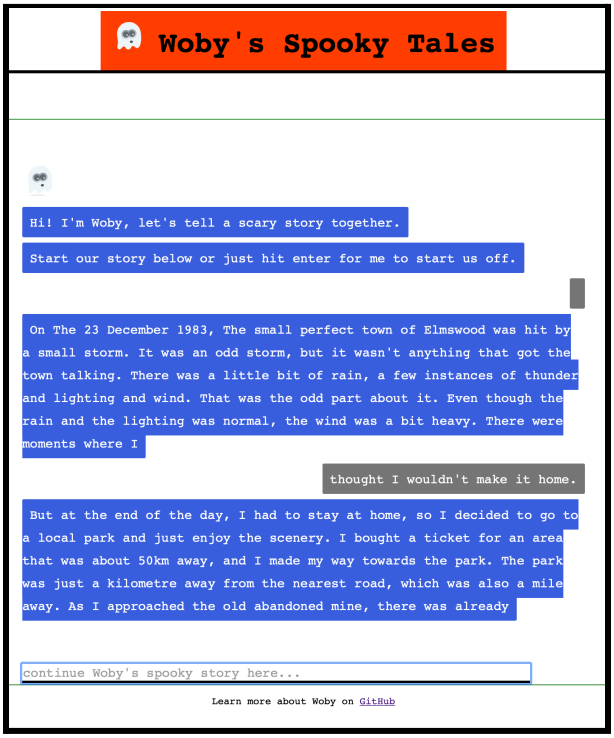
Woby’s Spooky Tales:

Scary Story Generation and Chatbot



George Washington University
Natural Language Processing - DATS 6312
Group 4
Joshua Ting

Table of Contents

1. Introduction

State of the art natural language models have made great strides in recent years due to new transformer architectures and ever increasing model parameter size. Recent language models now have the ability to produce text in various genres and domains where humans are not aware they are computer generated.

Horror stories have been an integral part of humanity's outlet to explore and bring to life our collective deepest fears and imagination. A well written horror story will not only need to understand the semantics and structure of a particular language but also the different nuanced elements that elicit a response from our primal fight or flight instincts.

In this paper, we explore several methodologies to perform the downstream task of story generation. Specifically, we attempt to create a language model that can generate coherent horror stories for readers who enjoy a good scare. We also built a chatbot front end for users to interface with the model; we will call this bot *Woby*. Additionally in this paper, we outline the methodologies for:

- Data acquisition and storage
- Data preprocessing and EDA
- Model selection, training, and evaluation
- Front end chatbot interface

2. Background

2.1 State of the Art Natural Language Models

Current state of the art natural language models use attention based mechanisms in a network architecture called transformers[1]. Although not strictly restricted to natural language tasks, transformer based architectures have performed greatly on various language tasks. Transformers have surpassed prior state of the art architectures in RNN based models such as LSTMs and GRUs[2].

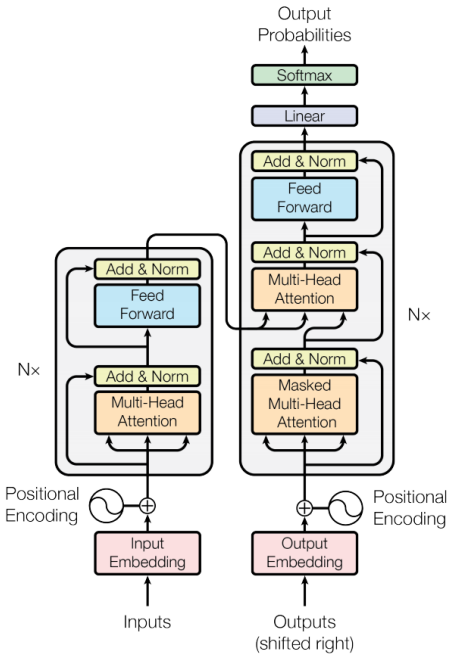


Figure 1: The Transformer - model architecture.[1]

In Figure 1 of a transformer architecture, we observe that it has an encoder-decoder architecture. The encoder encodes the input into a learned latent representation while the decoder takes the latent representation and generates an output. The novel component being the multi-headed and masked multi-headed attention layers as shown in Figure 2. We can think of attention as weight vectors that define which components of the entire source input are most important, highest weighted, for that particular sentence input.

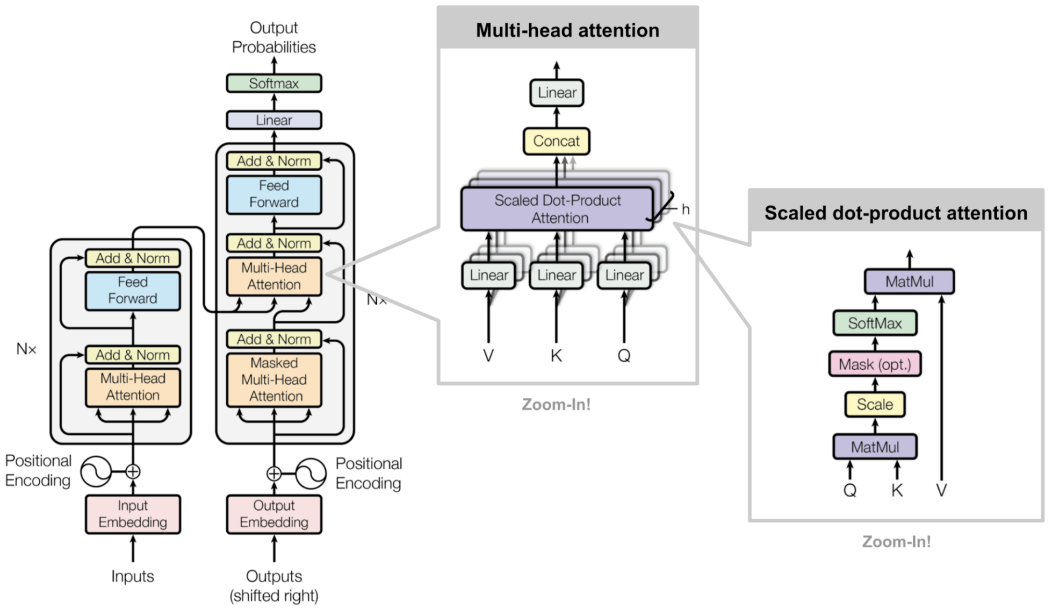


Figure 2: Multi-head attention.[1]

$$Attention(Q, K, V) = softmax(QK^T/\sqrt{d_k})V \qquad (1)$$

Equation (1) shows mathematically how attention can be calculated where Q is query, K is keys, and V is values. Q,K,V are word vectors where Q and K are compared to get weights that represent the relevance between each Q and K and V is then reweighted with these weights.

In 2018, Google's BERT[3] and OpenAI's GPT[4] saw the spark of building ever larger transformer networks with more model parameters. These large models are often not trainable without access to large numbers of high end GPU machines and at high cost. For the most part, users would use transfer learning on pretrained models and finetune them based on a particular downstream task.

2.2 Corpus

Our corpus is sourced from a number of SubReddits meant for various categories of horror writing and posts. A SubReddit is an online forum where anyone can create a post and users can upvote or downvote that particular post. Each SubReddit would also have its own niche and set of rules. We sourced the following 18 SubReddits with variations of horror themed posts.

Number	SubReddit Name	Link
1	r/nosleep	Link
2	r/stayawake	Link

3	r/DarkTales	Link
4	r/LetsNotMeet	Link
5	r/shortscarystories	Link
6	r/Thetruthishere	Link
7	r/creepyencounters	Link
8	r/truescarystories	Link
9	r/Glitch_in_the_Matrix	Link
10	r/Paranormal	Link
11	r/Ghoststories	Link
12	r/libraryofshadows	Link
13	r/UnresolvedMysteries	Link
14	r/TheChills	Link
15	r/scaredshitless	Link
16	r/scaryshortstories	Link
17	r/Humanoidencounters	Link
18	r/DispatchingStories	Link

Figure 3: Corpus SubReddit Data Sources.

Each post can vary in content, tone, length, and writing style as they are from different users from the internet which makes this a particularly good source of data to train our models with.

2.3 Prior Similar Works

In 2017, a group of people leveraged deep learning to create Shelley, a Twitter bot to complete scary stories from Twitter users. In their own words,

"*Shelley is a deep-learning powered AI who was raised reading eerie stories coming from r/nosleep. Now, as an adult—and not unlike Mary Shelley, her Victorian idol—she takes a bit of inspiration in the form of a random seed, or a short snippet of text, and starts creating stories emanating from her creepy creative mind. But what Shelley truly enjoys is working collaboratively with humans, learning from their nightmarish ideas, creating the best scary tales ever. If you want to work with her, respond to the stories she'll start every hour on her Twitter account, and she will write with you the first AI-human horror anthology ever put together!*"[5]

The actual architecture of Shelley was never released to the public but since this was developed prior to 2017, it is likely some sort of RNN architecture. Our group intends to add contribution to this prior work by utilizing current state of the art transformer models and with a much larger corpus.

2.4 Perplexity

Perplexity is our key intrinsic method to evaluate our models' progression, especially during training. Perplexity represents the normalized inverse probability of the test set. If a model assigns a high probability to a particular test set, it would mean that the model is not surprised (perplexed) that the test set occurs. This would mean that the model has a good understanding of how that language and corpus works. We then normalize it by the total number of words to get a comparable metric.

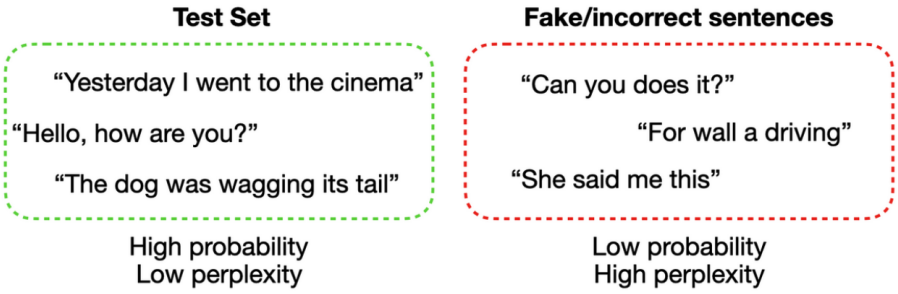


Figure 4: Perplexity example.[12]

Mathematically, perplexity is often denoted by equation (2) below where *P(wN)* is the probability of the particular test set *N* and the entire test set from 1 to N is normalized by taking the Nth root of the inverse.

$$PP(W) = \sqrt[N]{\frac{1}{P(w1,w2,\ldots,wN)}} \qquad (2)$$

Additionally, perplexity can also be calculated from the cross entropy loss by equation (3) below where the perplexity of a model M is e to the cross entropy between model M for the true language and L for the generated data.

$$P(M) = e^{H(L,M)} \qquad (3)$$

3. Experimental Setup - Corpus

3.1 Data Acquisition

To start, we first sought to download each available post from each of the 18 SubReddits shown in Figure 3. This was performed using the *Python Reddit API Wrapper* (PRAW)[6] and *Python Pushshift.io API Wrapper* (PSAW)[7]. PRAW is the official Reddit API Wrapper but runs into limitations of only 1000 posts being available per SubReddit. PSAW is a community open-sourced API wrapper that collects more posts than PRAW but can run into issues where certain hosting shards are not online. We used both in order to have the widest coverage then removed any duplicates afterwards.

Each post is saved first into a MongoDB database with the following schema in order to capture metadata about each post.

```
data_dict = {
            'doc_id': ,
            'full_name': ,
            'subreddit': ,
            'subreddit_name_prefixed': ,
            'title': ,
            'little_taste': ,
            'selftext': ,
            'author': ,
            'upvote_ratio': ,
            'ups': ,
            'downs': ,
            'score': ,
            'num_comments': ,
            'permalink': ,
            'kind': ,
            'num_characters': ,
            'num_bytes': ,
            'created_utc': ,
            'created_human_readable': ,
            'filepath': ,
            'train_test':
      }
```

Figure 5: Sample MongoDB schema.

Additionally, the raw post text is saved to disk in .txt files in the following structure.

```
FINAL-PROJECT-GROUP4
|── Code
|── Woby_Log
|── ...
|
└── Corpus
|    |
|    └──nosleep
|    |    | 1_t3_diuucz.txt
|    |    | 2_t3_dyqd5e.txt
|    |    | ...
|    |
|    └──creepyencounters
|    |    | 5931_t3_i3l009.txt
|    |    | 5931_t3_i3l009.txt
|    |    | ...
|    |
|    └──Ghoststories
|    |    | 9845_t3_jdedeb.txt
|    |    | 9846_t3_hvu2ko.txt
|    |    | ...
```

Figure 6: Sample corpus directory structure.

The metadata and individual .txt files are then packaged and pushed into our Kaggle account as a dataset for ease of distribution.

Figure 7 below shows a snippet of the relevant scripts and custom Python modules that were used for this section.

Python File	Description
`MongoDBInterface.py`	`Object to handle MongoDB client connection and CRUD tasks.`
`RedditAPI.py`	`Object to handle scrapping of Reddit posts from various sub-reddits.`
`scrape_reddit.py`	`Script to scrape Reddit to get corpus data and then save to disk/MongoDB`

`post_scrape_reddit.py`	`Script to run post scrape_reddit.py to:`
	`1. Remove duplicated stories`
	`2. Create .csv of metadata to save in /corpus`

`KaggleAPI.py`	`Object to handle connection to Kaggle API to upload and download files.`
`kaggle_dataset_push.py`	`Script to upload Kaggle datasets`

Figure 7: Scripts used for data acquisition.

Our final corpus dataset comes out to around 91MB and represents 14,715 different posts. We also randomly split the dataset into train, validation, and testing at the 84%/8%/8% splits respectively. This split is stored within the corpus' metadata.

3.2 Data Preprocessing

Next, we re-download our data from Kaggle into a new directory **Final-Project-Group4/corpus_data/** and treat that as our data source in order to keep consistency for future replications. We then did some text cleaning, specifically, we removed the following from each post:

Step	Cleaning Task	Description
1	Remove all text after: "TLDR", "TLDR:",	TLDR stands for Too Long Didn't Read

	“TL;DR”, “TL DR”, “TL DR:”.	and the text that follows often is not part of the actual story.
2	Remove all links	Not relevant for stories.
3	Remove “&amp” and “”	These are HTML elements that are not needed in our corpus.
4	Remove “***” or more *.	These are often used for formatting purposes and are not needed for our corpus.

Figure 8: Data cleaning performed on corpus data.

Next, we create ‘sentence chunks’ where each post is chunked into chunks of 46 sentences. This is done because of the token size limit of the transformer models that we aim to use later where each post would often exceed the token input size limit and would therefore need to be truncated. Sentence chunking is done to avoid truncation as much as possible while being able to squeeze as much text as possible into each input observation. The number of 46 sentences was selected after testing several thresholds and observing how much was being truncated or padded for the overall corpus. 46 sentence chunking was found to be a good tradeoff between the two.

The following show the scripts used for data preprocessing.

Python File	Description
`KaggleAPI.py`	`Object to handle connection to Kaggle API to upload and download files.`
`kaggle_dataset_down.py`	`Script to download Kaggle datasets`
`CorpusProcessor.py`	`Object to handle processing and I/O of downloaded corpus to disk.`
`preprocess_corpus.py`	`Script to preprocess corpus data`

Figure 9: Scripts used for data preprocessing.

4. Exploratory Data Analysis

A relatively simple EDA was done where we looked at the number of stories per SubReddit and the mean number of word counts per SubReddit as shown in Figures 10 and 11. We can observe that most SubReddit had just under 1000 stories and about 1000 words each. The longest stories seem to be from the r/libraryofshadows subreddit with a mean of over 3000 words per story. The shortest stories are from r/Humanoidencounters and r/scaryshortstories.

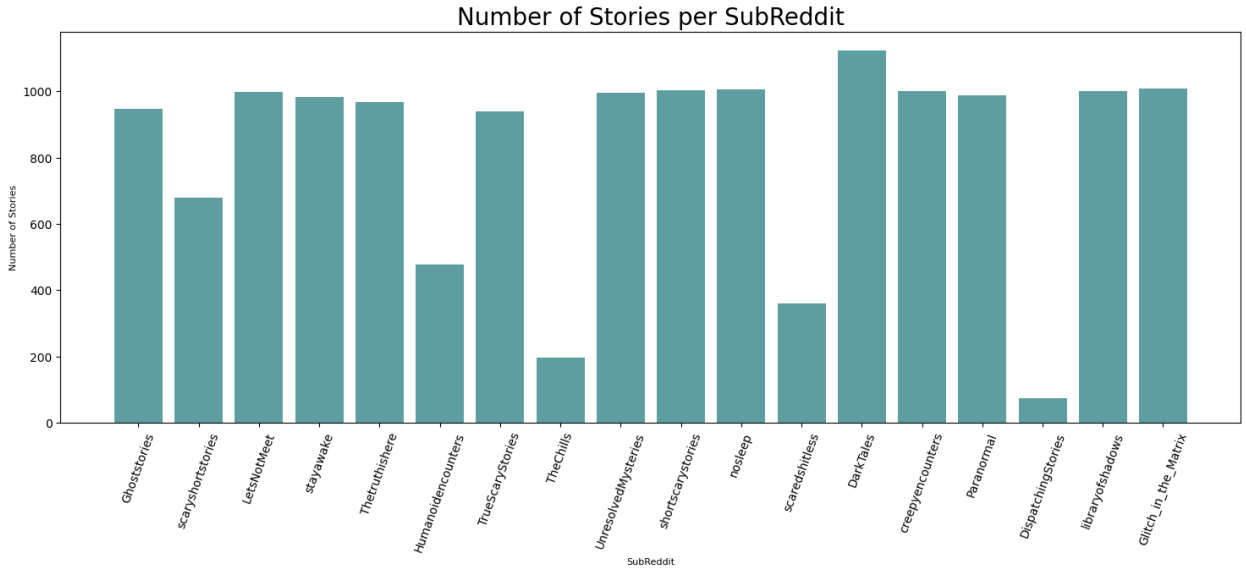


Figure 10: EDA - Number of Stories per SubReddit.

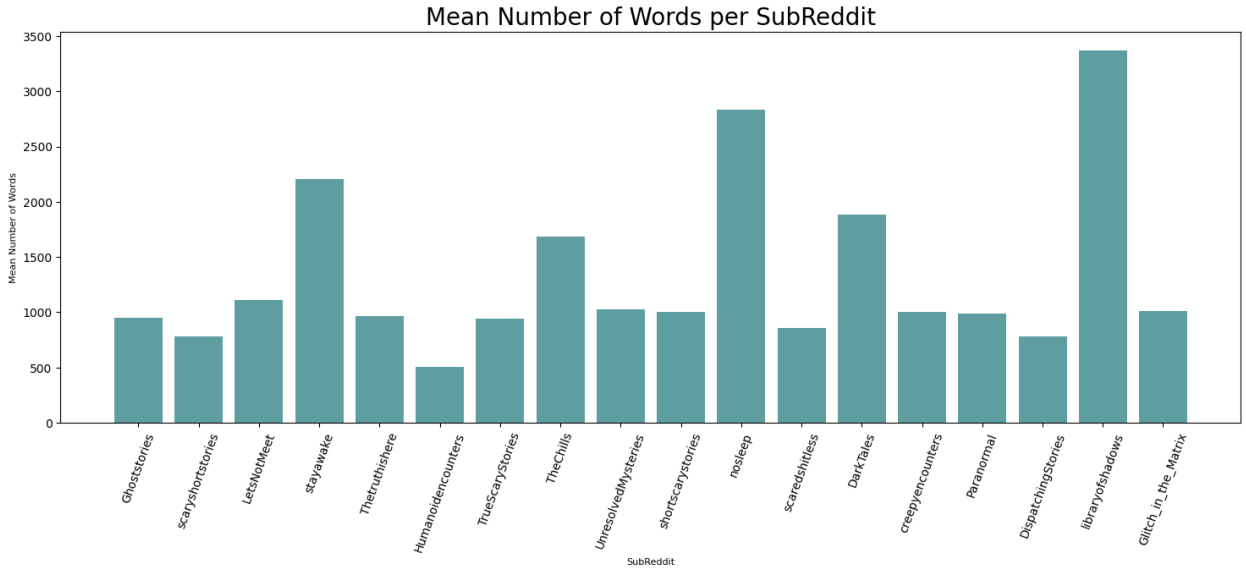


Figure 11: EDA - Mean Number of Words per SubReddit.

Additionally, we also removed stopwords and in order to analyze the most frequently used words in the corpus as shown in Figure 12. It is interesting that aspects of houses like ‘house’, ‘room’, ‘door’ are frequently mentioned.

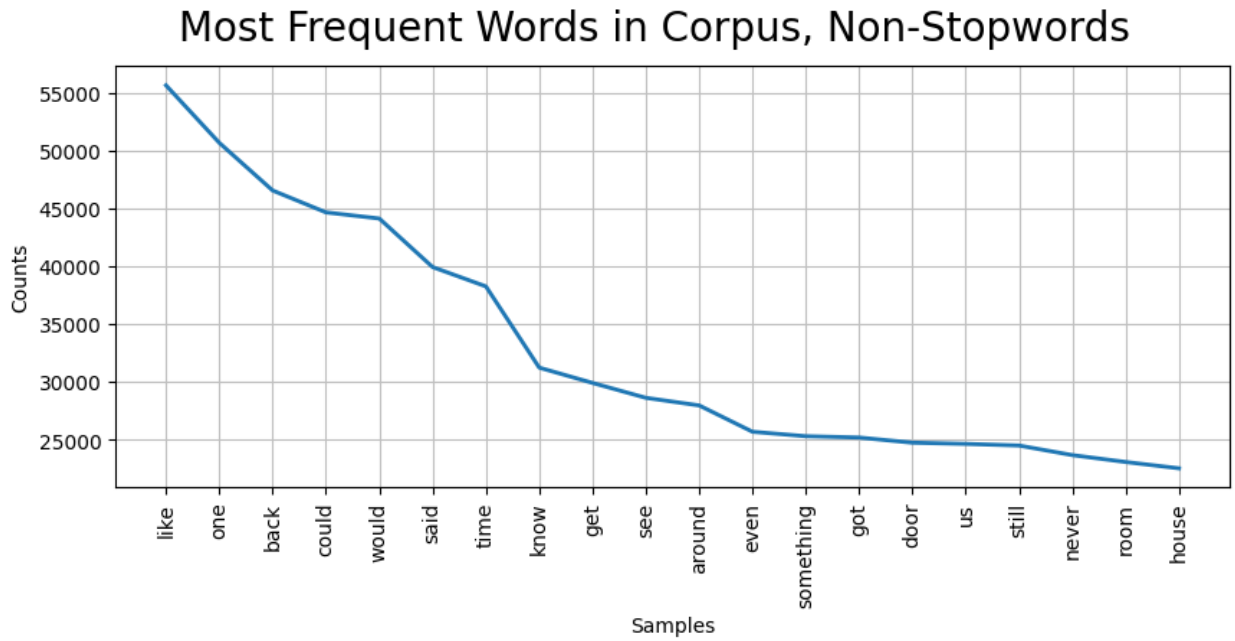


Figure 12: EDA - Most frequent words in the corpus, non-stopwords.

A word frequency plot for each SubReddit can be found under the Appendix 9.1. It is not included here as it is not too important to discuss.

The following show the scripts used for EDA.

Python File	Description
`CorpusProcessor.py`	`Object to handle processing and I/O of downloaded corpus to disk.`
`preprocess_corpus.py`	`Script to preprocess corpus data`

Figure 13: Scripts used for data preprocessing.

5. Experimental Setup - Modeling

5.1 Model Options

For modeling, our group focused on OpenAI's GPT family of transformer models which have proven to be good for text generation tasks. We tried three different models:

1. **GPT2** fine tuned on our corpus for text generation.[8]
2. **GPT-NEO** fine tuned on our corpus for text generation.[9]
3. A **custom GPT2 variant** pretrained on our corpus then again fine tuned on our corpus for text generation

The main frameworks we used were *Hugging Face*[10] to download the pretrained models and tokenizers and *PyTorch*[11] for the model training loop.

5.2 Data Loader

We created custom data loader objects for each of the train, validation, and testing datasets where the object has a *__getitem__()* method which returns the vectorized embeddings in the form of *input_ids* and *attn_masks* from tokenizing the input sentences. The data loader object becomes a generator object where each iteration of it returns one index within our dataset.

```
def __getitem__(self, idx):
        '''
        Params:
            self: instance of object
            idx (int): index of iteration
        Returns:
            input_ids (pt tensors): encoded text as tensors
            attn_masks (pt tensors): attention masks as tensors
        '''
        text = self.sentences_list[idx]
        encodings_dict = self.tokenizer('<|startoftext|>'+ text + '<|endoftext|>',
        truncation=True, max_length=self.max_length, padding="max_length")
        input_ids = torch.tensor(encodings_dict['input_ids'])
        attn_masks = torch.tensor(encodings_dict['attention_mask'])

        return input_ids, attn_masks
```

Figure 14: Data loader's _getitem__ method.

5.3 GPT2

For the first model, our group tried a pre-trained GPT2 model from *Hugging Face*. "*GPT-2 is a large transformer-based language model with 1.5 billion parameters, trained on a dataset of 8 million web pages. GPT-2 is trained with a simple objective: predict the next word, given all of the previous words within some text. The diversity of the dataset causes this simple goal to contain naturally occurring demonstrations of many tasks across diverse domains. GPT-2 is a direct scale-up of GPT, with more than 10X the parameters and trained on more than 10X the amount of data."*[8]

For our purposes, the full GPT2 model with 1.5 billion parameters is too large to train. Therefore, we will be using Hugging Face's 'gpt2' configuration which has 12-layers, 768-hidden, 12-heads, and 117M parameters. The figure below shows the rest of the model configurations and parameters.

Configuration	Value
Tokenizer	`from transformers import GPT2Tokenizer`
Model Head	`from transformers import GPT2LMHeadModel`
Optimizer	`from torch.optim import AdamW`
Custom Tokens	`bos_token='<|startoftext|>', eos_token='<|endoftext|>', pad_token='<|pad|>'`
Number of Epochs	25
Learning Rate	5e-5
Learning Rate Scheduler	Linear
Batch Size	1
Max Input Length	768 Tokens
Model Type	'gpt2'
Seed	42
Loss	Cross Entropy
Metric	Perplexity, Equation (3)
Number of Parameters	117M
Pretrained On	8 million web pages

Figure 15: GPT2 model fine tuning configuration and parameters.

We fine tuned this model on our training corpus for 25 epochs and below are the validation results where the training and validation loss and perplexity for each epoch is shown.

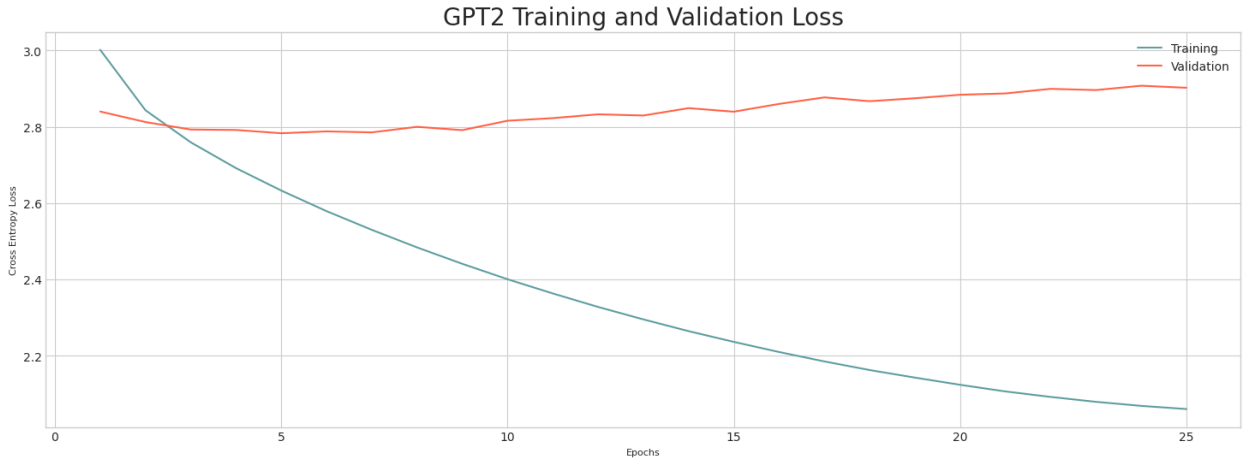


Figure 16: GPT2 model training and validation loss.

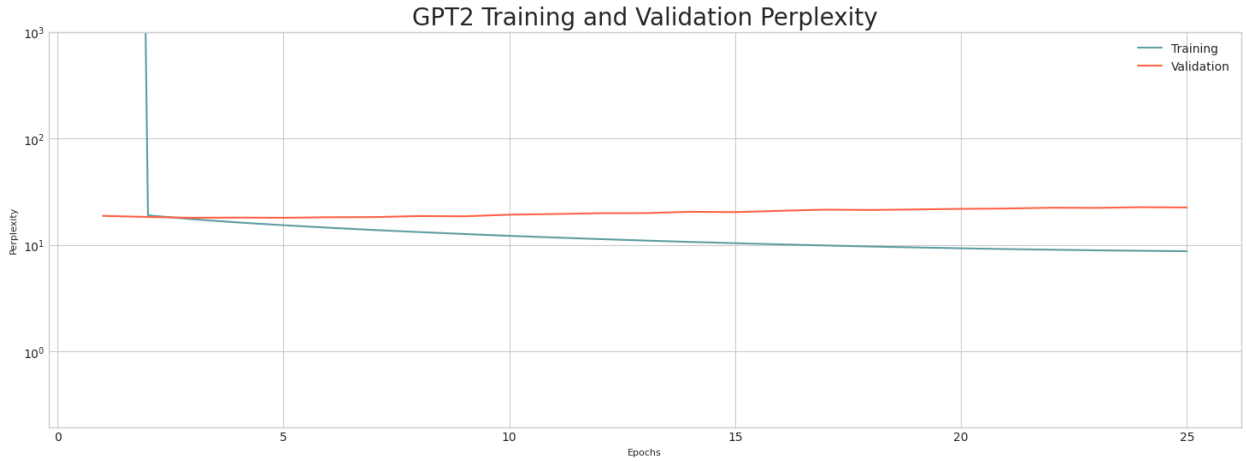


Figure 17: GPT2 model training and validation perplexity.

5.4 GPT-NEO

Our second model is GPT-Neo which was “released in the EleutherAI/gpt-neo repository by Sid Black, Stella Biderman, Leo Gao, Phil Wang and Connor Leahy [as a replication of the GPT3 architecture]. It is a GPT2 like causal language model trained on the Pile dataset. The architecture is similar to GPT2 except that GPT Neo uses local attention in every other layer with a window size of 256 tokens.”[9]

GPT-Neo has several model versions with the largest being 2.7B parameters. For our purposes, we will use the ‘EleutherAI/gpt-neo-125M’ version with 125M parameters instead. “This model was trained on the Pile for 300 billion tokens over 572,300 steps. It was trained as a masked autoregressive language model, using cross-entropy loss.”[9]

Configuration	Value
Tokenizer	`from transformers import GPT2Tokenizer`
Model Head	`from transformers import GPTNeoForCausalLM`
Optimizer	`from torch.optim import AdamW`
Custom Tokens	`bos_token='<|startoftext|>', eos_token='<|endoftext|>', pad_token='<|pad|>'`
Number of Epochs	25
Learning Rate	5e-5
Learning Rate Scheduler	Linear
Batch Size	1
Max Input Length	1024 Tokens
Model Type	‘EleutherAI/gpt-neo-125M’
Seed	42
Loss	Cross Entropy
Metric	Perplexity, Equation (3)
Number of Parameters	125M
Pretrained On	Pile dataset

Figure 18: GPT-Neo model fine tuning configuration and parameters.

We fine tuned this model on our training corpus for 25 epochs and below are the validation results where the training and validation loss and perplexity for each epoch is shown.

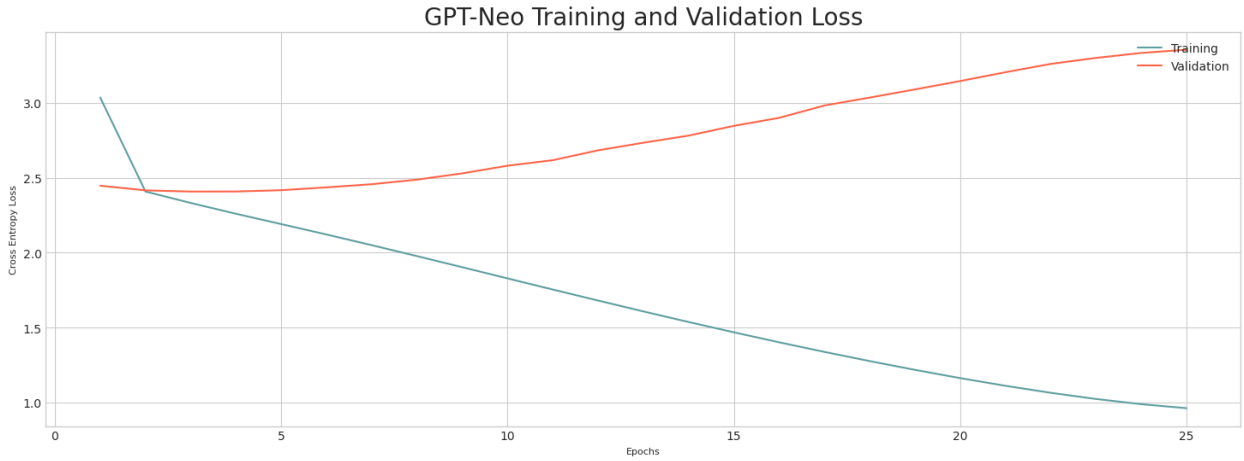


Figure 19: GPT-Neo model training and validation loss.

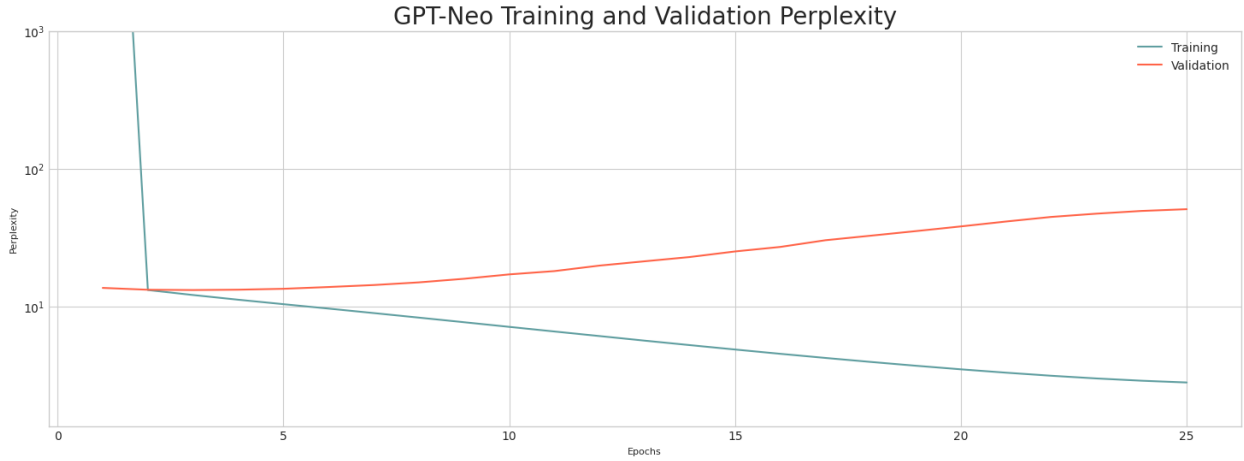


Figure 20: GPT-Neo model training and validation perplexity.

5.5 GPT2Spooky (Custom Pretrained)

Our last model is a custom pretrained model on our own corpus dataset. It uses the same architecture as the first model of GPT2, except it is completely pretrained on just our own custom corpus. The pretraining configurations and parameters are shown in the figure below.

Configuration	Value

Tokenizer	`from tokenizers import ByteLevelBPETokenizer`
	`from tokenizers.implementations import ByteLevelBPETokenizer`
	`from transformers import GPT2TokenizerFast`

Custom Tokens	`bos_token='<|startoftext|>', eos_token='<|endoftext|>', pad_token='<|pad|>'`

Model Head	`from transformers import GPT2LMHeadModel`
Vocab Size	`8000`
Number of Attention Heads	`12`
Number of Hidden Layers	`6`
Optimizer	`from torch.optim import AdamW`
Number of Epochs	5
Batch Size	64
Max Input Length	512 Tokens
Seed	42

Figure 21: GPT2Spooky pre training model configuration and parameters.

After pretraining on our custom corpus dataset, we then fine tuned it on our downstream task of story generation with the following configurations.

Configuration	Value
Tokenizer	`from transformers import GPT2Tokenizer`
Model Head	`from transformers import GPT2LMHeadModel`
Optimizer	`from torch.optim import AdamW`
Custom Tokens	`bos_token='<|startoftext|>', eos_token='<|endoftext|>', pad_token='<|pad|>'`
Number of Epochs	25
Learning Rate	5e-5
Learning Rate Scheduler	Linear
Batch Size	1
Max Input Length	512 Tokens
Model Type	‘gpt2spooky’
Seed	42
Loss	Cross Entropy

Metric	Perplexity, Equation (3)
Number of Parameters	5M
Pretrained On	Custom Corpus (Scary Stories)

Figure 22: GPT2Spooky model fine tuning configuration and parameters.

We fine tuned this model on our training corpus for 25 epochs and below are the validation results where the training and validation loss and perplexity for each epoch is shown.

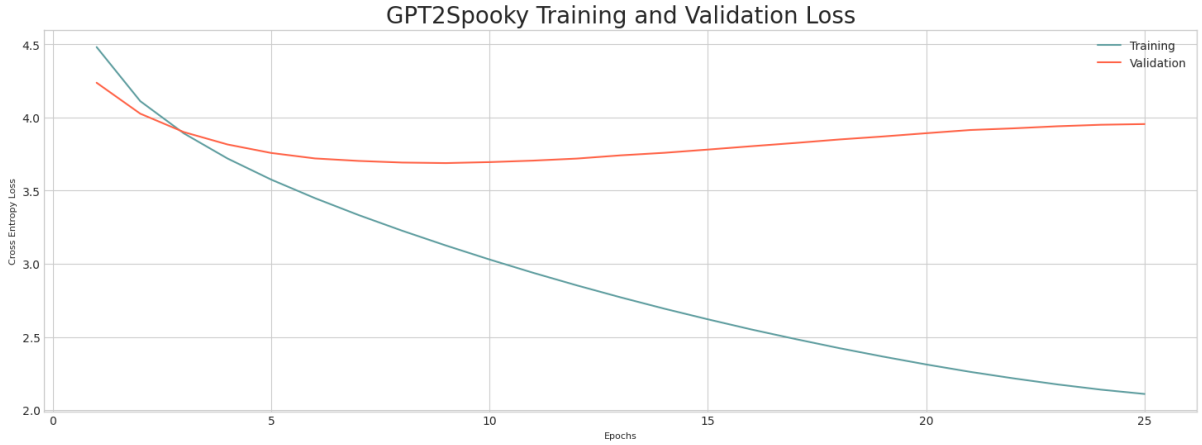


Figure 23: GPT2Spooky model configuration and parameters.

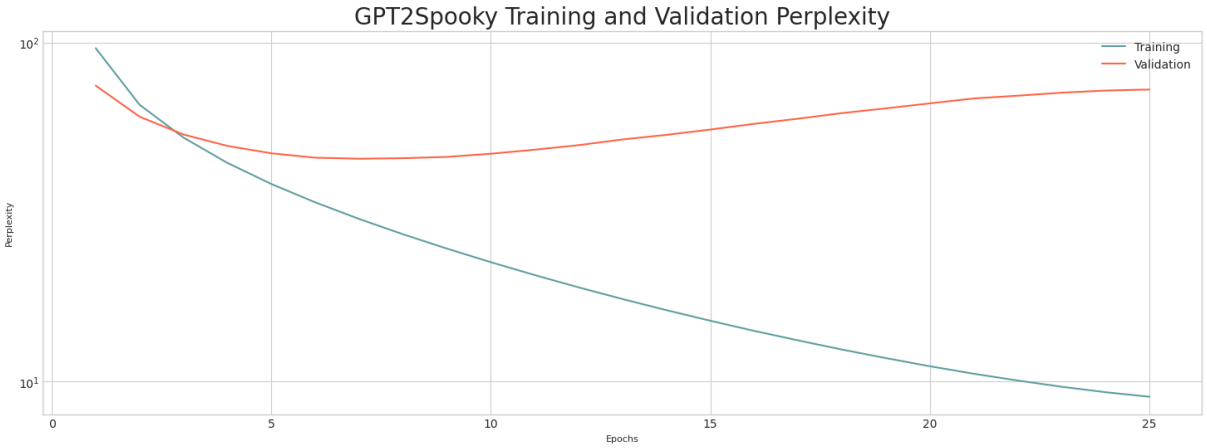


Figure 24: GPT2Spooky model training and validation perplexity.

5.6 Model Weights Distribution on GitHub Releases

We saved our model weights for each of the three final models and created a GitHub release to allow for easy distribution of the model weights since they can be large in size.

6. Results

6.1 Model Evaluation

From the validation perplexity results in Figures 17, 20, and 24, we can observe that the perplexity of GPT2 (22.5) after 25 epochs is the lowest with GPT-Neo (50.7), and GPT2Spooky (72.7) coming in second and third. From these results, we can surmise that GPT2 is the best model at understanding and replicating our corpus while GPT-Neo is second, and GPT2Spooky in third.

However, perplexity is not the best metric for evaluating horror stories which are usually more subjective. Therefore, our group decided to do human evaluation on 30 prompts. For each of the three models we randomly sample a prompt from the testing dataset and let each model generate a story. Then we read each generated story and flagged whether each story was 'scary' in nature and whether it was coherent for an average reader.

prompts	gpt2_25_generate	gpt_neo_25_generate	gpt2spooky_generate	gpt2_25_scary	gpt_neo_25_scary	gpt2spooky_scary	gpt2_25_choherent	gpt_neo_25_coherent	gpt2spooky_coherent
Lorem ipsum…	Lorem ipsum…	Lorem ipsum…	Lorem ipsum…	1	0	0	0	1	1
Lorem ipsum…	Lorem ipsum…	Lorem ipsum…	Lorem ipsum…	0	1	1	1	0	1

Figure 25: Sample of human evaluation table.

We then calculated the proportion of the 30 evaluated generated stories for each model whether the generated story was scary in nature and if it was coherent. The results are shown in the two figures below. It is interesting to observe that GPT2 has the largest number of proportions that are scary stories and are coherent, with GPT2Spooky coming in second, and GPT-Neo coming in last.

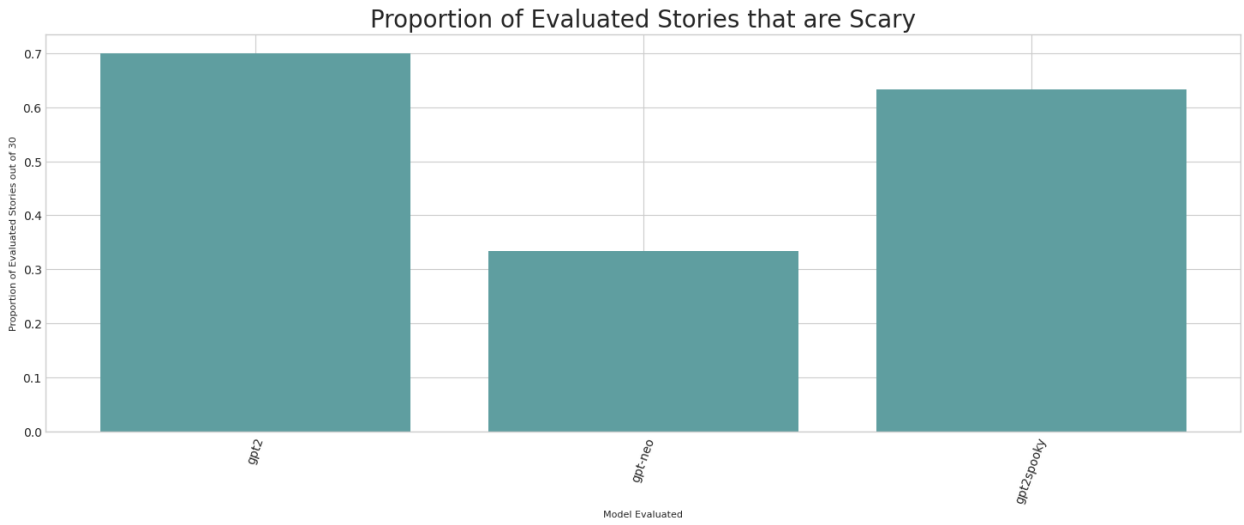


Figure 26: Proportion of evaluated stories that are scary in nature, GPT2 is best

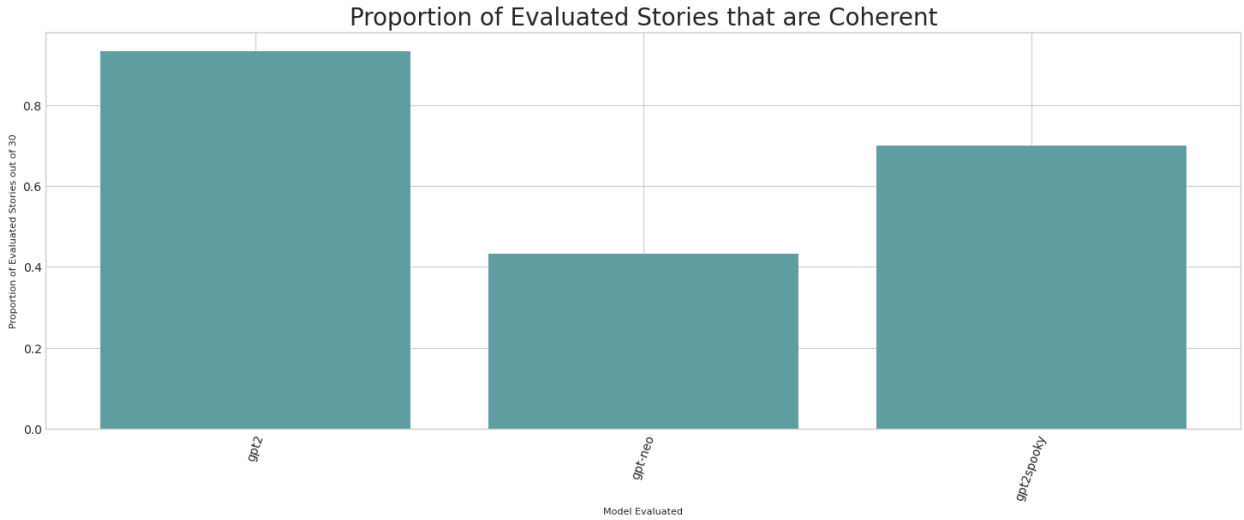


Figure 27: Proportion of evaluated stories that are coherent to read, GPT2 is best

6.2 Chat Bot Front End Interface

Finally, our group built a Flask app to host our best model, GPT2. We decided to build a chatbot that allowed users to either let Woby complete its own stories or allow users to interweave their own responses to continue their stories.

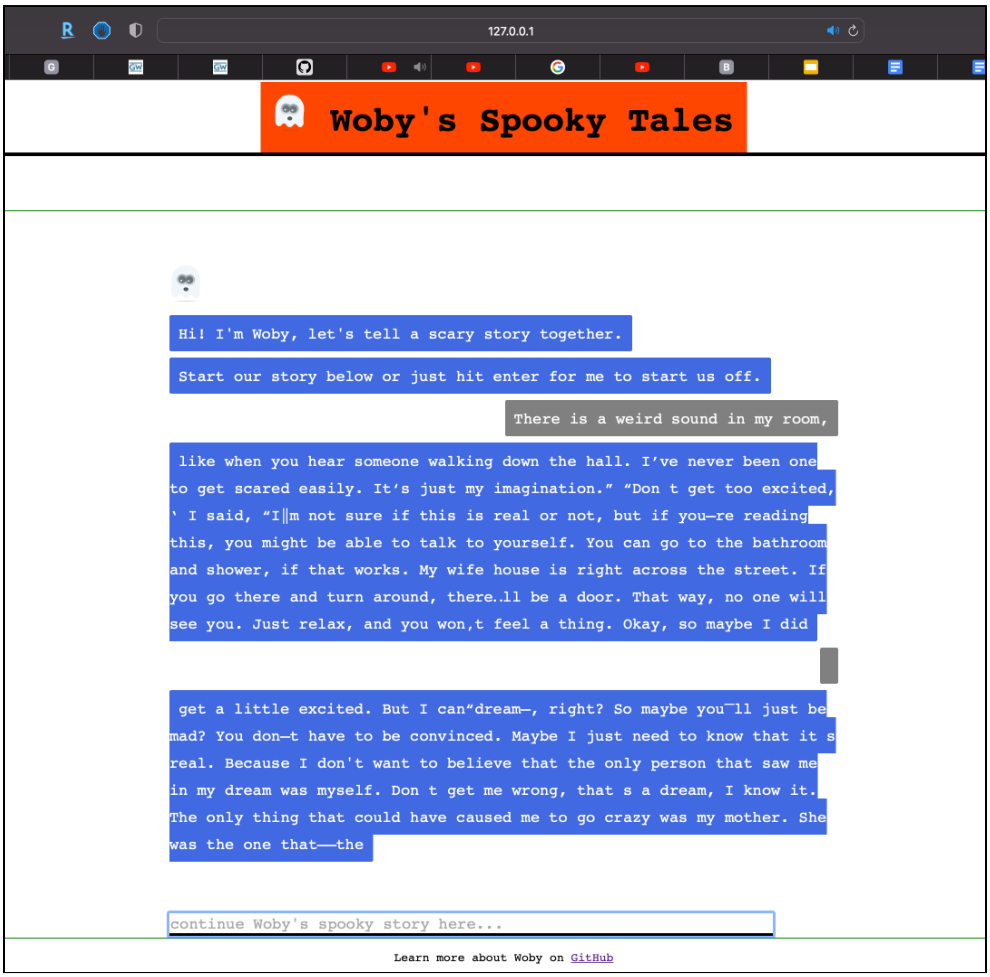


Figure 28: Woby chat bot front end interface, Flask app

7. Conclusions

Our group succeeded in our objective to train and compare several models to perform the downstream task of horror story generation. Although the generated stories from each of the models can at times be not scary in nature and incoherent, there are large proportions where that is not the case.

As can be seen from the results section, our best model for our downstream task of horror story generation is the fine tuned GPT2 model. Our second best model from human evaluation is the custom trained GPT2Spooky. The least performing model is GPT-Neo. It is interesting to observe that GPT2 and GPT2Spooky are not too far off from the GPT2 in proportion of *scary* and *coherent* counts while GPT2 has 117M model parameters and GPT2Spooky only has 5M model parameters. Perhaps with some more tweaking, fine tuning, and increasing the number of model parameters slightly, we may end up with a model that performs just as good as GPT2 while still being significantly smaller in model parameter size.

Other improvements that can be done is to increase the size of our corpus by pulling from more data sources like horror novels, horror movie scripts, etc. Our group learned a lot from working on this project in how to tackle a NLP project and the state of the art methodologies to complete such a project.

8. References

1. Vaswani et. al (2017) Attention is All You Need
2. Glue Benchmark
3. Devlin et. al (2018) BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
4. Radford et. al (2018) Improving Language Understanding by Generative Pre-Training
5. Shelley: Human-AI Collaborated Horror Stories
6. Python Reddit API Wrapper
7. Python Pushshift.io API Wrapper
8. Hugging Face GPT2
9. Hugging Face GPT-Neo
10. Hugging Face Transformers
11. PyTorch
12. Perplexity in Language Models
13. Hugging Face Transformers Fine Tuning
14. Hugging Face How to Text Generation
15. Hugging Face Text Generation
16. Fine Tune GPT2
17. Flask Chatbot
18. Conditional Text Generation for Harmonious Human-Machine Interaction, Guo et al., 2020
19. Transformers: State-of-the-Art Natural Language Processing, Wolf et al., 2020
20. Transformer-based Conditional Variational Autoencoder for Controllable Story Generation, Fang et al., 2021
21. Improving Neural Story Generation by Targeted Common Sense Grounding, hhmao et al., 2019
22. Evaluation of Text Generation: A Survey, Celikyilmaz et al., 2021

9. Appendix

9.1 Word Frequency Plots per SubReddit

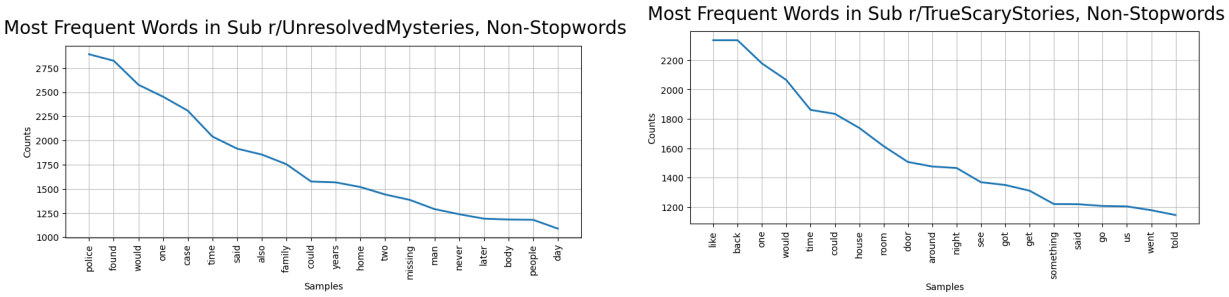


Figure 29: Most frequent words for r/UnresolvedMysteries and r/TrueScaryStories

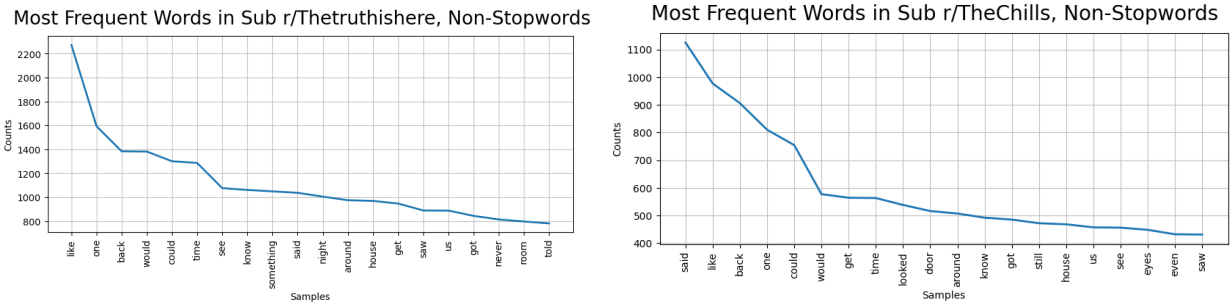


Figure 30: Most frequent words for r/Thetruthisher and r/TheChills

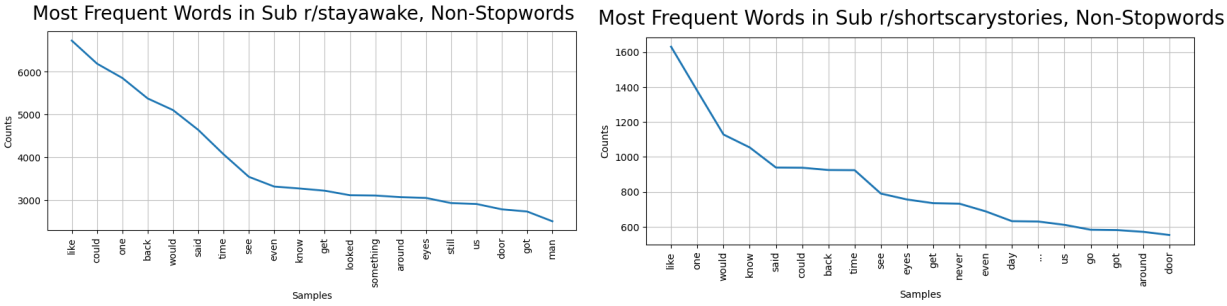


Figure 31: Most frequent words for r/stayawake and r/shortscarystories

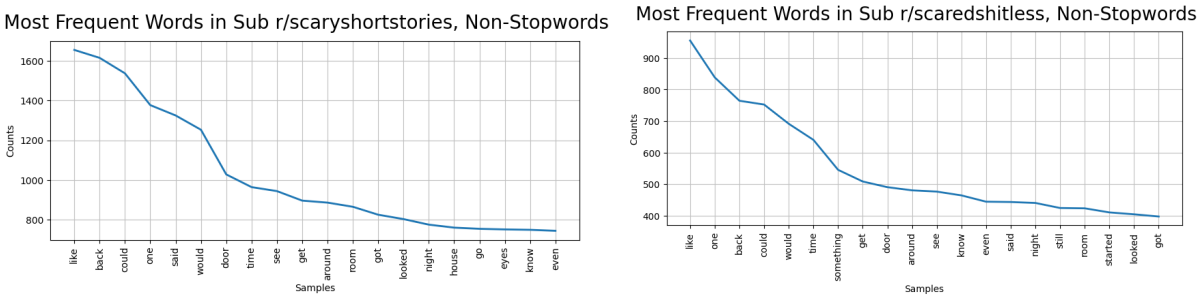


Figure 32: Most frequent words for r/scaryshortstories and r/scaredshitless

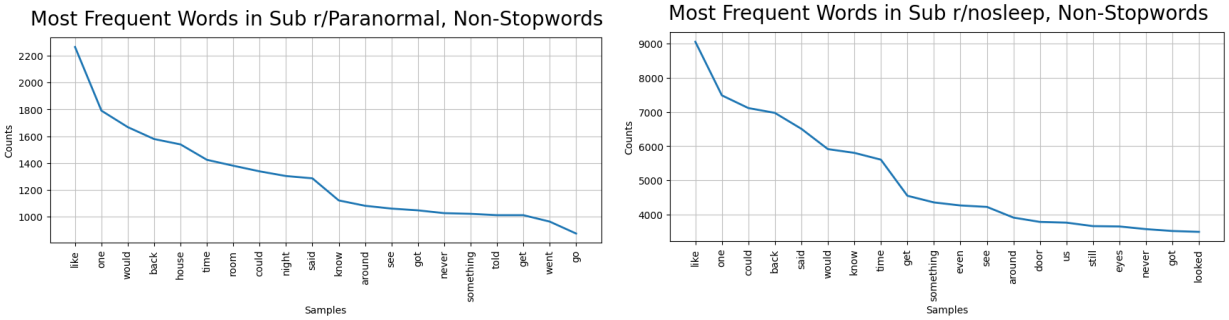


Figure 33: Most frequent words for r/Paranormal and r/nosleep

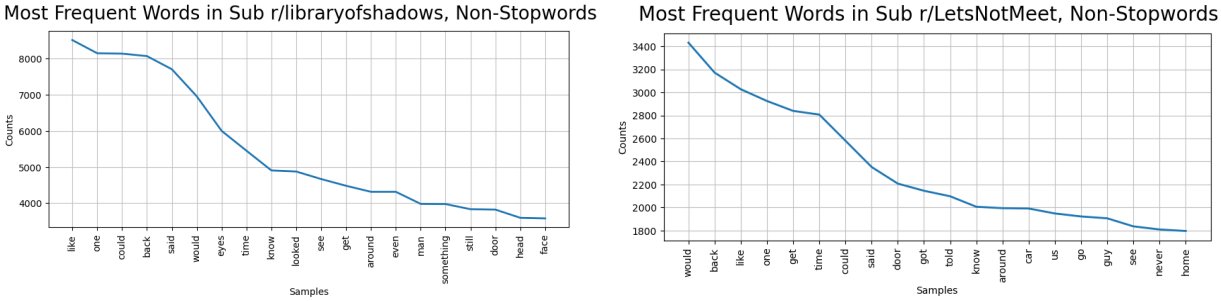


Figure 34: Most frequent words for r/libraryofshadows and r/LetsNotMeet

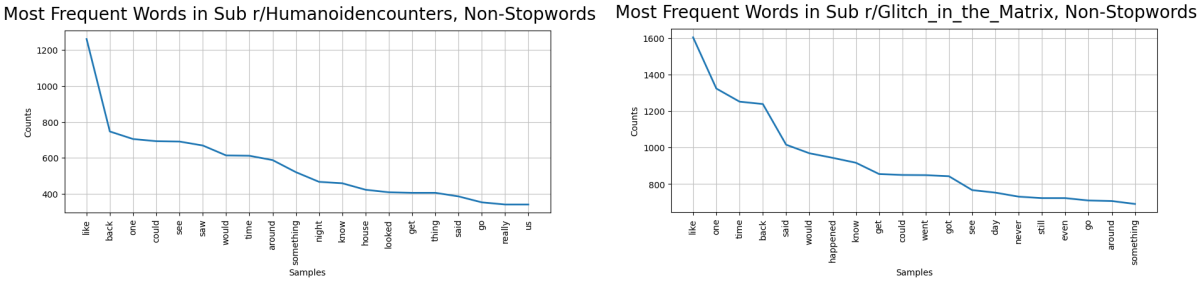


Figure 35: Most frequent words for r/Humanoidencounters and r/Glitch_in_the_Matrix

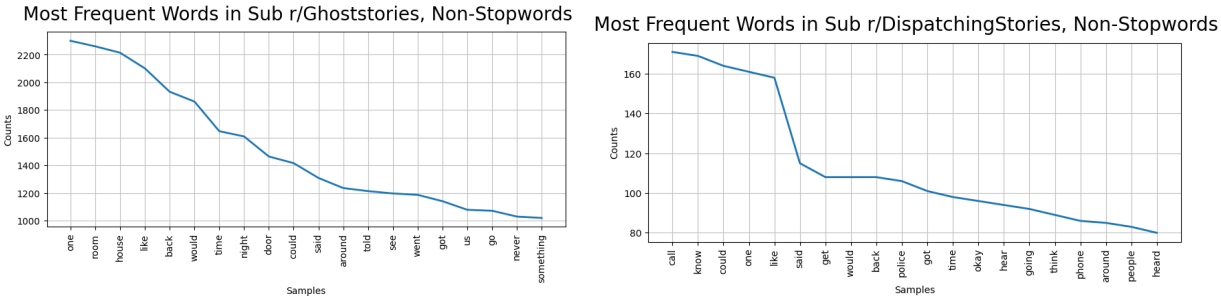


Figure 36: Most frequent words for r/Ghoststories and r/DispatchingStories

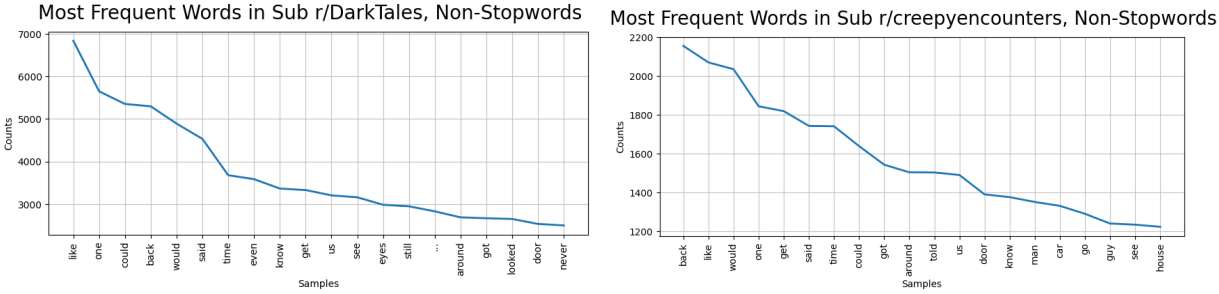


Figure 37: Most frequent words for r/DarkTales and r/creepyencounters

9.3 Random Seed

Random seed used in the modeling phase was *42*.

9.4 Computer Listing

The Python packages that we used are in the *requirements.txt* file.

```
aiohttp==3.8.1
aiosignal==1.2.0
async-timeout==4.0.2
attrs==21.4.0
cachelib==0.6.0
certifi==2021.10.8
charset-normalizer==2.0.12
click==8.1.2
cycler==0.11.0
datasets==2.1.0
dill==0.3.4
et-xmlfile==1.1.0
filelock==3.6.0
Flask==2.1.1
Flask-Session==0.4.0
fonttools==4.33.3
frozenlist==1.3.0
fsspec==2022.3.0
huggingface-hub==0.5.1
idna==3.3
importlib-metadata==4.11.3
itsdangerous==2.1.2
Jinja2==3.1.1
joblib==1.1.0
kaggle==1.5.12
kiwisolver==1.4.2
MarkupSafe==2.1.1
matplotlib==3.5.1
multidict==6.0.2
multiprocess==0.70.12.2
nltk==3.7
numpy==1.22.3
openpyxl==3.0.9
packaging==21.3
pandas==1.4.2
patsy==0.5.2
```

```
Pillow==9.1.0
psaw==0.1.0
pyarrow==7.0.0
pymongo==4.1.1
pyparsing==3.0.8
python-dateutil==2.8.2
python-slugify==6.1.2
pytz==2022.1
PyYAML==6.0
regex==2022.4.24
requests==2.27.1
responses==0.18.0
sacremoses==0.0.49
scipy==1.8.0
six==1.16.0
statsmodels==0.13.2
text-unidecode==1.3
tokenizers==0.12.1
torch==1.11.0
tqdm==4.64.0
transformers==4.18.0
typing-extensions==4.2.0
urllib3==1.26.9
waitress==2.1.1
Werkzeug==2.1.1
xxhash==3.0.0
yarl==1.7.2
zipp==3.8.0
```