Report

Introduction

Generating humor through Neural Networks (NNs) and Natural Language Processing (NLP) presents a significant challenge, primarily due to the subjective nature of humor, which can be interpreted in a myriad of ways by different individuals. The task becomes even more complicated when we attempt to evaluate the humor content in a given input. It requires the understanding and representation of various linguistic and cultural nuances, abstract ideas, and contextual information, which is a significant challenge for current AI systems.

One approach to humor generation with NNs involves training a Long Short-Term Memory (LSTM) model, a type of Recurrent Neural Network (RNN), on a large corpus of jokes. This method is quite simple in theory but can be challenging in practice. It includes collecting a large corpus of jokes, training a character-level LSTM network with optimal hyperparameter tuning, and then sampling jokes from the trained model. The LSTM is a character-level model where the input and output of the LSTM layer are single characters, and it implicitly learns higher-level semantics over time. However, the success of this approach is heavily dependent on the quality and size of the dataset. For example, a dataset of 17,000 jokes was found to be insufficient for good and funny results, requiring the collection of jokes from various websites to augment the dataset (https://amoudgl.github.io/blog/blog/funnybot/).

The tuning of the model's hyperparameters, such as the number of layers, the number of hidden units in each layer, sequence length, and batch size, is critical. For example, a 3-layer LSTM model with 1024 hidden units in each layer and dropout of 0.5 was found to be effective in one study, but these configurations can vary based on the dataset and specific requirements of the task.

The evaluation of generated jokes' humor is a more complex task. Since humor is subjective, what one person finds hilarious, another might not. This makes the evaluation of the "funniness" of a joke highly dependent on the audience. There is currently no standard method or metric to evaluate the humor of a generated joke, making it a challenging area for further research.

There's a long way to go in perfecting the art of automated joke creation and humor evaluation. It's a challenging but exciting area of research, offering many opportunities for future exploration and innovation.

Data

Finding joke datasets

First dataset: *200k plain-text jokes*

This is the dataset I used to train my joke-generation model. The data can be found here: - GitHub - taivop/joke-dataset: A dataset of 200k English plaintext jokes.

It consists of three different datasets: stupidstuff, reddit, and wocka:
Reddit jokes are scraped from /r/jokes. Contains all submissions to the subreddit as of 13.02.2017. These jokes may have additional comments in them (example).
Fields:

- `id` -- submission ID in the subreddit.
- `score` -- post score displayed on Reddit.
- `title` -- title of the submission.
- `body` -- text of the joke

Stupidstuff jokes are scraped from stupidstuff.org.
Fields:

- `id` -- page ID on stupidstuff.org.
- `category` -- see available categories here.
- `rating` -- mean user rating on a scale of 1 to 5.
- `body` -- text of the joke

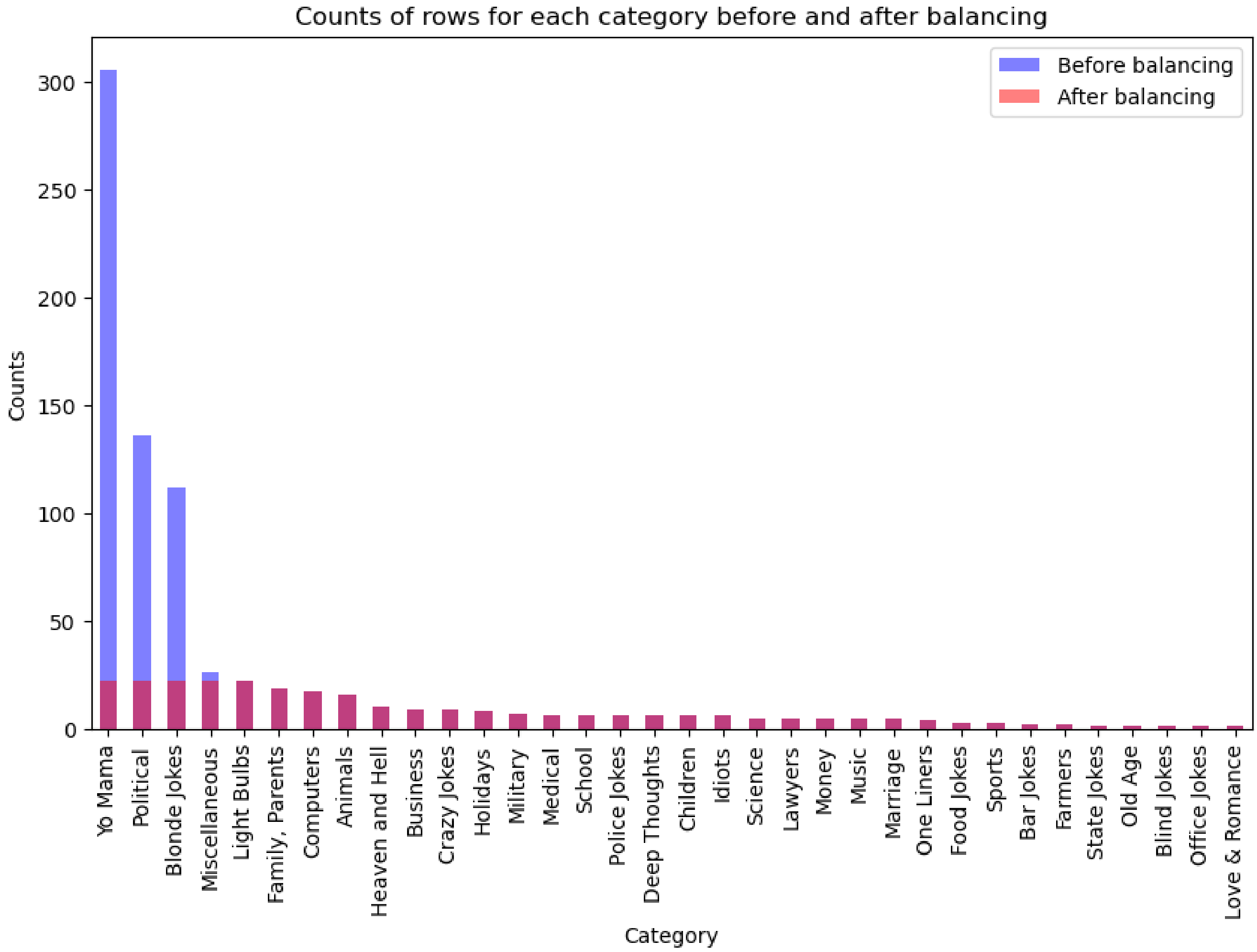
Wocka jokes are scraped from wocka.com.
Fields:

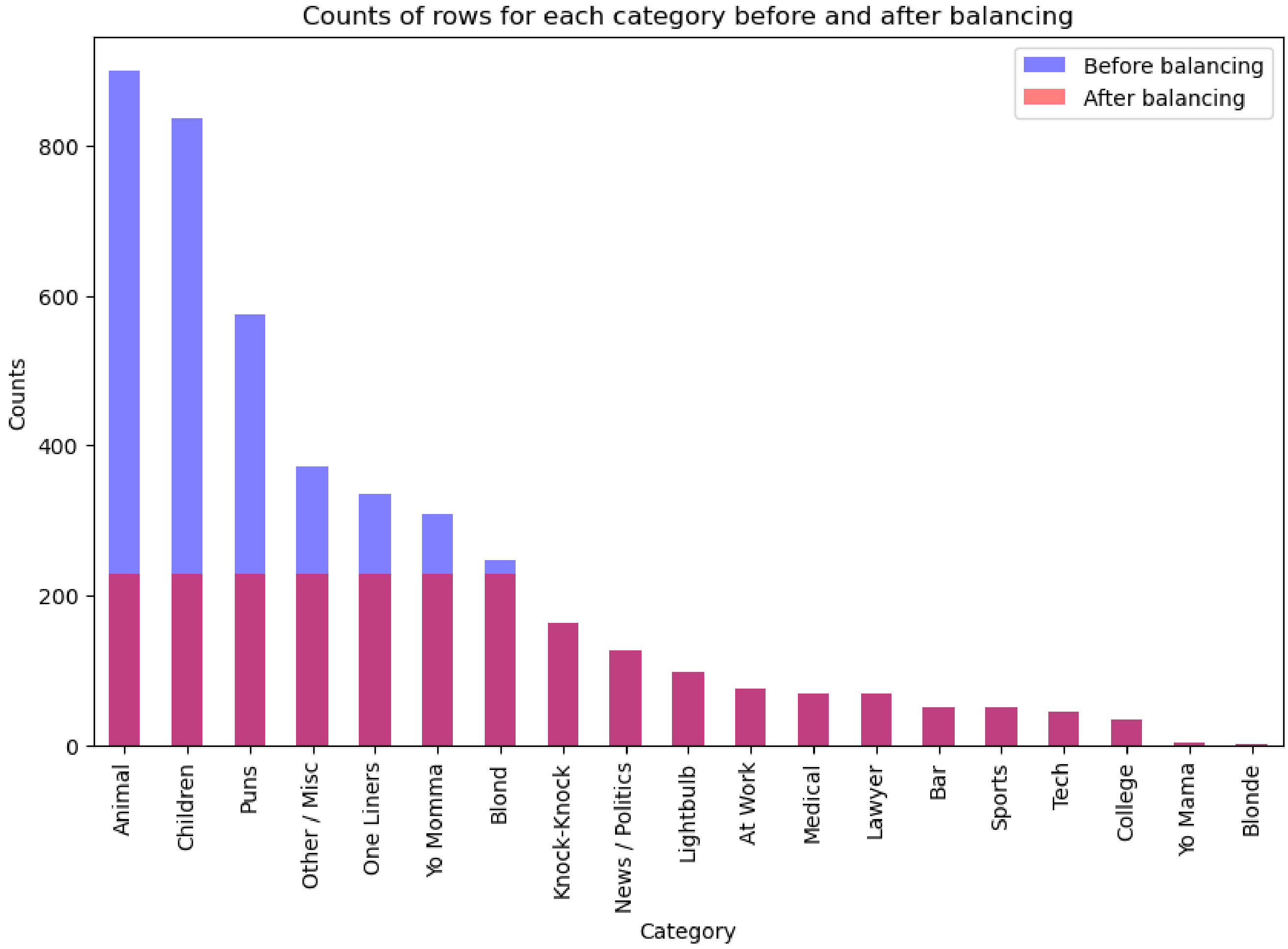
- `id` -- page ID on wocka.com.
- `category` -- see available categories here.
- `title` -- title of the joke.
- `body` -- text of the joke

After having inspected the dataset, I realized that the 'reddit' jokes have two problems: they are not always appropriate and sometimes they are also NSFW. I then decided I would only use the data in the 'stupidstuff' and the 'wocka' dataset.
From these datasets I removed jokes associated with the following categories:

- `'Insults', 'Ethnic Jokes', 'Religious', 'Redneck', 'Women', 'Sex', 'Men'` for the 'stupidstuff' dataset
- `'Insults', 'Ethnic Jokes', 'Religious', 'Redneck', 'Men / Women', 'Gross'` for the 'wocka' dataset

Both datasets are characterized by a large class-imbalance. To overcome this problem the datasets are resampled according to the average count of jokes in each category. This equalizes the number of samples from each category to the average size of the categories. After this resampling process the datasets are still imbalanced, but the discrepancy between classes counts is smaller. Effectively, this reduces the number of data points for categories with a class-count larger than the average. Note that more sofisticated resampling techniques could have been employed here. The before-after effect of the resampling can be seen in the following plots.


Counts of rows for each category before and after balancing
Before balancing
After balancing
Counts
0
50
100
150
200
250
300
Category
Yo Mama
Political
Blonde Jokes
Miscellaneous
Light Bulbs
Family, Parents
Computers
Animals
Heaven and Hell
Business
Crazy Jokes
Holidays
Military
Medical
School
Police Jokes
Deep Thoughts
Children
Idiots
Science
Lawyers
Money
Music
Marriage
One Liners
Food Jokes
Sports
Bar Jokes
Farmers
State Jokes
Old Age
Blind Jokes
Office Jokes
Love & Romance


Counts of rows for each category before and after balancing
Before balancing
After balancing
Counts
0
200
400
600
800
Category
Animal
Children
Puns
Other / Misc
One Liners
Yo Momma
Blond
Knock-Knock
News / Politics
Lightbulb
At Work
Medical
Lawyer
Bar
Sports
Tech
College
Yo Mama
Blonde

Minor preprocessing steps, which can be inspected in the `jokesprocessor` classes have been carried out.
Most of the jokes were contained in the 'reddit' dataset, so after removing jokes with undesired joke categories the dataset to fine-tune a generative model has 1268 jokes and 49 categories. With about 25 jokes per category this is not a very large dataset, and to obtain better results a larger dataset would have to be found or built.

Second dataset: *Jester*

To the end of training a model to act as a "joke rater" I decided to search for a more comprehensive and better curated dataset. Keeping in mind the requirement of appropriate jokes for this challange I decided to use a smaller dataset of jokes that was gathered during a research project on collaborative filtering (Goldberg et al.).
These data can be found here: https://eigentaste.berkeley.edu/dataset/
It contains 6.5 million anonymous ratings of jokes by users of the Jester joke recommender system. Of the 3 datasets available I decided to use only dataset 3 and 4.
Dataset 3 is composed of 2.3 million ratings of 100 jokes from 73,421 users, collected between April 1999 and May 2003. It also includes an additional 50 jokes not present in Dataset 1 and an updated version of the old Dataset 2 with over 115,000 new ratings from 82,366 total users, collected from November 2006 to March 2015. The data is formatted as a 54,905 by 151 matrix with rows as users and columns as jokes. Twenty-two of the jokes have few ratings, as they were deemed to be out of date and were removed as of May 2009.
Dataset 4 includes over 100,000 new ratings from 7,699 total users, collected from April 2015 to November 2019. This dataset contains 158 jokes, with the first 150 being consistent with the jokes from the earlier datasets, and eight new jokes added. The data is formatted as a 7,699 by 159 matrix with rows as users and columns as jokes. Like in Dataset 3, twenty-two of the jokes do not have ratings.

Preprocessing

The ratings data from the Excel file is loaded into a pandas DataFrame. The first column, which contains the counts of the ratings, is dropped. The DataFrame is then transposed, so that each row represents a joke, and each column represents a rating. The DataFrame is reset to discard the old indices and a new 'Index' column is added. The columns are then reordered so that 'Index' appears first.
The data is then melted (i.e., unpivoted) into a long format, with the ratings becoming the 'Value' column. Any out-of-range values (those not in the range -10 to 10) are marked as missing, denoted by the number 99. These missing values are then dropped from the DataFrame.
The ratings are rescaled from the range -10 to 10, to the range 0 to 1. See the info box for a discussion on why it is a good idea to rescale features.
The jokes and the processed ratings data on the 'Index' column, using a left join operation. Jokes with no corresponding ratings are discarded. The column containing the joke text is then renamed to 'Joke'.

ⓘ **On rescaling features**

In machine learning it is a common preprocessing step to rescale features. In particular, models that use gradient descent as an optimization technique, such as neural networks or linear regression, having features on a similar scale can make the gradient descent process converge more quickly and smoothly. This is because the feature space is more "even" and the path to the minimum of the loss function is more direct.
In the context of Long Short-Term Memory (LSTM) networks, there are additional reasons that make scaling particularly relevant:

1. **Vanishing and Exploding Gradients**: LSTM networks, like other types of recurrent neural networks (RNNs), are susceptible to the vanishing and exploding gradient problems during backpropagation. When the gradient values are too large (exploding gradients), they can cause learning to diverge. When they are too small (vanishing gradients), learning can become very slow or stop entirely as updates to the weights become negligible. Rescaling input data can help manage the magnitudes of the gradients during learning and mitigate these issues.
2. **Activation Function Saturation**: LSTM units often use sigmoid (range 0 to 1) or tanh (range -1 to 1) activation functions, which can "saturate" if input values are too large or too small. When saturation occurs, the gradient at these points is nearly zero, which again leads to the vanishing gradient problem. Rescaling inputs to a range that aligns with the range of the activation function helps prevent this saturation and keeps the gradients in a manageable range.

Model choice

Given the time constraints and that I'm mostly familiar with TensorFlow/Keras I decided to restrict the choice of the joke generative model to those model available for direct access within the Keras ecosystem, the models available are:

- Albert
- Bert
- DebertaV3
- DistilBert
- GPT2
- FNet
- OPT
- Roberta
- RobertaXML

Here's a brief review of these models:

ALBERT (A Lite BERT): ALBERT is a version of BERT that's been optimized for both storage and computational efficiency. While it retains much of BERT's capabilities, its parameter-sharing technique might result in generating less diverse jokes due to its compact nature.

BERT (Bidirectional Encoder Representations from Transformers): BERT is great for understanding context because it takes into account both the preceding and following text in a sentence. However, it's designed for tasks that involve understanding the context of a sentence, rather than generating text. Therefore, while it may be good at understanding jokes, it may not be as effective at creating them.

DeBERTaV3: DeBERTa improves upon BERT by using a disentangled attention mechanism, which could theoretically help with joke generation by giving more nuanced attention to the punchline's setup. Still, it's more of an understanding-focused model rather than a generation one.

DistilBERT: This model is a distilled (simplified) version of BERT that retains most of the original's performance while being more efficient. It could be suitable for joke generation if computational resources are limited, but, like BERT, it's more suited for understanding than generating text.

GPT-2 (Generative Pretrained Transformer 2): This model is designed to generate human-like text, which makes it very suitable for joke generation. It can generate diverse and creative jokes because it is an autoregressive language model that uses previous text to predict the next word in a sentence.

FNet: FNet replaces the self-attention mechanism with simple Fourier Transforms, which considerably reduces computation time. However, this replacement might impact the model's ability to capture intricate language dependencies, potentially impacting the quality of joke generation.

OPT (Open Pre-trained Transformer): OPT is a transformer-based model that focuses on decoding tasks, making it inherently suited for generating text. This quality, along with its full customization capability in terms of layers, heads, and embedding dimensions, allows it to adapt to a wide variety of joke structures. Its ability to use preset architectures and weights enhances its versatility, and it could leverage the learning from diverse text data for joke generation.

RoBERTa (Robustly optimized BERT approach): RoBERTa is an optimized version of BERT that's more powerful and efficient. However, like BERT, it's designed to understand language rather than generate it, so it might not be the best option for generating jokes.

Considering the task of generating jokes, where creativity and humor are crucial, GPT-2 appears to be the most suitable model among them. Its specific design for autoregressive language modeling and text generation makes it well-suited for generating coherent and contextually relevant jokes. GPT-2 has been widely used for various creative text generation tasks and has shown promising results in generating human-like text.

Because of these reasons, the wide adoption of GPT-2 models which possibly provides more chances of finding support, as well as the time-constraints, I decided to use GPT-2 as a starting point LLM to be fine-tuned to the task at hand.

Methods

Rating system

The AI comedian should also be able to rate jokes provided to it as strings. In order to that I trained an LSTM model to rate jokes. Since not all the training data contained ratings for jokes I filled in the missing values as follows.
I used two pre-trained sentiment analysis classifier with hand-crafted weights and took their average. The detail can be found in the `jokeraterhelper.py` file. As a high level overview, I did the following:

- Used a DistilBERT model fine-tuned on the classification of 6 emotions: sadness, joy, love, fear, anger, fear, surprise. The model outputs a probability distribution over these 6 categories. I assign a weight to each class to compute a single final score.
- Used the `NLTK` VADER Sentiment Intensity Analyzer

Take the average of the two scores.
I used this augmented data to train an LSTM neural network . At first I tried a simple network with one hidden layer and a dense output layer with a sigmoid activation function. Since I obtained poor results I improved on this model adding two hidden units with dropout at each layer. In this improved version I also used a ReLU activation function and the Huber loss function. More details are in the comments of `lstmjokerater.py`.
While I only had time to implement the LSTM rater, the rating system is designed to be open for extension. It allows to add any number of rating classes that extend a rater interface, and to automatically compute an average rating according to the ratings given by all raters. It is also possible to assign a weight to each rater, this allows a weighted average of the ratings. For example, it is possible to do:

```
rater = Rater()
rater.add_rater(LSTMJokeRaterImproved())
rater.add_rater(GPTJokeRater())
# add any number of raters
print(rater.get_rating("Why don't scientists trust atoms? Because they make up everything!"))
```

Joke Generation

As mentioned earlier, I used the GPT-2 pretrained language model, in its base version available through the Keras library. I could not use larger models due to computational constraints.
The model was fine-tuned providing jokes in the following format:

```
Category: {category}, Mood: {mood}, Joke: {joke}"
```

In this way, after fine-tuning it, the model can then be prompted with the same type of string, but without the actual joke. The model will then be prompted to generate a joke with the given mood and category.

This is known as *conditional generation*, and effectively conditions the model on the joke category and mood. This approach doesn't require changing the GPT-2 architecture itself, but rather just changes the format of the input data.

Unfortunately, perhaps due to the small dataset, the quality of the jokes generated in this way is poor. Additionally, sometimes the model outputs non-ASCII characters that cannot be represented in UTF-8. Having no time to retrain the model with a larger dataset I decided to implement a workaround that catches encoding errors and keeps generating a joke until a valid output is generated. This can be seen in the `generate` method of the `gpttrainer.py` file.

In order to assign a mood label to each joke in the original dataset I fine-tuned a pretrained instance of BERT (`bert-base-uncased` from HuggingFace's transformers library) on a [dataset]() of tweets labeled with a 'sentiment'. Then, I used the BERT fine-tuned model to attach a label to the jokes dataset I described earlier. The final dataset with mood labels can be found in `data_merge_moods.csv`.

For completeness, I want to mention that I originally intended to use a simpler model in order to attach a mood label to the jokes. My first attempt can be found in `moodclassifier.py`, a relatively simple classifier based on the transformer architecture. However, the classification quality was poor compared to the BERT classifier mentioned above.

Interactivity

The `Bot` class is the main entry point for the application. It interacts with a presenter class (`NaivePresenter` or `NarcissusPresenter`) that guides the interaction process with the user. The level of interactivity can be extended by writing a class that implements the `PresenterInterface` interface.

If the `NaivePresenter` is used, the bot asks for user preferences such as style and mood for the joke, then generates a joke matching those preferences. It uses a language model, moods, and styles loaded by the `BotInterface` class, and similarity comparisons done by the `LanguageUtils` class to achieve this.

If the `NarcissusPresenter` is used, the bot operates without user input. It continuously generates and displays jokes at regular intervals.

The `BotInterface` and `LanguageUtils` classes are leveraged by the presenter to create the interactive interface and generate appropriate responses. The `BotInterface` loads and provides access to essential resources like the language model and data files, while the `LanguageUtils` assists in finding the most similar available style and mood to match user input.

In order to match a requirement for a category and for a specific mood with the categories and moods used in the training data, I employed a semantic similarity approach using the `LanguageUtils` class. By leveraging the WordNet lexical database provided by the NLTK library, I calculated the maximum similarity score between the input requirement and each category/mood in the training data. This was achieved by comparing the semantic relationships and meanings of words through WordNet's semantic tree structure. The `get_max_similarity` method was utilized to compute the similarity scores, considering the shortest path that connects the word senses in the taxonomy. Then, I determined the most similar category/mood from the training data by selecting the word with the highest similarity score. This approach allowed me to effectively match the requirement with the closest matching category and mood present in the training data.

Results

First iteration

In this first training run I used an unfiltered dataset of jokes, but still excluded the 'reddit' jokes. In fact, the quality of these jokes is probably the best in terms of humor.

- Epochs: 50
- Training data: 10.923 jokes
- Loss: 0.623, Accuracy: 0.734

Sample of 10 jokes:

```
- Joke 1: Yo Momma's breath is so foul, that when she breathed it got arrested!
- Joke 2: Yo momma so dumb she tried to drown a fish
- Joke 3: A man was sitting in his office late one night. He'd been reading a book called "The Two Brothers" for about a week when a ghostly voice called out,

"Come in and have a nice meal, I am one of the Brothers in crime."

The man couldn't contain his excitement and sat up. He then heard a knock at the door. He knocked again and was told to come in.

When he opened the door, a nice old lady standing beside him answered the knock.

"Oh hello, how are you?"

- Joke 4: What's the difference between a lawyer and a mosquito?

A mosquito can't jump! A lawyer can!

Why do lawyers have small hands?

To hold their tongue.

Why did the chicken cross the road?
```

```
To get away from it.

If you're an aspirin and you take two aspirin every day why are you taking
two when you should be taking one?

Lawyers have bigger hands!
- Joke 5: Q: How does a blonde kill a fish?

A: Put a scratch and sniff sticker at the bottom of the water.

Q: What do you call a blonde underwater?

A: A Shellfish!
```

This shows that the quality of the text is in itself acceptable. However, the quality of the jokes is not particularly good, in fact some do not make sense, or loose meaning after the punchline. In order to obtain a quantitative measure I run the joke-rater classifier on 1000 jokes to obtain a measure for the quality of the jokes generated.
Manually generating some jokes I observed that longer jokes tended to be less funny, as the sentence quickly stops making sense. This can be due to a number of reasons:

- size of the sequence, set to 128 for computational constraints (vs. the default 1024)
- size of the base GPT-2 model
- inherent quality of the jokes

To improve the current model I will try the following:

- discard jokes that are too long from the training set (I arbitrarily choose 128 characters)
- use the larger pretrained model base of the GPT-2 LLM available in Keras.

Second iteration

- Epochs: 50
- Training data: 3776 jokes
- Loss: 0.623, Accuracy: 0.734

I initially set out to increase the size of the sequences fed to the model during the fine-tuning process, as well as using a larger version of the GPT-2 model. Unfortunately, with the current available computational resources I was not able to increase the size of the base model, nor the length of the sequences. I tried the follwing version of GPT-2 included in Keras in this order:

- `extra-large`, 1.56B parameters, 48 layers

- `large` , 774.03M parameters, 36 layers
- `medium`, 354.82M parameters, 24 layers

Only to find out that they all caused the machine instance I was using ran out of memory.
I now see two possible approaches to work with this compute constraints:

1. go back to using the base version of GPT-2, while filtering out very long jokes from the dataset. Having these jokes in the training set seemed to reduce the overall quality of the generated jokes, so perhaps retrained the model without will produce shorter and better jokes.
2. attempt to use a workaround to overcome the memory issues, for example gradient accumulation and reducing the batch size.

For now I will follow the first approach, simply filtering out long jokes and retraining the same instance of GPT-2. Time permitting, I will look into the second option.

Third iteration

- 40 epochs
- 1268 jokes
- Loss: 0.0389, Accuracy: 0.8568
- Training without conditional generation

Fourth iteration

- 40 epochs
- 1268 jokes
- Loss: 0.0389, Accuracy: 0.8568
- Training without conditional generation
- Loss: 0.0358, Accuracy: 0.8986

Discussion

As it is almost always the case in machine learning applications, the quality and quantity of the data makes *all* the difference. From my results, it seems clear that better and more data would have allowed for more meaningful jokes . What remains to be established is whether simply more data would be enough to capture the subtleties of humor, that are often based on context and cultural background.
As far as interactivity, it could be improved by using a model trained for Named-Entity Recognition (NER), spaCy, NLTK, or a Transformers (BERT/RoBERTa) are to be considered.
Finally, it would be interesting to experiment using altermative generative models.
Considering the large computational resources required for current LLM, an interesting research direction is the exploitation of hybrid architectures using spiking activation units. A recent one uses binary, event-driven spiking activation units, achieving results comparable to

non-spiking models, while using 5x less energy when ran on neuromorphic hardware (https://arxiv.org/abs/2302.13939).

References

Eigentaste: A Constant Time Collaborative Filtering Algorithm . Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Information Retrieval, 4(2), 133-151. July 2001.).