Collin Nelson
December 2023
CPTS 437 Intro to Machine Learning

437 Final Project

Pathfinder 2E "What's My Level"

Collin Nelson

Collin Nelson
December 2023
CPTS 437 Intro to Machine Learning

Key Project Links

Thank you for checking out the project for grading, these links below reference all of the important deliverables for this project. My apologies for the fact that this document is technically over the page limit, but I figured it would be acceptable given that the extra two pages are just because I added a separated Title page and Appendix to increase clarity overall.

Github repo

https://github.com/Nelsocol/P2ELevelFinder

Because I opted for a real deployment to the internet over a demo colab notebook, you can simply try the app using the link below without needing to download or install anything. If you would like to download and run the app locally, please view the README in the github repo above.

The README also contains quick links to the core operational code that is most relevant to this machine learning class, for easier grading.

Presentation & Slides

https://drive.google.com/drive/folders/1Lyfj87PsNAKPv-aYj0NjcxBlmF38kHKN?usp=sharing

Live App Deployment

https://whatsmylevelp2e.netlify.app/

This link to my live site allows you to try the app online. There is no colab notebook because my project is built in C# with a Blazor web-UI frontend, and it wouldn't operate well in a Colab environment. For the best results, open in Chrome with a standard HD (1920x1080) screen size. The site should function in other contexts, but some UI may not look/function as expected.

Collin Nelson
December 2023
CPTS 437 Intro to Machine Learning

I. Introduction & Overview

For the last 7 to 8 years, I have been someone who often serves as a Gamemaster (GM) for tabletop roleplaying games. In this time, I have made extensive use of what is referred to as "homebrew" monsters, meaning monsters whose lore, stats, and abilities were created from scratch instead of taken from one of the game's many bestiaries. One of the biggest challenges of working with homebrew creatures, however, is judging from stats alone how strong or weak your creature will be when actually set against players in game. This is often something that requires a great deal of experience in a single game system to be able to accomplish quickly for a variety of creatures. This document is a complete overview of my project, which aims to solve this problem for the complex system Pathfinder 2nd Edition (P2E), by utilizing machine learning to predict the anticipated level of a creature in P2E from some or all of its primary stats. In this document, I will explore the project motivations, goals, and approach. I will break down my implementation in detail and provide the results of the empirical analysis I performed both to improve my model, as well as to verify its accuracy upon completion. Finally, I will discuss briefly what the limitations of my approach are, as well as future work I would like to do on this project.

II. Project Statement

i. Motivations

P2E is far from a simple system. In fact, its complexity and the depth of its mechanics actually make it one of the most complex systems to learn and build homebrew for. This is challenging for a few reasons, which provide the onus for this machine learning application.

- **The Number of Stats**: A single creature in P2E can consist of over 15 different stats and any number of attacks and abilities. Balancing all of these with each other as well as making them appropriate to the creature is very difficult.
- **The Nuance of Stats:** Even seemingly similar stats can sometimes have nuanced differences. For instance, every monster has six different "attributes" labelled; STR, DEX, CON, INT, WIS, and CHA. These can often blend together, but because of the interplay between these stats and player abilities, not all of these stats are created equal, and they may have different impacts on a creature's overall power.
- **The Diversity of Abilities:** Along with many stats, creatures have a great number of abilities. These abilities, which include attacks, special actions, and contextual abilities, have a great impact on the strength of a creature, but these are so numerous and diverse in how they operate, that it can be very difficult to balance them to create interesting creatures of appropriate level for the players.

The final motivation for the project is the general lack of any comparable tools. While there are some guidelines for Pathfinder homebrew, there is *no* rigorous way to estimate the level of a homebrew monster, and I couldn't find any similar tools applying machine learning during my research.

ii. Project Goals

The goal of this project is to create a usable *public facing* tool for estimating the level of a homebrew pathfinder 2nd edition monster. There are 3 major pillars to this tool's design that serve as core project objectives.

- **Speed:** A user may want to experiment or iterate on their monster designs many times in quick succession. The tool should allow for easy real-time modification of candidate monster stats without the prediction causing delay.
- **Versatility:** The prediction engine should be able to operate on very incomplete information. Users *should not* have to input all of the possible parameters of their monster design to get accurate predictions from the model.
- **Usability:** Modern AI applications are expected to not only employ machine learning principles, but to do so in a clean and user-friendly package. This application should not only be functional, but easy to use and view outputs for someone with no understanding of the underlying machine learning implementation.

iii. Approach Overview

My general approach for solving this problem was to implement a multidimensional KNN implementation. While KNN is one of the simpler models we explored in class, it was very appropriate to the problem at hand, as it scaled well to the number of input parameters I was working with, and notably it suits itself to the *versatility* pillar discussed above. Because KNN is a model that can operate effectively without needing all of its parameters to be filled (assuming proper implementation anticipating this design requirement) it is especially appropriate. Key advancements I made over the implementation we already did in class include automatic normalization of all input parameters at runtime based on metric obtained during data ingestion, the addition of a weights layer that can help to balance the strength of various dimensions, as well as the implementation of the null-forgiving estimation process that dynamically ignores dimensions in which the input parameter is null.

I built the UI using Blazor WASM, a client-side web framework that allows for a C# backend for faster runtime execution. This allows me the flexibility of HTML and CSS in designing a UI for this application, as well as the power of C# for the core implementation. This also allowed me the ability to deploy the finished app to the web, allowing users to interact with it easily without downloading and running the raw code.

III. Architecture & Implementation

i. Data Structures

Data is stored on disk by my application in the form of JSON. This JSON contains raw data about every monster in the official Pathfinder bestiaries. It totals over 190,000 lines of text and contains just short of 2000 monsters.

In my code, there are 3 core classes used to perform the core functionality of the app.

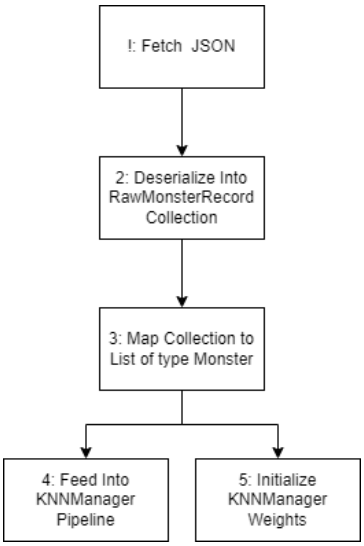
- **RawMonsterRecord:** For ease of JSON deserialization, it's important to have a data structure that closely mirrors the raw JSON file. Because this JSON was not

created by me for my application specifically the RawMonsterRecord class exists to provide a C# representation of how the original file creator formatted the JSON.

- **Monster:** This class is a single "point" in my KNN cloud. It contains a number of bits of context information about the monster that I want preserved to be able to display to the user, as well as the n-dimensional "hyperposition" of the point in the KNN cloud. Notably, it contains a function that allows an object of this type to be directly initialized from a *RawMonsterRecord*.
- **KNNManager:** The KNNManager is the class that operates the KNN cloud, stores the *Monster* objects that comprise the point cloud, and provides utilities for adding points, reinitializing the cloud, and making estimations with it. It serves mostly as a storage class, and as a singleton interface for interacting with the KNN model. It does, however, store a small amount of additional data, such as the weights layer used to improve the accuracy of results, as well as the maximum and minimum values of each dimension (used for normalization).

ii. Ingestion Data Flow

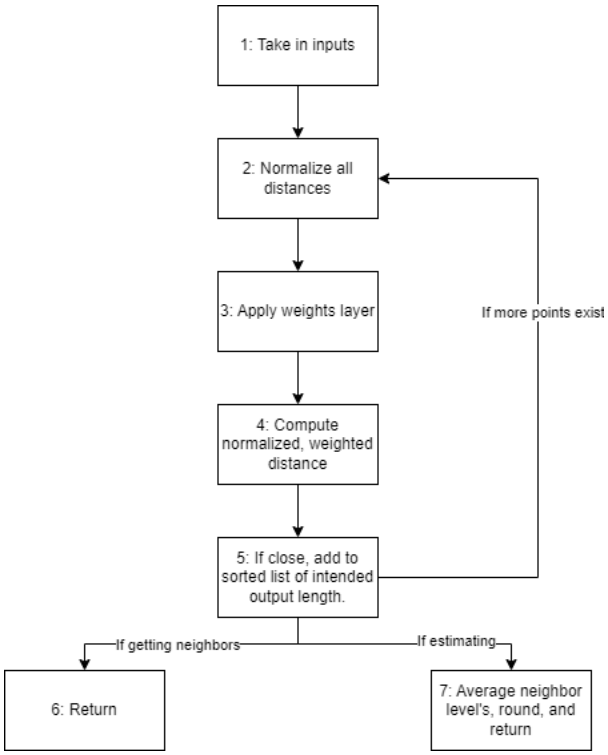
When the app is initialized the following flow of operations is performed to initialize and construct the KNN cloud.



1. **JSON Fetch:** Due to the web deployment framework of this application data cannot be accessed locally. Instead, JSON is stored in an adjacent directory and fetched using an HTTP request.
2. **Deserialize:** Using built in JSON utilities, deserialize into a RawMonsterRecord[]
3. **Map:** Using the Monster.FromRawMonsterRecord() function, map to List<Monster>
4. **Inject:** Feed the generated Monster points into the KNNManager, this also will automatically initialize the minimums and maximums within the KNNManager
5. **Init Weights:** Separately from feeding in the Monster points, initialize the weights layer of the KNNManager with appropriate weights.

iii. Runtime Operational Behaviors

After the KNNManager has been initialized, the following describes the flow of information used to actually perform a single estimation operation via the KNNManager.



1. **Ingest Inputs:** Inputs are fed in the form of an array of *nullable* integers, as well as the desired neighborhood size.
2. **Normalize:** The max and min values recorded in the KNNManager are used to normalize all values to the range 0-1. *Note: though it's separated in this diagram for clarity, this is actually done inline with the distance calculation.*
3. **Apply Weights:** Apply weights to the computed distances. These weights help to ensure that although parameters are normalized, not all are equally powerful in defining adjacency.
4. **Compute Distance:** Along with normalizing and weighting, distance is computed using a standard Euclidean distance function.
5. **Update Output List:** During estimation, the system maintains a list of *n* elements. This list is maintained sorted to avoid a separate sorting pass.
6. **Return:** If the KNNManager was asked to just provide the near neighbors, it returns the neighbor list directly.
7. **Average Values and Return:** If the KNNManager was asked to produce an estimated level, it averages the neighbor list's levels, rounds to the nearest integer, and returns.

iv. User Interface

The user interface was one of the components of the software that I spent the most time working on. It was very important to me in constructing this project that it has a clean, usable UI that could be effectively deployed to the web. The general layout of the tool appears below with annotations.

1. **Input Pane:** The input pane is the panel in which the user can type the stats of their homebrew monster. It contains the ability to set all 16 of the inputs taken in by the machine learning model. All input fields are optional, and all input fields have placeholder text when not filled to indicate what should be filled into that slot. The rest of the UI updates in real time any time that the user tabs, clicks, or presses enter on any of the input fields in the input pane.
2. **Level Output:** This circle contains the output of the predicted level of your monster. It updates in real time as changes are made to the stats of your creature in the input pane and displays "?" if all inputs are null.
3. **Neighbors Display:** This list displays a collection of the nearby neighbors that were used to estimate the level shown above. This serves a very important purpose, as it allows the user to browse monsters that might be similar to what they're trying to create to find interesting abilities or examples of how to complete the stat block for their monster.
4. **Monster Name & AonPRD Link:** When one of the neighbors in the neighbors display is clicked on, its details appear on the right, in the details pane. The header of this section, with the name of the creature, is also a link to the online Pathfinder wiki AonPRD, automatically providing the user a way of viewing the official wiki entry for the creature if needed.'
5. **Details Pane:** The details pane contains all of the details of the selected monster from the neighbors display. This includes the monster's description, all of the notable stats tracked by the KNNManager, as well as the full text of the monster's abilities.

IV. Empirical Testing

i. Test Methodology

In order to calibrate and evaluate my model, I created a version of the source JSON in which a large group (roughly 10%) of the monsters were removed and moved to another

file. This file was then used as test data that was loaded and used to evaluate the accuracy of my estimations.

ii. Testing Ideal Neighborhood Size

The first part of my model I wanted to calibrate was the ideal neighborhood size. Below is a graph showing the results of different neighborhood sizes on the accuracy of the model as well as on the average distance between mispredictions and the real values.

The raw data used to create these graphs can be found in *(Appendix I: Table 1)*

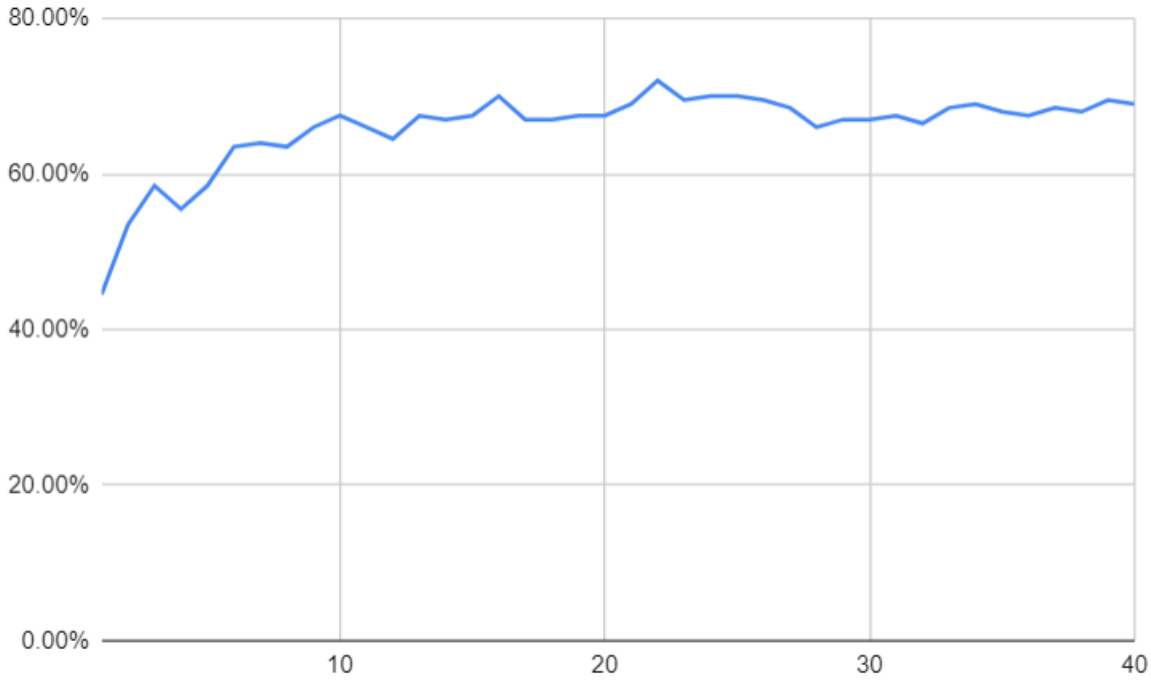


Figure 1: Neighborhood Size vs. Model Accuracy

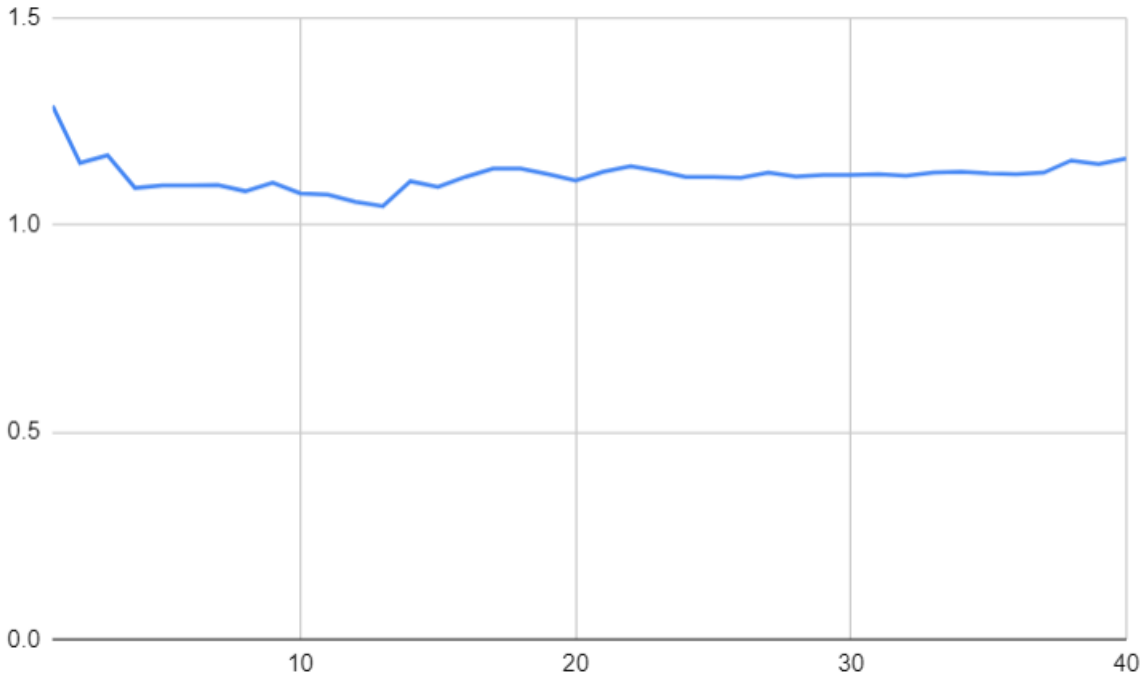


Figure 2: Neighborhood Size vs. Model Distance

My goal in finding the ideal neighborhood size is to maximize accuracy while minimizing the "model distance" which represents how "off" the incorrect predictions are on average. Based on a examination of the data, I concluded that the best neighborhood size was **22**. This provided the highest accuracy of any neighborhood size and had acceptably low model distance.

iii. Testing Ideal Weight Parameters

One of the aspects of my system is the weight parameters. These weight parameters help to make sure that the model respects that not all parameters have equal impact on the output. These (for now) are handpicked hyperparameters of the model, and so I also experimented with a couple different ideas for weighting. The effects that these weight changes had on accuracy are recorded below.

Weight-Style	Description	Correct	Incorrect	Accuracy	Prediction Distance
Default	Groups stats and tries to give each "group" equal weight.	144	56	72%	1.14286
Flat-weights	Totally flat weights with no bias towards any parameters	117	83	58.5%	1.13253
Favor-defense	Assigns a slightly higher bias towards defensive parameters	153	47	76.5%	1.12766
Favor-offense	Assigns a slightly higher bias towards offensive parameters	112	88	56%	1.14773
Hp-ac-dmg	Assumes that the most important parameters are hp,	138	62	69%	1.17742

	ac, and dmg, and that other parameters should be more lightly weighted.				
Balanced-defensive	Made by tweaking the most successful weight set from before. Favor-defense	154	46	77%	1.13043

After experimentation, I settled on the weights classification that I labelled "balanced-defensive" which favors defensive stats like hp and ac, and lowers the value on things like attributes, and dmg numbers. This appears to give the best results that I can determine by hand.

iv. Evaluating Completed Model

My final model has an evaluated test accuracy of 77%. It also has a predicted distance of 1.13043. This means that when the model *is* incorrect, it by and large only off by 1 level, with an occasional 2 level misprediction. Since the range of levels for monsters is -1 – 25 in my dataset, being off by 1 level doesn't significantly affect the usefulness of the outputs, meaning that this model is highly successful at predicting the correct level of a given creature. A quick analysis shows that if I extend the definition of correctness to "within 1 level" of the expectation, the model is actually 98% accurate.

V. Project Conclusions

i. Concluding Remarks

Overall, I think that this project was very successful. I managed to create an application that I feel is smooth, responsive, and useful. It accomplishes the goals I set out to achieve, and it does so with a high degree of accuracy. One of the features that I am the proudest of is the implementation of the weighting hyperparameter. As you can see in the table above when I was calibrating weights, if the weighting parameter didn't exist and all weights were totally flat, the model would have only been able to achieve 58% perfect accuracy. With it, however, we increase the accuracy to almost 80% with almost 98% "within one" accuracy.

ii. Project Limitations

This project was successful, but the approach does have a couple notable limitations that should be acknowledged.

- **Exceptional Cases:** Because the KNN model relies on knowledge of similar creatures, it isn't ever going to be particularly good at handling exceptional cases. In my case, I begin to lose accuracy when predicting the level of any creature between levels 20-25, because there are only one or two creatures in the entire database in those level ranges, and it's impossible to predict anything higher than that by virtue of how KNN operates.
- **Load Times:** The limits of building this application specifically for web deployment meant that I wasn't able to use most database frameworks and had to fetch my raw data over HTTP, which isn't particularly fast. This means that

unfortunately, there is a noticeable load time on startup. It isn't so slow as to necessitate that it be addressed, but it is noticeable.

- **System Specificity:** While I would like it if this tool could easily be extended to all manner of different tabletop systems, and the code is set up in such a way that this could be possible, the system as it stands is very specific to Pathfinder 2nd edition.

iii. Future Work

There are several things I would like to do with this project if I have the time to extend it beyond the scope of this class.

- **Spells:** Spells are an important aspect of a creature's capabilities, and certainly affect their power level. Due to the complexity of parsing what spells a creature has, and how powerful those spells are, prevented me from including it as a parameter in this project, but I would like to add support for it in the future.
- **Learned Weights:** For now, the weights layer is a manually set hyperparameter. If I got more time to improve this system, it would be interesting to try to build it to learn the weights automatically using something like linear regression. Layering the two models to achieve the highest possible accuracy could be very useful in producing more refined outputs.
- **Stat Autocomplete:** In order to make the tool more useful for building creatures, I would like to add a feature to the app that allows you to finish building your monster by "autocompleting" the missing fields using an autocompletion algorithm that fills in the missing fields with values that don't affect the overall level of the creature but do fill in the blanks.

VI. Appendix

i. Tables

Table 1: Neighborhood Size vs. Accuracy

Neighborhood	Correct	Incorrect	Average Misprediction	Accuracy
1	89	111	1.28829	44.5%
2	107	93	1.15038	53.5%
3	117	83	1.16867	58.5%
4	111	89	1.08989	55.5%
5	117	83	1.09639	58.5%
6	127	73	1.09589	63.5%
7	128	72	1.09722	64%
8	127	73	1.08219	63.5%
9	132	68	1.10294	66%
10	135	65	1.07692	67.5%
11	132	68	1.07353	66%
12	129	71	1.05634	64.5%
13	135	65	1.04615	67.5%
14	134	66	1.10606	67%
15	135	65	1.09231	67.5%
16	140	60	1.11667	70%
17	134	66	1.13636	67%
18	134	66	1.13636	67%
19	135	65	1.12308	67.5%
20	135	65	1.10769	67.5%
21	138	62	1.12903	69%
22	144	56	1.14286	72%
23	139	61	1.13115	69.5%
24	140	60	1.11667	70%
25	140	60	1.11667	70%
26	139	61	1.11475	69.5%
27	137	63	1.12698	68.5%
28	132	68	1.11765	66%
29	134	66	1.12121	67%
30	134	66	1.12121	67%
31	135	65	1.12308	67.5%
32	133	67	1.11940	66.5%
33	137	63	1.12698	68.5%
34	138	62	1.12903	69%
35	136	64	1.12500	68%
36	135	65	1.12308	67.5%
37	137	63	1.12698	68.5%
38	136	64	1.15625	68%
39	139	61	1.14754	69.5%
40	138	62	1.16129	69%