Language modeling

0: Install dependencies:

You can only use the libraries imported for you in this assignment

```
!pip install transformers==4.24.0 datasets==2.7.0 tqdm==4.64.1 sentencepiece==0.1.97 gensim==4.2.0 apache-beam==2.42.0 sentence-transformers=
```

```
Requirement already satisfied: apache-beam==2.42.0 in /usr/local/lib/python3.8/dist-packages (2.42.0)
Requirement already satisfied: sentence-transformers==2.2.2 in /usr/local/lib/python3.8/dist-packages (2.2.2)
Requirement already satisfied: googledrivedownloader in /usr/local/lib/python3.8/dist-packages (0.4)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.8/dist-packages (from transformers==4.24.0) (2022.6.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.8/dist-packages (from transformers==4.24.0) (6.0)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.8/dist-packages (from transformers==4.24.0) (21.3)
Requirement already satisfied: filelock in /usr/local/lib/python3.8/dist-packages (from transformers==4.24.0) (3.8.0)
Requirement already satisfied: requests in /usr/local/lib/python3.8/dist-packages (from transformers==4.24.0) (2.28.1)
Requirement already satisfied: tokenizers!=0.11.3,<0.14,>=0.11.1 in /usr/local/lib/python3.8/dist-packages (from transformers==4.24.0
Requirement already satisfied: huggingface-hub<1.0,>=0.10.0 in /usr/local/lib/python3.8/dist-packages (from transformers==4.24.0) (0.
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.8/dist-packages (from transformers==4.24.0) (1.21.6)
Requirement already satisfied: multiprocess in /usr/local/lib/python3.8/dist-packages (from datasets==2.7.0) (0.70.9)
Requirement already satisfied: dill<0.3.7 in /usr/local/lib/python3.8/dist-packages (from datasets==2.7.0) (0.3.1.1)
Requirement already satisfied: pyarrow>=6.0.0 in /usr/local/lib/python3.8/dist-packages (from datasets==2.7.0) (7.0.0)
Requirement already satisfied: fsspec[http]>=2021.11.1 in /usr/local/lib/python3.8/dist-packages (from datasets==2.7.0) (2022.11.0)
Requirement already satisfied: pandas in /usr/local/lib/python3.8/dist-packages (from datasets==2.7.0) (1.3.5)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.8/dist-packages (from datasets==2.7.0) (3.8.3)
Requirement already satisfied: responses<0.19 in /usr/local/lib/python3.8/dist-packages (from datasets==2.7.0) (0.18.0)
Requirement already satisfied: xxhash in /usr/local/lib/python3.8/dist-packages (from datasets==2.7.0) (3.1.0)
Requirement already satisfied: smart-open>=1.8.1 in /usr/local/lib/python3.8/dist-packages (from gensim==4.2.0) (5.2.1)
Requirement already satisfied: scipy>=0.18.1 in /usr/local/lib/python3.8/dist-packages (from gensim==4.2.0) (1.7.3)
Requirement already satisfied: typing-extensions>=3.7.0 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (4.4.0)
Requirement already satisfied: zstandard<1,>=0.18.0 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (0.19.0)
Requirement already satisfied: grpcio!=1.48.0,<2,>=1.33.1 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (1.51.
Requirement already satisfied: orjson<4.0 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (3.8.3)
Requirement already satisfied: fastavro<2,>=0.23.6 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (1.7.0)
Requirement already satisfied: cloudpickle~=2.1.0 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (2.1.0)
Requirement already satisfied: python-dateutil<3,>=2.8.0 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (2.8.2)
Requirement already satisfied: crcmod<2.0,>=1.7 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (1.7)
Requirement already satisfied: hdfs<3.0.0,>=2.1.0 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (2.7.0)
Requirement already satisfied: protobuf<4,>=3.12.2 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (3.19.6)
Requirement already satisfied: pymongo<4.0.0,>=3.8.0 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (3.13.0)
Requirement already satisfied: pydot<2,>=1.2.0 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (1.3.0)
Requirement already satisfied: proto-plus<2,>=1.7.1 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (1.22.1)
Requirement already satisfied: httplib2<0.21.0,>=0.8 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (0.17.4)
Requirement already satisfied: pytz>=2018.3 in /usr/local/lib/python3.8/dist-packages (from apache-beam==2.42.0) (2022.6)
Requirement already satisfied: torchvision in /usr/local/lib/python3.8/dist-packages (from sentence-transformers==2.2.2) (0.14.0+cu11
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.8/dist-packages (from sentence-transformers==2.2.2) (1.0.2)
Requirement already satisfied: torch>=1.6.0 in /usr/local/lib/python3.8/dist-packages (from sentence-transformers==2.2.2) (1.13.0+cu1
Requirement already satisfied: nltk in /usr/local/lib/python3.8/dist-packages (from sentence-transformers==2.2.2) (3.7)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets==2.7.0) (1.3.3)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets==2.7.0) (1.3.1)
Requirement already satisfied: async-timeout<5.0,>=4.0.0a3 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets==2.7.0)
Requirement already satisfied: charset-normalizer<3.0,>=2.0 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets==2.7.0)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets==2.7.0) (22.1.0)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets==2.7.0) (6.0.3)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.8/dist-packages (from aiohttp->datasets==2.7.0) (1.8.2)
Requirement already satisfied: docopt in /usr/local/lib/python3.8/dist-packages (from hdfs<3.0.0,>=2.1.0->apache-beam==2.42.0) (0.6.2
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.8/dist-packages (from hdfs<3.0.0,>=2.1.0->apache-beam==2.42.0) (1
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.8/dist-packages (from packaging>=20.0->transformers
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.8/dist-packages (from requests->transformers==4.24.0) (20
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.8/dist-packages (from requests->transformers==4.24.0) (2.10)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.8/dist-packages (from requests->transformers==4.24.0)
Requirement already satisfied: joblib in /usr/local/lib/python3.8/dist-packages (from nltk->sentence-transformers==2.2.2) (1.2.0)
Requirement already satisfied: click in /usr/local/lib/python3.8/dist-packages (from nltk->sentence-transformers==2.2.2) (7.1.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.8/dist-packages (from scikit-learn->sentence-transforme
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.8/dist-packages (from torchvision->sentence-transforme
```

1: Introduction

This section give you a quick introduction/refresher to language models

What is a language model?

A language model is a statistical model that give you the probability of some given text

What is a token?

You can't find the probability of most sequences longer than a few words directly since the 26^N possible sequences of length N only including the lower case letters in the English alphabet. That number can become astronomically large quickly.

Solution: break the text up into small units (tokens)

Each token is typically a word or punctuation (but, can be other short sequences of characters)

Question: a) Finish the implementation exercises

Your first exercise is to create a tokenizer which take some text as input and outputs a list of tokens

To make things a little easier you can assume that all tokens are separated by " " or "-"

You may use the `re` module, but there are simpler solutions that do not need it

```
import re

# Basic tokenizer function
def tokenize(text: str) -> list:
  """
  Input: text, the string to be tokenized
  Output: tokens, a list of token strings

  Turns text into a list of tokens
  """
  # !!!!!!!!!!!!! Your code starts here !!!!!!!!!!!!!
  tokens = re.split(' |-',text)

  # !!!!!!!!!!!!! Your code ends here !!!!!!!!!!!!!
  return tokens

example_text = "Is water a Non-Newtonian fluid ?"
tokenized_example_text = tokenize(example_text)
print(tokenized_example_text)
# Expected output: ['Is', 'water', 'a', 'Non', 'Newtonian', 'fluid', '?']
```

```
['Is', 'water', 'a', 'Non', 'Newtonian', 'fluid', '?']
```

One of the simplest language models is the unigram model. It stores the probability of encountering each token, ignoring surrounding tokens(it does not use conditional probability):

$$P(sentence) = P(token_1)P(token_2)\dots P(token_N)$$

```
import numpy as np

class Unigram:

  def __init__(self):
    """
    Initializes log probabilities
    """
    self.log_probabilities = {}
    self.unknown_log_probability = 0.0

  def train(self, sentences: list)->None:
    """
    Input: sentences, list of already tokenized sentences
    Ex. [['Hello','my','name','is','HAL'],['Hi','HAL']]

    Save log probability of seeing each token using `np.log` to obtain the log probabilities

    """
    # Add a single unknown token
    sentences.append(['<unknown token>'])
    # !!!!!!!!!!!!! Your code starts here !!!!!!!!!!!!!
    len = 0
    unique_tokens = set(x for l in sentences for x in l)
    self.log_probabilities = dict.fromkeys(unique_tokens, 0)
    for i in sentences:
      for j in i:
        self.log_probabilities[j] += 1
        len += 1
```

```
    for key in self.log_probabilities:
      self.log_probabilities[key]  = np.log(self.log_probabilities[key]/len)

    # !!!!!!!!!!!!! Your code ends here !!!!!!!!!!!!!
    # Assign probability for unseen tokens
    self.unknown_log_probability = self.log_probabilities.pop('<unknown token>')

  def token_log_prob(self, token:str) -> float:
    """
    Get the log probability of a single token with self.unknown_log_probability use if a token was not found during training
    """
    # !!!!!!!!!!!!! Your code starts here !!!!!!!!!!!!!

    if token in self.log_probabilities:
      return self.log_probabilities[token]
    else:
      return self.unknown_log_probability

    # !!!!!!!!!!!!! Your code ends here !!!!!!!!!!!!!

  def sentence_log_prob(self, sentence:list) -> float:
    """
    Get the log probability of an already tokenized sentence
    """
    # !!!!!!!!!!!!! Your code starts here !!!!!!!!!!!!!
    probability = 0

    for token in sentence:
      probability += self.token_log_prob(token)

    return probability

    # !!!!!!!!!!!!! Your code ends here !!!!!!!!!!!!!

model = Unigram()
model.train([['Hello','my','name','is','HAL'],['Hi','HAL']])
print('"Hello" log prob:',model.token_log_prob('Hello'))
print('"Hi my name is HAL" log prob:',model.sentence_log_prob(tokenize("Hi my name is HAL")))
```

```
"Hello" log prob: -2.0794415416798357
"Hi my name is HAL" log prob: -9.704060527839234
```

We can use the Unigram model to classify text (but, may not have the highest accuracy)

```
import pandas as pd

from google_drive_downloader import GoogleDriveDownloader as gdd

gdd.download_file_from_google_drive(
    file_id='1hyziAb67N3RuhhBSCtBhpQEPgr6ZIpJP',
    dest_path='/tmp/emotion.csv'
)
df_full = pd.read_csv('/tmp/emotion.csv')
df_train = df_full[df_full['subset']=='train'].copy()
df_test = df_full[df_full['subset']=='test'].copy()
label_key = ["sadness", "joy", "love", "anger", "fear", "surprise"]

# Init models
total_count = len(df_train)
label_counts = df_train['label'].value_counts().sort_index()
models = [{
     'index': i,
     'label': label,
     'log_prior': np.log(label_counts.iloc[i]/total_count),
     'unigram_model': Unigram(),
} for i, label in enumerate(label_key)]

# Train models
for model in models:
  df_train_matching_label = df_train[df_train['label']==model['index']]
  tokenized_sentences = df_train_matching_label['text'].apply(tokenize).tolist()
  model['unigram_model'].train(tokenized_sentences)
```

```
# Predict classes
def predict(sentence:str)->int:
  tokenized_sentence = tokenize(sentence)
  highest_log_prob = float('-inf')
  highest_log_prob_index = 0
  for model in models:
    # !!!!!!!!!!!!! Your code starts here !!!!!!!!!!!!!
    # Compute log prob of the sentence using the unigram model + the log prior of the label

    log_prob = model['unigram_model'].sentence_log_prob(tokenized_sentence) + model['log_prior']

    # !!!!!!!!!!!!! Your code ends here !!!!!!!!!!!!!
    if log_prob > highest_log_prob:
      highest_log_prob = log_prob
      highest_log_prob_index = model['index']
  return highest_log_prob_index

df_test['predicted_label'] = df_test['text'].apply(predict)

tp_count = sum(df_test['predicted_label']==df_test['label'])
accuracy = tp_count/len(df_test)
print(f'Accuracy: {accuracy*100}%')
```

```
Accuracy: 62.3%
```

2: Types of Language Models

This sections explains different types of language models. We will go over 3 of the most used language model types:

1. Causal
2. Masked
3. Sequence to sequence

2.1: Causal language model

A causal language model provides the probability of a token given the tokens before it

$P(token_T|token_1, token_2, \ldots, token_{T-1})$

It is useful for a variety of NLP tasking including sequence generation and sequence classification

Example: Hello, my name is ...

Output: Hello, my name is HAL

```
from transformers import OpenAIGPTTokenizer, OpenAIGPTLMHeadModel, set_seed
from tqdm import tqdm

import torch
import torch.nn as nn

tokenizer = OpenAIGPTTokenizer.from_pretrained("openai-gpt")
model = OpenAIGPTLMHeadModel.from_pretrained("openai-gpt")
```

```
ftfy or spacy is not installed using BERT BasicTokenizer instead of SpaCy & ftfy.
Some weights of OpenAIGPTLMHeadModel were not initialized from the model checkpoint at openai-gpt and are newly initialized: ['lm_head.w
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

Here is an example

```
def generate_gpt_text_greedy(input_text,sequence_max_length = 25):
  """
  This uses greedy decoding which is not optimal for most tasks,
  but requires little system resources and is simple to implement

  Recommended for deterministic tasks: Beam search
  Recommended for creative tasks: Nucleus sampling
  Recommended for low latency/real time tasks: Greedy decoding or Nucleus sampling

  Note: Optimal decoding/generation algorithm depends on the task
  """
```

```
  generated_text = input_text

  # Generate a sequence
  for i in tqdm(range(sequence_max_length)):
    with torch.no_grad(): # Better preformance
      inputs = tokenizer(generated_text, return_tensors="pt")
      outputs = model(**inputs)
      next_token_logits = outputs.logits[0, -1, :]
      next_token_index = torch.argmax(next_token_logits)
      generated_text = tokenizer.decode(
          torch.cat((inputs['input_ids'][0],torch.tensor([next_token_index]))) # generated_text = generated_text + new_token
      )

  return generated_text

input_text = "Hello, my name is John Smith. I am the"

print(f'Generated: {generate_gpt_text_greedy(input_text)}')
```

```
100%|██████████| 25/25 [00:04<00:00,  6.16it/s]Generated: hello, my name is john smith. i am the head of the department of defense. "
```

```
alt_text = "John said it again three times: \"No No No"
print(f'Generated: {generate_gpt_text_greedy(alt_text)}')
```

```
100%|██████████| 25/25 [00:07<00:00,  3.35it/s]Generated: john said it again three times : " no no no no no no no no no no no no no no r
```

Question: b)

i) What do you notice about the generated text?

The word "no" is repeated for the rest of the sentence. As we are working with a causal model which uses the probabilities of the previous tokens, so the probability of the word that has the highest number of repeats will be higher than the others (and by trial and error, I found that the last words of the sentence are more important). On top of that, we are using a greedy policy that picks the most probable word each time. As a result, at each time step we get a "no".

ii) How can this be avoided? As it is stated in the previous block, Nucleus sampling is a better choice for creative tasks. Methods that do not choose the highest probability and allow for exploration, will result better.

Question: c) (*CMPUT 566 Students Only*) Implement the Nucleas Sampling method described in Section 3.1 of https://arxiv.org/pdf/1904.09751.pdf. You can use any `torch` or `torch.nn` (`nn`) functions

Hint: Use `torch.multinomial` for sampling

```
softmax = nn.Softmax(dim=0)

def generate_gpt_text_nucleus_sampling(input_text, sequence_max_length = 25, p=0.9):
  """
  This uses greedy decoding which is not optimal for most tasks,
  but requires little system resources and is simple to implement

  Recommended for deterministic tasks: Beam search
  Recommended for creative tasks: Nucleus sampling
  Recommended for low latency/real time tasks: Greedy decoding or Nucleus sampling

  Note: Optimal decoding/generation algorithm depends on the task
  """
  generated_text = input_text

  # Generate a sequence
  for i in tqdm(range(sequence_max_length)):
    with torch.no_grad(): # Better preformance
      inputs = tokenizer(generated_text, return_tensors="pt")
      outputs = model(**inputs)
      # !!!!!!!!!!!!! Your code starts here !!!!!!!!!!!!!

      next_token_logits_probs = softmax(outputs.logits[0, -1, :])
      sorted_next_token_logits_probs, idx = torch.sort(next_token_logits_probs, descending=True)

      nucleus = sorted_next_token_logits_probs.cumsum(dim=0) < p
      nucleus[(~nucleus).nonzero()[0].item()] = True
      nucleus_probs = sorted_next_token_logits_probs[nucleus]
```

```
      next_index = torch.multinomial(nucleus_probs, num_samples=1)
      next_token_index = idx[next_index].item()

      # !!!!!!!!!!!!! Your code ends here !!!!!!!!!!!!!
      generated_text = tokenizer.decode(
          torch.cat((inputs['input_ids'][0],torch.tensor([next_token_index]))) # generated_text = generated_text + new_token
      )

  return generated_text


torch.manual_seed(314159)
input_text = "Hello, my name is John Smith. I am the"

print(f'Generated: {generate_gpt_text_nucleus_sampling(input_text)}')
```

```
100%|██████████| 25/25 [00:04<00:00,  5.00it/s]Generated: hello, my name is john smith. i am the physician in charge of the civil rights
```

```
inputs = tokenizer("Hello, my name is John Smith. I am the", return_tensors="pt")
torch.cat((inputs['input_ids'][0],torch.tensor([1000])))
```

```
tensor([3570,  240,  547, 1362,  544, 2476, 4920,  239,  249, 1048,  481, 1000])
```

```
inputs['input_ids']
```

```
tensor([[3570,  240,  547, 1362,  544, 2476, 4920,  239,  249, 1048,  481]])
```

2.2: Masked language models

A masked language model provides the probability of a token given the tokens before it and after it (fill in the blanks)

$P(token_T|token_1, \ldots, token_{T-1}, token_{T+1}, \ldots, token_N)$

It is useful for a variety of NLP tasking including sequence classification and grammar correction

Example: Hello, my name is ...

Output: Hello, my name is HAL

```
from transformers import BertTokenizer, BertForMaskedLM

import torch

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertForMaskedLM.from_pretrained("bert-base-uncased")
```

```
Downloading: 100%  232k/232k [00:00<00:00, 348kB/s]
Downloading: 100%  28.0/28.0 [00:00<00:00, 850B/s]
Downloading: 100%  570/570 [00:00<00:00, 15.9kB/s]
Downloading: 100%  440M/440M [00:08<00:00, 54.9MB/s]
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertForMaskec
- This IS expected if you are initializing BertForMaskedLM from the checkpoint of a model trained on ar
- This IS NOT expected if you are initializing BertForMaskedLM from the checkpoint of a model that you
```

```
example_text = "The capital of Alberta is [MASK]."

def predict_mask(input_text):

  with torch.no_grad():
    inputs = tokenizer(input_text, return_tensors="pt")
    logits = model(**inputs).logits
    mask_token_index = (inputs.input_ids == tokenizer.mask_token_id)[0].nonzero(as_tuple=True)[0]
    predicted_token_id = logits[0, mask_token_index].argmax(axis=-1)
  return tokenizer.decode(predicted_token_id)

print('Input: ',example_text)
print('Mask prediction: ',predict_mask(example_text))
```

```
Input:  The capital of Alberta is [MASK].
Mask prediction:  edmonton
```

Question: d) Use the `predict_mask` function and the `[MASK]` token to exract a fact from the language model(similar to the example above). Include your input and the model's prediction in your pdf report

```
your_prompt = "The temperature in winter is [MASK]."

print('Input: ',your_prompt)
print('Mask prediction: ',predict_mask(your_prompt))
```

```
Input:  The temperature in winter is [MASK].
Mask prediction:  freezing
```

2.3: Sequence to sequence models

A sequence to sequence models provides the probability of a token given the tokens before it and all tokens in another related sequence

It is useful for a variety of NLP tasking including translation and summarization (primarily used for text generation)

Example: Bonjour, je m'appelle HAL (French)

Output: Hello, my name is HAL (English)

```
from transformers import T5Tokenizer, T5ForConditionalGeneration
from tqdm import tqdm

tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = T5ForConditionalGeneration.from_pretrained("t5-small")
```

```
Downloading: 100%  792k/792k [00:00<00:00, 644kB/s]
Downloading: 100%  1.20k/1.20k [00:00<00:00, 34.4kB/s]
/usr/local/lib/python3.8/dist-packages/transformers/models/t5/tokenization_t5.py:164: FutureWarning: Th
For now, this behavior is kept to avoid breaking backwards compatibility when padding/encoding with `tr
- Be aware that you SHOULD NOT rely on t5-small automatically truncating your input to 512 when padding
- If you want to encode/pad to sequences longer than 512 you can either instantiate this tokenizer with
- To avoid this warning, please instantiate this tokenizer with `model_max_length` set to your preferre
  warnings.warn(
Downloading: 100%  242M/242M [00:07<00:00, 34.0MB/s]
```

```
def t5_summarize(text, max_length=20):
  # inference
  input_ids = tokenizer(
      f"summarize: {text}", return_tensors="pt"
  ).input_ids
  outputs = model.generate(
      input_ids,
      max_length=max_length,
  )
  return tokenizer.decode(outputs[0], skip_special_tokens=True)
# The Road Not Taken
# Poem by Robert Frost (1916)
# (Public Domain)
poem = """Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that the passing there
Had worn them really about the same,

And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way,
I doubted if I should ever come back.
```

```
I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I—
I took the one less traveled by,
And that has made all the difference."""

print('Summary: ',t5_summarize(poem))
```

```
Summary:  two roads diverged in a yellow wood, and I took the one less traveled by
```

The metaphors are lost on the model, but it still does a fairly good job summarizing the literal meaning of the poem

Question: e)

i) Find a short piece of text (article, poem, section of a paper) and get the model to summarize it. Include the summary in your report

```
short_text = "For example, consider factory workers at several different locations who are monitored to count the number of errors that occur

print('Summary: ',t5_summarize(short_text,max_length=34)) # You can change `max_length` if summary seems truncated
```

```
Summary:  factory workers at several different locations are monitored to count errors. if one location is monitored more stringently or
```

ii) Is the summary accurate? If yes, explain why the summary is accurate? If not, explain how the summary could be improved

Answer - It somehow has summarized the gicen text. But the words are exactly copied from the text, and the max_length argument only cuts the sentence and doesn't enforce the model to summarize more.

I think it could be improved if some words or phrases could be substituted with shorter paraphrases, and mabe even the structure of the sentence could be changed. It might be a good idea to extact the most important words and try to make a sentence that makes sense.