Trends in Good and Evil - D&D Character Data (2018-2022)

Overview

We are Yarin Mendel and Asher Burrows, computer science students from the Open University of Israel. Sharing an interest in both social trends and gaming, we decided to look at societal views of good and evil through the lens of Dungeons and Dragons (from here on - DnD or D&D).

Millions of players across the world create hundreds of thousands of DnD characters every day, give them a race, a profession, a body type, mental and emotional characteristics and a name, and label them 'good' or 'evil'. We saw these characters as a unique window into human opinion on good and evil and set out to see what the data could tell us on what our societies deem as good and what they deem evil.

Our intuition told us that race and class (profession) would be the biggest indicators on the good/evil scale and our intial data exploration indicated that beyond that, good and evil characters might not have the same attribute distributions. Good characters seem to be stronger than evil characters. Evil characters more charismatic. How significant are these differences and what do they mean?

We set out to answer these questions and to answer a larger question: given a character - it's race, background and characteristics, could we predict what kind of character the player was creating, could we predict if it was created to be good or evil.

Dataset and Context

The dataset comes from this git repository. The repo contains several datasets collected from the owner's web apps for creating and updating DnD character sheets, and represents approximately 8000 user submitted DnD characters over the course of four years: 2018-2022.

A number of issues exist with the dataset, many of which the owner has done significant work to iron out. The following are some of the more critical ones:

- The web apps allow free text entry for many fields, leading to misspellings and nonstandard class, race and backgrounds.
- The app does not have a standard encoding for race and subraces.
- Some characters are dupicates, remade at higher levels as the characters progressed.
- Some characters are near duplicates, with some small change, a different race, for example. These presumably reflect users who tried several variations of the same character.
- The app allows users to create characters with fields left blank. Relevant especially to our research, only 2000 or so of the total 8000 characters have been assigned an alignment.

The owner went to great lengths to correct and standardize the data. Spell and weapon names were autoassigned standardized labels by Levenshtien distance, classes were standardized and alignment (a free text field) was manually corrected. For the unified dataset (a collection of all the datasets collected by the owner) which we will be using, duplicate characters with different levels were discarded except for the highest level instance.

For our purposes we will be ignoring any entries without an alignment.

```
!pip install pandas
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (1.5.3)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2.8.
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2023.3.post1)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from pandas) (1.23.5)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1->pan
```

Data Analysis

Our objective is to examine the impact of various character attributes on the 'good'/'evil' alignment within our dataset. We will begin by analyzing the distribution of alignments across different races and classes, looking at heatmaps to visualize the trends.

For the numerical character attributes—Wisdom, Intelligence, Constitution, Dexterity, Strength, and Charisma—we plan to explore potential correlations by plotting their distributions for good and evil alignments and consider the application of Principal Component Analysis (PCA) for more refined insights.

Additionally, we will assess the 'countryCode' and 'castingStat' fields as potential categorical predictors. Although spells known could be indicative of alignment, the extensive variety and categorical nature of this data impose limitations on our current analysis framework, prompting its exclusion. Similarly, we will not be incorporating character names into our analysis due to the complexity they introduce, which would require more sophisticated text analysis methods beyond the scope of this project.

```
import pandas as pd

df = pd.read_csv('dnd_chars_unique.csv')
```

Good and Evil Tagging Analysis

First let’s get a general picture of our data and the tagging. We will be using the “processedAlignment” field as tags. D&D characters are given alignments on two axes: the good-neutral-evil axis and the chaotic-neutral-lawful axis. For example, a character could be labled “chaotic neutral” or “lawful evil”. The raw data in the free text “alignment” field is highly irregular, but a processed standardized alignment has been provided in the “processedAlignment” field. Each alignment is represented by a two letter tag. The first letter denoting chaotic/lawful alignment and the second good/evil alignment. NG for Neutral Good or CE for Chaotic Evil, for example.

Let's see how complete our dataset is and what the tagging distribution looks like:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

non_empty_count = df['processedAlignment'].count()
unique_counts = df['processedAlignment'].value_counts()

print(f"Non-empty count for processedAlignment: {non_empty_count}")
print(f"Unique count for processedAlignment:\n {unique_counts}")
```

```
Non-empty count for processedAlignment: 2024
Unique count for processedAlignment:
 CN    447
CG    445
NG    373
NN    229
LN    211
LG    199
LE     55
NE     40
CE     25
Name: processedAlignment, dtype: int64
```

Only about 2k entries of the total 8k in the dataset are tagged. We will note here that this is a small dataset to work with for this kind of problem. The distribution over alignments is pretty wide, showing a heavy bias towards good and neutral characters.

It's difficult to work with a tag that has eight possible values so we would like to condense this to a simpler tag: good, neutral or evil. This will also allow us to represent the alignment as the numerical values -1, 0 and 1 which will make analysis simpler.

There are two ways to condense the field. We could simply drop the chaotic/lawful element and be left with a good/evil tag, or we could try and incorporate both elements into a unified positive/negative alignment. Although it is tempting to do the former, that approach is problematic. Besides for possible information loss, the distribution recieved is quite imbalanced: only 120 evil to 887

neutral and 1,017 good. The incredibly small evil sample size might even make it impossible to find clear trends. In light of that we have decided to go with the second approach.

To make the transformation we have designated good and lawful as positive values worth '1' and evil and chaotic as negative values worth '-1'. The values are added together to get the positivity. Positive values are tagged as '1' and negative as '-1', with 0 being neutral.

```
def new_alignment_symbol(a, b):
  addition = a + b

  if addition > 0:
    return 1

  if addition < 0:
    return -1

  return 0

# lawfulness (Lawful, Natural, C) and the second one is for goodness (G,N,E)
string_to_int_mapping = {
    'CN': new_alignment_symbol(-1, 0),
    'CG': new_alignment_symbol(-1, 1),
    'NG': new_alignment_symbol(0, 1),
    'NN': new_alignment_symbol(0, 0),
    'LN': new_alignment_symbol(1, 0),
    'LG': new_alignment_symbol(1, 1),
    'LE': new_alignment_symbol(1, -1),
    'NE': new_alignment_symbol(0, -1),
    'CE': new_alignment_symbol(-1, -1)}

df['intAlignment'] = df['processedAlignment'].map(string_to_int_mapping)

# Count the occurrences of each integer
integer_counts = df['intAlignment'].value_counts()

# Display the results
print(f"Mapping of unique strings to integers:")
print(string_to_int_mapping)
print()
print(f"Count of each integer:")
print(integer_counts)
```

```
    Mapping of unique strings to integers:
    {'CN': -1, 'CG': 0, 'NG': 1, 'NN': 0, 'LN': 1, 'LG': 1, 'LE': 0, 'NE': -1, 'CE': -1}
```

```
Count of each integer:
 1.0    783
 0.0    729
-1.0    512
Name: intAlignment, dtype: int64
```

Our new alignment tag has a fairly even distribution between the three fields which should give us the best shot at getting any useful information out of this amount of data.

Race and Class Analysis

More than the particular attributes a character may have, its class, and to a lesser extent race, determine the core of the character both in narrative and in mechanics. Let's take a look at the class and race data for our labled dataset. The raw Class and Race columns are rather unstructured so again we will be using the processed columns: "justClass" and "processedRace".

```
print("Classes:")
print(df['justClass'].unique())

print()
print("Races:")
print(df['processedRace'].unique())
```

```
Classes:
['Ranger' 'Druid' 'Fighter' 'Cleric' 'Wizard' 'Warlock' 'Bard' 'Paladin'
 'Barbarian' 'Monk' 'Rogue' 'Sorcerer' 'Mystic' 'Artificer' 'Blood Hunter'
 'Revised Ranger' 'Gunslinger' 'Crafting Commoner' 'Battle Clown'
 'commoner' 'Fighter|Cleric' 'Fighter|Paladin' 'Barbarian|Sorcerer'
 'Wizard|Bard' 'Bard|Fighter' 'Artificer|Cleric' 'Fighter|Warlock'
 'Monk|Bard' 'Monk|Cleric' 'Barbarian|Rogue' 'Fighter|Barbarian'
 'Wizard|Warlock' 'Cleric|Bard|Fighter|Druid' 'Fighter|Rogue'
 'Barbarian|Fighter' 'Cleric|Rogue' 'Fighter|Monk'
 'Druid|Paladin|Barbarian' 'Rogue|Cleric|Bard|Fighter|Druid|Wizard'
```

```
 'Bard|Sorcerer' 'Bard|Warlock' 'Warlock|Sorcerer' 'Wizard|Rogue'
 'Cleric|Druid' 'Paladin|Bard' 'Rogue|Ranger' 'Rogue|Bard'
 'Paladin|Fighter' 'Rogue|Fighter' 'Fighter|Bard' 'Rogue|Monk'
 'Sorcerer|Fighter' 'Paladin|Warlock' 'Paladin|Barbarian'
 'Barbarian|Druid' 'Barbarian|Paladin' 'Monk|Barbarian' 'Bard|Barbarian'
 'Rogue|Warlock' 'Sorcerer|Cleric' 'Ranger|Rogue' 'Fighter|Druid'
 'Fighter|Warlock|Ranger' 'Bard|Cleric' 'Cleric|Bard' 'Wizard|Fighter'
 'Fighter|Warlock|Sorcerer' 'Druid|Ranger' 'Ranger|Cleric' 'Monk|Rogue'
 'Barbarian|Bard' 'Fighter|Warlock|Cleric' 'Warlock|Bard' 'Fighter|Ranger'
 'Wizard|Barbarian|Bard|Cleric|Druid|Fighter|Monk|Paladin|Ranger|Rogue|Sorcerer|Warlock|Artificer'
 'Rogue|Paladin' 'Druid|Rogue' 'Fighter|Artificer'
 'Sorcerer|Cleric|Warlock' 'Revised Ranger|Cleric' 'Druid|Barbarian'
 'Paladin|Sorcerer' 'Rogue|Wizard' 'Sorcerer|Warlock'
 'Wizard|Bard|Barbarian|Cleric|Fighter|Ranger|Monk|Paladin|Druid|Rogue|Sorcerer|Warlock'
 'Barbarian|Fighter|Paladin' 'Monk|Fighter' 'Rogue|Fighter|Paladin'
 'Cleric|Fighter' 'Cleric|Sorcerer' 'Fighter|Wizard' 'Barbarian|Monk'
 'Cleric|Wizard' 'Cleric|Monk' 'Druid|Monk' 'Fighter|Sorcerer|Wizard'
 'Rogue|Warlock|Fighter' 'Bard|Rogue|Warlock' 'Ranger|Monk'
 'Sorcerer|Warlock|Fighter' 'Warlock|Rogue' 'Rogue|Sorcerer'
 'Warlock|Barbarian' 'Fighter|Sorcerer' 'Fighter|Ranger|Rogue'
 'Paladin|Warlock|Bard' 'Rogue|Fighter|Ranger' 'Barbarian|Warlock'
 'Ranger|Druid' 'Ranger|Fighter|Rogue' 'Barbarian|Monk|Fighter'
 'Druid|Fighter' 'Blood Hunter|Rogue' 'Bard|Rogue'
 'Fighter|Wizard|Artificer' 'Artificer|Wizard' 'Paladin|Cleric'
 'Sorcerer|Barbarian' 'Ranger|Fighter' 'Rogue|Barbarian' 'Druid|Cleric'
 'Wizard|Artificer' 'Barbarian|Cleric' 'Artificer|Bard' 'Druid|Bard'
 'Monk|Fighter|Cleric' 'Ranger|Barbarian'
 'Fighter|Ranger|Druid|Artificer|Bard' 'Artificer|Warlock'
 'Revised Ranger|Fighter' 'Fighter|Wizard|Rogue' 'Ranger|Sorcerer'
 'Wizard|Monk' 'Rogue|Cleric' 'Sorcerer|Druid' 'Druid|Sorcerer'
 'Wizard|Cleric' 'Wizard|Barbarian' 'Warlock|Wizard' 'Rogue|Artificer'
 'Warlock|Fighter' 'Ranger|Warlock' 'Druid|Wizard'
 'Fighter|Bard|Warlock|Rogue|Sorcerer' 'Cleric|Artificer' 'Bard|Druid'
 'Barbarian|Revised Ranger' 'Barbarian|Fighter|Wizard' 'Sorcerer|Wizard'
 'Bard|Monk' 'Bard|Paladin' 'Sorcerer|Bard' 'Paladin|Rogue' 'Wizard|Druid'
 'Warlock|Paladin' 'Druid|Warlock' 'Artificer|Fighter|Wizard'
 'Rogue|Druid' 'Cleric|Warlock']

Races:
['Kenku' 'Aasimar' 'Leonin' 'Warforged' 'Tiefling' 'Human' 'Turtle'
```

```
'Half-Orc' nan 'Firbolg' 'Dragonborn' 'Tabaxi' 'Loxodon' 'Genasi' 'Dwarf'
'Goliath' 'Halfling' 'Elf' 'Half-Elf' 'Gnome' 'Triton' 'Orc' 'Aarakocra'
'Minotaur' 'Eladrin' 'Gith' 'Centaur' 'Yaun-Ti' 'Custom' 'Kobold'
'Goblin' 'Lizardfolk' 'Satyr' 'Shifter' 'Bugbear' 'Changeling'
'Simic hybrid' 'Kalashtar' 'Hobgoblin' 'Vedalken']
```

"Race" is straightforward categorical data, but "Class", as we can see, is more complicated. A character can have more than one class. For this project we have chosen to unpivot the data and split each mutliclass character into multiple single-class characters for ease of analysis.

```
# We will do this code very not "Panda like" because of the complexity of
# the string format of multiple classes.

concated_classes = df['justClass'].unique()

all_classes = list()
for c in concated_classes:
  all_classes.extend(c.split('|'))

classes = list(set(all_classes))

classes
```

```
['Ranger',
 'Paladin',
 'Battle Clown',
 'Druid',
 'Barbarian',
 'Crafting Commoner',
 'commoner',
 'Artificer',
 'Revised Ranger',
 'Fighter',
 'Mystic',
 'Sorcerer',
```

```
      'Blood Hunter',
      'Monk',
      'Gunslinger',
      'Rogue',
      'Cleric',
      'Wizard',
      'Warlock',
      'Bard']
```

Now to get a picture of our data let's count and check the commonality of Class-Race pairs:

```
import numpy as np

class_index = dict()
race_index = dict()

for idx, c in enumerate(classes):
  class_index[c] = idx

for idx, race in enumerate(df['processedRace'].unique()):
  race_index[race] = idx

print(class_index)
print(race_index)

counts = np.zeros((len(classes), len(df['processedRace'].unique())))

for _, row in df.iterrows():
  for c in row['justClass'].split('|'):
    counts[class_index[c]][race_index[row['processedRace']]] +=1

print(counts)
```

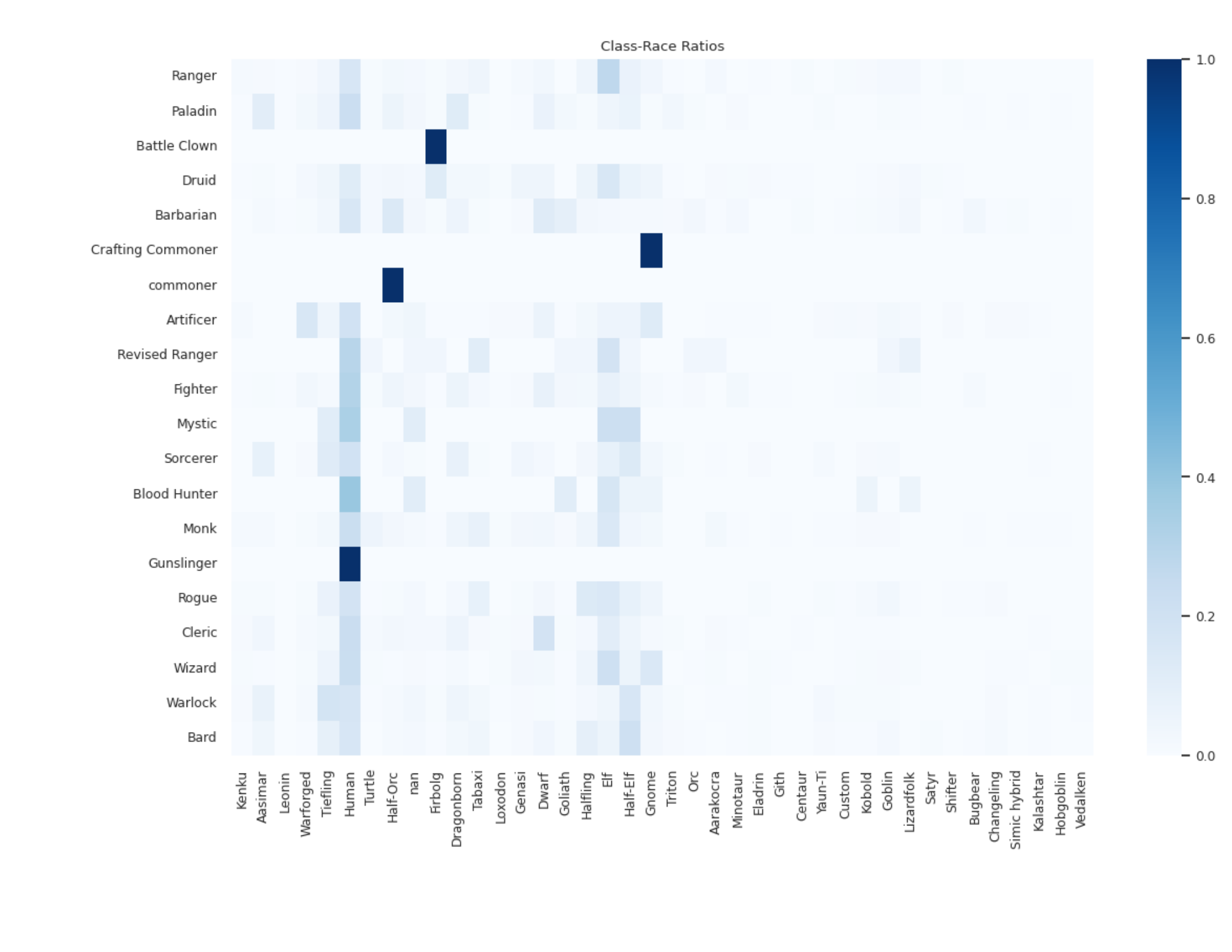
```
     {'Ranger': 0, 'Paladin': 1, 'Battle Clown': 2, 'Druid': 3, 'Barbarian': 4, 'Crafting Commoner': 5, 'commoner': 6,
     {'Kenku': 0, 'Aasimar': 1, 'Leonin': 2, 'Warforged': 3, 'Tiefling': 4, 'Human': 5, 'Turtle': 6, 'Half-Orc': 7, nan
     [[ 11.   7.   4.   7.  22.  93.   5.  14.  12.   6.  16.  28.   1.  11.
```

```
  19.   1.  25. 155.  40.  21.   3.   2.  13.   2.   4.   0.   5.   0.
   5.   7.  12.  13.   1.   5.   1.   0.   1.   0.   0.   0.]
[  3.  68.   3.  16.  34. 141.   7.  35.  18.   5.  76.   7.   2.   4.
  42.  15.   7.  30.  41.   6.  20.   6.   2.   8.   1.   1.   2.   5.
   2.   2.   5.   3.   0.   1.   4.   1.   3.   0.   4.   0.]
[  0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[  4.   6.   0.  11.  22.  67.  13.  17.  11.  66.  14.  15.   6.  24.
  24.   1.  31.  89.  35.  27.   3.   1.   7.   6.   7.   3.   4.   2.
   1.   4.   7.  13.   5.   4.   1.   1.   1.   2.   1.   0.]
[  1.  12.   5.   8.  24. 115.  15. 103.  20.   6.  39.   6.   1.   9.
  89.  68.  20.  16.  12.  14.   9.  22.   5.  17.   0.   1.   6.   1.
   3.   8.  12.  20.   1.   5.  23.   5.   6.   1.   3.   0.]
[  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   1.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[  0.   0.   0.   0.   0.   0.   0.   1.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[  3.   0.   0.  26.   8.  32.   0.   4.   7.   1.   1.   1.   2.   2.
  10.   1.   4.   9.   9.  21.   0.   0.   1.   1.   1.   0.   0.   2.
   3.   2.   4.   3.   0.   2.   0.   2.   2.   1.   0.   0.]
[  0.   0.   0.   0.   0.   8.   1.   0.   1.   1.   0.   3.   0.   0.
   0.   1.   1.   5.   1.   0.   0.   1.   1.   0.   0.   0.   0.   0.
   0.   0.   1.   2.   0.   0.   0.   0.   0.   0.   0.   0.]
[  6.  13.   5.  40.  20. 356.  13.  57.  35.   8.  64.  26.   8.  16.
  87.  35.  31.  89.  53.  18.   8.  17.   3.  28.   6.   7.   4.   0.
   8.  11.  19.  10.   2.   3.  21.   3.   3.   1.   8.   2.]
[  0.   0.   0.   0.   1.   3.   0.   0.   1.   0.   0.   0.   0.   0.
   0.   0.   0.   2.   2.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[  2.  41.   1.   7.  64. 100.   1.  12.   6.   1.  39.   5.   0.  19.
  12.   2.  16.  42.  69.  17.   5.   1.   4.   1.   8.   1.   0.   9.
   2.   7.  10.   1.   2.   1.   2.   2.   1.   4.   0.   1.]
[  0.   0.   0.   0.   0.   7.   0.   0.   2.   0.   0.   0.   0.   0.
   0.   2.   0.   3.   1.   1.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   1.   0.   1.   0.   0.   0.   0.   0.   0.   0.   0.]
[ 10.   9.   1.   6.  14. 122.  28.  16.  10.   4.  25.  40.   6.  16.
  18.   8.  25.  77.  19.  14.   2.   1.  14.   4.   1.   3.   0.   3.
```

```
   3.   7.   7.   2.   0.   2.   4.   1.   3.   3.   3.   1.]
[  0.   0.   0.   0.   0.   2.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
[ 12.  10.   0.  11.  69. 194.   5.   9.  24.   1.  21.  85.   0.  10.
  29.   5. 138. 153.  85.  48.   3.   0.   4.   0.   9.   2.   3.   9.
   6.  17.  34.   5.   3.   5.   8.  14.   3.   0.   4.   1.]
[ 11.  30.   1.  13.  21. 181.  14.  23.  16.  18.  41.  10.   9.  12.
 146.   8.  18.  84.  34.  14.   7.   3.  11.   5.   3.   3.   4.   3.
   6.   6.   6.   3.   3.   2.   0.   3.   1.   6.   3.   0.]
[  5.   3.   0.   6.  31. 136.   5.   3.   9.   3.  11.   2.   5.  18.
  16.   3.  26. 125.  34.  88.   1.   3.   5.   1.   5.   3.   1.   2.
   4.   6.   9.   5.   0.   0.   1.   3.   3.   2.   5.   5.]
```

Ok, now let's plot this into a nice heatmap

```
average_array = counts / counts.sum(axis=1, keepdims=True)
average_df = pd.DataFrame(average_array, index=class_index.keys(), columns=race_index.keys())

plt.figure(figsize=(14, 8))
sns.set(font_scale=0.8)
ax = sns.heatmap(average_df, annot=False, cmap='Blues', fmt='.2f')
plt.title('Class-Race Ratios')

# Add extra space between the colorbar and the heatmap
plt.tight_layout(rect=[0, 0, 0.85, 1])

plt.show()
```


Class-Race Ratios
Ranger
Paladin
Battle Clown
Druid
Barbarian
Crafting Commoner
commoner
Artificer
Revised Ranger
Fighter
Mystic
Sorcerer
Blood Hunter
Monk
Gunslinger
Rogue
Cleric
Wizard
Warlock
Bard
Kenku
Aasimar
Leonin
Warforged
Tiefling
Human
Turtle
Half-Orc
nan
Firbolg
Dragonborn
Tabaxi
Loxodon
Genasi
Dwarf
Goliath
Halfling
Elf
Half-Elf
Gnome
Triton
Orc
Aarakocra
Minotaur
Eladrin
Gith
Centaur
Yuan-Ti
Custom
Kobold
Goblin
Lizardfolk
Satyr
Shifter
Bugbear
Changeling
Simic hybrid
Kalashtar
Hobgoblin
Vedalken
1.0
0.8
0.6
0.4
0.2
0.0

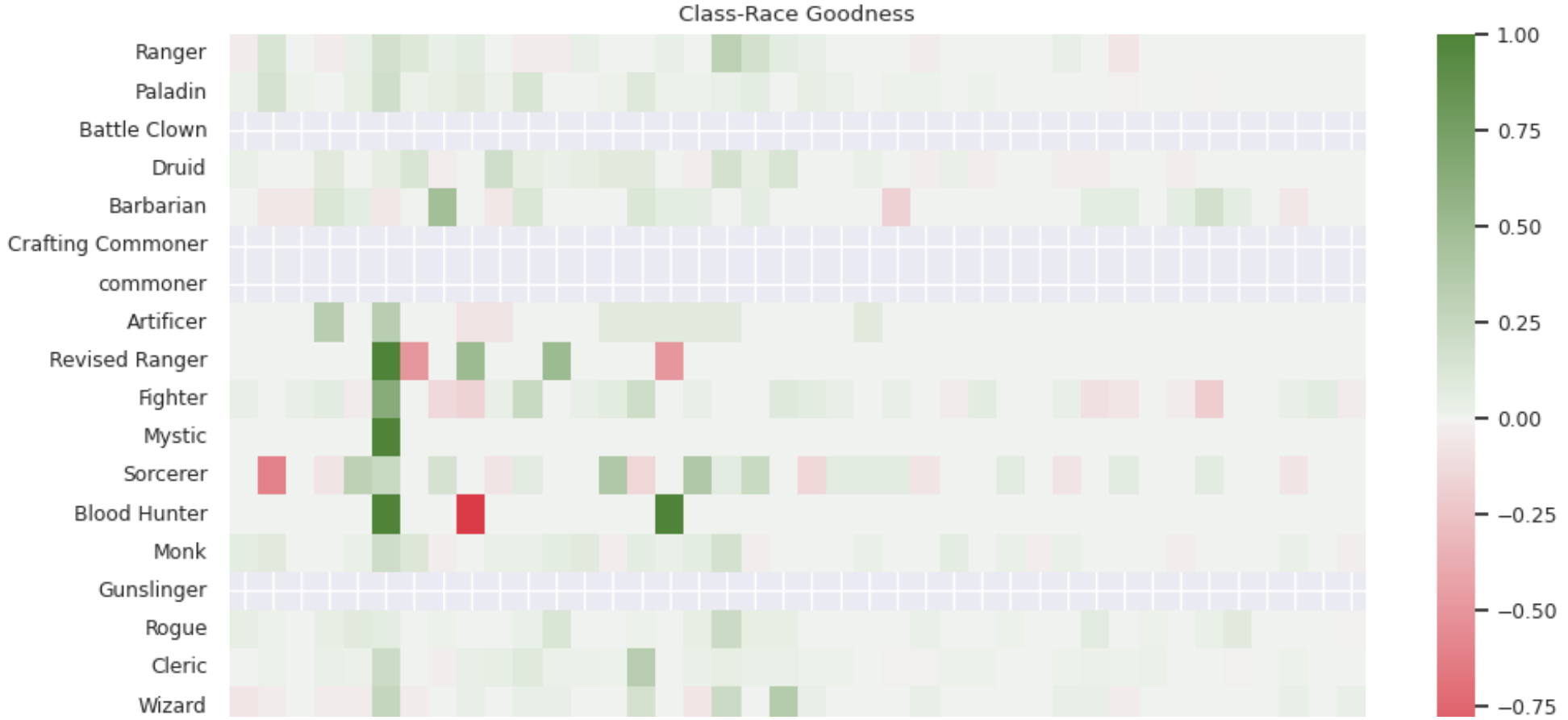
This heatmap is normalized for race and class which makes it somewhat difficult to interpret at a glance. (There are almost no firbolgs, so firbolg battleclowns show up as very high frequency even though there are almost none of them) In any case, some trends are obvious: Humans and Elves are popular. No one plays as Battle-Clowns.

- We should note that to a D&D player this map also shows that there is obviously some garbage in the dataset. No-one plays as the 'commoner' class and it certainly isn't the class of choice for half-orcs.

Now that we have a general picture of the race/class distribution, let's look at the good and evil trends for different race/class combinations:

```
counts = np.zeros((len(classes), len(df['processedRace'].unique())))

for _, row in df.iterrows():
  if row['intAlignment'] == 1:
    for c in row['justClass'].split('|'):
      counts[class_index[c]][race_index[row['processedRace']]] +=1
  if row['intAlignment'] == -1:
    for c in row['justClass'].split('|'):
      counts[class_index[c]][race_index[row['processedRace']]] -=1

import seaborn as sns
import matplotlib.pyplot as plt

average_array = counts / counts.sum(axis=1, keepdims=True)
average_df = pd.DataFrame(average_array, index=class_index.keys(), columns=race_index.keys())

plt.figure(figsize=(12, 6))
sns.set(font_scale=0.8)
cmap = sns.diverging_palette(10, 120, as_cmap=True)
ax = sns.heatmap(average_df, annot=False, cmap=cmap, center=0, fmt='.2f')
plt.title('Class-Race Goodness')

# Add extra space between the colorbar and the heatmap
plt.tight_layout(rect=[0, 0, 0.85, 1])

plt.show()
```

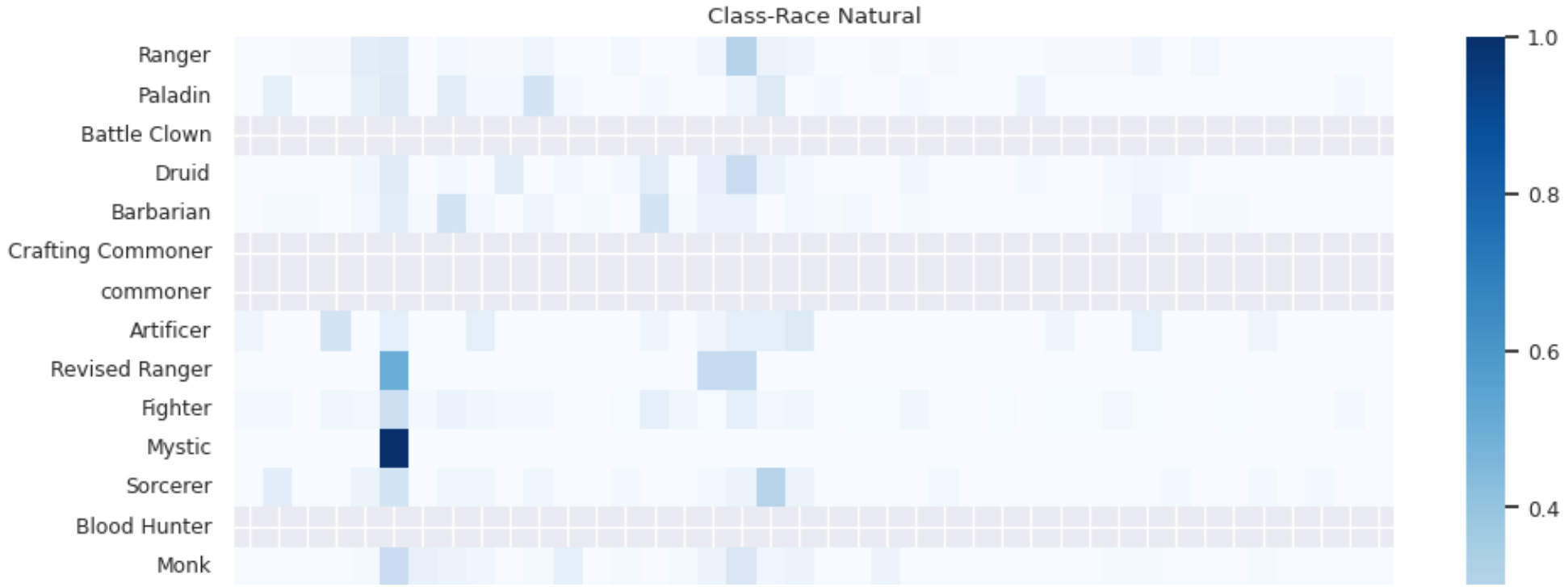
```
<ipython-input-10-420efe4c5a02>:14: RuntimeWarning: invalid value encountered in divide
  average_array = counts / counts.sum(axis=1, keepdims=True)
```



This heatmap illustrates the prevalence of good and evil tendencies across various class-race pairings in our dataset. The color gradient from green to red denotes the proclivity of each pairing towards good or evil alignments, respectively. Shades closer to green signify a stronger association with good alignments, while those closer to red indicate an inclination towards evil. Neutral or balanced alignments are represented by colors close to white. The reveals some mildly pronounced patterns, suggesting that some class-race combinations have a marked predisposition for a moral alignment. Humans, for example seem to be mostly good across classes while many races of warlock tend towards evil. Notably, most pairings do not show a strong bias towards good or evil, indicating a diverse moral landscape within the player-created characters.

```
counts = np.zeros((len(classes), len(df['processedRace'].unique())))

for _, row in df.iterrows():
  if row['intAlignment'] == 0:
    for c in row['justClass'].split('|'):
      counts[class_index[c]][race_index[row['processedRace']]] +=1

import seaborn as sns
import matplotlib.pyplot as plt

average_array = counts / counts.sum(axis=1, keepdims=True)
average_df = pd.DataFrame(average_array, index=class_index.keys(), columns=race_index.keys())

plt.figure(figsize=(12, 6))
sns.set(font_scale=0.8)
ax = sns.heatmap(average_df, annot=False, cmap='Blues', fmt='.2f')
plt.title('Class-Race Natural')

# Add extra space between the colorbar and the heatmap
plt.tight_layout(rect=[0, 0, 0.85, 1])

plt.show()
```

```
<ipython-input-11-841cd6cf3252>:11: RuntimeWarning: invalid value encountered in divide
  average_array = counts / counts.sum(axis=1, keepdims=True)
```



This heatmap depicts the propensity for neutral alignment within the various class-race pairings in our dataset. Darker shades of blue indicate a higher frequency of characters with a neutral alignment, shedding light on the commonality of each pairing's inclination to remain ethically ambiguous. The visualization offers an additional layer of understanding to character creation dynamics, suggesting that a significant portion of the D&D characters are crafted with a balanced approach to morality, possibly to allow a richer personal narrative development or to avoid constraining role-play with a fixed moral outlook.

Particularly, Humans and elves show a strong trend towards neutrality. Given the prevalence of Human and Elven characters in the dataset with a strong neutral representation, this is not surprising.

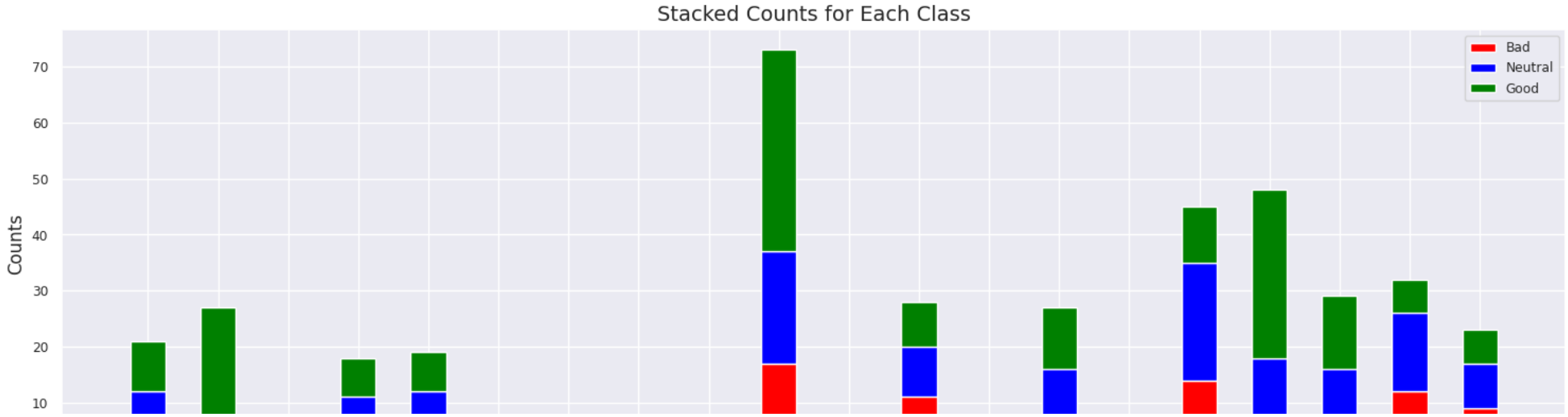
Human Classification Accuracy as a KPI

Given the prominence of human characters within our dataset, it's prudent to consider the accuracy of human classification as a key performance indicator (KPI) for our predictive model. This metric will serve as a benchmark for the model's capability to correctly identify and categorize human characters, which is vital given their significant representation in the data. By focusing on enhancing the accuracy of human classification, we aim to ensure that our model remains robust and reliable, particularly in its interpretation and prediction of the most commonly occurring race-class combinations. This targeted approach in our model evaluation will help us fine-tune its performance, ensuring that the most frequently represented race in our dataset is predicted with the highest precision.

```
counts = { c: [0, 0, 0] for c in classes}

for _, row in df[df['processedRace'] == 'Human'].iterrows():
  if row['intAlignment'] == 1:
    for c in row['justClass'].split('|'):
      counts[c][2] += 1

  if row['intAlignment'] == 0:
    for c in row['justClass'].split('|'):
      counts[c][1] += 1

  if row['intAlignment'] == -1:
    for c in row['justClass'].split('|'):
      counts[c][0] += 1

print(counts)

print("in summary:")

sum_of_counts = [0, 0, 0]
for c in classes:
  sum_of_counts[0] += counts[c][0]
  sum_of_counts[1] += counts[c][1]
  sum_of_counts[2] += counts[c][2]

print(sum_of_counts)
print(sum_of_counts[2]/sum_of_counts[1])
print(sum_of_counts[2]/sum_of_counts[0])
print(sum_of_counts[1]/sum_of_counts[0])

class_counts = counts

class_names = list(class_counts.keys())
counts_array = np.array(list(class_counts.values()))

bar_width = 0.5
index = range(len(class_names))
```

```
plt.figure(figsize=(15, 6))

plt.bar(index, counts_array[:, 0], color='red', width=bar_width, label='Bad')
plt.bar(index, counts_array[:, 1], bottom=counts_array[:, 0], color='blue', width=bar_width, label='Neutral')
plt.bar(index, counts_array[:, 2], bottom=counts_array[:, 0] + counts_array[:, 1], color='green', width=bar_width, label

plt.xlabel('Classes', fontsize=12)
plt.ylabel('Counts', fontsize=12)
plt.title('Stacked Counts for Each Class', fontsize=14)
plt.xticks(index, class_names, fontsize=10, rotation=45)
plt.legend()

plt.tight_layout()
plt.show()
```

```
{'Ranger': [4, 8, 9], 'Paladin': [1, 6, 20], 'Battle Clown': [0, 0, 0], 'Druid': [5, 6, 7], 'Barbarian': [6, 6, 7],
in summary:
[98, 138, 167]
1.210144927536232
1.7040816326530612
1.4081632653061225
```



To summarize this KPI analysis, when we train our model we will aim for minimal error in human alignment classification. In addition we would like to see that our model maintains the good:evil:neutral ratios in human classification of our dataset, predicting:

- good/neutral at a ratio of: 1.2
- good/evil at approximately: 1.7
- neutral/evil at around: 1.4

If our model is predicting a lot of bad humans, for example, it could be an indication that it is failing to distinguish outliers.

Numerical Character Attributes Analysis

The other main source of information in the data that we will be analyzing are the nine numerical fields every character has. These are:

- The six character attributes:
    - Strength
    - Constitution
    - Dexterity
    - Intelligence
    - Wisdom
    - Charisma
- Health (HP)
- Armor (AC)
- Level

Let's look at summary statistics and the distribution of this data.

```
import pandas as pd
filtered_df = df[df['intAlignment'].notnull()]

numerical_attributes = ['Cha', 'Wis', 'Int', 'Con', 'Dex', 'Str', 'AC', 'HP', 'level']

mean_values = filtered_df[numerical_attributes].mean()
std_deviation = filtered_df[numerical_attributes].std()
max_values = filtered_df[numerical_attributes].max()
min_values = filtered_df[numerical_attributes].min()

result_df = pd.DataFrame({
    'Mean': mean_values,
    'Std Deviation': std_deviation,
    'Max': max_values,
    'Min': min_values
})

result_df
```

	Mean	Std Deviation	Max	Min
Cha	13.239625	3.550273	29.0	3.0
Wis	13.341897	2.971724	24.0	1.0
Int	12.076087	3.080585	22.0	3.0
Con	14.329545	2.333024	26.0	4.0
Dex	14.630435	3.025478	29.0	3.0
Str	12.628458	3.854022	29.0	2.0
AC	15.608696	2.646226	28.0	10.0
HP	40.803360	38.733316	450.0	-1.0
level	4.625494	3.841916	20.0	1.0

This analysis provides a picture of a diverse group of characters, mostly in the early to mid stages of development, with a variety of strengths and abilities. The data seems to be in line with standard D&D character creation and progression rules, barring a few exceptions like the negative or extremely high HP value.

```
sum = (filtered_df['HP'] == -1).sum()
sum
```

```
1
```

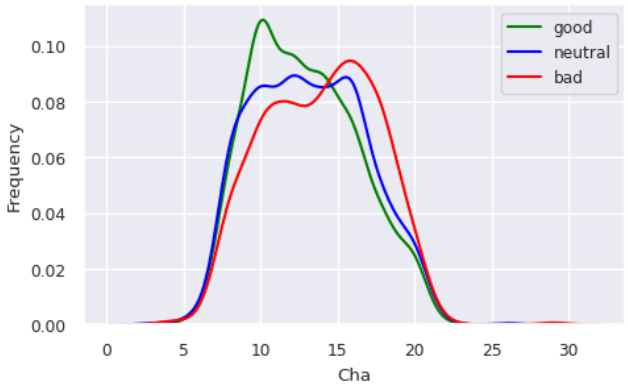
The negative HP entry is a single outlier and probably a just a data error.

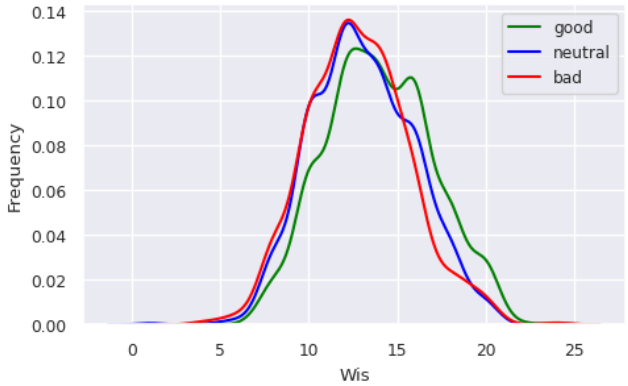
This series of Kernel Density Estimation (KDE) plots provides a comprehensive visual analysis of how different numerical attributes of Dungeons & Dragons characters vary across alignments. Each plot in the 3x3 grid represents the distribution of a specific attribute, segmented by the character's moral alignment (good, neutral, bad). The use of distinct colors for each alignment category (red for bad, blue for neutral, green for good) allows for a clear comparison and understanding of trends. These plots

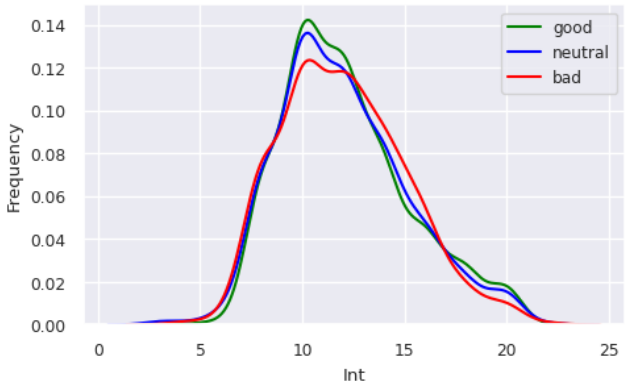
effectively highlight the differences in attribute distributions among the alignments, offering insights into the characteristics that define each alignment category.

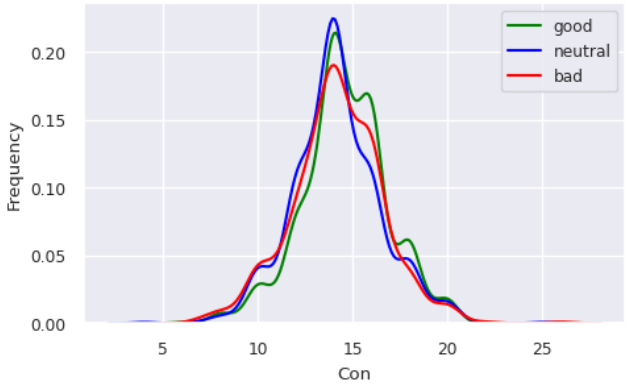
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

fig, axes = plt.subplots(3, 3, figsize=(15, 10))
fig.suptitle('Exploratory Analysis of Numerical Attributes by Alignment', fontsize=16)

alignment_categories = sorted(filtered_df['intAlignment'].unique(), reverse=True)
alignment_colors = {-1.0: 'red', 0.0: 'blue', 1.0: 'green'}
alignment_lables = {-1.0: 'bad', 0.0: 'neutral', 1.0: 'good'}

for i, attribute in enumerate(numerical_attributes):
    row, col = divmod(i, 3)
    ax = axes[row, col]

    for alignment in alignment_categories:
        sns.kdeplot(filtered_df[filtered_df['intAlignment'] == alignment][attribute], label=alignment_lables[alignment],

    ax.set_xlabel(attribute)
    ax.set_ylabel('Frequency')
    ax.legend()

plt.tight_layout()
plt.subplots_adjust(top=0.9)
plt.show()
```

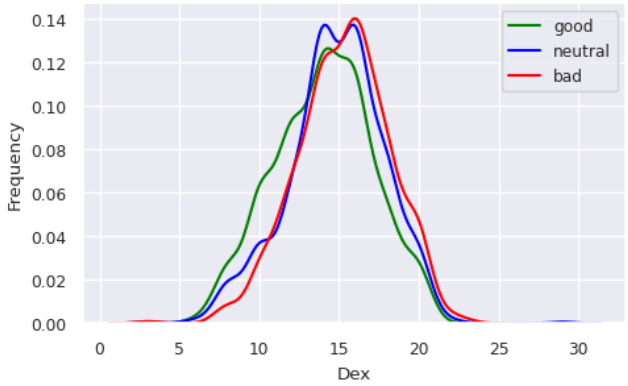
Exploratory Analysis of Numerical Attributes by Alignment

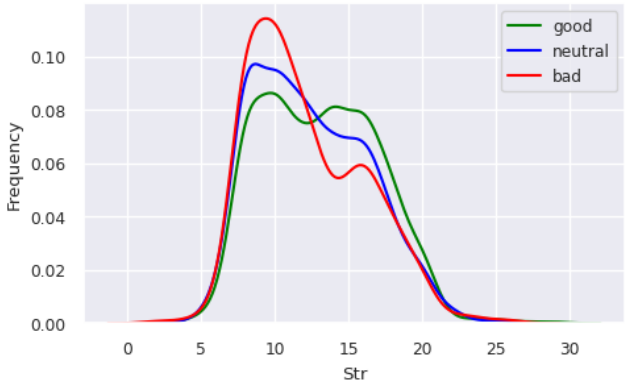


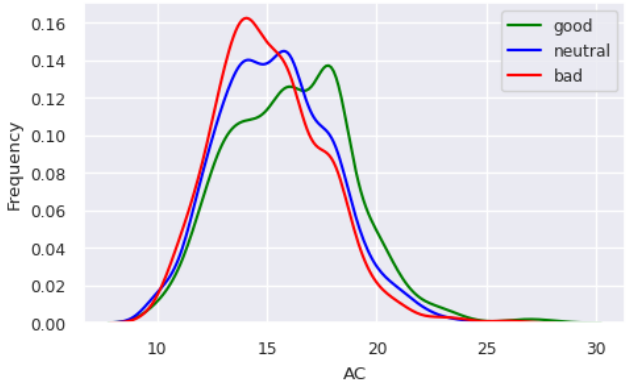


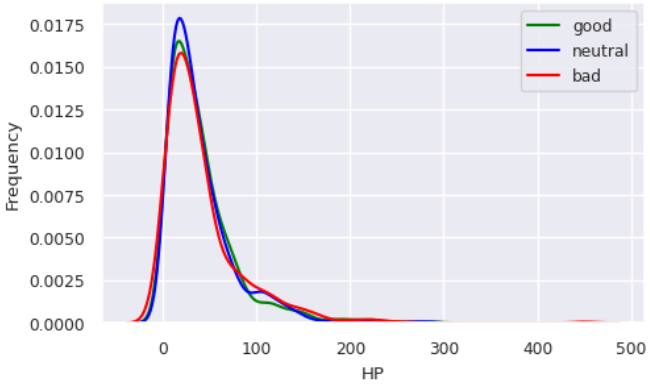


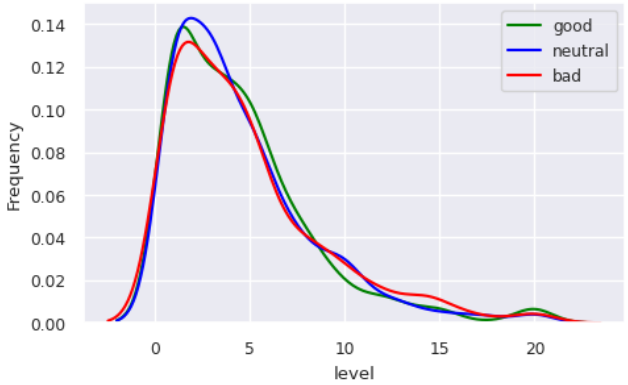












Upon careful examination of the Kernel Density Estimation plots for various numerical attributes by alignment, we observe subtle yet discernible distinctions in the distribution of certain attributes across alignments. Most notably, attributes like Strength and Charisma exhibit distinct peaks and troughs at varying numerical values for each alignment. This variation suggests a potential to utilize these attributes to distinguish between alignments.

For instance, the Strength and Charisma plots reveal that characters with higher Charisma and lower Strength are more likely to be associated with a 'bad' alignment. However, it is crucial to note that the differences in frequency across these attributes are minimal, peaking at a difference of about 0.4. While this does indicate a correlation, the marginal nature of this difference suggests it may not be a strong indicator on its own.

From these insights, we can draw three key observations:

1. Certain attributes demonstrate a tangible correlation with character alignment.
2. The magnitude of these correlations is relatively minor, with frequency differences being quite modest.
3. Several attributes do not offer substantial differentiation between alignments, as their distributions are nearly identical across the spectrum.

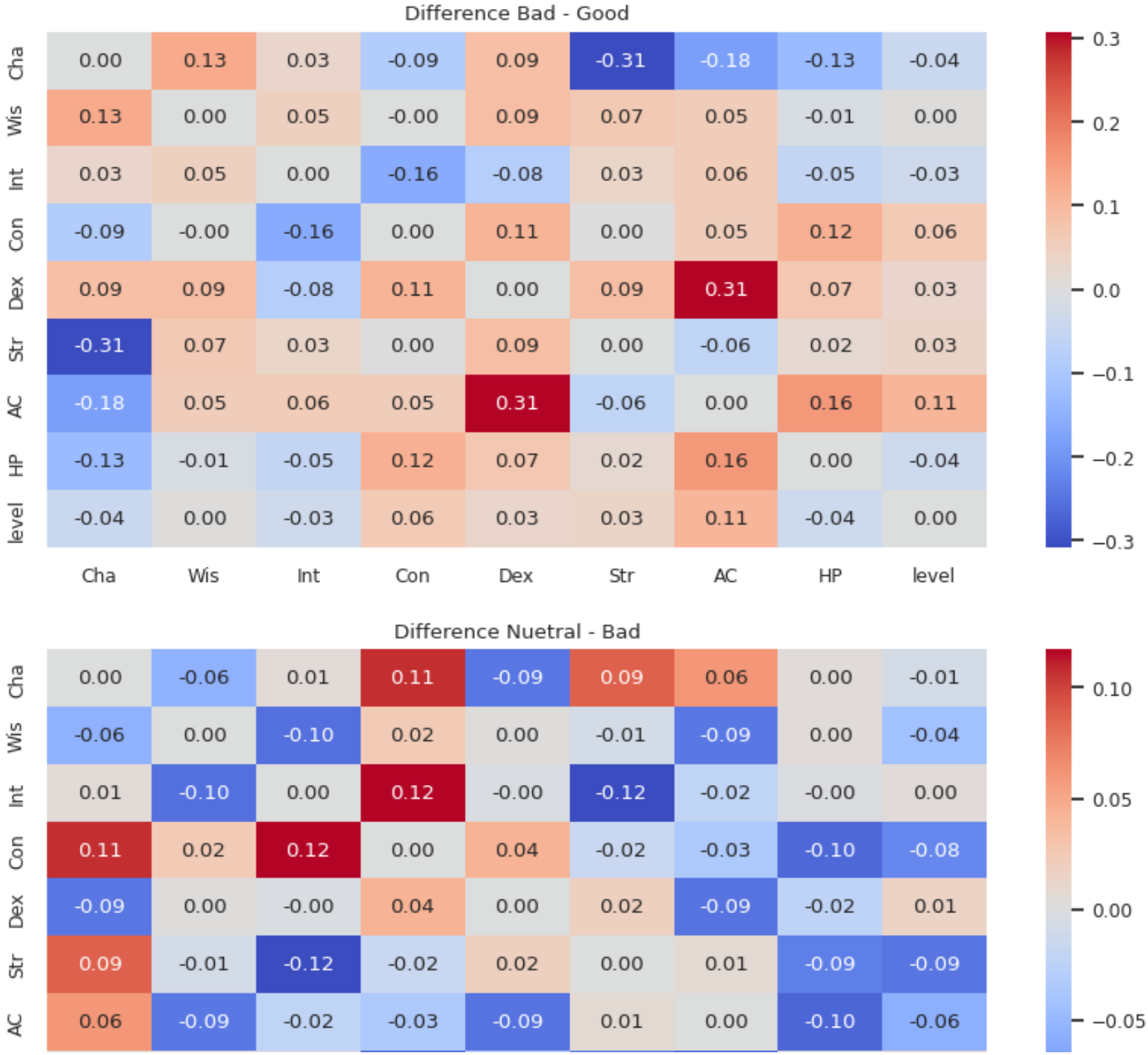
By quantifying the differences in correlation between each attribute and alignment, we can better understand the nuances of character traits as they relate to alignment, which could inform more targeted analyses or character development strategies.

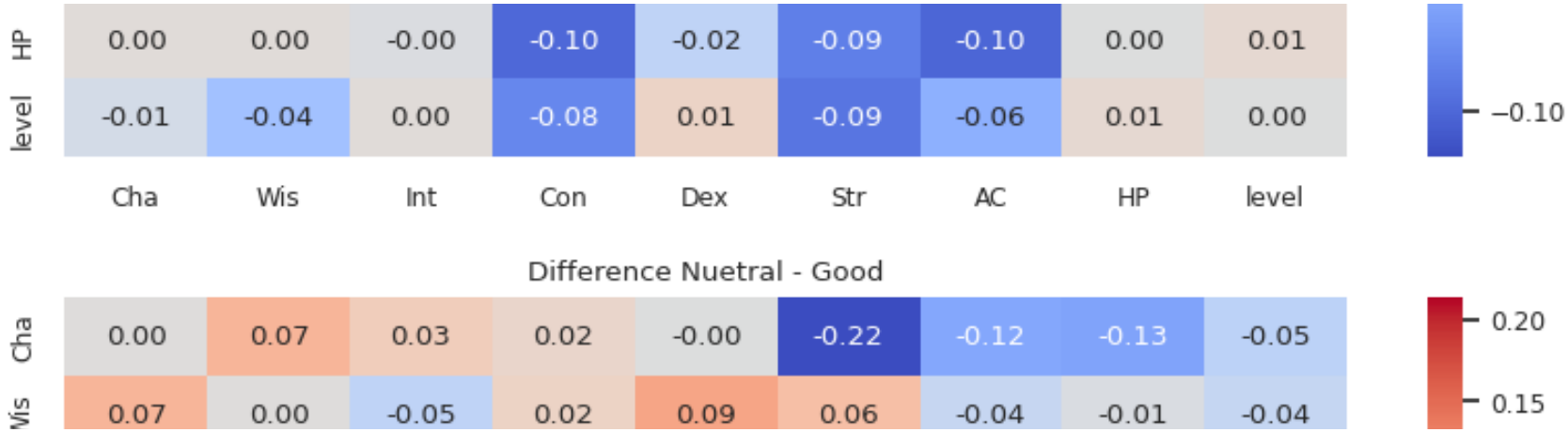
Our next analytical step involves constructing a difference matrix. This matrix will compare the correlations among all attributes across different alignments to uncover more nuanced patterns that may not be immediately apparent from individual distributions. The goal is to quantify and visualize the distinctions between alignments in terms of how their attributes correlate with one

another. By calculating and plotting the differences in correlation matrices for 'Evil,' 'Neutral,' and 'Good' alignments, we can identify specific attribute interactions that are unique to each alignment. This comparison will enhance our understanding of the interplay between character traits and alignment, providing a clearer picture of the underlying structure in the data.

The following code sets up this comparative analysis, which will be illustrated through a series of heatmaps, allowing us to observe and interpret these subtle but informative differences.

```
numerical_attributes = ['Cha', 'Wis', 'Int', 'Con', 'Dex', 'Str', 'AC', 'HP', 'level']

correlation_matrices = {}

alignment_name = {-1: 'Bad', 0: 'Nuetral', 1:'Good'}
for alignment_value in [-1, 0, 1]:
    alignment_df = filtered_df[filtered_df['intAlignment'] == alignment_value]
    correlation_matrix = alignment_df[numerical_attributes].corr()
    correlation_matrices[alignment_value] = correlation_matrix

# Create a single figure with subplots for the non-identical differences
fig, axes = plt.subplots(3, 1, figsize=(10, 15))

non_identical_pairs = [(-1, 1), (0, -1), (0, 1)]

for idx, (alignment_value1, alignment_value2) in enumerate(non_identical_pairs):
    difference_matrix = correlation_matrices[alignment_value1] - correlation_matrices[alignment_value2]

    sns.heatmap(difference_matrix, annot=True, cmap='coolwarm', fmt=".2f", ax=axes[idx])
    axes[idx].set_title(f'Difference {alignment_name[alignment_value1]} - {alignment_name[alignment_value2]}')

plt.show()
```


Difference Bad - Good
Cha 0.00 0.13 0.03 -0.09 0.09 -0.31 -0.18 -0.13 -0.04
Wis 0.13 0.00 0.05 -0.00 0.09 0.07 0.05 -0.01 0.00
Int 0.03 0.05 0.00 -0.16 -0.08 0.03 0.06 -0.05 -0.03
Con -0.09 -0.00 -0.16 0.00 0.11 0.00 0.05 0.12 0.06
Dex 0.09 0.09 -0.08 0.11 0.00 0.09 0.31 0.07 0.03
Str -0.31 0.07 0.03 0.00 0.09 0.00 -0.06 0.02 0.03
AC -0.18 0.05 0.06 0.05 0.31 -0.06 0.00 0.16 0.11
HP -0.13 -0.01 -0.05 0.12 0.07 0.02 0.16 0.00 -0.04
level -0.04 0.00 -0.03 0.06 0.03 0.03 0.11 -0.04 0.00
Cha Wis Int Con Dex Str AC HP level
0.3 0.2 0.1 0.0 −0.1 −0.2 −0.3
Difference Nuetral - Bad
Cha 0.00 -0.06 0.01 0.11 -0.09 0.09 0.06 0.00 -0.01
Wis -0.06 0.00 -0.10 0.02 0.00 -0.01 -0.09 0.00 -0.04
Int 0.01 -0.10 0.00 0.12 -0.00 -0.12 -0.02 -0.00 0.00
Con 0.11 0.02 0.12 0.00 0.04 -0.02 -0.03 -0.10 -0.08
Dex -0.09 0.00 -0.00 0.04 0.00 0.02 -0.09 -0.02 0.01
Str 0.09 -0.01 -0.12 -0.02 0.02 0.00 0.01 -0.09 -0.09
AC 0.06 -0.09 -0.02 -0.03 -0.09 0.01 0.00 -0.10 -0.06
0.10 0.05 0.00 −0.05



Upon reviewing the correlation differences between alignments, two pairs of attributes—Strength and Charisma, as well as Armor Class (AC) and Dexterity (Dex)—stand out, particularly when comparing 'Good' alignment against 'Neutral' and 'Evil'. These pairs exhibit noticeable differences in correlation, especially between 'Good' and the other two alignments, confirming our earlier suspicions regarding Strength and Charisma.

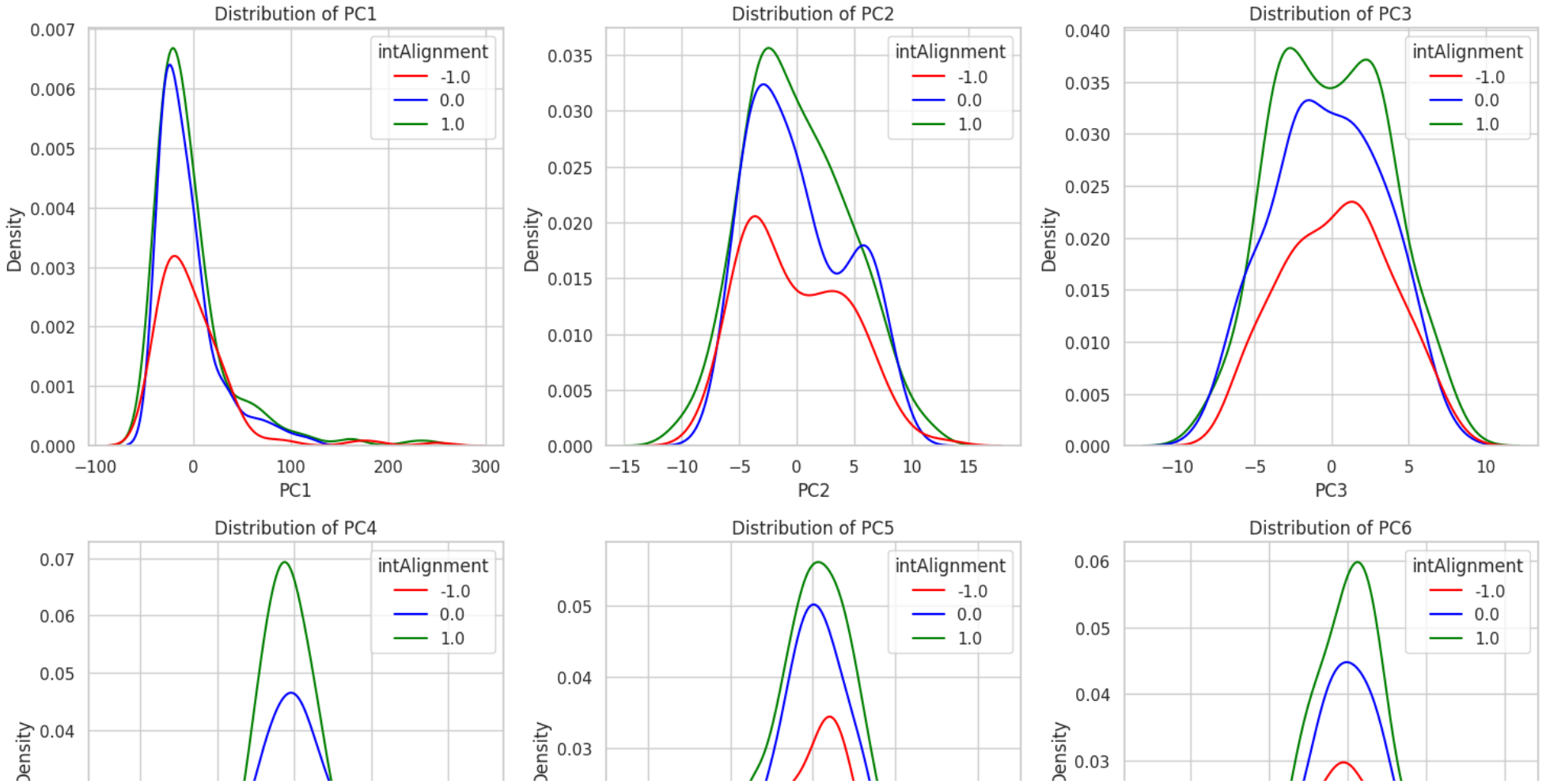
The heatmaps vividly display these differences, with pronounced colors indicating significant variance in covariance across alignments. Such distinct patterns in the data suggest that Principal Component Analysis (PCA) could effectively discern and enhance the separation between alignments based on numerical attributes.

By applying PCA, we aim to mitigate the previously identified issues: the presence of noise and the small frequency differences in the data. PCA achieves this by prioritizing the dimensions with the highest variance, thereby emphasizing the most impactful attributes. In doing so, we expect to diminish less informative data, focusing instead on the attributes with stronger covariance differences—namely Charisma, Strength, Dexterity, Armor Class, and Constitution.

To accommodate an additional degree of freedom and to capture the essence of the data, we elect to implement PCA with six components. This choice is based on the attributes that show the most pronounced covariance differences. Then we will illustrate the transformed data by plotting density graphs for each alignment post-PCA application, anticipating clearer distinctions and a more defined structure that could potentially enhance the predictive performance of a model trained on this data.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

numerical_columns = ['Cha', 'Wis', 'Int', 'Con', 'Dex', 'Str', 'AC', 'HP', 'level', 'intAlignment']

# Select the numerical columns from the DataFrame
data = filtered_df[numerical_columns]

# Separate the features and the target variable
X = data.drop('intAlignment', axis=1)
y = data['intAlignment']

# Apply PCA with n_components=6
pca = PCA(n_components=6)
principal_components = pca.fit_transform(X)

# Create a DataFrame with the principal components
pca_df = pd.DataFrame(data=principal_components, columns=['PC1', 'PC2', 'PC3', 'PC4', 'PC5', 'PC6'])

# Concatenate the PCA components and the target variable
result_df = pd.concat([pca_df, y], axis=1)

# Set the style for the plots
sns.set(style="whitegrid")

# Create a single figure with subplots
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# Loop through each principal component
for i, pc in enumerate(['PC1', 'PC2', 'PC3', 'PC4', 'PC5', 'PC6']):
    # Plot the distribution of the current principal component for each intAlignment value
    sns.kdeplot(data=result_df, x=pc, hue='intAlignment', palette={-1.0: 'red', 0.0: 'blue', 1.0: 'green'}, ax=axes[i//3

    # Set plot labels and title
    axes[i//3, i%3].set_title(f'Distribution of {pc}')
```

```
    axes[i//3, i%3].set_xlabel(pc)
    axes[i//3, i%3].set_ylabel('Density')

# Adjust layout
plt.tight_layout()
plt.show()
```



The emerging patterns indicate a promising advantage for our classifier's learning process in distinguishing between alignments more effectively.

The PCA-transformed features reveal that characters with 'evil' alignment display a broad dispersion across the newly created principal component axes. This wide distribution suggests a diverse set of characteristics within the 'Evil' alignment, offering a rich dataset from which the classifier can learn. There is also a discernible separation between 'good' and 'neutral' characters in this transformed feature space. The PCA has evidently simplified the complexity of the data while preserving critical information, setting the stage for a classifier that can better understand and act upon the nuances of each alignment.

Time Related Data Analysis

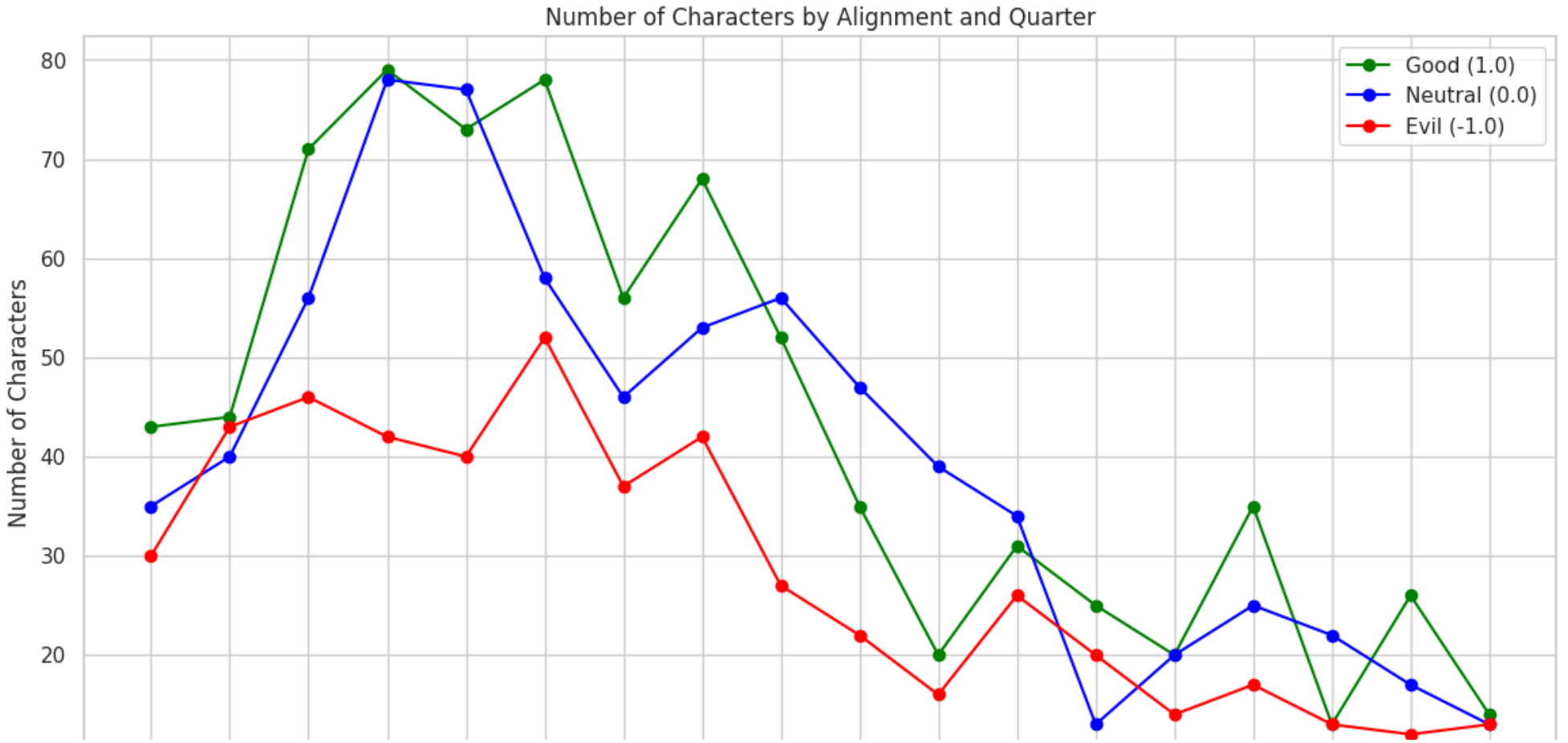
The dataset we are using was collected during a time of shifting trends in the D&D world. Traditional D&D lore cast Orcs, Goblins, Dark Elves and Tieflings in almost exclusively evil roles. The period during which these characters were recorded (2018-2022) saw some major shifts in those narratives. We will mention some of the ones that stand out:

- Critical Role: Critical Role is a popular web series that features a group of voice actors playing Dungeons & Dragons. The show revolutionized spectator D&D games in pop culture and challenged many traditional D&D tropes. In particular, Critical Role has featured several orc characters who are complex and sympathetic, such as Grog Strongjaw and Keyleth.
- The Adventure Zone: The Adventure Zone is a popular podcast that features three brothers and their father playing Dungeons & Dragons. The show has been praised for its humor and its ability to connect with a wide audience. In particular, The Adventure Zone has featured several goblin characters who are funny and endearing, such as Magnus Burnsides and Merle Highchurch.
- Dungeons & Dragons: Beyond: Dungeons & Dragons: Beyond is a digital platform that provides players with a variety of tools for playing D&D. In 2020, Beyond released a series of articles called "Beyond the Monstrous" that explored the diversity of D&D's monstrous races. The articles argued that these races should not be seen as inherently evil and that players should be encouraged to create more nuanced and complex characters.
- Wizards of The Coast themselves, the company behind D&D, reworked a number of the more brutish races, which may have been an attempt to paint them in a better light, for example:
    - Goblins: Goblins were given Fey Ancestry and changes to their Fury of the Small trait, making them more versatile and aligning them more closely with elves.
    - Hobgoblins: They were reworked to focus on team tactics and helping allies, a departure from their traditional portrayal as mere brute soldiers.
    - Kobolds: They lost the Grovel, Cower, and Beg trait and gained Draconic Cry, shifting from a subservient image to one that's more empowered and dynamic.

Considering these social trends, we decided to include date of character creation in our analysis. We expect to see a decreasing frequency of 'evil' characters over the four year period.

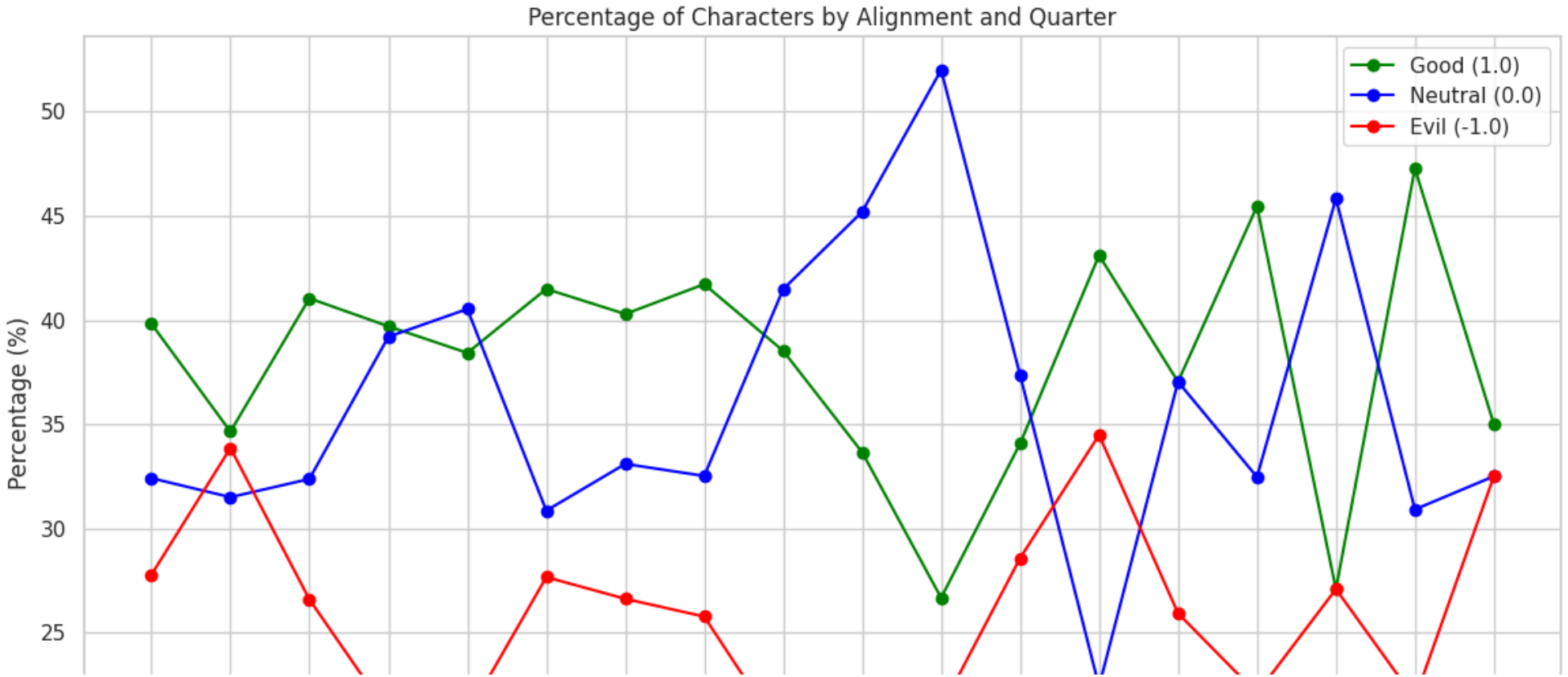
```
df_filtered = df[df['intAlignment'].notnull()]
print(df['intAlignment'].value_counts())

df_filtered['date'] = pd.to_datetime(df_filtered['date'])
df_filtered['quarter'] = df_filtered['date'].dt.to_period("Q")

alignment_by_quarter = df_filtered.groupby(['quarter', 'intAlignment']).size().reset_index(name='count')

plt.figure(figsize=(14, 7))
for alignment, label, color in zip([1.0, 0.0, -1.0], ['Good', 'Neutral', 'Evil'], ['green', 'blue', 'red']):
    subset = alignment_by_quarter[alignment_by_quarter['intAlignment'] == alignment]
    plt.plot(subset['quarter'].astype(str), subset['count'], label=f"{label} ({alignment})", marker='o', color=color)

plt.title('Number of Characters by Alignment and Quarter')
plt.xlabel('Quarter')
plt.ylabel('Number of Characters')
plt.legend()
plt.grid(True)
plt.show()
```

```
 1.0    783
 0.0    729
-1.0    512
Name: intAlignment, dtype: int64
<ipython-input-18-fcfcd7dd7ad2>:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returnin
  df_filtered['date'] = pd.to_datetime(df_filtered['date'])
<ipython-input-18-fcfcd7dd7ad2>:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returnin
  df_filtered['quarter'] = df_filtered['date'].dt.to_period("Q")
```



It's difficult to see the frequency of the different alignments because of the general downtrend. The app peaked in popularity in 2019 and declined in usage from that point on. To get a clearer picture, let's plot the percentage of characters created that were 'good', 'evil' and 'neutral' by quarter.

```
total_by_quarter = df_filtered.groupby('quarter').size().reset_index(name='total_count')
alignment_by_quarter = pd.merge(alignment_by_quarter, total_by_quarter, on='quarter')
alignment_by_quarter['percentage'] = (alignment_by_quarter['count'] / alignment_by_quarter['total_count']) * 100

plt.figure(figsize=(14, 7))
for alignment, label, color in zip([1.0, 0.0, -1.0], ['Good', 'Neutral', 'Evil'], ['green', 'blue', 'red']):
    subset = alignment_by_quarter[alignment_by_quarter['intAlignment'] == alignment]
    plt.plot(subset['quarter'].astype(str), subset['percentage'], label=f"{label} ({alignment})", marker='o', color=colo

plt.title('Percentage of Characters by Alignment and Quarter')
plt.xlabel('Quarter')
plt.ylabel('Percentage (%)')
plt.legend()
plt.grid(True)
plt.show()
```



Contrary to expectations, the data does not reveal any definitive trends in character alignment across the observed time period. This outcome suggests that our initial hypotheses regarding alignment trends may not align with the actual player behavior, or it may indicate that the dataset at hand is not sufficiently representative or lacks the volume to effectively capture and demonstrate such trends.

Country Code and Casting Stat

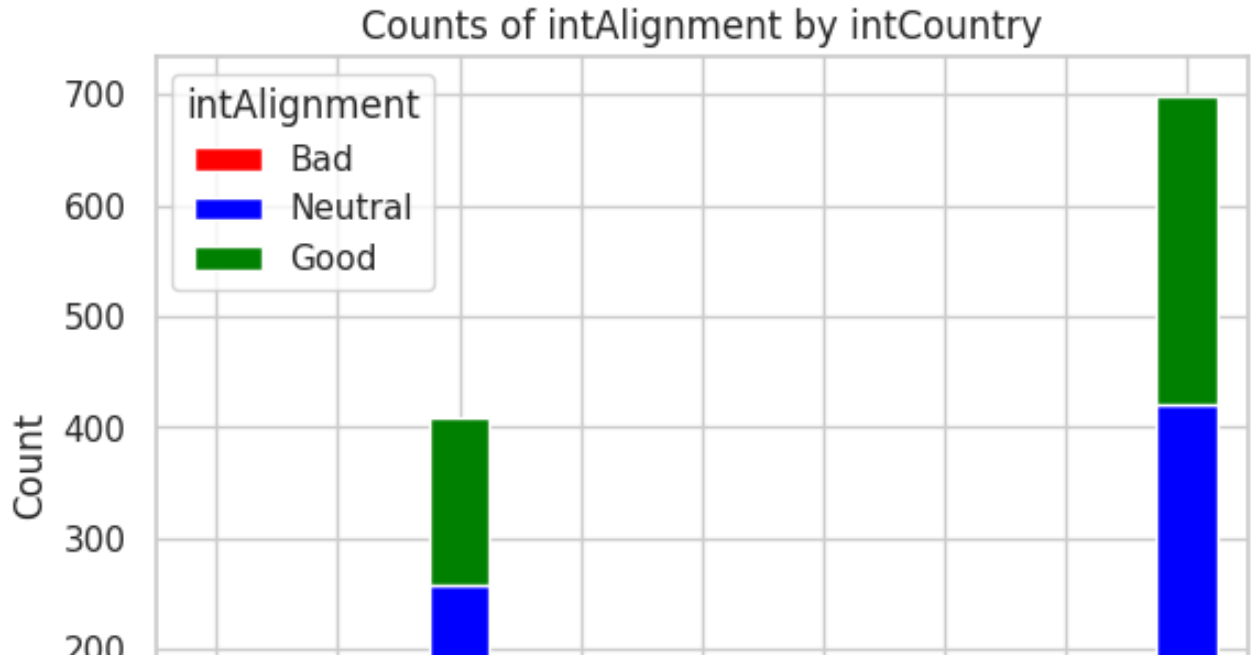
Let's start by counting the value counts of the two textual columns we are interested in

```
filtered_df = df[df['intAlignment'].notnull()]

print(filtered_df['countryCode'].value_counts())
print(filtered_df['castingStat'].value_counts())
```

```
US    699
CA    409
GB     61
BR     32
DE     25
AU     24
NL     18
IT     15
CR     11
TR     10
CL      9
ES      7
NZ      7
PH      7
NO      6
MX      5
SG      4
AT      4
JP      3
SE      3
BE      3
FR      2
CH      2
GR      2
HU      2
BH      1
PE      1
IN      1
IS      1
PT      1
HR      1
DO      1
VE      1
CY      1
```

```
BG      1
LT      1
PR      1
Name: countryCode, dtype: int64
Int    906
Cha    576
Wis    534
Str      4
Con      2
Dex      2
Name: castingStat, dtype: int64
```

```
country_counts = filtered_df.groupby(['countryCode', 'intAlignment']).size().unstack(fill_value=0)

# Filter out countries with less than 10 instances
country_counts = country_counts[country_counts.sum(axis=1) > 10]

colors = ['red', 'blue', 'green']

country_counts.plot(kind='bar', stacked=True, color=colors)

plt.title('Counts of intAlignment by intCountry')
plt.xlabel('Country')
plt.ylabel('Count')
plt.legend(['Bad', 'Neutral', 'Good'], title='intAlignment')

plt.show()
```



Here we have plotted the countries with significant representation, showing only those countries with more than ten instances. The overwhelming majority of characters come from the Unites states and Canada, and these countries reflect a general bias towards 'good' characters, with the United Stated showing a stronger trend. Surprisingly, almost none of the other countries show a tendency towards good characters. This might indicate a difference in societal norms between North American countries and other nationalities. Let's take a look at casting stat.

Country

```
counts = filtered_df.groupby(['castingStat', 'intAlignment']).size().unstack(fill_value=0)

colors = ['red', 'blue', 'green']

# Plotting
counts.plot(kind='bar', stacked=True, color=colors)

plt.title('Counts of intAlignment by castingStat')
plt.xlabel('castingStat')
plt.ylabel('Count')
plt.legend(['Bad', 'Neutral', 'Good'], title='intAlignment')

plt.show()
```

In D&D, a character's casting stat is the main stat it uses in determining success in casting spells. This field does not introduce much new, as a character's class maps directly to casting stat, but as there is only a single casting stat in the data even for multiclass characters, it could be a determination of *primary* class across multiclass. It does also represent a unique grouping of classes. It is hard to pull much conclusively from this plot, except to say that the bias towards good and neutral characters exists across the spectrum and is strongest in Wisdom casters. This is not surprising, as Clerics, Druids, some Monks and Rangers all use Wisdom as their casting stat.

Wisdom Casters as a KPI

```
# Print the counts DataFrame
print("Data being plotted:")
print(counts)
# Assuming counts is the DataFrame you obtained earlier

# Extract counts for 'Wis' casters
wis_counts = counts.loc['Wis']

# Calculate the ratios
good_to_evil_ratio = wis_counts[1.0] / wis_counts[-1.0]
good_to_neutral_ratio = wis_counts[1.0] / wis_counts[0.0]
neutral_to_evil_ratio = wis_counts[0.0] / wis_counts[-1.0]

# Print the results
print("Ratios for Wisdom (Wis) casters:")
print(f"Good to Evil Ratio: {good_to_evil_ratio:.2f}")
print(f"Good to Neutral Ratio: {good_to_neutral_ratio:.2f}")
print(f"Neutral to Evil Ratio: {neutral_to_evil_ratio:.2f}")
```

```
Data being plotted:
intAlignment  -1.0   0.0   1.0
castingStat
Cha            162   193   221
Con              0     1     1
Dex              0     1     1
Int            261   358   287
Str              0     0     4
Wis             89   176   269
Ratios for Wisdom (Wis) casters:
Good to Evil Ratio: 3.02
Good to Neutral Ratio: 1.53
Neutral to Evil Ratio: 1.98
```

In light of the above, we would like to see that our model favors good wisdom casters dispreportionately. Ideqally in similar ratios as the training data displays.

Our Predictive Model

First we need to prepare the data. This step consists of the following:

1. Seperate all the multiclass rows into seperate rows.
2. Calculate the 'intAlignment' like we did before.
3. Map all the textual data into numbers.
4. Standerdize the data.
5. Calculate the PCA values for the numerical data and drop the old numerical attributes because we now have the PCA ones.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

df = pd.read_csv('dnd_chars_unique.csv')
df = df[df['processedAlignment'].notnull()]
print(len(df.index))

new_rows = pd.DataFrame(columns=df.columns)

indices_to_drop = []

for index, row in df.iterrows():
    classes = row['justClass'].split('|')

    indices_to_drop.append(index)

    for class_name in classes:
        new_row = row.copy()
        new_row['justClass'] = class_name
        new_rows = new_rows.append(new_row, ignore_index=True)

df = df.drop(indices_to_drop)
df = pd.concat([df, new_rows], ignore_index=True)

df = df.reset_index()
print("Classes after dealing with multiclass:")
print(df['justClass'].unique())
print(len(df.index))
print("Step one has finished")

def new_alignment_symbol(a, b):
  addition = a + b
  if addition > 0:
    return 1
  if addition < 0:
    return -1
  return 0
```

```
string_to_int_mapping = {
    'CN': new_alignment_symbol(-1, 0),
    'CG': new_alignment_symbol(-1, 1),
    'NG': new_alignment_symbol(0, 1),
    'NN': new_alignment_symbol(0, 0),
    'LN': new_alignment_symbol(1, 0),
    'LG': new_alignment_symbol(1, 1),
    'LE': new_alignment_symbol(1, -1),
    'NE': new_alignment_symbol(0, -1),
    'CE': new_alignment_symbol(-1, -1)}

df['intAlignment'] = df['processedAlignment'].map(string_to_int_mapping)
print("Printing the value count for the new alignemnt")
print(df['intAlignment'].value_counts())
print("Step two has finished")

europe = ['GB', 'DE', 'SE', 'BG', 'IT', 'NL', 'LT', 'TR', 'AT', 'NO', 'CH', 'ES', 'GR', 'BE', 'HU', 'HR', 'CY', 'FR', 'P

def country_to_number(countryCode):
  if countryCode == 'US' or countryCode in europe:
    return 1

  return 2

df['intCountry'] = df['countryCode'].map(country_to_number)
print(df['intCountry'].value_counts())

unique_casting_stats = df['castingStat'].unique()
casting_stat_dict = {stat: idx for idx, stat in enumerate(unique_casting_stats)}
df['intCastingStat'] = df['castingStat'].map(casting_stat_dict)
print(casting_stat_dict)
print(df['intCastingStat'].value_counts())

unique_classes = df['justClass'].unique()
class_dict = {stat: idx for idx, stat in enumerate(unique_classes)}
df['intClass'] = df['justClass'].map(class_dict)
```

```
print(class_dict)
print(df['intClass'].value_counts())

unique_races = df['processedRace'].unique()
race_dict = {stat: idx for idx, stat in enumerate(unique_races)}
df['intRace'] = df['processedRace'].map(race_dict)
print(race_dict)
print(df['intRace'].value_counts())

print("Step three has finished")

columns_to_keep = ['intCastingStat', 'intCountry', 'intAlignment', 'intRace', 'intClass', 'Cha', 'Wis', 'Int', 'Con', 'D
df = df.loc[:, columns_to_keep]
print("Columns of the DataFrame:")
print(df.columns)

print("Step four has finished")

print("Lets check for missing values acrros all the dataframe")
any_missing_values = df.isna().any().any()
print(any_missing_values)

y = df['intAlignment'].to_frame()
df = df.drop('intAlignment', axis=1)

scaler = StandardScaler()
df_standardized = scaler.fit_transform(df)
df_standardized = pd.DataFrame(df_standardized, columns=df.columns)

print("Step five has finished")

from sklearn.decomposition import PCA

numerical_columns = ['Cha', 'Wis', 'Int', 'Con', 'Dex', 'Str', 'AC', 'HP', 'level']
data = df[numerical_columns]

pca = PCA(n_components=6)
```

```
principal_components = pca.fit_transform(data)
pca_df = pd.DataFrame(data=principal_components, columns=['PC1', 'PC2', 'PC3', 'PC4', 'PC5', 'PC6'])
df = pd.concat([df, pca_df], axis=1)

print("Columns before dropping the pre PCA integers")
print(df.columns)

df = df.drop(columns=['Cha', 'Wis', 'Int', 'Con', 'Dex', 'Str', 'AC', 'HP', 'level'])

print("Columns after dropping the pre PCA integers")
print(df.columns)
print("And the tagging column")
print(y.columns)

print("Step six has finished")
```

```
2024
<ipython-input-24-ac602cbafda8>:20: FutureWarning: The frame.append method is deprecated and will be removed from
  new_rows = new_rows.append(new_row, ignore_index=True)
Classes after dealing with multiclass:
['Druid' 'Fighter' 'Warlock' 'Cleric' 'Paladin' 'Barbarian' 'Sorcerer'
 'Monk' 'Ranger' 'Bard' 'Mystic' 'Wizard' 'Rogue' 'Blood Hunter'
 'Artificer' 'Revised Ranger']
2174
Step one has finished
Printing the value count for the new alignemnt
 1    817
 0    790
-1    567
Name: intAlignment, dtype: int64
Step two has finished
2    1229
1     945
Name: intCountry, dtype: int64
{'Wis': 0, 'Int': 1, 'Cha': 2, 'Str': 3, 'Con': 4, 'Dex': 5}
1    970
2    633
```

```
0    563
3      4
4      2
5      2
Name: intCastingStat, dtype: int64
{'Druid': 0, 'Fighter': 1, 'Warlock': 2, 'Cleric': 3, 'Paladin': 4, 'Barbarian': 5, 'Sorcerer': 6, 'Monk': 7, 'Ran
1     282
12    270
3     221
2     174
4     163
5     152
8     151
11    151
6     141
9     141
0     138
7     131
14     44
15     10
13      3
10      2
Name: intClass, dtype: int64
{'Aasimar': 0, 'Leonin': 1, 'Turtle': 2, 'Human': 3, 'Dragonborn': 4, 'Half-Orc': 5, 'Tabaxi': 6, nan: 7, 'Half-El
3     403
```