CSE 582: Natural Language Processing

The Pennsylvania State University

Department of Computer Science & Engineering

Final Project Report

Submitted to:

Dr. Wenpeng Yin
Assistant Professor
Department of Computer Science & Engineering

Submitted by:

Sadia Anjum Tumpa
sbt5360@psu.edu
PSU ID: 947231364

Submission Date: May 5, 2023

Automatic Movie Title Generation from User-defined Plot

Abstract

A great movie title describes the most salient event compactly and captures the viewer's attention. Although generating a movie title automatically is a very useful task, it is much less addressed than generating tags, plots, reviews. The goal of this project is to design and implement a text generation task that generates the name/title of a movie based on its plot summary. The text generation task of predicting movie titles based on plot summaries has numerous practical applications and can greatly benefit the entertainment industry. To accomplish this task, I use the transformer based model and fine-tuned it on a dataset of movie plot summaries. For the task, I preprocess and clean the data, choose appropriate covariates, evaluate the model's underlying assumptions, and measure the model's performance on the data used and finally document the limitation and challenges found while doing the task.

1 Introduction

The title of a movie plays a crucial role in determining the success of the movie. It is often the first thing that a prospective audience sees, and therefore it is essential that the title is interesting and attention-grabbing. However, it is not enough for the title to be catchy. The title should also be representative of the content of the movie. Inaccurate or misleading titles can lead to disappointment and frustration among audience. They can also harm the reputation of the movie team. A good title should give readers a clear idea of what to expect from the movie. In addition to its importance to audience, the title also serves as an important feature for information retrieval systems. Search engines and other information retrieval systems often give more weightage to the keywords that occur in the title. This means that a movie with an accurate and representative title is more likely to be retrieved by search engines when users search for related keywords.

In general, title generation is an important and challenging problem in the field of Natural Language Processing (NLP) as it requires the model to understand the core essence of the object and encapsulate it in a few words. Title generation is essentially a special case of summarization, which is the task of shortening a given document while retaining its main ideas. However, title generation is more specific in nature as it requires the model to generate a title that accurately reflects the content of the object while adhering to certain linguistic constructions. This is a challenging problem in NLP that requires the model to capture the essence of the object in a concise and informative title while adhering to certain linguistic conventions. The task has many practical applications and is essential for effective communication and information retrieval.

For movie title generation, one of the challenges in title generation is capturing the essential information in the movie while keeping the title concise and informative. The model needs to identify the most salient features of the text and present them in a way that is easy to understand for the reader. Additionally, the model needs to take into account the conventions of language, such as grammar and syntax, in order to generate a title that is grammatically correct and readable.

An automatic movie title generator may create some excellent ideas for naming a film. In addition, it helps create fake movie names or develop some titles that might be the inspiration for next movie script. A recurrent debate is around creativity and if computers would be able to compete with humans at tasks we consider creative: coming up with interesting concepts, designing key visuals, writing good copy.

Previous works has explored generating titles while providing movie titles and genres as input[2]. I designed my final project in a different aspect where instead of providing genres/keywords, can we generate movie titles if the plot/summary of the movie is provided. In my final project in CSE 582, I decided to put this idea to the test to see how good machine learning algorithms could be at generating creative copy. Movie title seemed like a good place to play with as such writing requires good imagination and creativity. The small scale of this generation task (no more than a few words) also makes it easy to experiment with the idea. I use IMDb movie dataset [3] to automatically generate movie title from user defined plot/synopsis with the help of natural language processing.

2 Methodology

2.1 Data collection

For my project I considered data from Internet Movie Data Base (IMDb). IMDb is an online database of information related to films, television series, podcasts, home videos, video games, and streaming content online – including cast, production crew and personal biographies, plot summaries, trivia, ratings, and fan and critical reviews. IMDb began as a fan-operated movie database in 1990, and moved to the Web in 1993. Since 1998, it has been owned and operated by IMDb.com, Inc., a subsidiary of Amazon. As of 2019, IMDb is the 52nd most visited website in the world, according to website ranker Alexa [4].

The movie dataset includes 85,855 movies with 22 relevant attributes. The attributes are imdb title id', title, original title, year, date published, genre, duration, country, language, director, writer, production company, actors, description, avg vote, votes, budget, usa gross income, worlwide gross income, metascore, reviews from users, reviews from critics. These facts and figures provide a wealth of information for researchers to study and explore the impact of various factors on entertainment business. Among these features title, language, description columns are necessary for my project.

2.2 Data preprocessing and preparation

The original movie dataset that I collect as comma separated value (CSV)format from [3] has 85,855 movies with 22 relevant attributes. To generate the movie title from user-defined plot-summary we considered 'movie title', 'language' and 'description' as our relevant variables. In this section we describe how we prepare our data for analyzing the above mentioned natural language generation task.

- **Dealing with Necessary columns:** Since we considered 'movie title', 'language' and 'description' as our relevant variables, I converted the original CSV file to a 3 column data-frame to process conveniently using 'pandas' library of 'python'.

- **Dealing with Missing values:** First, let's see what columns have missing data, how many values are missing, and what percent of the column has missing data. This will give us an idea of not only the number of missing values; but also the significance of this number of missing data. I observed only one row has 'NA' value, which can be easily dropped for the sake of further experiments and so I dropped the row.

- **Dealing with Duplicate values:** Then I checked for the duplicate values in the 'description' field. And 2243 duplicate plots were found which were also discarded. Now there are info of 83611 movies.

- **Picking only English language movies:** Finally for text generation purpose, I have considered only movies that are of 'English' language. Although initially I considered all language movies, but picking only 'English' movies helped me to evaluate if the generated movie titles are making sense or not. To do that, I kept only the rows those have 'language' column set as 'English'.

Then I have prepared dataset of different size (500, 1000, 5000, 10000) and saved them as CSV files so that I can experiment with limited compute capability. Mostly I did not have access to GPU so I had to use a small subset of data for my further experimentation. This is how I performed some basic data analysis to ensure that the data was clean and free of any errors.

2.3 Tools

I have used HuggingFace's excellent 'Transformers' library to fine-tune GPT2 [1] and 'pytorch' framework and 'cpu' as the compute device mostly in my project. As I have been using 'colab' platform which distributes 'GPU' as runtime if available.

2.4 Model Implementation

2.4.1 Model Selection

The 2017 paper *Attention Is All You Need* [5] from the Google Brain team introduced a new neural network architecture for encoder-decoder models based solely on attention. This was a first as attention mechanisms were mostly used on top of other architectures, like recurrent neural networks for example, and not as a stand-alone. One of the encoder-decoder based GPT-2 model comes from this recent wave of large models based on attention, and was created by OpenAI's research team. It made headlines when published, as it was deemed "too dangerous" to be released

in its full version. Recently people have been experimenting with it to generate anything from news articles to poetry. Essentially, it is a very large language model trained on 40GB of text from the web. A language model is a model trained to predict the probability distribution of the next "token" considering the preceding tokens that came before it. A token can be a word, a letter, a subpart of a word. It is up to whoever build the model to decide what the tokens will be. If a series of tokens are given to GPT2, and it outputs the probabilities for what comes next (among all possible tokens in the vocabulary). By appending a token with high probability to the sequence, and then repeat this step again and again large spans of coherent text can be generated. I chose the encoder-decoder based GPT-2 model for this task because it has been shown to be highly effective at text generation tasks and can generate coherent and diverse text. Additionally, it has a large number of pre-trained parameters, which can help the model to learn from a smaller amount of fine-tuning data. Particularly I have used a model called "DistilGPT2" [?] here, which is an optimized version of GPT2's small model trained by the HuggingFace team . Its performance are slightly lower than the original model but it is also much faster to train and run. This is ideal for the sake of experimenting quickly , but I could also switch to larger versions of GPT2 for better results. The model takes as input the movie plot/short description , and then output relevant suggestions of movie title based on the plot summary.

2.4.2 Tokenizer

I needed to slightly adjust GPT2 to let it understand context and distinguish between movie plot and title. For this purpose, I use two special delimiter tokens to separate between these two kinds of information: '$<$ *movie plot* $>$' and '$<$ *movie title* $>$' Also,GPT2 was pre-trained by OpenAI on large spans of text (1024 tokens) and is not originally made for short sentences like slogans. We will thus need to pad sequences with another special token in order to be able to train with variable-length sequences. For each movie title, 3 sequences are needed as input for the model:

1. The context and the slogan delimitated by '$<$ *movie plot* $>$' and '$<$ *movie title* $>$' (as described above)
2. The "token type ids" sequence, annotating each token to the movie plot or title segment
3. The label tokens, representing the ground truth and used to compute the cost function

As mentioned above, cross-entropy will be the cost function to minimize. However, we do not want to penalize our model for what it is not supposed to predict, hence we will only compute cross-entropy on the movie title's tokens. Using the Transformer library, setting label ids to -1 will tag them to be ignored during cross-entropy computation. This is how corpus is converted to embeddings.

2.4.3 Training Parameters

As I mentioned above, I used the PyTorch library to implement the GPT-2 model and fine-tune it on my movie plot summary dataset. I set the maximum sequence length to 64 tokens and used a

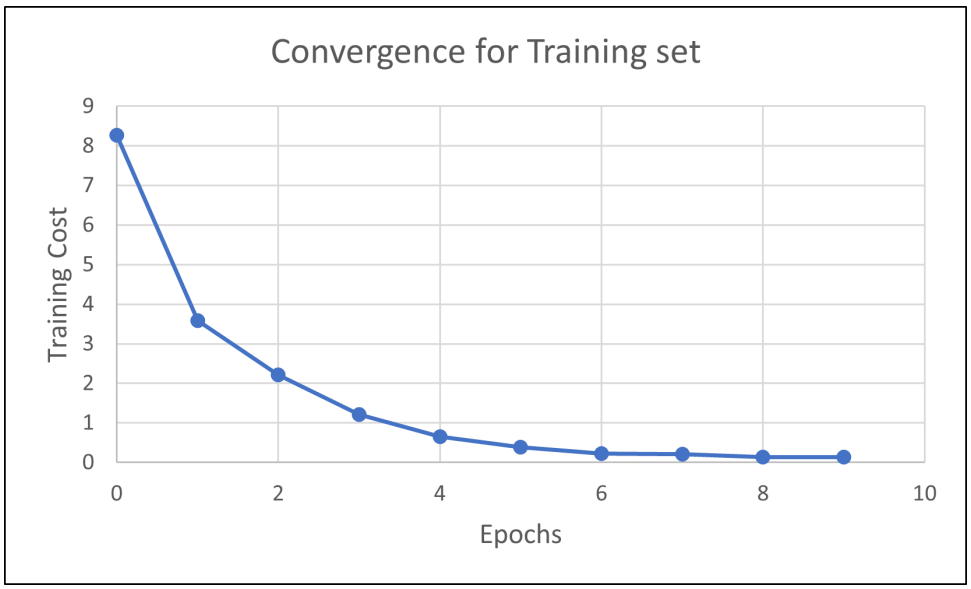

Figure 1: Training convergence plot for experiment

batch size of 8. I trained the model for 10 epochs using the AdamW optimizer with a learning rate of 1e-1. We also used a learning rate scheduler to gradually reduce the learning rate over the course of training. With these, I train the model and keep track of cross-entropy for both the training and validation set using a function that will perform back-propagation and validation for the number of epochs specified. After experimenting with different settings, I found that training for 2 epochs is the maximum I could do before the model starts to seriously overfit to the validation set.

3 Evaluation

3.1 Metrics

Because movie title generation is a creative task, the ultimate performance indicator will be human judgement about the coherence, wit, originality of the slogans generated. However we still need a good proxy measure to train our model the right way.

Language models are mostly evaluated using perplexity and cross-entropy which measures how "confident" our model is about its predictions, the lower the better. We only need to keep track of one of these as both measures are related through the following formula:

Perplexity Equation: $Perplexity(P, Q) = 2^{H(P,Q)}$

Where H(P,Q) is the cross entropy of our language model's probability distribution Q relative to the real probability distribution of the language P.

The objective is to minimize cross-entropy for our domain language. The model should generalize well to new examples and not merely repeat movie titles that it saw in the training set. It should be creative and come up with unique and new movie titles as often as possible. The convergence plot is shown in Figure 1 As this is text generation task, instead of accuracy measurement, it is more appropriate to measure the Perplexity and/or ROUGH metrices.

3.2 Generating New Movie-title

There are multiple methods to generate a sequence of tokens from a language model, including:

- **Greedy sampling :** Greedy sampling consists of always choosing the token with the highest probability. This leads to very predictable & repetitive results.
- **Beam search :** Beam search will generate multiple sequences at the same time (how many is defined by its beam width parameter), and will return the sequence whose overall probability is the highest. This means that the algorithm might end up choosing tokens with lower probability at some point in the sequence BUT these will lead to a final sequence of higher probability in the end.
- **Top-k and Top-p sampling :** Top-k and Top-p will sample random tokens according to their probabilities given by the model, but the choice will be made only from the top K tokens with highest probabilities in the list, or the top tokens who together represent at least P probability (the sum of their probabilities together is $>= P$).

Top-k, Top-p, or a combination of the two usually lead to better results for most applications. Here I re-used the convenient 'sample sequence' function from the 'Transformers' library to easily sample a full sequence with (optionally) Top-k and Top-p. I used k=10 with max length of movie title as 15. The value '15' is used as the average movie title length is 14.42.

Now it's time to generate some movie-title by providing movie plot as input context.

==========

Given plot: "Paul has no faith in his son, Martin, to inherit his prestigious family wine estate. Paul dreams of a harder-working, successful son - a dream that one day seemingly materializes."

Generated Movie-Titles:

The Music Conspiracy Xing
One Hell Follows Sun
Stake Land
Marmony i Somet
The Sushi Brightness Effect
Freedom State
Peppa PigTodd
Sold in Africa
Home
The Daughter

==========

Given plot: "A notorious father-son duo shares a friendly rapport. Things take a turn when a village woman stands against them and circumstances force them to go their separate ways."

Generated Movie-Titles:

The Sea

Beats
Caught Land
Wexford Plaza
The Next Mountain
The Woods Have Eyes, the Woods Have Eyes Like the Woods Have Eyes
azy vs Tulke
Marla Mirchi Tower
Somers Town
Olly, Olly, Olly, Olly, Da Anarchy
==========

Given plot: "The biological and adoptive mothers of a young boy are involved in a bitter, controversial custody battle."

Generated Movie-Titles:

The Creeper
Tigers vtigers
Limb
As Coron Shin
The Lifell Keep
The Perfect 46
Junction
After Image
The Tribe
Maternal Instincts

3.3 Limitations of the Study

Some of the limitations of generating movie-names from user defined plot are as follows:

- **Limited Compute Capacity:** Mostly the experiments are done in 'cpu' setting, accessing to 'GPU' during training improves the performance and makes the training a faster one.
- **Noisy Data:** It's possible that there can be more approaches to clean the data which is not explored in this study. Investigating those can improve the overall text generation task.
- **Subset of Data:** For quick experimentation purpose, I used 500, 1000 and 5000 samples from the processed data of over 21000. Incorporating all those samples can add to the performance.
- **Include other attributes as context**: Here I considered only movie plot as the context, including other relevant attributes like 'genre' could make the movie-title generation task smoother.

However, there are a number of constraints to take into account even if automatic movie generator generates pretty good names from movie plot can yield insightful information. To prevent coming to incorrect conclusions when studying the data, researchers should be aware of these constraints.

4 Conclusion

Overall, the text generation task of predicting a movie name based on its plot summary is a useful and well-defined task that can be evaluated objectively using a range of metrics. The GPT-2 model is a highly effective choice for this task, and our experimental results demonstrate its ability to make accurate predictions on the held out set. However, there is still room for improvement, and future work could explore alternative models or training strategies to further improve the accuracy and diversity of the generated movie names.

References

[1] https://pypi.org/project/transformers/

[2] https://www.nbshare.io/notebook/976197999/Movie-Name-Generation-Using-GPT-2/

[3] Data collection: Github
https://github.com/sahildit/IMDB-Movies-Extensive-Dataset-Analysis/tree/master/data1

[4] https://en.wikipedia.org/wiki/IMDb

[5] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

[6] https://jonathanbgn.com/gpt2/2020/01/20/slogan-generator.html

5 Appendix

5.1 Code for Automatic Movie Title Generation

```
# %% [code]
from transformers import GPT2Tokenizer, GPT2LMHeadModel
MODEL_NAME = 'distilgpt2'
tokenizer = GPT2Tokenizer.from_pretrained(MODEL_NAME)
model = GPT2LMHeadModel.from_pretrained(MODEL_NAME)


# %% [code]
# Declare special tokens for padding and separating the movie_plot from the
                                            movie_title:
SPECIAL_TOKENS_DICT = {
    'pad_token': '<pad>',
    'additional_special_tokens': ['<movie_plot>', '<movie_title>'],
}
```

```
# Add these special tokens to the vocabulary and resize model's embeddings:
tokenizer.add_special_tokens(SPECIAL_TOKENS_DICT)
model.resize_token_embeddings(len(tokenizer))

# Show the full list of special tokens:
print(tokenizer.special_tokens_map)


# %% [code]
import csv

import torch
from torch.utils.data import Dataset


class MovieDataset(Dataset):
  def __init__(self, filename, tokenizer, seq_length=64):

    movie_plot_tkn = tokenizer.additional_special_tokens_ids[0]
    movie_title_tkn = tokenizer.additional_special_tokens_ids[1]
    pad_tkn = tokenizer.pad_token_id
    eos_tkn = tokenizer.eos_token_id

    self.examples = []
    with open(filename) as csvfile:
      reader = csv.reader(csvfile)
      for row in reader:

        # Build the movie_plot and movie_title segments:
        movie_plot = [movie_plot_tkn] + tokenizer.encode(row[0], max_length=
                                                seq_length//2-1)
        movie_title = [movie_title_tkn] + tokenizer.encode(row[1], max_length=
                                                seq_length//2-2) + [eos_tkn]

        # Concatenate the two parts together:
        tokens = movie_plot + movie_title + [pad_tkn] * ( seq_length - len(
                                                movie_plot) - len(movie_title) )

        # Annotate each token with its corresponding segment:
        segments = [movie_plot_tkn] * len(movie_plot) + [movie_title_tkn] * (
                                                seq_length - len(movie_plot) )

        # Ignore the movie_plot, padding, and <movie_title> tokens by setting their
                                                 labels to -100
        labels = [-100] * (len(movie_plot)+1) + movie_title[1:] + [-100] * (
                                                seq_length - len(movie_plot) -
                                                len(movie_title) )

        # Add the preprocessed example to the dataset
        self.examples.append((tokens, segments, labels))
```

```
  def __len__(self):
    return len(self.examples)

  def __getitem__(self, item):
    return torch.tensor(self.examples[item])


# Build the dataset and display the dimensions of the 1st batch for verification:
movie_title_dataset = MovieDataset('movie_titles.csv', tokenizer)
print(next(iter(movie_title_dataset)).size())

# %% [code]
import math, random

from torch.utils.data import DataLoader
from torch.utils.data.sampler import SubsetRandomSampler


# Create data indices for training and validation splits:

indices = list(range(len(movie_title_dataset)))

random.seed(42)
random.shuffle(indices)

split = math.floor(0.1 * len(movie_title_dataset))
train_indices, val_indices = indices[split:], indices[:split]

# Build the PyTorch data loaders:

train_sampler = SubsetRandomSampler(train_indices)
val_sampler = SubsetRandomSampler(val_indices)

train_loader = DataLoader(movie_title_dataset, batch_size=32, sampler=train_sampler
                                                )
val_loader = DataLoader(movie_title_dataset, batch_size=64, sampler=val_sampler)



# %% [code]
import numpy as np
from tqdm import tqdm


def fit(model, optimizer, train_dl, val_dl, epochs=1, device=torch.device('cpu')):

  for i in range(epochs):

    print('\n--- Starting epoch #{} ---'.format(i))
```

```
model.train()

# These 2 lists will keep track of the batch losses and batch sizes over one
                                                    epoch:
losses = []
nums = []

for xb in tqdm(train_dl, desc="Training"):
  # Move the batch to the training device:
  inputs = xb.to(device)

  # Call the model with the token ids, segment ids, and the ground truth (
                                                      labels)
  outputs = model(inputs[:,0,:], token_type_ids=inputs[:,1,:], labels=inputs[:,
                                                      2,:])

  # Add the loss and batch size to the list:
  loss = outputs[0]
  losses.append(loss.item())
  nums.append(len(xb))

  loss.backward()

  optimizer.step()
  model.zero_grad()

# Compute the average cost over one epoch:
train_cost = np.sum(np.multiply(losses, nums)) / sum(nums)


# Now do the same thing for validation:

model.eval()

with torch.no_grad():
  losses = []
  nums = []

  for xb in tqdm(val_dl, desc="Validation"):
    inputs = xb.to(device)
    outputs = model(inputs[:,0,:], token_type_ids=inputs[:,1,:], labels=inputs[
                                                        :,2,:])
    losses.append(outputs[0].item())
    nums.append(len(xb))

val_cost = np.sum(np.multiply(losses, nums)) / sum(nums)

print('\n--- Epoch #{} finished --- Training cost: {} / Validation cost: {}'.
                                                    format(i, train_cost, val_cost))
```

```
# %% [code]
from transformers import AdamW

# Move the model to the GPU:
device = torch.device('cpu')
model.to(device)

# Fine-tune GPT2 for two epochs:
optimizer = AdamW(model.parameters())
fit(model, optimizer, train_loader, val_loader, epochs=2, device=device)


# %% [code]
# Sampling functions with top k and top p from HuggingFace:

import torch.nn.functional as F
from tqdm import trange


def top_k_top_p_filtering(logits, top_k=0, top_p=0.0, filter_value=-float('Inf')):
    """ Filter a distribution of logits using top-k and/or nucleus (top-p)
                                            filtering
        Args:
            logits: logits distribution shape (batch size x vocabulary size)
            top_k > 0: keep only top k tokens with highest probability (top-k
                                                    filtering).
            top_p > 0.0: keep the top tokens with cumulative probability >= top_p (
                                                    nucleus filtering).
                Nucleus filtering is described in Holtzman et al. (http://arxiv.org
                                                        /abs/1904.09751)
        From: https://gist.github.com/thomwolf/1a5a29f6962089e871b94cbd09daf317
    """
    top_k = min(top_k, logits.size(-1))  # Safety check
    if top_k > 0:
        # Remove all tokens with a probability less than the last token of the top-
                                                k
        indices_to_remove = logits < torch.topk(logits, top_k)[0][..., -1, None]
        logits[indices_to_remove] = filter_value

    if top_p > 0.0:
        sorted_logits, sorted_indices = torch.sort(logits, descending=True)
        cumulative_probs = torch.cumsum(F.softmax(sorted_logits, dim=-1), dim=-1)

        # Remove tokens with cumulative probability above the threshold
        sorted_indices_to_remove = cumulative_probs > top_p
        # Shift the indices to the right to keep also the first token above the
                                                threshold
        sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[..., :-1].
                                                clone()
        sorted_indices_to_remove[..., 0] = 0
```

```
        # scatter sorted tensors to original indexing
        indices_to_remove = sorted_indices_to_remove.scatter(dim=1, index=
                                                sorted_indices, src=
                                                sorted_indices_to_remove)
        logits[indices_to_remove] = filter_value
    return logits


# From HuggingFace, adapted to work with the movie_plot/movie_title separation:
def sample_sequence(model, length, movie_plot, segments_tokens=None, num_samples=1,
                                        temperature=1, top_k=0, top_p=0.0,
                                       repetition_penalty=1.0,
                    device='cpu'):
    movie_plot = torch.tensor(movie_plot, dtype=torch.long, device=device)
    movie_plot = movie_plot.unsqueeze(0).repeat(num_samples, 1)
    generated = movie_plot

    with torch.no_grad():
        for _ in trange(length):

            inputs = {'input_ids': generated}
            if segments_tokens != None:
              inputs['token_type_ids'] = torch.tensor(segments_tokens[:generated.
                                                    shape[1]]).unsqueeze(0).
                                                    repeat(num_samples, 1)


            outputs = model(**inputs)  # Note: we could also use 'past' with GPT-2/
                                                   Transfo-XL/XLNet/CTRL (cached
                                                    hidden-states)
            next_token_logits = outputs[0][:, -1, :] / (temperature if temperature
                                                    > 0 else 1.)

            # repetition penalty from CTRL (https://arxiv.org/abs/1909.05858)
            for i in range(num_samples):
                for _ in set(generated[i].tolist()):
                    next_token_logits[i, _] /= repetition_penalty

            filtered_logits = top_k_top_p_filtering(next_token_logits, top_k=top_k,
                                                    top_p=top_p)
            if temperature == 0: # greedy sampling:
                next_token = torch.argmax(filtered_logits, dim=-1).unsqueeze(-1)
            else:
                next_token = torch.multinomial(F.softmax(filtered_logits, dim=-1),
                                                      num_samples=1)
            generated = torch.cat((generated, next_token), dim=1)
    return generated


# %% [code]
movie_plot = "Starbucks, coffee chain from Seattle"
```

```
movie_plot_tkn = tokenizer.additional_special_tokens_ids[0]
movie_title_tkn = tokenizer.additional_special_tokens_ids[1]

input_ids = [movie_plot_tkn] + tokenizer.encode(movie_plot)

segments = [movie_title_tkn] * 64
segments[:len(input_ids)] = [movie_plot_tkn] * len(input_ids)

input_ids += [movie_title_tkn]

# Move the model back to the CPU for inference:
model.to(torch.device('cpu'))

# Generate 20 samples of max length 20
generated = sample_sequence(model, length=20, movie_plot=input_ids, segments_tokens
                                          =segments, num_samples=20)

print('\n\n--- Generated movie titles from your plot ---\n')

for g in generated:
  movie_title = tokenizer.decode(g.squeeze().tolist())
  movie_title = movie_title.split('<|endoftext|>')[0].split('<movie_title>')[1]
  print(movie_title)
```