D&&D Merchant

Pricing Items in Dungeons and Dragons Using Machine Learning

Ian McConachie

March 2023

Contents

1 **Introduction** **1**

2 **Details of the Approach** **2**

2.1 Data Collection 2

2.2 Encoding Data 2

2.3 Experimenting With Architectures 3

3 **Results** **9**

4 **Discussion and Conclusions** **11**

4.1 Personal Learning 12

4.2 Future Work 12

5 **References** **14**

Introduction

Dungeons and Dragons is a popular tabletop role-playing game where one player (the Dungeon Master or “DM”) leads the other players (playable characters or “PCs”) through an interactive story usually set in a Tolkienesque fantasy world. Playing as a DM requires extensive planning and world building before each session in order to create a convincing and interesting setting for the story to take place in. This often entails constructing engaging game mechanics so that players can interact with the world you are creating.

One of the most common mechanics in D&D is the bartering between PCs and merchants who buy and sell items that can be used in game. As a long-time DM myself, I have always found this aspect of the game to be one of the hardest components to plan for. Often I want to offer an object that I think would be interesting to introduce, but I have no idea how to price it; or players may want to sell one of their items and I am forced to improvise an asking price.

In both of these scenarios, I would find myself combing through internet resources and reference books in order to find a similar item with a predefined price on which I could base the price of the new item.

The problem with just throwing out some random price is that the world of D&D doles out money in a certain way and a price too low or too high may cause issues that can wreck any planning or world-building you may have already completed. So what I wanted was a solution that would give DMs automated appraisals for any object by providing a price that makes sense in the D&D canon without the need for laborious research and guesswork.

This is where D&&D Merchant comes in. Pronounced “D ‘Logical And’ D Merchant,” it is an ai-powered tool built by DMs for DMs (apologies for the cheesy marketing line). The purpose of this application is to use data from established D&D merchant guides to create a machine learning model that can price items based on their name and category.

Over the course of the term my project has evolved many times. Originally I intended to construct a predictive multi-output Bayesian neural network which would take an input of encoded text and a discrete category and output a predicted price as well as assigning a discrete label that indicated the denomination that the price should be in. (**Note** that in this report ”denomination” refers to the type of D&D currency the prices are in which can be gold pieces (gp), silver pieces (sp), or copper pieces (cp)).

In the development process I wanted to build a simpler frequentist model first and then move on to the murkier waters of Bayesian inference. However, in the prototyping stage I ran into a series of obstacles which eventually led my final model to be a linear classifier that simulates a course price prediction. In the report below I detail the development process that got me to this end and what results I was able to achieve with the network I built.

Details of the Approach

Data Collection

This was the first step in creating D&&D Merchant and pretty much the only one that went according to plan. In order to train and test my model I had to create my own dataset as there was no existing data openly available for this task. To do this, I web scraped two sites: The Thieves Guild[6] and D&D Beyond[7]. The web scraping was done using Python's built-in html parser and the Beautiful Soup library[5]. Later, I increased the size of my dataset by using data from the Discerning Merchant's Price Guide [d] which was compiled in a csv file available here[2]. This data was easily extracted using Python's built-in csv library.

After collecting the data it had to be processed and cleaned. To do this, I wrote a script which got rid of duplicate data points, standardized the formatting between the 3 sources, and removed certain data points which were missing key pieces of information (e.g., had no definite price). I also reduced the total number of categories. This mostly consisted of making sure category names had the same grammar (e.g., "Gemstone" would be the same as "Gemstones") and grouping more specific categories into wider classifications (e.g., instead of having "Menu - Lunch" and "Menu - Breakfast" I grouped them together as "Food & Drink").

Encoding Data

Before I was able to do any modeling using the machine learning techniques we've learned in class I had to decide how to format my input and output data. My first thought was to use the BERT Tokenizer from Google[3] because it was available from Hugging Face[1] and has many of the mathematical/semantic properties you want from an encoding scheme. I found the longest non-zero encoding that the BERT Tokenizer outputted among all the names in the dataset and then I truncated all the tensors to this max size. I added one more index at the end of the tensor so that I could include the category of the item in the input vector. I then standardized the BERT encodings (subtracted the mean and divided by the standard deviation) and normalized the category integer (divided by the total number of categories). This resulted in the following encoding scheme:

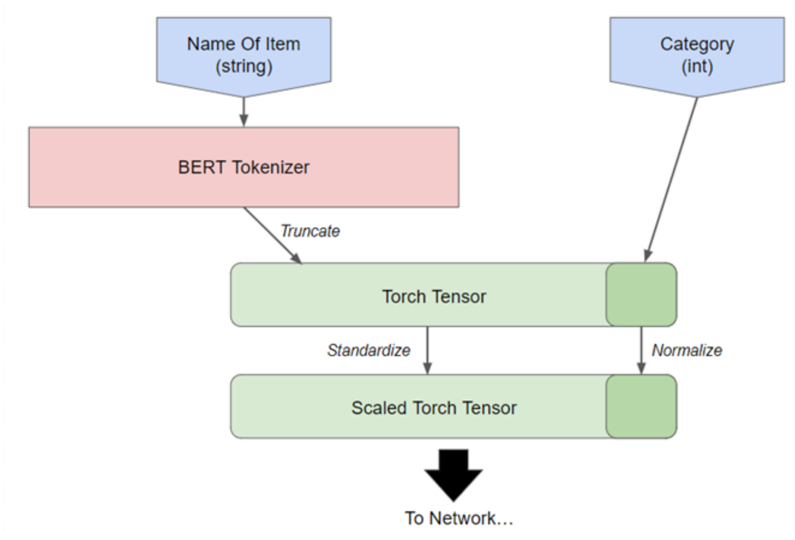

Figure 1: Tokenizing and encoding with BERT (rather than Ernie).

For the output values, I played around with standardizing the prices in the predictive schemes below and also simply converting their integer evaluations to floating point numbers so I could use them with Torch's built-in L1 loss function. Ultimately, encoding for the output values didn't matter much as I pivoted to a classification model (see Pivoting to Classification). In a way this is a form of encoding which consists of rounding the log10 value of the price—this method will be discussed more below.

Experimenting With Architectures

Multi-Output

The first network I built for this project was a multi-output linear network. The idea was that the network would take the name of the item and its discrete category (in a single tensor) and produce a prediction for price as well as an assigned classification for the denomination. I wanted to use a multi-output framework because it was something new that we hadn't covered in class, but also because using the same network for multiple outputs can be beneficial in terms of modeling. Specifically, the loss calculation and back propagation that goes on for each output adjusts the weights on the shared network and this sharing of information can result in more effective models[8]. For this task the two outputs were a discrete price prediction and an integer classification which led me to create the following prototype model.

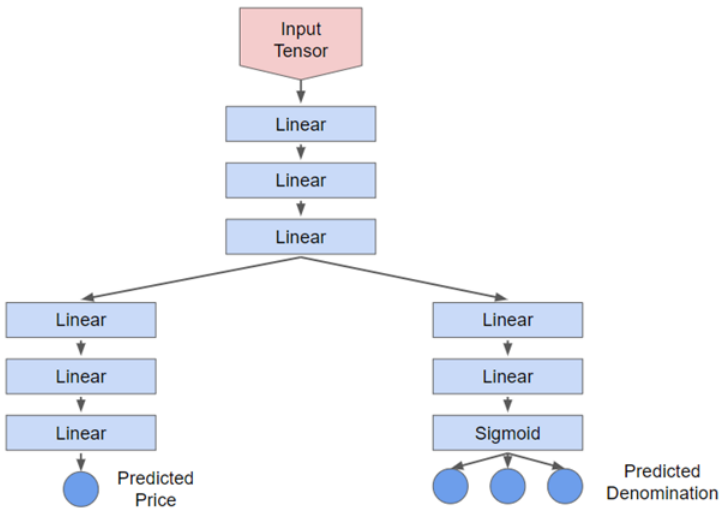

Figure 2: First iteration multi-output network.

What I found after training this model is that the model was very effective at predicting the denomination (usually about 93%). However, the interesting shape of the loss curve made me reexamine the data (See Figure 3 below).

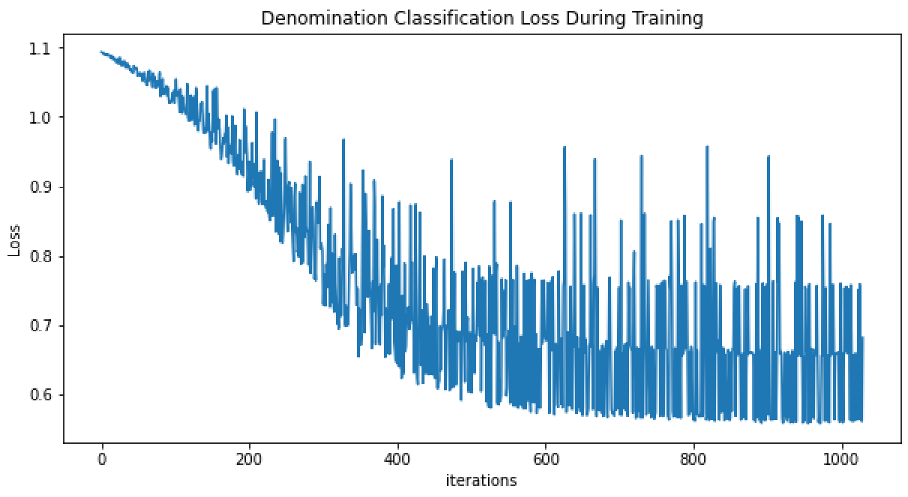

Figure 3: Loss on denomination classification over 1000 iterations.

The loss definitely converges, but you can see that the noise appears to increase as we have more iterations. I then took a look at the classifications the model was producing and saw that my model was consistently classifying all items as being priced in gold pieces or “gp.” That made me reexamine the distribution of denominations in my dataset which gave the following result:

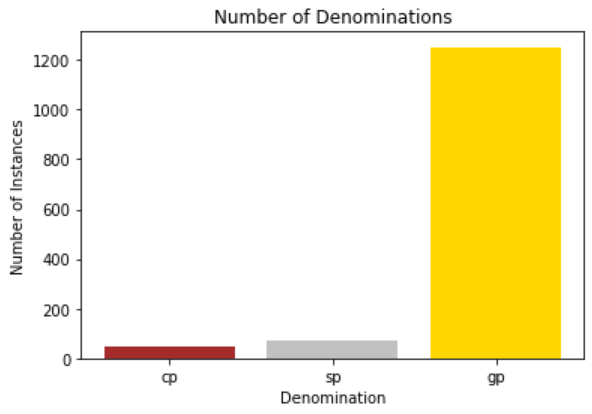

Figure 4: Distribution of denominations in the dataset.

This graph suggests a reason why the model most likely learned to have a strong bias towards classifying items in the “gp” category and learned this very quickly. The relative infrequency of the “cp” and “sp” classes make it so there needs to be some sort of anomaly in the data to assign a class that isn’t gp. The relatively small size of the dataset means that the network didn’t have enough information to learn to distinguish these rare cases. From this, I concluded that the model was going to price everything in gp, so it made sense to reduce the output to just the predicted price of the item.

Predictive Networks

After scrapping the idea of a multi-output network, I instead focused on training a single output predictive model. The multi-output experiment that was my first prototype produced very poor results for price predictions, so I thought I should change the input encoding as well. This led me to the following network architecture.

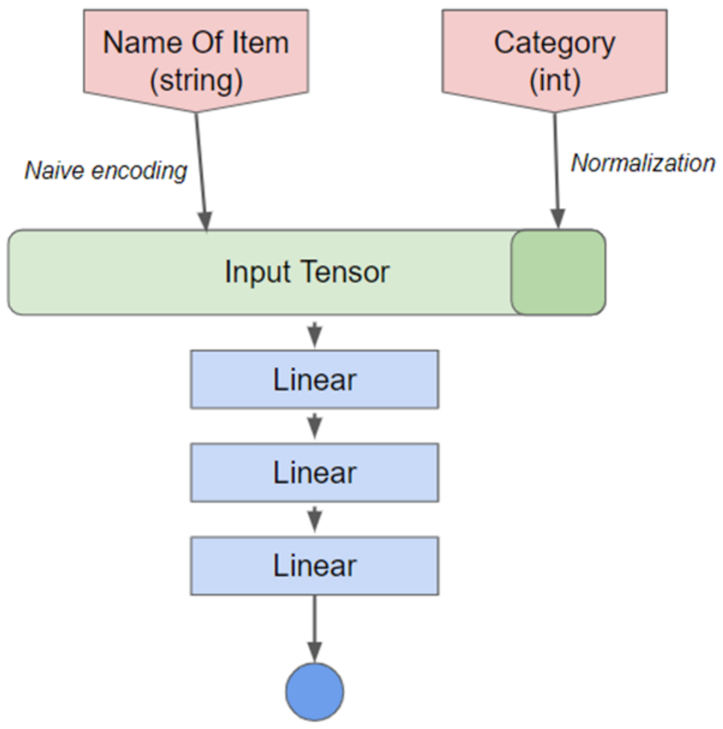

Figure 5: Basic linear predictive network.

You can notice in the figure above that instead of the BERT tokenizer we have a naive encoding step between the raw input values and the input tensor. This choice was made in order to see if the encoding was the problem with the previous model. The naive encoding was essentially to just enumerate all the ASCII characters used, change the string to an integer tensor, and then normalize. The encoding for the category input was kept constant.

The above design was the result of many iterations of trial and error with less than ideal results. I kept on finding that no matter how I tweaked the loss hyperparameters or formatted the output I simply could not get the model to converge on anything but very small subsets of the data. I played around with overfitting to these small subsets, but found when I tried to have my network learn on the whole training set I would get noisy, stagnant loss like what is shown below in Figure 6.

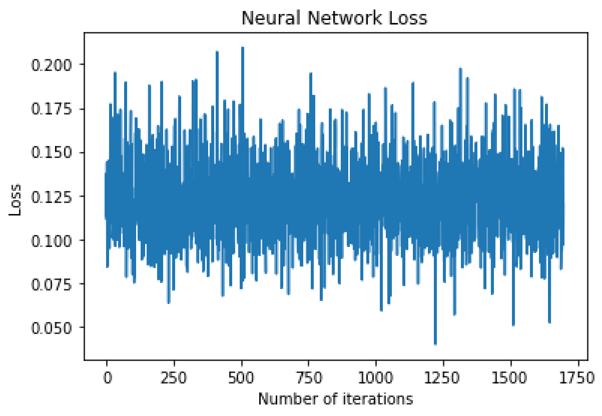

Figure 6: Loss on price prediction over 1750 iterations.

At this point I was considering changing the goal of my project completely because no matter how much debugged and tweaked the design of my networks I was not getting any kind of convergence with my loss. I was proud of the dataset I collected, so I still wanted to use it in my task, but I was considering switching to a generative task (see Final_Attempt3 in the scratch work directory) or something where I could better utilize a pre-trained model. Instead what I landed on was reducing the resolution of my predictions by mapping this predictive paradigm to a classification task.

Pivoting To Classification

After having a lot of difficulty with the predictive framework, I decided to modify the predictions to instead be classifications for the rounded down log base 10 value of the price (which ranges from 1 to 6) and is equivalent to the number of digits in the number. The idea behind this is that by making more coarse predictions I can reach some kind of convergence and be able to predict the price within a certain range. I also reasoned that the rounded log10 value of a price is one of the most important pieces of information in the prices. The difference between 3gp and 5gp is much less significant than the difference between 3gp and 30gp in this system of evaluation. So, I constructed the following architecture:

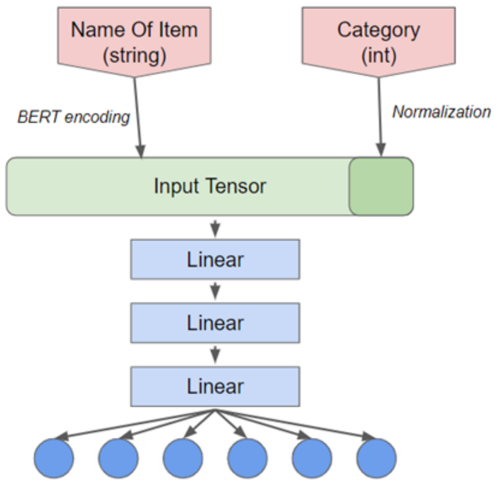

Figure 7: Classification model for price prediction.

One thing to note with the diagram shown above is that we have once again utilized the BERT encoding. This choice was made after switching to the naive encoding showed no notable improvement in the second predictive framework attempt. I reasoned I should go with the BERT encoding because it was more common in the NLP domain which this project falls under.

Results

For evaluating the model I split the data into a training and testing set using a 75-25 split. Specifically, for my dataset this gave us 1028 data points in the training set and 343 in the test set. For the final version of the classification model on which I evaluated my results I had 3 hidden layers and 4096 nodes in each of the hidden layers. For the training I had a batch size of 64, a learning rate of 0.001, stochastic gradient descent as my optimization function, and cross entropy loss as my loss function.

Below we see the loss graph over 200 epochs which took around 40 minutes to train. This shows some level of convergence with the flattening of the curve towards the end.

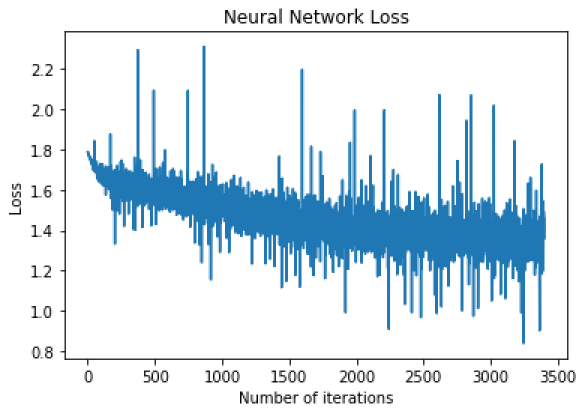

Figure 8: Loss on classification task over 3500 iterations.

The final classification accuracy on the validation set of the model produced was 35.28%. This is not as good of results as I may have hoped for with the amount of time I invested into the trial and error design process. But, it is notably better than random (which would be 16.67% with 6 classes). Below we can see how this accuracy changed over the 200 epochs:

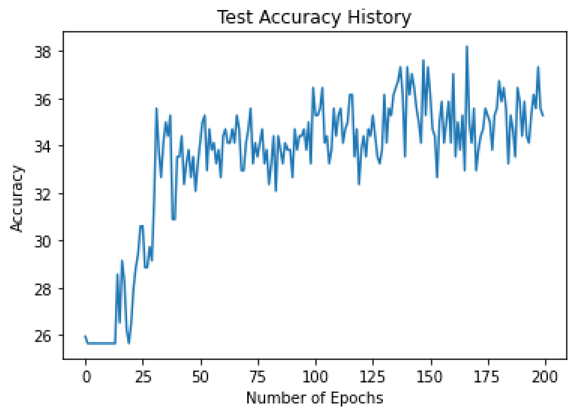

Figure 9: Test accuracy over 200 epochs of training.

The stagnation of accuracy that we see above towards the end of our epoch iterations should raise some red flags (in particular the noise and the inconsistent increase in accuracy). In order to investigate this behavior further we can construct a confusion matrix (see figure below).

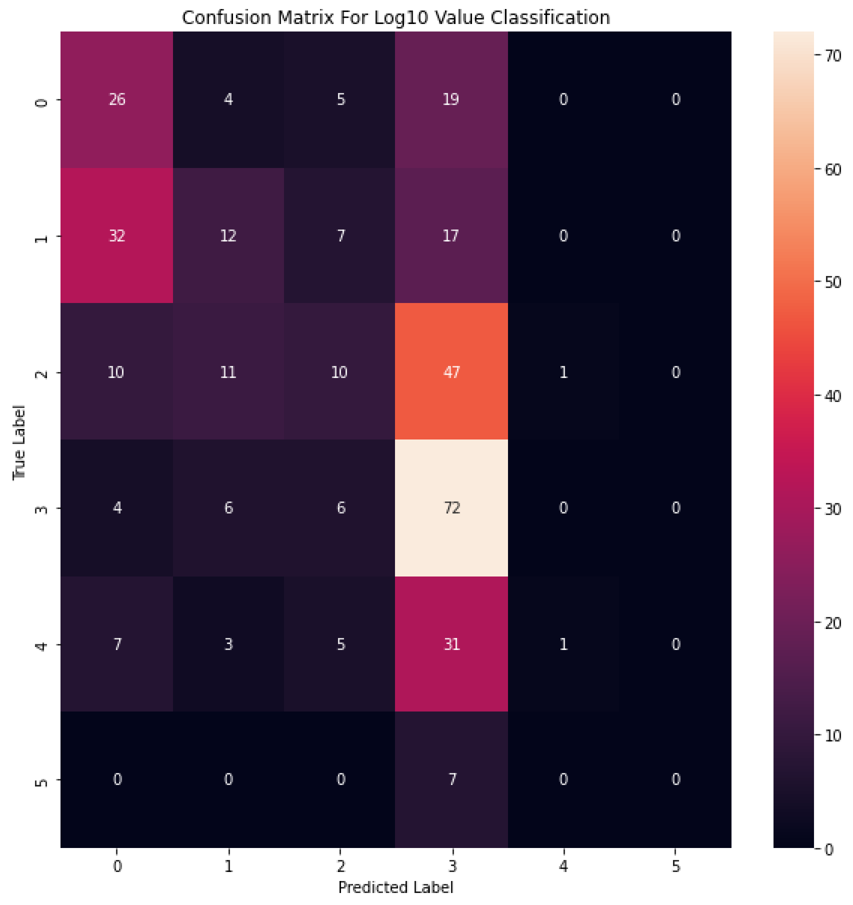

Figure 10: Confusion matrix for classification of log10 price values.

What we can see from the above confusion matrix is that the network most often predicts a class of 3 which corresponds to a rounded log10 number of 4 (i.e., 4 digits in the price). However it does not always predict this class and we can see that we have decent accuracy for classes lower than 3. In general, we get a pretty good rate of being at most one log10 value off—which is a somewhat promising result.

Discussion and Conclusions

Ultimately, this project demonstrated some of the pitfalls of applying machine learning to a task without first evaluating the feasibility of solving the problem with machine learning. In particular, here we saw that using the name of an item as a proxy for its price is a very limited metric—especially with a smaller sample of data points.

Even very similar names can have drastically different prices; For instance, among the data set we have ”Bag Of Devouring” with a price of 12,000gp and ”Bag Inside Flap” with a price of 1gp. As humans we can understand the difference between these two, but it is difficult for the computer to differentiate between the similar names without a much larger sample. This leads me to believe that much of the class prediction is based on the category of the item. However, even this is not the best proxy as items in the same category can also have very different prices.

The second issue that I believe held back the effectiveness of my model was the small size of the dataset and the lack of diversity among data points. In total, there were only 1371 unique items. As shown above (Figure 4) the denomination data had a very strong bias towards gp with hardly any cp or sp. We can also analyze the distribution of the different log10 powers in the data. This is done visually in the bar graph shown below.

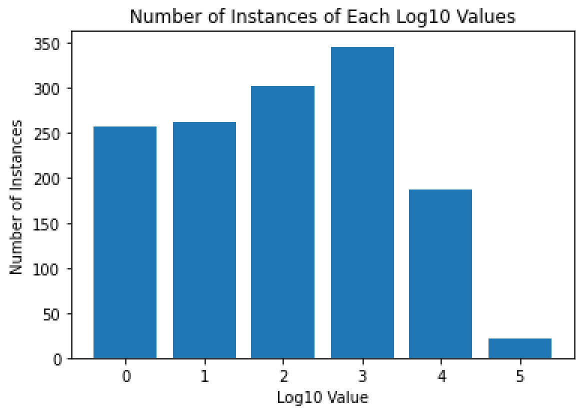

Figure 11: Distribution of log10 price values in the dataset.

This distribution aligns with what we observed in our confusion matrix. We see that log10 value of 3 is over-represented which tracks with the model’s tendency to assign that classification. The decent amount of data for lower log10 values also makes sense with what we see in Figure 4 above. Before, graphically representing this data, I expected the bias towards log10 3 classifications to be stronger. This bar graph shows that lack of diversity among log10 values was likely less of an obstacle than the poor proxies and small data sample.

Personal Learning

This project definitely did not go as expected, but as seen from the writing above it was a learning process. I believe the root cause of the chaos was taking on too large of a task without fully understanding what I was getting into at the start of the term. Despite the difficulties I faced, I definitely gained a lot in terms of knowledge and experience in the field of machine learning.

The biggest, and what I believe to be the most valuable, lesson I learned from this final project was what to do when a model isn't working. For all the homework assignments and exercises, we were playing with toy examples so ultimately we knew that the task at hand could be reasonably completed using tools available to us. When I ran into a wall (or rather several walls) predicting the prices in this self-made dataset—there was really no resource that I could turn to that would tell me exactly how to fix it. In fact, there was no one who could even tell me if it was fixable. The only way out was to mess around in the sandbox of several drafts of colab notebooks until I could produce some sort of notable result. This instilled in me the adaptability necessary to do any work in the machine learning field.

I also ended up learning a lot of concrete development skills in this iterative experimentation process. In particular, I was able to figure out how to create multi-output networks as well as predictive networks. Even if they were not effective, they were operational (I tested them by over-fitting on very small subsets of the overall data). I also learned how to web scrape and data engineer in order to create my own dataset. This was not part of the curriculum for this course, but it is definitely a valuable skill in the broader field of data science.

Future Work

Ultimately, the obstacles I ran into when developing my models prevented me from experimenting with Bayesian neural networks like I had originally intended to. Bayesian inference is a personal interest of mine, so this is definitely at the top of potential future experiments.

If I wanted to improve the performance of my current model, I would likely have to grow the dataset and perhaps integrate more input information for my predictions to learn from. This would be difficult because I have already searched for data on most accessible websites for D&D, but considering the popularity of the game I am sure there exists more sources out there.

At the beginning when I selected this task, I was hoping that this work could be the starting point for many future projects where I applied machine learning to existing D&D data to make automated DM tools. After playing around with this merchant data, I realized that it may take a lot larger datasets than what are already out there in order to make effective AI tools. I still think this would be a cool application of AI, so maybe in the future I will expand D&&D by improving the merchant price predictor and creating new applications.

In the end, I was glad I got to apply machine learning techniques to a hobby of mine. Although debugging caused a lot of stress towards the end, I definitely gained a lot from this experience and I am proud of what I did accomplish.

References

1. BERT. Hugging Face. (2023). Retrieved March 20, 2023, from https://huggingface.co/docs/transformers/model_doc/bert

2. Cruz, R. (2019, September 18). Accessing D&D magic items in the Guild! Pathfinders' Guild of Berkeley. Retrieved February 1, 2023, from https://guildberkeley.wordpress.com/tag/discerning-merchants-price-guide/

3. Devlin, Ming-Wei, C., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv.org.

4. Eisinger, D. (2022, July). The Discerning Merchant's Price Guide. Dungeon Masters Guild. Retrieved February 1, 2023, from https://www.dmsguild.com/product/205126/Discerning-Merchants-Price-Guide

5. Richardson, L. (2023). Beautiful Soup documentation. Beautiful Soup Documentation - Beautiful Soup 4.12.0 documentation. Retrieved March 20, 2023, from https://www.crummy.com/software/BeautifulSoup/bs4/doc/

6. Shops and pricing for D&D 5E. The Thieves Guild. (2023, March 21). Retrieved February 1, 2023, from https://www.thievesguild.cc/shops

7. Wizards of the Coast LLC. (2023). Equipment . D&D Beyond. Retrieved March 20, 2023, from https://www.dndbeyond.com/equipment

8. Xu, Shi, Y., Tsang, I. W., Ong, Y.-S., Gong, C., & Shen, X. (2020). Survey on Multi-Output Learning. IEEE Transaction on Neural Networks and Learning Systems, 31(7), 2409–2429. https://doi.org/10.1109/TNNLS.2019.2945133