text_classification

April 3, 2023

1 Classifying Prompt Preformance

With the emergence of ChatGPT, I would like to try and base this initial expermiment in using ML models for classifying the success of ChatGPT prompts. A simple test, would be testing the "style" of prompt responses. Often when I client wants to create a chat bot, they would like it to talk in a certain way, or mimic a certain person.

in this example, I'll be using various ML tools (Naive Bayes, Logistic Regression, Neural Networks) to train a model to recogize Rick's lines from the show *Rich and Morty*

With a database that contains an annotated record of every line from the show, we may create a model that predicts a positive result if it thinks the line is spoken by Rick. In full disclosure, there probably won't be a high success rate in the final trained model, just due to the complexity of the data.

I'd be curious to see if I am wrong.

Data found at: https://www.kaggle.com/datasets/andradaolteanu/rickmorty-scripts?resource=download

1.1 Read in Data

This model is built in Google Colab, so to access the data, the data must be uploaded to the environment (To run this notebook, upload the csv downloaded from Kaggle to files/directory of this notebook). We are only interested in the names of characters and the lines they have, so we will read that in. I make sure to set the names as categorical as well.

I also make a wordcloud using a picture of Rick, which should also be loaded into this notebook's directory.

[352]:
```
import pandas as pd

# Random seed for reproducability:
seed = 1123

df = pd.read_csv('RickAndMortyScripts.csv',
                 header=0,
                 usecols=[4,5],
                 dtype={'name':'category', 'line': 'str'})
print('rows and columns:', df.shape)
df
```

```
rows and columns: (1905, 2)
```

[352]:
```
      name                                                line
0     Rick  Morty! You gotta come on. Jus'… you gotta co…
1    Morty                        What, Rick? What's going on?
2     Rick                   I got a surprise for you, Morty.
3    Morty  It's the middle of the night. What are you tal…
4     Rick  Come on, I got a surprise for you.  Come on, h…
…     …                                                   …
1900 Morty                                  That was amazing!
1901  Rick                      Got some of that mermaid puss!
1902 Morty  I'm really hoping it wasn't a one-off thing an…
1903  Rick  Pssh! Not at all, Morty. That place will never…
1904 Morty                      Whoo! Yeah! Yeaah! Ohhh, shit!

[1905 rows x 2 columns]
```

[353]:
```
df['name'].head()
```

[353]:
```
0     Rick
1    Morty
2     Rick
3    Morty
4     Rick
Name: name, dtype: category
Categories (48, object): ['Agency Director', 'Alan', 'Alien Doctor', 'All
Mortys', …,
                          'Testicle Monster A', 'Vance', 'Vet', 'Young Rick']
```

Seems good! Just to make sure we don't have nulls

[327]:
```
df.isnull().sum()
```

[327]:
```
name    0
line    0
dtype: int64
```

[354]:
```
df['name'].dtype
```

[354]:
```
CategoricalDtype(categories=['Agency Director', 'Alan', 'Alien Doctor', 'All
Mortys',
                  'All Ricks', 'All Summers', 'Announcer', 'Beth',
                  'Birdperson', 'Brad', 'Campaign Manager Morty',
                  'Candidate Morty', 'Cop Morty', 'Cop Rick',
                  'Cornvelious Daniel', 'Cromulon', 'Dr. Wong', 'Drunk Rick',
                  'Glasses Morty', 'Ice-T', 'Jerry', 'Jessica', 'Lizard Morty',
                  'Million Ants', 'Morty', 'Morty 1', 'Morty 2',
                  'Mr. Goldenfold', 'Mrs. Pancakes', 'Narrator', 'Nathan',
```

```
                    'Pickle Rick', 'President', 'Principal Vagina', 'Rick',
                    'Rick J-22', 'Riq IV', 'Scary Terry', 'Slick', 'Snuffles',
                    'Summer', 'Summer 1', 'Supernova', 'Teacher Rick',
                    'Testicle Monster A', 'Vance', 'Vet', 'Young Rick'],
, ordered=False)
```

1.2 Train and Test

Now we want to create individual subsets of the data that we can use for training and testing our models. The split should use a classic 80-20 split with a random shuffle. However, we may still get a biased dataset.

1.2.1 Even Distribution

An ideal dataset has an even distribution of the target class when compared to the rest of the dataset. In our case, we want ~50% of the database to be 'Rick' and ~50% of the database to be Not Rick.

So, we'll modify the data, so it only has a catgory for Ricks, and non Ricks.

[355]:
```
def change_name(row):
  if 'rick' in row['name'].lower():
    return 'Rick'
  else:
    return 'Others'

# Note that the output is still a category!
df['name'] = df.apply(change_name, axis=1).astype('category')
df.head()
```

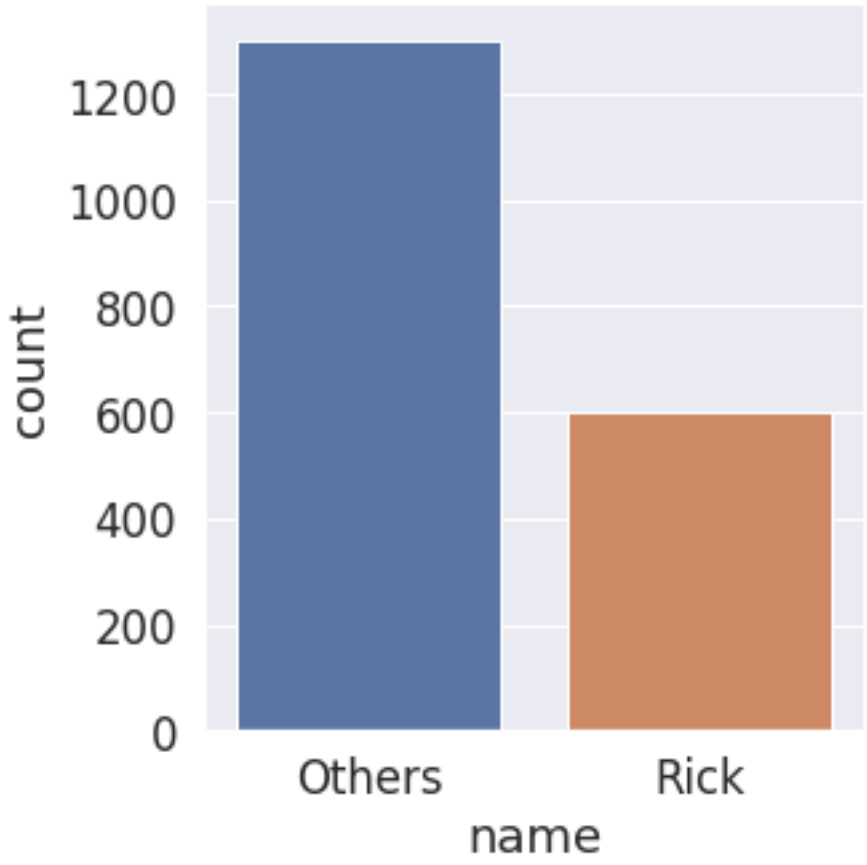
[355]:

	name	line
0	Rick	Morty! You gotta come on. Jus'… you gotta co…
1	Others	What, Rick? What's going on?
2	Rick	I got a surprise for you, Morty.
3	Others	It's the middle of the night. What are you tal…
4	Rick	Come on, I got a surprise for you. Come on, h…

[356]:
```
import seaborn as sb

sb.catplot(data=df, x='name', kind='count')
```

[356]:
```
<seaborn.axisgrid.FacetGrid at 0x7fc6f30035e0>
```



We would prefer to have the distribution above be 50%. Before we split into train and test sets, we should create a dataset that contains all of Ricks lines, and the same amount of lines that aren't Ricks.

Sorting through the names data at this point benefits us from using categorical data instead of strings. When we check the value of a given row's categorical attribute 'name', we are actually using a hash table look up vs just searching through the whole database. Meaning any operations we do involving this column will have a search time of O(1) time instead of O(N).

To create this even dataset, we just have to remove a random sample from the rows of the dataset that are not Rick.

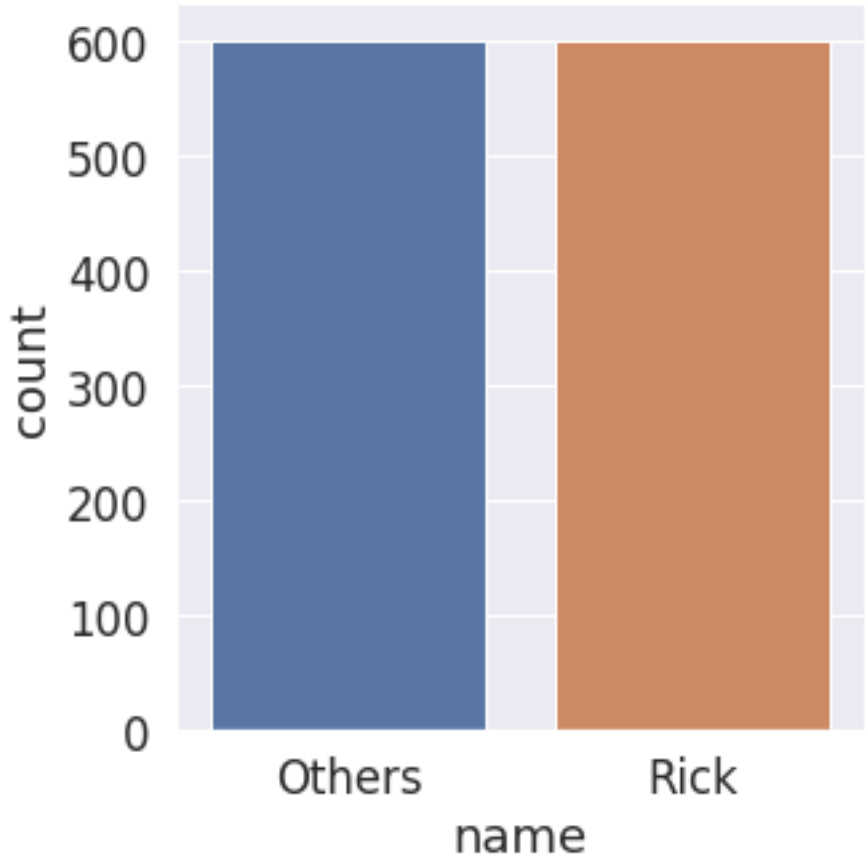
[357]:
```
num_rick = df['name'].value_counts()['Rick']
print("Number of Rick lines: %d" % (num_rick))
num_not_rick = df['name'].value_counts()['Others']
print("Number of Other lines: %d" % (num_not_rick))
diff = num_not_rick-num_rick
print("Number of lines to remove: %d" % (diff))

df_subset = df[df['name']=='Others'].sample(diff, random_state=seed)
```

```
even_df = df.drop(df_subset.index)
sb.catplot(data=even_df, x='name', kind='count')
```

```
Number of Rick lines: 602
Number of Other lines: 1303
Number of lines to remove: 701
```

[357]: <seaborn.axisgrid.FacetGrid at 0x7fc6f2b941c0>



Just to see what our data looks like:

[358]:
```
from wordcloud import WordCloud, STOPWORDS, ImageColorGenerator
from matplotlib import pyplot as plt
from PIL import Image
import numpy as np

rick_coloring = np.array(Image.open("Rick_Sanchez-0.png"))

wordcloud = WordCloud(background_color='white', stopwords = STOPWORDS,
                max_words = 2000, max_font_size = 100,
```

```
                random_state = seed,
                mask=rick_coloring)

plt.figure(figsize=(16, 12))
wordcloud.generate(''.join(even_df.loc[even_df['name'] == 'Rick', 'line']))

image_colors = ImageColorGenerator(rick_coloring)

# Show Image
plt.axis("off")
plt.imshow(wordcloud.recolor(color_func=image_colors), interpolation="bilinear")
```

[358]: <matplotlib.image.AxesImage at 0x7fc6f2bbbdc0>


Oh
Yeah
thing
know
gonna
people
think
want
need
around
let
mean
Rick
got
go
Well
Come
never
go
take
shit
us
Jerry
good
guy
right
something
time
really
Look
die
anything
Morty
seeds
now
make
one
Uh
place
whole
see
Alright

1.2.2 Dividing the Data

Now we can use sklearn's simple train-test split function to divide our data from our well distributed dataset.**bold text**

[359]:
```
from sklearn.model_selection import train_test_split

X = even_df['line']
y = even_df['name']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=seed)
```

1.3 Naive Bayes

Now that the data is split, we can try and classify our target using Naive Bayes. This is a probabilistic model that first calculates the prior probability of a class label. In this case, the probability that a label is Rick is about 50%. Then, it calculates the likelyhood of a label given a particular combination of feature values.

Sklearn comes with MultinomialNB, a naive bayes class that assumes feature values are multinomially distributed, or represent counts or frequencies of discrete events like word occurrences. In ourcase this works, as long as we can process our text to word counts.

Scikit-learn also has feature extraction tools for this purpose, so we can either extract the count of all word in document for each line, or the TFIDF (rarity in the overall corpus compared to frequency in a line).

[406]:
```
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
import nltk
from sklearn.pipeline import Pipeline

nltk.download('stopwords')

pipe1_1 = Pipeline([
        ('tfidf', CountVectorizer(binary=True)),
        ('nb', MultinomialNB()),
])

pipe1_2 = Pipeline([
        ('tfidf', TfidfVectorizer(binary=True)),
        ('nb', MultinomialNB()),
])
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

[407]:
```
pipe1_1.fit(X_train, y_train)
pipe1_2.fit(X_train, y_train)
```

[407]:
```
Pipeline(steps=[('tfidf', TfidfVectorizer(binary=True)),
                ('nb', MultinomialNB())])
```

[410]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score, confusion_matrix, classification_report
pred1_1 = pipe1_1.predict(X_test)
print(classification_report(y_test, pred1_1))
pred1_2 = pipe1_2.predict(X_test)
print(classification_report(y_test, pred1_2))
```

```
              precision    recall  f1-score   support

      Others       0.66      0.57      0.61       110
        Rick       0.68      0.76      0.71       131

    accuracy                           0.67       241
   macro avg       0.67      0.66      0.66       241
weighted avg       0.67      0.67      0.67       241

              precision    recall  f1-score   support

      Others       0.64      0.63      0.63       110
        Rick       0.69      0.70      0.70       131

    accuracy                           0.67       241
   macro avg       0.67      0.66      0.66       241
weighted avg       0.67      0.67      0.67       241
```

Naive Bayes is actually well suited to this complexity and size of data, but there is a flaw in that our individual lines are quite short. Notice that the first prediction using just counts has a higher recall then the model using tfidfs. This tells me that factoring word rarity isn't helping like I would hope it would. There probably isn't long enough lines from rick to observe unique word usage (compared to the rest of the lines)

Other characters might just reuse rare words more often within a single line, since they might have gimmicks where they repeat words more than Rick would. With lines this short.

1.4 Logistic Regression

The logistic regression model works by modeling the probability of the binary outcome using a logistic function, which maps the input features to a probability value between 0 and 1. Specifically,

the logistic function takes a linear combination of the input features and applies a sigmoid function to the result, which transforms the output to a probability value.

During training, the model adjusts the parameters of the logistic function to maximize the likelihood of the observed data, using a technique called maximum likelihood estimation. This involves minimizing a loss function that measures the discrepancy between the predicted probabilities and the actual outcomes.

We once again convert the lines to numerical data using a vectorizer. This difference between TfidVectorizer and CountVectorizer is minimal (this was tested), so we just show TfidVectorizer below.

[421]:
```
from sklearn.linear_model import LogisticRegression

pipe2 = Pipeline([
    ('tfidf', TfidfVectorizer(binary=True)),
    ('logreg', LogisticRegression(solver='lbfgs', class_weight='balanced')),
])

pipe2.fit(X_train, y_train)
```

[421]:
```
Pipeline(steps=[('tfidf', TfidfVectorizer(binary=True)),
                ('logreg', LogisticRegression(class_weight='balanced'))])
```

[422]:
```
pred2 = pipe2.predict(X_test)
print(classification_report(y_test, pred2))
```

```
              precision    recall  f1-score   support

      Others       0.63      0.70      0.66       110
        Rick       0.72      0.65      0.68       131

    accuracy                           0.67       241
   macro avg       0.67      0.67      0.67       241
weighted avg       0.68      0.67      0.67       241
```

The similar accuraccy is showing me that this dataset is way too small! Considering how the characters say similar phrases, the lack of a "naive assumption" that the word counts are independent really didn't increase accuracy. Testing with higher ngrams then unigrams decreased accuracy. Another sign that the data is a bit too small to reveal phrasing trends.

1.5 Neural Networks

A neural network is a computational model inspired by the structure and function of the human brain. It consists of interconnected layers of artificial neurons that can learn to map input data to output predictions through a process called training. This very generalized approach to machine learning works better on larger data sets, but I don't expect to see much variation in accuracy given the tuning of hidden layers and such.

[441]:
```
from sklearn.neural_network import MLPClassifier

pipe3 = Pipeline([
    ('tfidf', TfidfVectorizer(binary=True)),
    ('neuralnet', MLPClassifier(solver='lbfgs', alpha=1e-5,
                    hidden_layer_sizes=(15, 7), random_state=seed,␣
  ↪max_iter=1000)),
])

pipe3.fit(X_train, y_train)
```

[441]:
```
Pipeline(steps=[('tfidf', TfidfVectorizer(binary=True)),
                ('neuralnet',
                 MLPClassifier(alpha=1e-05, hidden_layer_sizes=(15, 7),
                               max_iter=1000, random_state=1123,
                               solver='lbfgs'))])
```

[440]:
```
pred3 = pipe3.predict(X_test)
print(classification_report(y_test, pred3))
```

```
              precision    recall  f1-score   support

      Others       0.64      0.68      0.66       110
        Rick       0.72      0.67      0.69       131

    accuracy                           0.68       241
   macro avg       0.68      0.68      0.68       241
weighted avg       0.68      0.68      0.68       241
```

As expected, after picking an optimum number of layers and nodes, we approached the same accuracy of the previous models, within a small margin. There were some configurations of layer counts that yeiled quite bad results but that is besides the point.

1.6 Testing

The goal of testing these 3 tiny models was to see what models produce the best accuracy for deciding which model had the best accuracy. Yet our 3 results above yielded the same accuracy. That would be partly because of the extremely small test size. But also, the actual writing of Rick and Morty is all rather similar!

The characters are all written with different personalities of course. But when presented to identify such a complex relationship as how a certain character speaks within this writing style, these models simply don't work that well.

To do a little bit of real world testing, lets test the different models on text that as much more varied from the data. I asked ChatGPT: > I've created a bot that tests if a line from the show "Rick and Morty" is coming from Rick. Can you generate a 10 long list of lines (python) that look like Rick talking from Rick and Morty. And then a 10 long list of lines that look like Bob Ross

quotes?

I then stored the results. Now we can test see if the model gets close to deciding how well chatGPT did.

[443]:
```
type(X_test)
```

[443]:
```
pandas.core.series.Series
```

[444]:
```
rick_lines = [    "Listen, Morty, I hate to break it to you, but what people␣
 ↪call love is just a chemical reaction that compels animals to breed.",   ␣
 ↪"Wubba lubba dub dub!",    "Morty, I've got a very important mission for us.␣
 ↪Get your shit together.",    "I don't like this place, Morty. I don't think␣
 ↪we should be here.",    "I'm not a superhero, Morty. I'm just some guy who's␣
 ↪trying to do some good.",    "It's not a place for smart people, Morty. A␣
 ↪lot of people get together and they start saying dumb stuff, and then␣
 ↪everybody else starts agreeing with it.",    "Morty, get in the car. We're␣
 ↪going on an adventure.",    "I'm not arguing that I'm not an asshole, Morty.␣
 ↪I'm just saying that if there's a grand plan, then I don't believe that I'm␣
 ↪part of it.",    "Science isn't about why, Morty. It's about why not.",   ␣
 ↪"Morty, the point of being a scientist is that you have to do things that␣
 ↪are unethical."]
bob_ross_lines = [    "We don't make mistakes, just happy little accidents.",  ␣
 ↪ "We don't have mistakes here, we just have happy accidents.",    "There's␣
 ↪nothing wrong with having a tree as a friend.",    "I think there's an␣
 ↪artist hidden at the bottom of every single one of us.",    "Talent is a␣
 ↪pursued interest. In other words, anything that you're willing to practice,␣
 ↪you can do.",    "You have to allow the paint to break to make it beautiful.
 ↪",    "There's nothing in the world that breeds success like success.",   ␣
 ↪"Every day is a good day when you paint.",    "Look around. Look at what we␣
 ↪have. Beauty is everywhere-you only have to look to see it.",    "All you␣
 ↪need to paint is a few tools, a little instruction, and a vision in your␣
 ↪mind."]
rick_lines = pd.Series(rick_lines)
bob_ross_lines = pd.Series(bob_ross_lines)
```

[447]:
```
pred_NB_rick = pipe1_1.predict(rick_lines)
print(pred_NB_rick)
pred_NB_bob = pipe1_1.predict(bob_ross_lines)
print(pred_NB_bob)
```

```
['Rick' 'Others' 'Rick' 'Rick' 'Rick' 'Rick' 'Rick' 'Others' 'Rick' 'Rick']
['Rick' 'Rick' 'Rick' 'Others' 'Rick' 'Rick' 'Rick' 'Rick' 'Rick' 'Rick']
```

[448]:
```
pred_LR_rick = pipe2.predict(rick_lines)
print(pred_LR_rick)
pred_LR_bob = pipe2.predict(bob_ross_lines)
print(pred_LR_bob)
```

```
['Rick' 'Others' 'Rick' 'Rick' 'Rick' 'Rick' 'Rick' 'Rick' 'Rick' 'Rick']
['Others' 'Rick' 'Rick' 'Others' 'Rick' 'Others' 'Rick' 'Rick' 'Others'
 'Rick']
```

[449]:
```
pred_NN_rick = pipe3.predict(rick_lines)
print(pred_NN_rick)
pred_NN_bob = pipe3.predict(bob_ross_lines)
print(pred_NN_bob)
```

```
['Rick' 'Others' 'Rick' 'Rick' 'Rick' 'Rick' 'Rick' 'Others' 'Rick' 'Rick']
['Others' 'Others' 'Rick' 'Rick' 'Others' 'Others' 'Rick' 'Rick' 'Others'
 'Rick']
```

Well these results aren't too good! It has predicted that Bob Ross talks like Rick…. Or more accurately after some thought is has predicted that Bob Ross talks more like Rick then any other character in Rick and Morty.

If you look at the results the Neural Network and Logistic Regression, they actually did do a bit better than Naive Bayes, which is a clear symptom of how Naive Bayes overfits data.

1.7 Analysis

1.7.1 Changes I'd Make

Considering I really enjoy this concept there are somethings I would like to do differently: - Generate a dataset that is much larger, using text that is completely randomly chosen or generated for the "not rick" category - Get a dataset that contains more Rick quotes, spanning the whole of the show and other content - I would like to learn how to weigh more prevelant quotes from a character in a fandom. This could be done by checking what kind of language makes it into quote compilations.

1.7.2 Generalization of Models

Overall I wasn't able to compare the pros and cons of the various approaches. But still would like to analyze the different models

- *Naive Bayes*: This was the best model for this data, as it is able to handle a small dataset with complex features. Yet in the end this dataset was a bit too small. NB isn't that good at dealing with data that didn't appear in the training data, and I needed a much more general approach. If I created a data set that was much more broad in it's scope, with a larger vocabulary this model could do better
- *Logistic Regression*: While the underlying assumption at play with this algorithm wasn't really on display here, there was direct comparison to Naive Bayes here. LR has a lower bias, so even though Naive Bayes is expected to perform better on small datasets, in a general test the Logistic Regression preformed better. -*Neural Net*: The observation from above still works. In the small dataset the model was given, the accuracy wasn't great. Yet, in a general case, it did perform better than the other models

While I didn't get to observe the performance of different models in contrast to each other, it's a bit clear how I might prefer to underfit my models for usecases where I am trying to be accurate on such a small subset of the data.

1.7.3 Balanced vs Imbalanced Data

I unfortunately don't have time to see if I was wrong to so heavily balance my starting dataset. In my experience, since I wanted an accurate global prediction, but increased my chances of being correct in the rare chance text really did sound like Rick. A better, more general model might just have the original distribution of the data.

1.7.4 Conclusion

There is a lot of good work to be done with this project, and a fun approach to validating AI results. I've learned over the course of this week I would like to get specifically into AI Alignment research, and this is really where it starts.