Project: Prompt Autocompletion

Project Overview

Prompt Autocompletion is mainly a text generation application. Text generation is an interesting method which leverages various deep learning algorithms to output a text given input as a text. It involves training these models on large datasets of text to learn patterns, grammar, and contextual information. These models then use this learned knowledge to generate new text based on given prompts or conditions.It is an extremely interesting and useful field of study that has many practical applications including but not limited to

- Content generation
- Conversational AI
- Paraphrasing
- Summarization

Problem Statement

One interesting tangential field that endlessly fascinates me is text-to-image models like stable diffusion, DALLE etc. A crucial aspect of the operation of those models is prompting. The right set of prompting often makes or breaks the end result. Also often times the general and non-technical people often cannot express their needs succinctly for those text-to-image generation models to work as expected leaving them with dissatisfied results.
In this project, I aim to reduce that pain point by building a text generation model that can generate high quality prompts, ready to be ingested by text-to-image models. More formally I aim to build a text generation model that given an input word or phrase, generates a detailed and relevant prompt to be used by text to image models like Stable Diffusion.
Example: if I input *woman portrait* it should generate a relevant prompt like:

Portrait of a beautiful woman with long hair on bicycle with a Vangog style.

Metrics

Since prompt autocompletion is mainly a text generation problem, there are two major metrics used in the project

1) Cross entropy loss - used to keep track of model performance during training
2) Perplexity - used to evaluate the performance of the model after training, the lower the better.
3) Human Evaluation: In addition, human judgments are essential for assessing aspects of text generation that are difficult to quantify with automated metrics. Evaluators may rate the generated text on criteria such as coherence, relevance, fluency, and overall quality. This subjective assessment helps capture nuances that automated metrics might miss.

Analysis

Data Exploration

Text generation models rely heavily on the input data used to train the model. Hence a quality dataset is paramount to the success of this task. Aside from being high quality, the dataset should also be sufficiently large to capture the entire diversity of related text of this domain.
I selected this https://www.kaggle.com/datasets/tanreinama/900k-diffusion-prompts-dataset publicly available dataset to train or finetune my system. I used a 1k subset of the dataset that is further available here https://huggingface.co/datasets/poloclub/diffusiondb for finetuning the transformer since I wanted to build a MVP with minimal data and it was already publicly available for easy access to integrate with huggingface ecosystem.

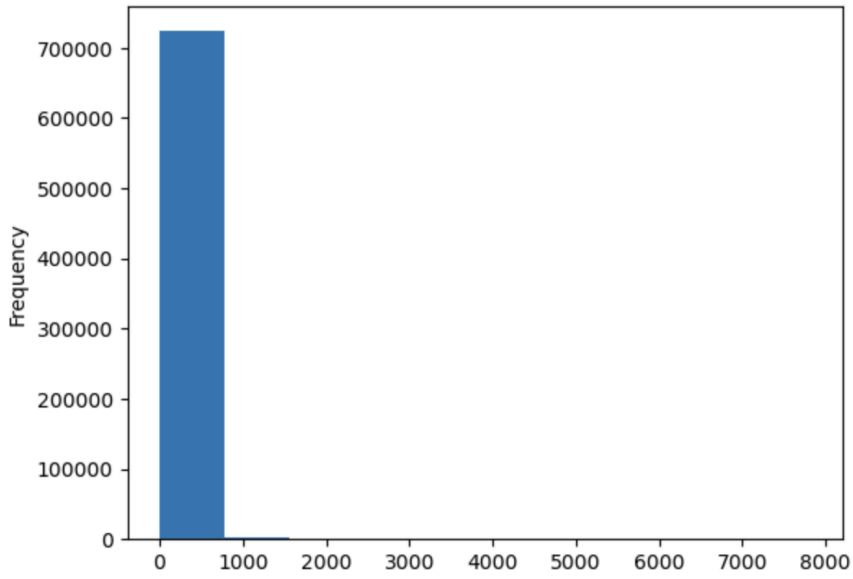
The dataset had 726362 prompt instances. I only needed 1k samples for a decept MVP development. A thorough data exploration is essential to understand the dataset's characteristics and to inform preprocessing and modeling decisions for developing the solution. In my case, analysing Text Length Distribution and Tokenization Patterns made sense since I want to perform language modeling.
I particularly analysed the **distribution of sentence** and **paragraph lengths**. This helps in setting appropriate sequence lengths for the model.

Text/ paragraph length distribution

```
In [12]: data["prompt_length"] = data['prompt'].apply(len)

In [16]: data["prompt_length"].plot.hist()

Out[16]: <Axes: ylabel='Frequency'>
```



This suggested that most of the prompts were of reasonable length. The mean length was 129 and there are cases where prompt length is 1 and data also has prompt in other format than string. So dataset needs cleaning.
I also analysed **word count** of each prompt

Word count

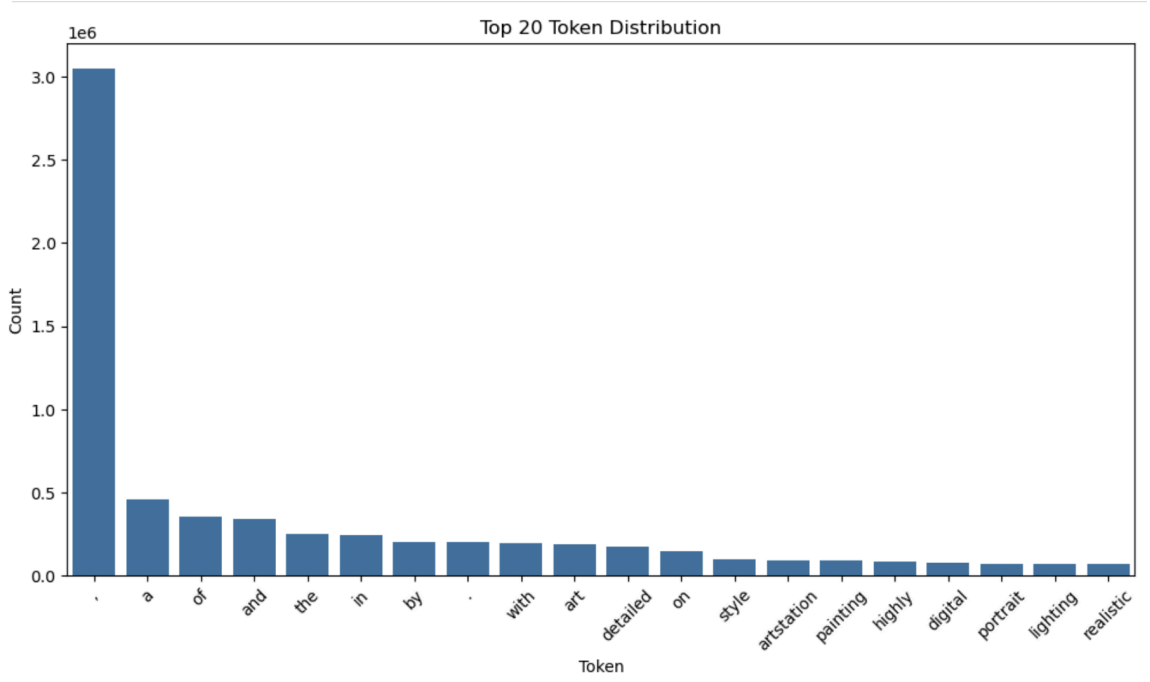
```
def word_count(text):
    # Split the text by whitespace and count the resulting list's length
    return len(text.split())

# Apply the function to the 'text' column and create a new column 'word_count'
data['word_count'] = data['prompt'].apply(word_count)
```

```
data["word_count"].describe()
```

```
count    726362.000000
mean         19.729982
std          21.468646
min           1.000000
25%           7.000000
50%          14.000000
75%          28.000000
max        1163.000000
Name: word_count, dtype: float64
```

This provided an idea of word distribution in the prompts.
Most importantly , I analyzed the **distribution of token lengths** to inform the choice of tokenization strategy (e.g., word-level vs. subword-level) by toekning the text using nltk library.



This gave me insight that using a word level tokenizer would be most helpful as there are important keywords like in the above diagram that make prompts rich.

Algorithms and techniques

As a starter task, training a word-level Long Short-Term Memory (LSTM) **[1]** network is particularly suited for text generation tasks due to its ability to capture and generate long-range dependencies in sequential data, which is crucial for producing coherent and contextually relevant text. Unlike character-level models that predict one character at a time, word-level LSTMs handle entire words as the basic unit of prediction, enabling them to understand and generate text at a more meaningful semantic level. This helps in preserving grammatical structures and contextual integrity across sentences and paragraphs. Furthermore, LSTMs are designed to manage the vanishing gradient problem, allowing them to maintain and utilize information from earlier parts of the text effectively. This is especially beneficial for text generation, where maintaining context over longer sequences is essential for producing logical and human-like text. Consequently, word-level LSTMs are powerful tools for creating applications that require sophisticated language modeling, such as automated story generation, dialogue systems, and content creation.

Based on train, evaluation and overall performance, an even better approach is finetuning a pretrained Transformer on our chosen dataset. Particularly, fine-tuning a distilroberta-base **[2]** model for text generation tasks offers several advantages over traditional models like word-level LSTMs, making it a superior choice for many applications. distilroberta-base, a distilled version of the Bidirectional Encoder Representations from Transformers (BERT), retains the powerful contextual understanding of its larger counterpart while being more efficient in terms of computational resources. This model is pre-trained on a vast corpus of text, enabling it to grasp intricate nuances and contextual relationships within the language, which are essential for generating coherent and contextually appropriate text. The fine-tuning process involves adapting this pre-trained model to specific tasks, allowing it to leverage its deep understanding of language patterns while honing in on the stylistic and contextual needs of the desired output. This approach significantly reduces the amount of task-specific data and training time required

compared to training a model from scratch. Additionally, distilroberta-base's transformer architecture excels at capturing long-range dependencies and understanding the bidirectional context of sentences, which is critical for maintaining coherence over longer passages of generated text. Its ability to process entire sentences at once, rather than sequentially like LSTMs, allows for more sophisticated handling of language structures and nuances. As a result, fine-tuning distilroberta-base not only leads to more contextually relevant and grammatically correct text but also achieves this with improved efficiency, making it an ideal solution for a wide range of text generation applications.

Masked language Modeling

In particular, the task is further called Masked Language Generation using transformer based models. Masked language modeling (MLM) with distilroberta-base for text generation leverages the model's ability to predict missing or masked words within a sentence, enhancing its understanding of context and improving its generative capabilities. distilroberta-base, a streamlined version of BERT, is trained on MLM tasks where some percentage of input tokens are randomly masked, and the model learns to predict these masked tokens based on their surrounding context. This pre-training approach allows distilroberta-base to develop a deep comprehension of language structure and semantics, making it adept at generating coherent and contextually relevant text. For text generation, fine-tuning distilroberta-base on specific datasets further refines its ability to produce text that adheres to desired styles and topics. During generation, techniques such as masked token prediction can be iteratively applied to fill in gaps within an initial text prompt, resulting in high-quality, contextually accurate completions. This process exploits distilroberta-base's bidirectional understanding of language, ensuring that generated text maintains logical flow and consistency. The efficiency of distilroberta-base, combined with its powerful contextual learning, makes it a compelling choice for text generation tasks where both performance and computational efficiency are paramount.

Benchmark

I chose an LSTM (Long Short-Term Memory) networks trained on words and sentences serve as a robust benchmark for text generation tasks due to their proven capability to model sequential data with long-range dependencies. Their architecture is specifically designed to address the limitations of traditional recurrent neural networks (RNNs), such as the vanishing gradient problem, which enables them to effectively retain and utilize information over extended text sequences. This makes LSTMs particularly adept at understanding context and maintaining coherence over longer passages, which is essential for generating fluent and meaningful text. Furthermore, LSTMs have been extensively studied and applied across various natural language processing tasks, leading to a wealth of research, optimized techniques, and best practices that enhance their performance. Their widespread adoption and success in text generation provide a solid baseline against which newer models can be compared. By serving as a benchmark, LSTMs offer a reliable measure of progress and innovation in the field, enabling researchers to evaluate improvements in model architecture, training techniques, and overall text generation quality. Therefore, despite the advent of more advanced models like transformers, LSTMs remain a fundamental and valuable reference point in the continuous evolution of text generation technologies.
For this task, several metrics are helpful for assessing the abilities of LSTM models as a benchmark in text generation tasks. These metrics evaluate various aspects of the generated text, including its fluency, coherence, and overall quality:

Further analysis

In my case, I trained a word-level LSTM as the benchmark model, since it is the simplest workable deep learning architecture that solves the problem.
The final cross-entropy loss for the training of this model after 30 epochs was 0.002 for training and a validation loss of 0.00. On a side note, this suggested overfitting

of the model to the dataset. The model performed poorly by not being able to generate any new sequence or words given some previous words or texts.
Which made me choose a more robust and powerful architecture of Transformers particularly a distilroberta-base model to fine-tune on the data at hand. The training loss of just 3 epochs were 1.64 for training loss and 1.34 for validation loss. The perplexity score for the model was 3.83.

Methodology

Data Preprocessing

Both training the LSTM and distilroberta-base model required some text preprocessing
For the LSTM the steps were as outlined below:
NOTE: All the below steps are outlined in the notebook folder called train_word_level_lstm_for_text_generation/lstm.ipynb in the project folder of github
1. **Data Collection**: Gather and compile the text data from various sources into a single dataset.
2. **Text Cleaning**: Remove unwanted characters, punctuation, and extra spaces. Normalize the text by converting it to lowercase.
3. **Tokenization**: Split the text into individual words (tokens).
4. **Vocabulary Creation**: Build a vocabulary of all unique words in the dataset.
5. **Mapping Tokens to Integers**: Create a mapping from each word to a unique integer (word-to-index mapping).
6. **Sequence Creation**: Convert the text into sequences of fixed length (n-grams), where each sequence is made up of integers representing the words.
7. I**nput and Output Separation**: For each sequence, define the input as all but the last word, and the output as the last word in the sequence.
8. **Padding Sequences**: Ensure all sequences are of equal length by padding shorter sequences with a special token or truncating longer ones.
9. **One-Hot Encoding**: Convert the output words into one-hot encoded vectors for use in the LSTM.

10. **Train-Validation Split**: Split the processed data into training and validation sets.
11. **Batch Preparation**: Organize the data into batches for efficient training.

Here are the data preprocessing steps needed for fine-tuning a distilroberta-base model for masked language modeling (MLM):

NOTE: All the mentioned steps are outlined in the project notebook

finetune_distilbert_for_text_generation/masked_language_modeling.ipynb

1. **Data Collection**: Gather and compile the text data from various sources into a single dataset.
2. **Text Cleaning**: Remove unwanted characters, punctuation, and extra spaces. Normalize the text by converting it to lowercase (if needed, depending on the model's requirements).
3. **Sentence Segmentation**: Split the text into individual sentences.
4. **Tokenization**: Use distilroberta-base's tokenizer to convert sentences into tokens. This includes splitting words and subwords and adding special tokens like [CLS] and [SEP].
5. **Masking Tokens**: Randomly mask a certain percentage of tokens in each sentence. Typically, 15% of the tokens are selected for masking. Replace these tokens with the [MASK] token, ensuring some tokens are left unchanged or replaced with random tokens as per BERT's original MLM strategy.
6. **Mapping Tokens to Integers**: Convert the tokens to their corresponding integer IDs using distilroberta-base's vocabulary.
7. **Input and Output Creation**: Prepare the input sequences with the masked tokens and the output sequences with the original tokens for the masked positions.
8. **Padding and Truncation**: Pad or truncate the sequences to a fixed length suitable for the model's maximum input size.
9. **Attention Masks**: Create attention masks to differentiate between padded and actual tokens in the sequences.
10. **Train-Validation Split**: Split the processed data into training and validation sets.

11. **Batch Preparation**: Organize the data into batches for efficient training, ensuring each batch contains sequences of equal length.

Implementation

Implementing a word-level LSTM for text generation involves several technical steps, beginning with data preprocessing. First, the raw text data is cleaned by removing unwanted characters and normalizing case. The text is then tokenized into words, and a vocabulary of unique words is created. Each word is mapped to a unique integer, and the text is converted into sequences of these integers. These sequences are typically of fixed length, achieved by padding shorter sequences and truncating longer ones.

Training procedure

The LSTM was trained for 30 epochs with Adam optimizer.
During model training, each sequence is divided into input and output pairs, where the input is a sequence of words and the output is the next word in the sequence. These outputs are one-hot encoded to create target vectors. The LSTM model is built using layers such as an embedding layer to transform integer-encoded words into dense vectors, followed by one or more LSTM layers to capture temporal dependencies, and a dense layer with a softmax activation function to predict the next word. The model is trained using an appropriate loss function, such as categorical cross-entropy, and an optimizer like Adam. Regularization techniques, such as dropout, may be applied to prevent overfitting. Finally, the trained model is used to generate text by predicting the next word iteratively, feeding the predicted word back into the model as input for subsequent predictions. Here is the detailed architecture of the model that I designed for the task

```
# 8. Model Building
class LSTMModel(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, output_size, num_layers=2, dropout=0.2):
        super(LSTMModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, num_layers=num_layers, dropout=dropout, batch_first=True)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.embedding(x)
        x, _ = self.lstm(x)
        x = self.dropout(x)
        x = self.fc(x[:, -1, :])
        return x
```

Here's a breakdown of its components and their roles:

1. **Embedding Layer**: The model begins with an embedding layer that transforms input words, represented as integer indices, into dense vectors of a specified size (embed_size). This layer helps in capturing the semantic meaning of the words.
2. **LSTM Layer**: Following the embedding layer, there is an LSTM layer with a specified number of hidden units (hidden_size). The LSTM can have multiple layers (num_layers), which enhances its ability to learn complex patterns in the sequential data. The dropout parameter introduces a dropout regularization technique to prevent overfitting by randomly setting a fraction of the input units to zero at each update during training time.
3. **Dropout Layer**: After the LSTM, there is a dropout layer that further applies dropout to the outputs of the LSTM layer. This additional dropout helps in regularizing the model and improving its generalization capability.
4. **Fully Connected (Linear) Layer**: The final layer is a fully connected (linear) layer that takes the hidden state output from the last time step of the LSTM and maps it to the output size, which corresponds to the vocabulary size. This layer is responsible for producing the final prediction for the next word in the sequence.
5. **Forward Pass**: In the forward pass, the input data (x) is first passed through the embedding layer, then through the LSTM layer where it processes the sequential data. The output from the LSTM is then regularized by the dropout layer. Finally, the output is passed through the fully connected layer to generate the prediction.

Refinement

As improvement, I chose a transformer based architecture for Masked Language Modeling. Fine-tuning a distilroberta-base model for masked language modeling (MLM) involves several key technical steps focused on adapting the pre-trained model to the specific characteristics of the target dataset. Initially, the pre-trained distilroberta-base model is loaded, including its tokenizer, which is essential for converting text into the appropriate input format. The architecture of distilroberta-base includes multiple transformer layers, each consisting of self-attention mechanisms and feed-forward neural networks, designed to capture bidirectional context. Here is the detailed architecture of my model

```
RobertaForMaskedLM(
  (roberta): RobertaModel(
    (embeddings): RobertaEmbeddings(
      (word_embeddings): Embedding(50265, 768, padding_idx=1)
      (position_embeddings): Embedding(514, 768, padding_idx=1)
      (token_type_embeddings): Embedding(1, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): RobertaEncoder(
      (layer): ModuleList(
        (0-5): 6 x RobertaLayer(
          (attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): RobertaIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): RobertaOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
  )
  (lm_head): RobertaLMHead(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (layer_norm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
    (decoder): Linear(in_features=768, out_features=50265, bias=True)
  )
)
```

During fine-tuning, a small percentage of tokens in the input text are masked randomly, and the model is trained to predict these masked tokens. This training process leverages the pre-trained weights, which already encode a broad

understanding of language, and further adjusts them based on the specific nuances of the new dataset. The training involves defining a loss function suitable for MLM, typically the cross-entropy loss, which measures the difference between the predicted token probabilities and the actual tokens. An optimizer, such as AdamW, is used to update the model weights. Fine-tuning typically involves a lower learning rate to avoid catastrophic forgetting, where the model could lose the pre-trained knowledge.

Training procedure

Fine tune the last few layers by freezing the upper layers

Training hyperparameters: The following hyperparameters were used during training:

- learning_rate: 2e-05
- train_batch_size: 8
- eval_batch_size: 8
- seed: 42
- optimizer: Adam with betas=(0.9,0.999) and epsilon=1e-08
- lr_scheduler_type: linear
- num_epochs: 3

The process also includes careful monitoring of the validation loss to prevent overfitting. Finally, the fine-tuned model is saved for downstream tasks, capable of generating contextually accurate text by filling in masked tokens based on its enhanced language understanding.

Results

Model Evaluation and validation

The final cross-entropy loss for the LSTM model after 30 epochs was 0.002 for the training set and 0.00 for the validation set. These results indicate significant overfitting, as evidenced by the model's inability to generate new sequences or

words based on provided input text. Here is a screenshot of the performance of the trained LSTM. Even though cross-entropy scores were good, it does not pass basic human evaluation metric.

```
# 10. Generate Text (example function)
def generate_text(model, seed_text, next_words, max_sequence_len, device):
    model.eval()
    words = seed_text.split()
    for _ in range(next_words):
        token_list = [word_to_idx[word] for word in words if word in word_to_idx]
        token_list = [0] * (max_sequence_len - len(token_list)) + token_list
        token_list = torch.tensor(token_list[-max_sequence_len:], dtype=torch.long).unsqueeze(0).to(device)
        with torch.no_grad():
            predicted = model(token_list)
            predicted_word_index = torch.argmax(predicted, axis=-1).item()
            if predicted_word_index == word_to_idx['<pad>']:
                break
            words.append(idx_to_word[predicted_word_index])
    return ' '.join(words)
```

```
print(generate_text(model, "Many friendly", 5, max_sequence_len, device))
```

```
Many friendly
```

Consequently, a more robust and powerful architecture was selected, specifically the DistilRoBERTa-base model, to fine-tune on the dataset. After just 3 epochs of training, the DistilRoBERTa model achieved a training loss of 1.64 and a validation loss of 1.34.

[513/513 01:34, Epoch 3/3]

Epoch	Training Loss	Validation Loss
1	No log	1.518225
2	No log	1.402045
3	1.641800	1.347369

```
3]: TrainOutput(global_step=513, training_loss=1.6369536773503175, metrics={'train_runtime': 94.8624, 'train_samples_
per_second': 43.041, 'train_steps_per_second': 5.408, 'total_flos': 135373709078784.0, 'train_loss': 1.6369536773
503175, 'epoch': 3.0})
```

Additionally, the model attained a perplexity score of 3.83, demonstrating its improved capability in generating coherent and contextually appropriate text.

Evaluate the trained model on Perplexity metric

```
1]: import math

eval_results = trainer.evaluate()
print(f"Perplexity: {math.exp(eval_results['eval_loss']):.2f}")
```

[45/45 00:02]

```
Perplexity: 3.83
```

The finetuned model also surpasses basic human evaluation
Here's a screeccap of using the finetuned model for actual prompt completion

Inference

Now that we've finetuned the model, you can use it for inference!. Let's come up with some text we'd like the model to fill in the blank with, and use the special `<mask>` token to indicate the blank:

```
text = "A portrait of <mask>"
```

The simplest way to try out your finetuned model for inference is to use it in a pipeline(). Instantiate a `pipeline` for fill-mask with your model, and pass your text to it. If you like, you can use the `top_k` parameter to specify how many predictions to return:

```
from transformers import pipeline

mask_filler = pipeline("fill-mask", "Shamima/diffusion_prompt")
mask_filler(text, top_k=3)
```

```
[{'score': 0.023939495906233788,
  'token': 10,
  'token_str': ' a',
  'sequence': 'A portrait of a'},
 {'score': 0.019102510064840317,
  'token': 2864,
  'token_str': ' herself',
  'sequence': 'A portrait of herself'},
 {'score': 0.01888210140168667,
  'token': 16423,
  'token_str': ' Hitler',
  'sequence': 'A portrait of Hitler'}]
```

Justification

The decision to use a fine-tuned DistilRoBERTa model for text generation is justified by its ability to efficiently capture and generate contextually rich and coherent text from the evaluated metrics above. DistilRoBERTa, a distilled version of the robust RoBERTa model, retains much of its parent model's power while being more computationally efficient. Fine-tuning this model on specific datasets enhances its understanding of the target domain's nuances, leading to more accurate and contextually relevant text generation. This approach leverages the sophisticated transformer architecture, which excels at understanding long-range dependencies and bidirectional context, making it highly suitable for producing high-quality text that maintains logical consistency and semantic depth.

NOTE: All the above mentioned steps from data collection to model evaluation for both the models are detailed as separate notebooks in the project folders in the git repo.

References

1. Hochreiter, Sepp & Schmidhuber, Jürgen. (1997). Long Short-term Memory. Neural computation. 9. 1735-80. 10.1162/neco.1997.9.8.1735.

2. Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). distilroberta-base, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.