ppshift machine learning

In this document, we will be discussing methods of obtaining a credible way of classifying difficulty in VSRG maps. We will first establish what makes a map difficult, then we build from there!

Part I
Define Difficulty

1 Difficulty in Playing

What makes a map difficult, what is a difficult map? Could it be the following? The map was difficult because of ...

1. Failing
2. Combo Breaks
3. High Stamina Requirement
4. Low Accuracy

We discuss all of these scenarios and we will choose one to tackle, possibly integrate the other options into our calculations in the future.

Failing The most significant way that we can readily control if players fail is via **Health Drain** in which most VSRGs will implement. However, this value is inconsistent and will not provide useful information on higher **Health Drain** values due to lack of players passing certain maps.

Combo Breaks Combo Breaks analysis is another method that isn't consistent, whereby chokes can be random, creating too much noise on higher skill plays. Combo breaks mainly can only determine the **hardest** points on the map, it doesn't depict a difficulty cure.

High Stamina Requirements While stamina is a good way to look at difficulty, it can readily be derived from accuracy, which is conveniently what we'll be looking at next

Low Accuracy This is the best way to look at difficulty, because not only it gives us a figure, it tells us the story and correlation between **accuracy** and **patterning**. This will be the main focus of the document.

2 Difficulty from Accuracy

It is possible to measure difficulty, by just looking at accuracy, in-fact, it's quite straight-forward to do so.

Consider this...

1. $Player_1$ plays $Beatmap_A$ and $Beatmap_B$
2. $Player_1$ achieves $Accuracy_A > Accuracy_B$
3. Considering $Accuracy_A$ and $Accuracy_B$ are independent events
4. We deduce $Difficulty_A < Difficulty_B$

Now, on a larger scale...

1. $Player_{all}$ play $Beatmap_A$ and $Beatmap_B$
2. $\prod(Accuracy_A/Accuracy_B) > 1$
3. Considering $Accuracy_A$ and $Accuracy_B$ are independent events
4. We deduce $Difficulty_A < Difficulty_B$

We can further develop this algorithm to estimate what the difficulties would be by just looking at the ratios, I won't elaborate this further as this won't be the main aim of this document because **this only works if you have scores to pivot from**. This will not work with newly developed maps.

2.1 Replay data to actions

In order to compare *accuracy* and *patterning* we need to change our perspective, instead of looking at *accuracy* as a number, we will look at it as a vector. Whereby we extract player data from provided replays.

2.1.1 GETting beatmap_id data from osu!API

Using osu!API, I was able to extract all **beatmap IDs** where $StarRating \leq 3.0$ as any maps below this $StarRating$ will usually only have **SS** scores, which will prove to be redundant in analysis.

With this list, we are able to know what maps we are tackling in the following sections.

2.1.2 GETting Replay data from osu!API

osu!API provides us with essential data, including the replay file itself. Not going into detail, we are able to extract all key taps (including releases) of the player during the play.

I will provide python code I have used in the GitHub Repository or in the Annex of the document.

The format we get from running the python code goes as follows

$$action_{replay} := \{(offset_1, action_1), (offset_2, action_2), ..., (offset_n, action_n)\}$$

Whereby,

$$n \in \{-9, -8, ..., -2, -1, 1, 2, ..., 8, 9\}$$

$offset$ is when the *action* happens. For *action*, $-n$ means the key **n** is released, n means the key **n** is pressed.

We will save this data in a file with $< beatmap\_id >$ *.acr* extension.

2.2 Difficulty data to actions

In order to make use of $Action_{replay}$, we need to obtain $Action_{difficulty}$.

2.2.1 Downloading Difficulties through web crawling

Using python, we can download *.osu* (osu! difficulty file extension) using the format below with authentication:

$$https://osu.ppy.sh/osu/ < beatmap\_id >$$

We save all of these using $< beatmap\_id >$ *.osu* file naming system.

2.2.2 Converting Difficulties to Action format

As to be aligned with $action_{replay}$ format, we need to convert all difficulties to $action_{difficulty}$. This can be done with a python script.

However, we should note that maps with **variable scroll speeds** will cause anomalies in our calculation, so we need to further define what maps defy a threshold we set to remove these from our calculations.

Scroll Speed Manipulation Threshold A simple way of tackling this issue would be to skip all beatmaps that has any of the following:

1. Any **slight** SV Change $0.97 \leq TP_{SV} \leq 1.03$
2. Any BPM Change

After skipping beatmaps, we will grab all remaining *.osu* and convert them to *.acd*. The action format is the same idea, where we have a list of offset and action.

2.3 Mapping $action_{replay}$ to $action_{difficulty}$

This is the last essential step to find out *accuracy* as a vector. We will match all similar actions in their respective columns together.

There are a few things we need to take note of when matching:

1. Not all $action_{difficulty}$ will have a matching $action_{replay}$
2. We put the threshold of this matching as $100ms$.
3. The nearest $action_{replay}$ will match the $action_{difficulty}$, not the earliest one.
4. We will deviate on how osu! calculate accuracy due to the above pointers, but its difference is insignificant.

We will expect the output of:

$$deviation := \{(offset_1, deviation_1), (offset_2, deviation_2), ..., (offset_n, deviation_n)\}$$

Where:

$$n = length(action_{difficulty})$$

And if there's no match, $deviation = 101$, this is to allow us to understand that it's a **miss** instead of a $100ms$ hit.

However, this is a bit ugly, so what we will use is the following:

$$accuracy := \{(offset_1, accuracy_1), (offset_2, accuracy_2), ..., (offset_n, accuracy_n)\}$$

Where:

$$accuracy_n = \frac{1}{deviation_n}$$

$$n = length(action_{difficulty})$$

Therefore, a **miss** would simply just be $accuracy = 0$ instead of the ugly $accuracy = 1/101$
So accuracy will only span:

$$(Miss)0 \leq accuracy_n \leq (Perfect)1$$

$$deviation_n \in [0, 1, 2, ..., 99, 100]$$

3 Difficulty from Patterning

We turn our attention to how we can figure out difficulty from the map itself, the expected output we want would be:

$$difficulty := \{(offset_1, difficulty_1), (offset_2, difficulty_2), ..., (offset_n, difficulty_n)\}$$

Whereby we estimate difficulty from the map itself:

$$difficulty \approx reading + \sum_{n=1}^{keys} (density_n + strain_n) + ...$$

There are more factors (denoted by ...) that contribute to difficulty, but we will regard them as noise in this research and fine tune this equation later.

Reading This denotes how hard is it to read all the patterns on the screen. We can draw similarities between this and density, however this looks beyond note density and estimates the difficulty of reading different similar density patterns.

Density This focuses on the imminent density of the offset (contrary to strain), whereby it disregards the global trends of patterns.

Strain This is reliant on *density* whereby continuous high values of *density* will result in a high *strain*. This has an additional hyperparameter, *decay*, where it denotes how fast the player can recover from $strain_n$. Finger *strain* on the same hand will likely affect the other *strain* values of the other fingers.

3.1 Reading

$$reading_{(i,j)} = \frac{\sum_i^j (note + (longNote_{head} + longNote_{tail}) * \Gamma))}{j - i}$$

Where,

i is the initial offset

j is the end offset

Γ is the hyperparameter for how difficult a long note is to read
We will not take into consideration the length of $note_{long}$