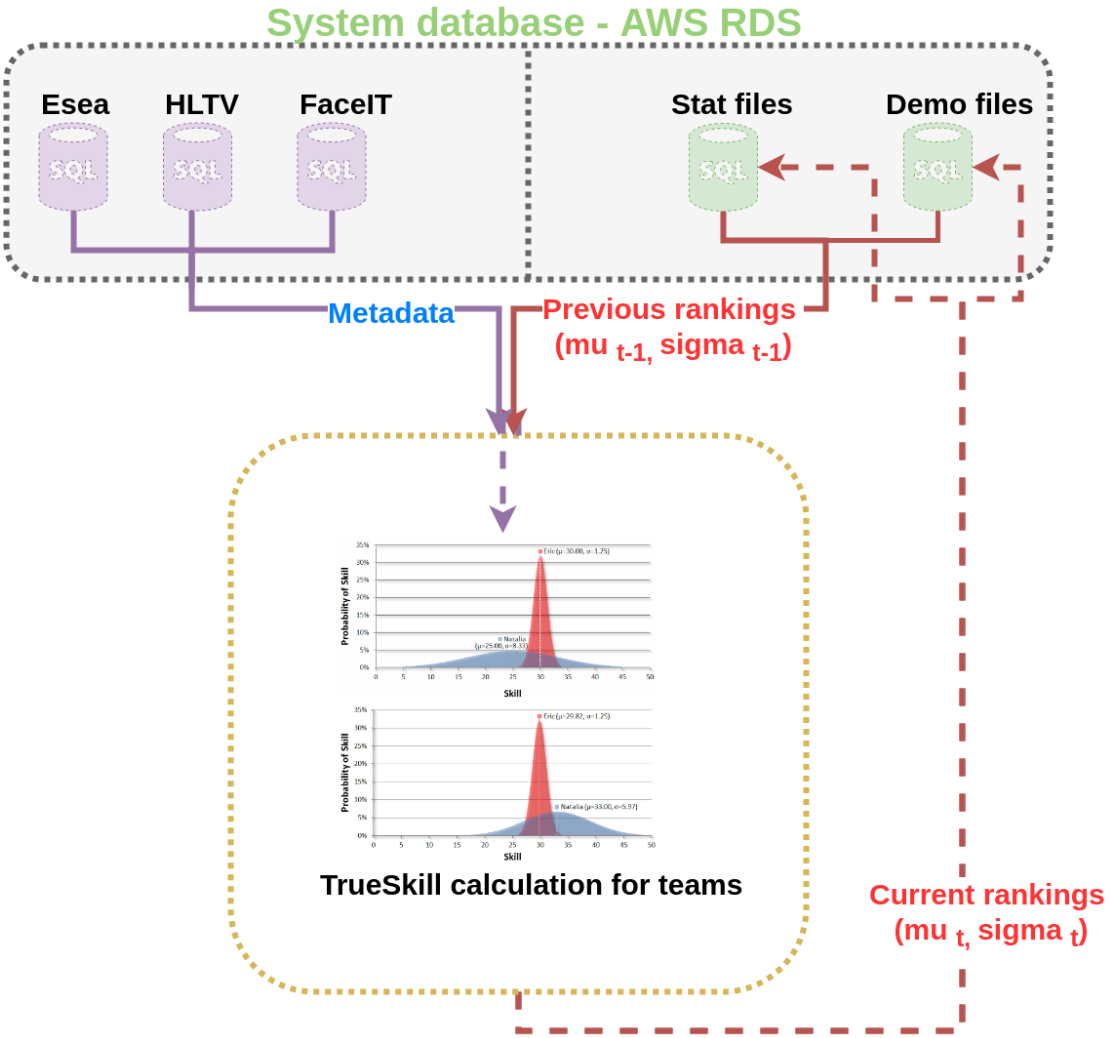


Figure 3.7: Summary of data aggregation algorithm.

Chapter 4

Experiments

In the following sections, we will first introduce the common properties and components for all experiments and then continue with the content and configurations of experiments. For each experiment, we will provide data statistics, and describe feature set; after that, we will provide detailed explanations about data analysis pipeline including data wrangling, feature selection and learning models tested for this task. For the first experiment, we will use match results to update player skills. On the second experiment, we will change the several parameters in configuration and test their effects. Contrary to the first two experiments, we will focus on the temporal relation of player records rather than evaluating each record as an independent entity in the third experiment.
The methodology of the experiments as follows:

1. First, the analyzer described in the previous chapter has been modified to extract more features, as mentioned above. Therefore, the analyzer can be perceived as a modifiable feature extractor component. After modifying feature extractor for each experiment, several tests have been conducted to validate the correctness of extracted features and robustness of analyzer (not giving run time crash, or logical error while analyzing a demo file).

2. In the second step, the whole demo files have been analyzed afresh with the distributed architecture described in the previous chapter in order to extract a newly modified feature set from demo files.

3. After analyzing all demo files, we need to build labels (skill levels of players via TrueSkill) for each processed match to learn from data. Therefore, the data aggregation step is executed at this stage using either match results or round results to update player skills.

4.1 Dataset properties

There are two origins for the data mentioned previously. The first one is statistics files (called stat or stats files in the thesis), which were scraped from matchmaking server

web pages and stored on S3 bucket in text file format. These files have a limited feature set because matchmaking servers provide a very brief analysis of a match compared to our implemented analyzer. In the first experiment, these files have not been used for any purpose because of incompatibility with our extracted feature set during the demo analyzing stage. In the second experiment, these files were only used for visualization and pretraining of a neural network model.
The second source of data has been acquired from analyzed demo files. Via the system described in the previous chapter, all demo files have been analyzed, and the output files of analysis, which are text files as well, have been stored in the S3 bucket. These files have a broad feature set compared to stat files, and it can be modifiable as long as feature extractor (analyzer) has been improved. In all experiments, the primary data source of conducted analysis is these text files because of involved feature set.
These two types of files generate a time series per player together. For each sample of a time series, there is a player ranking recorded and a set of features collected from a match. The sampling frequency and length of each series are varying player to player. For each experiment, except the last small one, statistics of the dataset such as number of users and number of records are provided.

4.2 Features

It is important to remark that extracting all diverse aspects of a played match in order to model a match environment precisely was one of the uttermost subtasks in this project; we have inserted new features for each experiment by modifying the analyzer. This step enlightens another important part of feature extraction, which is crafting incrementally unique and creative feature ideas to extract hidden properties of the match. This was especially challenging and requires a special focus on creative statistics elicited from in-game events in a match.
To properly summarize the game, we have grouped the type of features into several diverse categories. The feature groups and example features for each group are:

- **Economy management**: Includes everything about money management on individual and team-basis. Saved money, type of rounds in terms of equipment values, spent money, and values of dropped items for team members are just some examples for this category.

- **Weapon management**: Includes how a player manages his weapons and ammo capacity. The average number of bullets to kill an opponent, type of weapons to kill opponents are just several samples for this category.

- **Skill-based features**: The most extensive group among them, and it represents all aspects of raw player skill. Percentage of kill distributed on body hit groups, kill and damage execution speed, hit accuracy metrics, response time, and distance-based metrics are just several examples for this group.

- **Team-based features**: All features based on team strategy, management, and adaptation. These features usually are hidden for a naive observer, and they are tough to design, and craft as well as implement. Communication efficiency of a team, role assignment on a team, round win percentage and execution pace, team distribution on a map are just some illustrative instances for this group.
- **Player-environment interactions**: Includes all player interaction with the match map. Trajectory analysis of players, distance covered by a player are just some examples for it.

In the appendix, a survey to elaborate feature types and craft new feature ideas are placed appendix A. Moreover, we have provided all possible feature ideas and discussion about possible features in appendix B.

4.3 Validation of labels

Before diving into the data processing part, we need to ensure that the labels calculated by TrueSkill are logical and valid. The validation of labels is especially a challenging task because there is no ground truth labels for player skill assessment that we can validate the set of crafted labels. Therefore, we need to find an alternative way to validate our data labels. We have concluded several parameters to sanity check of data labels by focusing on individual player and teams.
The most intuitive way to check labels is to create a leaderboard of players by using latest mu values of each player and check whether the ranking order is somehow similar to a matchmaking server leaderboard. The first problem in this approach is that we have mixed types of players ranging from absolute amateurs to top-ranked professional players from different matchmaking servers. Because of the type difference among players, these players have different communities to organize CS:GO matches, and they are playing different types of players similar to their levels. Therefore, the TrueSkill algorithm evaluates players based on the match results with their community. It leads an unfair benchmark on a possible leaderboard because an amateur player can play hundreds of matches with other amateur players and being ranked better than a professional player just because of community difference. This problem even exists on a particular subdomain on the dataset like a selection of records on only one matchmaking server such as HLTV. For a further discussion of the problem, please check the chapter 5.
The second problem is that the TrueSkill algorithm evaluates players based on the result of a match rather than the individual statistics of a player observed during the match. It is logical to choose such a way to evaluate players in the project; otherwise, skill assessment based on player statistics would invalidate our mission, namely discovering the relation between player statistics and skill level of players. Contrary to our approach of player skill assessment, matchmaking servers utilize basic player statistics such as kill, death, and assist counts to appraise a player skill, which causes further mismatching of leaderboards. Despite the mentioned problems, we have observed a good correlation

between leaderboard created by us and the overall leaderboard of HLTV by picking on HLTV related records.
After defining the problems, we have chosen two different approaches to provide a sanity check of labels. The first one is observing the mu pattern of selected players to conclude whether the pattern makes sense. The conditions of a logical pattern, in this case, are that an increase on mu value is expected on a match win, and a decrease on a match lose (The TrueSkill ranking of a player may decrease on a match win depending on the opponent team; however, it is a rare case). Another condition is to notice a smooth pattern (small changes on mu value) on small sigma values (remember that sigma is the indication of in which degree the algorithm is precise for a player skill). To support the idea, figure 4.1 has been provided as a visual guide. As can be seen in the figure, after a number of n matches (player matches already filtered with 30 matches, which means first 30 matches have been discharged), a smooth pattern with a small deviation has been observed. Moreover, a small increase for a won match and a small decrease for a lost match on mu value has been inspected as well.

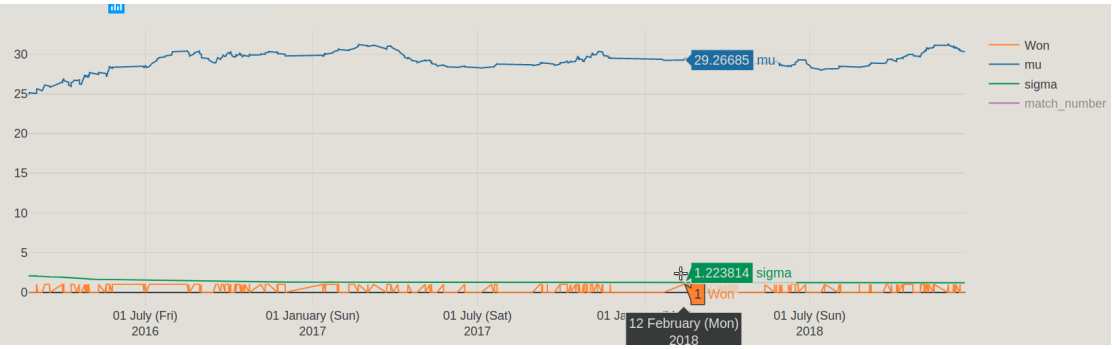


Figure 4.1: Plotting of time series for a player (pashaBiceps).

The second indicator of sanity check is to observe the pattern of professional team members. We are anticipating a strong correlation between skill patterns of team members and nearly similar mu values (The mu value would not be the same among team members because a team member can participate in a team later than others, and because of substitutes, a team member may not participate in a team squad for all matches). Figure 4.2 has been provided as a visual guide. The trend of mu values for team members are going the same after the player participated in the team.

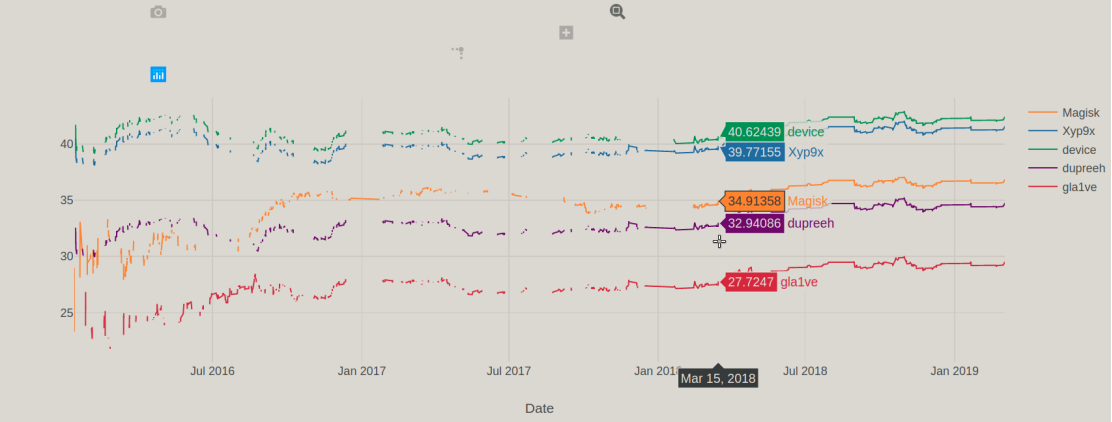


Figure 4.2: Plotting team members' mu values (Astralis).

4.4 Initial experiment: assessment on match results

The main purpose of initial experiments was providing a baseline accuracy while testing various learning models rapidly. In this experiment, we have used results of matches to update player skills by TrueSkill. Therefore, at the end of each match, we have fetched the most recent skill records of players involved in the match and updated these skill records by considering the outcome of the match.

4.4.1 Dataset properties and features

First of all, we start by providing information about the amount of data for the first experiment. In figure 4.3, the amount of data after the pre-processing step of raw data (such as filtering invalid records, imputation of raw data) is given. As suggested in the figure, the records have been filtered with a threshold of the minimum number of matches per player in order to guarantee nearly convergence of TrueSkill (see the discussion of convergence property of TrueSkill on the previous chapter). The pink features are index features, used to identify each player record, and not contribute to the learning process. The green ones are common features for both stats files and demo files. Not all features have been colored with green because, as we already mentioned, there is a data heterogeneity because of different feature sets for different types of datasets (stat files and analyzed demo files). Finally, the black ones are the features that do not belong to either category of previously mentioned. When we examine the number of data points in the figure, there are not much data points to accommodate a complex learning task such as a deep neural network. There are two reasons for the shortage of data. First, as described before, the source of data is limited caused by the fact that the effort of data collection is quite high at the time the project implemented. Second, the convergence threshold is relatively high (30) for the sake of conducting reliable data labels.
The initial feature set and explanation of each feature are given below. They are categorized to boost comprehensiveness of features and code reference of features has also

been provided on the title of each category for those who want to check the implementation of features.

Identifier features

- **Name (string)**: Name of the player involved in the match.
- **Date (string)**: Date of the match played. It is stored in timestamp format.

Skill based features (player.go)

- **HS percentage (float)**: Percentage of headshot (HS) kills done by the player divided by the total kill count.
- **ADR (float)**: Average damage per round given by the player.
- **FPR (float)**: Average number of kills done by the player per round.
- **APR (float)**: Average number of assists done by the player per round.
- **KD Dif. Round (float)**: Kill - Death count difference of the player averaged by round.
- **Flash Assist Round (float)**: Average number of flash assists done by the player per round.
- **KAST (float)**: Total number of kills, assists, survives and trades done by the player per round.
- **Cluthes Won (float)**: Average number of cluthes[1] done by the player per round.
- **Blind Players Killed Round (float)**: Average number of blind players killed by the player per round.
- **Blind Kills (float)**: Average number of players killed per round while the killer (player) was blinded by an opponent.
- **Grenade Damage Round (float)**: Total grenade damage given to the opponent team by the player averaged per round.
- **Fire Damage Round (float)**: Total fire damage given opponent team by the player averaged per round.
- **Time Flashing Opponent Round (float)**: Total amount of seconds opponents got blinded by the player averaged per round.

[1]Link: https://www.quora.com/What-is-the-meaning-of-clutch-and-ace-in-CS-GO

- **Accuracy (float)**: Accuracy of the player while hitting an opponent player. It is calculated by dividing total shots hit to an opponent player to total shots done by the player.
- **Number of times trader (float)**: Number of times the player was in a trader position averaged per round.
- **Number of times Tradee (float)**: Number of times the player was in a tradee position averaged per round.

Team based features (player.go)

- **Won (integer)**: Outcome of the match for the team the player involved. It is a binary number either 0 or 1.
- **Pistol Round Won Percentage (float)**: Percentage of winning pistol rounds among all rounds.

Crafted feature

- **match_number (int)**: For the filtering records, the dataset has been grouped by the name of players, and then each row has been numerated with an integer number. **match_number** is the number of the row for each data point.

Target features

- **Mu (float)**: TrueSkill mu value of the player after the match played.
- **Sigma (float)**: TrueSkill sigma value of the player after the match played. Not used for the learning task.

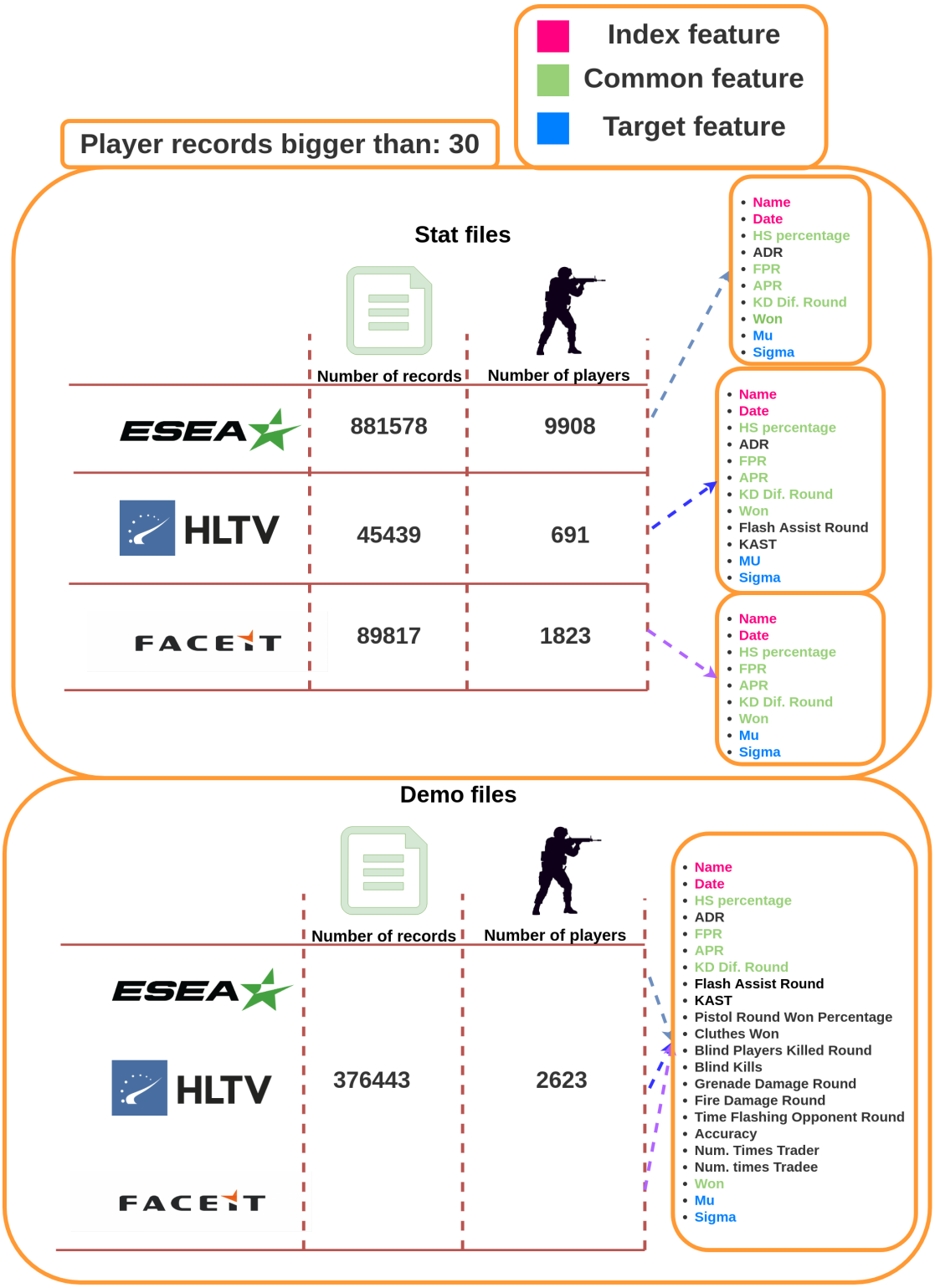


Figure 4.3: Initial data statistics.

4.4.2 Data pre-processing

Data exploration and visualization

The initial step of a standard data pipeline is visualizing the different aspects of the dataset in order to visually examine the correctness of data and explore the hidden properties of the dataset. The first figure provided in this section is the figure 4.4 visualizing the distribution of mu value, which is the target label will be used in a supervised learning task, with the count of players. As can be seen in the figure, the distribution of mu values is like standard distribution with a mean around 28. 28 is a near number to default mu value (25) however, it is differentiated a bit, which is a good sign of correctness of data labeling algorithm (TrueSkill labeling). If we would have more data, then we would expect a more balanced distribution with longer tails on both sides.
The second figure 4.5 represents the correlation coefficients of all possible pair of features in a heatmap format. In the figure, white color represents the highest correlation, while black represents no correlation. When we scrutinize the map, we can see that many features do not correlate with others, that is because we tried to eliminate the correlated features during the feature crafting in analyzer design. Highly correlated features may cause many problems, we called this Multicollinearity. Essentially, Multicollinearity is an observation on multi-regression models that one predictor feature can be deduced from a linear combination of other features with decent accuracy. This situation mainly causes the numerical instability meaning small changes in input data may lead to massive changes in the regression model, even parameters of the model, which named high variance in the model. High variance is one source of generalization error on inference step, and it can drastically reduce the prediction performance of a model on real observations. Other models immune to Multicollinearity by default such as decision forest; feature correlation may cause to hide beneficial interactions and combinations between useful features, and the feature redundancy may misguide the model for the learning task. Therefore, it is beneficial to have a unique and uncorrelated feature set for any types of learning models. When we reconsider figure 4.5, we can say that there is a strong correlation between win status of a match and player statistics (especially KAST, K-D DIF., FPR, ADR). This outcome makes sense in general because the players of the winner team have a higher probability of making better statistics compared to the loser team players. One interesting point is that FPR (killed player per round) has a robust correlation with ADR (average damage per round), which also make sense that the change of giving severe damage to opponent players is higher in general for a better fragger. Another aspect of the correlation graph is that KAST value has strong-mild correlations with FPR and APR, that is because KAST value includes the number of players killed as well as the number of assists done by a player by definition.
The next figure 4.6 is about the distribution of each feature with respect to the target outcome of the learning task, namely mu. For each feature, on the left-hand side, the feature distribution with respect to mu value is given, and on the right-hand side, the histogram of each feature with a smoothed probability distribution function is given (that

is why y-axis value can exceed 1). Moreover, the normal distribution has been fitted for each histogram in order to observe how well a normal distribution model fits for each distinct feature (black line on the right plot). Visually controlling the distribution of each feature can provide information about data anomalies, central tendencies, and populated areas as well as transformation techniques to get a nearly normal distribution. Transforming distribution of an individual feature to normal distribution can provide many benefits including utilizing normality assumption, which many learning models can benefit from with different degrees, and Homoscedasticity. Homoscedasticity is a situation where the variance of the error term is the same across all possible values of an independent variable. It is one of the main assumptions of reliability of regression model as well.

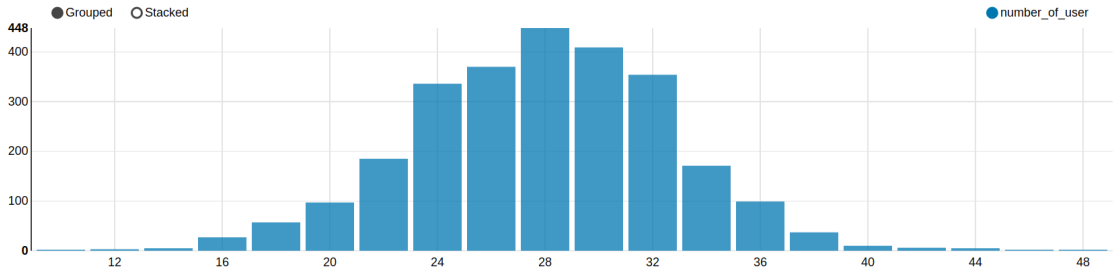


Figure 4.4: Distribution of mu respect to user numbers.

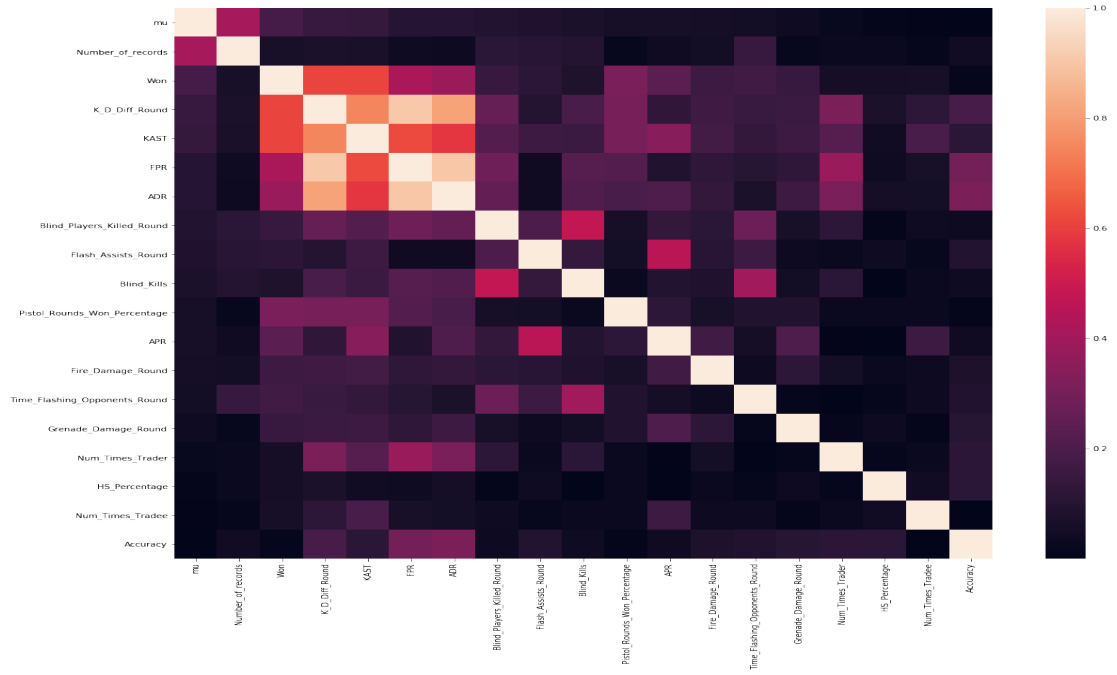
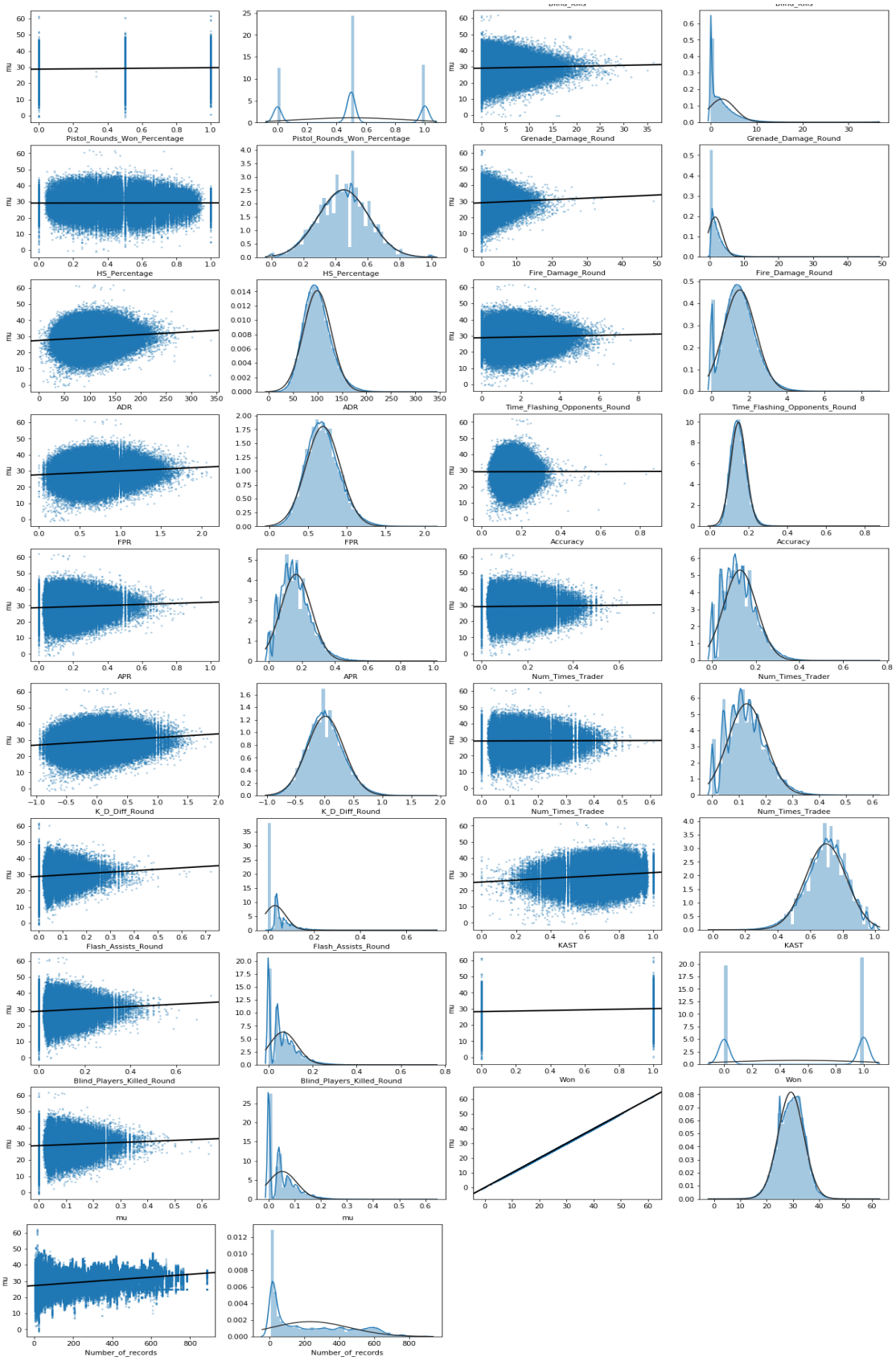


Figure 4.5: Feature correlation heatmap.


mu
Pistol_Rounds_Won_Percentage
Pistol_Rounds_Won_Percentage
Grenade_Damage_Round
Grenade_Damage_Round
HS_Percentage
HS_Percentage
Fire_Damage_Round
Fire_Damage_Round
ADR
ADR
Time_Flashing_Opponents_Round
Time_Flashing_Opponents_Round
FPR
FPR
Accuracy
Accuracy
APR
APR
Num_Times_Trader
Num_Times_Trader
K_D_Diff_Round
K_D_Diff_Round
Num_Times_Tradee
Num_Times_Tradee
Flash_Assists_Round
Flash_Assists_Round
KAST
KAST
Blind_Players_Killed_Round
Blind_Players_Killed_Round
Won
Won
mu
mu
Number_of_records
Number_of_records

Data cleaning and imputation

After checking the attributes of the dataset and visualize the individual features, we need to pre-process the raw data to leverage the quality of it by using several techniques, including data cleaning and missing value imputation. For the data cleaning, the main problem stems from stats files collected from matchmaking websites. There are many rows which have non-numeric values or directly blank fields which make it impossible for many learning models to work with it. For alleviating the problem, we have dismissed those rows with invalid fields if the number of rows relatively low with respect to the whole dataset like percentage of %1-2. If there are many rows suffer from missing/invalid fields, we have marked those fields with NaN (Not a Number) identifiers and imputed these NaN fields using several imputation algorithms ranging from basic imputation techniques such as mean or median of a feature or advanced ones like the techniques based on KNN, or SVD. After benchmarking several imputation models by considering the square loss difference between the real value and imputed value, and execution speed of an algorithm, we decided to use Matrix Factorization [18]. In imputation case, basically, the algorithm takes rows and columns of the dataset like a matrix with missing values and tries to reconstruct the matrix using two low-rank matrices namely U and V, with an L1 sparsity penalty on the elements of U and an L2 penalty on the elements of V using a gradient descent algorithm. After factorization into two matrices, imputation is done for missing fields, since we can build the completed main matrix. For the analyzed demo files, because the analyzer has been tested extensively, the output of analyzer is complete and free of synthetic error at this stage (there are several issues for finding statistics overflowed or underflowed like flash assist; however, the problems mainly emerges from missing events in demo files or bugs in the parser).
It is important to note that the most basic cleaning of the raw data was done by filtering players with the number of matches smaller than the threshold of convergence (30). We have tested the situations of before and after the filtering step, we observed a drastic decrease in test loss. After passing all steps mentioned above, we have standardized the data for eliminating measurement scale effect of individual features and suppressing outliers by using sklearn standard scaler. The result for each dataset has been stored on table format in a CSV file.

4.4.3 Data modeling

After pre-processing data sets, now we have tried several learning models to represent the relation between in-game statistics of players and their skill levels. Fundamentally, we have tried most frequently used learning models for regression task ranging from linear regression to a simple neural network. In the benchmark, two different regression metrics have been used, namely R2 score and MSE (Mean Square Error) to rank algorithms. R2 score is a metric indicating the proportion of variance in the target variable can be deducible from the independent variable(s). It is in between 0 and 1 (can be negative in rare cases) and usually higher is better meaning better explainability of the variance of

outcome variable for a model. MSE is also another common regression metric indicates averaged sum of the squared difference between the observed values and predicted values by a model (residuals). It always provides positive values, and lower is better.
In table 4.1, two categories of error measures have been given with the metrics described in the previous paragraph. For measuring test error, a proportion of data has been split as a test dataset randomly, and test error has been calculated and noted. For training error, cross-validation score with standard deviation among different folds has been noted as a second category (presented as validation error).
For each algorithm, the best parameters set have been selected using random parameter search with cross-validation. All models are already implemented on the sklearn library, and they are directly used with optimized parameter set obtained from the result of the parameter search. Apart from linear regression noted in the table, regularized regression algorithms (ridge and lasso) have also been executed, however, because regression models are too simple to get an overtraining from data, they are not providing better results compared to linear regression. For elastic net (a parametric combination of ridge and lasso regularization), second-order polynomial features have been added to model nonlinear interactions between features (we are unable to train a model on higher degree polynomial features because of longer training time). We have also tested XGBoost learning model which is defined in the documentation like "an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable"[2]. For the neural network, a simple two layers feedforward neural network has been used to check a NN baseline. Confirming our expectation, because of limited data and feature set, the NN cannot outperform the best models at this stage. In the end, Random Forest was the best model with minimum MSE and maximum R2 score.
At the last figure 4.7 for this section, several properties of each model have been illustrated as graphs. On the left side, predicted mu values and observed mu values on the test set had been plotted in order to check correlation. For the ideal case, there would be a linear line indicating the highest correlation. In the middle, the plot of residuals versus predicted values of the target variable for a model is provided. This plot is particularly useful to check the assumption of Homoscedasticity visually. On the right graph, the histogram of residuals is provided with outliers bigger than three standard deviations. The histogram of residuals is useful to check the Normality assumption of a model. On ideal case, a normal distribution (symmetric bell-shaped histogram which is evenly distributed around zero) is expected.

[2]Link: https://xgboost.readthedocs.io/en/latest/

Algorithms	Validation Error (Stdev)		Test Error	
	R2	**MSE**	**R2**	**MSE**
Linear Regression	0.14(0.0008)	17.7(0.08)	0.14	17.6
Elastic Net (Poly. features = 2)	0.14(0.002)	17.5(0.034)	0.15	17.5
KNN	0.12(0.0017)	20.8(0.093)	0.12	20.7
Gradient Boosting Regression	0.28(0.002)	17.0(0.174)	0.28	16.8
Random Forest	**0.4(0.001)**	**12.5(0.09)**	**0.39**	**12.3**
XGBoost	0.36(0.0021	13.1(0.074)	0.37	12.9
Neural Network	-	19.2	-	18.5

Table 4.1: Several learning methods comparison

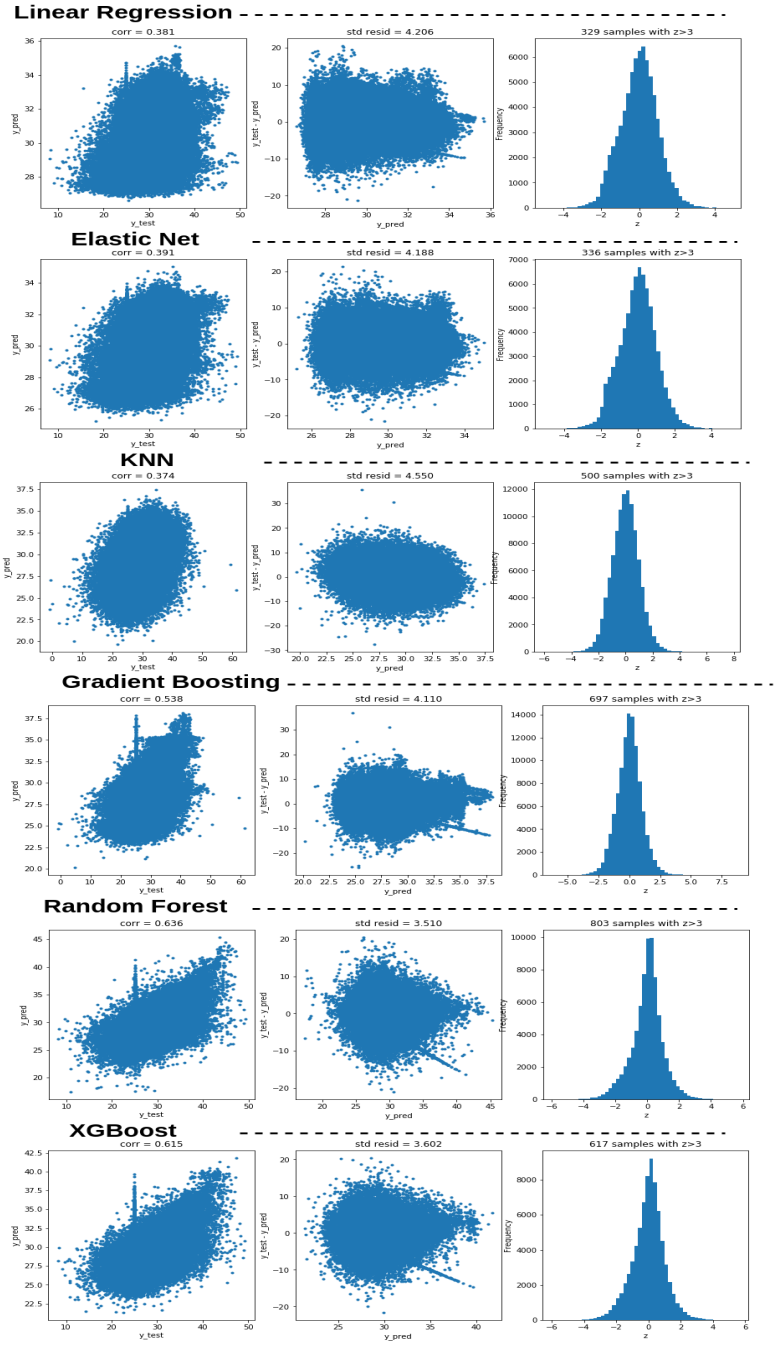


Figure 4.7: Learning model graphs.

4.5 The second experiment: assessment on round results

On the first experiment, we have prototyped fastly simple data pipeline for learning. For this experiment, we deep dive into each step of this pipeline to extend our horizon about data. The main considerations for this experiment are presented below:

- In the first experiment, we have used a feature called `match_number`. This feature is created as follows. First, the records have been grouped by player name, and then for each group, records have been numerated from earliest date to latest, which this enumeration is named `match_number`. Then, this feature is used for filtering records smaller than a threshold (30) on data cleaning phase. As expected, the feature is highly correlated with our target variable (mu) because it is changing with the date of records like mu. We have discharged this feature for this experiment (we have expected a noticeable drop for predictive powers of models overall) because our purpose is to model the relationship between pure statistics of players directly extracted from played matches and player skills rather than boosting the predictive power of a model by adding a highly correlated feature with our target variable.
- In this experiment, contrary to the first experiment, which the match results have been used to update player skills, we use round results of matches to assess player skills. Our expectation for this decision is that because of a relatively higher number of rounds compared to matches; we expect to converge latent player skill better (with smaller sigma values).
- Contrary to the first experiment, apart from a basic pre-processing of the raw data, we use more advanced techniques for processing raw data before modeling the relationship in this phase. We believe that advanced processing pipeline can make a remarkable impact on the predictive power of a learning model, especially in our situation where data is limited.
- We have crafted much more involved features compared to the first experiment. We expected that newly crafted features increase the expressive power of dataset, which a learning model can benefit from.

4.5.1 Dataset properties and features

The properties of the dataset are the same as the previous iteration. The difference in non-filtered users and record counts between two experiments is that we have considered stats file rankings as well for each player while filtering player records (`match_number`). Because of a higher number of matches per user thanks to this change, the number of user and non-filtered data points was increased on demo files compared to the previous experiment. The second reason for the increase in the number of users for demo files is the correction of minor bugs that cause discontinuity of player match series on the database. Overall, the summary of the dataset is given in figure 4.8.

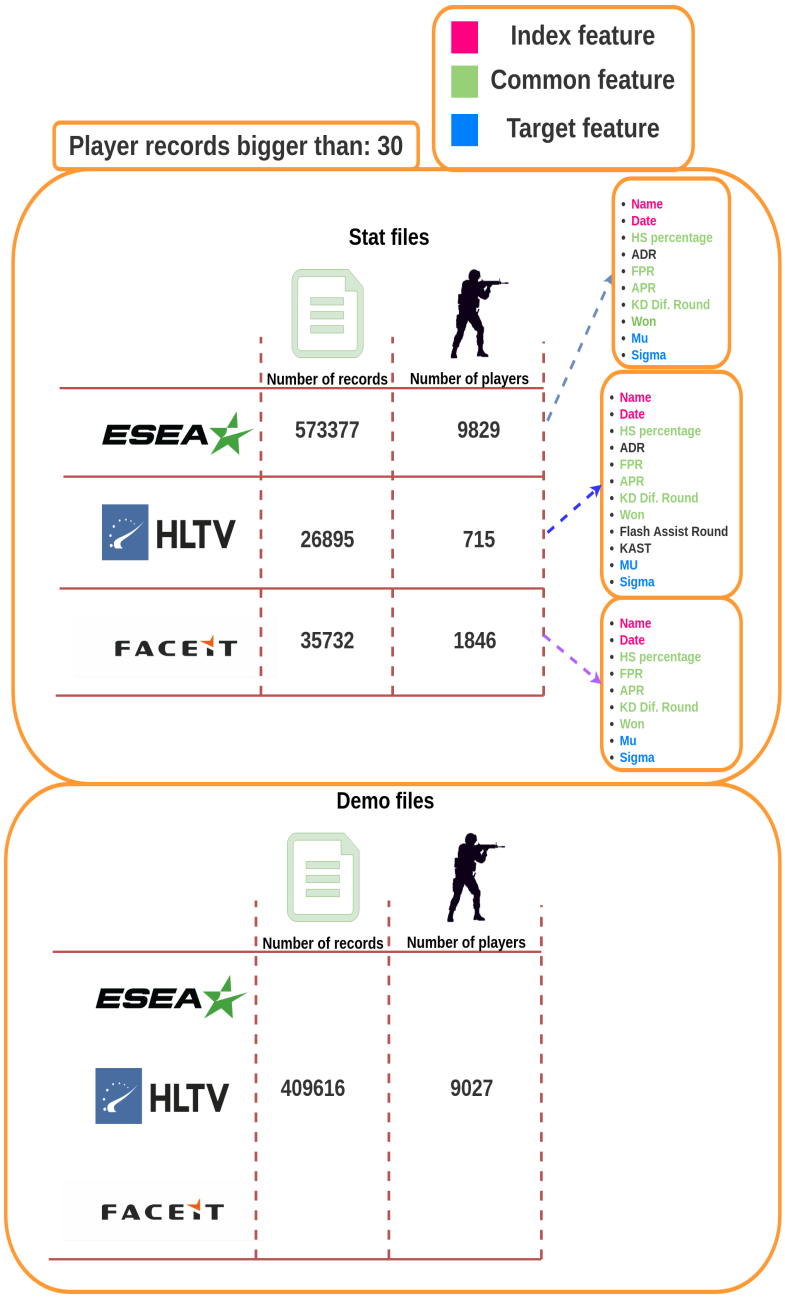


Figure 4.8: Summary of the dataset for this experiment.

For this iteration, new features in addition to initial ones have been provided below:

Economy features

- **moneySaved (float)**: Average amount of money saved by a player per round.
- **unitDamageCost (float)**: A feature to indicate how much does it cost for each

unit damage given by a player. It has been calculated in the formula 4.1:

$$\frac{\sum_{i=1}^{n} d_i * (eqP/eqO)}{totalDamage} \tag{4.1}$$

where n is the set of all damage amounts = $d_1, d_2, d_3.., d_i, .., d_n$, eqP is equipment value of the player giving damage at the moment of damage occur, eqO is equipment value of the opponent player receiving damage at the moment of damage occur, and totalDamage is the amount of total damage during the match by the player whom the damage is given by.

Skill based features

- **MVP (Most Valued Player) (float)**: Average number of MVPs a player has been selected. MVPs are selected on a round basis, and there are three reasons for selection of MVPs: killing most players in a round, planting the bomb and bomb exploded, and defusing the bomb.
- **sniperKill (float)**: Average number of kills done by a player using a sniper class weapon per round (check weapon classes here[3]).
- **meleeKill (float)**: Average number of kills done by a player using a melee class weapon per round.
- **shotgunKill (float)**: Average number of kills done by a player using a shotgun class weapon per round.
- **assultRKill (float)**: Average number of kills done by a player using an assault rifle class weapon per round.
- **pistolKill (float)**: Average number of kills done by a player using a pistol class weapon per round.
- **machineGunKill (float)**: Average number of kills done by a player using a machine gun class weapon per round.
- **smgKill (float)**: Average number of kills done by a player using an SMG class weapon per round.
- **headHit (float)**: Number of hits averaged by total hit done on the head part of the victim's body.
- **stomachHit (float)**: Number of hits averaged by total hit done on stomach part of the victim's body.

[3]Link: https://counterstrike.fandom.com/wiki/Weapons

- **chestHit (float)**: Number of hits averaged by total hit done on the chest part of the victim's body.
- **legsHit (float)**: Number of hits averaged by total hit done on legs part of the victim's body.
- **armsHit (float)**: Number of hits averaged by total hit done on arms part of the victim's body.
- **avarageKillDistance (float)**: Average euclidean distance between killer and his/her victim per kill. It is useful to observe how far a player usually gets a kill on average.
- **avaragePlayerSaved (float)**: Average number of team members saved per round. In the context, player saving means if a team member's health is below a certain threshold and a player killed the opponent player who hurt the player's team member.
- **playerWonHealth (float)**: Average health of a player per round after won a round.
- **playerLostHealth (float)**: Average health of a player per round after lost a round.
- **timeHurtToKill (float)**: Average amount of duration between a killer hurt the victim for the first time and killed the victim.
- **spraySniper (float)**: Total amount of spray averaged by kill count while a player killed an opponent one by using sniper class weapon.
- **sprayMachineGun (float)**: Total amount of spray averaged by kill count while a player killed an opponent one by using machine gun class weapon.
- **sprayShotgun (float)**: Total amount of spray averaged by kill count while a player killed an opponent one by using the shotgun class weapon.
- **spraySMG (float)**: Total amount of spray averaged by kill count while a player killed an opponent one by using SMG class weapon.
- **sprayARifle (float)**: Total amount of spray averaged by kill count while a player killed an opponent one by using assault rifle class weapon.
- **sprayPistol (float)**: Total amount of spray averaged by kill count while a player killed an opponent one by using pistol class weapon.
- **duckKill (float)**: Average number of kill done by a player while the player is ducking (camping).

- **memberDeathDistance (float)**: Average distance between a killed player and his/her team members while the player has been killed. It is a beneficial indicator to asses map distribution of team members while a team member has been killed.
- **sniperKilled (float)**: The number of snipers killed while the sniper has been zoomed (there are two zoom levels for sniper class weapons. The level of zoom is 2 in this feature).

Team based features

- **roundWinPercentage (float)**: The percentage of winning the round among all rounds.
- **roundWinTime (float)**: Average duration of won rounds.
- **occupiedArea (float)**: The amount of square meter each team occupied in a map averaged by round. The algorithm as follows:

  1. We are using NAV files[4] collected beforehand for each known map to get areas, places and connection points in a map. Originally, NAV files have been developed for pathfinding of AI bots in a map; however, we are using those files for the segmentation of places in a map[5]. Each place in a map is composed of several areas.
  2. A diameter threshold value has used for checking whether a player has occupied a place. If the player distance to place center (calculated as a weighted average of center of areas) is smaller than this diameter, and there is no opponent player (occupation rules given below) in the same place, we claim that the player has occupied the place. The diameter calculated as follows: Firstly, we are finding the boundary points of the place by iterating all areas in the place (furthest point of South, North, East, West). After that, we are calculating place width and height and dividing those dimensions with a constant > 1. Then we get the min value of the division results which named diameter in 2D. Lastly, we are reflecting this diameter to 3D space by using the angle between z-axis and 2D space. Therefore, we are using a separate diameter for each place by considering the dimensions of that place.
  3. The occupation rule as follows: If there is only one team in a place and the team player is near enough (consideration of place diameter and center of this place mentioned above), we assign the place occupation to this team. If there are two teams at the same time in a place (and both teams are near enough to the center of the place), we make the place unassigned, and the place will not be assigned to any team for this iteration of place checking. For each

[4]Link: https://developer.valvesoftware.com/wiki/Navigation_Meshes
[5]Link: https://github.com/mrazza/gonav

occupation, we also store the tick value of this occupation in order to check the conditions given below.

4. At the end of the match, we are calculating the total square meter occupied for each team and taking the average by round.

Match metadata feature

- **Mapname (string)**: Map name of the match played.

4.5.2 Data pre-processing

Data cleaning

For data cleaning, in addition to all steps explained in the first experiment, we have visualized a histogram of the count of players w.r.t the latest sigma values of each player after filtering the match records. To further improve the validity of target feature (mu), we have filtered all players, who have sigma values of the latest records bigger than a visually determined sigma threshold(4).
Contrary to the first experiment, we have much more features under suspicion whether each of them is indeed leveraging the prediction model performance or making it worse. Therefore, we need to process the raw feature set to boost representation power of data by conduction a comprehensive feature engineering step.

Feature elimination

To do so, first, we eliminate the features not contributing the learning model as expected by using an ensemble of several feature elimination methods. Before scrutinizing each selection algorithm, the overall feature selection table 4.9 has been provided below. In the table, True means a feature has been marked as a useful one by a feature selection algorithm, while False indicates the opposite situation.

Feature	recursive	boruta	genetic	spearman	pearson	mutual_info	f_regression	model	variance	Total
Unit_Damage_Cost	True	True	True	True	True	True	True	True	True	1
Time_Flashing_Opponents_Round	True	True	True	True	True	True	True	True	True	1
Spray_Sniper	True	True	True	True	True	True	True	True	True	1
Spray_Pistol	True	True	True	True	True	True	True	True	True	1
Spray_ARifle	True	True	True	True	True	True	True	True	True	1
Round_Wintime	True	True	True	True	True	True	True	True	True	1
Player_Won_Health_Round	True	True	True	True	True	True	True	True	True	1
Pistol_Kill_Round	True	True	True	True	True	True	True	True	True	1
Occupied_Area_Round	True	True	True	True	True	True	True	True	True	1
Money_Saved_Round	True	True	True	True	True	True	True	True	True	1
MVP	True	True	True	True	True	True	True	True	True	1
K_D_Diff_Round	True	True	True	True	True	True	True	True	True	1
KAST	True	True	True	True	True	True	True	True	True	1
Head_Hit	True	True	True	True	True	True	True	True	True	1
FPR	True	True	True	True	True	True	True	True	True	1
FKR	True	True	True	True	True	True	True	True	True	1
Duck_Kill	True	True	True	True	True	True	True	True	True	1
Chest_Hit	True	True	True	True	True	True	True	True	True	1
Av_Kill_Distance	True	True	True	True	True	True	True	True	True	1
AssultR_Kill_Round	True	True	True	True	True	True	True	True	True	1
APR	True	True	True	True	True	True	True	True	True	1
ADR	True	True	True	True	True	True	True	True	True	1
Member_Death_Distance_Round	True	True	True	True	True	False	True	True	True	0.969
Grenade_Damage_Round	True	True	True	True	True	False	True	True	True	0.969
Time_Hurt_To_Kill	True	True	False	True	True	True	True	True	True	0.874
Stomach_Hit	True	True	False	True	True	True	True	True	True	0.874
Sniper_Kill_Round	True	True	True	True	True	True	True	False	True	0.874
Round_Win_Percentage	True	True	False	True	True	True	True	True	True	0.874
Last_Member_Survived_Round	True	True	False	True	True	True	True	True	True	0.874
Fire_Damage_Round	True	True	False	True	True	True	True	True	True	0.874
Blind_Players_Killed_Round	True	True	False	True	True	True	True	True	True	0.874
Num_Times_Trader	True	True	False	True	True	False	True	True	True	0.843
SMG_Kill_Round	True	True	False	False	True	True	True	True	True	0.748
Pistol_Rounds_Won_Percentage	True	True	False	True	True	True	True	False	True	0.748
Flash_Assists_Round	True	True	True	True	True	True	True	False	False	0.748
Blind_Kills_Round	True	True	False	True	True	True	True	True	False	0.748
Arms_Hit	True	True	False	True	True	True	True	True	False	0.748
Legs_Hit	True	True	False	True	True	False	True	True	False	0.717
Sniper_Killed	True	True	False	True	True	True	True	False	False	0.623
HS_Percentage	True	True	False	True	False	True	False	True	True	0.623
Clutches_Won	True	True	False	True	True	True	True	False	False	0.623
Spray_SMG	True	True	True	False	False	False	False	True	True	0.591
Player_Lost_Health_Round	True	True	True	False	False	False	False	True	True	0.591
Accuracy	True	True	True	False	False	True	False	True	False	0.497
Num_Times_Tradee	True	True	False	False	False	False	False	True	True	0.465
Spray_Shotgun	True	False	True	False	False	True	False	False	True	0.409
Player_Saved_Round	True	True	True	False	False	False	False	False	False	0.34
Spray_Machinegun	False	False	True	False	False	False	False	False	True	0.252
Shotgun_Kill_Round	True	False	True	False	False	False	False	False	False	0.252
Melee_Kill_Round	True	False	False	False	False	True	False	False	False	0.157
MachineGun_Kill_Round	True	False	False	False	False	False	False	False	False	0.126

Figure 4.9: Feature selection result table.

On the first two columns, Pearson and Spearman correlation coefficients (column of **pearson** and **spearman**) have been used to measure the correlation of each feature with the target feature(mu). After calculation of correlation coefficients, smallest n(10) features with the lowest correlations for the target variable, have been eliminated.
Feature selection based on mutual information (column of **mutual_info**) uses Mutual Information of each feature w.r.t target feature. Mutual information is the measurement of dependency of two variables. If two variables are independent, MI is zero: otherwise, bigger than zero. Feature selection based on MI is trying to consider maximum relevance(max MI) of a feature w.r.t target feature while guaranteeing minimum redundancy (minimum MI with rest of the features) [19].
Feature selection based on f_regression score (column of **f_regression**) is an univariate linear regression tests algorithm. In the first step, correlation of each feature w.r.t target feature has been calculated, and then correlation values have been converted to F-score

and then p-score to filter redundant features(features with lowest F-score value).
Recursive feature elimination method(column of **recursive**) is an algorithm in sklearn library that recursively eliminates the least essential features until reaching the desired number of features.
Feature selection based on a learning model (column of **model**) is another algorithm that selects the desired number of features based on importance or coefficient of features in the model. For this instance, the random forest model has been used, which has an attribute for feature importance.
Boruta feature selection algorithm (column of **boruta**) is another wrapper selection algorithm based on an iterative selection of original features. The algorithm iteratively builds randomly shuffled features (shadow features) and compare original feature importances with the maximum feature importance of shadow features (by comparing z-score of each feature. Feature importance usually gathered from a tree-based model by considering Mean Decrease Accuracy or Mean Decrease Impurity). If original feature importance is higher than this value, we recorded the original feature as a hit to a vector for this iteration. After a certain number of iteration, n features with highest hits are reported back to the user as selected features [20].
Variance selection algorithm (column of **variance**) assumes that feature with lower variance is redundant, and therefore, it should be eliminated.
Another algorithm is genetic selection algorithm (column of **genetic**). For genetic selection, subsets of random features are selected and evaluated by an objective function, only the best-fitted candidate feature subsets survives, which the next generation will be formed from (with random mutations and recombinations), for each generation. After a predefined number of generation, survived features is reported back as selected features [21].
Considering the table 4.9, we have decided to eliminate the least important 7 features based on our domain knowledge and dataset familiarity, and the aggregation of selection algorithm results. After elimination, the demo_mapname column, which has string type, has been encoded by using a one-hot encoder. Because we have observed Heteroscedasticity [6] on the first experiment when analyzing residual graphs of predictors, we have applied a type of power transformation, Yeo–Johnson transformation [22], to independent variables (input features) to decrease the skewness of them (making more normally shaped distributions). Contrary to Box-Cox transformation, which only works for positive numbers, Yeo-Johnson transformation is an extension of it working with negative numbers as well. After all the steps, the correlation heatmap of the remaining features is given in figure 4.10. In the figure 4.11, the distribution of target variable w.r.t. player number is presented. Moreover, the figure 4.12 shows the distribution of each dependent variable and response graph w.r.t mu after applying power transformation.

[6] https://en.wikipedia.org/wiki/Heteroscedasticity

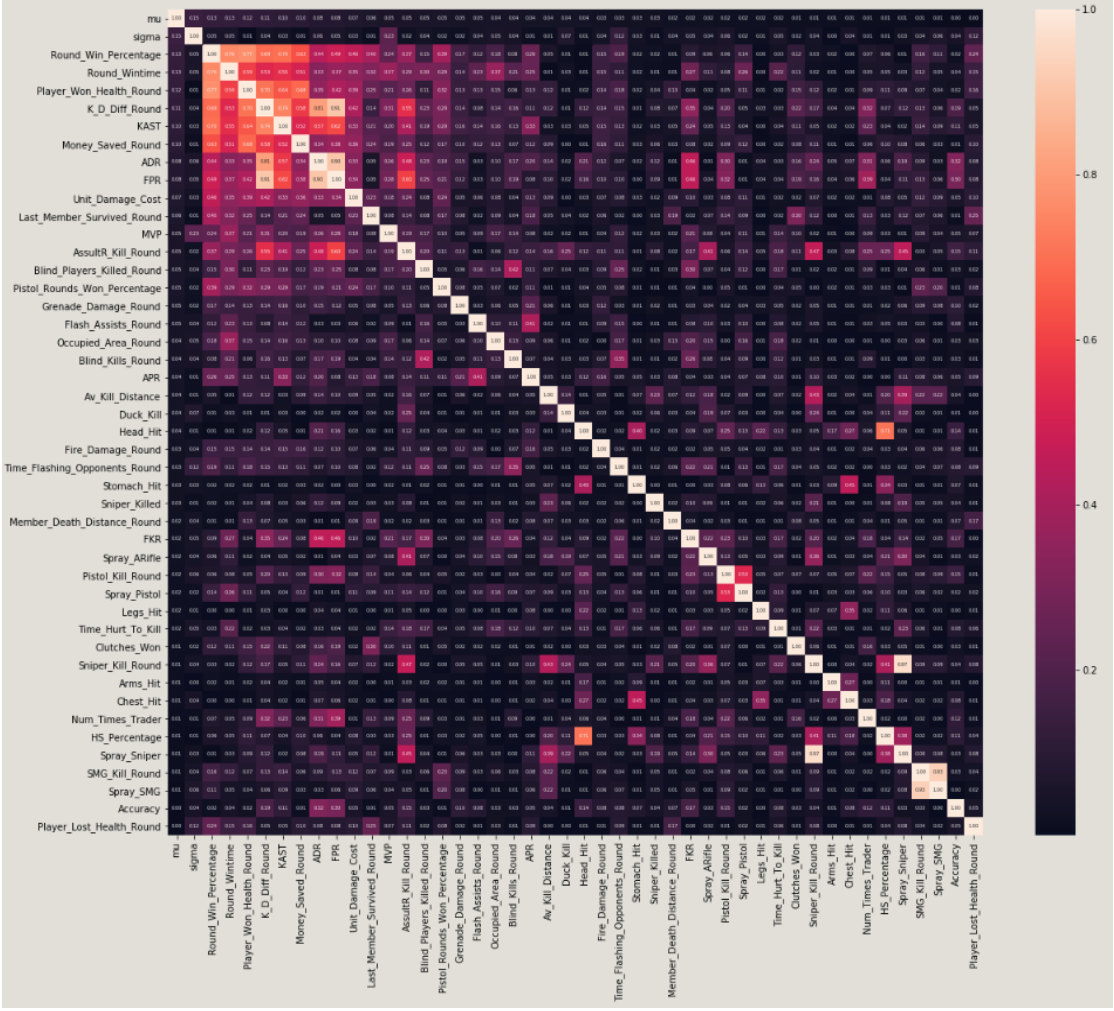


Figure 4.10: Feature correlation heatmap.

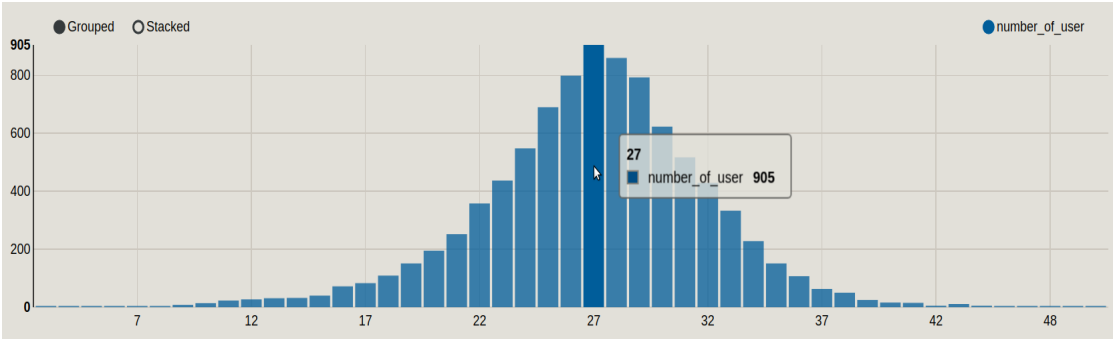


Figure 4.11: Target variable distribution w.r.t player number.

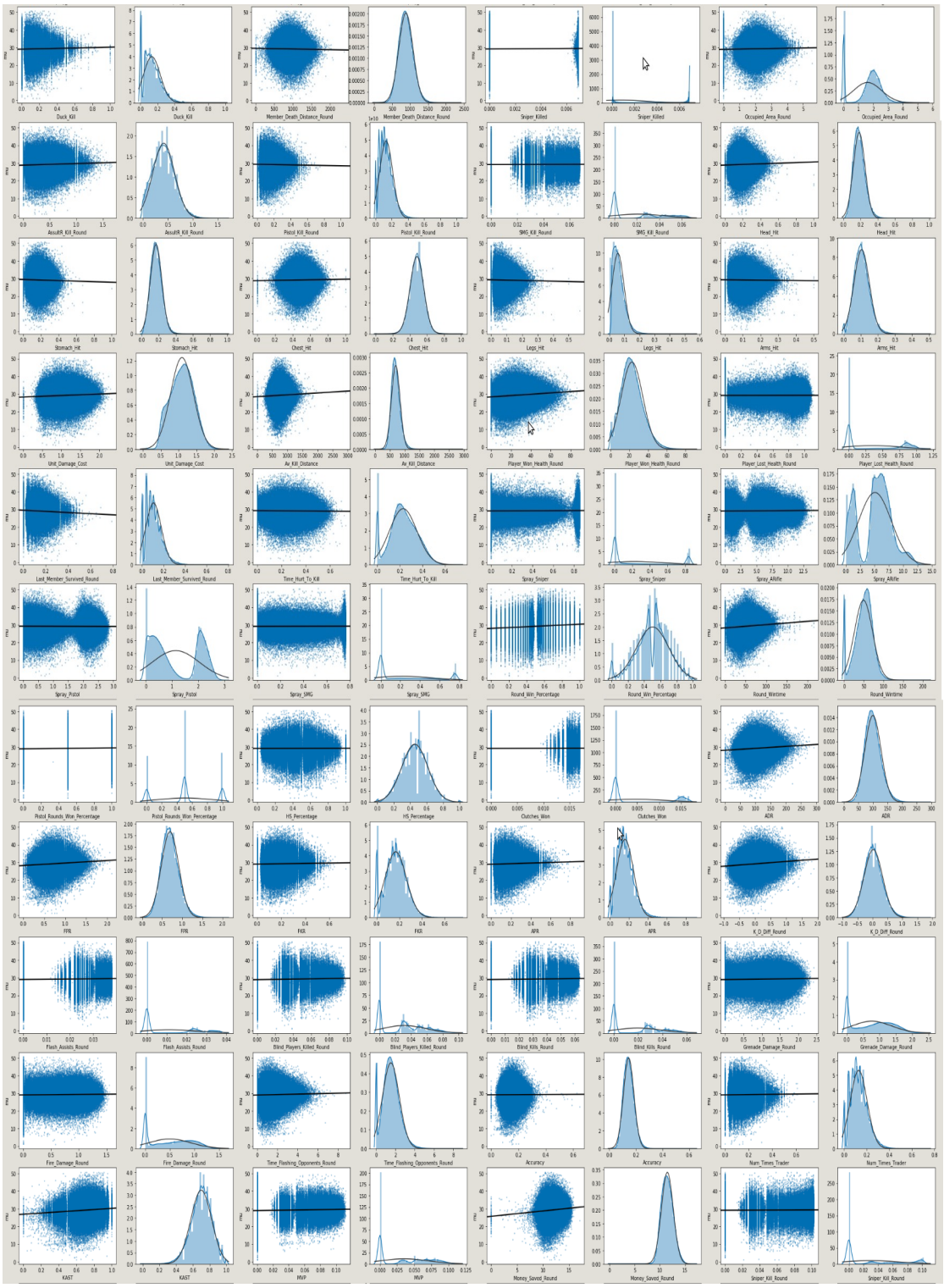


Figure 4.12: Feature distribution

Feature extraction

After filtering redundant features, due to the relatively high dimensionality of the data, we have decided to give a try to reduce the dimensionality by transforming features into another latent space. Possible theoretical benefits of dimensionality reduction are:

- Less use of in-memory space, and shorter training time for learning models.
- Boosting the performance of learning models trouble with the curse of dimensionality.
- Reducing the inter-correlation of features by transforming features on a less redundant latent space (caring of Multicollinearity).

The first task is that we need to find a good optimization point for dimension of new feature space. Our task is reduction of dimensionality as much while still getting a good prediction results. For this, we have tested several dimensions using PCA transformation and observe R2 score of random forest regressor. The results is given in the figure 4.13. 35 is selected for the number of dimension in latent space to craft new features after scrutinizing the graph.

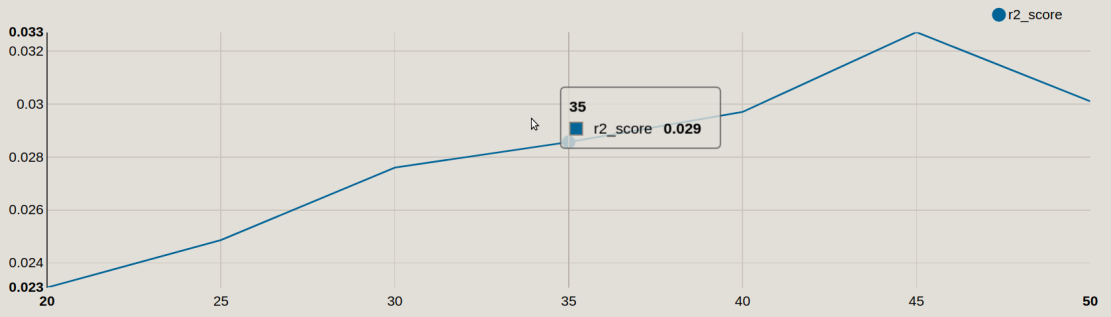


Figure 4.13: Number of dimension vs R2 score of regressor.

For dimensionality reduction, we have experimented several algorithms. For comparison of these reduction models, we have proposed two metrics namely Reconstruction error and R2 score. Reconstruction error is the mean square error between original input and reconstructed input after transformation. For R2 score, firstly, we transformed to original feature space into a lower-dimensional space (number of dimensions is the same for all models) and trained a random forest regressor on this new feature space. Then, we have tested this regressor by scoring a transformed test set, and the results have been provided in table 4.2.
The first algorithm for feature extraction is PCA. PCA is an orthogonal transformation algorithm that decomposes the data matrix into a relatively lower-dimensional feature space by keeping the essential orthogonal basis of this feature space (also called principal components, which has the highest eigenvalues). By eliminating the least important components (components related to lowest eigenvalues), we are maximizing the variance of data while eliminating the least important original features and noise. While PCA is a

Algorithm	Reconstruction Error	R2
No transformation	-	0.0339
PCA	0.084	0.0284
UMAP	-	-0.0142
Vanilla Autoencoder	3.94	0.028
Denoising Autoencoder	**0.558**	**0.0295**
Variational Autoencoder	-	0.0057

Table 4.2: Several dimensionality reduction methods comparison.

linear feature extraction algorithm, we were unable to test Kernel PCA (the nonlinear version of PCA) because of memory issues on execution (but Auto-encoder can be seen as a nonlinear version of PCA).
The second algorithm is UMAP (Uniform Manifold Approximation and Projection for Dimension Reduction) [23], which is a nonlinear dimensionality reduction algorithm that combines manifold learning techniques and topological data analysis. It first builds a fuzzy topological representation of given dataset and then it embeds this topological space into a lower-dimensional space thanks to manifold learning as close as possible to the original representation.
Another algorithm used for feature extraction is Auto-encoder. Basically, Auto-encoder is a type of neural network that tries to learn data coding by using the input data itself(unsupervised). It consists of two components, namely encoder, and decoder. Encoder part is learning the transformation from input space to embedded space, which has fewer dimensions compared to input space while decoder part is learning the reconstruction of data from embedding space to original space to be as close to the original input as possible. After training the whole network, the encoder part has been extracted, and the original data has been transformed to this embedded space. Three variants of auto-encoder have been used in the project.
The first variant of Auto-encoders is Vanilla Auto-encoder, which has the same attributes and algorithmic principles as the definition above, that tries to reconstruct the original data while minimizing reconstruction error. The second variant of Auto-encoder is Denoising Auto-encoder. Denoising Auto-encoder tries to reconstruct the original data from corrupted (noise added) version of it by optimizing reconstruction error. The main benefit compared to Vanilla Auto-encoder is that added noise partially eliminates the risk of learning an identity function (because of the same input and output on an Autoencoder), which is a redundant transformation for feature extraction.
Variational Auto-encoder is another type of them mainly categorized as a generative neural network because it learns a set of parameters to model continuous latent space, which is easy to sample and generate similar outputs with input data. By sampling from this latent space using learned parameters, decoder reconstructs the dataset by using the latent representation of dataset while minimizing reconstruction loss as well as KL divergence loss. While reconstruction loss ensures that decoder reconstructs the output

similar to input set, KL divergence loss ensures that the sampling process is not random (packing sample distributions of latent components to the center of the latent space). Because we got the best score from Denoising Autoencoder, we have extracted latent features using this model and we have used these features for training all models,

4.5.3 Data modeling

We have used nearly same learning models with the first experiment. Results is given in table 4.3.

Algorithms	**Validation Error (Stdev)**		**Test Error**	
	R2	**MSE**	**R2**	**MSE**
Linear Regression	0.024(0.0013)	13.49(0.112)	0.021	13.4
Elastic Net	0.025(0.0013)	13.48(0.108)	0.0219	13.4
KNN	**0.061(0.0036)**	**14.68(0.0609)**	**0.06**	**14.53**
Gradient Boosting Regression	0.0236(0.0007)	13.5(0.0866)	0.0191	13.44
Random Forest	0.041(0.002)	13.77(0.113)	0.021	13.67
XGBoost	0.024(0.001)	13.49(0.105)	0.02	13.42
Neural Network	0.0185	13.44	0.0227	13.48

Table 4.3: Several learning methods comparison.

The neural network model is a simple 5 layers feedforward neural network in this configuration. In addition to train the network from scratch, we tried to pre-train the network by using stats files of HLTV. In this process, first, we have trained a network with stats files of HLTV, and then we have transferred the weights of layers layer-wise to the weights of another network the same structure with the first network with different input layer (because feature set of demo files is different from the set of stats files). However, it does not boost the accuracy of NN as expected.
Moreover, we have also run a H2O[7] experiment, which is an automated machine learning framework experimenting with different learning models, and results are given in the figure 4.14.

[7]Link: http://docs.h2o.ai/h2o/latest-stable/h2o-docs/index.html

	model_id	mean_residual_deviance	rmse	mse	mae	rmsle
0	StackedEnsemble_AllModels_AutoML_20190809_100157	13.407741378219749	3.661658282557201	13.407741378219749	2.779293433294397	0.12865325940599398
1	StackedEnsemble_BestOfFamily_AutoML_20190809_100157	13.410685439776739	3.662060272548328 7	13.410685439776739	2.778929342194757	0.12891372898122622
2	XGBoost_grid_1_AutoML_20190809_100157_model_2	13.425356918837549	3.6640628977731193	13.425356918837549	2.7819448549195527	0.12884510948569222
3	GBM_1_AutoML_20190809_100157	13.437144739888097	3.665671117256443	13.437144739888097	2.7827399019628425	0.1289786140441524
4	GBM_2_AutoML_20190809_100157	13.437787426158506	3.6657587790467754	13.437787426158506	2.7827406462456055	0.1289809964532407
5	XGBoost_3_AutoML_20190809_100157	13.438205404063671	3.6658157897067976	13.438205404063671	2.7828176524920187	0.12894500371700615
6	XGBoost_grid_1_AutoML_20190809_100157_model_4	13.442817359742419	3.6664447847666297	13.442817359742419	2.7832915375498057	0.1289692057343006
7	XGBoost_grid_1_AutoML_20190809_100157_model_1	13.442916420713706	3.6664582938734904	13.442916420713706	2.7833703096764824	0.1289700846042791
8	GBM_grid_1_AutoML_20190809_100157_model_19	13.443999456813557	3.666605986033072	13.443999456813557	2.7828217868542002	0.12901922047079872
9	GBM_3_AutoML_20190809_100157	13.448465623883502	3.667214968321806	13.448465623883502	2.784403040349589	0.12901741210475243
10	XGBoost_1_AutoML_20190809_100157	13.451118967501241	3.6675767159667214	13.451118967501241	2.7852175302074174	0.12897804441565305
11	GBM_grid_1_AutoML_20190809_100157_model_13	13.453437605225252	3.6678928017630574	13.453437605225252	2.783812994337106	0.12906692840673392
12	GLM_grid_1_AutoML_20190809_100157_model_1	13.45668736837366	3.668335776394203	13.45668736837366	2.782882951521753	0.12906427949613836
13	GBM_grid_1_AutoML_20190809_100157_model_9	13.457809264462265	3.66848868942815	13.457809264462265	2.7842284957167376	0.12909013665370014
14	GBM_4_AutoML_20190809_100157	13.45891477136834	3.668639362402407	13.45891477136834	2.7852772680016575	0.12906323796627514
15	GBM_5_AutoML_20190809_100157	13.465172314897337	3.6694921058502548	13.465172314897337	2.7871592937239518	0.12905879820229482
16	GBM_grid_1_AutoML_20190809_100157_model_18	13.474570199945571	3.6707724255183094	13.474570199945571	2.7860905864858623	0.12917668909444294
17	GBM_grid_1_AutoML_20190809_100157_model_12	13.48352596557439	3.671992097700428	13.48352596557439	2.787347081127257	0.12922094984630156
18	GBM_grid_1_AutoML_20190809_100157_model_2	13.486501077402306	3.6723971840478127	13.486501077402306	2.785600272879565	0.12923389469695684
19	GBM_grid_1_AutoML_20190809_100157_model_6	13.493776785555388	3.6733876443353197	13.493776785555388	2.7860893230428547	0.1292821840597562
20	DeepLearning_grid_1_AutoML_20190809_100157_model_1	13.505658432900114	3.675004548691078	13.505658432900114	2.7858254098832917	0.12963971683437417
21	XGBoost_grid_1_AutoML_20190809_100157_model_5	13.506457389621405	3.675113248543697	13.506457389621405	2.7899609658160935	0.12904529993082717
22	GBM_grid_1_AutoML_20190809_100157_model_14	13.509813408805869	3.675569807364005	13.509813408805869	2.7880393789005384	0.1293515433959477
23	GBM_grid_1_AutoML_20190809_100157_model_1	13.514536012090446	3.676212182680761	13.514536012090446	2.788236507336791	0.12937476830613423

Figure 4.14: H2O leaderboard results.

In the figure, different from our base models, stacked models are the ensemble (bagging, boosting or other ensemble methods) of several best models and GBM stands for gradient boosting models.

4.6 The third experiment: a small experiment on time series

In this experiment, we have experimented the sequential nature and the temporal dimension of the player records. Due to the time constraint, this experiment is a small one; therefore, there are no extensive pre-processing and visualization steps contrary to previous experiments. As already discussed, we have fetched and stored a number of records for each player ordered with time, which forms a time series per player. Therefore, we have multiple (one for each player) multivariate time series. Moreover, each time series have a different length (the records per player is varying) and sampling frequency (sampling date between any pair of records is not constant), which further complicates the problem of modeling these time series. We aim to build a single model to work with all of these multivariate time series with different lengths.
In the first phase, we have queried the demo file records only for HLTV. Therefore, we have built a homogeneous player set in terms of player matching and skill assessment. To model the temporal aspect of player records, we have decided to add new artificial features. We have added mean, median, min, and max of each feature introduced in the second experiment via rolling and expanding window approach. For rolling window approach, there is a window with a fixed-length n assessing (taking mean, median or any other statistic of data) previous n records from the current time and adding assessment result as the field of a new artificial feature. Rolling window approach has been depicted in the figure 4.15.

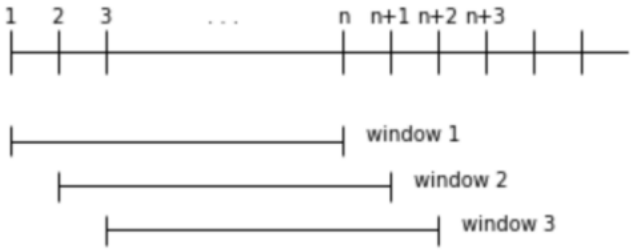


Figure 4.15: Illustration of rolling window method.

In addition to rolling window features, we have also added expanding window features. Expanding window is like a rolling window, but instead of a fixed-length window, it considers all available records until the current time. Same features (mean, median, max, min) have been calculated for expanding window as well.
After adding rolling and expanding window features, we have a few hundred features, including the original features as well. After standardization of data with sklearn standard scaler, we have randomly split the dataset into train and test partition, and we have trained few models to check how is the accuracy of the models. The results of two common and well-performed models have been given in the table 4.4.

Algorithms	**Validation Error (Stdev)**		**Test Error**	
	R2	**MSE**	**R2**	**MSE**
Random forest	0.882(0.0019)	1.34(0.0242)	0.898	1.16
XGBoost	0.908(0.0005)	1.04(0.009)	0.923	0.87

Table 4.4: Several learning methods comparison.

Chapter 5

Discussion

In this master thesis, we have implemented multiple architectures and pipelines to test the relationship between the player match statistics and player skills. In the first step, an infrastructure for processing the data has been implemented, including data collection, storage, feature augmentation, and label aggregation. In the second part, we have experimented with modeling the relationship by different configuration, features, data processing methods and learning models.

5.1 Evaluation of results

When we compare the first two experiments conducted in the master thesis, there is a major difference in the results of the predictive powers of models. The main cause of decreasing predictive power on the second experiment compared to the first one is the absence of any variable correlated with time. In the first experiment, match_number variable is an indirect indication of the temporal dimension of player records by indicating the order of records for each player. Contrary to the first experiment, there is no variable to specify the temporal or ordinal nature of the records in the second experiment, which yields in abysmal results for explainability of variance in the independent variable (check the R2 scores in the second experiment). It is clear that evaluating each record as separate (ignoring past records and statistics) has not an adequate predictive power to asses player skill. Any model aiming an accurate prediction should consider the temporal relevance of records of players.
This situation (poor results on independent evaluation) is also logical at a point. For a learning model, modeling statistics of players by using independent records is a confusing task and not enough to model the relationship (between the player statistics and skills). If a model gets the records as separate observations, it can observe many similar player statistics (input features) from different players with different skills. As mentioned, the problem is that individual records have not enough discriminatory power to separate players by skill. When we check the residual interval plot by models on the second experiment, we have observed that all learning models mostly predict in the same narrow

range (centered on the mean value of target variable(mu)) regardless of the type of a learning model and input features of a data point. This situation also proves that individual assessment of data points is not enough to model the relationship.
We have tested the effects of the addition of the temporal dimension to the modeling pipeline in the third experiment. We have evaluated each player as a separate multivariate time series, and we relate each record on this time series using a set of features indicating temporal trend and inclination. We got very accurate prediction results compared to the second experiment, which also validates our concerns mentioned in the previous paragraph.

5.2 Challenges on the project

There are several noticeable challenges in the methodology and implementation of the project. The toughest one is the absence of skill values of players. Despite the convergence of player skills by TrueSkill and validation by sanity check, we are unable to validate the generated skill of players using ground truth labels completely.
The second shortcoming is that we have collected various player records ranging from amateurs to professionals. Usually, amateur players match with other amateurs, and an amateur may be assessed with a higher ranking compared to a professional one, which is an undesired outcome. This situation also misleads the learning model to understand the relation between player statistics and skill level because of misleading skill assessment. As a supporting point, we have isolated HLTV demo files in the third experiment and got much better prediction accuracy compared to the previous experiments (there are several additional factors as well for good results).
The third challenge in the project is the limited amount of data, especially demo files. Demo files are precious to extract a full set of features of a match, and the increase in the amount of these files can improve learning task for any model, especially neural networks.

5.3 Further improvements

- The most notable improvement for the project is crafting a deep neural network to grasp the sequential nature of player records. A possible model can model round-based and interpret player actions during a round to make a relation with player skill. Alternatively, it can process player statistics with the temporal dimension of the player records based on matches, not rounds.

- New features from demo files for data modeling is another opportunity to enhance learning task. CS:GO is a complicated game that one can craft numerous new features to spot different aspects of a match such as team members interactions and player-environment interactions.

- For extracting match features, we have used mean values of round features (ex: we have averaged kill counts recorded in all rounds to deduce the match kill count).

However, different aggregation features can be added to deduce match features such as median, standard deviation or kurtosis of round features, and many more.

- There is an infinite amount of combination of methods on data processing, filtering, and model training to get the best accuracy. Hyperparameter optimization and fine-tuning each component in each phase would be a valuable improvement in the project.
- The amount of data need to be increased by using more data source or possibly Steam API. Agreements with matchmaking servers to provide data can make a difference as well.
- In the project, we have assessed each player as an individual one not considering team members. However, especially for professional players, a player is usually a part of a consistent team, and there is a direct relationship among the skills of team members. Integrating team factor to the system may increase the predictability of player skill.

Appendix A

Survey for feature ideas

A.1 Survey

CS: GO player feature investigation

Trying to find out which features/attributes on player and team-based effect the outcome of the game as well as an indication of a player skill set. Your ideas are REALLY IMPORTANT. If you think other options or want to make an explanation for it, please provide it. It will be really appreciated.

* Required

1. **On team bases, please rank the importance of features to a team win a match (assuming for each team, all team members have nearly the same skill sets)?** *
Mark only one oval per row.

	1	2	3	4	5	6
Economy management						
Map distribution / use of map						
Well defined role of players						
Good coordination and communication among players						
Weapon selection						
Fast response against opponent team strategies						

2. **Include your opinion, if exist for the above question.**

3. **For a player, please rank a player attributes that you think he is a skillful player.** *
Mark only one oval per row.

	1	2	3	4	5	6	7
Use of a match map							
Weapon selection							
Economy management							
Pure skills like aiming / reflexes							
Player equipments like mouse, keyboard etc.							
Player strategy like flashing, camping, movement trajectory etc.							
Reading opponent responses							

Figure A.1: Feature survey part 1.

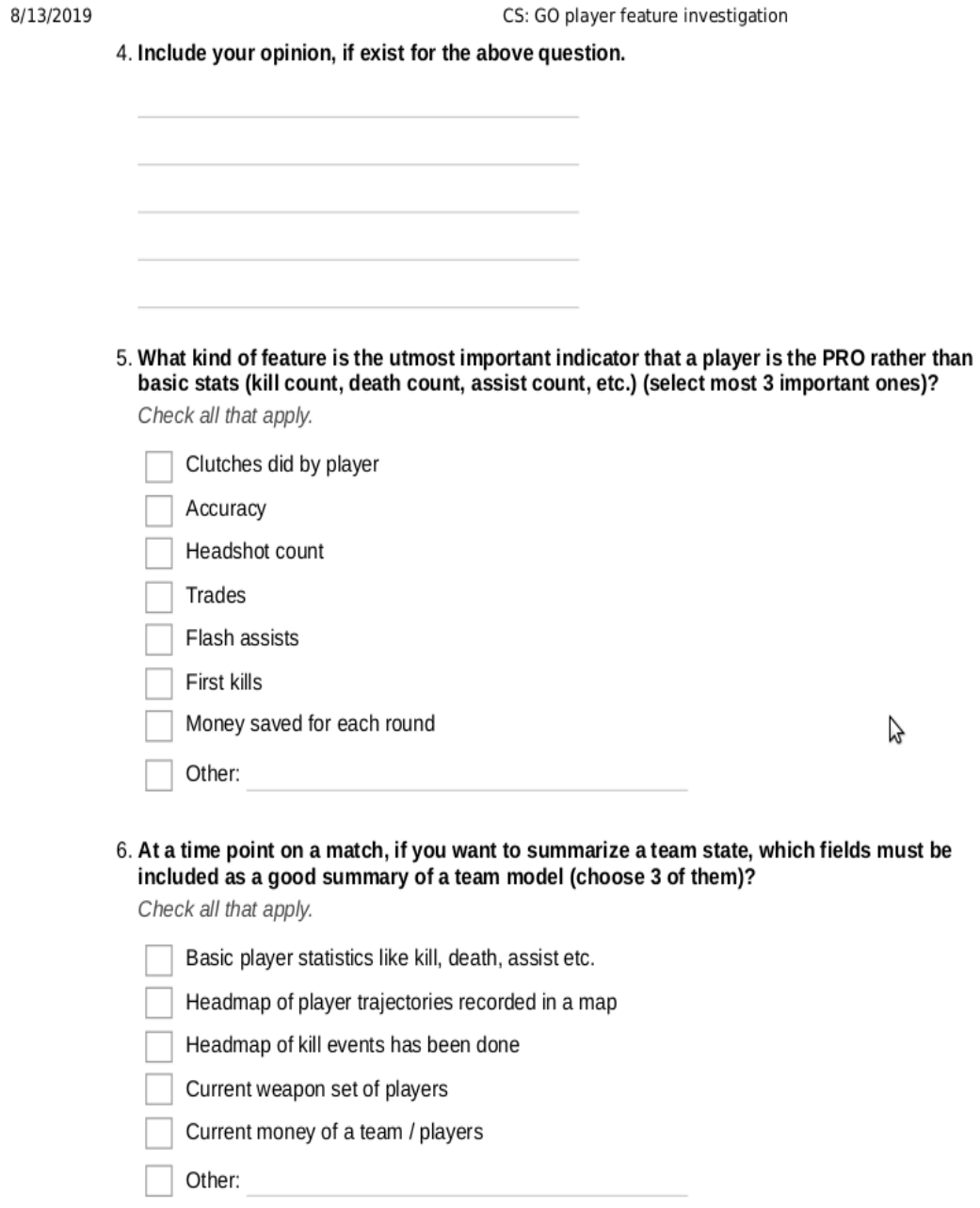
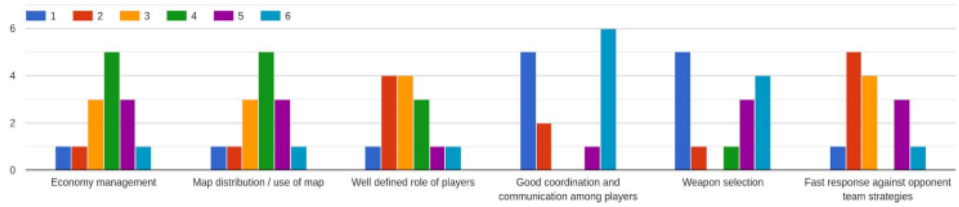


Figure A.2: Feature survey part 2.

A.2 Responses of the survey

On team bases, please rank the importance of features to a team win a match (assuming for each team, all team members have nearly the same skill sets)?

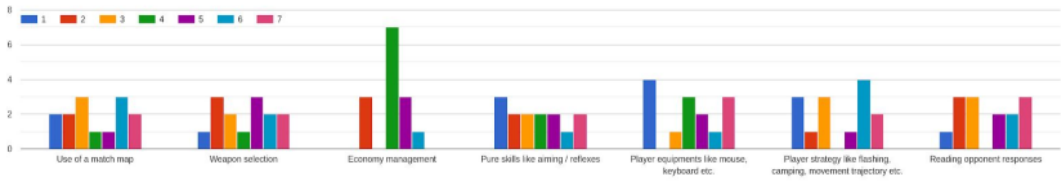


Include your opinion, if exist for the above question.

4 responses

- It really depend on the level of the players, could change a lot from a pro team to some random friends teams
- I think all of those are expremely important for a team to compete at a top level.
- create movment, a lot
- interessant

For a player, please rank a player attributes that you think he is a skillful player.



Include your opinion, if exist for the above question.

3 responses

- Same as above
- I think the equipment you got indeed help you to get a better level, but i think everybody can do it with medium - bad equipment.
- knows how to adapt himself to a team

Figure A.3: Feature survey response part 1.

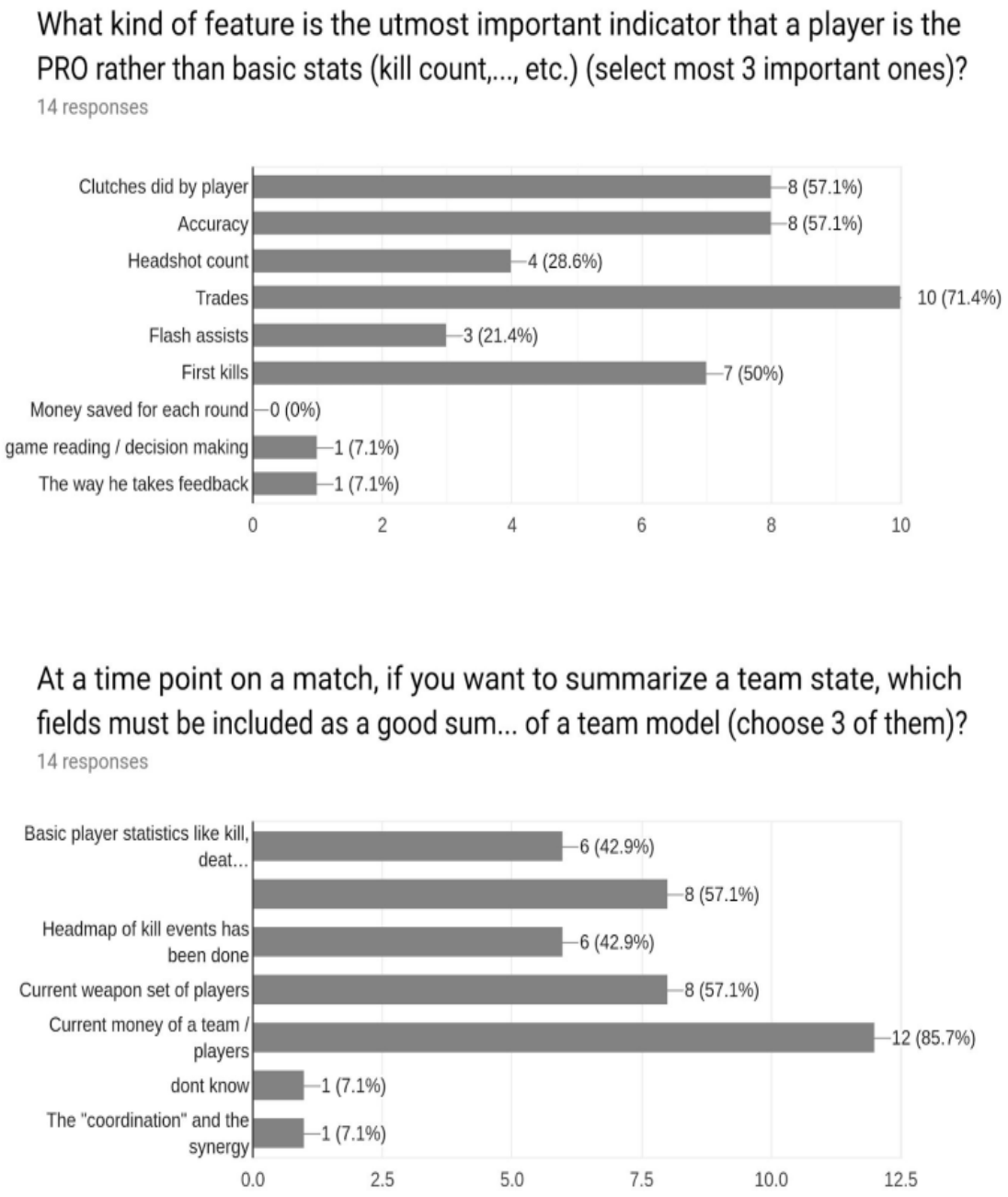


Figure A.4: Feature survey response part 2.

Appendix B

Discussion on feature ideas

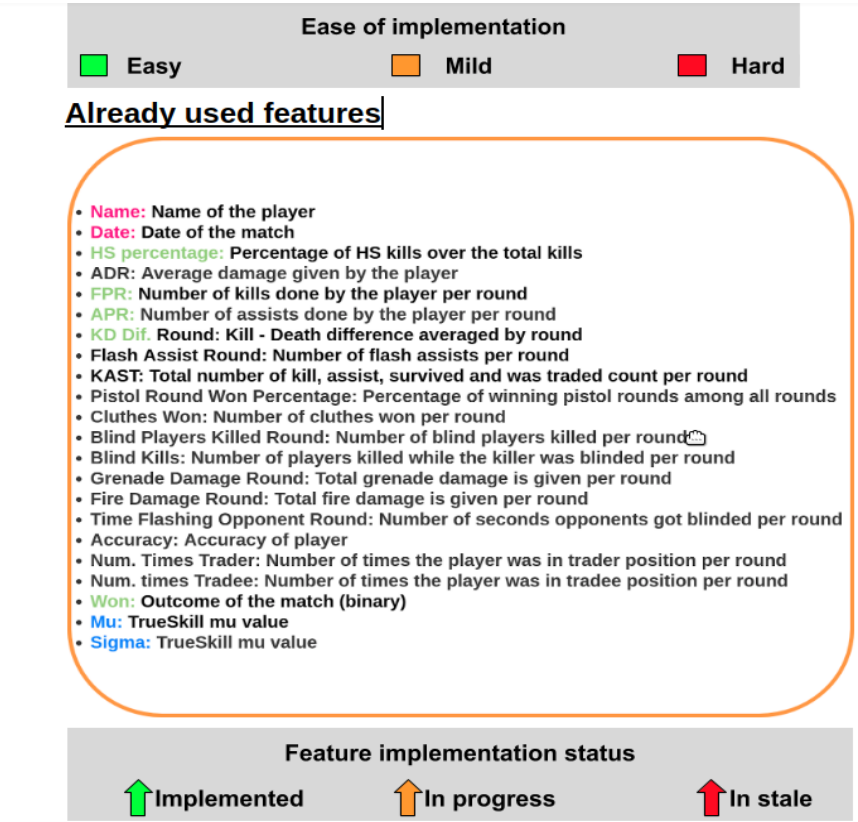


Figure B.1: Feature ideas header.

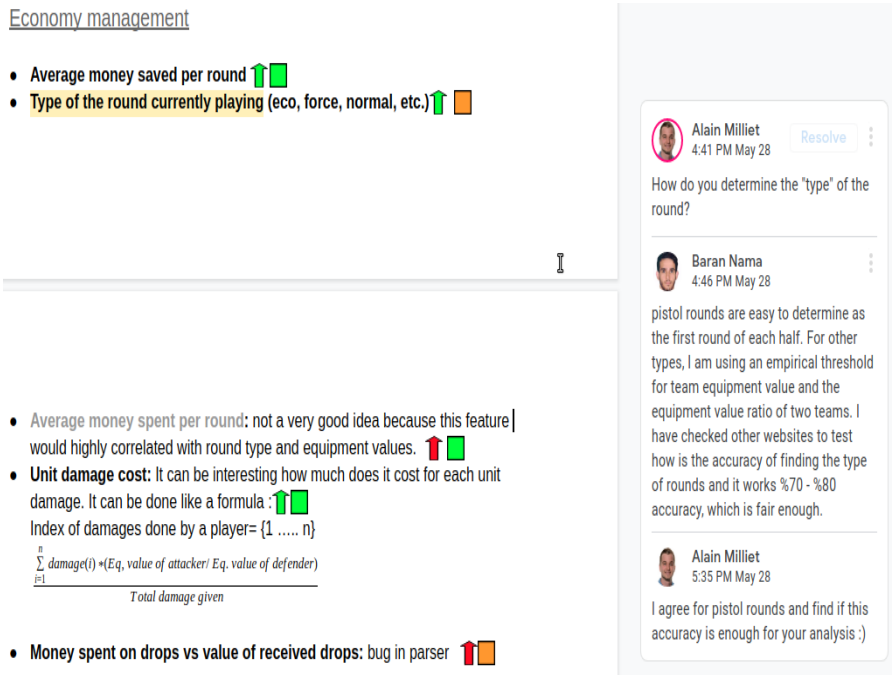


Figure B.2: Economy management feature ideas.

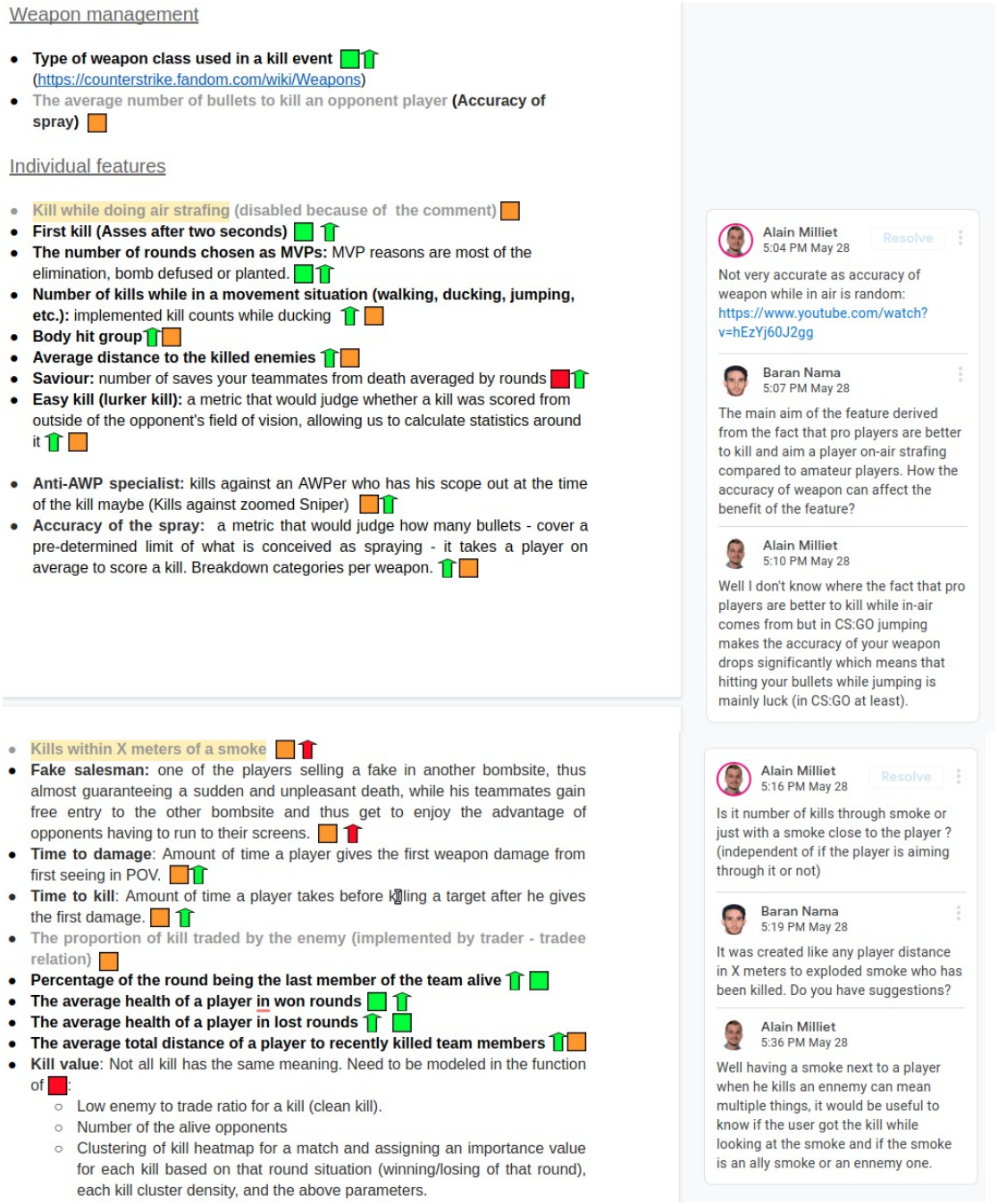
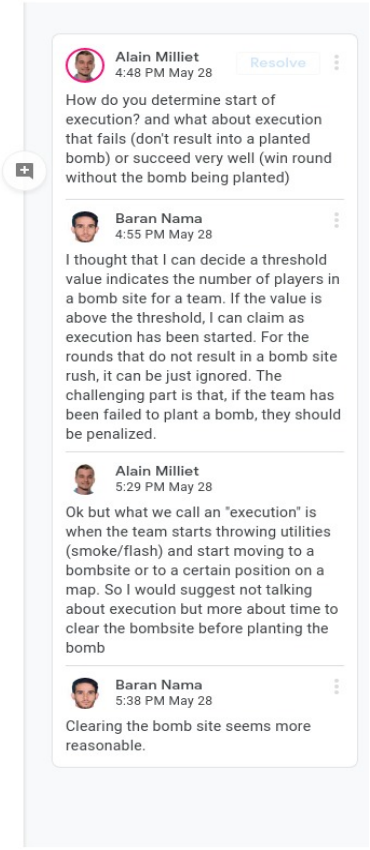


Figure B.3: Weapon management and individual feature ideas.

Team-based interaction

- **Soft assignment a player to a role**:
  - **Challenge**: Multi-role of players. Determination of role for a player.
  - Entry Fragger: first kill, good crosshair placement, recoil control and generally is the most aggressive player on the team, high kill - death counts
  - Secondary entry-fragger: clutches, the high survival ratio
  - Supporter: carrying the bomb, throwing flash, smoke, etc, the low kill ratio
  - AWPers: use of AWP, good reaction time
  - In-game Leader: Hard to model
  - Lurkers: good player - environment interaction, good aim, sneaking back oppose team, picking up "important" frags, stealthy nature, easy (lurker) kill, the ratio of traded deaths for team members is low, usually pretty far from his own team.

- **Communication** → Very hard to get an insight
- **Team pace**: the average starting time of execution on either of the bombsites. This could also include time spent from the start of execution to the time when the bomb is actually planted.
- **Round win percentage**
- **The average number of players remaining in won rounds**: a metric for tracking how cleanly a team wins rounds (explicitly modeled by average health of remaining players and it would be highly correlated with these features).
- **Money spread through the team**: a metric that tells if the team is spreading the money in the team effectively (known as dropping weapons for your mates) (will be modeled by money spent on drops vs taken drops)
- **Round pace:** average round time a winner team finished in seconds for a match
- **Map place occupation**: The amount of square meter each team occupied in a map averaged by round. The algorithm as follows:
  1. We are using NAV[1] files collected beforehand for each known map to get areas, places and connection areas in a map. Originally, NAV files have been developed for pathfinding of AI bots in a map however, we are using those files for the segmentation of places in a map. Each place in a map is composed of several areas.
  2. A diameter threshold value has used for checking whether a player has occupied a place. If the player distance to place center (calculated as a weighted average of center of areas) is smaller than this diameter, and there is no opponent player (occupation rules given below) in the same place, we claim that the player has occupied the place. The diameter calculated as follows: Firstly, we are finding the boundary points of the place by iterating all areas in the place (furthest point of South, North, East, West). After that, we are calculating place width and height and dividing those dimensions with a constant >1. Then we get the min value of the division results which named diameter in 2D. Lastly, we are reflecting this diameter to 3D space by using the angle between z-axis and 2D space. Therefore, we are using a separate diameter for each place by considering the dimensions of that place.
  3. **The occupation rule as follows**: If there is only one team in a place and the team player is near enough (consideration of place diameter and its center at this place mentioned above), we assign the place occupation to this team. If there are two teams at the same time in a place (and both teams are near enough to the center of the place), we make the place unassigned and the place won't be assigned to any team for this iteration of place checking. For each occupation, we also store the tick value of this occupation in order to check the conditions given below.
  4. At the end of the match, we are calculating the total square meters occupied for each team and taking the average by round.



[1] https://developer.valvesoftware.com/wiki/Navigation_Meshes

Figure B.4: Team interaction feature ideas.

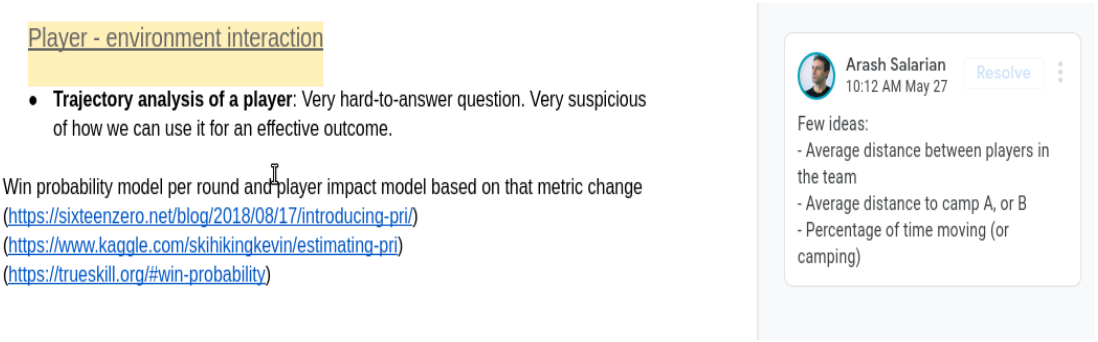


Figure B.5: Player-environment interaction feature ideas.

Bibliography

[1] Joseph Absher. *Counter-Strike: Global Offensive Wins Esports Game of the Year*. [Online; accessed 28-April-2019]. 2016. URL: https://www.12up.com/posts/4153670-counter-strike-global-offensive-wins-esports-game-of-the-year.

[2] Counter-Strike: Global Offensive. *Counter-Strike: Global Offensive — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-April-2019]. 2019. URL: https://en.wikipedia.org/wiki/Counter-Strike:_Global_Offensive.

[3] D. Bednárek et al. "Data Preprocessing of eSport Game Records - Counter-Strike: Global Offensive". In: (Jan. 2017), pp. 269–276. DOI: 10.5220/0006475002690276.

[4] Valve. *DEM Format*. [Online; accessed 28-April-2019]. 2018. URL: https://developer.valvesoftware.com/wiki/DEM_Format#Frame_Format.

[5] Amazon Web Services. *Amazon Web Services — Wikipedia, The Free Encyclopedia*. [Online; accessed 2-May-2019]. 2019. URL: https://en.wikipedia.org/wiki/Amazon_Web_Services.

[6] Amazon Relational Database Service. *Amazon Relational Database Service — Wikipedia, The Free Encyclopedia*. [Online; accessed 2-May-2019]. 2019. URL: https://en.wikipedia.org/wiki/Amazon_Relational_Database_Service.

[7] Amazon. *Amazon Elastic MapReduce*. [Online; accessed 2-May-2019]. 2019. URL: https://www.amazonaws.cn/en/elasticmapreduce/.

[8] R. Herbrich, T. Minka, and T. Graepel. "TrueSkill(TM) Ranking System - Microsoft Research". In: (2005). [Online; accessed 18-May-2019]. URL: https://www.microsoft.com/en-us/research/project/trueskill-ranking-system.

[9] S. Mackie. *The Math Behind ELO*. [Online; accessed 20-May-2019]. 2017. URL: https://blog.mackie.io/the-elo-algorithm.

[10] M. E. Glickman. "The Glicko system". In: (1995). [Online; accessed 18-May-2019]. URL: http://www.glicko.net/glicko/glicko.pdf.

[11] J. Moser. "The Math Behind TrueSkill". In: (2010). [Online; accessed 18-May-2019]. URL: http://www.moserware.com/assets/computing-your-skill/The%20Math%20Behind%20TrueSkill.pdf.

[12] F. R. Kschischang, B. J. Frey, and H. -. Loeliger. "Factor graphs and the sum-product algorithm". In: *IEEE Transactions on Information Theory* 47.2 (Feb. 2001), pp. 498–519. ISSN: 0018-9448. DOI: 10.1109/18.910572. URL: https://ieeexplore.ieee.org/document/910572.

[13] T. P Minka. "Expectation Propagation for approximate Bayesian inference". In: *arXiv preprint arXiv:1301.2294* (2013). URL: https://tminka.github.io/papers/ep/minka-ep-uai.pdf.

[14] M. J. Wainwright and M. I. Jordan. "Graphical Models, Exponential Families, and Variational Inference". In: *Found. Trends Mach. Learn.* 1.1-2 (Jan. 2008), pp. 1–305. ISSN: 1935-8237. DOI: 10.1561/2200000001. URL: http://dx.doi.org/10.1561/2200000001.

[15] J. Hershey and P.A. Olsen. "Approximating the Kullback Leibler Divergence Between Gaussian Mixture Models". In: vol. 4. May 2007, pp. IV–317. ISBN: 1-4244-0728-1. DOI: 10.1109/ICASSP.2007.366913.

[16] G. V. Bard. "Spelling-error tolerant, order-independent pass-phrases via the Damerau-Levenshtein string-edit distance metric". In: *Proceedings of the fifth Australasian symposium on ACSW frontiers-Volume 68*. Citeseer. 2007, pp. 117–124.

[17] Microsoft Research Lab. *TrueSkill™ Ranking System*. [Online; accessed 23-May-2019]. 2005. URL: https://www.microsoft.com/en-us/research/project/trueskill-ranking-system/.

[18] Y. Koren, R. Bell, and C. Volinsky. "Matrix Factorization Techniques for Recommender Systems". In: *Computer* 42.8 (Aug. 2009), pp. 30–37. ISSN: 0018-9162. DOI: 10.1109/MC.2009.263.

[19] H. Peng, F. Long, and C. Ding. "Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27.8 (Aug. 2005), pp. 1226–1238. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2005.159.

[20] M. B. Kursa and R. R. Witold. "Feature selection with the Boruta package". In: *J Stat Softw* 36.11 (2010), pp. 1–13.

[21] M. Calzolari. *manuel-calzolari/sklearn-genetic: sklearn-genetic 0.2*. Apr. 2019. DOI: 10.5281/zenodo.3348077. URL: https://doi.org/10.5281/zenodo.3348077.

[22] I. Yeo and R. A. Johnson. "A New Family of Power Transformations to Improve Normality or Symmetry". In: *Biometrika* 87.4 (2000), pp. 954–959. ISSN: 00063444. URL: http://www.jstor.org/stable/2673623.

[23] L. McInnes, J. Healy, and J. Melville. "Umap: Uniform manifold approximation and projection for dimension reduction". In: *arXiv preprint arXiv:1802.03426* (2018).