Predicting Player Difficulty in 2D Platformers
Martin Jennings-Teats
CMPS 242
December 9, 2009

Abstract
This report details my attempts to predict the difficulty that the average player would experience given a procedurally generated level segment from a 2D platformer video game. I describe the broader motivation for this project, including its desirability as a game design feature and some of the prior work on the topics of dynamic difficulty adjustment and procedural level generation in games. I also describe the data collection tools that I employed and the algorithms I used on the collected data. I conclude with the results from the work on this project and the additional work that I plan to complete in the future.

Motivation
This project is one step in the creation of a larger research project, the final product of which will be a playable (and hopefully fun!) video game. 2D Platformers are a genre of video game in which the player controls a character which they must move across a 2-dimensional, side-scrolling game level to some ending location. Typically the obstacles that the player encounters along the way include gaps that must be jumped over, enemies that must be defeated or avoided, and various impediments to movement such as moving platforms or falling obstacles (called “thwomps” for the remainder of this report). Super Mario Bros. and Sonic the Hedgehog are two well-known games in this genre.

The game that I aim to create will be unique in that the game level ahead of the player will be procedurally generated to fit the skill level of the player, providing a constant, but not overwhelming, challenge. This is an idea known as dynamic difficulty adjustment, and in commercial games it is usually accomplished by tweaking some parameters of the physics simulation or AI in the game. But instead of giving an enemy a slight speed boost or improved pathfinding, this game will adjust the difficulty by dynamically generating new level segments for the player. The entire game world, not just a single enemy, will be radically altered specifically to match the player's skill level. Some notable examples of commercial games that have used dynamic difficulty adjustment are Sin, which altered the AI controlling the enemies to give them improved teamwork and other intelligent behaviors as an added challenge for skilled players, and Left 4 Dead, which makes some small alterations to the level layout by changing the locations of enemies to adjust the level of challenge.

Dynamic difficulty adjustment is a desirable feature of game design, as is shown by its use in the aforementioned popular games. In Csikszentmihalyi’s influential paper Flow: The Psychology of Optimal Experience, he explains how an optimal experience in a video game maintains an increasing level of challenge, so that as the player improves her skills the game continues to keep up [1]. This can be accomplished through good game design, in which challenges are carefully hand-placed and tuned by a human designer, but this often will not fit players of all skill-levels. However, if the computer is intelligently designing the challenges of the game as the player progresses, it would be able to detect the player’s performance and compensate accordingly.

As I already hinted at earlier, generating content (such as the design of a level) while the human plays is another desirable quality for game design. As Smith points out in her 2009 paper Rhythm-based level generation for 2D platformers, level design in commercial games requires a very large amount of resources, so having the computer procedurally generate the levels could be very beneficial to the video games industry [2]. Compton and Mateas also say that procedural level generation
improves the player experience through nearly endless variation and replayability that is not present in hand-designed levels [3].

The closest work to this project is Pedersen's 2009 paper *Modeling Player Experience in Super Mario Bros.*, in which he uses neural networks in an attempt to predict the player's ranking of fun-ness of a level based solely on the average width of gaps in the level [4]. His work is inspirational, but the results leave a lot of room for improvement, since the feature space was so limited, and the level generation was at a full-level scale rather than the finer granularity that I am using.

![Figure 1: An example image from my game.](image)

**Data Collection**

The data I collected for this project comes from people playing a modified version of the game as shown in figure 1. The data collection tool (which can be played at http://users.soe.ucsc.edu/~mjennin1/segments/PlayTestLevelSegment.html), generates a short level segment, which the player will likely complete in five to ten seconds, followed by a survey question asking the player to rank the level they just played on a one to six, easy to difficult, scale. The answer to this question becomes the label for the data example. I received 283 data points, each from the playthrough of a different level segment. The features are primarily collected from the level generation process, though some data is also collected based on the player's behavior. Some example features are: the number of enemies in the level, the number of upward jumps over a gap, the average width of gaps, the number of coins, the amount of time the player spent standing still and moving backwards, etc. There are 24 features overall. Because the features do not all have values with the same units, or even the same types, I normalized each feature so that it summed to 1. This way I could use different types of features together, such as the total milliseconds taken to complete the level and whether or not the player died, and the weights will still make sense.

**Multiclass Logistic Regression**

In order to learn to predict the difficulty score that an average player would assign to a level segment, I used multiclass logistic regression with batch gradient descent on the data I collected. Mutliclass logistic regression differs from regular logistic regression by using more than two possible labels. So, the $y$ that is used as the input label is a vector of K classes, with all the classes set to zero
except for the label which is set to 1. This means that there is not only a label for each class, but there is also a weight vector for each class and a probability prediction for each class.

\[
\begin{align*}
    w_{t,i}^j &= w_{t-1,i}^j - \eta \frac{1}{N} \left( \sum_{n=1}^{N} \left( y_{n,i} - \hat{y}_{n,i}^j \right) x_{n}^j + \lambda * w_{t-1,i}^j \right) \\
    \hat{y}_{n,i}^j &= \frac{e^{d_{n,i}}}{\sum_{j=1}^{K} e^{d_{j}}} \\
    \eta & \quad \text{learning rate} \\
    \lambda & \quad \text{regularization term} \\
    K & \quad \text{number of classes} \\
    N & \quad \text{number of examples}
\end{align*}
\]

Figure 2: The update used for the weights.

I split the data into three quarters for the training set and one quarter for the test set. I then repeatedly ran the gradient descent step to update the weights (figure 2), stopping when the one-norm of the gradient reached the predefined precision value. Once the gradient reached the precision value, I determined that the weights were sufficiently trained on the training set. I then evaluated the trained weights on the test set.

I also ran the regular, binary-labeled version of logistic regression. I converted the labels into binary values by splitting them in half and calling everything 3 or below 'easy' and everything 4 or above 'hard'. By also running this 2-class version of logistic regression, I am able to see the accuracy trade-offs that come from the more precise predictions of multiclass algorithms.

Results

For gradient descent's stopping precision threshold, I tried several different values for comparison. As the precision value was decreased, causing the gradient descent step to run more times, I found that the average logistic loss over all the examples in the test set also decreased up to a point, after which the loss seemed to flatten out. For the 2-class case, the loss seemed to continue to decrease.

Multiclass Logistic Loss

Figures 3 and 4:

2-Class Logistic Loss (Difficult & Non-difficult)
past the point at which it flattened out for the multiclass version, as shown in figures 3 and 4.

The stopping condition for the multiclass case of gradient descent is not obvious. Since the one-norm of the gradient is a vector of the one-norm gradient for each of the classes. Stopping the gradient descent as soon as the one-norm of the gradient of one of the classes reached the precision value was not optimal. I also changed the stopping condition to require the one-norms of more of the classes to reach the precision value before stopping. As I required more of the gradients to reach the precision value, the average logistic loss over the examples in the test set improved, as shown in figure 5.

![Multiclass Gradient Descent Stopping Conditions](image)

Figure 5: Changing the number of required classes for the gradient descent stopping condition

To evaluate how well the algorithms did on the test set, I calculated the accuracy, the average difference between the most strongly predicted class and the actual label, and the average logistic loss. Table 1 shows these values for both the multiclass logistic regression and the binary-label logistic regression. The accuracy, which is the percentage of predictions that give the largest probability to the class that is the same as the actual label, is obviously much higher for 2-class regression since choosing randomly would give 50% accuracy, while choosing randomly for the 6-class case would lead to 17% accuracy. The average difference between the prediction with the largest probability and the actual label is a figure that I thought would be interesting to record, though it is not as good of an evaluation metric as the mean logistic loss, since the algorithms were motivated by the logistic loss function in the first place.

<table>
<thead>
<tr>
<th></th>
<th>% correct</th>
<th>mean (ŷ - y)</th>
<th>mean logistic loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiclass Logistic Regression:</td>
<td>52.11%</td>
<td>0.66197</td>
<td>1.3738</td>
</tr>
<tr>
<td>2-class Logistic Regression:</td>
<td>94.37%</td>
<td>0.056338</td>
<td>0.74974</td>
</tr>
</tbody>
</table>

Table 1: Results of evaluation on the test set
Future Work

This project could be improved by spending some more time tuning the parameters of the algorithms. Cross-validation would have helped me improve my algorithms significantly I think.

As this is one part of a larger project, I plan to improve upon these findings with future data collection and the application of some different learning algorithms. I would really like to have some better features that represent the intricacies of the game levels more accurately. For instance, I would like to keep track of the order in which different components appear in the level, and maybe the distances between the components. I don't think working with positions would be plausible, but maybe the inter-component distances would be more useful.

As well as improved features, I would like to try out different algorithms for predicting the difficulty rating. For instance, I didn't look at any machine-learned ranking algorithms, but from reading a brief overview of the techniques it seems like a useful path to pursue in the future.

Thanks Maya and Manfred for your help and feedback on the project.

References


