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
Preface

1 Define Difficulty

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.

1.1 Comparing Accuracy in Replays to Patterns

Details aside, how can we describe the impact of patterns on replays, what story does the replay tell us about the pattern?

Consider this...

1. $Player_1$ plays $Pattern_A$ and $Pattern_B$
2. $Player_1$ achieves $Accuracy_A > Accuracy_B$
3. Considering $Accuracy_A$ and $Accuracy_B$ are independent events
4. We deduce $Pattern_A < Pattern_B$ in difficulty

It is a simple idea to pitch, but we will have to dive into more details on how we can "teach" a machine this concept and "learn" from it!

2 Machine Learning

A tangent from this article, I will briefly talk about what is machine learning.

Making a Prediction Machine Learning is all about making a prediction, by looking at already curated data. *We show the someone 100 pictures of rabbits and explained to them that they are rabbits, can they tell if the next picture of an animal is a rabbit?*

Same idea we have here, we show the machine hundred of beatmaps, we tell the machine how difficult all of them are (according to score regression). Can the machine predict the difficulty of the next one? The answer is yes, but it won't be accurate!

2.1 Valid inputs

Think about a neural network, you've most likely seen one, it's circle(s) connected to more circle(s), in the end, there will be circle(s) that tell you something. Each circle represents a **neuron**.

Each neuron holds a numeric value (between 0 and 1), we need to put a value in each of these neurons. Ignoring Timing Points and Scroll Speed Changes, how do we squeeze a beatmap into these neurons?

Hit Object Neurons Remember that a neuron will only work with numeric values, what do you want to have in the neuron that represents a Hit Object? Putting either the **offset** or **column** will not work because both are vital!

Column Neurons It is a possible idea, to have a neuron for **each millisecond**, then put **column value** on those neurons that match the **offset**. We run into a glaring issue where input neurons will stretch to the **hundred thousands**, computational power required on this scale would be too high. If we decide to **bin** the offsets (binning is the act of grouping things together to reduce the number of fields) to **100ms** steps, it'll still be in the thousands.

2.2 Subdivide the Issue

Does it have to be the whole map?

It doesn't! That's what we are doing for this research, we don't look at the whole map, we look at parts of it, the inner mechanisms, the **patterns**!

Sure, even though the whole map must be present to calculate difficulty but if we break it into 2 distinct parts, it's easier to visualize and control the system.

1. Patterns to Difficulties
2. Difficulties to Map Rating

Patterns to Difficulties In the following sections we will be discussing how we can task the machine to learn what patterns are easy and which are harder

Difficulties to Map Rating Using the **difficulty** values defined previously, we can find out how to use that to reliably give the map a rating. Do we rate it by the average, maximum or median? We will discuss those later.

Part II
Data Analysis

2.3 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.

2.3.1 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.

2.3.2 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

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

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.

2.4 Difficulty data to actions

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

2.4.1 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.

2.4.2 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.

2.5 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:

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

Whereby we estimate difficulty at offset **n** from the map itself:

$$difficulty_n \approx reading_n + \sum_{k=1}^{keys} (strain_k) + ...$$

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.

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.

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

3.1 Note Type Weights

This will define the **weightages** of each note type.

$weight_{NN}$ defines for normal notes

$weight_{LNh}$ defines for long notes heads

$weight_{LNt}$ defines for long notes tails

$weight_{SSh}$ defines for **strain shift** for hands **(explained later)**

$weight_{SSb}$ defines for **strain shift** for body **(explained later)**

3.2 Reading

$$reading_{(n,n+\theta)} = \frac{count(NN) + [count(LNh) + count(LNt)] * \Gamma}{\theta} = \{n \leq offset \leq (n + \theta)\}$$

Where,

n is the initial offset

θ is the hyperparameter for length.

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

3.3 Density

We will look into density before strain as it's derived from this.
Considering the notes on the k column

$$\{n-2, n-1, n, n+1, n+2\}$$

$$\Delta_{nx}^{k} = \frac{1}{n-x}$$

$$density_{n}^{k} = \sum_{N=n-\sigma}^{n+\sigma} \left(\Delta_{nN}^{k}\right)$$

So for $\sigma = 2$ and

$$column_k := \{a, b, n, d, e\}$$

$$density_{n}^{k} = \Delta_{na}^{k} + \Delta_{nb}^{k} + \Delta_{nd}^{k} + \Delta_{ne}^{k}$$

Δ_{nx}^{k} will be the the inverse of the (ms) distance between notes n and x on column k. Notes that are further away will be penalized with a square.

σ defines the range, front and back of the search. Higher sigma may prove to be useless with further Δ_{nx}^{k} being too small.

3.4 Strain

This will work in relationship with *density*, whereby a *strain* is a cumulative function of *density* with a **linear decay function**.
Notes:

1. Better players have **higher decay gradients**
2. If $decay > density$, *strain* will **decrease**
3. If $decay < density$, *strain* will **increase**
4. There will be a point where *strain* is high enough to affect physical performance, indirectly affecting accuracy.

3.4.1 Strain Shift

Strain will not only affect one finger, it will affect the hand and both after time, just on a smaller scale

Hand We will denote the strain shift hyperparameter of one finger to another on the same hand to be SS_H

Body Likewise, for body, we will denote as SS_B

3.4.2 Strain Example

Consider the case, without **Strain Shift**

$$Where, weight_{NN} = 1, \sigma = 2$$

2500	0	0	0		0.022	0.016
2000	$weight_{NN}$	0	0	0.003	0.022	0.017
1500	$weight_{NN}$	0	0	0.005	0.019	0.015
1000	$weight_{NN}$	0	0	0.006	0.014	0.011
500	$weight_{NN}$	0	0	0.005	0.008	0.006
0	$weight_{NN}$	0	0	0.003	0.003	0.002
-500	0	0	0		0	0
-1000	0	0	0		0	0
Offset(ms)	k=1	k=2	k=3	$\approx Density$	Strain (dec=0)	Strain (dec=0.001)

Consider the case, with **Strain Shift**

2500	0	0	0
2000	$weight_{NN}$	$weight_{SSh}$	$weight_{SSb}$
1500	$weight_{NN}$	$weight_{SSh}$	$weight_{SSb}$
1000	$weight_{NN}$	$weight_{SSh}$	$weight_{SSb}$
500	$weight_{NN}$	$weight_{SSh}$	$weight_{SSb}$
0	$weight_{NN}$	$weight_{SSh}$	$weight_{SSb}$
-500	0	0	0
-1000	0	0	0
Offset(ms)	k=1	k=2	k=3

It's hard to include the calculations in the table, so we'll look at $density_{(1,1000)}$, we will also elaborate on the calculations without strain shift.

$$density^1_{1000} = (\Delta^1_{(1000,0)}) + (\Delta^1_{(1000,500)}) + (\Delta^1_{(1000,1500)}) + (\Delta^1_{(1000,2000)})$$

$$density^1_{1000} = \frac{1}{1000} + \frac{1}{500} + \frac{1}{500} + \frac{1}{1000} = 0.006$$

$$density^2_{1000} = (\Delta^2_{(1000,0)}) + (\Delta^2_{(1000,500)}) + (\Delta^2_{(1000,1500)}) + (\Delta^2_{(1000,2000)})$$

$$density^2_{1000} = (\frac{weight_{SSh}}{1000}) + (\frac{weight_{SSh}}{500}) + (\frac{weight_{SSh}}{500}) + (\frac{weight_{SSh}}{1000}) = \frac{3 * weight_{SSh}}{250}$$

$$density^3_{1000} = \frac{3 * weight_{SSb}}{250}$$

$$density_{1000} = density^1_{1000} + density^2_{1000} + density^3_{1000} = 0.006 + \frac{3 * weight_{SSh}}{250} + \frac{3 * weight_{SSb}}{250}$$

3.5 Density Generalization

In the case where we want to find $density_n$, where, n is the offset index, k is key count.

$$\begin{bmatrix} weight_{(n+\sigma,1)} & weight_{(n+\sigma,2)} & \dots & weight_{(n+\sigma,k)} \\ \vdots & \vdots & \iddots & \vdots \\ weight_{(n+1,1)} & weight_{(n+1,2)} & \dots & weight_{(n+1,k)} \\ weight_{(n,1)} & weight_{(n,2)} & \dots & weight_{(n,k)} \\ weight_{(n-1,1)} & weight_{(n-1,2)} & \dots & weight_{(n-1,k)} \\ \vdots & \vdots & \ddots & \vdots \\ weight_{(n-\sigma,1)} & weight_{(n-\sigma,2)} & \dots & weight_{(n-\sigma,k)} \end{bmatrix}$$

$$*$$

$$\begin{bmatrix} offset_{n+\sigma} & \dots & offset_{n+1} & offset_n & offset_{n-1} & \dots & offset_{n-\sigma} \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} density_{n+\sigma} & \dots & density_{n+1} & density_n & density_{n-1} & \dots & density_{n-\sigma} \end{bmatrix}$$

$$\sum \begin{bmatrix} density_{n+\sigma} & \dots & density_{n+1} & density_n & density_{n-1} & \dots & density_{n-\sigma} \end{bmatrix} = density_n$$

From here, we can calculate the strain by running the through a python code.

3.6 Assigning Hyperparameters

In this section alone, we have used quite a few hyperparameters. To recap:

(Reading) θ is the hyperparameter for reading length.

(Reading) Γ is the hyperparameter for how difficult a long note is to read

(Density) σ defines the range, front and back of the density search. Higher sigma may prove to be useless with further Δ_{nx}^{k} being too small.

(Density) $weight_{NN}$ defines for normal notes

(Density) $weight_{LNh}$ defines for long notes heads

(Density) $weight_{LNt}$ defines for long notes tails

(Density) $weight_{SSh}$ defines for **strain shift** for hands

(Density) $weight_{SSb}$ defines for **strain shift** for body

Now what we need to do is to assign a reasonable values to these, and run the results to find our:

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

Whereby we estimate difficulty at offset **n** from the map itself:

$$difficulty_n \approx reading_n + \sum_{k=1}^{keys} (strain_k)$$

4 Creating a Neural Network

Before we dive into creating a neural network to finalize the calculation, we need to define specifically these few things:

1. Input
2. Layers
3. Output

4.1 What is our input?

Our input will be simply the $difficulty$ (matrix/2D vector).