Crawl-O-Matic-O-Matic

Automated Optimization of Expert Systems With Machine Learning

Eli Ramthun
Department of Computer Science
Earlham College
Richmond, Indiana
ebramth15@earlham.edu

ABSTRACT

This project involves utilizing machine learning techniques to attempt to optimize an already existing artificial intelligence (AI) agent that plays the roguelike game Dungeon Crawl Stone Soup (DCSS). An expert system based AI agent has been developed that is capable of winning a game of DCSS without human assistance with a low frequency of success. This project aims to utilize machine learning to automatically tune the agent to improve its performance by optimizing values stored in its knowledge base concerning monster threat levels.

KEYWORDS

Roguelikes, Machine Learning, Artificial Intelligence, Software Optimization, Expert Systemsu

1 INTRODUCTION

Handcrafted, heuristically developed artificial intelligence (AI) algorithms currently exist for numerous applications. From gaming software to automated drone flight to assembly-line robotics, these algorithms are more and more becoming driving forces of modernity, enabling and enhancing industrial society from the production of goods to their consumption, and everywhere in between. [12] [5] [20]

These algorithms are often effective in completing their tasks, but are not necessarily optimized. This lack of optimization can have a broad swath of effects - from negative effects as minimal as slightly reduced movement efficiency to outright catastrophic failure. Manually tuning algorithms for peak performance can be a time intensive procedure, with the risk that the potentially marginal gains from optimization will not be worth the time invested to do so. However, if the process of optimization can be automated, then the time required to effectively optimize code can be greatly improved. [27]

This project investigating the utilization machine learning techniques to optimize an already existing AI agent, or expert system that plays the roguelike game *Dungeon Crawl Stone Soup* (DCSS). [15] Roguelike games feature high levels of randomness and unpredictability in each play through, creating a search space much larger than that of static games such as Chess, or Go, where the game is the same each time it is played. [7]

Despite the enormous complexity and stochasticity of DCSS, an AI has been developed that is capable of winning a game of DCSS named *qw*. [18] *qw*'s frequency of success at completing the entire game is low, though still notably higher than the average success rate of human players playing the game - 17% versus 0.73%. [18] [25] This project aims to optimize *qw*'s behavior for a specific section of the game to improve its performance. The optimization will be performed by using machine learning techniques to find the most appropriate values for a section of *qw*'s knowledge base. If successful, this project will showcase a technique for refining AI's for more efficient and/or successful operation in stochastic or unpredictable contexts.

Next, related works will be discussed. Following that, the design and implementation of the project will be documented, followed by results and analysis.

2 RELATED WORK

This paper focuses upon the optimization of a game playing agent for the roguelike game Dungeon Crawl Stone Soup. In accordance, it is important to examine both the history and development of gameplay agents for roguelike games. Attempts have been made to optimize gameplay agents for games both roguelike and otherwise - often utilizing machine learning - and these attempts at optimization are worth investigating as well. Finally, works based on applications of expert systems similar to roguelike gameplay agents will be discussed.

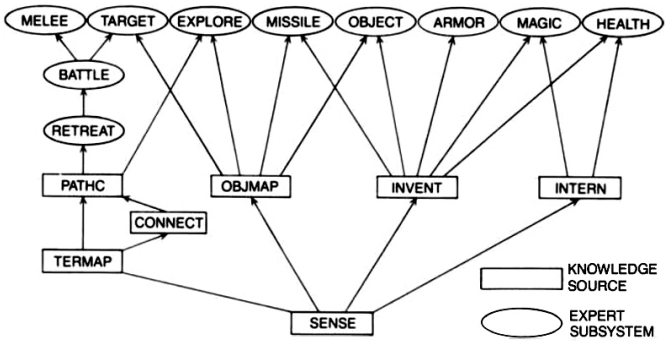
2.1 Gameplay Agents for Roguelike Games

Gameplay agents exist that can conquer many of the games within the roguelike genre, despite their renowned difficulty, complexity, and inherent stochasticity. This paper catalogs an attempt to optimize *qw*, an agent created to beat DCSS. Following are brief descriptions of agents that are successful at beating the roguelike games Rogue, Angband, and Nethack.

- **Rogue and Rog-O-Matic**
  In 1980, Rogue, the titular *rogue*like, was released. In 1981, *Rog-O-Matic* followed. The source code for Rog-O-Matic has been preserved by various archivists and is currently hosted on Github, among other places. [26] Designed as an AI tooled to utilize an "expert system" so as to conquer Rogue, *Rog-O-Matic* excelled at the task. As documented by popular media representations such as "An Expert System Outperforms Mere Mortals As It Conquers The Feared Dungeons Of Doom" found in an issue of *Scientific American* from 1985, "*Rog-O-Matic* had a higher median score than any of the 15 top Rogue players at" Carnegie Mellon University. [16] The expert systems organized the decision making processes needed to defeat Rogue into a "hierarchy of various subsystems," depicted in figure 1. [16] Rogue in the early 1980s was a much simpler game to attempt to solve than DCSS is today. Regardless, the success of the developers of *Rog-O-Matic* shows that expert systems adept at winning

games that undergo such permutations as DCSS are well within reach.

Figure 1: A schematic depicting the layout of the "expert system" of *Rog-O-Matic*, the first gameplay agent to defeat the game Rogue. Knowledge sources feed into expert subsystems which ultimately handle the decision making processes of the agent.



- **Angband and Borg**
  Angband is a Roguelike game that has been in development since 1994, though it has inherited code from older roguelike games such as Moria and UMoria. It has since spawned numerous variants featuring different dungeon styles and playable character archetypes. [1] One of Angbands defining features is non-persistent dungeons; leaving a floor of the games dungeon will effectively destroy it. While a character can, say, enter the first floor of the dungeon, leave, and then enter the first floor again, a new layout with new monsters, items, and traps will be generated each time a given depth is entered. This allows for an effectively unlimited amount of resources in the game, if the player follows a strategy known as "level-scumming." [6] This strategy is easily exploitable by automated gameplay agents, and is largely responsible for the success of *Borg*, the premier Angband bot. By abusing such a strategy, *Borg* is able to proceed through the game safely and effectively. *Borg* functions by following an internal list of goals, evaluated sequentially until the game is beaten. These goals range from very simple, such as destroying worthless items in the characters inventory, to very complex, such as determining the optimal set of equipment to wear given the current contents of the characters inventory. [2] This goal listing functions similarly to the expert systems of *Rog-O-Matic* - handing off information about inventory contents to functions designed to sort and prioritize them, for instance - and ultimately, this goal setting and following paradigm is what makes *qw*, the DCSS bot, tick.
- **Nethack and BotHack**
  NetHack is a famous and incredibly complicated roguelike under development since 1987. [11] Renowned for its immense amounts of items, creatures, features, and interactions between all of the above, the phrase “The devs think of everything” has become enshrined in the canon of the game’s culture. [3] With that complexity in mind, creating a bot able to “ascend” (beat the game) unassisted seemed like a pipe dream for many years, until Reddit user duke-nh showed their success with *BotHack* in 2015. [17] Similarly to the Angband *Borg* tactic of level-scumming, *BotHack*’s success is at least partially reliant on the arguably exploitative tactic of "pudding farming" to generate an infinite amount of items for the character to utilize, trivializing many aspects of the game. This bot was programmed utilizing some knowledge sources from previous NetHack bots *TAEB* and *Saiph*, such as monster and item dictionaries, but the expert subsystems those sources fed into were hand coded by duke-nh. Thus, this bot again follows a similar paradigm to previous expert systems designed to beat roguelike games - encyclopedic knowledge of possible threats and tools, combined with hand-coded procedures for solving the various game states the bot is bound to encounter. This bot is often successful at achieving its task, but in the words of its developer, “all the bot logic is hand-coded.” [17] This shows that this bot as well has room for application of machine-learning based optimization.

2.2 Optimization of Gameplay Agents

Creation of a gameplay agent for a given game - particularly a game featuring procedurally generated content - is often an impressive feat. Optimization of these agents goes further, allowing for improvement of performance over a number of different possible axes, from speed to score to success rate.

- **Rogue-Like Games as a Playground for Artificial Intelligence - Evolutionary Approach**
  This paper by Cerny et. al [14] specifically deals with issues like procedural content generation and their effects on algorithmic attempts to beat roguelikes. They describe their procedure towards creating an AI that has reached a 72% winrate at the roguelike game Desktop Dungeons - a simpler game than DCSS, but one featuring complex rule sets and procedural content generation nonetheless. The authors document their usage of recombination of greedy strategies using evolutionary algorithms and demonstrate a remarkable improvement over their initial hand made greedy algorithm - which was only able to achieve victory 1% of the time. [14] This shows that machine learning approaches can have enormous potential in optimization of even rudimentary hand designed algorithms.
- **Maximizing Flow as a Metacontrol in Angband**
  In their 2015 paper, Mariusdottir et. al utilize hierarchical reinforcement learning to match different decision making strategies with different levels of difficulty that arise naturally from the many different possible game states of the roguelike Angband. [24] They demonstrate the applicability of machine learning strategies towards optimization within the genre of roguelikes when applied not only to single decisions, but overall strategy decisions. They sought to optimize player score as a metric, attempting to push the Angband *Borg* to complete content as quickly as possible while still surviving. While their new metacontrol strategies

failed to produce success rates at clearing the game any better than the default configuration of the *Borg*, they showed a clear improvement between a handcrafted metacontrol strategy selection algorithm and a similar algorithm that was tuned utilizing machine learning techniques. This again demonstrates the capability of machine learning to optimize gameplay strategies for roguelike games.

- **Learning Combat in NetHack**
  In this article by Campbell et. al, the authors describe "a machine learning approach for a subset of combat in the game of NetHack". [13] They utilize a "dueling double deep-Q network" to attempt to optimize what they term "low level decision making" in the complicated roguelike NetHack. By pairing a player agent against a varied combination of different significantly threatening monsters and using a machine learning toolkit to analyze the results, the authors were able to demonstrate a broad improvement in weapon choice and micro-strategy selection against a broad array of mid-game threats. The authors note that the hard coded combat strategies of *BotHack* are "not well tuned to [their] limited, single-room test environment," [13] but that indeed combining the proven success of more complicated bots with their optimization strategy is a key domain of future work.

2.3 Applications of Expert Systems

Understanding the process necessary to optimize game agents and expert systems has benefits outside the domain of video game AI. The following works are examples of this applicability.

- **Dynamic Asset Protection & Risk Management Abstraction Study**
  Rog-O-Matic has been further analyzed by scholars of other fields, such as Henderson et al.'s appraisal of *Rog-O-Matic* as an expert system in the article "Dynamic Asset Protection & Risk Management Abstraction Study". [21] The authors analyze the ways in which Rog-o-Matic was designed so as to "create an expert system with the ability to solve an exploration problem", going on to describe the contours of "exploration problems" in mathematical terms, as systems containing planar graphs which are explored from starting points or nodes and which share observable characteristics and the ability to transition between observable nodes. [21] This sort of theoretical work on AI for procedurally generated games shows the applicability of this sort of work to more generalized and mathematical problem sets, and real world problem spaces such as automated network security as described by Henderson et. al.
- **An expert system for detection of breast cancer based on association rules and neural network**
  In this paper by Karabatak and Ince, an expert system is described which is capable of detecting breast cancer with a correct classification rate of 95.6%. [23] This work shows not just the applicability of expert systems towards real world problem sets, but additionally the potent combination represented by expert systems and machine learning. Originally based solely on neural network derived classification, the author's system performed better when an expert system based on association rules was incorporated into their classifier. This shows that real world applications of expert system schemas can greatly benefit from machine learning based optimization, and machine learning based classifiers can likewise be improved by the addition of heuristically designed expert systems.

3 DESIGN & FRAMEWORK

The design of this project has three major focuses. The software components of the project - DCSS, *qw*, and scikit-learn - is the first focus to be discussed. The second is the framework of the hardware utilization scheme used to collect data and generate results based on said data. Finally, the specific design choices of what exactly is being optimized will be discussed.

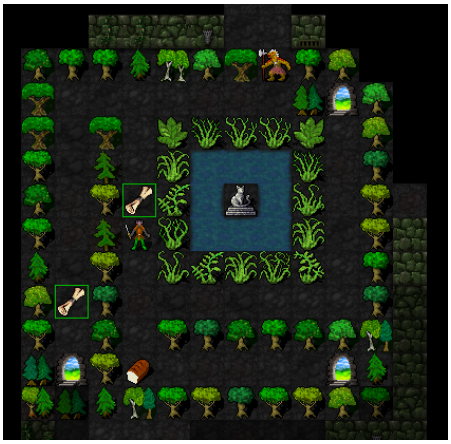
3.1 Software Components

This project focuses upon optimizing the expert system AI agent *qw* for the game DCSS, utilizing the open source machine learning toolkit scikit-learn. The justifications for the selection of these pieces of software shall be discussed in this section.

3.1.1 DCSS. Dungeon Crawl Stone Soup (DCSS), a game in the roguelike genre represents a facet of the problem space for this work. "Roguelike" is a term for a genre of computer game featuring a number of distinct elements. Frequently, these elements are taken to be procedural content generation (PCG), "permadeath", grid-based gameplay, and turn-based gameplay. [19] These attributes present a design space for AI systems that is at once challenging, interesting, and suitable.

The combination of "random environment generation" (an element of PCG) and permadeath create an experience where "[n]ot only with every new game does the player enter a different location, but also s/he always does it as a new character with only one life" [19]. This serves to enhance re-playability for the consumers of such games, and to create an interesting space to explore in terms of AI development. Where games like Chess or Go feature static content, predictable beginnings and a static board, games with procedural content must be adapted to for any given attempt to beat it. Lessens learned in failing must be generalized, as specific contexts and encounters have minute chances of occurring again, diminishing the usefulness of a "book" of board states such as in chess. Many roguelikes do employ "seeds" in the algorithms which generate procedural content[4], so it is theoretically possible to encounter the same dungeon multiple times, which can be useful for occasions such as debugging or challenges between players competing for high-scores or fast victories. Figure 2 depicts a procedurally placed vault at the start of a game of DCSS; while the layout of trees and water is a fixed element, it was randomly selected to be present in the instance of the game featured, and the items and monster appearing in it were randomly placed. DCSS has a vast amount of such vaults, allowing them to be interspersed with more stochastic seeming content for unique instances of the game to still have chunks of structure. These vaults reward familiarity with the game, as some feature specific valuable items or dangerous enemies that a seasoned player can know are worth searching for or avoiding when they encounter the vault. [22]

Figure 2: A screen capture of a game of DCSS. The player character is wearing green boots and wielding a sword; three exits are visible, and a fierce gnoll wielding a halberd threatens the playerâĂŹs progression further into the dungeon. The game can alternately be played in ASCII mode in a Unix-like console environment, where symbols take the place of tiles and sprites in representing gameplay information.



While the procedural generation of roguelike games provides an interesting twist to the approach needed to develop effective AI, the grid and turn based nature of the genre facilitates such work. A grid based game provides a discrete amount of movement options an order of magnitude or two smaller than a game with full 360°movement and rotation. Additionally, the turn-based nature of the genre enables "thoughtful" AI that is not dependent upon quick decisions. With no reflexes to account for, the AI can take as long as is necessary to ponder individual choices, turns, and courses of action.

These factors are all present in DCSS, and make it a compelling problem space to optimize expert systems within. Additional features specific to the game also make it an interesting and worthwhile testbed for automated gameplay agent optimization. These features are rooted in the structure of the game itself - DCSS utilizes "lua bindings" to allow for easy extension of the game and creation of macro functions to allow players greater efficiency on input. It is with these lua bindings that *qw* is written. Additionally, DCSS contains a few automation features for reduction of gameplay tedium that make it particularly suitable for developing AI systems to play it. These are "autoexplore" and "autofight," which respectively allow the player to automatically explore the dungeon until they reach one of a designated stopping point (e.g. seeing a new monster, finding a particular kind of item) and allow the player to either take a single step towards an enemy or to attack the nearest enemy. These features make implementation of an AI agent to play the game much easier.

Finally, it is worth noting that the version of DCSS this project uses is 22.0. This version was selected as it was the most recent mainline version released at the start of the Fall Semester 2018.

3.1.2 qw. *qw* was the first AI agent created that is capable of winning a game of DCSS. It reportedly wins games using the character configuration of Deep Dwarf Berserker at a rate of approximately 17%. [18]

The bot is structured like an expert system; it contains knowledge sources such as a dictionary associating monsters with their "scariness" and functions that are hard coded to return value weights for item comparison.

qw features goal seeking behavior similar to that of the Angband "Borg", where smaller goals, such as finding a downward staircase or escaping from a dangerous monster take place sequentially as necessary on the way to larger, more overarching goals, such as finding the three runes of Zot necessary to win the game.

3.1.3 scikit-learn. Scikit-learn is an open source, Python based machine learning framework that allows users to easily implement machine learning. [8] It was selected due to its ease of implementation and the wide availability of guides on getting it up and running.

Two kinds of machine learning were attempted with scikit-learn for this project: Stochastic Gradient Descent (SGD) and Logistic Regression with Cross Validation (LogisticRegressionCV). SGD was selected for its speed and efficiency at processing large sets of data [9], and LogisticRegressionCV for its accuracy and ease of tuning hyperparameters due to built in cross validation. [10]

The two applied machine learning algorithms will be performing classification: classifying whether given permutations of monster scariness weights are likely to lead to a successful run of the first floor of the dungeon or not.

3.2 Optimization Focus

Instead of attempting to optimize all of *qw* for all of the game of DCSS, this project aims to reduce the scope of experimentation. In experimental trials, it took several minutes for *qw* to complete a playthrough of DCSS, so instead of optimizing *qw*'s performance for the whole game, the project aims to optimize the success of the bot at traversing the first floor of the dungeon. DCSS allows for a wide variety of starting character permutations The trials will all be conducted Additionally, the focus of the optimization itself will be on a particular knowledge source within *qw*: the monster scariness function, which returns a value corresponding to the threat level of a given monster.

3.2.1 D 1. **D 1** is the in-game name for the first floor of the dungeon. By focusing on optimizing *qw*'s success at traversing **D:1**, a few things are made easier. Runs of qw on the first floor take approximately 3 seconds to complete on the hardware utilized for this project; this allows for much faster data collection and result validation than if the entire game was the focus. Additionally, there is a small subset of monsters that will appear on the first floor of the dungeon in the course of normal gameplay. This small subset acts as a sort of natural feature selection so as to simplify the amount of monsters under consideration when attempting different scariness weight permutations.

3.2.2 Character Selection. The first floor of the dungeon has a drawback for these purposes. *qw* is so effective with standard character configurations that the most touted *qw* character, the Deep Dwarf Berserker (DDBe), sees a 98% success rate at traversing the first floor of the dungeon. As the Berserker (Be) class is necessary to enable the berserk ability to the character from the beginning

of the game, this project will attempt to optimize a different race of Be for the first floor. The character race of "human" (Hu) was selected, due to its relatively lower proficiencies compared to the DD. Additionally, DD features a powerful self-healing ability that Hu lacks, which could also contribute to its incredibly high success rate.

3.2.3 Monster scariness. Optimizing for proper permutation of monster scariness was selected as it makes for an interesting decision within the confines of the first floor of the dungeon. *qw*, when evaluating its strategy against a given threat, appraises the "scariness" rating of a monster, an integer between 1 and 27. If this integer is higher than the player's level, *qw* will attempt to initiate combat by berserking. This powerful ability gives the player character extra damage in combat, increases their attack and movement speed, and prevents them from taking actions outside of movement, attacking, butchering corpses, and eating. In addition to restricted actions while berserking, there are costs associated with it: the character loses satiation, pushing them closer to starvation, and after berserking the character often becomes paralyzed for 4-7 turns, preventing them from taking actions while monsters can still attack them. Additionally, the character is slowed after berseking, and is unable to berserk again for a period of time. While the combat boon of berserking is necessary to survive some encounters, especially early in the dungeon when other resources are limited, it is not a one sided endeavor, and berserking too frequently can lead to premature death for an adventurer delving the depths of the DCSS dungeon. As such, the decision of whether or not to berserk is an interesting one, especially within the context of the first floor of the dungeon.

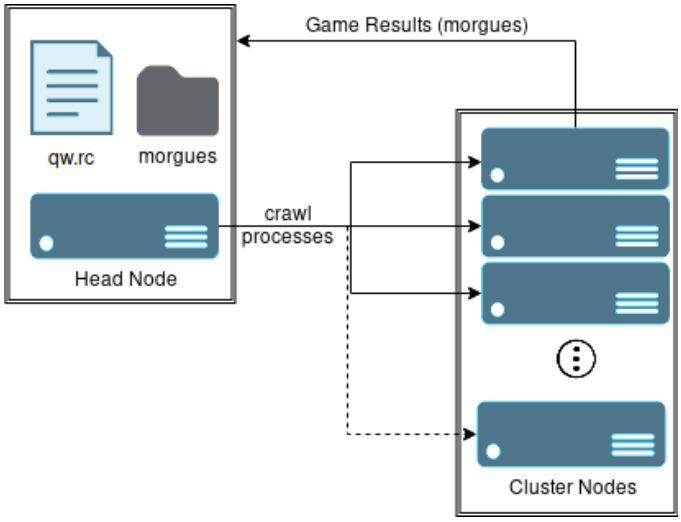
3.3 Hardware Utilization & Parallelization

To collect enough data to properly utilize machine learning, parallel hardware was used to run many trials of crawl simultaneously. *qw* is embodied in a configuration file known as an "rc file", i.e. *qw.rc*. Different permutations of *qw* were stored on a central head node of the Al-Salam cluster. These permutations varied by the scariness weights assigned to the different encounter-able monsters on the first floor. When a game of crawl ends or saves, a morgue file is created. These morgue files have different file-name structures based on whether the character in question was alive or dead at their creation. The cluster's head node sent out simultaneous instructions to run DCSS on the 12 compute nodes of the Al-Salam cluster, with each node assigned a different rc file with which to run *qw*. Seeds were utilized to ensure reproducibility; seeds in sequence from 1 to *n* were utilized, where *n* was the given amount of runs for a batch of trials. The morgue files created were then scanned by a script run from the head node, and data was collected in the form of a binary success/failure ratio for each permutation of scariness weights. See figure 3 for a diagram detailing the parallelization process. This data was used to train the machine learning models.

3.4 Scikit-learn and Python

The scikit-learn pipeline used to generate classification models is rather straightforward. A Python dictionary was created associating run tags (unique identifiers given to each set of trials) with lists. These lists contain further lists, which contain integer values of

Figure 3: This figure depicts the process by which data was collected. A head node sent out simultaneous instructions to run DCSS on the 12 compute nodes of the Al-Salam cluster. When a game finished the first floor of the dungeon, a morgue file was created and the results were sent to the head node. In some trials, the head node ran DCSS processes as well.



wins and losses, alongside the permutation of scariness weights for the given node on the given run. This dictionary was then iterated through to produce two NumPy arrays, one of shape (x, y) where x is the total number of runs completed and y is the permutation of weights for each given run, to serve as a samples array, and the other array of shape (x, 1), to serve as a target array, with each element being a 1 or 0, corresponding with success or failure for the equivalent row in the samples array.

These arrays were then used to train a classifier, either using SGD or LogisticRegressionCV.

After training, an additional list of every possible permutation of the 10 scariness weights in question was created. This list was iterated over, and each possible permutation was stored alongside the probability of the permutation being a 1 (success), determined by the trained classifier. The permutation with the highest probability of success was then extracted from the final list, and a new rc file was created with those values. These new rc files were used to run more trials of *qw* to compare relative rates of success between the machine learning selected models and the training data.

Since SGD operates stochastically, different permutations were noted to have the highest probability of success after training the model on the same data with the same hyperparameters different times. Three different models were generated and tested. LogisticRegressionCV lacks this variation, and produced the same model when run on the training data multiple times. This model was also tested for success. The following section will catalog both the results of the training runs and the results of the permutations created with scikit-learn.

4 RESULTS

First, the data collected in training will be documented. Note that while nodes were instructed to run round numbers of trials (e.g. on seeds 1-10000), there was some level of unpredictable failure stemming from bugs in the code.

4.1 Scariness Encoding

Scariness ratings for 10 monsters on the first floor of the dungeon were encoded in a string of ten integers, each corresponding to a monster. These strings were used to create the models on the training data. Monster names and their corresponding index are shown below.

Monster Name	Index
Adder	0
Bat	1
Giant Rat	2
Gnoll	3
Goblin	4
Hobgoblin	5
Jackal	6
Kobold	7
Leopard Gecko	8
Worm	9

As an example, the default scariness encoding of *qw* for the monsters of the first floor of the dungeon puts gnolls at 5, worms at 4, and no other monsters have any values. This is encoded thusly:

Default Scariness Values	
Monster Name	Scariness Value
Adder	0
Bat	0
Giant Rat	0
Gnoll	5
Goblin	0
Hobgoblin	0
Jackal	0
Kobold	0
Leopard Gecko	0
Worm	4

4.2 Training Data

With scariness weight set to 2, *qw* will use the berserking strategy only against the selected monster if the character level is at 1.

Scariness Weight Set To 2				
Node	Trials	Deaths	Scariness Emphasis	Rate of Success
as1	8871	457	Adder	94.85%
as2	8851	500	Bat	94.35%
as3	8870	8409	Giant Rat	94.80%
as4	8894	405	Gnoll	95.45%
as5	8866	484	Goblin	94.54%
as6	8875	449	Hobgoblin	94.94%
as7	8867	461	Jackal	94.80%
as8	8898	408	Kobold	95.41%
as9	8876	452	Leopard Gecko	94.91%
as10	8875	452	Worm	94.91%
as11	8868	475	All Monsters	94.64%
Total	97611	5004	Average	94.87%

With scariness weight set to 3, *qw* will use the berserking strategy against the selected monster if the character level is at 1 or 2. Slightly stronger characters will be willing to berserk against threats in these trials.

Scariness Weight Set To 3				
Node	Trials	Deaths	Scariness Emphasis	Rate of Success
as1	4932	250	Adder	94.93%
as2	487	35	Bat	92.81%
as3	489	27	Giant Rat	94.48%
as4	494	16	Gnoll	96.76%
as5	72	1	Goblin	98.61%
as6	494	25	Hobgoblin	94.94%
as7	4922	273	Jackal	94.45%
as8	4945	220	Kobold	95.55%
as9	491	26	Leopard Gecko	94.70%
as10	4923	249	Worm	94.94%
as11	4920	286	All Monsters	94.19%
Total	27169	1408	Average	95.12%

The maximum level a character can reach on the first floor of the dungeon is 3. Therefore, any scariness weight set to 4 or higher will have equivalent results for a character piloted by *qw* on the first floor of the dungeon.

Scariness Weight Set To 5				
Node	Trials	Deaths	Scariness Emphasis	Rate of Success
as0	29717	1174	Default Behavior	96.05%
as1	8871	446	Adder	94.97%
as2	8851	590	Bat	93.33%
as3	8870	461	Giant Rat	94.80%
as4	8895	354	Gnoll	96.02%
as5	8865	488	Goblin	94.50%
as6	8875	457	Hobgoblin	94.85%
as7	8867	493	Jackal	94.44%
as8	8897	410	Kobold	95.39%
as9	8876	446	Leopard Gecko	94.98%
as10	8875	446	Worm	94.97%
as11	8868	510	All Monsters	94.25%
as12	8870	460	Never berserk	94.81%
Total	136197		Average	94.87%

4.3 SGD Derived Scariness Values

Three versions of SGD derived scariness weights were tested. Following are those weight assignments and their results.

4.3.1 SGD_1. Following are the derived weights from the first utilization of SGD.

SGD_1 Scariness Values	
Monster Name	Scariness Value
Adder	4
Bat	0
Giant Rat	4
Gnoll	4
Goblin	0
Hobgoblin	0
Jackal	4
Kobold	4
Leopard Gecko	0
Worm	4

The results from trials run with these weights are as follows. These trials were run with random seeds so as to confirm the models were not over-fitting to the input data based on the first *n* seeds.

SGD_1 Success Rates				
Node	Trials	Deaths	Rate of Success	
as2	494	21	95.75%	
as3	492	24	95.12%	
as4	496	15	96.98%	
as5	496	23	95.36%	
as6	496	20	95.97%	
Total	2474	103	Average	95.84%

4.3.2 SGD_2. Following are the derived weights from the second utilization of SGD.

SGD_2 Scariness Values	
Monster Name	Scariness Value
Adder	4
Bat	0
Giant Rat	4
Gnoll	4
Goblin	0
Hobgoblin	4
Jackal	4
Kobold	4
Leopard Gecko	0
Worm	0

The results from trials run with these weights are as follows. These trials were run with random seeds so as to confirm the models were not over-fitting to the input data based on the first *n* seeds.

SGD_2 Success Rates				
Node	Trials	Deaths	Rate of Success	
as2	9875	483	95.11%	
as3	9858	539	94.53%	
as4	9893	464	95.31%	
as5	9893	432	95.63%	
as6	9875	470	95.24%	
Total	49394	2388	Average	95.17%

4.3.3 SGD_3. Following are the derived weights from the third and final utilization of SGD.

SGD_3 Scariness Values	
Monster Name	Scariness Value
Adder	0
Bat	0
Giant Rat	0
Gnoll	4
Goblin	4
Hobgoblin	0
Jackal	0
Kobold	0
Leopard Gecko	0
Worm	0

The results from trials run with these weights are as follows. These trials were run with random seeds so as to confirm the models were not over-fitting to the input data based on the first *n* seeds.

SGD_3 Success Rates				
Node	Trials	Deaths	Rate of Success	
as2	9874	449	95.45%	
as3	9880	435	95.60%	
as4	9901	382	96.14%	
as5	9899	410	95.83%	
as6	9897	363	96.33%	
Total	49451	2039	Average	95.88%

4.4 LogisticRegressionCV Derived Scariness Values

Following are the derived weights from LogisticRegressionCV.

LogisticRegressionCV Scariness Values	
Monster Name	Scariness Value
Adder	0
Bat	0
Giant Rat	0
Gnoll	4
Goblin	0
Hobgoblin	0
Jackal	0
Kobold	4
Leopard Gecko	0
Worm	4

The results from trials run with these weights are as follows. These trials were run with random seeds so as to confirm the models were not over-fitting to the input data based on the first n seeds.

SGD_3 Success Rates				
Node	Trials	Deaths	Rate of Success	
as1	2379	90	96.22%	
as2	2383	86	96.39%	
as3	2374	105	95.58%	
as4	2367	113	95.23%	
as5	2376	93	96.09%	
as6	2378	96	95.96%	
as7	2375	99	95.83%	
as8	2370	85	96.41%	
as9	2374	91	96.17%	
as10	2379	79	96.68%	
as11	2383	88	96.31%	
as12	2372	79	96.67%	
Total	28510	1104	Average	96.13%

4.5 Comparison and Analysis

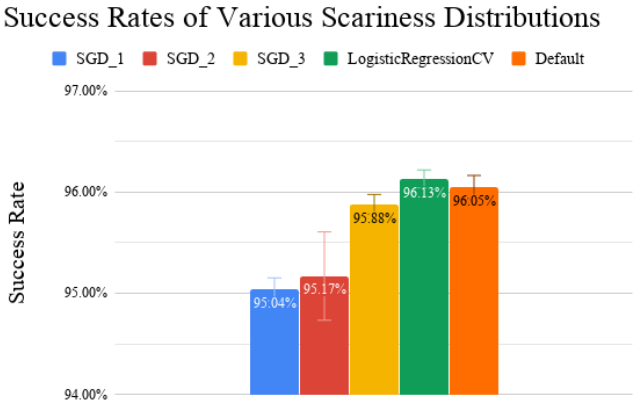
Figure 4 depicts the success rates of the different scariness distributions derived by the machine learning models, with the default results included for comparison's sake.

Statistical analysis of results was conducted with Chi-squared testing at a significance level equal to 0.05. The first SGD model was 1.01% worse than the default behavior with P = 0.0141. The second SGD model was 0.88% worse than the default behavior with P < 0.0001. The third SGD model is similar to the default behavior; it is 0.17% smaller, but with P = 0.2404, we cannot reject the null hypothesis. The LogisticRegressionCV model performed 0.08% higher than the default behavior, but P = 0.6186. Therefore, that model does not show a statistically significant difference in performance from the default behavior.

5 CONCLUSION

Hundreds of thousands of data points were collected using various monster scariness weights. These data points were fed into Scikit-learn, and models were created and utilized to attempt to find more optimal distributions of scariness weights to improve one of qw's knowledge sources, thus improving its performance when that knowledge was handed off to the subsystem in charge of combat strategy.

Figure 4: This chart shows the difference in success rates between the trials run on the machine learning optimized scariness weights. Error bars shown are calculated standard error for each series.



The attempted optimizations of qwâĂŹs berserk logic failed to improve the success of HuBe's in traversing the first floor of the dungeon. Initial testing with a weaker character race, Deep Elves (DE's), shows promise that the LogisticRegressionCV model can indeed create significant performance improvements for the focused optimization. Under default behavior, Deep Elf Berserkers (DEBe's) succeeded at a rate of 86.86%, much less than the Human rate of 96.05%. DEBe testing showed a 2.16% improvement (up to 89.02%) in success rate with P = 0.0028, showing statistically significant improvement. This indicates that such optimization methods may be more impactful when initial rates of success are lower.

5.1 Future Work

Further tuning of hyperparameters could improve the results of SGD modeling, which produced substandard results. Additionally, some features could be pruned to improve performance - monsters like bats, for instance, could simply always be set to null, as no model suggested they should be feared.

Creating models from data based on weaker character races shows promise based on initial DEBe analysis. By lowering initial success rates of the bot, further room for improvement is available.

This same sort of knowledge source tuning could be performed for different parts of qw, optimizing progression through more of the game than the first floor. For instance, item valuation weights could be looked at to judge which items are more effective overall. Other, larger strategic choices such as optional rune branch selection could also be areas of interest.

These techniques also have potential applications for software optimization in fields unrelated to gaming, such as search and rescue drone navigation. The promise shown by the initial tests of DEBe on the LogisticRegressionCV sourced weights indicates that this form of machine learning can lead to significant improvements in the performance of expert systems under various metrics.

6 ACKNOWLEDGEMENT

Thanks to Dr. David Barbella for his motivational and informative advising, Dr. Xunfei Jiang for her constant guidance and leadership, and Dr. Charles Peck for his inspiration and encouragement. Additional thanks to all of my peers in the Earlham Department of Computer Science for their feedback and support. Thanks to the DCSS community, including the ##crawl and ##crawl-dev IRC channels for their valuable knowledge of the inner workings or DCSS. Special thanks to Sigmund, for staying out of the first floor of the dungeon.

REFERENCES

[1] [n. d.]. About - rephial.org. http://rephial.org/develop
[2] [n. d.]. Angband Borg – News. http://www.phial.com/angborg/news.html
[3] [n. d.]. NetHack / Developers Foresight - TvTropes. https://tvtropes.org/pmwiki/pmwiki.php/DevelopersForesight/NetHack
[4] [n. d.]. Random number generator - RogueBasin. http://www.roguebasin.com/index.php?title=Random_number_generator
[5] 1998. A heuristic-based genetic algorithm for workload smoothing in assembly lines. 25 (Feb 1998), 99âĂŞ111. https://doi.org/10.1016/S0305-0548(97)00046-4
[6] 2016. Cheating - Roguebasin. http://www.roguebasin.com/index.php?title=Cheating
[7] 2018. http://roguebasin.roguelikedevelopment.org/index.php?title=What_a_roguelike_is
[8] 2018. https://scikit-learn.org/stable/
[9] 2018. https://scikit-learn.org/stable/modules/sgd.html#classification
[10] 2018. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html
[11] 2018. NetHack - NetHack Wiki. https://nethackwiki.com/wiki/NetHack
[12] A. Afshar, O. Bozorg Haddad, M. A. MariÃśo, and B. J. Adams. 2007. Honey-bee mating optimization (HBMO) algorithm for optimal reservoir operation. *Journal of the Franklin Institute* 344, 5 (Aug 2007), 452âĂŞ462. https://doi.org/10.1016/j.jfranklin.2006.06.001
[13] Jonathan Campbell and Clark Verbrugge. 2017. Learning Combat in NetHack. https://aaai.org/ocs/index.php/AIIDE/AIIDE17/paper/view/15815
[14] Vojtech Cerny and Filip Dechterenko. 2015. Rogue-Like Games as a Playground for Artificial Intelligence âĂŞ Evolutionary Approach. In *Entertainment Computing - ICEC 2015 (Lecture Notes in Computer Science)*. Springer, Cham, 261–271. https://doi.org/10.1007/978-3-319-24589-8_20
[15] Crawl Devteam. 2018. Dungeon Crawl Stone Soup. https://crawl.develz.org/
[16] A. K. Dewdney. 1985. An expert system outperforms mere mortals as it conquers the feared dungeons of doom. *Scientific American* 252, 2 (1985), 18–21. https://web.archive.org/web/20160625035902/http://science-fiction.fch.ir/rogue/doc/An_expert_system_outperforms_mere_mortals_as_it_conquers_the_feared_Dungeons_of_Doom.html
[17] duke nh. 2015. YAAP: full-auto bot ascension (BotHack). https://www.reddit.com/r/nethack/comments/2tluxv/yaap_fullauto_bot_ascension_bothack/
[18] elliptic. 2018. The DCSS-playing bot qw. https://github.com/elliptic/qw original-date: 2014-09-21T22:32:57Z.
[19] Maria B Garda. 2013. Neo-rogue and the essence of roguelikeness. (2013), 15.
[20] Antonio GonzÃąlez-Pardo, Fernando Palero, and David Camacho. 2015. An empirical study on collective intelligence algorithms for video games problem-solving. (2015). https://repositorio.uam.es/handle/10486/674486
[21] G Henderson, E Bacic, and M Froh. 2005. Dynamic Asset Protection & Risk Management Abstraction Study. (2005), 50.
[22] Dr Mark R Johnson. 2015. Handmade Detail in a Procedural World. http://twvideo01.ubm-us.net/o1/vault/gdceurope2015/Johnson_Mark_HandmadeDetailIn.pdf
[23] Murat Karabatak and M. Cevdet Ince. [n. d.]. An expert system for detection of breast cancer based on association rules and neural network. 36, 2 ([n. d.]), 3465–3469. https://doi.org/10.1016/j.eswa.2008.02.064
[24] Thorey Maria Mariusdottir, Vadim Bulitko, and Matthew Brown. [n. d.]. Maximizing Flow as a Metacontrol in Angband. In *AIIDE* (2015).
[25] Colin Morris. 2018. *Analyzing completed games of Dungeon Crawl Stone Soup: colinmorris/crawl-coroner*. https://github.com/colinmorris/crawl-coroner
[26] retrobits. [n. d.]. retrobits/rogomatic - github. https://github.com/retrobits/rogomatic original-date: 2016-01-22T18:33:47Z.
[27] P. B. Thanedar, J. S. Arora, G. Y. Li, and T. C. Lin. 1990. Robustness, generality and efficiency of optimization algorithms for practical applications. *Structural optimization* 2, 4 (Dec 1990), 203âĂŞ212. https://doi.org/10.1007/BF01748225