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 input

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.

2.1.1 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 procedures**, 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.

2.2 Expected output

Just like the example in **valid inputs**, we have to tell the machine that **that picture is of a cat**, else it doesn't know what is and what isn't.

In this case, we can relate back to *Accuracy* (Player), where it is the most reliable source of difficulty rating as explained earlier.

One good source of this data is to grab replays (not scores!), because we can then analyse small parts of it in order to match with the input.

2.2.1 Differing and Multiple Accuracies

However, there are two problems:

Multiple Players There will be multiple replays, there isn't a replay that is all-encompassing (it'll be too biased!). So we need to create a middle-ground for these data.

Different Players Not everyone will perform similarly for every beatmap, there will be discrepancies. We can't just take the accuracy as a raw value and run a function through them, we will have to make use of relative accuracies instead of raw accuracies before finding the "ideal" replay.

2.3 Recap

To sum it up

1. Grab beatmap from server
2. Grab replay from server
3. Convert beatmap into simpler bite-sized patterns
4. Convert replay to a list of accuracies for each note
5. For each pattern, there's an expected accuracy
6. Teach that concept to the machine and create a model
7. Predict accuracies with the model
8. Give the map a rating according to the expected accuracies

Part II
Process

3 Grabbing Beatmaps

3.1 Beatmap Downloading

In this study we will only look at maps where $StarRating \leq 3.0$. Any maps below this $StarRating$ will usually only have **SS** scores, which will prove to be redundant in analysis.

osu!API GET beatmaps Using this, we can find out the IDs of the maps to download.

3.1.1 Downloading 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.

3.2 Beatmap Validation

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.

4 Grabbing Replays

4.1 Replay Downloading

We need to look at *replays* as a mean to find out how well the player does (in other words, $difficulty$ of the map)

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

These files are not saved, instead they are instantly decoded into the following format.

4.2 Replay Decoding

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.

5 Difficulty from Beatmaps

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 at offset **n** from the map itself:

$$difficulty_n \approx model\left(reading_n, \sum_{k=1}^{keys}(strain_k), ...\right)$$

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 density focuses a lot more on its previous and future surroundings where reading looks at the future ones only.

Density This focuses on the **imminent** density of the offset. contrary to strain, it disregards the global trends of patterns. We will not use this in our network as strain does a better job in calculation.

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.

5.1 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)**

5.2 Reading

$$reading_{(n,n+\theta)} := count(NN), count(LNh), count(LNt) = \{n \leq offset \leq (n+\theta)\}$$

Where,

n is the initial offset

θ is the hyperparameter for length. We will not take into consideration the length of $note_{long}$

5.3 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

$$column_k \coloneqq \{a, b, n, d, e\}$$

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

$$density_{n}^{k} = \frac{1}{|n-a|} + \frac{1}{|n-b|} + \frac{1}{|n-d|} + \frac{1}{|n-e|}$$

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.

5.4 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.

5.4.1 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

5.4.2 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.

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

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

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

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

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

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

5.5 Density Generalization

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

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

$$*$$

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

$$=$$

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

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

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

5.6 Allocating Notes to Fingers

We cannot assume that $column_1$ where $keys = 4$ is the same as $column_1$ where $keys = 7$. This is due to how **different fingers interact with the same column**.

As to counter this, we need to find out the **most common set-up** for players.

key	LP	LR	LM	LI	S	RI	RM	RR	RP
4			1	2		3	4		
5			1	2	3	4	5		
6		1	2	3		4	5	6	
7		1	2	3	4	5	6	7	
8	1	2	3	4		5	6	7	8
9	1	2	3	4	5	6	7	8	9

L represents left, **R** is right, the second letter will be th e name for the fingers (LP: Left Pinky, LR: Left Ring and so on...)

This allocation will give us a consistent result for all beatmaps, so $key = 1$ will always mean **Left Pinky**, $key = 2$ for **Left Ring**, and so on...

5.6.1 8 Key Scratch Bias

The issue with 8 Key maps is how maps will usually have a *scratch column*, this will create a setup that **excludes a pinky but includes the thumb**. This assumption may prove to be important if the results are heavily affected by including/excluding 8 Key maps.

5.7 Assigning Hyperparameters

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

(Reading) θ is the hyperparameter for reading length.

(Density) σ defines the range, front and back of the density search. Higher sigma may prove to be useless with further Δ^k_{nx} 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:

$$difficulty_n \approx model\left(reading_n, \sum_{k=1}^{keys}(strain_k), ...\right)$$

We can further expand this to:

$$difficulty_n \approx model\left(count(NN), count(LNh), count(LNt), \sum_{k=1}^{keys}(strain_k)\right)$$

5.7.1 Values of Hyperparameters

There will definitely be issues when it comes to assuming hyperparameters because of how it will affect accuracy of the model. However, as long as we **reasonably** assign them, most of the errors will be offset by the neural network learning. We just need to focus on if a certain value should be **larger/smaller** or **negative/positive**.

(Reading) θ 1000 (ms)

(Density) σ 2

(Density) $weight_{NN}$ 1

(Density) $weight_{LNh}$ 0.75 (we will follow Reading)

(Density) $weight_{LNt}$ 0.75 (we will follow Reading)

(Density) $weight_{SSh}$ 0.25

(Density) $weight_{SSb}$ 0.1

6 Difficulty from Replays

We are able to extract data from replays, however, they are not linked to the beatmaps themselves. This means that it only has data of what keys and when did the player press it, there's no data on accuracy achieved.

To recap, we managed to decode the replay sent into the following format:

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

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.

6.1 Mapping a Replay to Beatmap

In this, we match all similar actions in their respective columns.

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

1. Not all $action_{beatmap}$ will have a matching $action_{replay}$
2. We put the threshold of this matching as $100ms$.
3. The nearest $action_{replay}$ will match the $action_{beatmap}$, 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:

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

And if there's no match, $deviation > 100$, 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:

$$accuracy_n = 100 - deviation_n$$

Therefore, a **miss** would simply just be $accuracy = 0$ instead.

So accuracy will only span:

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

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

6.2 Replay Soloing

As discussed in the preface, we need to resolve two issues. **Multiple Replays** and **Multiple Players**. We will expect an output similar to a replay, this is where the first major assumption kicks in.

6.2.1 Assumption of the Top 50

In this, we assume that if we took the **median** all top 50 replays, we will end up with a replay that is all-encompassing.

This leads some problems:

Top Player Bias The neural network will perform worse on easier maps $\approx 3.0 S.R.$ due to the median in easier maps being too consistently perfect, this leads to amplification of noise (in this case, chokes).

Population Interaction This assumption will only hold if the beatmaps are old enough such that most of the general playerbase has played it, else it doesn't represent the population well enough due to low participation

Population Decay/Improvement The beatmaps we check must be ranked within close proximity with each other, this is to avoid the Top 50 median from adjusting too much

6.2.2 Assumption of the Player

In this, instead of looking at it **per beatmap**, we will do it **per player**. This is much more consistent in data, however it's consistent **for that player only**.

In this method, we grab the all replays that **Player X** has played and compare them with each other. However this still leads to assumptions:

1. The player played all the beatmaps at a similar time, or the player never improved
2. The player is representative of the community (bias)

The problem of the machine being **biased** to only the player can be fixed if we ran an averaging function on the output with hundreds of other players.

However, due to how osu! sends API scores, I couldn't get more than **50** scores **per player**. This means that there will be no data for these replays.

6.3 Choosing an Assumption

Despite this, we will work with **Assumption of the Top 50** as it's reasonable enough, and it's easier.

This means that the median of all replays will be saved in a different data file with $<beatmap\_id>.acrv$ extension. *(v represents virtual)*

6.4 Virtual Player/Replay

The median of the top 50 creates a **virtual replay**, where we expect it to be a **good enough** representation of a **virtual player**. We can now pivot from this player as we calculate difficulty!

7 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. Output
3. Layers

Input Our input will be a

Output The output will be the expected *accuracy*, which is gathered from the replays.

Layers The amount and neurons of layers we use will be a hyperparameter. We will assign a value for those later

Part III
Refinement

8 Refining Neural Network