Classifying *Dungeons & Dragons* Races Using Machine Learning Algorithms

Reg Gonzalez
Erik Jonsson School of Engineering and Computer Science
The University of Texas at Dallas
Richardson, Texas
rdg170330@utdallas.edu

***Abstract*—D&D is a popular fantasy tabletop role-playing game created by Dave Arneson and Gary Gygax. In the game, players have the ability to create their own characters. One of the features players can choose for their character is their race—ranging from all sorts of fantastical humanoids and creatures. Other features like height, weight, speed, and abilities also get to assigned. This project will attempt to classify randomly-generated D&D characters' races based the aforementioned features. Three different machine learning algorithms will be trained and tested: logistic regression, K-nearest neighbors, and decision trees. They will be evaluated on several metrics such as accuracy, precision, recall, etc. and the results of those metrics will determine which algorithm is best at classifying this dataset. It was concluded that decision trees were best, followed by K-nearest neighbors, and lastly, logistic regression. Further research could be conducted to improve the implementation of these algorithms or implement brand-new ones.**

I. INTRODUCTION

A. Machine Learning Classification

One primary use for various machine learning algorithms is classification. Classification is the process of sorting data based on labels (in the case of supervised learning) and on similarity clusters (in the case of unsupervised learning).

There are many algorithms that can classify data; however, the three chosen for this project were logistic regression, K-nearest neighbors, and decision trees. These algorithms were specifically chosen because they are all very different in the way they classify data. Logistic regression is a linear classifier that models probability, meaning that its output is the probability that a sample from the data belongs to a particular class [1]. K-nearest neighbors (KNN), on the other hand, makes no such probabilistic outcomes. Instead, the outcome is the class label of the majority of the k closest data points to the data point being classified [2]. Finally, decision trees create a tree structure in which the internal nodes of the tree represent features/attributes of the dataset and leaf nodes represent a specific class label (in the case of pure nodes) and the majority class label (in the case of non-pure nodes) [3].

These algorithms will be trained and tested on various metrics such as accuracy, precision, recall, etc. Depending on their results, one of the algorithms will be determined the best at classifying this D&D-related dataset.

B. D&D Background and Dataset

In order to understand how/why this project is classifying the dataset the way that it is, it is

important to understand the fundamentals of D&D and this dataset. D&D is a popular fantasy role-playing tabletop game where players get to create their own characters. The character creation process can be very intensive and detailed, though for the purposes of this project and report, the only important features that will be discussed are a character's race, traits, and their abilities (and by extension, ability scores).

Because this is a fantasy game, races vary a lot. Players can either be standard humans, tieflings (hybrids between humans and infernal beings), dragonborn (humanoid dragons), elves, etc. [4]. Players can also assign heights, weights, and speed of their characters.

Similarly, players also assign ability scores to their characters. Abilities "provide a quick description of every creature's physical and mental characteristics" [5]. There are six main abilities: Strength, Dexterity, Constitution, Intelligence, Wisdom, and Charisma. Players assign a numerical value to each ability for their character, generally ranging from 1 to 20 [5].

The dataset used for this project contained 10,000 randomly-generated D&D characters. Ability scores were generated using the 4d6 Drop Lowest method. In this method, the player rolls four six-sided die, drops the lowest number, and adds the three remaining numbers together for that ability score [6]. Racial ability modifiers were also used to boost ability scores. For example, tieflings add +2 and +1 to their Charisma and Intelligence scores respectively [4]. Heights, weights, and speed also have racial modifiers that were added when those values were generated [7]. Lastly, each row of the dataset contains the race of the character. These races were picked with uniform probability, ensuring that there is a roughly equal number of rows for each race.

II. PROBLEM DESCRIPTION

As mentioned previously, the goal of this project is to perform classification on this particular D&D character dataset. Because the value of the "race" feature in the dataset can take multiple values, this will be a multi-class classification problem instead of a simple binary one. The algorithms used are logistic regression, KNN, and decision trees. These algorithms were chosen because they each have a unique way of classifying data: logistic regression models probability, KNN looks at its closest neighbors (and does not create a model), and decision trees divide data in groups based on features and maximum information gain.

In order to evaluate these algorithms, the metrics accuracy, precision, recall, and F1 score will b used. A confusion matrix will also be displayed so that anyone who decides to run the program can visually see how the data was classified per algorithm. Simply using accuracy as the sole statistic would not be sufficient because it does not provide a holistic representation of the algorithm's classification abilities.

Based on the results of the algorithms, it will then determine which one was the best at classifying the D&D dataset. More detail regarding the results will be discussed in the "RESULTS AND DISCUSSION" portion of the report.

III. TECHNIQUES AND ALGORITHMS USED

A. Logistic Regression

The first algorithm that coded and tested was logistic regression. This is a discriminative classifier, which means that it creates a model for $P(Y|X)$—where Y represents the classification label and X represents the given data. For binary classification (i.e., the presence of only two classes, often labeled 0 and 1), the logistic regression formula is often denoted as: $P(Y = 1|X) = \frac{1}{1+e^{w_0+\sum_{i=1}^{n} w_i x_i}}$, where $P(Y = 1|X)$ represents the probability of the classification being in class 1 given the input

data X, the w_i's represents the weights of the model, and the x_i's represents the values of the features in the dataset. In the binary case, since $P(Y = 0|X)$ can be implied from $1 - P(Y|X)$, the equation for $P(Y = 0|X)$ can be described as: $P(Y = 1|X) = \frac{e^{w_0+\sum_{i=1}^{n} w_i x_i}}{1+e^{w_0+\sum_{i=1}^{n} w_i x_i}}$. To assign the class label Y=0, the formula is $1 < \frac{P(Y=0|X)}{P(Y=1|X)} \rightarrow 0 < w_0 + \sum_{i=1}^{n} w_i x_i$. Likewise, to assign the class label Y=1, simply do the reciprocal of the prior equation [1].

However, because this is multi-class classification, the methodology is slightly different. In multi-class classification, there is the notion of a "One-vs-All" (sometimes called "One-vs-Rest") approach. In this approach, several different binary classifiers, say *c*, are trained. These classifiers determine whether or not a test sample is part of a particular class or not. To determine which class this test sample belongs to, the algorithm chooses the maximum probability obtained by the *c* classifiers [8]. When the algorithm begins, the parameters (weights and biases_ are initially set to 0. Once training starts, the model, using test examples, will predict classification labels using the following equation: $\hat{y} = \frac{1}{1+e^{-wx+b}}$. This is very similar to the equation defined previously, and is essentially the Sigmoid equation, which is the functional form $P(Y|X)$ is assumed to be. This equation is useful because it outputs a value between 0 and 1, which is desirable because this algorithm is looking for probabilities.

As the algorithm continues, gradient descent is used by calculating the gradient of the error function, in this case MSE (mean squared error) is the error function. Calculating the gradients is important because they're used to update both the weights and biases—the parameters the algorithm hopes to optimize. The equations for updating the weights and biases are: $w = w - \alpha * dw$ and $b = b - \alpha * db$ respectively, where *w* represents the weights, *b* represents the bias, *dw* represents the gradient of the weights, *db* represents the gradient of the biases, and α is the learning rate. The learning rate determines how fast or slow the algorithm approaches the minimum [9]. If the learning rate is low, then the algorithm will take longer to approach the minimum. Conversely if the learning rate is high, then the algorithm will reach the minimum faster; however, it might oscillate on points surrounding it. In the end, the algorithm terminates when either the maximum number of iterations is reached or the parameters reach a stable value and no longer update.

B. K-Nearest Neighbors

The next algorithm programmed was KNN. This algorithm is simpler than logistic regression because it does not create a model. Instead, it stores all of the training examples and when a testing example is queried, it finds the closest training examples to that testing example based on some distance metric. KNN is the most popular algorithm instance-based learning [2].

The algorithm gets the name "*K-nearest* neighbors" because the K represents the number of training examples for the algorithm to examine. For instance, if K = 5, then the algorithm would look at the five closest training examples to the testing example. K is also a hyperparameter, meaning that it is something that the programmer has to continuously test to find the best value. Hyperparameters differ from parameters. Hyperparameters are settings that need to be defined prior to the algorithm's execution. Examples of this include K and the learning rate (seen in logistic regression). Parameters, on the other hand, are tuned as the model is being trained; they're not values the programmer directly sets. Examples of this include weights and biases (also seen in logistic regression) [2].

Like other ML algorithms, there are two phases to KNN: training and testing. However, as explained earlier, these phases do not build a

model. In the training phase, the algorithm simply stores the training examples. In the testing phases, it uses some distance metric in order to determine the K-nearest points to the testing example [2].

When it comes to the distance metric, different equations can be used depending on the dataset and what the programmer believes to be the best representation of what the "distance" should be. In this project, Euclidean distance was used: $D(\boldsymbol{X}, \boldsymbol{Y}) = \sqrt{[\sum_{a=1}^{d}(x_a - y_a)^2]}$. There was no inherent reason to use the Euclidean distance for this dataset—another formula like the Manhattan distance could've been used as well. It was simply used for convenience [2].

When KNN finds the K-nearest points to the testing example, it classifies it using the majority label. For example, if K = 5 and 3/5 of the closest training examples were labeled "tiefling" and 2/5 were labeled "gnome," the classification of that particular testing example would be tiefling since that is the majority. Unlike logistic regression there is not stopping criteria in which this algorithm terminates. When the algorithm returns the majority label, that's it.

C. Decision Trees

The final algorithm in the project/report is decision trees. Like the previous two algorithms, decision trees are another supervised learning algorithm—that is, it classifies data based on labels. It's similar to logistic regression in the sense that it builds a model; however, this time it's not based on probability. Decision trees, as the name suggests, build representations of the data in the form of trees. They could also be thought of as a set of rules. Internal nodes in the tree represent the features of the dataset (in this case, it'd represent attributes like height, weight, speed, strength, etc.). Leaf nodes represent the output of the tree. If the leaf node contains multiple class labels (i.e., it's not pure), then the value of that leaf node would be the majority class label, similar to KNN. If the leaf node is pure, then the value would naturally be the class label in that node. Lastly, the edges of the tree represent the splitting criteria of the internal node. For example, let's say that one of the internal nodes is the intelligence feature. If an example's intelligence score was less than or equal to 10, the algorithm would traverse down the left-side of the tree. Contrarily, if the intelligence score was greater than 10, the algorithm would traverse down the right-side of the tree [3].

The idea of decision trees is intuitive, but the problem comes when determining which attribute is best to split the tree on. This is where metrics like information gain and entropy come into play. Information gain is the metric that lets us know how important an attribute is to the dataset. It is defined by the following: $IG = E(parent) - average\ E(children)$, where the function E() represents the entropy. Ideally, information gain is maximized; that is, splitting the tree on attributes that are the most useful [3]. Entropy can be thought of as the lack of order in set (or subset) of data. The value of this can range from 0 to 1. Datasets that have more classes in them have higher entropy while datasets that have less classes in them have lower entropy. If a dataset just consists of one class, then the entropy is 0. Meanwhile, if a dataset has an equal proportion of classes (e.g., a binary set consisting of 50% of the examples labeled as 0 and 50% of the examples labeled as 1), then the entropy is 1. The equation for entropy is: $E = \sum_i -p_i * log_2 p_i$, where p_i represents the probability of class *i* (i.e., the proportion of class *i* to all classes in the dataset). While the goal was to maximize information gain, the goal is to minimize entropy. Minimizing this value is important because the end goal is to have pure leaf nodes. Pure leaf nodes occur when the entropy is 0, meaning that the [subset of] data only consists of one class [3].

The stopping criteria for logistic regression is when the weights and biases no longer update and KNN has no stopping criteria other than returning the class label of the K-nearest training examples. To the contrary, decision trees can have multiple stopping criteria. The obvious one being when the algorithm reaches a pure leaf node. If all the examples in the node belong to one class, there is no use in splitting it further. Another criterion is the max depth. Depending on how complex the tree is, a hyperparameter can be set that stops the tree from growing any further. The next, and last, stopping criterion used in the project takes a look at the minimum number of examples in a node. If splitting a node causes it to underflow this minimum number of examples, then the algorithm will not allow it to grow anymore [10].

IV. RESULTS AND DISCUSSION

A. Changing Parameters within the same Algorithm

Before delving into the results, it's important to note that the table of results presented in this report does not contain all the results. This is subsection of them that shows the best evaluations of each respective experiment. To see all the results, please see the document "CS 6375 Project – Results."

The metrics that were used to evaluate these algorithms were accuracy, precision, recall, and the F1 score. There were also confusion matrices displayed by the program for each algorithm so that users can visually see how the well the algorithm classified each race.

Accuracy tests how well the algorithm classified the true positive and true negative examples (i.e., examples that truly belonged to a class vs. those that didn't truly belong to a class). This is perhaps the most widely-used performance evaluation metric. However, accuracy can be misleading since it doesn't give the full picture. Depending on the distribution of the class labels in the dataset, accuracy can give a misleading number. For example, in a 2-class problem, if 99 of the examples were labeled class 0 and 1 was labeled class 1, if the model predicted all the examples to be in class 0, then the accuracy would be 99%. This, of course, isn't accurate since the model didn't predict anything for class 1 [11].

To give a more holistic view of an algorithm's performance, the metrics precision, recall, and F1 score (sometimes called F1-measure) are going to be introduced. Precision is calculated as $p = \frac{TP}{TP + FP}$, where *TP* is the true positive classifications and *FP* is the false positive ones. This metric answers the question "What proportion of positive identifications was actually correct?" [12]. Conversely, recall is defined as $r = \frac{TP}{TP+FN}$, where *TP* was defined earlier and *FN* represents the false negative classifications. This metric answers the question "What proportion of actual positives was identified correctly?" [12]. Lastly, the F1 score is a combination of both precision and recall; it attempts to combine the two and is generally described as the "harmonic mean" between the two values. F1 score is calculated as $F = \frac{2TP}{2TP+FP+FN}$ [11]. Overall, these metrics give a more comprehensive overview of the algorithms' performance.

Before comparing how the algorithms fared against each other, it is important to see how changes in the parameters of the algorithms affect the results. First, logistic regression. There were two main parameters that could've been changed: the learning rate and total number of iterations. Three different values for learning rate were tried (0.001, 0.01, and 0.1). The number of iterations (1000) were kept the same during this processing. Increasing the learning rate had a drastic effect on the results. The accuracy plummeted from 54.6% to 38.6% and 32.9% respectively. This might be an indication that at this range—as explained earlier—the algorithm oscillates around the minimum

because the learning rates are too high. The recall and F1 score also dropped significantly, but surprisingly the precision increased from ~60% to ~80%. The number of iterations was also changed from 1,000 (base level) down to 500 and then up to 2,000. Compared to the changes in learning rate, this had a lesser effect on the overall results. The accuracy still stayed about 50% and likewise, the precision, recall, and F1 score were relatively stable.

Next, KNN. Here there was only one hyperparameter to change, which was the value of K itself. The values tested were 5, 10, 15, 20, and 19. As the value of K increased, so did the accuracy, precision, recall, and F1 score; however, not to any dramatic measure. For instance, the difference between accuracy from K = 5 to K = 20 was only ~3%. The metrics seemed to stabilize around K = 15, with no significant difference after that. After some more testing, K = 19 returned the "best" results—again, relatively speaking.

Lastly, decision trees had two hyperparameters to change: the minimum number of samples per node and the maximum tree depth. The values of minimum number of samples tested were 3, 5, and 1; during these tests, the maximum tree depth stayed the same (at 100). The difference in results between these tests was insignificant. Though overall, going from 3 to 5 made accuracy and recall decrease while precision and F1 score increased. Going down from 3 to 1 decreased accuracy, precision, recall, and the F1 score, but again, to no significant degree. Changes to the maximum tree depth were also made: going from 100 down to 50, and then 100 up to 200. Decreasing the tree depth, not surprisingly decreased accuracy, precision, recall, and the F1 score. Increasing the tree depth increased the precision and F1 score, but decreased the recall. That being said, like with the minimum number of samples per node, changes to this hyperparameter were insignificant at best.

Machine Learning Algorithm	**Parameter(s)**	**Results**
Logistic Regression	Learning rate = 0.001 Iterations = 2000	Train/Test Split: 80:20 Size of dataset: 10,000 Accuracy: 0.5465 Precision: 0.6227017606815628 Recall: 0.5455614503077837 F1 Score: 0.49790568850624095
KNN	K = 19	Train/Test Split: 80:20 Size of dataset: 10,000 Accuracy: 0.6335 Precision: 0.6275920912764241 Recall: 0.6288817473992449 F1 Score: 0.621962746662528
Decision Trees	Min # of samples = 3 Max tree depth = 100	Train/Test Split: 80:20 Size of dataset: 10,000 Accuracy: 0.677 Precision: 0.6713338748380095 Recall: 0.6724877813055057 F1 Score: 0.6712170265704344

Fig. 1. Table showing results using the three different machine learning algorithms

B. Differences in Experiments using Different Algorithms

Figure 1 shows the results of the best experiments per algorithm. Again, to view all the results see the "CS 6375 Project – Results" document. Comparing the three algorithms, it's clear that decision trees were the best. They not only had the best accuracy, but the best precision, recall, and F1 scores as well. On the other hand, logistic regression performed the worst: all of its metrics except for precision are just within the 50% threshold. KNN performed in the middle of the two.

The reason why logistic regression failed to perform any better is because the algorithm is a linear classifier. That is, it works well with data that is linearly-separable; however, it does not work well with data that isn't [1]. For the D&D dataset, it certainly was not linearly-separable. As explained earlier in this report, the data for the dataset was more or less generated at random. There were racial modifiers applied to some ability scores and features like height, weight, and speed, but because of the nature of the data, it was not possible to find a linear decision boundary that classified the dataset well.

In regards to KNN, the results were evidently better than logistic regression, but still not great. It increased ~10% in accuracy vs. logistic regression and had similar increases with recall and F1 score. The precision; however, was almost the same as logistic regression. The reason why KNN performed better is two-fold. Firstly, it classifies non-linearly-separable data better. In fact, it's shown that people can get fairly complex decision boundaries if they use a simple KNN model [2]. Also, because the algorithm is classifying the algorithm based on the testing examples' nearest neighbors, this is where the impact of the racial modifiers come in handy. For example, as stated earlier, tieflings gets a +2 and +1 modifiers to their Charisma and Intelligence scores. This means that examples that have a higher Charisma and Intelligence scores might be more likely to be classified as tieflings versus a race like dwarfs, who have no such modifiers to Charisma or Intelligence. The same logic could be applied to the height, weight, and speed features. For instance, dragonborn have heights in the 70s-80s range while gnomes have a significantly lower height ranging in the 30s-40s [4]. Big differences like this could've made it easier for KNN to classify the testing examples based on the specificities of the racial modifiers and traits.

Lastly, decision trees. This algorithm performed the best out of the three and had solid results across the board (comparatively speaking). Like KNN, decision trees are not bounded by linearly-separable data, which is why it fared better than logistic regression. And while the differences between the results of KNN and decision trees aren't too far off, it is noticeable. The reasoning for why this algorithm did better is similar to KNN in the sense that racial modifiers and traits are what made testing examples more identifiable. Again, looking at features such as height, weight, and speed in the dataset, noteworthy differences between each race can be observed. The reason why this algorithm—and by extension, the other two algorithms—aren't able to classify the dataset more accurately is because the remaining features are more or less randomly generated. The ability scores are generated using the 4d6 Drop Lowest method, which is inherently random because the method requires the player to roll four dice to assign ability scores. The racial modifiers for abilities might be somewhat helpful, but they are not significant enough to provide any meaningful distinctions between the races. When the value of an ability score can range from 0-20, a +1 or +2 modifier doesn't tip the scale that much one way or another. That said, through procedures like information gain and entropy, decision trees found that focusing on more perceptible features like height, weight,

and speed made it easier to classify the dataset.

Overall, decision trees performed the best, followed by KNN, and lastly logistic regression. This was because they are not bounded by linearly-separable data and through information gain and entropy, are able to make important distinctions, like using height, weight, and speed as determining factors in classifying a character's race as opposed to the ability scores.

V. CONCLUSIONS AND FUTURE WORK

There are many different machine learning algorithms that can classify datasets. The ones that were programmed and evaluated in this project were logistic regression, KNN, and decision trees. These algorithms were chosen because they each have a unique way of performing supervised learning classification: logistic regression creates a model for probability, KNN doesn't create a model but instead looks at the nearest training examples, and decision trees creates a model based on splits in features determined by information gain and entropy.

The dataset for this project was a D&D dataset that included 10,000 randomly generated characters. The end goal was to predict a character's race based on the features: height, weight, speed, strength, dexterity, constitution, intelligence, wisdom, and charisma. The values for the latter six features were randomly generated, but all racial modifiers and traits were taken into account. For example, half-orcs gain +2 and +1 to their Strength and Constitution abilities. Similarly, the height, weight, and speed of half-orcs varies differently from those of half-elves, halflings, dragonborn, etc.

Based on the testing conducted, decision trees proved to be the best classifier for the dataset, followed by KNN, and lastly logistic regression. Decision trees were the best because they were not limited by non-linearly-separable data and could use metrics like information gain and entropy to determine that features such as height, weight, and speed were more important than the ability scores. Conversely, logistic regression performed poorly because it requires linearly-separable data to work well.

In terms of any future work for this project, modifications to the parameters and hyperparameters can always be experimented with more. Finding new values for K, the learning rate, the minimum number of samples per node, etc. can always be improved upon. When it comes to KNN specifically, perhaps a different distance metric could've yielded better results. The one used in this project was Euclidean distance, but other metrics like the Manhattan distance could always be used.

Of course, logistic regression, KNN, and decision trees are not the only machine learning algorithms out there. More algorithms could be coded and compared to these results to determine which algorithm is truly the best at classifying this dataset. This certainly was not an exhaustive test and many more iterations of this testing could be done with different algorithms and parameters.

REFERENCES

[1] Nagar, Anurag. "Logistic Regression." Class lecture, Machine Learning, The University of Texas at Dallas, Richardson, Texas, April 1, 2024.

[2] Nagar, Anurag. "Instance-Based Learning." Class lecture, Machine Learning, The University of Texas at Dallas, Richardson, Texas, April 1, 2024.

[3] Nagar, Anurag. "Decision Trees." Class lecture, Machine Learning, The University of Texas at Dallas, Richardson, Texas, April 1, 2024.

[4] "Character Races for Dungeons & Dragons (D&D) Fifth Edition (5e) - D&D Beyond," D&D Beyond. https://www.dndbeyond.com/races

[5] "Using Ability Scores," Using Ability Scores.

https://www.dndbeyond.com/sources/basic-rules/using-ability-scores#AbilityScoresandModifiers (accessed May 05, 2024).
[6] “How to roll for stats in DnD 5E | Dice Cove,” dicecove.com, Feb. 13, 2021. https://dicecove.com/how-to-play/how-to-roll-for-stats/ (accessed May 05, 2024).
[7] R. Kibble, “Height, Age, and Weight in DnD 5e: How 38 Races Stack Up,” Dec. 03, 2021. https://blackcitadelrpg.com/height-age-weight-5e/ (accessed May 05, 2024).
[8] Yeh, Chris. “Binary vs. Multi-Class Logistic Regression | Chris Yeh,” *Github.* (accessed May 05, 2024).
[9] AssemblyAI, “How to implement Logistic Regression from scratch with Python,” AssemblyAI. Sep. 14, 2022. Accessed: May 05, 2024. [Online].
[10] AssemblyAI, “How to implement Decision Trees from scratch with Python,” AssemblyAI. Sep. 14, 2022. Accessed: May 05, 2024. [Online].
[11] Nagar, Anurag. “Model Evaluation.” Class lecture, Machine Learning, The University of Texas at Dallas, Richardson, Texas, April 1, 2024.
[12] Google Developers, “Classification: Precision and Recall | Machine Learning Crash Course | Google Developers,” Google Developers, Mar. 05, 2019. https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall