National Research University Higher School of Economics

Faculty of Computer Science

Programme' Master of Data Science'

Master Thesis

Visualization of different decoding strategies

Student: Vusal EYVAZOV

Supervisor: HSE Dr. Ekaterina ARTEMOVA

Moscow, 2021

Abstract

Holtzman et al.[1] present different decoding strategies, which improve the quality of text generation. The paper presents probability distributions for different decoding strategies. The task is to develop such a visualization tool, which visualizes probability distribution for top-N tokens for the model's dictionary and allows generating texts, by having an initial prompt and using different decoding strategies.

The application was built using the open-source Panel library, which allows to build and deploy applications through jupyter notebook. The language models are taken from the Huggingface library. For the showcase, the application was built using GPT-2 models. Python, Tensorflow, and other common Python libraries were used to build the required functions.

KEYWORDS: transformers, GPT2, greedy search, beam search, sampling, top-k sampling, top-p sampling, nucleus sampling, Python, Huggingface, Panel.

Acknowledgment

I would like to thank my thesis supervisor Ekaterina Artemova for allowing me to work on this project and providing guidance throughout the project.

Introduction

Recent developments in natural language processing led to the building of powerful transformer-based language models that are able to generate human-like texts. These models are trained on several millions of web pages and millions of parameters. Even though training the model on a larger volume of dataset and on bigger parameters have an essential role in the quality of the generated text, decoding methods also play a vital role in this matter.

The main idea of transformers-based language models is to provide the logits output covering the entire vocabulary for identifying the next token/word. These logits are then converted to conditional probabilities, and the next token/word is chosen based on the applied decoding method.

This thesis aims to build an application that will demonstrate the conditional probability distribution for each token/word generation, the difference in decoding methods, and how they affect text generation. The application is intended to be used for research and educational purposes or in the industry to explore models.

The thesis is split into two parts. We will first go through the theoretical foundations of the decoding methods and one of the methods for evaluating the language model's quality. Then, we will present the application by explaining each component for building the application.

Contents

1. Theoretical Foundations

Autoregressive language models are the models that are pre-trained to guess the probability $P(y_1, y_2, \dots y_t|\boldsymbol{x})$ of a sequence of words $[y_1, y_2, \dots y_t]$ given some initial sequence $\boldsymbol{x}$. As obtaining enough training data to estimate probability $P(y_1, y_2, \dots y_t|\boldsymbol{x})$ directly is technically unfeasible, the chain rule of probability is commonly used to factorize it as a product of conditional probabilities:

$$P(y_1, y_2, \dots y_t|\boldsymbol{x}) = \prod_{t=1}^{T} P(y_t|y_{1:t-1}, \boldsymbol{x}) \qquad (1)$$

where:

- $t = 1$: time should start from 1, as an initial sequence is required to predict the following words
- T: user-defined time for the required length of text generation
- $y_{1:t-1}$: a short representation of $y_1, y_2, \dots y_{t-1}$

There are several decoding methods that use the feature mentioned above of autoregressive language models for text generation. We will review the main decoding methods and explain the pros and cons of each technique. In the end, we discuss perplexity, one of the methods for evaluating the language model performance, which we will use to assess the performance of generated text with different decoding methods. This part mainly follows an explanation from "Natural Language Processing with Transformers" book [8]

Greedy Search [8]

Greedy search decoding is the simplest method where the word $\hat{y}_t$ is chosen based on the highest probability at each time step t of text generation

$$\hat{y}_t = argmax_w P(y_t | y_{1:t-1}, \boldsymbol{x}) \tag{2}$$

Probability for each word can be obtained by applying softmax function to the output logits of the model to get the conditional probability distribution of the words for each timestep:

$$P(y_t = w_i | y_{1:t-1}, \boldsymbol{x}) = \frac{e^{z_{t,i}}}{\sum_{j=1}^{K} e^{z_{t,j}}} \tag{3}$$

Where:

- w_i is the chosen word
- z_t is the logit for the current word
- K is the total number of words in the vocabulary

By using the "gpt2-medium" model from Huggingface to generate the next word for the "I would like to …." sentence fragment, we get:

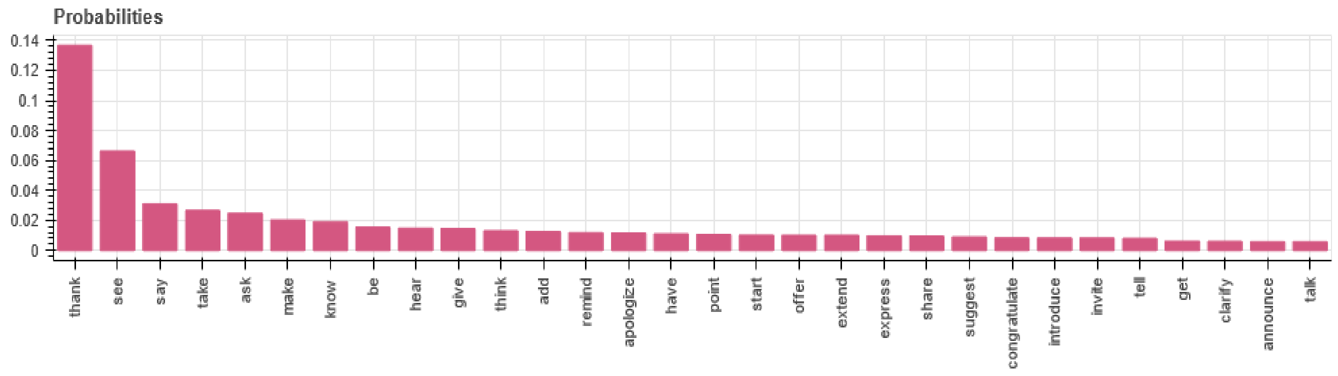


Figure 1. Probabilities for the top 30 words for choosing the word "thank"

Where "thank" will be chosen as the next word and "the" will succeed it based on the below chart which shows the probabilities for the next term for the "I would like to thank …."

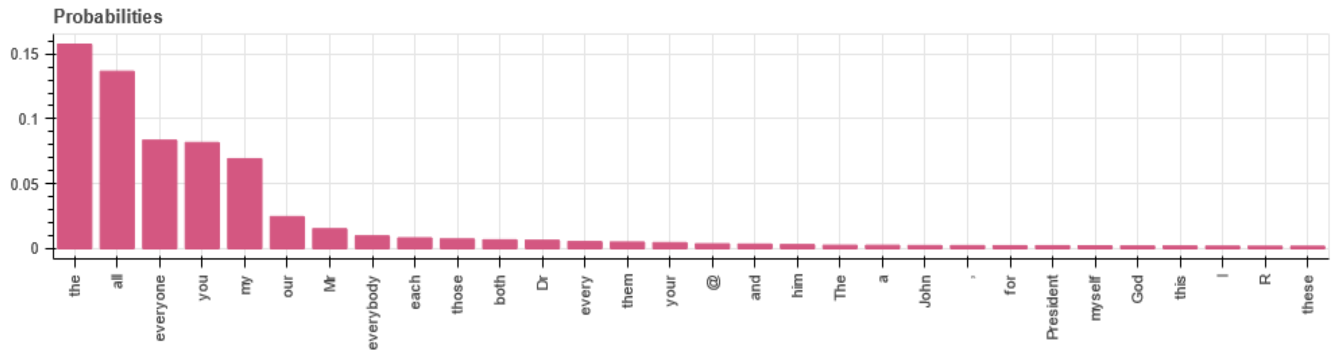


Figure 2. Probabilities for the top 30 words for choosing the word "the"

By continuing the above example, we will get the sentence, **"I would like to thank the staff at the hospital for their efforts and for their compassion and concern for my family."** which is quite reasonable and meaningful. Probabilities for the words after "the" are given below:

staff	at	the	hospital	for	their	efforts	and	for	their	compassion	and	concern	for	my	family
0.029	0.286	0.182	0.041	0.352	0.344	0.067	0.233	0.090	0.256	0.045	0.192	0.097	0.370	0.493	0.205

As shown from Figures 1 and 2, choosing the highest probability words makes us lose the chance to use the following high probability words that can be more meaningful in an overall context. Therefore, the main disadvantage of the greedy search decoding method is that it may lead to missing the word sequence whose overall probability is higher due to preceding higher probability words during text generation.

Below are other examples that were generated using the greedy search method where the first 4 words were given as a starting prompt:

- **He was going to** be a great player for us," said coach Mike Babcock. "He's a guy that's going to be a great player for us."

- **Once upon a time**, there was a man who lived in a village called Krakow. He was a very good man, and he was very kind to his children.

- **Harry Potter is a** fictional character created by J.K. Rowling. The character is a wizard who lives in the fictional town of Hogsmeade, England.

- **Artificial intelligence is going** to be a big part of the future of computing. "We're going to see a lot more of it," said Dr. David A. Li, a professor of computer science at the University of California, Berkeley.

Beam Search [8]

Beam search decoding helps to overcome the main disadvantage of the greedy search method, where beam search keeps track of B, which can be referred to as beam width, to identify the beams with the overall highest probabilities. Beam search involves the multiplication of conditional probabilities to find overall probability. To overcome numerical instability and deal with extremely small numbers, log-probabilities are used instead of probabilities to find overall probability. Considering product rule of logarithms and rule of conditional probabilities, we can express the overall probability as:

$$\log P(y_1, y_2, \ldots y_t|\boldsymbol{x}) = \sum_{t=1}^{N} \log P(y_t|y_{1:t-1}, \boldsymbol{x}) \qquad (4)$$

As an example, by using the previous case, we can build the beam search diagram with 2 beams for the following 6 steps/words:

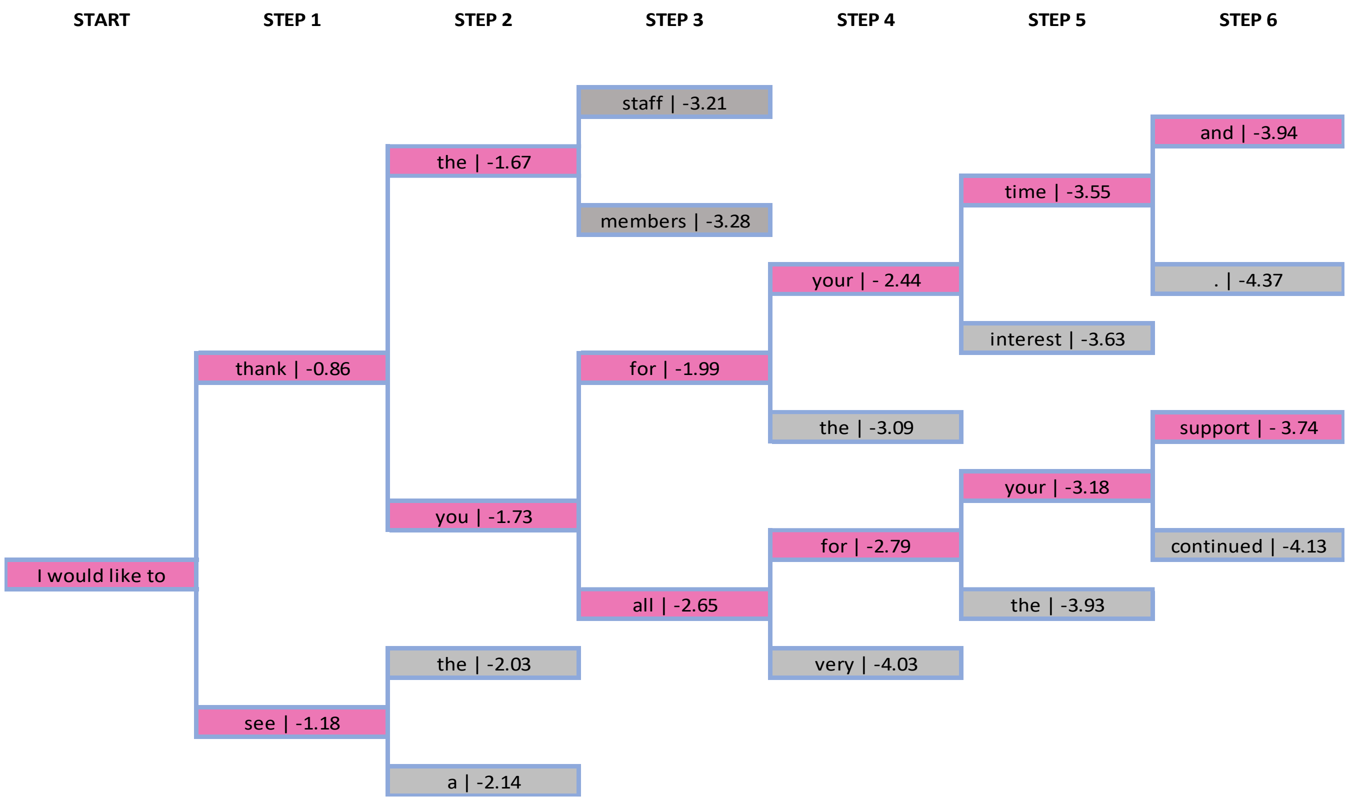


Figure 3. Beam search decoding example with words and overall log probability at that step (pink – selected word for the dedicated step, gray – dismissed word for the next step)

Sampling

According to Ari Holtzman 2020, even though it may seem counter-intuitive, naturally occurring texts would show high variation in the probabilities of the chosen word compared to the text generated by the beam search decoding method.

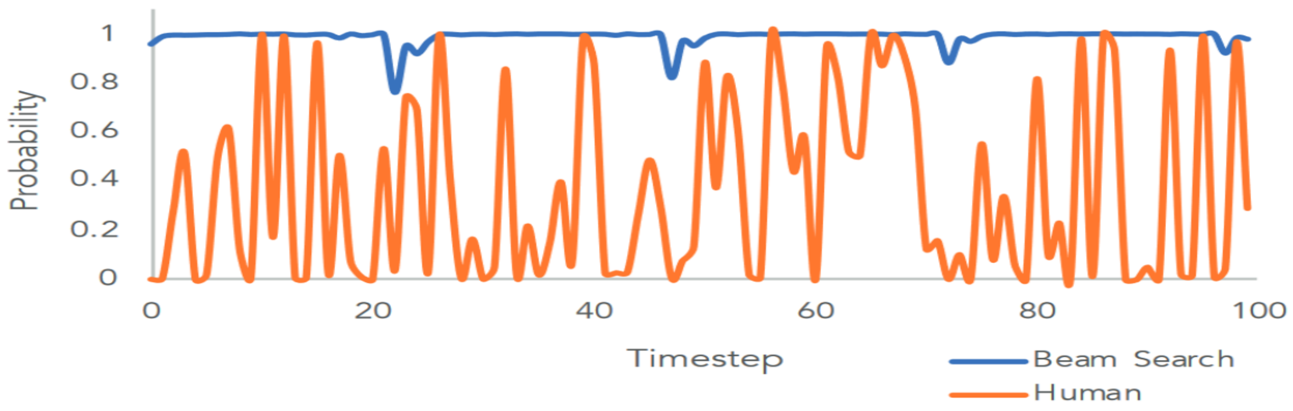


Figure 4. Comparison of Beam search vs. Human-generated text probabilities [1]

Sampling methods are used to overcome the issue mentioned above, which introduces randomness to the text generation.

The simplest sampling method is to randomly sample among the output conditional probability distribution of the words. But this will give us some gibberish texts as the output distribution will include the results with low probabilities that are irrelevant to the context of the initially given word sequence.

Temperature

The probability distribution of the words can be controlled by adding T – temperature parameter to the probability calculation in softmax function to rescale the distribution of the probabilities.

$$P(y_t = w_i | y_{1:t-1}, \boldsymbol{x}) = \frac{e^{z_{t,i}/T}}{\sum_{j=1}^{K} e^{z_{t,j}/T}} \tag{5}$$

Again, by using the "gpt2-medium" model to generate the next word for the previous example "I would like to …." sentence fragment, we get the below figure where the word "help" with 0.0021 probability was chosen as a result of random sampling:

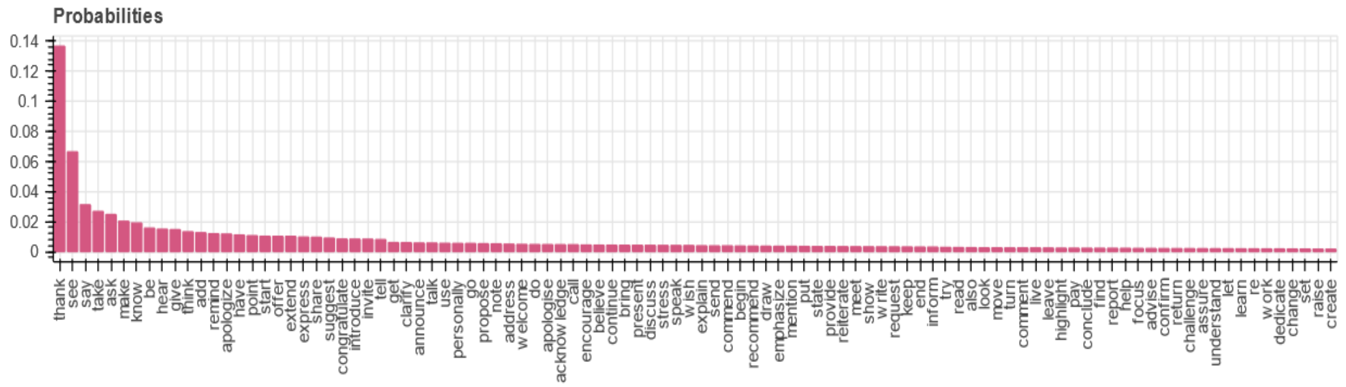


Figure 5. Probabilities for the top 100 words for choosing the word "help")

After applying temperature (lower the temperature: T=0.6) to the same example, we get the word "take" with 0.0312 probability as the next randomly sampled word :

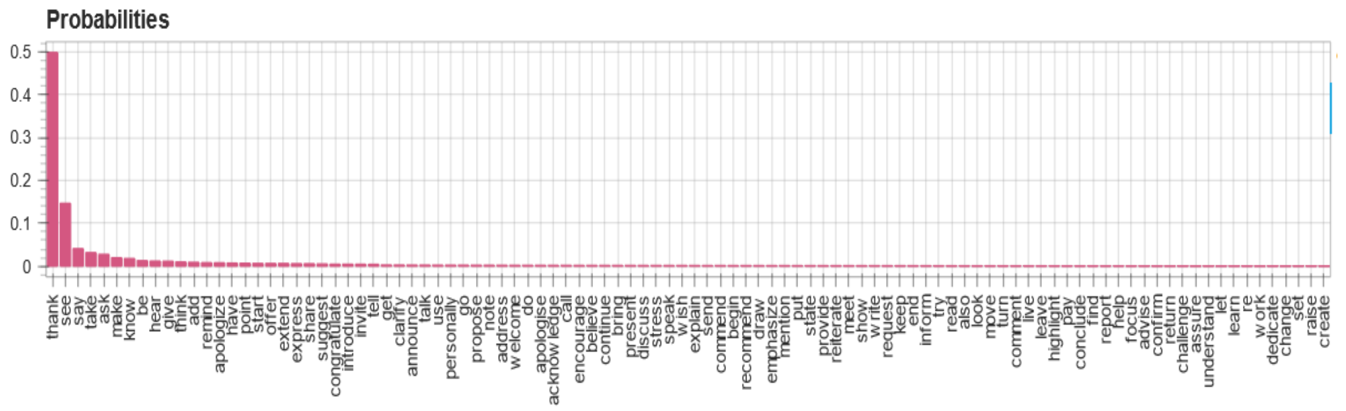


Figure 6. Probabilities for the top 100 words for choosing the word "take")

Comparing both figures (Figure 5 and 6), we see that after applying low temperature (temperature <1.0), the confidence for the high probability words increases, and confidence for low probability words decreases. As expected from the above results, if we increase temperature (temperature >1.0), the probability distribution will be smoothed, increasing the probability of the non-relevant words. Next figure (Figure 6)is for the probability distribution of T=2.0, where word "contact" is randomly chosen as a next word. Note that, word "contact" is not in in the figure (word contact is not among the first 100 words with highest probabilities)

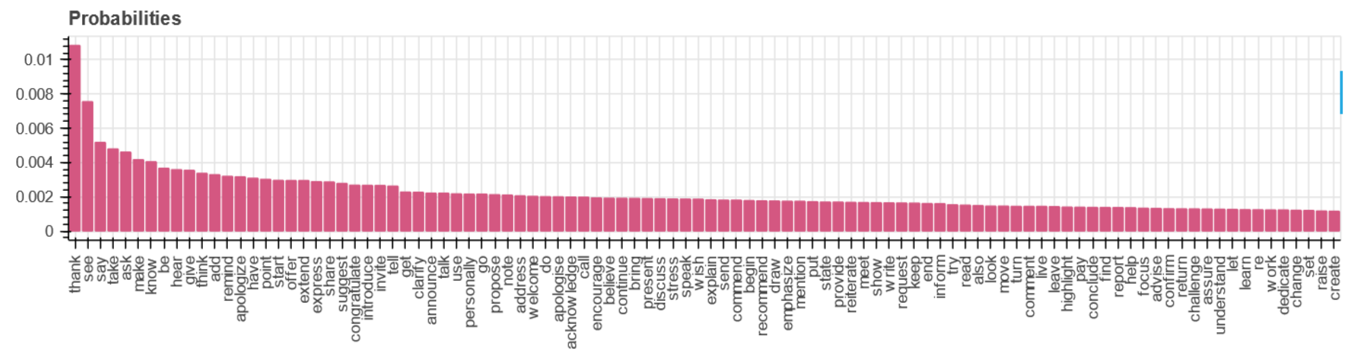


Figure 6. Probabilities for the top 100 words for choosing the word "take"

As a summary, below figure shows how lowering and raising temperature sharpens and smooths the probability distribution respectively:

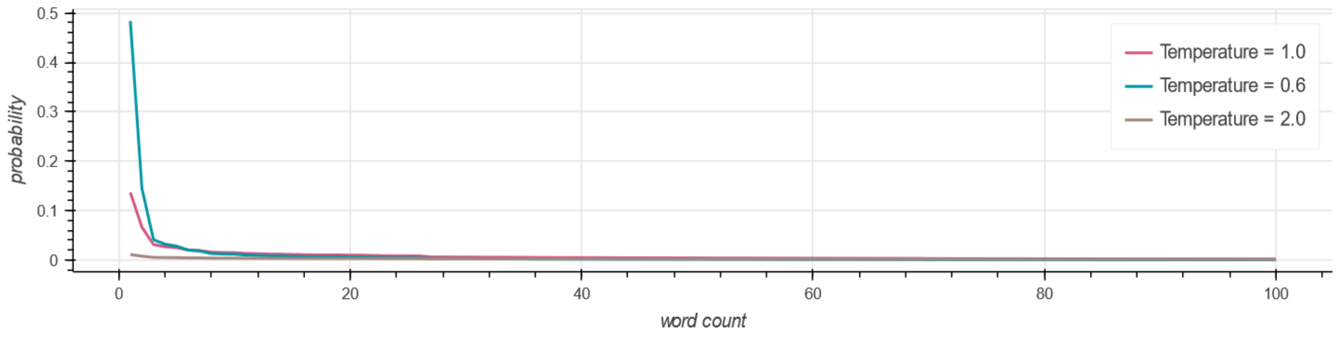


Figure 7. Effect of temperature application on the probability distribution

Below examples shows the texts generated with different temperature values by using the previously mentioned case:

T=1.0:

- **I would like to** help you at all times. I would like to help you with any problem that you would have.
- **He was going to** say'�Clinton met Trump on the campaign trail in July, and at the time said that after having previously been forced to condemn him for his remarks about Muslims.
- **Once upon a time** in the magazine's glory days you could stumble upon something like this from Tito Argenziano, editor-in-chief. This started with an article listing hidden AAA collections, and ultimately whet the appetite for more indications of vast experiments in why words don't mean anything
- **Harry Potter is a** film that has been important to me for as long as I can remember. I don't want to decry it, by any means, but I am going to say that I don't think it's as good as he could have made it.
- **Artificial intelligence is going** to finally make a good living in this industry. Although this is why I have a soft spot for robots.

T=0.80:

- **I would like to** point out that the following two stories are not part of the book. Hank's story is not about the death of an entire family.
- **He was going to** come back to the U.S. and play. But taking him out of the game, the later you let him go, the more you let him go.
- **Once upon a time** when caste was still pretty much the same and there's a fundamental difference between the two, the entire society and the land was run by the caste system. The castes don't own the land.
- **Harry Potter is a** character that is very much in the crosshairs of the White House. It's not about who is on the other side of the bed, it's about who is on the other side of the bed.
- **Artificial intelligence is going** to be the next big thing. It's going to be the next big thing in our lives. It's going to be the next big thing in all of our lives. -- It's not like you were born with that experience.

T=2.50:

- **I would like to**anderiki oddTypecar Boy alongside initiallyboyuminiments Maerg Fe63 legsBecause 51 shifplace
- **He was going to**ocation finished listen waiting scientists marijuana Mexico carrying registered ownedTube Video
- **Once upon a time**RC initially MaybeplayerMany matches Todayigg rainwater Parliament fantasy hypot tested Op
- **Harry Potter is a** fraud commercial symca mallaw Waylog links mob 95 announcement countyBack myth rally
- **Artificial intelligence is going** blog); remote feelings matters entirely passedESTcho attend discover 1970GE

As it can be seen from above examples with T=1.0 we still get "proper enough" sentences, but boring, obviously computer-generated, and more importantly due to wide choice for sampling generally meaningless. In contrast, by applying T=0.8, we manage to get more diverse, entertaining, and close to normal human speech text. Finally with T=2.50 the program gives gibberish results.

Top-K Sampling [2]

Even though, applying the temperature parameter for probability distribution calculation helps to increase the quality of the sampling, to have better control over the set of the words that is used for sampling in each time step and to get better quality of the text we can use Top-K sampling method. This method was first introduced in "Hierarchical Neural Story Generation" paper, which consist of simply choosing top-K words for sampling among the sorted words based on probability distribution [2]. In the paper authors originally sed top-K parameter where ***k=10***, to demonstrate the improvement over beam search decoding method.

This method can be used as an alternative or as a supplement to the temperature sampling method by choosing any arbitrary ***k*** value. By using k=10, for "I would like to ….", we get word "make":

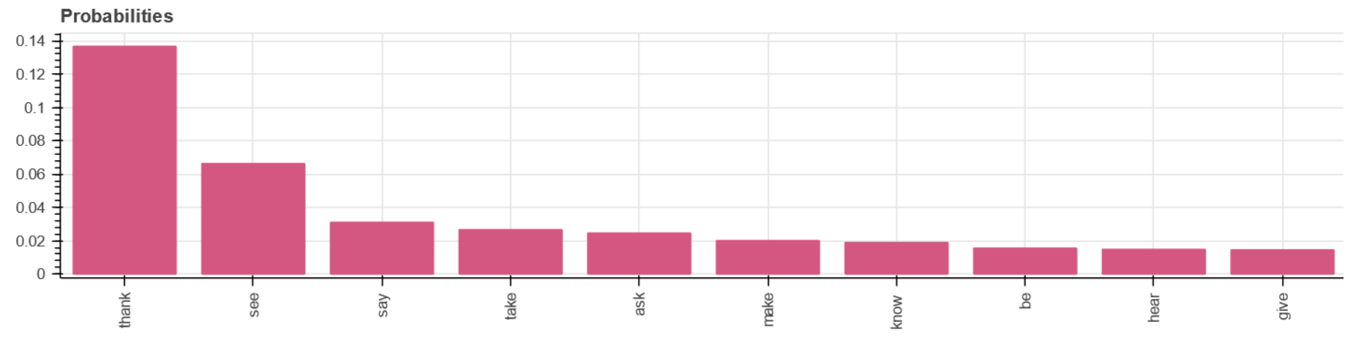


Figure 8. Probabilities for the top 10 words for choosing the word "make"

By continuing the above example, we generate "I would like to make an effort to have a little more time to think about it. ". Note that top 10 words in above example limits the sampling to the **0.3696** of the total probability distributions to generate the word "make".

Below examples shows the texts generated with top-K method where ***k*** =10:

- **I would like to** make an effort to have a little more time to think about it. I am not going to do it in a few days, I am just going to do it in a few days.
- **He was going to** get me a new one. I had been looking for one since he left. He was a very nice guy but it was just so much work and it was just not worth it.
- **Once upon a time** I thought it should have been possible for people in America to vote with their hearts." – Bernie Sanders in response to comments made by Hillary Clinton during her presidential debate last fall that many voters did not understand how she felt about voting for Clinton and instead chose instead to cast their ballots for Sanders.
- **Harry Potter is a** fictional character created by J. K. Rowling for her Harry Potter fanfiction series and published under her pen name JK Rowling. Her characters were developed from various fanfiction authors for various Harry Potter fanfiction stories and fanfiction novels by fans who liked her stories as much as she liked writing them.
- **Artificial intelligence is going** to change everything about how people interact with computers." But even though artificial intelligence will eventually replace humans as our primary interface with computers, there are already plenty of jobs where computers don't understand human languages or understand human behaviors in ways human beings cannot.

Top-p (Nucleus) Sampling

Another popular alternative for sampling is top-p or nucleus sampling that was introduced in the paper [1]. This method is like top-k sampling, where the basic idea is to restrict the range of tokens used for sampling. However, on top-p sampling, as opposed to top-k sampling, we limit the range of selected tokens based on their total conditional probability. This helps to dynamically adjust the range of sampling based on the probability distribution at each time step.

By using the "gpt2-medium" model and top-p method where p=0.5, we get the next word "thank" for the "I would like to …." sentence fragment:

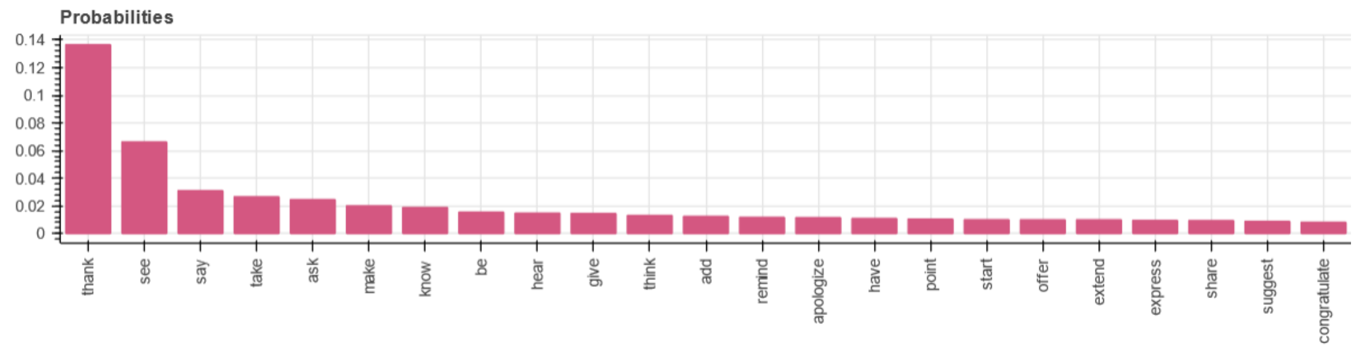


Figure 8. Probabilities for the top-p words with p=0.5 for choosing the word "thank".

By continuing as per above, we get word "everyone" among below given distribution:

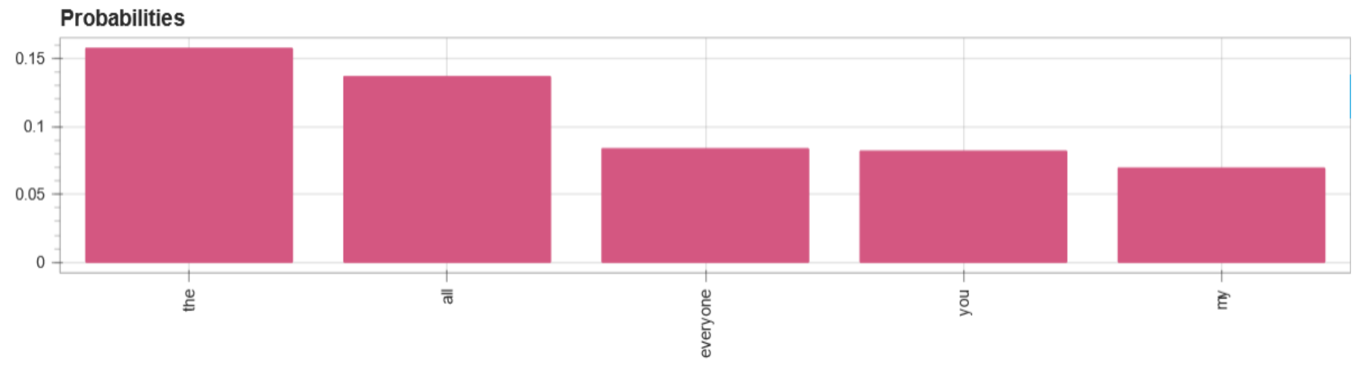


Figure 9. Probabilities for the top-p words with p=0.5 for choosing the word "everyone".

As it can be seen from figure 8 and 9, due to sampling based on top-p value, for each timestep we get different range of words.

The main benefit of top-p sampling compared to top-k, is that it helps us not to loose reasonable options for sampling which will be distributed across the confidence interval with small probabilities. As can be seen from figure 8 and 9, for choosing word "thank" based on p=0.5, we managed to keep more reasonable verbs to sample among but for choosing everyone, due to concentration of probabilities on first five words we managed to eliminate unnecessary word based on our threshold probability.

Below examples shows the texts generated with top-p method where ***p*** =0.95:

- **I would like to** compare lower vitamin D concentrations or serum concentrations with elevated BP status from baseline with assessed cardiovascular outcomes after adjustment for BMI (SBP > 120 mm Hg).

- **He was going to** reject friends," Porter explained to Richmond Magazine about Billy Mulligan's troubled childhood in northwest Sacramento's Hyde Park neighborhood.

- **Once upon a time** they owned massive tracts of forest near Metro Manila that served as breeding grounds for monkeys and lions and tigers and elephants and elephants that escaped into surrounding landscapes for centuries," Clark noted.

- **Harry Potter is a** character who tries to do good, not because he is particular about his goals, but because he's a spirit-worker, an enlightenment seeker, an abstracted consumer who comes to life as a negative parody of every common duty we've ever laid down for an existence grounded in objective reality and god-like powers.

- **Artificial intelligence is going** to grow exponentially in the near future. As we all know, Google has already picked up on this to some degree with all of its AI, and now the Internet giant is also looking to take down Pastebin, the online chat application that has helped millions have a shared workspace.

Perplexity [3]

One of the methods to evaluate the performance of the language model is perplexity. Perplexity can be defined as the exponential of cross entropy:

$$PPL(X) = e^{CE(X)} \tag{6}$$

Cross entropy is the loss function that is mainly used for classification tasks which is also the case for predicting the next token in language models. Cross entropy equation is:

$$CE(X) = -\frac{1}{t}\log P(X) \tag{7}$$

where:

- t is the timestep – or total number of tokens at each timestep
- $\log P(X)$ – log-likelihood of a sequence.

Likelihood of a sequence is a product of each element's probability, which gives the probability of each generated token based on the previous tokens:

$$P(X) = \prod_{i=0}^{t} p(x_i|x_{<i}) \tag{8}$$

Based on the above equation (8) and considering the rule of the logarithms log-likelihood of a sequence is going to be:

$$\log P(X) = \sum_{i=0}^{t} \log p(x_i|x_{<i}) \tag{9}$$

By inserting equations 7 and 9 into equation 6, we get the definition of perplexity:

$$PPL(X) = e^{-\frac{1}{t}\sum_{i=0}^{t}\log p(x_i|x_{<i})} \tag{10}$$

2.Practical Assignment

Library / module loading

For this project, I used Huggingface pre-trained models with language modeling head on top (linear layer with weights tied to the input) for text generation and probability extraction. Due to personal computer limitations below models were used to build the application:

- **DistilGPT2** - 6-layer, 768-hidden, 12-heads, 82M parameters [4]
- **GPT2 (small)** - 12-layer, 768-hidden, 12-heads, 117M parameters [4]
- **GPT2-medium** - 24-layer, 1024-hidden, 16-heads, 345M parameters [4]

Before starting building and testing the functions, we need to import all required libraries and modules:

```
from bokeh.plotting import figure, show
from bokeh.models import ColumnDataSource

from math import pi

import numpy as np

import tensorflow as tf

from transformers import TFGPT2LMHeadModel, GPT2Tokenizer
from transformers import tf_top_k_top_p_filtering

import panel as pn
import panel.widgets as pnw
from panel.template import DefaultTheme
```

Below are the short descriptions of the modules:

- **Bokeh:** library for building interactive plots for using in web pages.
- **Math:** module to access standard mathematical functions
- **Numpy:** a library that is mainly used for working with arrays
- **Tensorflow:** library for building end-to-end machine learning products
- **Panel:** a library that gives the ability to create interactive web apps by using jupyter notebook directly.

After loading the required modules, we need to load tokenizers and models for each model mentioned above. Below is the example of tokenizer and model loading. For loading other models, we change "distilgpt2" and assign model and tokenizer names accordingly:

```
#tokenizer and model for DistilGPT2
gpt2_distil_tokenizer = GPT2Tokenizer.from_pretrained("distilgpt2")
gpt2_distil_model = TFGPT2LMHeadModel.from_pretrained(
    "distilgpt2", pad_token_id=gpt2_distil_tokenizer.eos_token_id)
```

Text generation and probability extraction

The first step is to get the next token logits, which will help us derive probabilities to generate the next word. The below function allows returning the logits.

```
def get_next_token_logits(sequence='Please input some text',
                          model=gpt2_small_model,
                          tokenizer=gpt2_small_tokenizer):
    '''
    Get the logits to derive the probabilities for each prediction.
    - the last layer of the logits output provides the logits for the next token
    '''
    input_ids = tokenizer.encode(sequence, return_tensors="tf")
    next_token_logits = model(input_ids).logits[:, -1, :]
    return next_token_logits
```

As this function is one of the building blocks of the application, detailed descriptions of the variables are given below:

- `input_ids` in the above function are the numerical representation of tokens for building the sequence that is passed as an input to the function.
- `model(input_ids).logits[:,-1,:]` : when `input_ids` are passed to the model, it returns logits as one of the outputs. The logits for the next token is the last layer of the logits output which is derived by `logits[:,-1,:]`

By running above given `get_next_token_logits` function for the "I would like to" sequence, we would get a 2-dimensional array of elements which consist of logits for the entire vocabulary:

```
next_token_logits = get_next_token_logits(sequence='I would like to',
                                          model=gpt2_small_model,
                                          tokenizer=gpt2_small_tokenizer)
print(next_token_logits)
```

```
tf.Tensor([[-148.85843 -148.43329 -154.88771 ... -158.26872 -158.68793 -150.37239]], shape=(1, 50257), dtype=float32)
```

From the above example, we understand that the entire vocabulary for the GPT2 (small) model consists of 50257 tokens.

The following steps are functions to get predictions for the next word and probabilities for filtered tokens. The below-given function is for getting the predictions:

```
def get_prediction(next_token_logits=[[]],
                   decoding_type='Greedy Search',
                   model=gpt2_small_model,
                   tokenizer=gpt2_small_tokenizer,
                   temperature=0.7,
                   top_k=50,
                   top_p=0.95):
    '''
    Get the prediction for the next word
    Greedy search -  returns the word with highest probability
    Sampling - sample based on filtering
    '''
    tf.random.set_seed(60)
    if decoding_type == 'Greedy Search':
        next_token = tf.math.argmax(next_token_logits,
                                    axis=-1,
                                    output_type=tf.int32)
    elif decoding_type == 'Sampling':
        # apply a Temperature
        if temperature != 1.0:
            next_token_logits = next_token_logits / temperature
        #filter to extract logits
        filtered_next_token_logits = tf_top_k_top_p_filtering(
            next_token_logits, top_k, top_p)
        next_token = tf.random.categorical(filtered_next_token_logits,
                                           dtype=tf.int32,
                                           num_samples=1)
    prediction = tokenizer.decode(next_token.numpy().tolist()[0])
    return prediction
```

As seen from the above function, we can generate the exact text by setting seed for random selection, which is performed by `tf.random.set_seed(60)`. Then, we have two options for predicting the following word: the greedy search method and several sampling options.

By using the `next_token_logits` from the previous example, we get the word "extend" as the next prediction for the below-given case:

```
prediction = get_prediction(next_token_logits,"Sampling", gpt2_small_model, gpt2_small_tokenizer,1, 0, 0.95)
print(prediction)
```

```
 extend
```

The following function is to get the filtered probabilities:

```
def filter_next_token_probabilities(sequence='Please input some text',
                                    decoding_type='Greedy Search',
                                    model=gpt2_small_model,
                                    tokenizer=gpt2_small_tokenizer,
                                    temperature=0.7,
                                    top_k=50,
                                    top_p=0.95):

    '''
    Get the probabilities for the next word options
    Greedy search -  returns the word with highest probability
    Sampling - sample based on filtering
    '''

    next_token_logits = get_next_token_logits(sequence, model, tokenizer)

    if decoding_type == 'Greedy Search':
        filtered_next_token_probabilities = tf.nn.softmax(next_token_logits)

    elif decoding_type == 'Sampling':
        # apply a Temperature
        if temperature != 1.0:
            next_token_logits = next_token_logits / temperature

        #convert logits to probabilities
        next_token_probabilities = tf.nn.softmax(next_token_logits)

        #filter to extract probabilities
        filtered_next_token_probabilities = pr_top_k_top_p_filtering(
            next_token_probabilities, top_k, top_p)

    return filtered_next_token_probabilities
```

As seen from the above functions `get_prediction` and `filter_next_token_probabilities`, there are duplications. It was done due to some challenges and getting cleaner and more self-explanatory code. Initially, these functions were combined and returned prediction, filtered next token logits as an output (later, those logits were converted to probabilities). By doing this, I realized that probability outputs were incorrect, as due to filtering, I was getting false conditional probabilities by passing filtered logits into the `softmax` function. So then decided to pass the logits to the `softmax` function and filter probabilities rather than the logits. But this time, while trying to sample among filtered probabilities, the function was unable to sample among the probabilities randomly. Instead, it tended to choose only the word with the highest probability. To overcome this, I checked to multiply the probability values to 10, 100, 1000, assuming that it may help to randomize the sampling, which ended in behaving like a greedy search.

`get_prediction` and `filter_next_token_probabilities` function, both have `top_k_top_p_filtering` functions, but one is `tf_top_k_top_p_filtering` and the other one is `pr_top_k_top_p_filtering`. `tf_top_k_top_p_filtering` is from Huggingface transformers library which filters based on the probabilities that takes logits as an input. `pr_top_k_top_p_filtering` is the adaptation of the `tf_top_k_top_p_filtering` that directly filters from probability inputs. The main difference is the application of `softmax` function before or inside the function after filtering the values.

The below function provides the details for top_k and top_p filtering:

```
#adapted from tf_top_k_top_p_filtering function to filter probabilities rathen than logits
#source - https://huggingface.co/transformers/v2.9.1/_modules/transformers/modeling_tf_utils.html
def pr_top_k_top_p_filtering(probabilities,
                             top_k=0,
                             top_p=1.0,
                             filter_value=-float("Inf"),
                             min_tokens_to_keep=1):
    """ Filter a distribution of probabilities using top-k and/or nucleus (top-p) filtering
        Args:
            probabilities: probabilities distribution shape (batch size, vocabulary size)
            if top_k > 0: keep only top k tokens with highest probability (top-k filtering).
            if top_p < 1.0: 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)
            Make sure we keep at least min_tokens_to_keep per batch example in the output
        From: https://gist.github.com/thomwolf/1a5a29f6962089e871b94cbd09daf317
    """
    probabilities_shape = shape_list(probabilities)

    if top_k > 0:
        top_k = min(max(top_k, min_tokens_to_keep),probabilities_shape[-1])  # Safety check
        # Remove all tokens with a probability less than the last token of the top-k
        indices_to_remove = probabilities < tf.math.top_k(probabilities, k=top_k)[0][..., -1,None]
        probabilities = set_tensor_by_indices_to_value(probabilities, indices_to_remove,filter_value)

    if top_p < 1.0:
        sorted_indices = tf.argsort(probabilities, direction="DESCENDING")
        sorted_probabilities = tf.gather(
            probabilities, sorted_indices, axis=-1, batch_dims=1
        )  # expects logits to be of dim (batch_size, vocab_size)

        cumulative_probs = tf.math.cumsum(sorted_probabilities, axis=-1)

        # Remove tokens with cumulative probability above the threshold (token with 0 are kept)
        sorted_indices_to_remove = cumulative_probs > top_p

        if min_tokens_to_keep > 1:
            # Keep at least min_tokens_to_keep (set to min_tokens_to_keep-1 because we add the first one below)
            sorted_indices_to_remove = tf.concat(
                [
                    tf.zeros_like(
                        sorted_indices_to_remove[:, :min_tokens_to_keep]),
                    sorted_indices_to_remove[:, min_tokens_to_keep:],
                ],
                -1,
            )
        # Shift the indices to the right to keep also the first token above the threshold
        sorted_indices_to_remove = tf.roll(sorted_indices_to_remove,1,axis=-1)
        sorted_indices_to_remove = tf.concat(
            [
                tf.zeros_like(sorted_indices_to_remove[:, :1]),
                sorted_indices_to_remove[:, 1:]
            ],
            -1,
        )
        # scatter sorted tensors to original indexing
        indices_to_remove = scatter_values_on_batch_indices(sorted_indices_to_remove, sorted_indices)
        probabilities = set_tensor_by_indices_to_value(probabilities, indices_to_remove,filter_value)
    return probabilities
```

As seen from the above function, there are certain helper functions. These functions were taken from the Huggingface transformers library without any change [5].

Below the helper functions for `pr_top_k_top_p_filtering` with all the required details:

```python
def shape_list(x):
    """Deal with dynamic shape in tensorflow cleanly."""
    static = x.shape.as_list()
    dynamic = tf.shape(x)
    return [dynamic[i] if s is None else s for i, s in enumerate(static)]


def set_tensor_by_indices_to_value(tensor, indices, value):
    # create value_tensor since tensor value assignment is not possible in TF
    value_tensor = tf.zeros_like(tensor) + value
    return tf.where(indices, value_tensor, tensor)


def scatter_values_on_batch_indices(values, batch_indices):
    shape = shape_list(batch_indices)
    # broadcast batch dim to shape
    broad_casted_batch_dims = tf.reshape(
        tf.broadcast_to(tf.expand_dims(tf.range(shape[0]), axis=-1), shape),
        [1, -1])
    # transform batch_indices to pair_indices
    pair_indices = tf.transpose(
        tf.concat(
            [broad_casted_batch_dims,
             tf.reshape(batch_indices, [1, -1])], 0))
    # scatter values to pair indices
    return tf.scatter_nd(pair_indices, tf.reshape(values, [-1]), shape)
```

We can use the previously obtained `next_token_logits` to get the `filtered_next_token_probabilities`:

```python
filtered_next_token_probabilities = filter_next_token_probabilities(next_token_logits, "Sampling",
                                                                    gpt2_small_model, gpt2_small_tokenizer,
                                                                    1, 0, 0.95)
print(filtered_next_token_probabilities)
```

executed in 51ms, finished 18:25:36 2021-12-18

```
tf.Tensor([[-inf -inf -inf ... -inf -inf -inf]], shape=(1, 50257), dtype=float32)
```

`filter_next_token_probabilities` function turned the logit values to probabilities by applying the softmax function and assigned "-inf" to the filtered values.

After extracting filtered next token probabilities, we need to get data for the probability plot, composed of the words/tokens and their respective probabilities.

```python
def get_plot_data(filtered_next_token_probabilities, tokenizer):
    """
    Get the list of words and probabilities for plotting
    """
    #convert -inf values to zero
    probabilities = tf.nn.relu(filtered_next_token_probabilities)
    #count non-zero values
    k = tf.math.count_nonzero(probabilities).numpy()
    #to limit the print barplot on screen to 100 words
    k = min(100, k)
    #extract top k probabilities
    filtered_probabilities_data = tf.math.top_k(probabilities[0], k)
    #convert probabilities to list
    filtered_probabilities = filtered_probabilities_data.values.numpy()
    probability_list = filtered_probabilities.tolist()
    #get list of words
    word_list = list()
    for i in filtered_probabilities_data.indices.numpy():
        word_list.append(tokenizer.decode([i]))
    return word_list, probability_list
```

In this function, we limit the number of tokens to be displaced to 100, as more than that is unreadable on the screen while plotting the data.

While running the application, sometimes it was not giving the plot data. While looking at the word and probability data, it was identified that the reason was duplications in the word list, which was due to decoding some unknown characters or punctuation marks. So, helper function was built to clean the repetitions:

```
def clean_plot_data(word_list, probability_list):
    """
    Prepares the data for plotting
    - Aggregates words that appear multiple times
    """
    result = {}
    for w, p in zip(word_list, probability_list):
        if w not in result:
            result[w] = p
        else:
            result[w] += p

    sorted_keys = sorted(result, key=result.get, reverse=True)
    result = {k: result[k] for k in sorted_keys}
    return list(result.keys()), list(result.values())
```

By running `get_plot_data` for `filtered_next_token_probabilities` that was obtained for previous example, we get below list of words and probabilities:

```
word_list, probability_list = get_plot_data(filtered_next_token_probabilities, gpt2_small_tokenizer)
print(word_list[:20])
print(probability_list[:20])
```

```
[' thank', ' say', ' ask', ' see', ' take', ' add', ' apologize', ' make', ' hear', ' remind', ' express', ' congratu
late', ' know', ' share', ' give', ' invite', ' offer', ' think', ' be', ' extend']
[0.2688286304473877, 0.037480223923921585, 0.030456334352493286, 0.029252052307128906, 0.01968269795179367, 0.0190704
3159008026, 0.017907360568642616, 0.01717122457921505, 0.017010578885674477, 0.015843752771615982, 0.0153940636664628
98, 0.014498946256935596, 0.014483246020972729, 0.014051360078155994, 0.013667550869228606, 0.012989435344934464, 0.01
1911231093108654, 0.011033246293663979, 0.010269062593579292, 0.009216371923685074]
```

Perplexity Calculation

Perplexity for the generated text can be calculated with the help of the below-given function based on the method discussed in the theoretical part:

```
def get_perplexity(sequence, gpt2_model, gpt2_tokenizer):
    input_ids = gpt2_tokenizer.encode(sequence, return_tensors="tf")
    loss = gpt2_model(input_ids=input_ids, labels=input_ids).loss
    perplexity = tf.math.exp(tf.math.reduce_mean(loss)).numpy()
    return perplexity

sequence = """Azerbaijan, the nation and former Soviet republic, is bounded by the Caspian Sea and\
Caucasus Mountains,which span Asia and Europe. Its capital, Baku, is famed for its medieval walled Inner City.\
Within the Inner City lies the Palace of the Shirvanshahs,\
a royal retreat dating to the 15th century, and the centuries-old stone Maiden Tower,\
which dominates the city skyline."""

perplexity_distilgpt2 = get_perplexity(sequence, gpt2_distil_model, gpt2_distil_tokenizer)
perplexity_gpt2_small = get_perplexity(sequence, gpt2_small_model, gpt2_small_tokenizer)
perplexity_gpt2_medium = get_perplexity(sequence, gpt2_medium_model, gpt2_medium_tokenizer)

print("perplexity for distilgpt2 = ", perplexity_distilgpt2)
print("perplexity for gpt2 (small) = ", perplexity_gpt2_small)
print("perplexity for gpt2_medium = ", perplexity_gpt2_medium)
```

```
perplexity for distilgpt2 =  44.382935
perplexity for gpt2 (small) =  30.905354
perplexity for gpt2_medium =  22.356934
```

In the above example, we calculate the perplexity for the text taken from Wikipedia. Note that, in the application, the perplexity is calculated for the generated text, which helps us to compare the decoding quality between different methods.

Main features of Panel Holoviz [6]

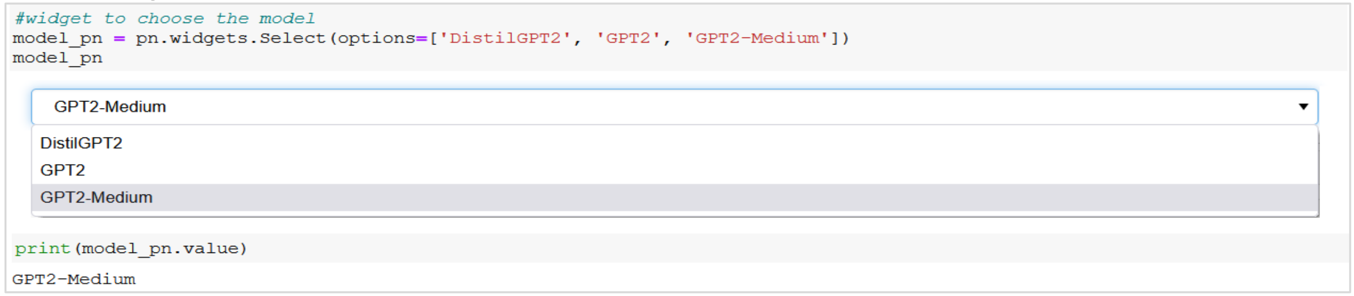
Panel allows adding interactivity to the code in jupyter notebooks. It enables building widgets and panes to interactively update the variable values to control the function outputs rather than manually changing the code to run the function with different values. There are several types of available widgets and panes. Here we discuss the ones that are to build the application.

Before going through the different components of the web application, we need to run `pn.extension()`, which is required to initialize the JavaScript in the notebook context. Below are the components of the web app to select a single value for different required parameters:

Widgets

- **Select**

`Select` widget was used to select the model:



```
#widget to choose the model
model_pn = pn.widgets.Select(options=['DistilGPT2', 'GPT2', 'GPT2-Medium'])
model_pn
```

GPT2-Medium ▾
DistilGPT2
GPT2
GPT2-Medium

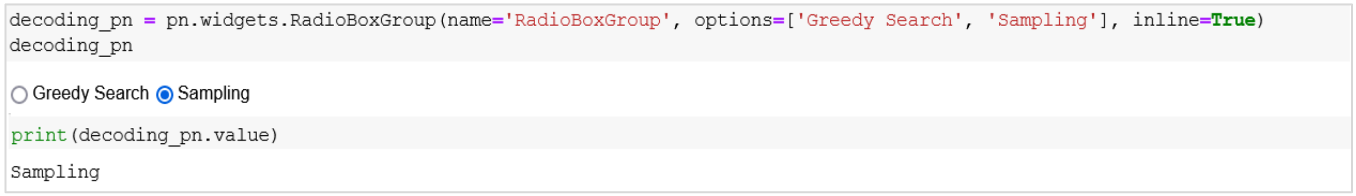
```
print(model_pn.value)
```

```
GPT2-Medium
```

We can access the selected model by `value`. For the example given above, the model chosen to run is GPT2-Medium.

- **RadioBoxGroup**

Another type of widget that allows selecting the value from the list or dictionary as Select is RadioBoxGroup. This button was used to choose the method of decoding.



```
decoding_pn = pn.widgets.RadioBoxGroup(name='RadioBoxGroup', options=['Greedy Search', 'Sampling'], inline=True)
decoding_pn
```

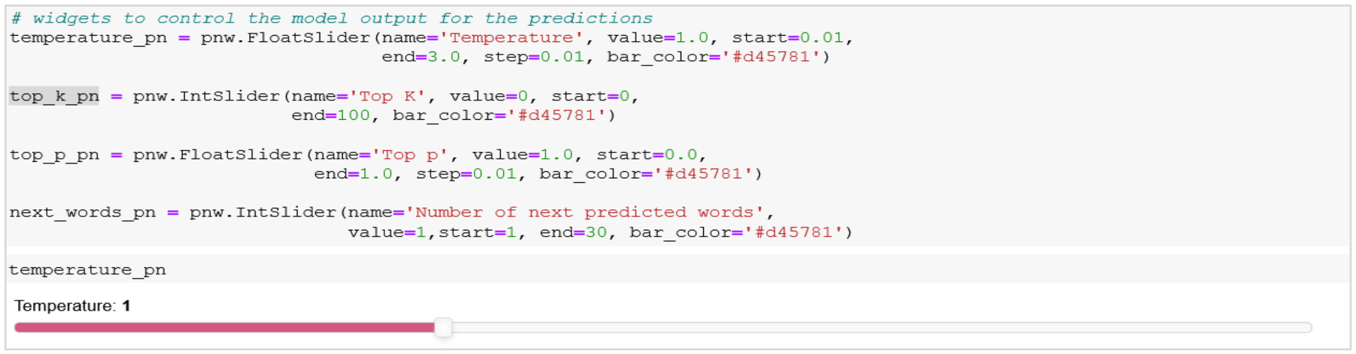
Greedy Search [✓ Sampling]

```
print(decoding_pn.value)
```

```
Sampling
```

- **Slider**

There are several types of sliders for setting numeric values. For example, float and integer sliders were used in the application to set the values for temperature, top-k, top-p, and repetition of the text generation:



```
# widgets to control the model output for the predictions
temperature_pn = pnw.FloatSlider(name='Temperature', value=1.0, start=0.01,
                                 end=3.0, step=0.01, bar_color='#d45781')

top_k_pn = pnw.IntSlider(name='Top K', value=0, start=0,
                         end=100, bar_color='#d45781')

top_p_pn = pnw.FloatSlider(name='Top p', value=1.0, start=0.0,
                           end=1.0, step=0.01, bar_color='#d45781')

next_words_pn = pnw.IntSlider(name='Number of next predicted words',
                              value=1,start=1, end=30, bar_color='#d45781')
```

```
temperature_pn
```

Temperature: **1**

Now we can go through the widget that allows us to trigger events, start, and run the required functions:

- **Button**

The Button widget allows triggering events when the button is clicked. There are several types of buttons that allow choosing the required theme. Button widget's **value** toggles from False to True while the event is processed.

```
button = pn.widgets.Button(name="Generate", button_type='primary')
button
```

We want to be able to trigger events when a button is clicked. For this purpose, we can use the **on_click** method, which allows starting function when a button is clicked:

```
dummy_button = pn.widgets.Button(name='Dummy button', button_type='success')
text = pn.widgets.TextInput(value='Start')

def b(event):
    text.value = 'Dummy button  clicked for {0} times'.format(dummy_button.clicks)

dummy_button.on_click(b)
pn.Row(dummy_button, text)
```

- **TextInput**

TextInput allows entering any string using a text input box. Like other widgets, the input text can be accessed by value parameter.

```
text_input = pn.widgets.TextInput(placeholder='Enter a string here...')
text_input
```

```
print(text_input.value)
```

```
I would like to
```

After going through the widgets to manipulate the values for the different variables, we got through the panes to handle text data.

Panes

- **Markdown**

Markdown pane renders any string containing valid Markdown, which can also be modified by using custom CSS styles.

```
header_parameters = pn.pane.Markdown("""#Parameters""")
header_parameters
```

executed in 32ms, finished 17:48:32 2021-12-19

Parameters

- **HTML**

HTML pane allows rendering any string containing valid HTML, which can also be modified by custom CSS styles. Furthermore, the string displayed by the HTML pane can be accessed by `object` parameter.

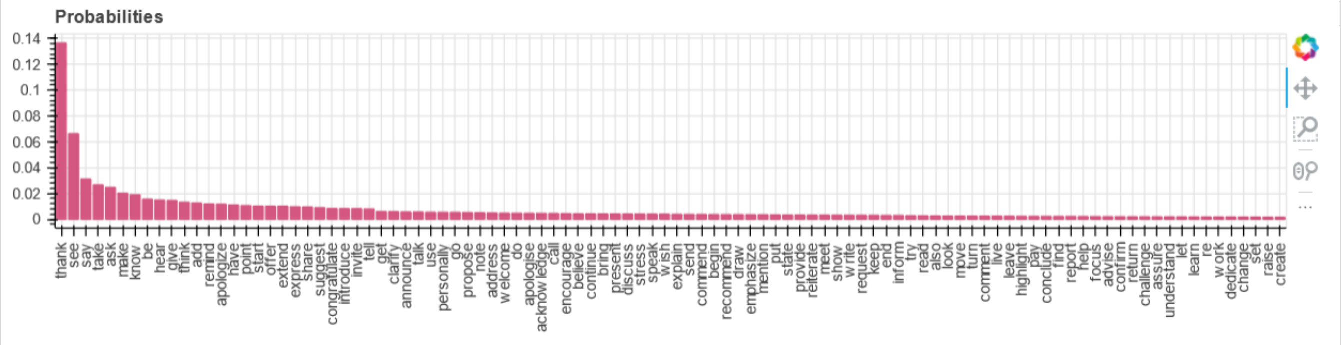
```
generated_text = pn.pane.HTML(object="Hello World!",
                              background='#f0f0f0',
                              min_height=200,
                              sizing_mode="stretch_width")

generated_text
```

```
Hello World!
```

- **Bokeh**

The Bokeh pane allows displaying any displayable Bokeh plots.

```
plot = get_plot(word_list, probability_list)
bokeh_plot = pn.pane.Bokeh(plot, sizing_mode="stretch_width")
bokeh_plot
```



The details of the `get_plot` function shown above are discussed separately in a later topic.

After going through the pane and widgets, we can discuss the functionality of that components. All those widgets can be shown separately or grouped as a set of widgets and displayed accordingly with the help of Panes.

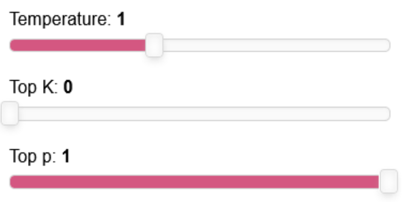
Panels

There are several types of panels that allow arranging widgets and panes. Here we will showcase only one of the panels used in the application.

- **Column**

Column panel allows arranging a list of components in a vertical direction.

```
params = pn.Column(temperature_pn,top_k_pn,top_p_pn)
params
```



Temperature: **1**

Top K: **0**

Top p: **1**

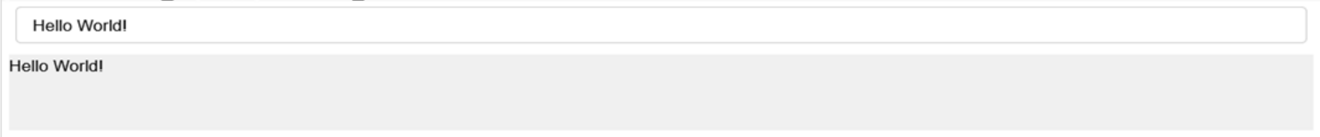
Links

Another essential feature of the Panel for adding functionality to the components are Links. First, we go through the link parameters used in the application. Link methods help generate links between different parts, which allows to manipulate and display the outputs dynamically.

- **`link` method**

In the application, **`the link`** method links the **`TextInput`** widget to the **`HTML`** pane object.

```
text_input = pn.widgets.TextInput(placeholder='Enter a string here...')
generated_text = pn.pane.HTML(object=text_input.value,
                              background='#f0f0f0',
                              min_height=200,
                              sizing_mode="stretch_width")

text_input.link(generated_text, value='object')

pn.Column(text_input, generated_text)
```


Hello World!

Hello World!

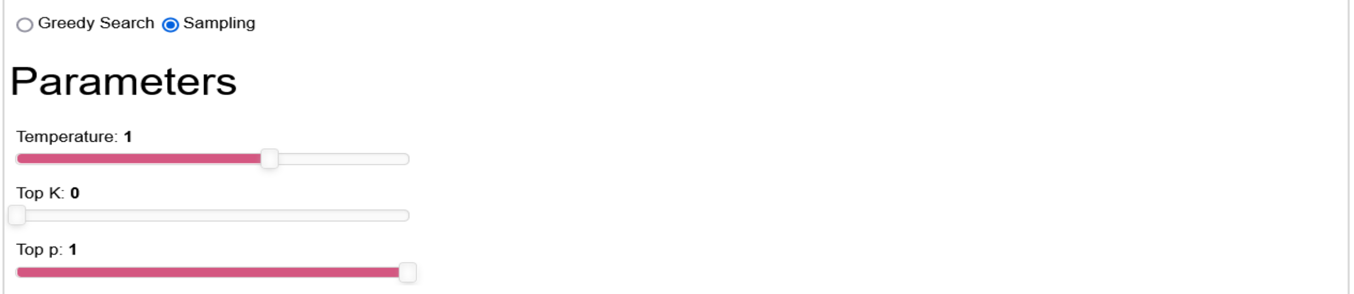
As seen in the above example, when we update the text in the **`text_input`** widget, it automatically updates the content of the **`generated_text`** pane.

- **`watch` method**

For explaining the **`watch`** method, we first need to understand **`Event`** objects. Event is used to signal the changes in parameters. **`Event`** objects provide several functional attributes for getting information about the event. For example, in the application, a new attribute is used, which helps to get the parameter's new value.

Watch method helps to ty a parameter to another one with the help of the Event object. For example, one of the cases that are used in the application is given below:

```
decoding_pn = pn.widgets.RadioBoxGroup(name='RadioBoxGroup', options=['Greedy Search', 'Sampling'], inline=True)
parameter_widgets = pn.Column()
#function to hide/show the parameters depending on the chosen decoding method
def hide_parameters(event):
    if event.new == 'Sampling':
        parameter_widgets.append(header_parameters)
        parameter_widgets.append(temperature_pn)
        parameter_widgets.append(top_k_pn)
        parameter_widgets.append(top_p_pn)
    else:
        parameter_widgets.remove(header_parameters)
        parameter_widgets.remove(temperature_pn)
        parameter_widgets.remove(top_k_pn)
        parameter_widgets.remove(top_p_pn)
#tying hide_parameters function to the decoding_pn
decoding_pn.param.watch(hide_parameters, 'value')
pn.Column(decoding_pn,parameter_widgets)
```


Greedy Search [✓ Sampling]

Parameters

Temperature: 1

Top K: 0

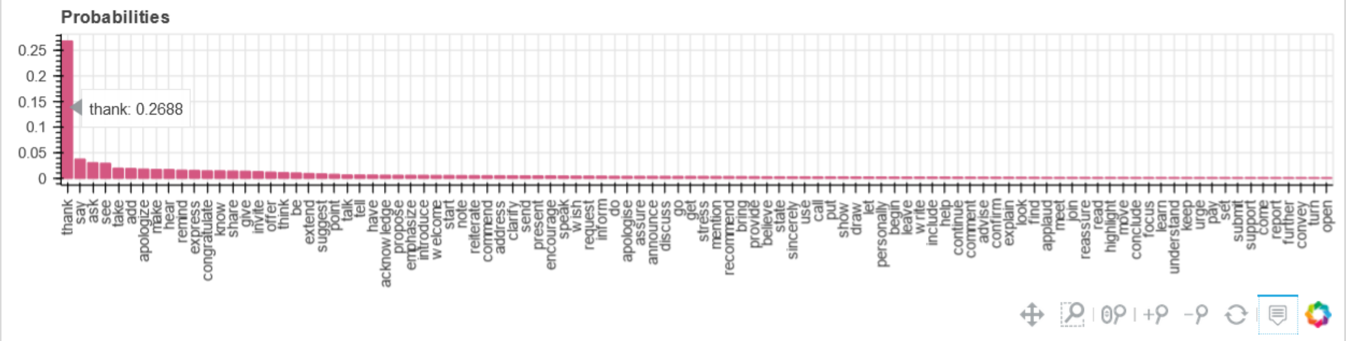
Top p: 1

When RadioBoxGroup widget **`decoding_pn`** value changes to "Sampling", the **`hide_parameters`** function triggers and adds the widgets to the **`parameter_widgets`** panel

Interactive probability plots

As mentioned earlier, we use Bokeh pane to display the Bokeh plot. The below-given function is used to get the plot data to pass to the Bokeh pane.

```python
def get_plot(word_list, probability_list):
    source = ColumnDataSource(
        data=dict(word_list=word_list, probability_list=probability_list))
    plot = figure(x_range=source.data['word_list'],
                  height=250,
                  title="Probabilities",
                  #define toolbar location
                  toolbar_location="below",
                  #add interactivity to the plot by different tools
                  tools="hover,pan,wheel_zoom,box_zoom,zoom_in,zoom_out,reset",
                  #define displayed text while hovering through the plot
                  tooltips="@word_list: @probability_list{0.0000}")

    plot.vbar(x='word_list',
              top='probability_list',
              width=0.8,
              color='#d45781',
              source=source)
    plot.xaxis.major_label_orientation = pi / 2
    return plot

plot = get_plot(word_list, probability_list)
bokeh_plot = pn.pane.Bokeh(plot, sizing_mode="stretch_width")
bokeh_plot
```



Now we can go through the components of the **`get_plot`** function to understand the working mechanism of Bokeh plots.

`ColumnDataSource` is the fundamental data structure used in Bokeh, enabling the plot to be interactively updated by updating source data. Moreover, it gives the ability to interact with the plot data using different tools.

Another essential part of Bokeh plots is the **`tools`** that allow interaction with plot data and changing the plot parameters. For example, when the hover tool is activated as shown in the above plot, we can read the data by hovering through the bars.

The **`tooltips`** parameter helps define the text that will be displayed during hovering through the plot. ColumnDataSource data are used to describe the data that will be displayed.

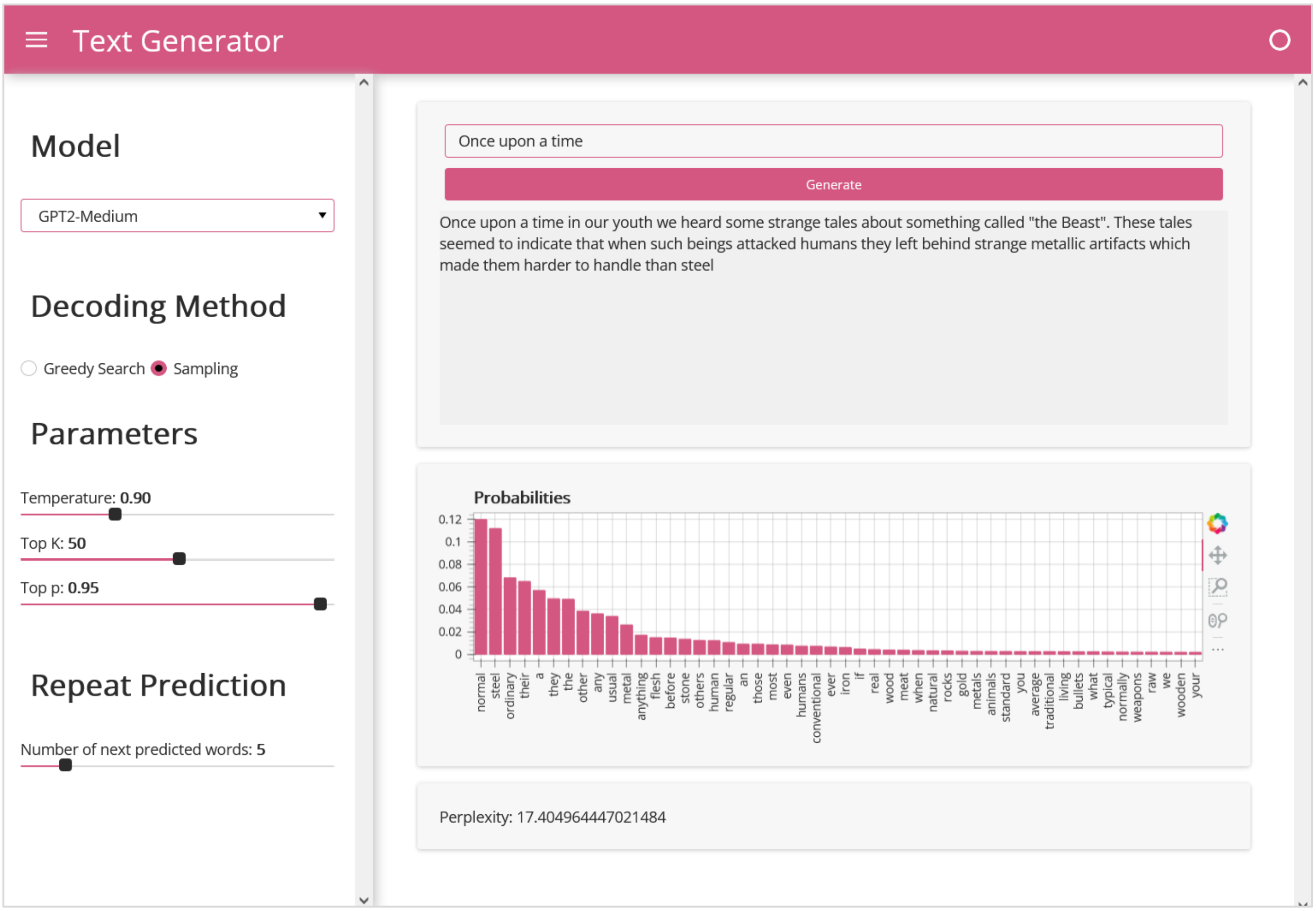
As the **`source`** parameter is defined in the **`plot.vbar()`**, we pass the data for the x and y-axis as the string, which are the names for the data in **`ColumnDataSource`**.

Web Application

For building the web application, we use Panel, which comes with ready templates in addition to all the interactivity that it provides for working within or separately from jupyter notebooks. The application can be directly run from jupyter notebooks or as a separate python file from localhost or remote server. The most beautiful part is that it requires a few lines of code to build the template for the application by using all previously mentioned functions, widgets, panes, and panels, as shown in below-given example:

```
template = pn.template.FastListTemplate(
    title="Text Generator",
    sidebar=[model_widget, decoding_widget, parameter_widgets, repeat_widget],
    main=[text_part, bokeh_plot, text],
    theme=DefaultTheme,
    theme_toggle=False,
    accent_base_color='#d45781',
    header_background='#d45781').show()
```

```
Launching server at http://localhost:61277
```



Text Generator

Model

GPT2-Medium

Decoding Method

Greedy Search [✓ Sampling]

Parameters

Temperature: 0.90

Top K: 50

Top p: 0.95

Repeat Prediction

Number of next predicted words: 5

Once upon a time

Generate

Once upon a time in our youth we heard some strange tales about something called "the Beast". These tales seemed to indicate that when such beings attacked humans they left behind strange metallic artifacts which made them harder to handle than steel

Perplexity: 17.404964447021484

3. Future Considerations

In future iterations of the application several items can be added:

- **Number and type of models** – Only 3 version of GPT-2 was used in this application.
- **n-gram limitation** - n-gram limitation can be added as an extra parameter which will avoid repetition that happen after certain length of text generation
- **Beam search method** – beam search method can be added to the application, but with respective visualizations. Need to be considered that even though text generation can be achieved easily by using beam search, the visualization will be very difficult to achieve.

4. Application

In practical part most of the code was covered and each component was explained separately. To get the code that was used in the project refer to the GitHub repository https://github.com/vusaleyvaz/text-generator-app .

Even though all the components of the Panel library that were used are discussed in the report to get detailed and extra information about Panel library visit official webpage [6].

5. References

[1] Holtzman, Ari, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. "The Curious Case of Neural Text Degeneration." In International Conference on Learning Representations. 2019 https://arxiv.org/pdf/1904.09751.pdf

[2] Angela Fan, Mike Lewis, and Yann Dauphin. "Hierarchical Neural Story Generation" 2018. https://arxiv.org/pdf/1805.04833.pdf

[3] Video by HuggingFace - What is perplexity. https://www.youtube.com/watch?v=NURcDHhYe98

[4] Details of the GPT2 models - https://huggingface.co/transformers/v2.2.0/pretrained_models.html

[5] Source code for tf_top_k_top_p_filtering function: https://huggingface.co/transformers/v2.9.1/_modules/transformers/modeling_tf_utils.html

[6] Panel library - https://panel.holoviz.org/user_guide/index.html

[7] Patrick von Platen. Blog post - How to generate text: using different decoding methods for language generation with Transformers - https://huggingface.co/blog/how-to-generate

[8] Lewis Tunstall, Leonardo von Verra and Thomas Wolf, (February 2022). Natural Language Processing with Transformers - https://learning.oreilly.com/library/view/natural-language-processing/9781098103231/