The College of Wooster

Open Works

Senior Independent Study Theses

2019

# The D&D Sorting Hat: Predicting Dungeons and Dragons Characters from Textual Backstories

Joseph C. MacInnes
*The College of Wooster*, jmacinnes19@wooster.edu

Follow this and additional works at: https://openworks.wooster.edu/independentstudy

 Part of the Artificial Intelligence and Robotics Commons, and the Numerical Analysis and Scientific Computing Commons

## Recommended Citation

MacInnes, Joseph C., "The D&D Sorting Hat: Predicting Dungeons and Dragons Characters from Textual Backstories" (2019). *Senior Independent Study Theses.* Paper 8635.

This Senior Independent Study Thesis Exemplar is brought to you by Open Works, a service of The College of Wooster Libraries. It has been accepted for inclusion in Senior Independent Study Theses by an authorized administrator of Open Works. For more information, please contact openworks@wooster.edu.

© Copyright 2019 Joseph C. MacInnes

# The D&D Sorting Hat: Predicting Dungeons and Dragons Characters from Textual Backstories

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree of Computer Science in the
Mathematics and Computer Science Department at The
College of Wooster

by
Joe MacInnes
The College of Wooster
2019

**Advised by:**

Nathan Sommer



THE COLLEGE OF
WOOSTER

© 2019 by Joe MacInnes

# Abstract

Dungeons and Dragons is a tabletop roleplaying game which focuses heavily on character interaction and creating narratives. The current state of the game's character creation process often bogs down new players in decisions related to game mechanics, not a character's identity and personality. This independent study investigates the use of machine learning and natural language processing to make these decisions for a player based on their character's backstory - the textual biography or description of a character. The study presents a collection of existing characters and uses these examples to create a family of models capable of predicting a character's class, race, and attribute scores. The accuracy and limitations of these models are discussed, but they represent a first step in abstracting away some of the more tedious parts of character creation for new players.

# Acknowledgments

First and foremost, I'd like to thank Professor Sommer for his guidance over the course of this project. I'd also like to thank Dr. Byrnes for supporting my interest in computer science inside and outside of the classroom during my years at Wooster.

I would not be where I am without my family and friends. To my father, mother, and sisters I am grateful for your love and continued interest in what I do. To my friends Tommy Bacher, Gianni Ciccarelli, Ben Galen, Hunter Hoffman, Alex Kowitz, Thanh Nguyen, Dylan Orris, Kasey Plamondon, Kasey Porter, Scott Stoudt, Zane Thornburg, Ben Turner, and Phillip Wells I am indebted for both the laughter we've shared and counsel you've given me.

Lastly, I thank my dog, Stevie Wonderful, for showing me that even if you can't always see what's in front of you, it doesn't mean you can't enjoy yourself along the way.

# Contents


Abstract v

Acknowledgments vii

Contents ix

List of Figures xi

List of Tables xiii

CHAPTER PAGE

1 Introduction 1
1.1 Problem Outline . . . 2
1.1.1 Data Collection . . . 2
1.1.2 Model Construction . . . 3

2 Natural Language Processing Background 5
2.1 History . . . 5
2.2 Term Frequency Inverse Document Frequency . . . 7

3 Machine Learning Background 9
3.1 Supervised Learning . . . 9
3.1.1 Supervised Models . . . 11
3.1.1.1 Decision Trees . . . 11
3.1.1.2 Decision Tree Example . . . 14
3.1.1.3 Random Forests . . . 16
3.1.1.4 Multilayer Perceptron . . . 17
3.1.2 Logistic Regression . . . 20
3.1.3 Support Vector Machines . . . 23
3.2 Unsupervised Learning . . . 23
3.2.1 Feature Engineering . . . 23
3.2.1.1 Dimensionality Reduction . . . 24
3.2.1.2 Word2Vec . . . 25
3.2.2 Clustering . . . 28

4 D&D&Data 31
4.1 Acquiring Data . . . 32
4.2 Preprocessing . . . 34
4.3 Data Analysis . . . 36

5 Predicting Characters: Approach and Results 41
5.1 Class and Race . . . . . 41
5.1.1 Baseline Models and Metrics . . . . . 43
5.1.2 Dealing With Imbalance . . . . . 46
5.1.3 Model Selection . . . . . 47
5.2 Attribute Scores . . . . . 54
5.2.1 Baseline Models and Metrics . . . . . 59
5.2.2 Model Selection . . . . . 60
6 Conclusion 63
APPENDIX PAGE
A Character Sheet 67
References 69

# List of Figures


| Figure | | Page |
|---|---|---|
| 3.1 | Decision surfaces of decision trees with varying heights classifying wines. | 12 |
| 3.2 | Trained decision tree for predicting Dungeon of Corrosion survivability. | 16 |
| 3.3 | The structure of an MLP [22] | 18 |
| 3.4 | The sigmoid activation function: $\frac{1}{1+exp(-x)}$ | 19 |
| 3.5 | Example 2-dimensional word embeddings. | 26 |
| 3.6 | Creating the first few Word2Vec training examples for a toy text. | 27 |
| 3.7 | Clustered and actual wine class labels among wines with the varying phenol and flavanoid levels. | 29 |
| 4.1 | Distribution of word counts (up to the 99th percentile) among processed backgrounds. | 37 |
| 4.2 | Distribution of character race and class among processed backgrounds. | 38 |
| 4.3 | Distribution of character class within races among characters with recognized character class and race. | 39 |
| 4.4 | Distribution of attribute scores among characters. | 40 |
| 5.1 | Multiplication of TF-IDF matrix and word embedding table to create a new feature space. | 43 |
| 5.2 | Baseline classifier confusion matrices. | 45 |
| 5.3 | Confusion matrix of a logistic regression model trained to classify character class without sampling. | 47 |
| 5.4 | Confusion matrices for character class classifiers. The title of each matrix indicates the model trained what feature space was used. | 48 |
| 5.5 | Confusion matrix of random forest and logistic regression ensemble to classify character class. | 50 |
| 5.6 | Confusion matrices for character race classifiers. The title of each matrix indicates the model trained what feature space was used. | 52 |
| 5.7 | Confusion matrix of random forest and logistic regression ensemble to classify character race. | 53 |
| 5.8 | Distributions of attribute scores among class. | 56 |
| 5.9 | Distributions of attribute scores among race. | 57 |
| 5.10 | Raw errors of MLP and baseline on each attribute. | 62 |

A.1 D&D 5th Edition Character Sheet . . . . . 67

# List of Tables


| Table | | Page |
|---|---|---|
| 3.1 | Training examples for characters who entered the fictitious Dungeon of Corrosion. | 15 |
| 5.1 | The number of characters in training and testing sets for class and race classification. | 42 |
| 5.2 | Performance metrics of baselines for character class and race. | 46 |
| 5.3 | Performance metrics of classifiers for character class. | 51 |
| 5.4 | Performance metrics of classifiers for character race. | 54 |
| 5.5 | Training and testing sets for attribute regression. | 58 |
| 5.6 | Performance metrics of attribute score regressors. | 61 |

*CHAPTER 1*

# Introduction

Since its introduction in 1974, Dungeons and Dragons (D&D) has become perhaps the most recognizable entry in the tabletop roleplaying game genre. [1] In the traditional D&D setting, players take on the role of a fantasy character and guide them through adventures led by another player acting as the game master (something of a narrator). As a roleplaying game, the focus of D&D is on the interactions between the players, non-player characters, and the world. A player's character is a crucial piece of the game; it defines these interactions.

Characters in D&D are represented by two components: their character sheet and backstory. The character sheet holds information relevant to the mechanics of the game like attribute scores, skill proficiencies, and character class and race. The backstory, on the other hand, is a purely textual representation of the character. It often reads like a character biography or introduces the character through a short story. While the character sheet is necessary for actually playing the game, the backstory is what allows a player to relate to the character and effectively roleplay.

Unfortunately, the current state of character creation can lead to flat characters, creating lackluster roleplaying experiences. The current process often neglects the backstory. New players spend the first part of creating a character rolling dice

[1] As evidenced by its numerous products topping the list of Amazon's bestselling fantasy gaming books and current lead in number of games on roll20, a popular online tabletop gaming service. [1] [9]

to determine stats and choosing categories; by the time they are done checking boxes on the character sheet, the backstory feels like something of an afterthought. In the most recent edition of D&D there is even a small list of vague character backgrounds a player can choose from (e.g. Noble, Knight, and Sage). When the player brings this character to a session, it feels more like stats-on-a-stick than an actual individual.

This project aims to flip the character creation process on its head and use machine learning models to generate relevant character sheet information from textual backstories. This gives players a more character-centric character design process. Such a system allows a player to define their character from a storytelling perspective rather than constructing it with game mechanics as the forethought.

## 1.1 Problem Outline

The goal of this project is to produce a family of machine learning models to analyze character backstories and output relevant character sheet information. In order to limit scope, the project's domain is initially limited to 5th edition characters (the most recent set of rules), but later chapters will discuss the complications involved with this and resulting decisions. Wrapped up in the goal are two distinct problems: data collection and model construction.

### 1.1.1 Data Collection

Construction of any machine learning model relies on a body of data. This data is needed to train, validate, and test models. As a general rule of thumb, more data correlates to better models, and the objective here is to collect as much D&D character information as possible. In an optimal world, collected data will represent

any type of character players might think to create, since machine learning models struggle with data unlike the kind they are trained on.

While this is discussed further in future chapters, the lack of easily accessible datasets or APIs for D&D characters present the foremost hurdle in this problem: finding data sources. Once found, the data must be collected from these sources and represented.

### 1.1.2 Model Construction

The meat of the project is accurately generating character sheet information from the collected data. This problem involves training a family of models on collected backstories in order to make predictions about the categories and stats one finds in D&D character sheets. This involves a series of tasks.

The first task is that of feature generation and selection. Discussed more fully in future chapters, this essentially involves extracting information from the character backstories and restructuring them in such a way that machine learning models can effectively learn from them.

Following this, different model structures with different parameters must be compared. There are many different types of machine learning algorithms, each with their own pros and cons. In order find effective models, a variety should be considered. Finally, an analysis of the selected models performance is given.

*CHAPTER* 2

# Natural Language Processing Background

This chapter outlines the history of natural language processing (NLP) - the task of extracting information from human languages in order to analyze their meaning. Since D&D backstories are composed in human language, NLP is quite relevant to the project. The chapter discusses the general evolution of the field as well as its various phases in order to give context to the project's approach in analyzing D&D backstories. Finally, it describes an NLP technique used in the project.

## 2.1 History

The field of natural language processing was born in the early 1950s. With the onset of the Cold War, the governments of the United States and the Soviet Union were both interested in effective translation of each other's language. Around this time, Locke and Booth published the first book on machine translation which discussed how the cryptanalysis techniques of World War II might be used to perform machine translation [16]. Their book also provided a list of references that represented essentially the entire body of work on the subject. With this work, and the catalyst of contemporary politics, funding for NLP (specifically machine translation) exploded.

Throughout this period much of the work done followed Noam Chomsky's theory of transformational generative grammar [8]. As Chomsky's linguistic

research progressed, he would come to say that language is composed of a deep structure and a surface structure. A sentence, for example, has a deep structure - the semantic relationships between parts of a sentence - and a surface structure - the sentence as it appears.[1] The deep structure of a sentence is transformed into the surface structure using a set rules, and this set of rules is called a grammar. These rules take a string of at least one symbol (sort of like variables) and any number of terminals (the actual lexical elements of the sentence, like words and punctation) and transform them into another string of symbols and terminals. A sentence starts out as a single, root symbol and, after continually applying rules, evolves into a string consisting of only terminals - the surface structure. Thus, a language can be represented as a grammar which possesses the rules necessary to convert a sentence's deep structure into surface structure. Knuth's article [15] offers a more in-depth look at such grammars, providing examples and describing the ways in which a machine can process them. The bottom line, however, is that Chomsky's theories offered a computationally friendly, and formal definition of language for early NLP researchers.

Work in these early days started with simple word-to-word dictionary lookups for translation. Researchers soon realized that syntax often preceded the task of semantic analysis since sentence structure can play such a role in meaning. This led much of the research to be dedicated towards computational grammars capable of parsing sentences - breaking a sentence down into a tree of syntactic components [14]. These grammars consisted of complex, hand-written rule sets. In 1966, however, the infamous Alpac report effectively shut down NLP research [21]. It assessed the progress of machine translation as a whole, and concluded that efforts had not matched expectations.

[1] As with any discussion related to linguistics, it should be noted that semantics refers to the meaning behind symbols in a language (e.g words), while syntax refers to the grammatical structure of a language.

Research in NLP regained traction alongside the boom of statistical analysis in the 1980s which was supported by an increase in computational power and memory. At this time, NLP began to focus more on corpus linguistics - the study of a language using existing texts of that language. [2] A perfect example of this new kind of NLP is Kenner and O'Rourke's Travesty [11]. Travesty was a PASCAL program that used Markov chains to generate character sequences using the learned character frequencies from a textual example. In their paper, Kenner and O'Rourke describe feeding Travesty the names of 29 English cities. The program's output produced sequences of characters that looked remarkably like legitimate city names, such as Chire and Nettlewett. The authors noted that, simply by using character frequencies, they could pick up on recognizable styles in textual examples.

Since then, NLP research has continued on this trajectory, and today a large amount of it is data driven and based in machine learning. Because of the sheer quantity of data available and its many forms, much current work uses machine learning techniques capable of extracting meaning from unstructured, unannotated language. Zang and LeCun, for example, were able to create a convolutional neural network to perform a variety of NLP subtasks from purely character level input (i.e without knowledge of words, phrases, or any other syntactic or semantic structures) [24]. In subsequent chapters, some of these current techniques are explored.

## 2.2 Term Frequency Inverse Document Frequency

The approach taken by this project in analyzing D&D character backstories falls much more within the realm of corpus linguistics than Chomsky style formal linguistics. To underline this (and provide a concrete example of statistical NLP),

[2] Here, a corpus is defined as a collection of documents.

this section outlines a technique used in the project to extract meaning from the character backstories: term frequency inverse document frequency (TF-IDF).

TF-IDF is an extension of the concept of IDF, introduced by Karen Spärk Jones [13]. IDF essentially measures how rare a word is in a corpus, and by correlation, how much information it carries. Assume a corpus has a set of documents $D$ and we want to know the IDF of a term, $t$. In this case, the IDF can be calculated as:

$$idf(t, D) = \log \frac{|D|}{|d \in D : t \in d|}$$

In other words, it is the logarithm of the ratio of documents to documents in which the term appears. Words like "the" and "a" will have very low IDF values since they will most likely appear in every single document. Rare words, however, will have much higher IDF values, indicating that they carry a lot of information in relation to the corpus as a whole. TF, on the other hand, is a measure of how frequent a word is a single document. TF-IDF combines the two by multiplying TF and IDF. This value is an indication of, for a given word, how important it is for a given document.

Each document in a corpus will have a TF-IDF for every word that appears in the corpus (words in the corpus, but not in the given document will have a TF-IDF of 0 for that document). Therefore, these TF-IDF values can be represented as a matrix, where the columns are words and the rows are documents. A TF-IDF matrix can be a useful way to extract information from relatively small corpora, and proves to be extremely valuable for the models trained in the project.

*CHAPTER 3*

# Machine Learning Background

This chapter outlines and defines machine learning and grounds this background by walking through some of the models used in the project. Broadly, machine learning can be divided into two categories, supervised and unsupervised learning, and this chapter will be discuss each in turn. At its core, however, machine learning revolves around finding relationships within data to perform some sort of task. The following discussion relies on [12], [7], and [20] for its theoretical underpinnings.

## 3.1 Supervised Learning

In supervised machine learning, the task at hand is usually some kind of prediction - given unseen data, place a label on that data. The types of labels one might predict for a piece of data create two sets of supervised problems: classification and regression. Classification problems deal with predictions when the results have discrete values. Often times this is binary, like predicting whether or not it will rain on a given day, but this does not have to be the case. One might wish to classify the majors of incoming college first-years. Regression problems require predicting continuous values. A regressor might be used to estimate the value of a company's stock.

In its essence, prediction comes down to mapping an input space to an output

space. This is where the 'learning' in supervised models takes place. A model is fed training data - pairs of input and already-known output - and tries to find the optimal mapping between the two. We call each dimension of the input a feature, while each dimension of the output is called a target or label. After training, the model can be used on new examples without labels to predict them. Usually, the efficacy of the model is calculated by testing its performance on a subset of the known data not used during training.

Most supervised learning models learn through a process of optimization; they continually adjust their parameters and check their performance on the training data until satisfied. At the core of optimization is a loss function (e.g. mean-squared error), which a model can use to numerically assess its performance on some training data. Given a loss function, a model can calculate its error over a set of training examples and adjust its parameters accordingly. In general, consider a supervised learning problem where we have a set of training examples, $N$, where each example is of form $(\boldsymbol{x}, \boldsymbol{y})$. $\boldsymbol{x}$ is a vector of features and $\boldsymbol{y}$ is a vector of targets. Let $L$ be a loss function and $E$ be the error represented by the aggregate loss over all training examples. We seek to find some $f$ in,

$$E = \sum_{i=1}^{N} L(f(\boldsymbol{x}_i), \boldsymbol{y}_i) \tag{3.1}$$

such that $E$ is minimized. A significant portion of constructing a good model comes from what loss function is used.

The choice of loss function and the parameters of the model lead to two distinct types of supervised learning models: generative and discriminative. Models in the two groups both learn to make predictions, but differ slightly in what these predictions represent. The difference is perhaps easiest to understand in the context of classification, where the predicted probabilities of each target represent two different concepts.

Discriminative models cut the feature space up along decision boundaries such that the different regions will contain each class. Thus, they model the conditional probability of each class with the given input, $P(\boldsymbol{y}|\boldsymbol{x})$, where $\boldsymbol{y}$ is a class and $\boldsymbol{x}$ is a feature vector. Generative models, however, consider the actual distribution of each class and model this. In this way, they model the joint probability of each class, $P(\boldsymbol{y}, \boldsymbol{x})$.

To illustrate the difference, consider a musician who is given information about a song (key, tempo, etc.) and is asked to predict if the song is rock, country, or classical. If the musician tries to play a song from those features based on songs they've already learned and chooses whichever genre results, this would be a generative approach. If the musician simply compared the features to those of songs they've seen in the past to make a decision, this would be a discriminative approach.

Most of the models used in the project are discriminative, since discriminative models work better than generative ones on small amounts of data.

### 3.1.1 Supervised Models

Having discussed the basics of supervised machine learning, this section provides example models and their algorithms to ground the concepts.

#### 3.1.1.1 Decision Trees

Decision trees are the first and foremost example due to their success in the project. A decision tree performs a series of splits on the features of some training data. Each of these splits creates subsets of the data, which are then split on individually. Good splits are ones which lead to the most homogeneity in the targets of the resulting subsets. In the case of classification, these are subsets with a majority of one class, while in regression, they are subsets with tightly clustered values. Once the splits have been made, a decision is made by applying a new instance of data to each of

these splits until a leaf node is reached. The average value or class of the examples in that leaf node is the prediction.

The series of splits a decision tree performs is essentially just cutting up the feature space. Each split is a cut, separating a portion of the feature space into several regions where each region represents a leaf node in the decision tree. Each of these cuts is a hyperplane in the feature, but with higher dimensional spaces, this become harder to visualize. Thought of this way, it's clear that decision trees are discriminative models.

Figure 3.1 visualizes the results of training two decision trees on a toy dataset of different wines [2]. The first of these trees has a maximum depth of three, while the second has no maximum depth. Maximum depth is a way of telling a decision tree to stop training (i.e how many splits to consider). Without a maximum depth, a decision tree needs some sort of base case to stop splitting data. Usually, this is when the subsets of a split are totally pure (or some purity threshold is reached). In classification this means all of the training examples in the subset are one class, while in regression the target values are tightly enough clustered that they fall under some set threshold.

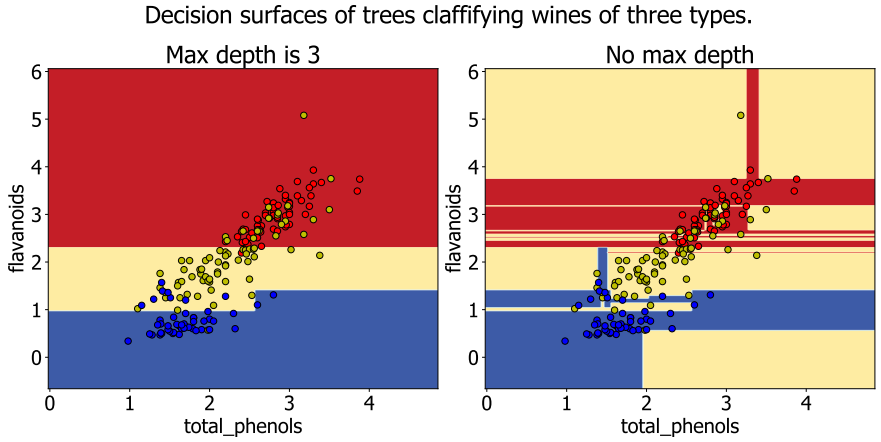



**Figure 3.1:** Decision surfaces of decision trees with varying heights classifying wines.

Trees with too many splits, however, usually lead to serious overfitting - a trained model that performs well on the training data but poorly on the testing data. Indeed, in the case of decision trees, having no maximum depth will lead to a model that can correctly predict every single training instance. Visually, it creates a fractured feature space with many tiny regions. To counteract this, two approaches are used: max depth and pruning. Using pruning, once decision trees are fully grown (i.e they cannot split any more), they continue to remove splits (merging subsets) from the bottom of the tree until their accuracy on a testing set stops improving. Using max depth, the trees grow until they hit a maximum depth. In this way, maximum depth and pruning are regularization techniques - a method to reduce the complexity of the model and thus prevent overfitting. Of course, a maximum depth too low can lead to underfitting - a model that fails to capture the important nuances of the training data. The trees in figure 3.1 reflect these two opposites by showing an underfit, small tree and an overfit, large tree.

With this in mind, the backbone of decision trees becomes how to perform the splitting. There are several flavors of decision tree algorithms (the trees in the project use CART - classification and regression trees), but all of them are greedy. At each split, they choose the feature that will lead to the purest subsets. There are two metrics commonly used to determine purity in a set of a data - entropy and the Gini index - but since models in the project use the Gini index, that is the metric discussed here.

The Gini index measures the likelihood of misclassifying a random instance from a dataset by using the distribution of labels in that set. The index falls in the range of $[0, 1)$ where impurity increases from a totally pure 0 to an impure 1 - total purity meaning all the instances are of the same class. For the set of training examples $T$ with $C$ classes, assume $i = \{1, 2, ..., C\}$ and let $p_i$ represent the proportion of training instances in $T$ with class $i$. We calculate the Gini index of $T$ as:

$$Gini(T) = \sum_{i=1}^{C}(p_i)(\sum_{j\neq i} p_j) = 1 - \sum_{i=1}^{C} {p_i}^2 \tag{3.2}$$

Thus, while decision trees don't have a loss function, they do use an optimization metric - in this case, the Gini index - to determine their parameters (the splits made). In order to perform a split, the algorithm iterates over each feature and calculates the weighted sum of the Gini indexes of the subsets resulting from a split on that variable. Consider a set $T$ of training examples and the set of sets, $S$ resulting from a split on some feature where the union of all sets in $s_i \in S$ is $T$ and the intersection of all sets in $S$ is the null set. The weighted average of the Gini indexes is then:

$$\sum_{i=1}^{|S|} \frac{|s_i|}{|T|} Gini(s_i) \tag{3.3}$$

Whichever feature results in the lowest weighted average Gini index is selected to split on, and the algorithm continues using the new subsets.

#### 3.1.1.2 Decision Tree Example

With the structure of decision trees outlined, the following shows the process of creating a decision tree on a fabricated example. The task is to predict whether future characters will survive the terrible Dungeon of Corrosion based on the following 10 training examples (the Dungeon of Corrosion and any references to it are made up for the purposes of this chapter). The features include each character's level, armor and class, while the target is a binary yes or no.

| Level | Armor | Class | Survive? |
|---|---|---|---|
| 1 | Leather | Wizard | Yes |
| 2 | Plate | Fighter | No |
| 2 | Leather | Wizard | Yes |
| 1 | Chain | Fighter | No |
| 2 | Chain | Wizard | Yes |
| 2 | Plate | Fighter | No |
| 1 | Leather | Fighter | No |
| 1 | Leather | Wizard | Yes |
| 1 | Plate | Wizard | No |
| 2 | Chain | Fighter | Yes |

**Table 3.1:** Training examples for characters who entered the fictitious Dungeon of Corrosion.

To perform the first split, we must find which of the three features results in the purest split using the Gini index. In the case of armor and class, this is straightforward, but level is a continuous value. In order to calculate the Gini index of splitting on a continuous feature, each possible midpoint for the values is selected and split upon (where the split is a binary, greater-than-or-less than decision) and the best one is chosen. In this case, the only possible midpoint is 1.5 so that is the only split required for assessing splitting on level. The weighted Gini index of each split are as follows:

$$WeightedGini_{levelSplit} = \frac{5}{10}(1 - (\frac{2}{5}^2 + \frac{3}{5}^2)) + \frac{5}{10}(1 - (\frac{3}{5}^2 + \frac{2}{5}^2)) = 0.48$$

$$WeightedGini_{armorSplit} = \frac{4}{10}(1 - (\frac{1}{4}^2 + \frac{3}{4}^2)) + \frac{3}{10}(1 - (\frac{3}{3}^2 + \frac{0}{3}^2)) + \frac{3}{10}(1 - (\frac{1}{3}^2 + \frac{2}{3}^2)) = 0.28$$

$$WeightedGini_{classSplit} = \frac{5}{10}(1 - (\frac{1}{5}^2 + \frac{4}{5}^2)) + \frac{5}{10}(1 - (\frac{4}{5}^2 + \frac{1}{5}^2)) = 0.32$$

The split on armor results in the purest subsets, so it is the initial split. Of the three subsets, the process is repeated for those of the chain and leather armor

types (this time without armor as a feature). The plate armor subset, however, is completely pure, so it becomes a leaf node with the prediction value of 'no.' Figure 3.2 shows the full tree once the process has completed and pure leaf nodes have been found.

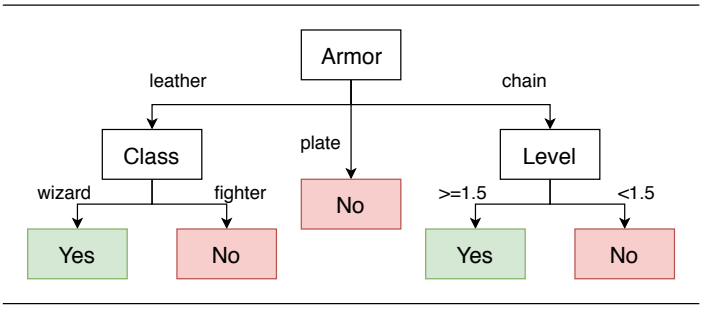



**Figure 3.2:** Trained decision tree for predicting Dungeon of Corrosion survivability.

It should be noted that the CART algorithm cannot take categorical values, so those in the example would be split into $n$ new features where $n$ is the number of categories for that feature (e.g. plate, leather, and chain for armor). The values for these new features would become 0 or 1 to indicate presence. In this way, each split becomes binary.

#### 3.1.1.3 Random Forests

Random forests are extensions of the decision tree model, and are an example of ensemble learning. Ensemble methods are those that combine many weak models in order to construct a strong model. When it comes to decision trees, this involves training many trees as opposed to just one. Each tree is trained using a random subset of both the training data and the features, and thus any individual tree is weaker than a regular decision tree. A random forest is a collection of such trees, and its output is the average output of all its trees.

Random forests are often preferred to single decision trees because of their higher stability. Decision trees themselves are quite unstable in that, if a few new

data points are added to the training data, they can easily change the splits made and lead to a vastly different tree. In the case of random forests, while each tree is slightly more biased since it has fewer features to pick from and examples to guide the splits, the averaging of output results in a lower variance.

#### 3.1.1.4 Multilayer Perceptron

A multilayer perceptron (MLP) is a simple feedforward artificial neural network discussed here because of its effectiveness in the project when performing regression tasks associated with predicting character stats. An MLP consists of an input layer, at least one hidden layer, and an output layer where each layer is fully connected (more on this in a moment). Each of these layers consists of a number of neurons whose output is real value. Thus, each layer of the perceptron can be represented as vector of real values. The first layer - the input layer - has the same dimensionality as the feature space of the training set, while the output layer has the same dimensionality of the prediction target. The dimensionality of the hidden layers is a hyperparameter of the model.

Between each layer is a matrix of weights, which, through matrix multiplication, maps values from one layer to the next. Assume there is an MLP with a set of layers, $L$ and a set of weight matrices $W$. Consider, that in $L$, there are two arbitrary layers, $L_i \in \mathbb{R}^m$ and $L_{i+1} \in \mathbb{R}^n$. To say that these layers are fully connected means that there exists a weight matrix $W_i \in W$ such that $W_i$ is $m \times n$ and maps $L_i$ to $L_{i+1}$; a neuron in $L_i$ has a weight for every neuron in $L_{i+1}$. Figure 3.3 shows this structure using an MLP with a 3-dimensional input layer, a single 4-dimensional hidden layer, and a 2-dimensional output layer. The arrows between each layer represent the weight matrices.

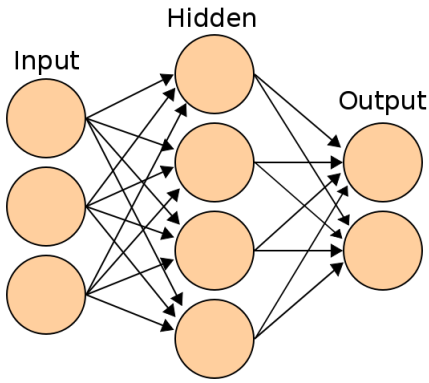



**Figure 3.3:** The structure of an MLP [22]

.

In order to make a prediction, data is fed through the MLP, starting with the input layer. At each layer, the output from the last layer is multiplied by the weight matrix corresponding to the next layer, a bias term is added to the sum, an activation function is applied to each element in the resulting vector, and finally this output is passed off to the next layer. The output of the final layer is the prediction.

To further build on the constructions above, assume $f$ to be the activation function associated with layer $L_{i+1}$ and let $\boldsymbol{b} \in \mathbb{R}^n$ be the biases for the neurons in $L_{i+1}$. We calculate the output of $L_{i+1}$ as

$$L_{i+1} = f \odot (L_i W_i + \boldsymbol{b})$$

where $\odot$ represents the elementwise application of $f$.

The activation function is a key part of the MLP and allows it to capture non-linear relationships in the data. Without an activation function, a perceptron is essentially a linear regression model; the input is multiplied by a series of weights to obtain the output. The activation function map the output of the matrix multiplication at each step to a new range, performing something of a squishification. Then the value in this range corresponds to how much a neuron has "fired." Figure 3.4 is an

example of the sigmoid function, a commonly used activation function which maps values to the range $(0, 1)$.

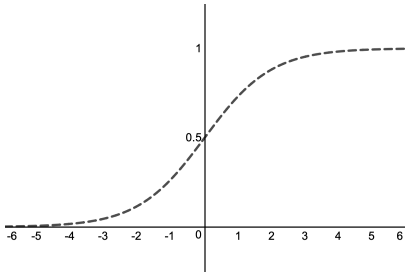



**Figure 3.4:** The sigmoid activation function: $\frac{1}{1+exp(-x)}$

.

The accuracy of an MLP depends on the weights and biases associated with each layer since these are ultimately what decide how the input is transformed as it is fed through the network. Thus, an untrained MLP is initialized with random weights, and, during training, these weights are slowly adjusted to lead to better performance on the training data. The most common method to achieve this is gradient descent through backpropagation. At its core, this approach determines how poor the output is for some training examples, figures out the impact of each weight matrix and bias vector in the network and tweaks them accordingly.

In order to determine the performance of the model on some training data, a loss function is required. Mean squared error (MSE) is a common choice for regression problems and makes a useful example. Consider a training data instance with target vector $\boldsymbol{y} \in \mathbb{R}^n$ and assume the predicted values of the training instance are represented by $\boldsymbol{y}' \in \mathbb{R}^n$. The MSE of this prediction is

$$\frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{y}_i - \boldsymbol{y}'_i)^2$$

Usually, an MLP is applied to all training examples, or a batch of them, and the average error is taken before any corrective actions are taken. Consider a set of

targets, $T$, for some training data, and a set of corresponding predictions $P$, such that each $p_i \in P$ and $t_i \in T$ is $n$-dimensional. The average MSE for the targets is

$$\frac{1}{|T|}\sum_{j=1}^{|T|}\sum_{z=1}^{n}(T_{jz} - P_{jz})^2$$

After determining the average loss, comes the process of weight adjustment. First, the partial derivative of the error function with respect to each of the weights is calculated. This is called backpropagation because, in order to do it, one starts at the output layer and calculates the partial derivatives of the weights linking this layer to its predecessor before moving backwards a layer and calculating those partial derivatives (and so on until reaching the input layer). Once all the partial derivatives have been calculated they represent a gradient vector with which to update the weights. These derivatives are then added to their respective weights after being multiplied by a learning rate. A higher learning rate leads to larger jumps in the weights, while a lower rate causes smaller changes.

After this is complete, the network can once again be used on the training data and the weights adjusted again. This process continues either until a set number of iterations have completed, or the network reaches a stopping point (its error stops changing). This behavior is what gives gradient descent its name. If the loss function is visualized as a surface plot, then then the network will slowly makes steps towards a minima of such a plot.

### 3.1.2 Logistic Regression

Logistic regression is a binary classification (output labels can be represented as 0 or 1) model that also met success in the project. At its core, logistic regression shares a lot in common with basic linear regression; both models rely on linear combinations of features. In linear regression, the predicted output of an instance is just a linear

combination of its features using some weights, but in the case of logistic regression, we model the log-odds that an instance is labeled 1 using a linear combination of its features. Suppose we have an $n$-dimensional feature vector $\boldsymbol{x}$ and a weight vector $\boldsymbol{w}$ with a weight for each feature plus a bias weight, $w_0$. In this case

$$log\text{-}odds(\boldsymbol{x}) = w_0 + \sum_{i=1}^{n} w_i x_i$$

This strategy may seem odd, but it ultimately allows a logistic curve (a sigmoid function like that in Figure 3.4) to be fit to the data. A logistic curve is locked within the range $[0, 1]$, and thus provides a convenient way to model probability. Moving to just odds from the log-odds only requires raising the base, $b$, of the logarithm to the linear combination of features:

$$odds(\boldsymbol{x}) = b^{w_0 + \sum_{i=1}^{n} w_i x_i}$$

In probability theory, the odds of something happening is simply the ratio of the number of times that thing happens to the number of times it does not. Therefore, converting the odds to a probability is as simple as

$$\frac{odds}{odds + 1}$$

.

Using this, a logistic regression model predicts the probability of an instance being class 1 as

$$\frac{b^{w_0 + \sum_{i=1}^{n} w_i x_i}}{b^{w_0 + \sum_{i=1}^{n} w_i x_i} + 1} = \frac{1}{1 + b^{-(w_0 + \sum_{i=1}^{n} w_i x_i)}}$$

This final form is a logistic curve. This curve is used to classify instances fed to the model. If the output of the model is $\geq .5$ then the instance belongs to class 1, otherwise it belongs to class 0.

As such, while the features have a linear effect on the log-odds of an instance, they do not have this linear effect on the odds (and as a result, probability). A constant increase in a feature will not lead to a constant increase or decrease in the probability. Despite this fact, logistic regression is still recognized as a generalized linear model since a linear combination is used to model log-odds - there is no multiplication of features to arrive at the class label probability.

The weights for a model are chosen such that they result in a maximum likelihood for the data at hand. Assuming the probability of being class 1 are independent among data instances, then the likelihood of a collection of data instances represented as feature-label pairs $(\boldsymbol{x}, y)$ for a model $m$ is calculated as

$$Likelihood(m) = \prod_i (m(\boldsymbol{x}_i))^{y_i} (1 - m(\boldsymbol{x}_i))^{1-y_i}$$

In other words, for each data instance, we use the model to predict the probability that the instance belongs to class 1. If it does belong to class 1, we use that probability, otherwise we subtract the prediction from 1 in order to get the probability that it is class 0. These probabilities will be high when the model correctly predicts data instances of each class. The product of all these probabilities represents the likelihood of model given the data. Choosing weights to maximize this results in a good model.

Gradient descent (as discussed in the MLP section) is one way to arrive at these weights. Here, the likelihood function serves as the cost function. However, this function often has local minima which gradient descent can get stuck in. Log likelihood is used to convert the cost function into a convex one - a function with a single, global minimum.

While logistic regression is used primarily for binary classification, it can be extended to multi-class problems. The simplest approach to this is to create a model

for each class label, where 1 corresponds to that class label and 0 corresponds to every other label. Once each model is trained, the prediction for a data instance is the label corresponding to the model which gave the highest probability.

### 3.1.3 Support Vector Machines

While they failed to achieve notable performance in the project, support vector machines (SVM) were used and so merit a short description. Like logistic regression, support vector machines deal with binary classification problems. They attempt to draw a hyperplane to divide the two classes. Once found, the line represents the maximum-margin between the two classes. That is, it is the line with the greatest distance to any point in first class as well as the second. SVMs can perform multi-class classification by drawing multiple hyperplanes.

## 3.2 Unsupervised Learning

In unsupervised learning, instead of making predictions, the task involves finding latent relationships or structures in data. Put simply, you might say supervision is learning with labels, while unsupervised learning is learning without labels. Unsupervised methods are capable of learning relationships within data that might be used to create new labels for the data or redefine the data itself.

### 3.2.1 Feature Engineering

While feature engineering is by no means a subset of unsupervised learning, unsupervised techniques are often applied during feature engineering, so this process is described here. As mentioned previously, the inputs to a machine learning model are called features. Supervised models make predictions by learning

the variations between the features of training data, and as a result, the actual representation of the features is extremely important.

Feature engineering encompasses the refinement and manipulation of this feature space before training a supervised model. If training a model is like baking a cake, then feature engineering is like preparing the ingredients. In the simplest case, feature engineering might just be dropping a feature that has no variance; if all the students in some training data watch one hour of TV a day, then this feature doesn't reveal any useful information about a student's grade. An example of more complicated feature engineering is combining features. One might simply average two features if they seem highly correlated or cross two features if they are not linearly separable.

#### 3.2.1.1 Dimensionality Reduction

One type of feature engineering that sometimes employs unsupervised learning is dimensionality reduction - shrinking the number of features in the feature space. Supervised models generally benefit from being trained on feature spaces with lower dimensionality due to the infamous "curse of dimensionality."

A term introduced by Bellman in 1961 [6], the "curse of dimensionality" refers to the explosion in volume that accompanies increasing the dimensions of a feature space. This impacts machine learning because it requires a greater number of observations to explain the variations in the features. Consider a scenario where a model is built to classify fruit as either apples or grapefruit. Imagine that there are ten total training examples, five apples and five grapefruits, and that the only features used are size and color; the classifier will probably perform quite well. Now consider, however, that instead of two features, there are 10,000 covering everything there is two know about these fruits - germination time, vitamin C content, weight,

water content, growing location, etc. With only five observations of each fruit, it will be extremely difficult to explain the variations in such a large number of features.

Another problem with high dimensionality is its impact on Euclidean distance [5]. As the number of dimensions expands, the distance between points becomes more uniform. Thus, in the example above, while there is most likely two distinct clusters of observations in two-dimensional space (one cluster for apples and another for grapefruits), this difference disappears in the 10,000-dimensional space as each observation seems just as far apart from all other observations.

#### 3.2.1.2 Word2Vec

Word2Vec is a group of models for learning word embeddings that has merit in the project and is discussed here to give a concrete example of unsupervised learning in the context of dimensionality reduction. A piece of text viewed as a collection of words is an inherently high dimensional space, and the simplest way to represent these words numerically is to use one hot-encoding. Under this scheme, if the text in question has a vocabulary of size $n$, then each word can be represented as an $n$-dimensional vector where each element is 0 except for the element at the same index as the word's location in the dictionary, which is set to 1. In this way, each word gets its own encoding, but the encodings are extremely sparse (the majority of values are 0).

Word2Vec learns new, more representative encodings for the words in a corpus based on their context. In this case, context is an indication of the words that appear near a given word in the corpus; words with similar contexts have the same words appear near them in the text. Numerically, words with similar contexts will have encodings that are close in distance. Figure 3.5 shows what this might look like after training a Word2Vec model to learn 2-dimensional word embeddings on a (wholly fictitious and nonexistent) story about the Dungeon of Corrosion.

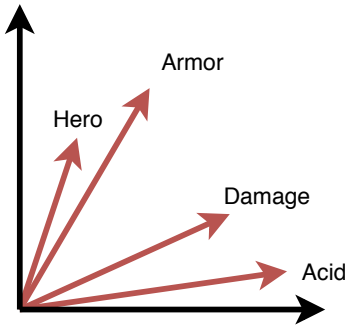



**Figure 3.5:** Example 2-dimensional word embeddings.

While several implementations of Word2Vec exist, this discussion assumes the one built on the skip-gram neural network architecture. This network takes one-hot encoded words as input and predicts a word it expects to see near that word as output. Once trained, the output of the network is no longer useful, and the Word2Vec model discards it, only retaining the first weight matrix of the network. These weights serve as the new embeddings for the words corpus; each row of the matrix corresponds to the embedding of a word in the vocabulary. In order to find a word's embedding, one simply takes the one hot encoding of the word and multiplies it by this weight matrix to isolate the word's embedding.

The network itself comprises three layers in a similar structure to the MLP described above. The input layer has as many neurons as there are words in a text's vocabulary, since it takes one hot word encodings. The middle layer (the only hidden layer) determines the dimensionality of the resulting word embeddings and is usually on the order of $10^2$ in size. Finally, the output layer, like the input is the same size as the text's vocabulary. The activation function of this final layer is the softmax function. If $\boldsymbol{o} \in \mathbb{R}^m$ is the vector of outputs of the network before the softmax is applied, then the softmax of any neuron $n \in \boldsymbol{o}$ is:

$$\frac{e^{\boldsymbol{o}_n}}{e^{\sum_{i=1}^{m} \boldsymbol{o}_i}}$$

After applying this function to each of the output neurons, the sum of the outputs will be 1. Because of this, and the fact that there is a neuron for each word in the vocabulary, the output of each node represents the probability of finding each word in the vocabulary near the input word.

That actual training examples for the network are word pairs (input, output) from the text. The number of these pairs depends on the context window, an important hyperparameter in the Word2Vec. The context window is a number that indicates how many words before and after a word are included in its context. Figure 3.6 shows how to craft the first few training examples on an example text with a window size of one.

Corrosive acid covered poor Thortoq, whose metal armor stood no chance.

Corrosive acid covered poor Thortoq, whose metal armor stood no chance.

Corrosive acid covered poor Thortoq, whose metal armor stood no chance.

| Input | Output |
|---|---|
| corrosive | acid |
| acid | corrosive |
| acid | covered |
| covered | acid |
| covered | poor |

**Figure 3.6:** Creating the first few Word2Vec training examples for a toy text.

Actually training the skip-gram network can be costly, since there is an output neuron for each word in the vocabulary of the corpus. Calculating the softmax over thousands, if not tens of thousands values is expensive; two methods exist to combat this. First, under-sampling is performed on words with high frequency in the corpus. These are often stop words, like “the", and once removed are not included

in the vocabulary or when creating training pairs. Secondly, the skip-gram network often uses negative sampling. When the weights of the network are updated, only a small sample of the neurons in the output layer, and the corresponding weights are updated. This sample always includes the neuron for the word that should've been predicted and a small number of so-called 'negative' neurons, those neurons that should not have fired. This greatly reduces the number calculations performed.

The Word2Vec model, once trained, produces a word embedding lookup table for the vocabulary of a corpus. These new embeddings are both much smaller than one-hot encodings for those words and much more representative of the words' meaning.

### 3.2.2 Clustering

Clustering techniques are discussed here because clustering is a well known family of unsupervised learning methods. They are a popular method for finding groups of similar data points in an unlabelled set. They can also be useful to see if class labels are actually representative of the data groups found in a dataset. Many clustering algorithms exist, but the underlying process involves finding data points that are close together in the feature space and assigning them to a cluster, where "close" is defined by some distance function or represented by some sort of generated distribution. Figure 3.7 shows how a clustering algorithm (fit with 3 clusters) might label a group of wines based on their phenol and flavanoid levels versus the actual labels of the wines. The labels are irrelevant, but the figure shows how clustering can find similar data points in a dataset even if the true labels are not known (the true labels are just for reference).

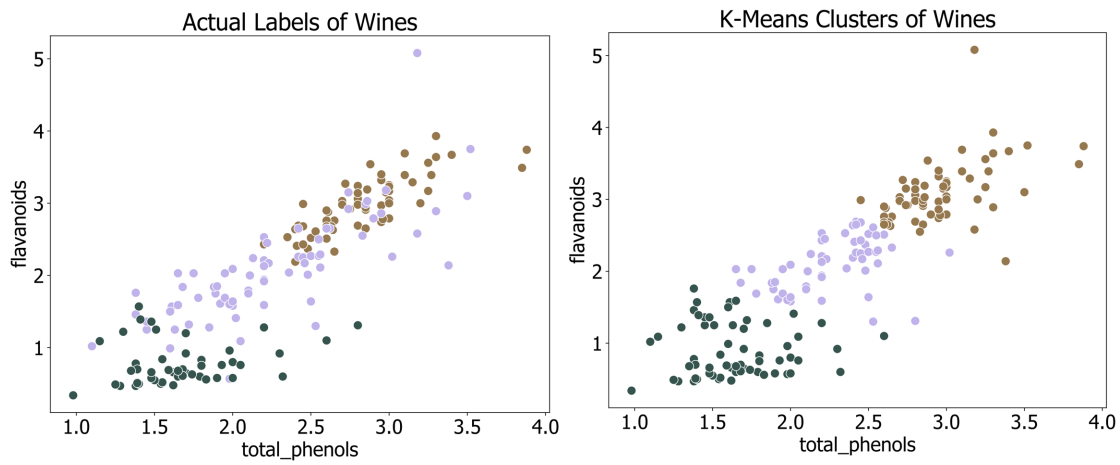



**Figure 3.7:** Clustered and actual wine class labels among wines with the varying phenol and flavanoid levels.

*CHAPTER 4*

# D&D&Data

This chapter first explores the structure of the data used, its acquisition, and preprocessing. It then provides a brief analysis of the processed data in the context of the problem at hand.

In this project, the data is composed of individual character instances, each with two general parts: a representation of the backstory and character information relevant to the mechanics of the game (the prediction targets). In the case of D&D, the first of these is simply raw text, while the latter can be any of the scores or categories found on the D&D character sheet.

Figure A.1 shows the official character sheet for 5th edition D&D. Many of the entries on this sheet are derivative of others or have can have values that depend on information not found in the character's backstory (e.g. Alice's character has a bond with Bob's even though Bob's character is not in Alice's backstory). Because of this, the following pieces of the character sheet are assumed to be valid targets for prediction:

- Attribute Scores: These are a series of numeric scores that describe the basic mental and physical aptitude of a character. Higher scores correspond to higher aptitude in that attribute.

- Class:[1] The class of a character is a categorical variable that describes their adventuring 'profession' of sorts and can take a number of discrete values.

- Race:[2] What would a fantasy universe be without Tolkien-esque races? Race is another categorical variable that can take a number of discrete values.

- Skill Proficiencies: The character sheets of D&D 5th edition feature a discrete list of skills which players can bubble in indicating 'proficiency.' A character performs more efficiently at a task using these skills.

## 4.1 Acquiring Data

With no readily available datasets of D&D characters online, acquiring character data requires different routes than the usual download of a compressed file or API query (for example - gathering tweets using Twitter' API). Thus, we use two methods to collect data: Google Forms and web scraping.

The Google Form originally only contained fields for character class, race, name, and backstory. As the project progressed to its current form, however, inputs for attribute scores and skill proficiencies have been added. The link to the form was distributed among several game stores, gaming club officers at a number of universities, and a Reddit post. As of the time of this writing, the form has collected 196 responses.

Unfortunately, 196 is a paltry number of examples to train any sort of model with, especially when considering the number of class targets in each of the discrete categories requiring classification. For example, when considering character class, there are twelve targets. Even if the responses are completely balanced against

[1] While there are many classes and races in the D&D universe, this project focuses on those found in 5th edition Player's Handbook. They are the most popular.

[2] See footnote 1

character class, then each class would only have 16 training examples. A comparable dataset in the domain of NLP is the subset of the Yahoo! Answers corpus that the authors of [24] use which includes questions from 10 different categories, each with 140,000 training and 6,000 testing examples. Even with this amount, their models could still only achieve 71.10% classification accuracy. While it is impossible to define the amount of data needed to adequately solve a certain problem, it's safe to say that 196 examples is not nearly enough.

In order to further gather character information, we scrape Shadowhaven's Dungeons & Dragons 3E Character Database, a site "dedicated to the use and storage of Dungeons & Dragons character files" [10]. Each character in the database has its own page, which is downloaded using the requests module for Python. Once downloaded, the raw HTML is parsed with the help of the beautiful4soup module; the fact that location of HTML elements containing relevant character information within each page is constant makes this possible. In total, the database contains pages for 16,515 characters - 6,684 of which have backstories or textual descriptions of any kind.

The biggest issue with this data is that all of the character information comes from D&D 3rd edition, while the goal of this project (and the information submitted through the Google Form) is in the domain of 5th edition characters. Attribute scores are identical between the two editions and there is a large overlap between races and classes. Unfortunately, the skill proficiencies in each are different enough that one cannot predict 5th edition proficiencies by training on 3rd edition ones.

As a result, we drop skill proficiency from the list of prediction targets. Also, the classes and races are reduced to the intersection of those found in the database and all 5th edition possibilities. There might be unforeseen differences between the two editions that could confuse models training on a large number of 3rd edition characters with 5th edition ones sprinkled in. Therefore, the 5th edition characters

collected through the Google Form are used exclusively in testing sets for the trained models. They indicate the sort of characters expected to be thrown at the model in production and are thus a good indication of model efficacy.

With these modifications realized, the data consists of characters defined as a combination of backstory and the following prediction targets:

- Character Class (classification problem with 11 classes)

  a) Barbarian b) Bard c) Cleric d) Druid e) Fighter

  f) Monk g) Paladin h) Ranger i) Rogue j) Sorcerer

  k) Wizard

- Character Race (classification problem with 7 classes)

  a) Dwarf b) Elf c) Gnome d) Half-elf

  e) Half-orc f) Halfling g) Human

- Character Attribute Scores (regression problem with 6 components)

  a) Strength b) Dexterity c) Constitution

  d) Intelligence e) Wisdom f) Charisma

## 4.2 Preprocessing

Before extracting features or fitting any models to the data, it is necessary to remove bad examples and attempt to remove noise from those which can be kept. This helps lower the confusion of trained models.

The first and simplest step is to drop duplicate characters, since duplicated characters will either give that character a slightly heavier weight when training a model or the character will show up in both the training and testing sets, causing a

leak of information. Following this, we drop examples with less than 100 characters in their backstory. This compensates for when the backstory is just a few words or is just a declaration of nonsensical character information, such as "Additional NotesSpells per Day:01254+13+1." The threshold of 100 characters is a just a heuristic chosen since it seems to work.

Following this, we drop punctuation and any stop words (words such as "the" or "a" which have little meaning) from the backstory. This shrinks the vocabulary of the entire corpus, since "Hammer" and "hammer" are now the same word. Also, since we use words as tokens, stripping punctuation and sticking to lower case helps reduce the dimensionality of any resulting feature space.

To reduce the vocabulary of the backstories further (and thus the dimensionality of any resulting features relying on it) we apply lemmatization to the backstories. A lemmatizer operates similarly to a stemmer, which applies a series of production rules to a word in order to reduce it to its word stem. A lemmatizer, however, incorporates the part of speech of a word and its morphological analysis in order to provide its dictionary form [17]. Consider the following examples:

He is learning to read.

A stemmer and lemmatizer will both reduce learning to "learn."

I went to the store.

A stemmer will reduce went to "went", while a lemmatizer will reduce went to "go."

We use the Natural Language Toolkit's (NLTK) WordNet lemmatizer to perform the lemmatization. This implementation conducts the necessary morphological analysis using WordNet, a lexical database for English where nouns, verbs, adjectives and adverbs are placed into into groups called synsets based on semantics and lexical relations [18] . Application of the lemmatizer reduces the size of the backstories' vocabulary from 66,713 to 58,680.

Apart from the backstories, the actual targets for classification and regression also require preprocessing. Many of the user entries for class, race, and even attribute scores are misspellings, abbreviations, or outside the range of accepted values. Originally, just a token replacer was written and applied to the entries. It matches entries to acceptable values close in spelling using a sequence matcher or replaces them using a dictionary of common abbreviations. Unfortunately, it is difficult to cover each edge case using this approach. Therefore, the character class and race values were hand annotated before running the token replacer on them. Attribute scores are simply coerced into integer values.

## 4.3 Data Analysis

This section provides a cursory analysis of the preprocessed data. We explore the patterns and distributions within the data, which is useful for understanding how it fits into the problem. A better understanding of the data at hand will lead to a better understanding of model performance in the future.

Figure 4.1 shows the distribution of word counts among the backstories. One must remember that these are the processed backstories, so stop words have been removed. As a result, the backstories will be slightly shrunken. The maximum backstory length is 5,983 words, while the mean and median are 261.2 and 173, respectively. Also to note, the x-axis of Figure 4.1 cuts off at the 99th percentile of the word counts.

The figure indicates that the distribution is heavily skewed right; most of the backstories are relatively short. This makes prediction harder, since a smaller amount of text is more likely to hold less information than a larger one. In addition, shorter backstories will have more unstable predictions, since each word will have a large importance; changing one could result in a vastly different decision.

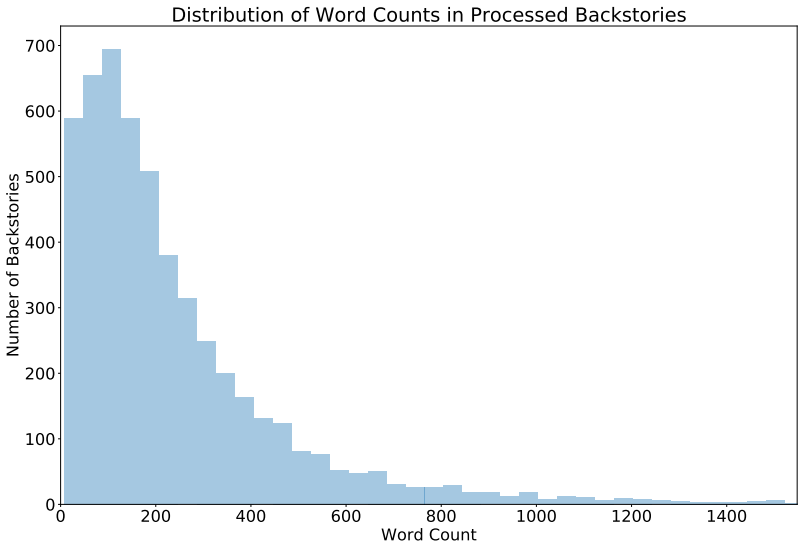



**Figure 4.1:** Distribution of word counts (up to the 99th percentile) among processed backgrounds.

Figure 4.2 shows the distributions of both the character class and race for the subsets of characters where the two are recognized (notice the different totals for each - just because a character's class is known does not mean that the race is, and vice versa). Here, we see that the dataset is not balanced on either target. In the case of race, this is extremely true, as the characters are overwhelmingly made up of humans. Imbalanced classes pose an issue in machine learning, as trained models will be naturally biased to the majority class.

A simple example portraying this is a Nobel Prize Laureate classifier that takes academic papers as input and decides whether or not the writer will win a Nobel Prize. Such a classifier will have > 99% classification accuracy if it simply says no to every paper (only 10 individuals won prizes in 2017 [4]). This behavior, however, is useless. Attempts to remedy this class imbalance are discussed in the results chapter.

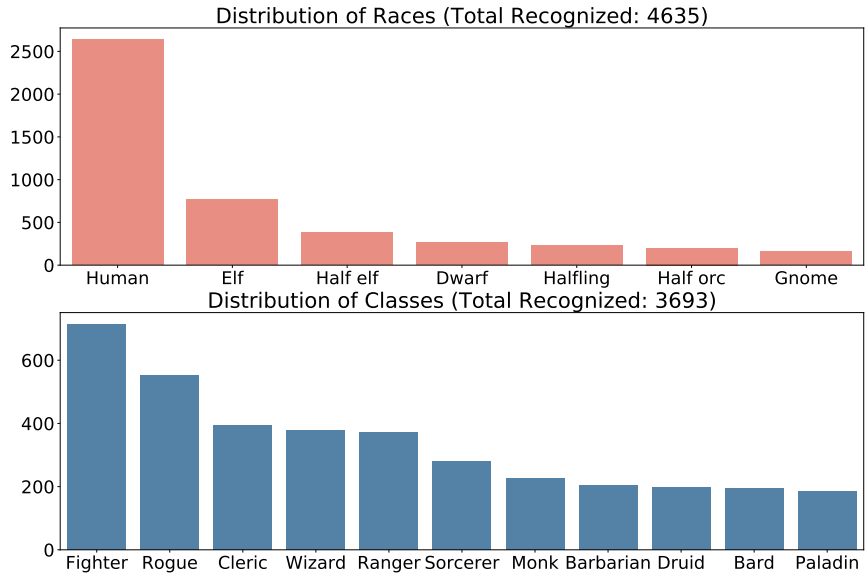



**Figure 4.2:** Distribution of character race and class among processed backgrounds.

Figure 4.3 expands upon the information in Figure 4.2 by showing the class composition of each race. That is, each pie chart shows the makeup of a single race in terms of the character classes present among characters of that race.

These distributions indicate some correlation between character class and race. We observe that some races seem to have a high bias towards certain classes (Dwarves and druids, Half Orcs and barbarians, Halflings and rogues).

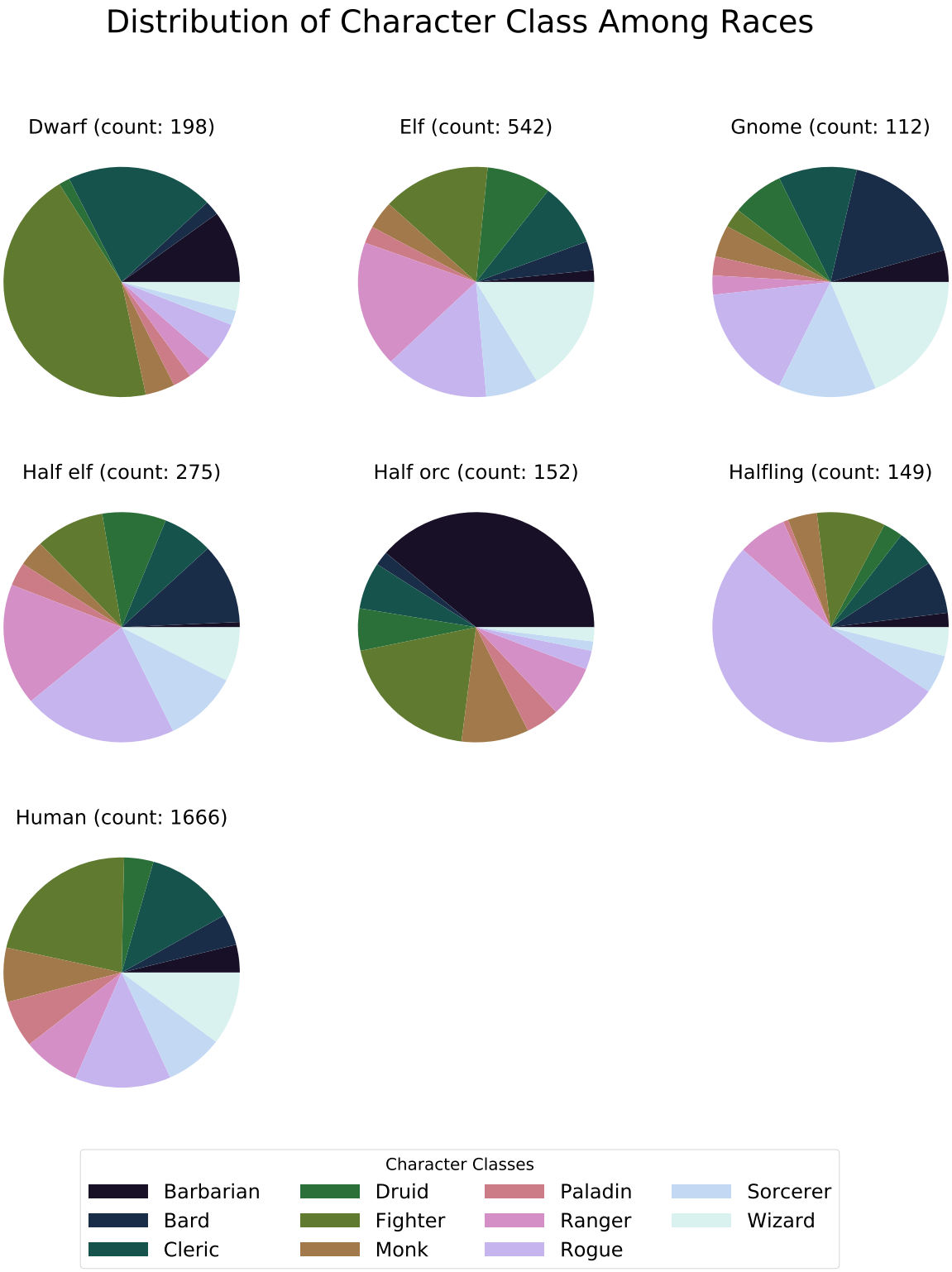



**Figure 4.3:** Distribution of character class within races among characters with recognized character class and race.

Figure 4.4 shows box plots for each of the six attribute scores. Each plot indicates the quartiles of an attribute. This whiskers are set to be $1.5 * IQR$ and, thus do not indicate the maximum and minimums. Indeed, the y axis is clamped from [0,40], so there are some outliers among the attributes approaching values of 1000 not shown.

Overall, it seems that for each attribute, over 50% of the characters' scores reside in the 10-20 range. This is expected, as anything below 10 gives a character negative modifiers for that attribute during gameplay. In addition, while most of the scores have similar spreads, dexterity has a higher mean and lower quartile than all the rest indicating at a least a small general preference for dexterity over the other attributes.

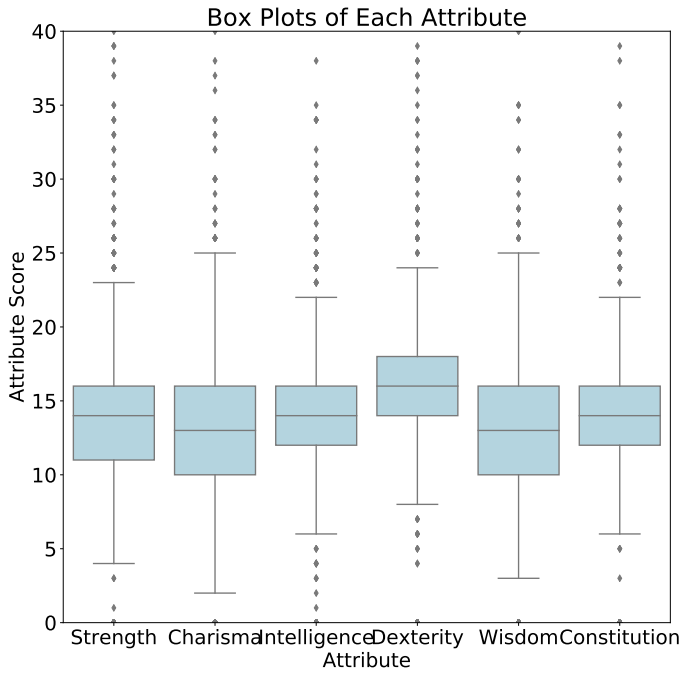



**Figure 4.4:** Distribution of attribute scores among characters.

*CHAPTER* 5

# Predicting Characters: Approach and Results

As stated in the previous chapter, there are three groups of prediction targets in the project: character class, race, and attribute scores. Rather than attempt to build a single model to learn all of these relationships, however, three separate models are constructed, one for each area. This chapter first discusses the approach used to predict class and race, two similar classification problems, before moving on to the attribute scores. For each area the feature engineering steps are outlined and the performance of any trained models are evaluated. The tasks are accomplished using the Scikit-Learn, Tensorflow, and Tensorlayer libraries for Python.

## 5.1 Class and Race

The underlying assumption made when predicting a character's class and race is that the composition of words in a character's backstory is indicative of where the character falls in these two categories. Because there are so few training examples, more complex models incorporating sentence structure into the feature set are not pursued. In other words, when considering "Thortoq hates humans" and "Thortoq loves humans," the meaning of the word "human" does not differ.

For both class and race, the available characters - those with recognized classes or races - are split into training and testing sets. Following the guidelines from

Andrew Ng, the split is 80/20, respectively [19]. In addition, the split is stratified so that the training and testing sets represent the same class distribution found in the data. The 5th edition characters from the Google Form are included in the test sets for both class and race. Table 5.1 shows the actual counts for these sets.

| Problem | Training | Preliminary Testing | 5th Edition Data | Total Testing |
|---|---|---|---|---|
| Class | 2954 | 739 | 135 | 874 |
| Race | 3708 | 927 | 113 | 1040 |

**Table 5.1:** The number of characters in training and testing sets for class and race classification.

With this in mind, two feature spaces are engineered for classifying class and race. The first contains TF-IDF values. The raw backstories of the training data are converted into a matrix of TF-IDF values where each row represents a character and each column corresponds to a word in the training data's vocabulary. If every word in the vocabulary is kept, then the dimension of the feature space is quite large. The upper bound is the size vocabulary of the entire corpus (i.e. 58,680), but usually ends somewhat smaller since it is only fit to the training data. In order to reduce this size, a minimum document frequency of .005 is used - words occurring with less frequency than this are dropped.

A second feature space is created for each problem using the learned contexts of the words in their respective training data. These contexts are learned using a Word2Vec model based on the skip-gram model architecture with a hidden layer containing 300 neurons. This number was chosen because it is the same dimension that Google uses in their Google News Vector Dataset [3]. The resulting contexts, represented as a table of embeddings, is multiplied by the TF-IDF matrix for the characters to create a new, 300-dimensional feature space that describes the average

context (weighted for the importance of each word as per TF-IDF) of a character's backstory. We call this the TF-IDF averaged context. Figure 5.1 shows this process.

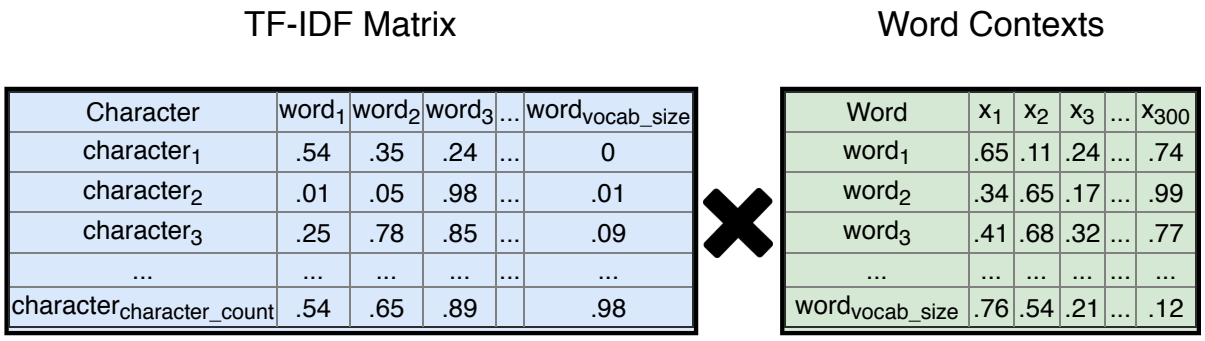



**Figure 5.1:** Multiplication of TF-IDF matrix and word embedding table to create a new feature space.

### 5.1.1 Baseline Models and Metrics

The most fundamental scoring metric for classification models is classification accuracy - the percentage of successfully classified instances in a group of examples. However, classification accuracy can often be misleading or a poor metric to evaluate performance. Consider the example of noble prize winners given in Chapter 4.

Precision and recall are two metrics that often reveal more about a model's performance. Each class label in a classification problem has an associated precision and recall value. Both are measured using the following counts using the predicted labels of some known data (assume $x$ to be the class label in question):

- True Positives (TP) - we predict $x$ and the actual class is $x$
- False Positives (FP) - we predict $x$ and the actual class is $\neg x$
- True Negatives (TN) - we predict $\neg x$ and the actual class is $\neg x$
- False Negatives (FN) -we predict $\neg x$ and the actual class is $x$

Precision represents how many instances predicted as the label in question are relevant and is calculated as $\frac{TP}{FP+TP}$. Recall represents how many of the relevant items for a class label are accurately predicted and is calculated as $\frac{TP}{FN+TP}$. A third metric, the $F_1$ Score combines precision and recall into one value by taking their harmonic mean, which is calculated as $\frac{(precision)\times(recall)}{precision+recall}$.

While a classification accuracy of 100% (precision and recall for each class are 1) indicates a perfect model, what constitutes 'good' performance varies depending on the problem domain. Because of this, it is useful to compare the performance of trained models to some baseline model when discussing their efficacy. Baseline models are naive models that (usually) follow simple rules in order to make a prediction.

In the case of a classification problem, perhaps the simplest baseline is just to always predict the dominant class found in the training data. However, knowing the domain of the project, there is a better baseline model. For each backstory, we count the frequency of the classification labels in the text. We then classify the character as the label with the highest frequency - facing a lack of labels in the text, we simply classify the character as the majority class. This captures characters with backstories that start with something like: "Thorqtoq is a half orc barbarian."

Figure 5.2 shows the confusion matrices for the baseline models on the testing data for both class and race. A confusion matrix is a heat map which visualizes the predictions of a model on some data versus the actual labels of the data. The rows of the matrix correspond to the actual labels in the data, while the columns are the predicted labels. The sum of each row is 1, and thus the value in a cell represents the percentage of the row's label classified as that column's label. Heat along the diagonal is good, since this represents correctly labeled instances. Heat in other areas can reveal where the model is performing poorly. Table 5.2 contains the discussed performance metrics of each baseline.

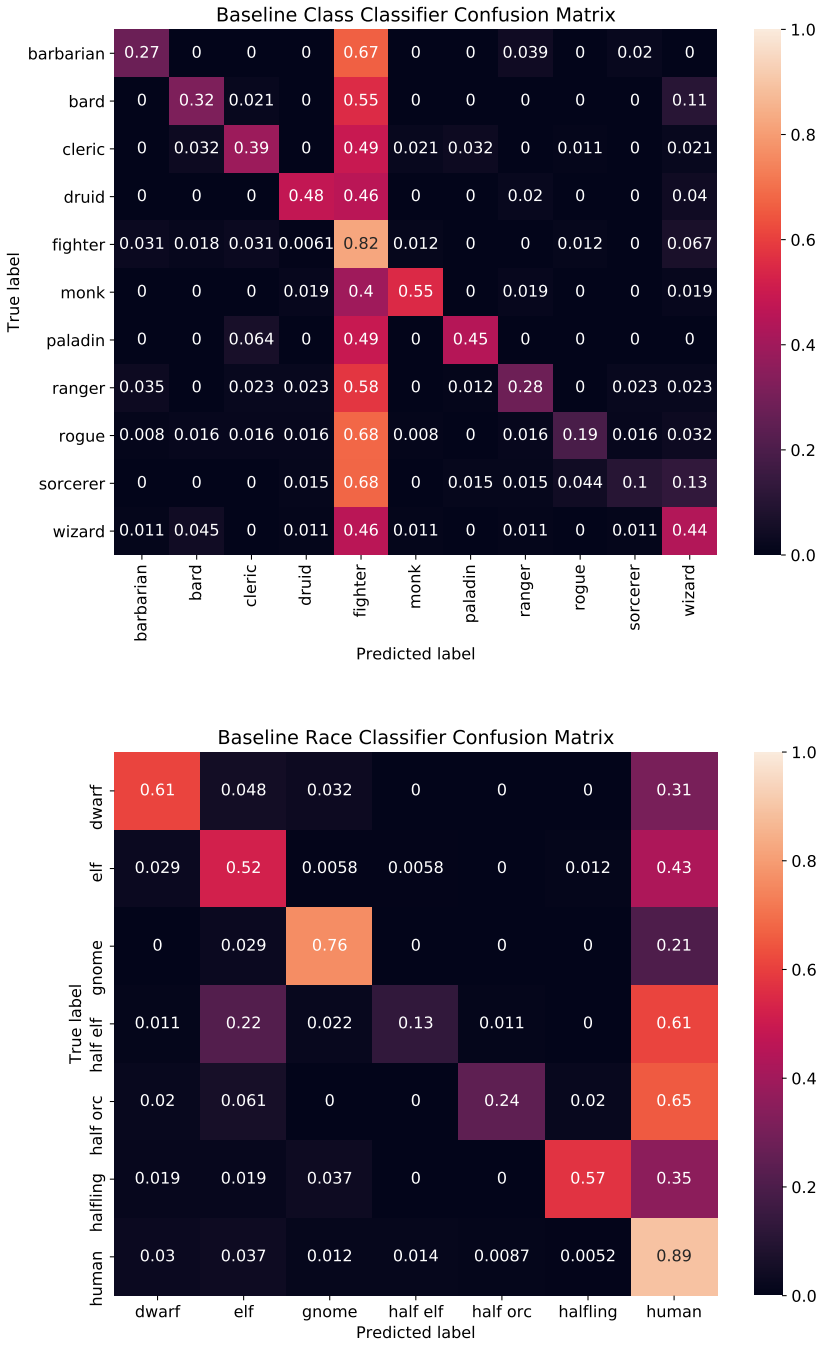



**Figure 5.2:** Baseline classifier confusion matrices.

| Baseline | Accuracy | Mean Precision | Mean Recall | Mean $F_1$ Score |
|---|---|---|---|---|
| Class | 42.10% | 0.65 | 0.39 | 0.44 |
| Race | 69.51% | 0.67 | 0.53 | 0.56 |

**Table 5.2:** Performance metrics of baselines for character class and race.

Overall, the baselines perform as expected; the significant heat in the *human* and *fighter* columns reveal the that the models are heavily biased toward the most common label. The raw accuracy of the race baseline outperforms the class. This is probably because there are more labels among class and the race counts were more imbalanced than the class counts.

### 5.1.2 Dealing With Imbalance

The problem of imbalanced class labels persists when training models. Figure 5.3 shows the results of a default logistic regression model trained on the TF-IDF matrix of backstories to classify character class. The intense heat in the *fighter* column reveals that models will learn the imbalance in the training data.

In order to combat the problem of imbalance, a combination of oversampling and undersampling is used. In the case of character class, where the imbalance is not as drastic, the training data is oversampled so that there are as many instances of the minority class labels as there are of the majority (fighter). The sampling method used is random (with replacement). Thus, after sampling, the training data will have quite a few duplicate instances. Character race is much more imbalanced, so the majority class label is undersampled before the minority class labels are oversampled to match it. The undersampling method is also random. Of the majority class, 75% are dropped before oversampling occurs.

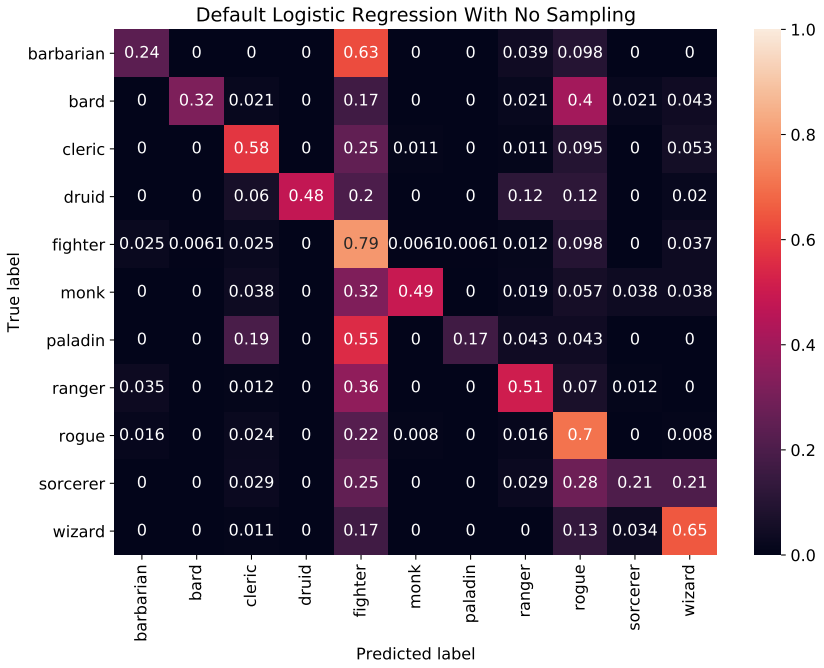



**Figure 5.3:** Confusion matrix of a logistic regression model trained to classify character class without sampling.

### 5.1.3 Model Selection

Three models are compared when predicting class and race: logistic regression, random forests, and support vector machines. These models were selected because of their popularity for classification and in order to keep the scope of the project within reason. The models are trained on each of the two feature spaces: the TF-IDF values for the backstories and the TF-IDF averaged context.

In order to select the best set of hyper-parameters for each model, exhaustive grid search is used. This technique takes a model type and a grid of possible parameters; a model is trained for each of the possible parameter combinations, and the model with the best performance is returned. Selecting the best combination by evaluating on the test set could lead to overfitting, so 5-fold cross validation is used instead.

Using 5-fold cross validation, the grid search splits the training data into 5 bins before trying a parameter combination. It then trains 5 version of the model, each on a different set of four bins. The last bin for each model is used as the testing set to find the performance of the model, after which, the performance of the 5 models is averaged.

Figure 5.4 shows the character class confusion matrices for each combination of model and feature space. The title of each matrix indicates the model type, the classification problem (class for all), and the feature space the model was trained on. The confusion matrices are similar in format to those already seen so the labels are elided.

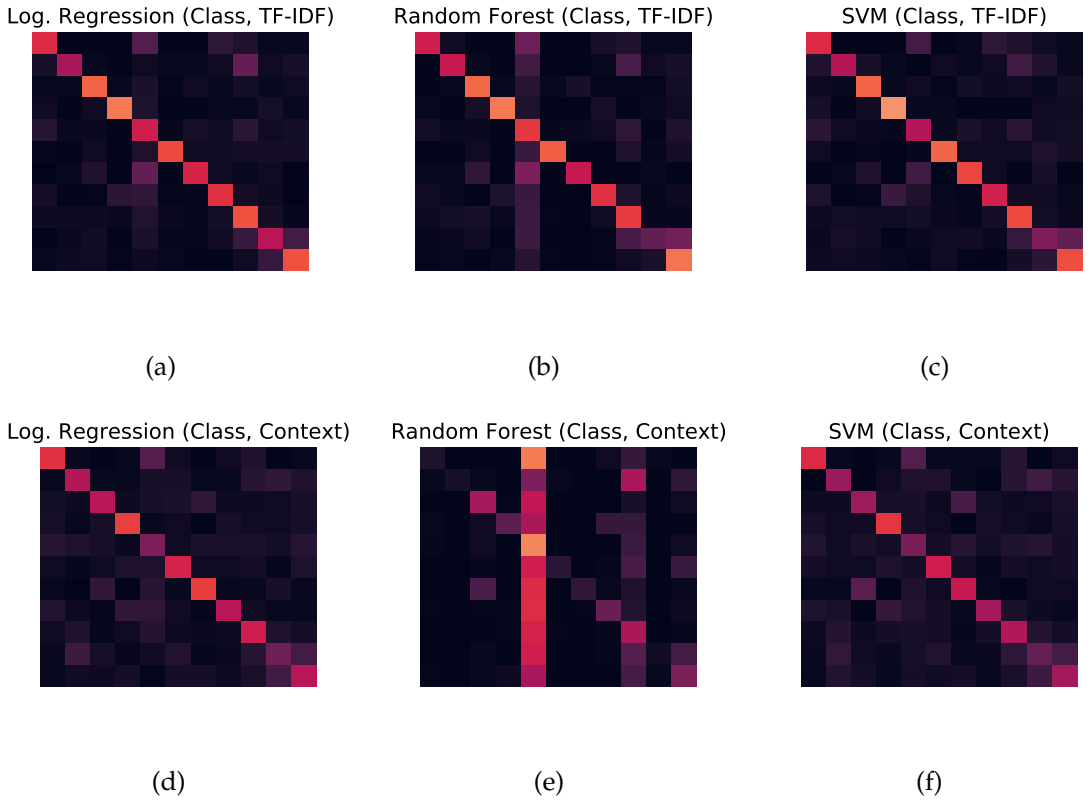



**Figure 5.4:** Confusion matrices for character class classifiers. The title of each matrix indicates the model trained what feature space was used.

Without even considering the actual metrics associated with each combination

it is clear that the models trained on the TF-IDF matrix outperform those trained on the TF-IDF averaged context feature space. The heat for the former models are much more focused along the diagonal than those in the latter group.

Interestingly, the models in Figure 5.4(a) and Figure 5.4(b) compensate each other's failing to a minor degree. For example, the 5.4(b) does better at classifying fighters, while 5.4(a) excels at classifying sorcerers. This can be observed by comparing the heats of each in both the fifth and tenth cells along the diagonal. This is relevant because logistic regression and random forest models are both capable of producing the predicted probabilities of all class labels for an instance. Thus, it is possible to combine their predictions (sadly, this is not the case for support vector machines).

Based on the indication that models 5.4(a) and 5.4(b) might complement each other, they are combined in a simple ensemble method - their predicted probabilities are averaged. The label corresponding to the maximum probability of this average is used to label the instance. Figure 5.5 shows the heat map for this ensemble, the combination of logistic regression and random forests, for classifying class.

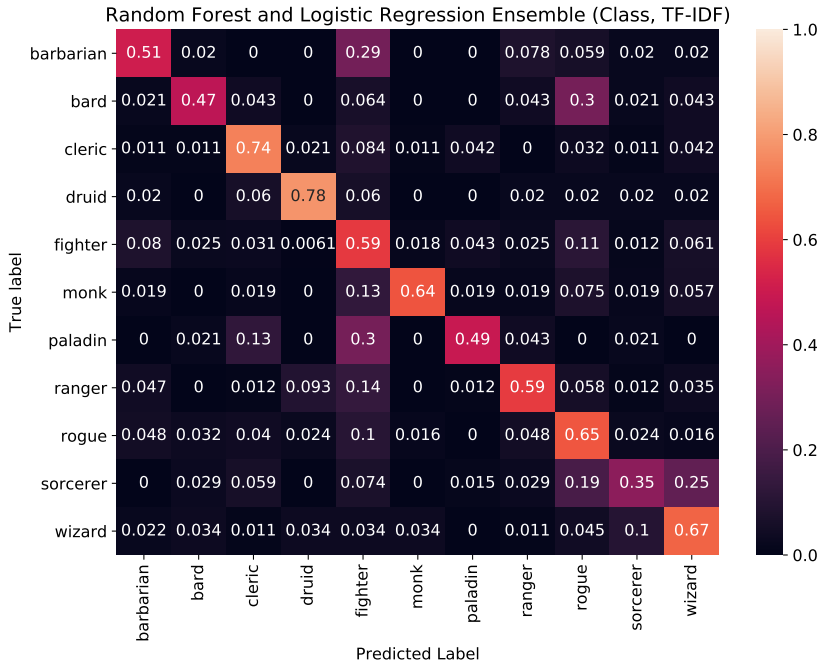



**Figure 5.5:** Confusion matrix of random forest and logistic regression ensemble to classify character class.

Using visual cues alone, it is impossible to determine which model is best, so the performance metrics of each model are presented in Table 5.3. Other than accuracy, the metrics represent the average metric across all class labels. These values reinforce the previous statements made. Both the classification accuracies as well as the $F_1$ Scores are higher for each of the models trained on the TF-IDF matrix feature space than those not. Further, while each of the models these models is extremely close in terms of accuracy and $F_1$ Score, the ensemble of logistic regression and random forest slightly outperforms the others . This model improved upon the accuracy of the baseline model by 18.10% and had a higher mean $F_1$ Score by .16.

While this might not seem like a miraculous improvement, there is some additional merit in *how* the ensemble model misclassifies instances compared to the baseline. In the case of the baseline, instances are almost entirely misclassified as the

majority class. With the ensemble model, however, classes are often misclassified as similar classes. For example, paladins are most frequently misclassified as fighters or clerics, and in the world of D&D, paladins are something of a fusion of these two classes. Sorcerers are misclassified as rogues and wizards, two labels with similar characteristics to the sorcerer class when it comes to gameplay. Thus, instances misclassified by the ensemble model are still somewhat useful.

| **Model** | **Features** | **Accuracy** | **Mean Precision** | **Mean Recall** | **Mean $F_1$ Score** |
|---|---|---|---|---|---|
| Class Baseline | - | 42.10% | 0.65 | 0.39 | 0.44 |
| Log. Regression | TF-IDF | 58.01% | 0.59 | 0.58 | 0.58 |
| Random Forest | TF-IDF | 57.89% | 0.61 | 0.57 | 0.58 |
| SVM | TF-IDF | 56.18% | 0.55 | 0.57 | 0.55 |
| Log. Regression | Context | 44.85% | 0.44 | 0.47 | 0.45 |
| Random Forest | Context | 34.21% | 0.58 | 0.26 | 0.28 |
| SVM | Context | 40.73% | 0.40 | 0.43 | 0.41 |
| Ensemble | TF-IDF | 60.20% | 0.62 | 0.59 | 0.60 |

**Table 5.3:** Performance metrics of classifiers for character class.

Figure 5.6 shows the character race confusion matrices for each combination of model and feature space. As before, the title of each matrix indicates the model type, the classification problem (race for all), and the feature space the model was trained on. Again, labels are elided.

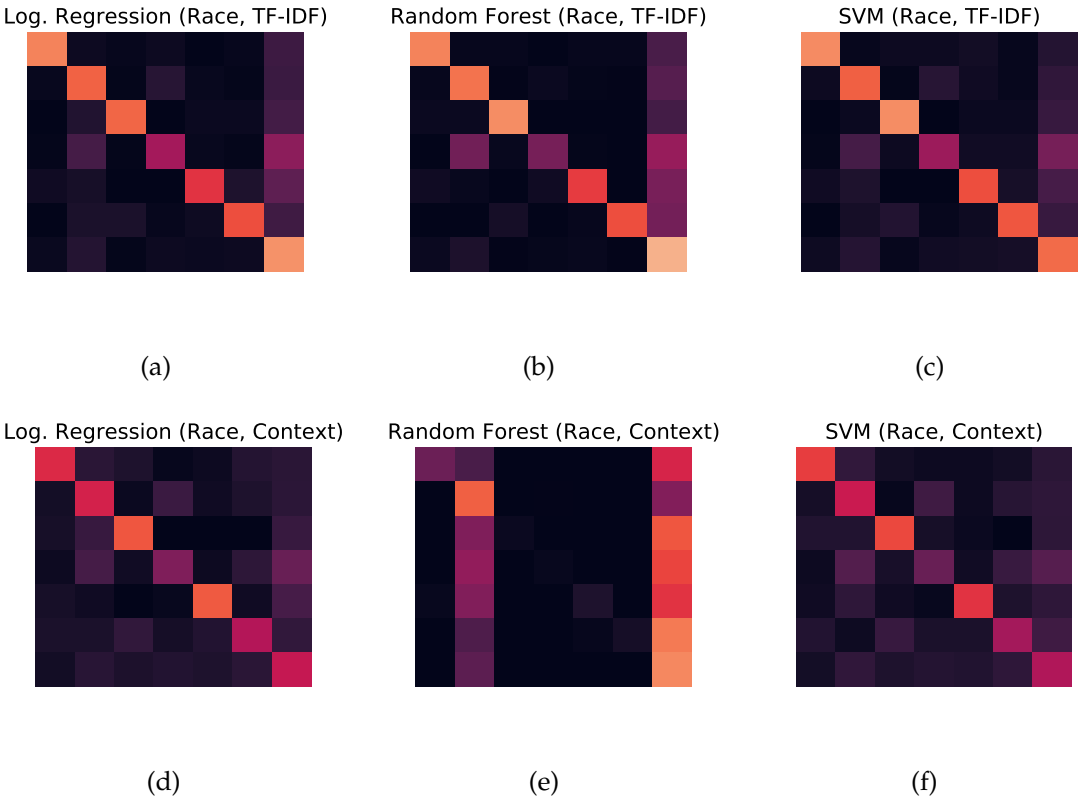



**Figure 5.6:** Confusion matrices for character race classifiers. The title of each matrix indicates the model trained what feature space was used.

The results here appear very similar to those for the character class classifiers. The TF-IDF matrix feature space outperforms that of TF-IDF averaged context. The same complementary behavior between logistic regression and random forests can be seen when observing the heat differences between Figures 5.6(a) and 5.6(b). Figure 5.7 shows the confusion matrix of the ensemble of these two classifiers. The performance metrics in Table 5.4 reveal the ensemble model to be the strongest performer. Here, however, there is an even smaller gap between the accuracy of the race baseline and this model (only 6.64%). Still, the same argument as to misclassification can be applied (many a dungeon master struggles to describe the difference between elves and half elves).

For both class and race, the best performing models are the ensemble of logistic regression and random forest classifiers trained on TF-IDF matrices for the backstories.

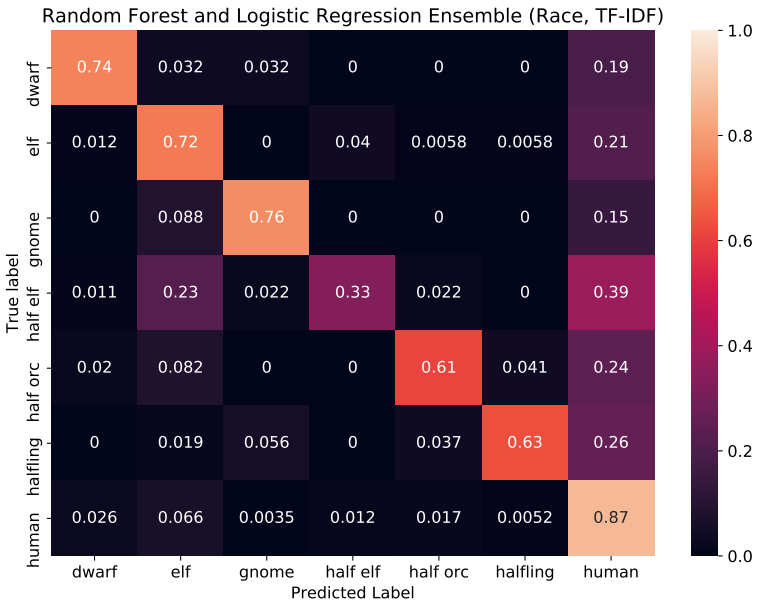



**Figure 5.7:** Confusion matrix of random forest and logistic regression ensemble to classify character race.

| Model | Features | Accuracy | Mean Precision | Mean Recall | Mean $F_1$ Score |
|---|---|---|---|---|---|
| Race Baseline | - | 69.51% | 0.67 | 0.53 | 0.56 |
| Log. Regression | TF-IDF | 70.19% | 0.61 | 0.64 | 0.62 |
| Random Forest | TF-IDF | 74.32% | 0.71 | 0.65 | 0.67 |
| SVM | TF-IDF | 65.96% | 0.55 | 0.65 | 0.58 |
| Log. Regression | Context | 49.33% | 0.39 | 0.52 | 0.42 |
| Random Forest | Context | 55.19% | 0.70 | 0.27 | 0.27 |
| SVM | Context | 45.19% | 0.36 | 0.49 | 0.39 |
| Ensemble | TF-IDF | 76.15% | 0.73 | 0.67 | 0.69 |

**Table 5.4:** Performance metrics of classifiers for character race.

## 5.2 Attribute Scores

Predicting attribute scores requires generating six different real values, one for each possible attribute: strength, dexterity, wisdom, charisma, intelligence, and constitution. The first task in this problem is choosing how to represent these scores. In the actual data, they show up as they would on a character sheet; each score has positive integer assigned to it. However, early attempts at training regressors to predict these scores were met with little success. This is probably because of the sheer range of scores found. Characters with higher levels have higher attribute scores than others, and level isn't something easily captured from a backstory.

In order to account for this, the attribute scores of each character are standardized. Each attribute score for a character is represented as the percentage of total attribute points that character has allocated for that attribute; these standardized scores add up to 1 and are in the range $[0, 1]$. In this way, a character with raw attributes $\{10, 12, 16, 12, 10, 10\}$ has the same standardized scores

({.142, .171, .229, .171, .142, .142}) as a character with raw attributes {15, 17, 21, 17, 15, 15}. These percentage representations can be transformed back into actual attribute score values by multiplying them by some point pool (e.g. 72 for a level 1 character).

With these targets, the feature spaces used to classify race and class, TF-IDF values and TF-IDF averaged contexts, were found to underperform when used to train regressors. Very little correlation was observed between the predictions of these models and any testing values. Following this, the assumption was made that class and race themselves might be good indicators of attribute scores. Indeed, in the Player's Handbook, there are suggestions for each class and race on how to tweak attribute scores. The data itself also seems to back this assumption. Figures 5.8 and 5.9 show the distribution of attribute score percentages for each class and race. For many of the attributes, there seems to be notable deviation between categories. Half orcs and dwarves appear to lack in charisma, for example, while barbarians and fighters excel in strength. Preliminary attempts to predict attribute scores using the actual class and race of a character proved much more successful than those using backstory features.

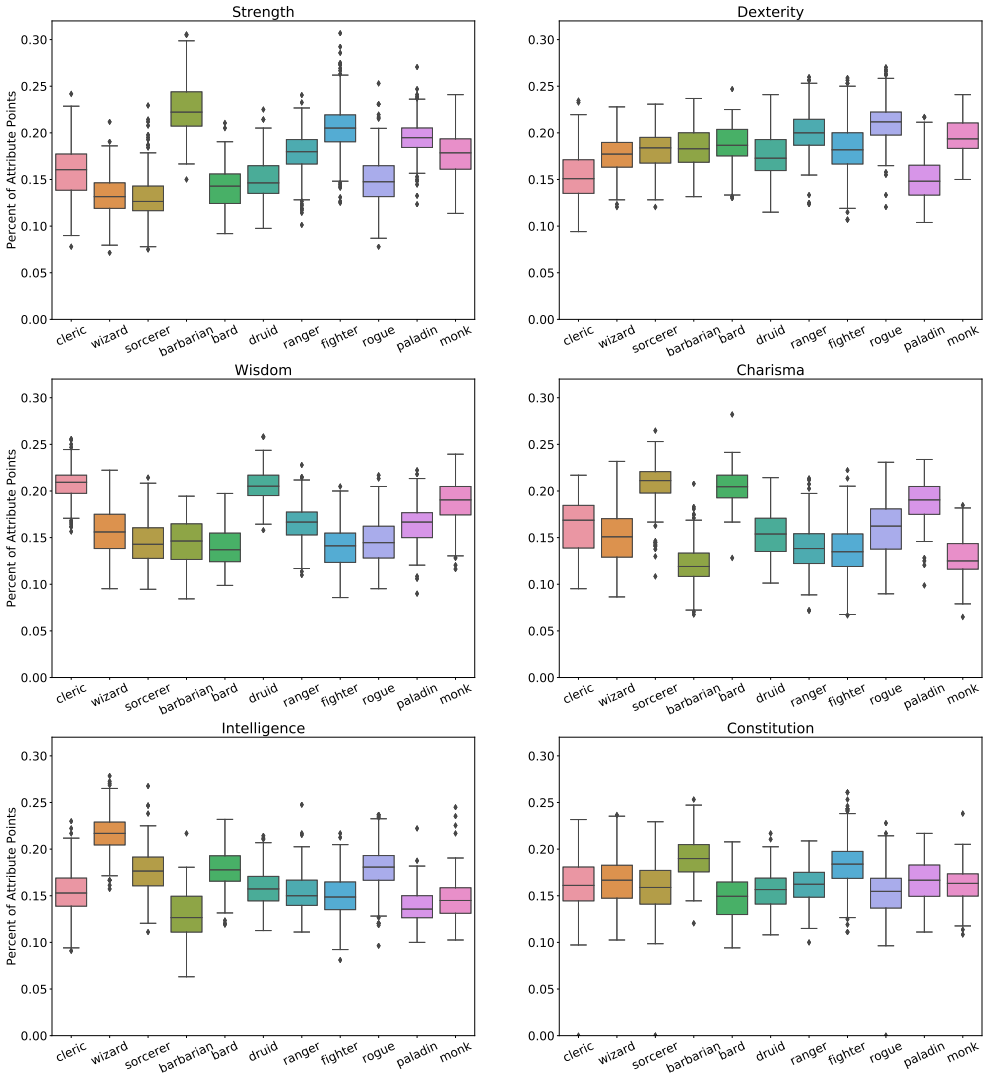



**Figure 5.8:** Distributions of attribute scores among class.

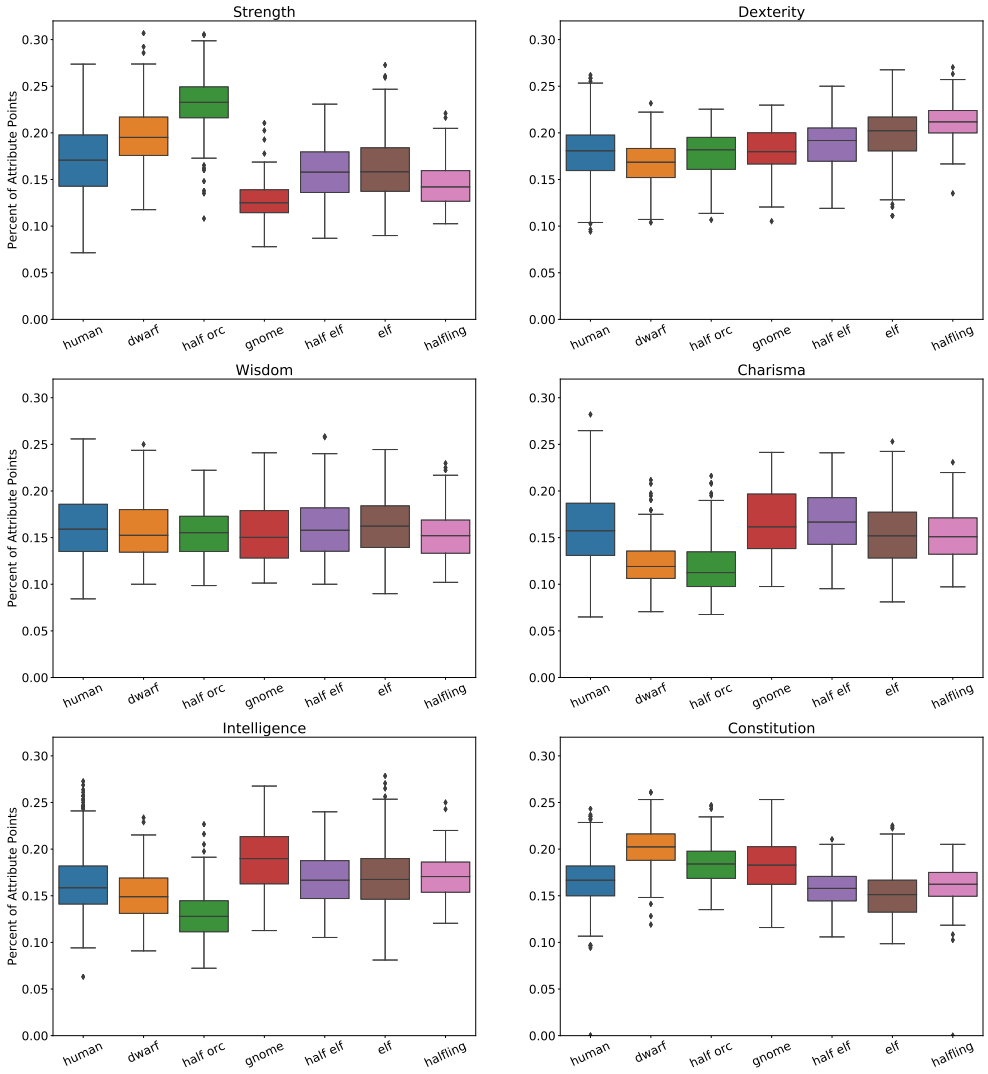



**Figure 5.9:** Distributions of attribute scores among race.

The problem with this approach, of course, is that the goal of the project is to use backstories alone to generate the character's stats - the class and race should be unknown. However, as shown, it is possible to *predict* a character's class and

race. Thus, the feature space used to train attribute score regressors is the predicted class and race for a character using the ensemble models discussed in the previous section. This feature space has 18 dimensions, one for each class and race and is the concatenation of the predicted probabilities of the the two ensemble models.

With this feature space in mind, the training and testing sets for attribute scores are constructed using only characters with known class and race. It doesn't make sense to include a character with an unrecognized class or race, since one cannot hope to accurately predict its class or race using a classifier that only spits out probabilities for known labels. This is however, a large compromise - some characters which were part of the training sets for class and race show up in the training data for the attribute scores. Predictions for class and race of these characters are informed predictions, since the class and race classifiers were trained on them. This is unavoidable, however, if large enough training and testing sets are to be used.

In addition, before creating the training and testing sets, the Local Outlier Factor algorithm is used to remove outliers from the available characters. This algorithm identifies anomalies in data as instances which have much lower density than their neighbors. In other words, if a training instance is farther away from its neighbors than those neighbors are close to their own neighbors, then the instance is dropped. Removing outliers is important in regression since outliers can have a dramatic pull, especially on linear models. Table 5.5 shows the final testing and training sets for attribute scores as well as how many outliers are detected.

| Total Instances | Outliers | Valid | Training | Testing |
|---|---|---|---|---|
| 3028 | 253 | 2775 | 2220 | 555 |

**Table 5.5:** Training and testing sets for attribute regression.

### 5.2.1 Baseline Models and Metrics

This project uses explained variance and absolute error as analysis for the final regression models. These metrics differ fundamentally from those used for the race and class classifiers just as the tasks of regression and classification do.

Explained variance reveals how good the model is at capturing the variance found within the testing data. If $y_{true}$ is a vector of true targets and $y_{pred}$ is a vector of predicted targets, then the explained variance score of these predictions is defined as

$$1 - \frac{Variance\{y_{true} - y_{pred}\}}{Variance\{y_{true}\}}$$

Thus, the explained variance ranges from $[0, 1]$, and can be viewed as a percentage. A perfect model captures 100% of the variance. In the case of targets with multiple values, such as the attribute scores, the explained variance of each output is calculated, and then the aggregate is averaged.

Absolute error is simply the absolute value of the difference between a prediction and the actual value for an instance. For targets with multiple outputs, the absolute error represents the averaged absolute error of all outputs. This isn't to be confused with mean absolute error, which is the average of absolute error across all predictions. In the context of attribute scores, the absolute error for a character represents the average percentage that any one attribute of that character is off.

In the case of the attribute scores, the baseline model finds the average attribute score in the training data and uses it as the prediction for any testing instances. This model makes no assumptions about the character backstories, unlike the classification baselines, but there doesn't seem to be any easily exploitable information to refine it with.

### 5.2.2 Model Selection

Two models are compared when predicting attribute scores: linear regression using least squares and an MLP. These models were selected because linear regression is one of the least complex regression models, while an MLP is capable of picking up on nonlinear relationships within data.

There are virtually no hyperparameters to tune for the linear regression model used and the MLP was implemented using Tensorflow, so neither conformed well to the exhaustive grid search method. The MLP's parameters (e.g hidden layer neuron counts, number of hidden layers, learning rate) were tuned manually until its performance stopped increasing on the testing set. This is by no means good practice, and in the future these parameters should be chosen using some sort of validation set. Because of this method, the MLP can be assumed to be slightly overfit to the testing data.

With this established, the final structure of the MLP used is an input layer with 18 neurons (one for each class and race predicted probability), two hidden layers with 75 neurons using the ReLU activation function, and an output layer with six neurons (one for each attribute score) using the soft-max activation function. The soft-max activation forces the final output to sum to 1.

Table 5.6 shows the results of each of these models as well as the baseline on the testing data. Immediately, it is apparent that the baseline is very poor at capturing the variance in the testing data, but this is expected since it is just a flatline average. The linear regression is slightly outdone by the MLP, probably because of slight nonlinear relationships between the probabilities of class and race and the attribute scores. While both of the models' explained variances are under 50%, this is to be expected in hard to predict problems. Values under 50% are seen in many studies trying to predict human behavior, for example.

Interestingly, the mean absolute error of all three models appear very close at

first glance. When converted to actual attribute points (using a level 1 character's point pool of 72), the discrepancy seems a little more impactful. The last column of Table 5.6 shows this adjustment and represents the attribute points each model would be off by for the average attribute (assuming a level 1 character).

| **Model** | **Explained Variance (%)** | **Mean AE (x$10^2$)** | **Mean AE as Attribute Points** |
|---|---|---|---|
| Baseline | 5.00e-14 | 2.67 | 1.92 |
| Linear Reg. | 39.5 | 1.99 | 1.43 |
| MLP | 42.5 | 1.93 | 1.38 |

**Table 5.6:** Performance metrics of attribute score regressors.

Still, these metrics don't reveal much about the spread of error in each model and lump the performance on each attribute together, omitting information. As already seen in Figure 4.4, the distribution of each attribute is not the same. Thus, Figure 5.10 explores the raw error of the baseline and MLP on each attribute separately (the linear regression model is excluded here since it performed similarly to the MLP). Each plot shown contains the estimated probability distribution of error for both the MLP and baseline model on an attribute. The y-axis denotes probability while the x-axis represents error in attribute points (assuming level 1 point pool of 72). In the background of each plot is the histogram of errors for both models used to estimate the probability distribution.

These plots reveal that for each attribute the standard deviation of the attribute scores for the MLP was lower than the baseline. In addition, we can see the the error distributions were much closer for the two models for the attributes with lower variations with respect to class and race as seen in Figures 5.8 and 5.9 (e.g. constitution).

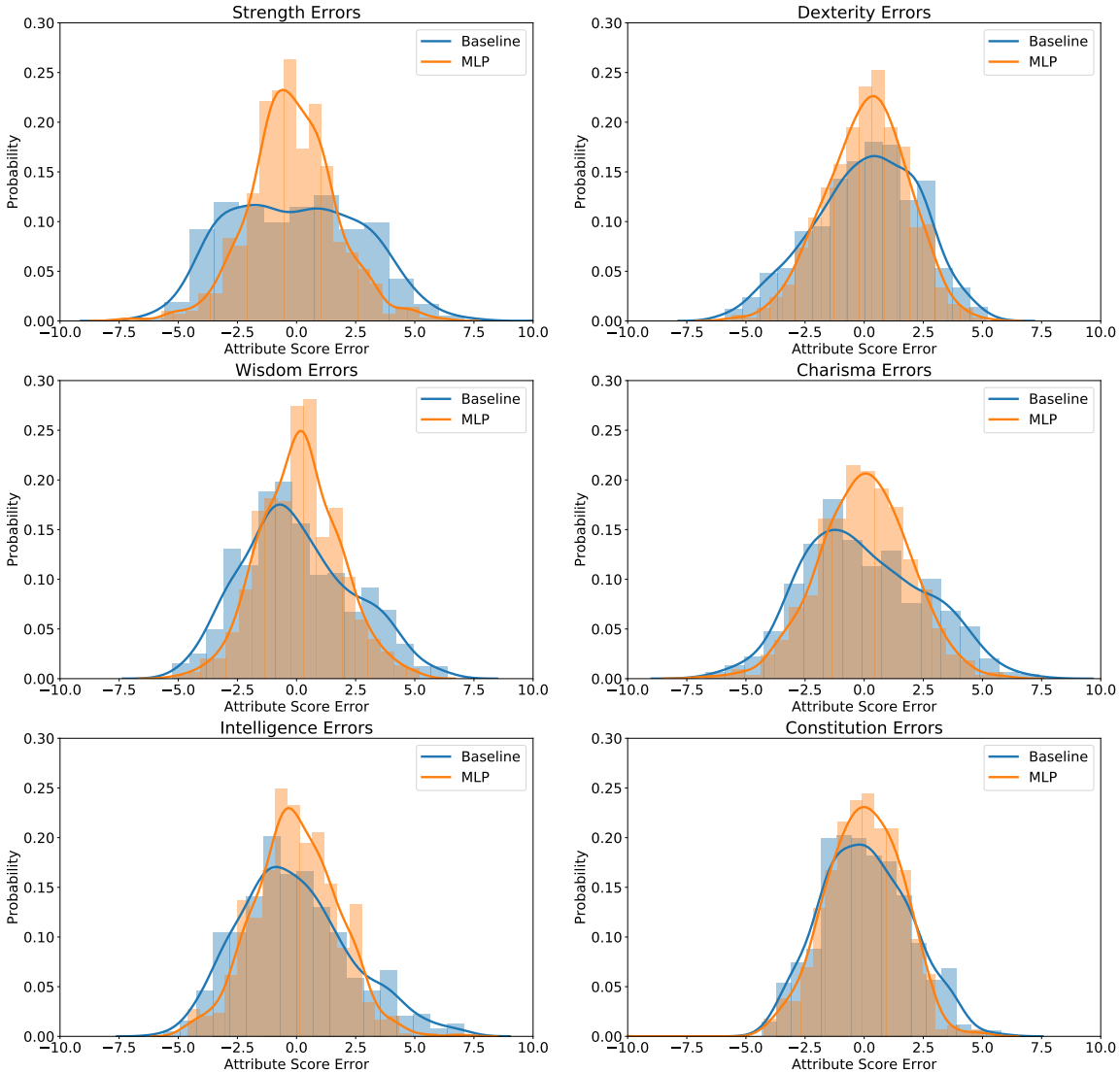



**Figure 5.10:** Raw errors of MLP and baseline on each attribute.

*CHAPTER 6*

# Conclusion

The results in the previous section indicate that it is possible to make predictions about a D&D character using only their backstory. In each category - class, race, and attribute scores - we are able to outperform the simple guesses modeled by the baselines. The classifier for race had a higher classification accuracy than that of class, but this is expected. Race was both more imbalanced and had fewer classification labels than class, which both contribute to this. Finally, while the attribute score regressors were not outstanding, it is important to keep in mind that they were trained on predicted outputs already containing error.

The question becomes, then, how *much* can one say about the information on a character sheet given the corresponding backstory. Often these two can clash; as a player develops their character over time through leveling up, it can start to look less and less like the character they describe in the backstory. Furthermore, the misclassifications of the classification models demonstrate that it there is a certain amount of overlap between some characters (e.g. wizards and sorcerers).

In order to explore this further, there are a number of changes that could be pursued. First, there are several parameters that could be tuned rather than using a 'best-guess' approach. These include the minimum frequency cutoff for the TF-IDF vectorization of the backstories, the percentage of the majority class dropped in the undersampling in the case of race, and the minimum character cutoff for dropping

backstories. In addition, when training the various classification models, accuracy score was used to choose the best model-parameter combinations. Other metrics could be used to choose this model (e.g $F_1$ score).

Other than the changes that could be made, there are also other venues of approach to the problem that might reveal more information. One of the big drawbacks to the existing approaches is failing to incorporate the impact of nearby words on a given word. We address each word in isolation, essentially, even when using the TF-IDF averaged contexts.

There are several techniques that might be used to tackle this problem. The simplest is to use an $n$-gram method; instead of considering individual words as tokens, one considers each $n$-combination of words as a token. For example using bi-grams, "hates humans" would be a token in the sentence "Thortoq hates humans." This approach explodes the feature space, but captures some of the context in sentences.

Another approach, which wouldn't lead to as large a feature space, is to incorporate sentiment analysis into the existing TF-IDF feature space. One could create two values for each word, the TF-IDF value of the word in negative sentences and the TF-IDF value of the word in positive sentences. This might capture some more information about what the word means for the backstory.

A more complicated approach is to use recurrent neural networks with long short term memory cells for text classification. Early in the project, this method was pursued, but the performance was poor. While these networks have been met with success in NLP tasks [23], they require large amounts of data, something this project does not have.

In an effort to account for this, however, and put the models into the hands of D&D players, a frontend was developed for the models (it currently lives at chardd.net ). Using Python's Flask framework, Chart.js and Highcharts, Docker,

and Amazon Web Services, this frontend allows users to enter in a backstory and receive their class, race, and attribute score predictions. It also logs these predictions and saves the backstory (and in the future will save what the user thought should be predicted). In this way, it gathers additional data of the sort used in the project. Hopefully, in the future, this can be used to create more accurate models.

*APPENDIX A*

# Character Sheet

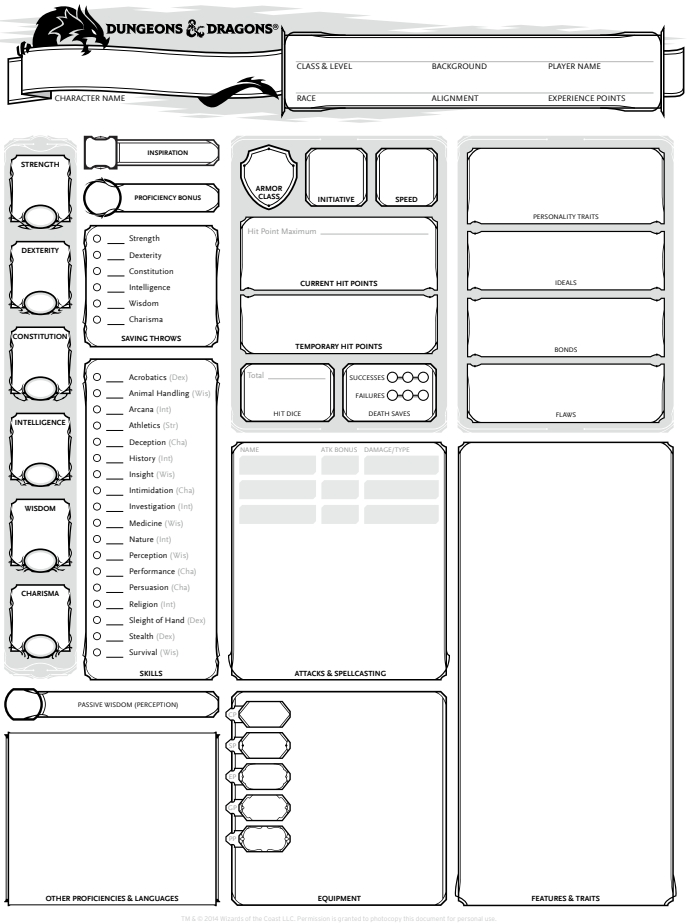



**Figure A.1:** D&D 5th Edition Character Sheet

# References


1. Amazon best sellers in fantasy gaming. https://www.amazon.com/Best-Sellers-Books-Fantasy-Gaming/zgbs/books/16211.

2. Sklearn datasets. URL https://scikit-learn.org/stable/datasets/index.html.

3. word2vec, 2013. URL https://code.google.com/archive/p/word2vec/.

4. List of 2017 nobel laureates. https://www.nobelprize.org/list-of-2017-nobel-laureates/, 2017.

5. Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. On the surprising behavior of distance metrics in high dimensional space. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory — ICDT 2001*, pages 420–434, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

6. R.E. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton Legacy Library. Princeton University Press, 2015.

7. C.M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer New York, 2016.

8. N. Chomsky and D.W. Lightfoot. *Syntactic Structures*. Mouton classic. Bod Third Party Titles, 2002.

9. deanbigbee20. The orr group industry report - q1 2018. http://blog.roll20.net/post/174833007355/the-orr-group-industry-report-q1-2018, 2018.

10. Wallace Gibbons. Dungeons & dragons 3rd edition character database. http://www.3edb.com.

11. Joseph O'Rourke Hugh Kenner. A Travesty Generator for Micros. *BYTE*, 9, 1984.

12. G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer Texts in Statistics. Springer New York, 2014.

13. Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

14. Karen Sparck Jones. *Natural Language Processing: A Historical Review*, pages 3–16. Springer Netherlands, Dordrecht, 1994.

15. Donald Knuth. On the Translation of Languages from Left to Right. *Information and Control*, 8:607–639, 1965.

16. William N. Locke and A. Donald Booth. Machine translation of languages. *American Documentation*, 7(2):135–136, 1956. doi: 10.1002/asi.5090070209.

17. C.D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

18. George A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38:39–41, 1995.

19. Andrew Ng. *Machine Learning Yearning*. Unpublished, 2019.

20. Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

21. John R. Pierce and John B. Carroll. *Language and Machines: Computers in Translation and Linguistics*. National Academy of Sciences/National Research Council, Washington, DC, USA, 1966.

22. Shalan. Week 11 multilayer perceptron, oct 2010. URL `http://shahlanbmi.blogspot.com/2010/10/week-11-multilayer-perceptron.html`.

23. Peilu Wang, Yao Qian, Frank K. Soong, Lei He, and Hai Zhao. Part-of-speech tagging with bidirectional long short-term memory recurrent neural network. *CoRR*, abs/1510.06168, 2015.

24. Xiang Zhang and Yann LeCun. Text understanding from scratch. *CoRR*, abs/1502.01710, 2015.