Estimating Difficulty in osu! with Sequential Models

Student: Ben Wonderlin (bmw49)

Adviser: Prof. James Glenn

0 Abstract

Osu! (stylized “osu!”) is a competitive, single-player rhythm game for PC. Since its release in 2007, as many as 19.3 million players have competed on its leaderboards each month. However, the algorithm it uses to estimate difficulty – and in turn, compute its leaderboards – is somewhat naive. This project addresses the current algorithm’s shortcomings by presenting a novel, data-driven approach in which a machine-learning model is trained on a large amount of replay files. The model’s classification probabilities are then used to compute the probability of performing any sequence of in-game actions. This yields a difficulty algorithm that is probabilistic, interpretable, and maintainable.

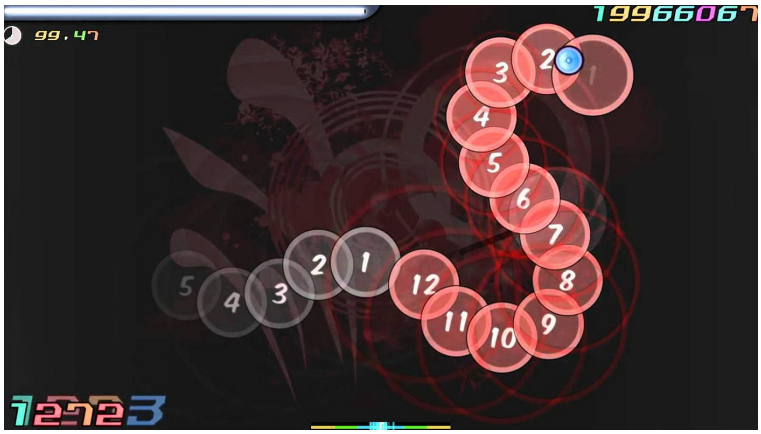
To validate this approach, tests were conducted on a publicly available dataset of 381,570 replays. This led to three main findings. First, sequence-to-sequence models were found to outperform sequential and feed-forward neural networks at predicting osu! data. The sequence-to-sequence model reduced test loss by 20% versus the sequential model, while the sequential model did not reduce test loss versus the feed-forward model. Second, probabilistic difficulty estimates, despite being novel, were found to closely align with current difficulty estimations. This demonstrated the viability of the approach. Third, probabilistic difficulty estimates were found to be sensitive to bias. The bias in the training dataset caused all three models to underestimate the difficulty of conventionally difficult actions. Together, these findings suggest that future efforts would be best put towards building a representative dataset of replays (rather than improving model design). A representative dataset may be all that is necessary to develop a viable data-driven algorithm.

1 Background

1.1 osu!standard

osu!standard is osu!’s flagship gamemode and the focus of this project. Its gameplay consists of clicking circles (or “hitting notes”) in time to the beat of a song. The notes appear in a 512-by-384 pixel window as specified by a song's “beatmap,” which is fixed across attempts. The player’s task is to quickly react, move, and click each note (or else they will “miss”). If the player hits enough notes to reach the end of the beatmap, they successfully complete a “performance.” Performances are the

primary measurement of skill in osu!; beatmap segments are not tracked individually, so each player's ranking is determined by their best performances. Thus from a competitive standpoint, the goal of osu! is to complete high-quality performances.



osu!standard gameplay (cursor in upper right)

There are two main components that comprise "performance quality." The first is beatmap difficulty, which is what this project seeks to quantify (and thus will be discussed at length in a later section). The second is rhythmic accuracy, which considers how well the player clicked in time to the beat of the song. Note hits in osu! are not binary (i.e., "hit or miss"); instead, there are three types of hits based on the rhythmic accuracy of the click, resulting in four total note outcomes (see below). This nuance increases the complexity of difficulty estimation because beatmaps can be easy to hit but difficult to time, and vice-versa. That said, there are no other major components of performance quality, so a satisfactory difficulty algorithm need not consider other factors.

Note Outcomes			
			
Perfect Hit	Mistimed Hit	Very Mistimed Hit	No Hit ("Miss")

1.2 Current Algorithm

Currently, osu! ranks its players by awarding each performance a number of "performance points." The number of points awarded to each performance is determined by a hand-built heuristic that estimates difficulty via basic features such as speed, spacing, and the proportion of notes hit. Although the point values aren't tied to a meaningful baseline, they do accurately represent difficulty in many cases. In some cases, though, the heuristic is very inaccurate (i.e., it over-rates an easy performance, or under-rates a difficult performance). Some notable weaknesses include:

- **Irregularity**: Common note patterns are easier to hit than their irregular counterparts. However, the simple features considered by the heuristic cannot capture this. (Patterns can be irregular in terms of geometry and rhythm, among other things.)
- **Length**: The length of a pattern can strongly influence its difficulty, but the heuristic does not consider the relevant long-range features. For example, the difficulty of a sequence of 1/16th notes scales superlinearly with the length of the sequence (which can be more than 200 notes long). Hence the heuristic often underweights long patterns.
- **Difficulty Spikes**: In determining a final performance point value, the heuristic considers note hits *in aggregate* (rather than on a note-by-note basis). This causes it to overestimate the difficulty of beatmaps with large spikes in difficulty, since in these maps, the overall proportion of notes hit is inflated by the longer, easier sections.

On top of these weaknesses, the current algorithm, being a hand-built heuristic, must be adjusted manually. This is problematic when players' skills change over time (i.e., when data drift occurs), since any update requires extensive developer attention. Moreover, the players are incentivized (by gaining ranks) to exploit weaknesses in the heuristic, so it must be frequently patched to maintain the integrity of the performance point system. Together these influences result in an algorithm that is not only inaccurate (relative to perceived difficulty), but also expensive (in terms of developer resources). Thus it can be improved along two main fronts: accuracy and maintainability.

2 Methodology

2.1 Overview

To address the problems of the current algorithm, I developed an alternative algorithm that uses machine learning to predict note outcomes. I accomplished this by parsing replay data to sequences of (note, outcome) pairs, which then supported a conventional, multi-class sequence classification task. The result is an algorithm that estimates the difficulty of a performance by computing the joint probability of its note outcomes, according to the model. Hence this algorithm defines the difficulty of a performance as the probability that the average player could replicate that performance. This is distinct from the current algorithm in that the resulting performance point values have a concrete, interpretable meaning.

I opted for this approach because in theory, it solves the problems discussed in the previous section, depending on the data and architecture used. Large ML models learn complex and long-range features, which capture irregularity and pattern length. Additionally, note-level predictions enable precise joint probability computations, which address difficulty spikes. Finally, data-driven approaches support continuous training, which increases maintainability (by automatically adjusting for data drift and exploitation attempts). Thus this problem was well-suited for a data-driven ML approach. In the following sections, I will discuss this approach at length.

2.2 Data

To develop this approach, I used a public Kaggle dataset of osu! replays. The replays originate from a third-party website called o!rdr (phonetically, “osu! render”), which renders .osu replay files to video. Because players typically use websites like o!rdr to upload their best performances to social media, this dataset is somewhat biased; rather than representing every beatmap *attempt*, the dataset represents the upper range of all players’ skill levels. The former is likely preferable for training a difficulty algorithm – the ideal dataset would contain 100/100 “bad” replays on the hardest beatmap, 99/100 “bad” replays on the second hardest beatmap, and so on – but the latter still contains valuable signal, especially when evaluating difficulty on the

note level. For example, the upper-range replays still allow the model to learn what types of *notes* cause players to miss (even if the replays themselves are not perfectly representative).

The other notable aspect of this dataset is its size, which is the reason that I ultimately chose it. The dataset is 28.8GB and contains about 400,000 replays or 300 million total notes. This size can support large (i.e., high parameter-count) models, which is crucial for the discovery of complex features. Additionally, the dataset can be downsampled without concerns about size, should bias become an issue. See below for a table summarizing the dataset.

o!rdr Replay Dataset	
File size	28.8 GB
Num. of replays	381,570
Num. of unique players	31,741
Num. of unique beatmaps	58,053
Total num. notes	309,026,976

2.3 Preprocessing

osu! replay files contain time series data of the player's actions, but not the note outcomes that result from them. Hence my first preprocessing step was to parse the replays into sequences of (note, outcome) pairs. To do this, I used Circleguard, an open-source Python package built for anti-cheat in osu!. Circleguard's parser is an imperfect reimplementation of the official osu! parser (which is closed-source), so relying on Circleguard's parser introduced noise into the dataset. However, the amount of noise is minimal (approximately 90% of the replays were parsed correctly, according to my validation script), and I did not have time to develop a more accurate parser. Thus Circleguard was the best option. Parsing and validating required all replays to be loaded into memory, so this step took approximately 25 hours of computation time.

Once the replays were parsed, I built a set of features to represent each note. osu! beatmap files do not contain a lot of information, so the number of potential features was rather small. I ultimately settled on 16 features that seemed representative. It may be possible for my feature set to be improved, as I did not test alternatives; however, provided that the data and model are large, manually engineering complex features may not substantially boost performance. Hence a minimal feature set seemed sufficient. See the table below for a summary of the features representing each note.

Note Feature	Description
x_position, y_position	note coordinates, in osu! pixels
in_x_offset, in_y_offset, out_x_offset, out_y_offset	x- and y-offset of the note, relative to previous & next notes
in_distance, out_distance	Euclidean distance between note and previous & next notes
in_timedelta, out_timedelta	time offset between note and previous & next notes, in milliseconds
cos_angle	cosine of the angle formed by previous_note -> current_note -> next_note
is_slider	Boolean indicating if the circle is a slider
slider_duration	duration of the slider, in milliseconds
slider_length	length of the slider, in osu! pixels
slider_num_ticks	number of checkpoints in the slider
slider_num_beats	number of beats that the slider spans

My last preprocessing step was to write the (note, outcome) pairs to a file format that allowed the models to be trained efficiently. To do this, I used TensorFlow Datasets. I chose TensorFlow Datasets because of a few key features. First, it integrated with TensorFlow, which is my preferred machine learning library. Second, it supported input pipelines that lazily loaded data into memory (which was necessary because my dataset was too large to fit into RAM). Finally, it included data compression, which decreased

the amount of training time spent on memory transfer. Altogether, the preprocessing steps reduced the original 29GB of replay files to 4GB of (note, outcome) pairs. The total computation time was approximately 50 hours.

2.4 Models

I used the processed data to train three probabilistic ML classifiers. For each, the goal was to predict a note outcome (300, 100, 50, or miss) based on the features of the note and notes that preceded it. In the first model, these notes were treated non-sequentially. Thus they were passed into a feed-forward neural network that performed multi-class classification. In the second model, these notes were treated sequentially. Thus the notes were passed into an LSTM that learned features for the classifier network. In the third model, the notes were again treated sequentially, but the previous note *outcomes* were also included. Thus the feature set was significantly different from that of the second model. All three models, hereafter referred to as the "naive", "sequential", and "seq2seq" models, were trained via categorical cross-entropy loss, which is the standard for multi-class classification.

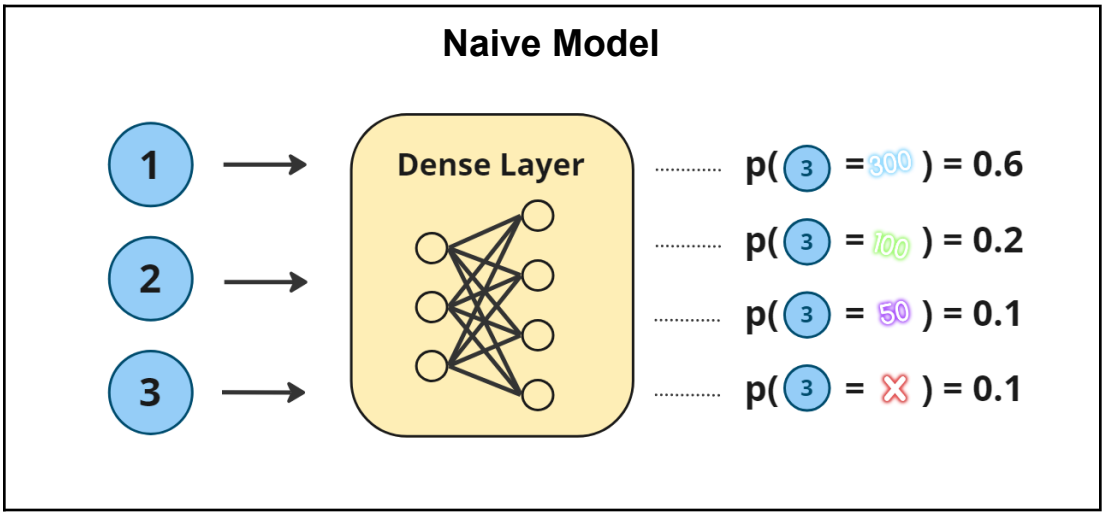


Diagram of the naive model. Note embeddings (depicted as the blue circles) are simultaneously provided to a dense neural network that classifies the note in the last position (which in this case is note three). The classification probabilities estimate the probability of each note outcome – i.e., a ***p****(300),* ***p****(100),* ***p****(50), and* ***p****(miss). Relevant hyperparameters to this model include number of preceding notes and the count and size of the dense layer(s).*

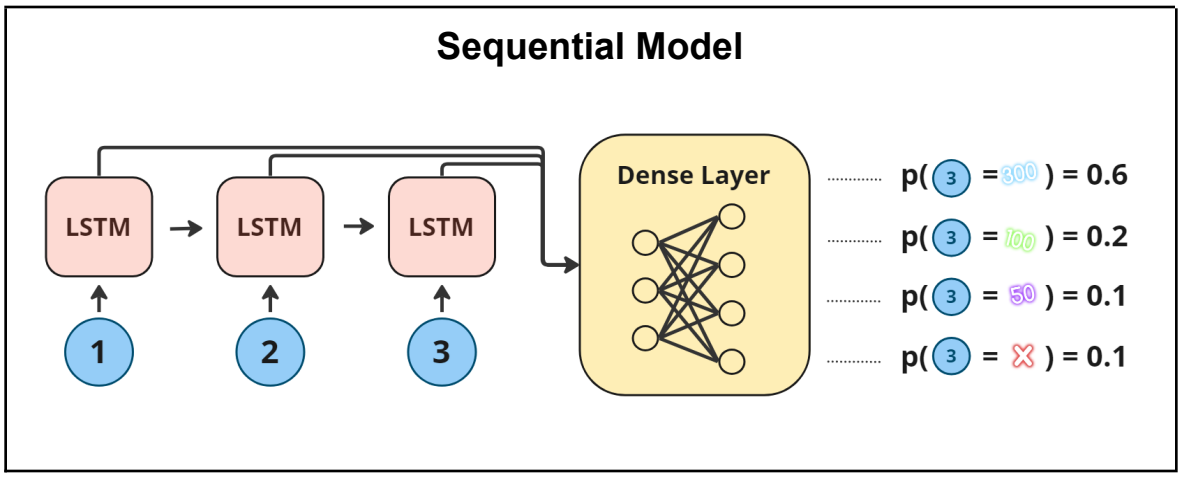


Diagram of the sequential model. Note embeddings are provided sequentially to an LSTM layer. Then, each LSTM hidden state is passed to the dense layer(s). Again, the resulting classification probabilities correspond to the probability of the outcome in the final position. In addition to the hyperparameters of the naive model, this model also requires the count and size of the LSTM layers to be specified prior to training.

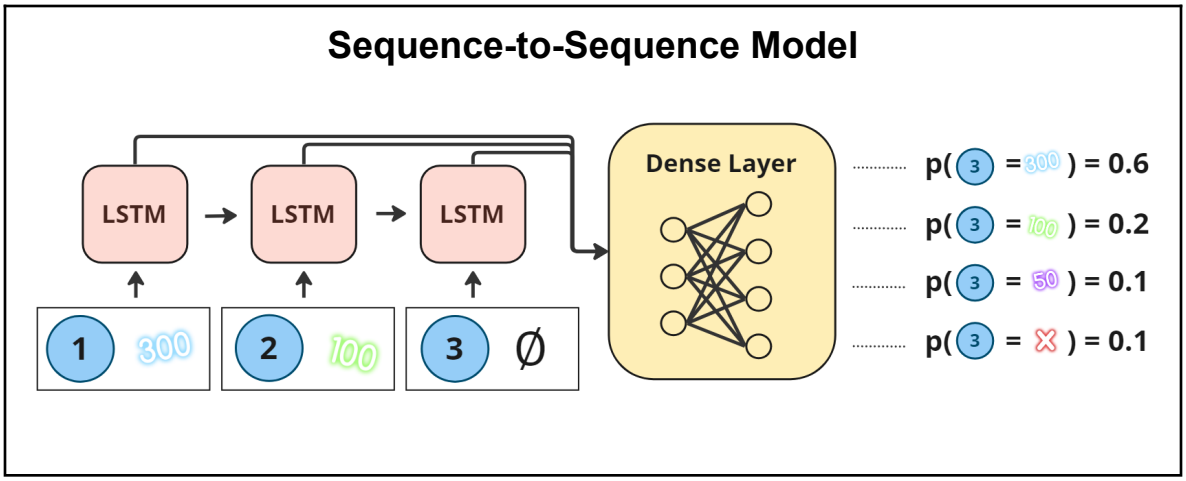


Diagram of the sequence-to-sequence model. Note embeddings and their outcomes are provided sequentially to an LSTM layer. Forward propagation proceeds identically to that of the sequential model. Note that the outcome of the note in the final position is masked (because that is what the model is trying to predict). The relevant hyperparameters are the same as those of the sequential model.

I chose to test these models for a few different reasons. The naive model was chosen as a baseline; although sequential models should generally outperform naive models on sequential data, I wanted to confirm this empirically. (It may have been the case that the sequential models were prone to overfitting, and that the naive model was superior, for example.) The sequential model was chosen because osu! data is inherently sequential, so the long-range memory gates of LSTMs were appealing. This was doubly true considering that the current algorithm struggles with long-range features. Finally, the seq2seq model was chosen because note hits are conditionally dependent events; if a player hits the first note in a pattern, they are more likely to hit the second, third, and so on. Thus previous note outcomes seemed like valuable features. However, it was unclear if the associated change in architecture would have unintended downstream effects on difficulty calculations, so note outcomes were only included in this model. Overall, the goal of considering each model was to determine the best option for learning from osu! replay data.

2.5 Training

To train these models, I followed the conventions of multi-class classification tasks. First, I partitioned my (note, outcome) pairs into three datasets, with 80% of examples used for training, 10% for validation (i.e., hyperparameter tuning), and 10% for evaluation. These partitions were stratified by the *modifications* associated with each replay, since some modifications influence difficulty and are quite rare. Then, each model was trained on 1 full epoch of the training dataset, amounting to about 45 minutes of training time per model (GPU: NVIDIA 2060 Super). This allowed for the training losses to converge in most cases. It also allowed for hyperparameter selection via the validation losses. As a final step, the models were retrained using the selected hyperparameters until the validation losses converged.

Below is a table describing the hyperparameters tested for each model. Cells containing several parameters indicate that each parameter was tested. The bolded parameter indicates the parameter that yielded the lowest validation loss. Note that the tested parameters do not exhaust the hyperparameter space (i.e., form a complete “grid search”). I restricted the search space in order to reduce training time, as certain

parameters (such as hidden layer size) had minimal effect on performance. Similarly, I only considered one optimizer (Adam with exponential learning rate decay) because its performance exceeded alternatives (e.g., stock Adam, Adam + norm clipping) while conducting informal tests. Thus I submit that the selected hyperparameters have been carefully considered, even if few alternatives are listed.

Hyperparameter	Naive	Sequential	Seq2Seq
Note count	8 (1 + 7 prev.)	64 (1 + 63 prev.)	64 (1 + 63 prev.)
Optimizer	Adam + Exp. Decay	Adam + Exp. Decay	Adam + Exp. Decay
Hidden layer count	[**5**, 6, 7]	[**4**, 5]	[4, **5**]
Hidden layer size	[**512**, 1024]	**512**	**512**
LSTM layer count	-	[1, 2, **3**]	[1, 2 ,**3**]
LSTM layer size	-	[**128**, 256]	[128, **256**]

2.6 Algorithm

Because these models are probabilistic, they produce class probabilities that represent the difficulty of a note. For example, provided that the model is accurate, a note with p(300) = 0.9 is almost certainly easier to hit than one with p(300) = 0.4. Hence the difficulty of a performance – which is simply a sequence of note outcomes – can be estimated via the *joint probability* of its note outcomes. For the naive and sequential models, this is simply the product

$$\mathbf{p}(\text{performance}) = \mathbf{p}(X_0 = x_0, \ldots X_n = x_n) \approx \prod_{i=0}^{n} \tilde{p}(X_i = x_i)$$

where p-tilde is the estimated probability and the $\mathbf{X_i}$ are independent multinomial random variables that represent the outcome of each note. The seq2seq case is similar, except that the $\mathbf{X_i}$ are not assumed to be independent (so each $\mathbf{X_i}$ must be conditioned on the previous $\mathbf{X_i}$). However, this does not affect the computation in practice, since the

seq2seq model estimates conditional probabilities (by virtue of considering previous note outcomes). So the difficulty calculation is straightforward in both cases: the probability of a performance is the product of the probabilities of its note outcomes. If this probability is low, the performance is unlikely, and is thus difficult.

There are, of course, a number of modifications that can improve this calculation. I made three:

- The first was to replace the probability of achieving a given note outcome with the probability of achieving that note outcome *or better*. This modification was almost necessary; 100s and 50s are many times rarer than 300s, but are also easier to achieve. Hence I replaced any instance of p(100) with p(300) + p(100), any instance of p(50) with p(300) + p(100) + p(50), and so on. A 300 is always preferable to a 100, so this did not adversely affect the algorithm.
- The second was to compute the negative log-probability of a performance (instead of just its probability). This allowed the probability product to be replaced by a sum of log-probabilities, avoiding the challenge of handling small floats. Also, the negative sign aligned the algorithm with the current algorithm in that larger values corresponded to higher difficulties.
- The third was to adjust for beatmap length by only considering the top 256 least-likely note outcomes. Although the models should theoretically predict high (i.e. near-one) hit probabilities for filler patterns, I did not have time to test this extensively. Hence the best way to ensure fair beatmap comparisons was to threshold note count. I chose 256 somewhat arbitrarily (there was certainly a case for other values) but this value did allow for short and long beatmaps to have comparable difficulty estimates.

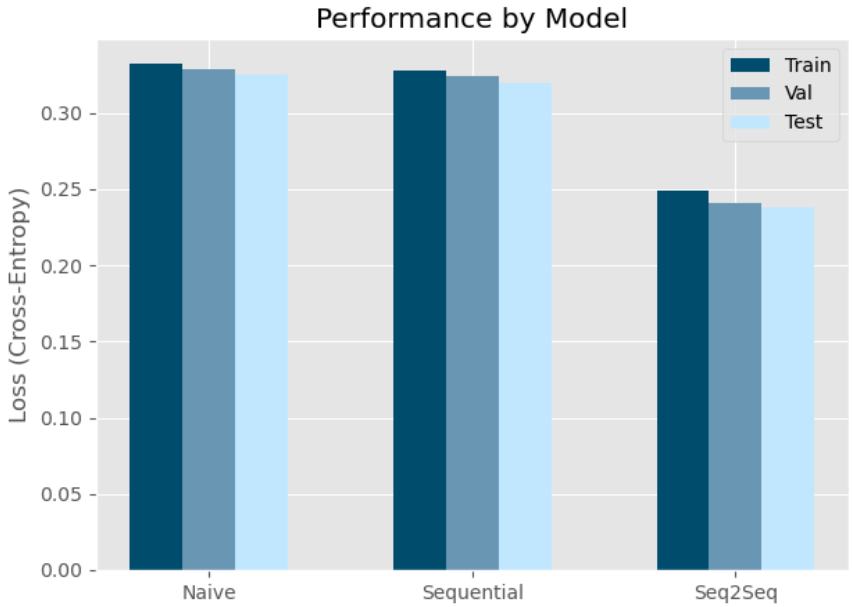
The modifications resulted in my final difficulty algorithm, which is given by

$$-\log \mathbf{p}(\text{performance}) \approx -\sum_{i=0}^{\min(\ell,\, 256)} \log \tilde{p}(X_i \geq x_i)$$

where the p($\mathbf{X}_i$) are sorted in descending order and $\boldsymbol{\ell}$ is the beatmap length. In this notation, $\mathbf{X}_i$ = 0 corresponds to a miss, $\mathbf{X}_i$ = 1 corresponds to a 50, $\mathbf{X}_i$ = 2 corresponds to a 100, and $\mathbf{X}_i$ = 3 corresponds to a 300.

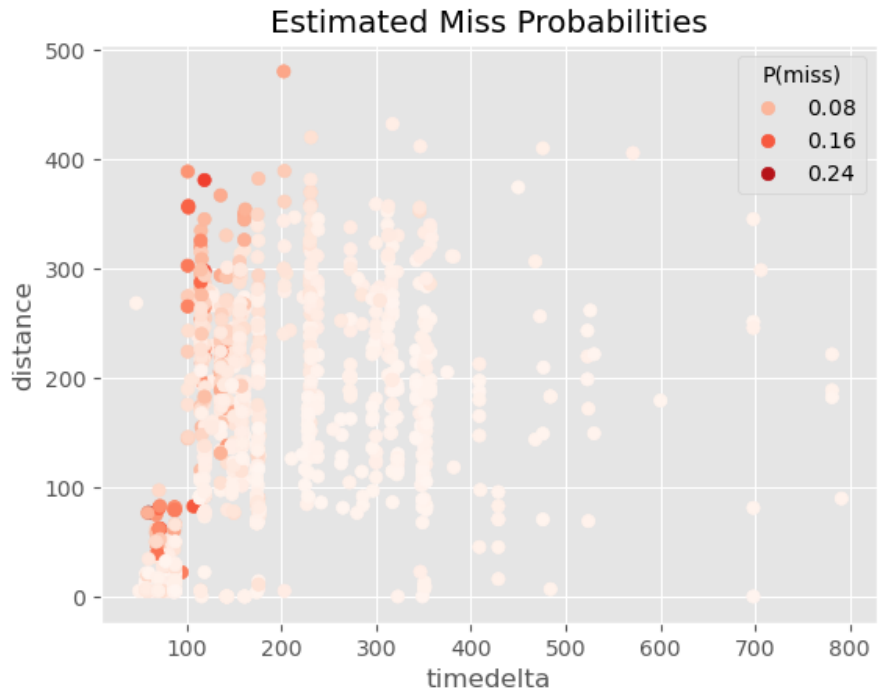
3 Results

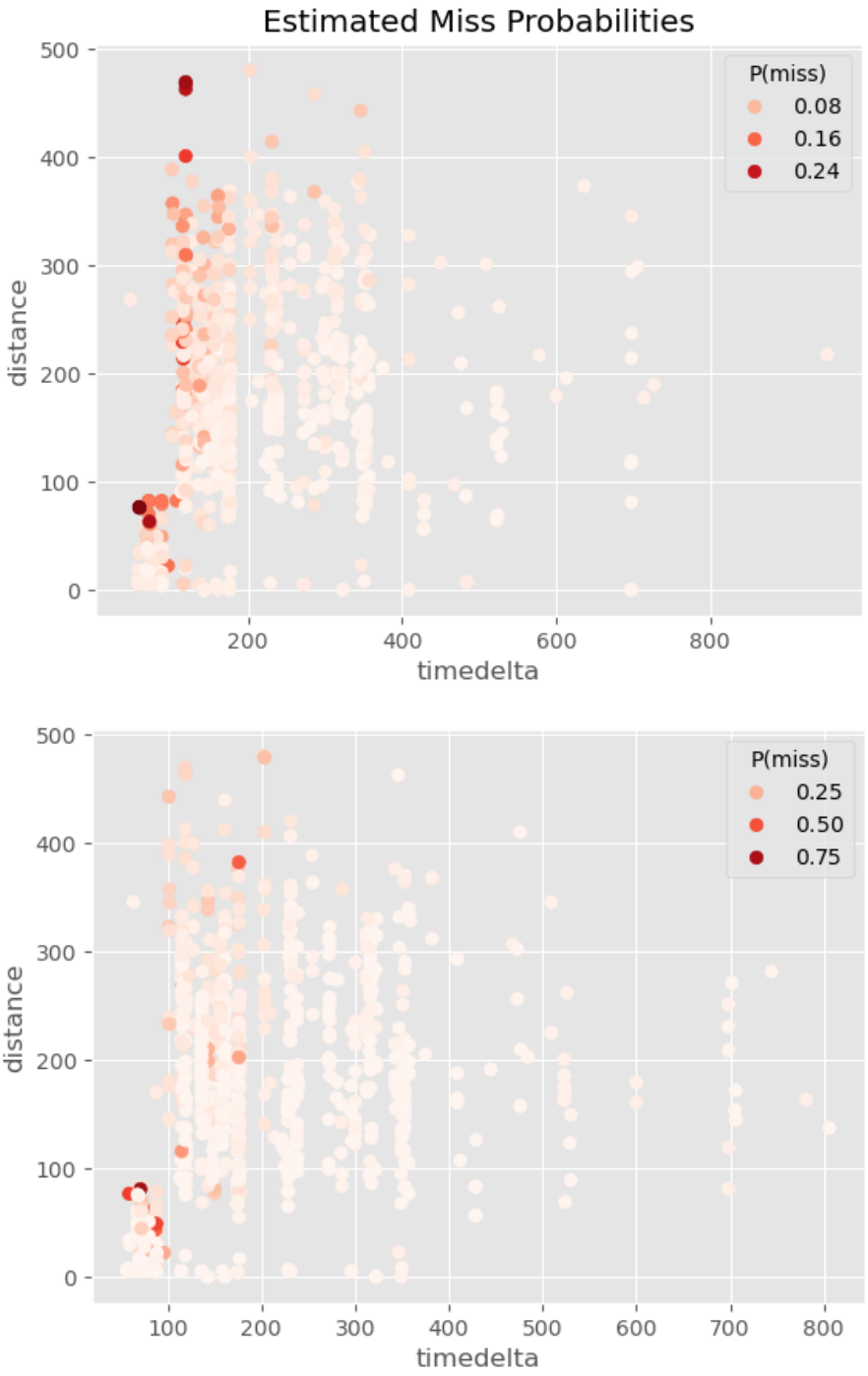
3.1 Model Performance



Bar chart depicting model performance on the train, validation, and test datasets.

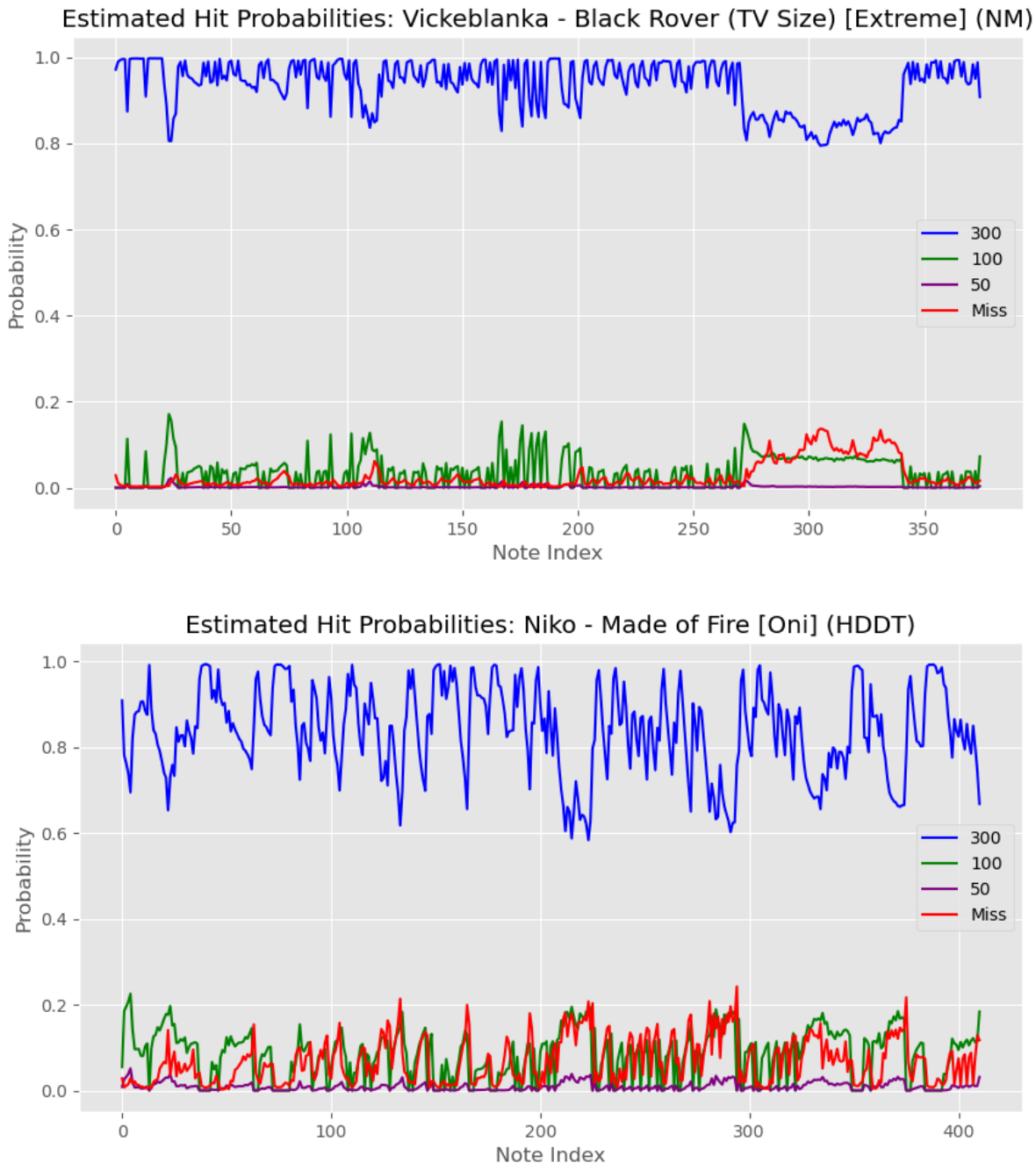
3.2 Note-Level Estimates



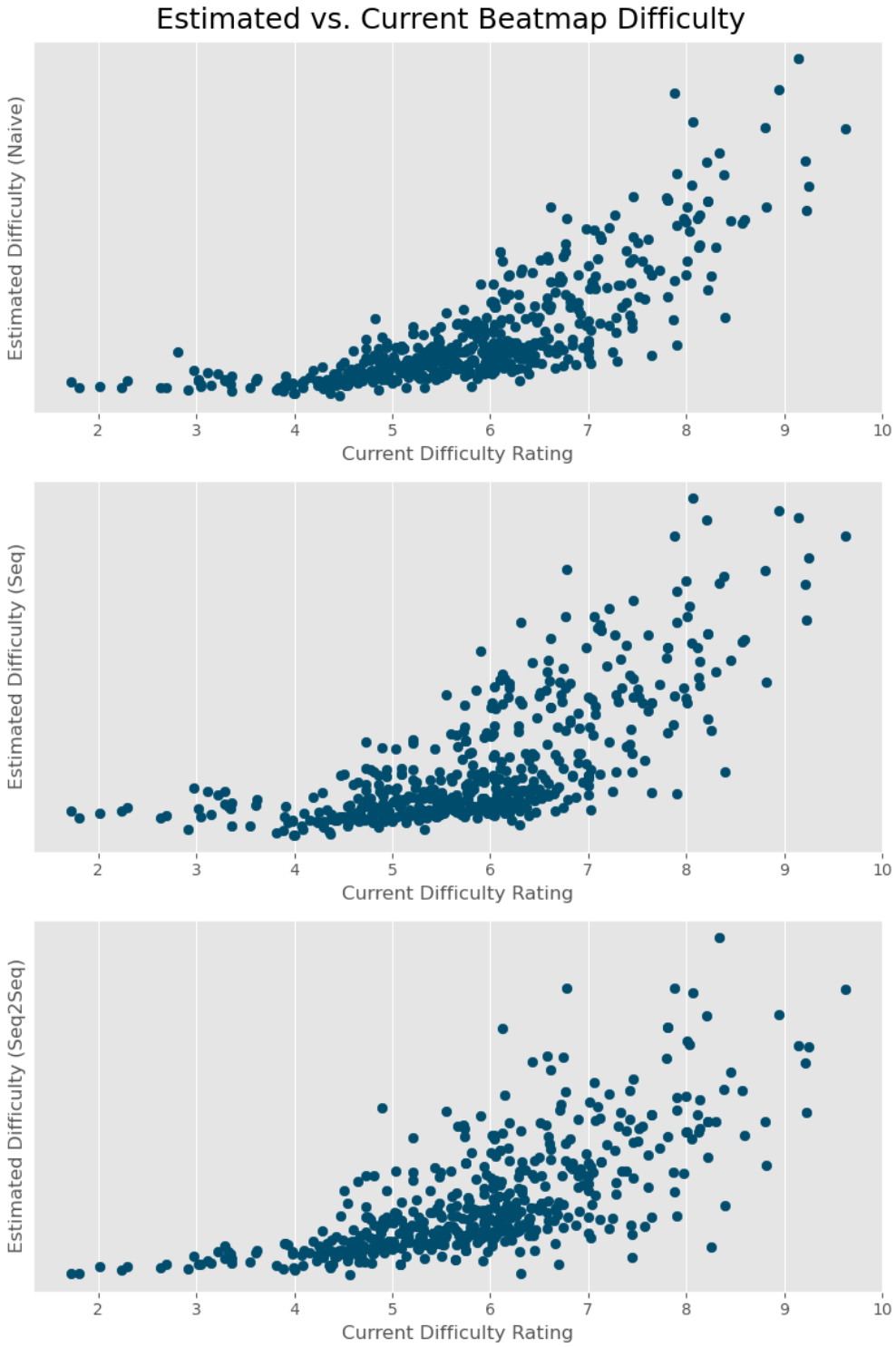


Scatter plots representing estimated miss probabilities generated by the naive, sequential, and seq2seq models. Each circle represents a note. The color of the circle represents the estimated miss probability of that note. The x-axis is the time (in ms) between the note and the previous note (i.e., the time the player has to click the circle). The y-axis is the distance between the note in the previous note (in osu! pixels). Note the change of scale in the bottom plot.

3.3 Beatmap-Level Estimates



Line plots depicting estimated outcome probabilities over the course of two beatmaps. The y-axis is the estimated probability according to the sequential model. The x-axis is the index of the notes in the beatmap. The beatmap depicted in the upper plot is known for having a large difficulty spike in its final section. The beatmap depicted in the lower plot is known for being consistently difficult.



Scatter plots depicting estimated beatmap difficulty (according to my algorithm and the three models) versus the current difficulty ratings. Each circle is a beatmap. Beatmap difficulty was calculated using the 256 hardest notes to hit (i.e., not miss). For the seq2seq estimates, all notes were assumed to be preceded by 63 300s.

3.4 Example Leaderboard

Rank	Player	Beatmap	Current Difficulty Estimate	Seq2Seq Difficulty Estimate
1	Accolibed	Reign of Fear [VI]	1378	84.524
2	Utami	Songs Compilation II [Can You See The Distant Journey Awaiting Us]	1036	59.473
3	shimon	United (L.A.O.S Remix) [Eternity]	1066	57.456
4	Utami	United (L.A.O.S Remix) [Eternity]	1048	57.184
5	mrekk	Seijouki no Pierrot ~ The MadPiero Laughs [Extra Stage]	1315	56.123
6	Accolibed	Sidetracked Day [Infinity Inside]	1711	53.759
7	Utami	Glory Days - [Maki's Extra]	1015	52.515
8	Freddie Benson	United (L.A.O.S Remix) [Eternity]	1031	52.028
95	BlackDog5	Team Magma & Aqua Leader Battle Theme (Unofficial) [Catastrophe]	1102	9.488
96	Benthonic	Hello Zepp [Twist]	349	8.648
97	Freddie Benson	Yubi Bouenkyou (TV Size) [Fate]	1030	8.540
98	BlackDog5	Imagination (TV Size) [Ambition]	1073	8.523
99	shimon	10 Things I Hate About You (Sped Up & Cut Ver.) [Mommy Issues]	1058	8.007
100	Benthonic	Marry You [Insane]	350	7.726
101	gnahus	Imagination (TV Size) [Ambition]	1166	5.410
102	maliszewski	PADORU / PADORU [Gift]	1015	4.316

A performance leaderboard generated by the seq2seq model and the algorithm described in 2.6. A total of 102 performances were considered: the five best performances from each of the top 20 players (under the current system), and two of my performances (as a control). The eight best and eight worst performances are shown above. My performances are highlighted in yellow.

4 Discussion

As the results illustrate, the algorithm developed in this report is far from perfect. However, portions of my methodology may still be viable, as the bias identified in (2.2) appears to have skewed the models' estimates. To explain why I believe this is the case, I will discuss each portion of the previous section. I will also comment on the notable (and perhaps tangential) aspects of each result. Finally, I will conclude by offering thoughts on how the modeling and data collection can be improved.

Beginning with model performance, it is clear that the seq2seq model outperformed the other two models, which performed equally well. It is not surprising that the seq2seq model achieved the best performance, because intuitively, note outcomes are not independent — all outcomes are affected by the player's skill level. However, it is surprising that the sequential model barely outperformed the naive model (by 0.006). I initially hypothesized that the sequential model would learn long-range features that would substantially increase performance, but this does not seem to be the case. A possible explanation for this is the rarity of long patterns; it might be that in the majority of cases, the difficulty of a note can be explained by only a few preceding notes (meaning that the increased context window of the sequential model has limited value). This suggests that the naive and sequential models would be comparable second choices in the event that the seq2seq model produces poor difficulty estimates.

(To understand why the seq2seq model could produce poor estimates, consider a beatmap performance in which a player misses easy notes before hitting a difficult pattern. Such a performance might "surprise" the seq2seq model more than hitting all of the notes, but hitting all of the notes would certainly be more impressive. Hence a model that does not consider previous note outcomes may be preferable.)

On the note level, the models appear to have learned the qualities that make a note difficult to hit. For example, all three learned that quick, spaced notes correspond to an increased probability of missing (i.e., an increased difficulty). The estimated miss probabilities also decreased smoothly as speed and spacing decreased, suggesting that the models learned reasonable relationships instead of unintuitive, rule-based relationships (which might lead to sensitive estimates). That said, it does seem that the models' classification probabilities fell into a small range (or were "compressed"). This is

concerning because it suggests that the models were unable to confidently distinguish between easy and difficult patterns – ideally, the most difficult notes would have very high miss probabilities (< 0.8) and the easiest notes would have very low miss probabilities (< 0.2). The fact that this was not the case implies that the bias identified in (2.2) was problematic. Compressed classification probabilities indicate that the training dataset contained too many upper-range replays (i.e., the replays were “too good,” even on the harder beatmaps).

The first set of beatmap-level results tells a similar story in that the classification probabilities were reasonable relative to each other, but were not reasonable on an absolute scale. Specifically, the models were able to relate the difficulties of notes within the same beatmap — as evidenced by being able to identify difficulty spikes — but were unable to determine reasonable probabilities for each note. This is especially evident when considering that the upper beatmap in the first set of plots has thousands of perfect performances, while the lower beatmap has exactly one. Hence the models’ estimated miss probabilities, which were similar in magnitude across the two beatmaps, were unreasonable. This finding again suggests that the bias identified in (2.2) was problematic. It seems that the models learned how to identify the most difficult *portions* of the beatmaps, but not the absolute difficulties of their notes.

The second set of beatmap-level results illustrates that my algorithm aligned with the current algorithm. Considering the differences in methodology between the two algorithms, this result is somewhat surprising. In some ways, it suggests that the methodology described in this report is theoretically sound. Compression was still a huge problem in practice, though; under no circumstances should a four-star beatmap be considered as difficult as an eight-star beatmap, yet this seemed to be common. Curiously, this problem seemed to worsen with model complexity, as the naive model’s estimates best tracked the current algorithm. This is especially notable considering that none of the models overfit (as evidenced by the performances on the test set). Thus the underlying data must not have been representative. Lastly, it also appears that the estimates were heteroskedastic, with difficult beatmaps receiving higher-variance estimates. A possible explanation for this is that the harder beatmaps were not only less

common in the training data, but were also played by a wider range of players (i.e., both high-ranked and low-ranked players). Either way, heteroskedasticity is not ideal.

The example leaderboard summarizes the previous results in a concrete way. I included two of my own replays to demonstrate the severity of the compression – it is unacceptable for my replays to be anywhere but the bottom two positions of a top-20 leaderboard. Hence the seq2seq algorithm is unacceptable. The leaderboard also indicates that a different transformation of the note-level probabilities may be appropriate, as the top-ranked plays are nowhere near twenty-times as difficult as the bottom-ranked plays. But this is an aesthetic concern that can be addressed after the algorithm's other problems.

Going forward, I intend to apply the proposed methodology to a more representative dataset. Of course, obtaining such a dataset will not be straightforward (as even osu!'s backend only saves the best replays on every beatmap). But regardless, I believe that the methodology described in this report is sound. My first reason for this belief is that the models learned accurate intra-beatmap estimates, as evidenced by the recognition of difficulty spikes and unusual patterns. My second reason is that the inter-beatmap estimates aligned with the current algorithm, even though the current algorithm is distinct (and based primarily on intuition). My third reason is that the compression cannot be explained by underfitting or overfitting. Because the models' performances generalized to the test sets, and because compression appeared to worsen as performance increased, it seems that the source of compression was the data rather than the methodology. Hence I am more eager to experiment with alternative datasets than models.

There are a number of options for assembling a more representative dataset. One option that has been discussed within the osu! community is to gather replays from third party servers. This option is appealing because third party servers often store more replays than the official osu! backend. However, this option is somewhat risky due to the prevalence of cheated replays, which could reintroduce bias. Another option is to conduct a survey among osu!'s top players. This option is appealing because it would allow for finer control over the type of replays that are collected. For example, the community could make an official "test beatmap" of relevant or unusual patterns to send

to top players. However, this option would require substantial buy-in from osu!'s top players, and would likely result in an algorithm that is relevant to a small segment of osu! players. A third, ideal option is to send a survey to *all* osu! players via the official client. This option would have all of the benefits of the second option without any of its drawbacks, as the resulting data would likely result in an algorithm that is relevant to all players. The survey could even be disguised as a feature, such as a gamemode, calibration test, or beatmap recommendation system. It goes without saying that this option would require substantial buy-in from osu! developers.

There are also a few aspects of the methodology that could benefit from reexamination. As mentioned earlier, the seq2seq model may ultimately be impractical due to its consideration of previous note outcomes. It would therefore be prudent to conduct formal sensitivity testing in order to determine whether or not this is the case. Additionally, the transformations applied to the note-level outcomes will probably need to be adjusted at some point, since the scales of the estimates were not particularly intuitive. The note-count cap of 256 was also hastily chosen. But for all of these aspects, I submit that efforts may be better spent on the dataset. It may be that a representative dataset is all that is necessary to develop a viable algorithm.

5 References

Hochreiter, S., & Schmidhuber, Jurgen. (1997). Long short-term memory. Neural Computation, 9(8), 1735–1780. https://dl.acm.org/doi/10.1162/neco.1997.9.8.1735

I Gold, K., & Olivier, A. (2010). Using Machine Translation to Convert Between Difficulties in Rhythm Games. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 6(1), 27-32. https://doi.org/10.1609/aiide.v6i1.12396

6 Acknowledgments

I would like to thank James Glenn for supporting this project. I would also like to thank Liam “tybug” DeVoe, a graduate student at Northeastern, for supporting my efforts to parse 29 GB of replay files. I would not have accomplished much if I had to write my own parser from scratch.

Lastly, I would like to thank my family. I sometimes forget about my family while I am absorbed in my work. It’s something I often regret.