CSE 256: NLP UCSD, Programming Assignment 4

Text Decoding From GPT-2 using Beam Search (40 points)

Due: Friday, Dec 2, 2024

IMPORTANT: After copying this notebook to your Google Drive, paste a link to it below. To get a publicly-accessible link, click the *Share* button at the top right, then click "Get shareable link" and copy the link.

Link: paste your link here: https://colab.research.google.com/drive/1o7BTXM4j4g4t4VoYa3bokMYGPGBKaC01

Notes:

Make sure to save the notebook as you go along.

Submission instructions are located at the bottom of the notebook.

Part 0: Setup

Adding a hardware accelerator

Go to the menu and add a GPU as follows:

```
Edit > Notebook Settings > Hardware accelerator > (GPU)
```

Run the following cell to confirm that the GPU is detected.

```
import torch

# Confirm that the GPU is detected
assert torch.cuda.is_available()

# Get the GPU device name.
device_name = torch.cuda.get_device_name()
n_gpu = torch.cuda.device_count()
print(f"Found device: {device_name}, n_gpu: {n_gpu}")
```

```
Found device: Tesla T4, n_gpu: 1
```

Installing Hugging Face's Transformers and Additional Libraries

We will use Hugging Face's Transformers (https://github.com/huggingface/transformers).

Run the following cell to install Hugging Face's Transformers library and some other useful tools.

```
pip install -q sentence-transformers==2.2.2 transformers==4.17.0
```

```
      ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 86.0/86.0 kB 4.5 MB/s eta 0:00:
  Preparing metadata (setup.py) ... done
      ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 67.9/67.9 kB 4.3 MB/s eta 0:00:
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.8/3.8 MB 37.0 MB/s eta 0:00:00
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 897.5/897.5 kB 31.5 MB/s eta 0:00
  Building wheel for sentence-transformers (setup.py) ... done
```

Part 1. Beam Search

We are going to explore decoding from a pretrained GPT-2 model using beam search. Run the below cell to set up some beam search utilities.

```
from transformers import GPT2LMHeadModel, GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2LMHeadModel.from_pretrained("gpt2", pad_token_id=tokenizer.eos_token_

# Beam Search
```

```
def init_beam_search(model, input_ids, num_beams):
    assert len(input_ids.shape) == 2
    beam_scores = torch.zeros(num_beams, dtype=torch.float32, device=model.device
    beam_scores[1:] = -1e9 # Break ties in first round.
    new_input_ids = input_ids.repeat_interleave(num_beams, dim=0).to(model.device
    return new_input_ids, beam_scores

def run_beam_search_(model, tokenizer, input_text, num_beams=5, num_decode_steps=

    # Tokenize
    input_ids = tokenizer.encode(input_text, return_tensors='pt')

    #
    input_ids, beam_scores = init_beam_search(model, input_ids, num_beams)

    token_scores = beam_scores.clone().view(num_beams, 1)

    model_kwargs = {}
    for i in range(num_decode_steps):
        model_inputs = model.prepare_inputs_for_generation(input_ids, **model_kwa
        outputs = model(**model_inputs, return_dict=True)
        next_token_logits = outputs.logits[:, -1, :]
        vocab_size = next_token_logits.shape[-1]
        this_token_scores = torch.log_softmax(next_token_logits, -1)

        # Process token scores.
        processed_token_scores = this_token_scores
        for processor in score_processors:
            processed_token_scores = processor(input_ids, processed_token_scores)

        # Update beam scores.
        next_token_scores = processed_token_scores + beam_scores.unsqueeze(-1)

        # Reshape for beam-search.
        next_token_scores = next_token_scores.view(num_beams * vocab_size)

        # Find top-scoring beams.
        next_token_scores, next_tokens = torch.topk(
            next_token_scores, num_beams, dim=0, largest=True, sorted=True
        )

        # Transform tokens since we reshaped earlier.
        next_indices = torch.div(next_tokens, vocab_size, rounding_mode="floor")
        next_tokens = next_tokens % vocab_size
        # Update tokens.
        input_ids = torch.cat([input_ids[next_indices, :], next_tokens.unsqueeze(

        # Update beam scores.
        beam_scores = next_token_scores

        # Update token scores.

        # UNCOMMENT: To use original scores instead.
        # token_scores = torch.cat([token_scores[next_indices, :], this_token_sco
        token_scores = torch.cat([token_scores[next_indices, :], processed_token_

        # Update hidden state.
        model_kwargs = model._update_model_kwargs_for_generation(outputs, model_k
        model_kwargs["past"] = model._reorder_cache(model_kwargs["past"], next_in

    def transfer(x):
      return x.cpu() if to_cpu else x

    return {
        "output_ids": transfer(input_ids),
        "beam_scores": transfer(beam_scores),
        "token_scores": transfer(token_scores)
    }


def run_beam_search(*args, **kwargs):
    with torch.inference_mode():
        return run_beam_search_(*args, **kwargs)


# Add support for colored printing and plotting.

from rich import print as rich_print

import numpy as np

import matplotlib
from matplotlib import pyplot as plt
from matplotlib import cm

RICH_x = np.linspace(0.0, 1.0, 50)
RICH_rgb = (matplotlib.colormaps.get_cmap(plt.get_cmap('RdYlBu'))(RICH_x)[:, :3]
def print_with_probs(words, probs, prefix=None):
  def fmt(x, p, is_first=False):
    ix = int(p * RICH_rgb.shape[0])
    r, g, b = RICH_rgb[ix]
    if is_first:
      return f'[bold rgb(0,0,0) on rgb({r},{g},{b})]{x}'
    else:
      return f'[bold rgb(0,0,0) on rgb({r},{g},{b})] {x}'
  output = []
  if prefix is not None:
    output.append(prefix)
  for i, (x, p) in enumerate(zip(words, probs)):
    output.append(fmt(x, p, is_first=i == 0))
  rich_print(''.join(output))

# DEMO

# Show range of colors.

for i in range(RICH_rgb.shape[0]):
  r, g, b = RICH_rgb[i]
  rich_print(f'[bold rgb(0,0,0) on rgb({r},{g},{b})]hello world rgb({r},{g},{b})'

# Example with words and probabilities.

words = ['the', 'brown', 'fox']
probs = [0.14, 0.83, 0.5]
print_with_probs(words, probs)
```

```
Downloading: 100%                                      0.99M/0.99M [00:00<00:00, 2.41MB/s]

Downloading: 100%                                      446k/446k [00:00<00:00, 2.05MB/s]

Downloading: 100%                                      26.0/26.0 [00:00<00:00, 362B/s]

Downloading: 100%                                      665/665 [00:00<00:00, 15.3kB/s]

Downloading: 100%                                      523M/523M [00:11<00:00, 33.8MB/s]
/usr/local/lib/python3.10/dist-packages/transformers/modeling_utils.py:1439: F
  state_dict = torch.load(resolved_archive_file, map_location="cpu")
hello world rgb(215,49,39)
hello world rgb(244,111,68)
hello world rgb(253,176,99)
hello world rgb(254,226,147)
hello world rgb(251,253,196)
hello world rgb(217,239,246)
hello world rgb(163,210,229)
hello world rgb(108,164,204)
the brown fox
```

Question 1.1 (5 points)

Run the cell below. It produces a sequence of tokens using beam search and the provided prefix.

```
num_beams = 5
num_decode_steps = 10
input_text = 'The brown fox jumps'

beam_output = run_beam_search(model, tokenizer, input_text, num_beams=num_beams,
for i, tokens in enumerate(beam_output['output_ids']):
    score = beam_output['beam_scores'][i]
    print(i, round(score.item() / tokens.shape[-1], 3), tokenizer.decode(tokens,
```

```
0 -1.106 The brown fox jumps out of the fox's mouth, and the fox
1 -1.168 The brown fox jumps out of the fox's cage, and the fox
2 -1.182 The brown fox jumps out of the fox's mouth and starts to run
3 -1.192 The brown fox jumps out of the fox's mouth and begins to lick
4 -1.199 The brown fox jumps out of the fox's mouth and begins to bite
```

To get you more acquainted with the code, let's do a simple exercise first. Write your own code in the cell below to generate 3 tokens with a beam size of 4, and then print out the **third most probable** output sequence found during the search. Use the same prefix as above.

```
input_text = 'The brown fox jumps'

beam_output = run_beam_search(model, tokenizer, input_text, num_beams=4, num_deco

for i, tokens in enumerate(beam_output['output_ids']):
    score = beam_output['beam_scores'][i]
    print(i, round(score.item() / tokens.shape[-1], 3), tokenizer.decode(tokens,

tokens = beam_output['output_ids'][2]
score = beam_output['beam_scores'][2]
print("\nThird Most Probable Output - score :", round(score.item() / tokens.shape
```

```
0 -0.424 The brown fox jumps out of the
1 -0.579 The brown fox jumps out of his
2 -0.627 The brown fox jumps up and down
3 -0.744 The brown fox jumps out of her

Third Most Probable Output - score : -0.627 , Sequence : The brown fox jumps u
```

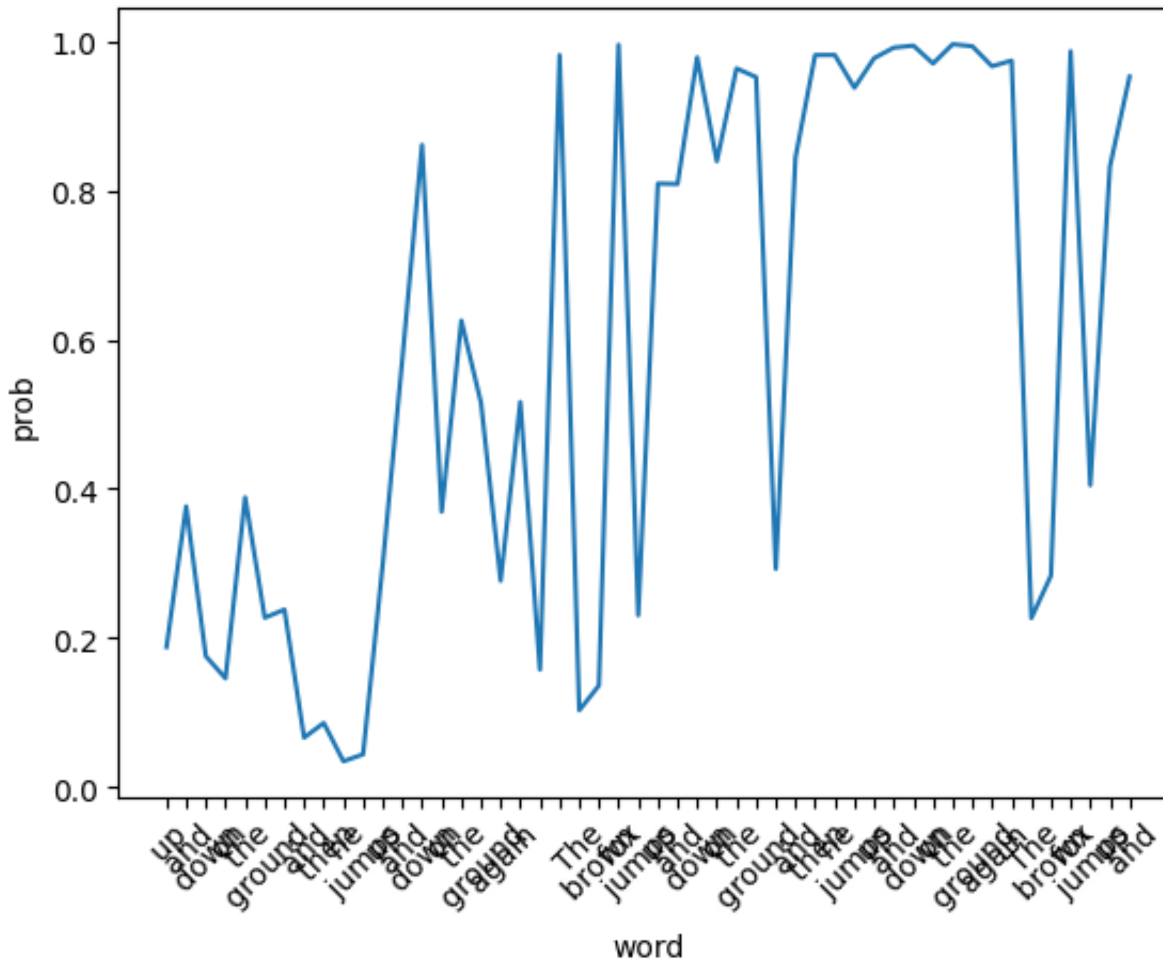
Question 1.2 (5 points)

Run the cell below to visualize the probabilities the model assigns for each generated word when using beam search with beam size 1 (i.e., greedy decoding).

```
input_text = 'The brown fox jumps'
beam_output = run_beam_search(model, tokenizer, input_text, num_beams=1, num_deco
probs = beam_output['token_scores'][0, 1:].exp()
output_subwords = [tokenizer.decode(tok, skip_special_tokens=True) for tok in bea

print('Visualization with plot:')

fig, ax = plt.subplots()
plt.plot(range(len(probs)), probs)
ax.set_xticks(range(len(probs)))
ax.set_xticklabels(output_subwords[-len(probs):], rotation = 45)
```

```
plt.xlabel('word')
plt.ylabel('prob')
plt.show()

print('Visualization with colored text (red for lower probability, and blue for h

print_with_probs(output_subwords[-len(probs):], probs, ' '.join(output_subwords[:
```

```
Visualization with plot:
```



```
Visualization with colored text (red for lower probability, and blue for highe
The  brown  fox  jumps up  and  down  on  the  ground ,  and  then  he  jumps
again .

 The  brown  fox  jumps  up  and  down  on  the  ground ,  and  then  he  jum
again .  The  brown  fox  jumps  up  and
```

Why does the model assign higher probability to tokens generated later than to tokens generated earlier?

Write your answer here

-> The model assigns higher probabilities to later tokens primarily due to the increasing context available during the generation process.

-> As more words are generated, the model has more information to work with, which helps narrow down the possible valid continuations based on both grammatical and semantic constraints.

-> This is similar to how humans become more confident in predicting the next word as a sentence progresses - early in a sentence, there are many possible directions it could take, but later on, the options become more limited by the established context.

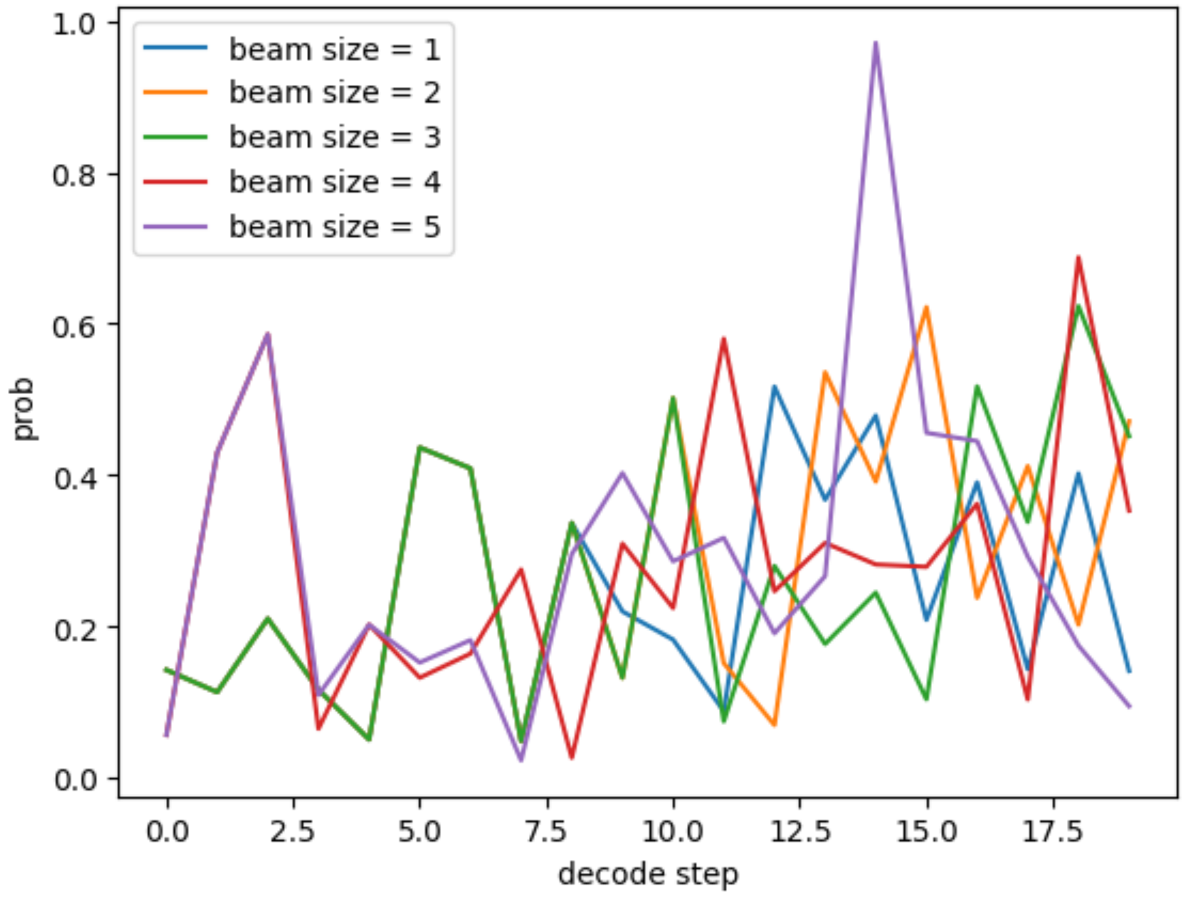
-> Additionally, since the model must distribute probability across all possible tokens at each step, having fewer plausible options (as is common later in a sequence) naturally leads to higher individual probabilities for those likely continuations.

Run the cell below to visualize the word probabilities when using different beam sizes.

```
input_text = 'Once upon a time, in a barn near a farm house,'
num_decode_steps = 20
model.cuda()

beam_size_list = [1, 2, 3, 4, 5]
output_list = []
probs_list = []
for bm in beam_size_list:
  beam_output = run_beam_search(model, tokenizer, input_text, num_beams=bm, num_d
  output_list.append(beam_output)
  probs = beam_output['token_scores'][0, 1:].exp()
  probs_list.append((bm, probs))

print('Visualization with plot:')
fig, ax = plt.subplots()
for bm, probs in probs_list:
  plt.plot(range(len(probs)), probs, label=f'beam size = {bm}')
plt.xlabel('decode step')
plt.ylabel('prob')
plt.legend(loc='best')
plt.show()
```

```
print('Model predictions:')
for bm, beam_output in zip(beam_size_list, output_list):
  tokens = beam_output['output_ids'][0]
  print(bm, beam_output['beam_scores'][0].item() / tokens.shape[-1], tokenizer.de
```

```
Visualization with plot:
```



```
Model predictions:
1 -0.9706197796445905 Once upon a time, in a barn near a farm house, a young b
2 -0.9286185177889738 Once upon a time, in a barn near a farm house, a young b
3 -0.9597569667931759 Once upon a time, in a barn near a farm house, a young b
4 -0.9205132108746152 Once upon a time, in a barn near a farm house, there was
5 -0.9058780670166016 Once upon a time, in a barn near a farm house, there was
```

Question 1.3 (10 points)

Beam search often results in repetition in the predicted tokens. In the following cell we pass a score processor called `WordBlock` to `run_beam_search`. At each time step, it reduces the probability for any previously seen word so that it is not generated again.

Run the cell to see how the output of beam search changes with and without using `WordBlock`.

```
import collections

class WordBlock:
    def __call__(self, input_ids, scores):
        for batch_idx in range(input_ids.shape[0]):
            for x in input_ids[batch_idx].tolist():
                scores[batch_idx, x] = -1e9
        return scores

input_text = 'Once upon a time, in a barn near a farm house,'
num_beams = 1

print('Beam Search')
beam_output = run_beam_search(model, tokenizer, input_text, num_beams=num_beams,
print(tokenizer.decode(beam_output['output_ids'][0], skip_special_tokens=True))

print('Beam Search w/ Word Block')
beam_output = run_beam_search(model, tokenizer, input_text, num_beams=num_beams,
print(tokenizer.decode(beam_output['output_ids'][0], skip_special_tokens=True))
```

```
Beam Search
Once upon a time, in a barn near a farm house, a young boy was playing with a
Beam Search w/ Word Block
Once upon a time, in a barn near a farm house, the young girl was playing wit
```

Is `WordBlock` a practical way to prevent repetition in beam search? What (if anything) could go wrong when using `WordBlock`?

Write your answer here

-> WordBlock is not a practical solution for preventing repetition in beam search, despite its effectiveness at eliminating exact word repetition. While it successfully prevents the repetitive loops seen in the first example, it creates new problems by being overly aggressive in blocking words.

-> Common words like articles ("the", "a"), prepositions ("in", "of"), and pronouns need to appear multiple times in natural text, but WordBlock prevents this entirely.

-> Additionally, some words legitimately need to be repeated for emphasis or clarity, or may have different meanings in different contexts, but WordBlock's blanket blocking approach doesn't account for these cases.

-> Also, WordBlock would be particularly problematic when writing in-depth about specific topics since key terminology often needs to be repeated for clarity and precision.

Question 1.4 (20 points)

Use the previous `WordBlock` example to write a new score processor called `BeamBlock`. Instead of uni-grams, your implementation should prevent tri-grams from appearing more than once in the sequence.

Note: This technique is called "beam blocking" and is described here (section 2.5). Also, for this assignment you do not need to re-normalize your output distribution after masking values, although typically re-normalization is done.

Write your code in the indicated section in the below cell.

```
import collections
import textwrap

class BeamBlock:
    def __call__(self, sequence_batch, token_scores):
        for batch_index in range(sequence_batch.shape[0]):
            # Convert sequence to list for current batch
            token_sequence = sequence_batch[batch_index].tolist()

            # Get existing trigrams from current sequence
            seen_trigrams = set()
            for pos in range(len(token_sequence) - 2):
                current_trigram = tuple(token_sequence[pos:pos+3])
                seen_trigrams.add(current_trigram)

            # Only proceed if we have at least 2 tokens to form a new trigram
            if len(token_sequence) >= 2:
                # Get the last two tokens
                previous_tokens = token_sequence[-2:]

                # For each possible next token
                for candidate_token in range(token_scores.shape[1]):
                    # Create potential trigram with this token
                    candidate_trigram = tuple(previous_tokens + [candidate_token]

                    # If this trigram already exists, block it
                    if candidate_trigram in seen_trigrams:
                        token_scores[batch_index, candidate_token] = -1e9

        return token_scores

input_text = 'Once upon a time, in a barn near a farm house,'
num_beams = 1

print('Beam Search')
beam_output = run_beam_search(model, tokenizer, input_text, num_beams=num_beams,
wrapped_text = textwrap.fill(tokenizer.decode(beam_output['output_ids'][0], skip_
print(wrapped_text)

print('\nBeam Search w/ Beam Block')
beam_output = run_beam_search(model, tokenizer, input_text, num_beams=num_beams,
wrapped_text = textwrap.fill(tokenizer.decode(beam_output['output_ids'][0], skip_
print(wrapped_text)
```

```
Beam Search
Once upon a time, in a barn near a farm house, a young boy was playing
with a stick. He was playing with a stick, and the boy was playing
with a stick. The boy was playing with a stick, and the boy was
playing with a

Beam Search w/ Beam Block
Once upon a time, in a barn near a farm house, a young boy was playing
with a stick. He was playing on the stick, and the boy was trying to
get a ball. The boy was holding the stick in his hand, and he was
trying
```

Submission Instructions

---

1. Click the Save button at the top of the Jupyter Notebook.
2. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
3. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
4. Once you've rerun everything, convert the notebook to PDF, you can use tools such as nbconvert, which requires first downloading the ipynb to your local machine, and then running "nbconvert" . (If you have trouble using nbconvert, you can also save the webpage as pdf. Make sure all your solutions are displayed in the pdf, it's okay if the provided codes get cut off because lines are not wrapped in code cells).
5. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing your graders will see!
6. Submit your PDF on Gradescope,

Acknowledgements

This assignment is based on an assignment developed by Mohit Iyyer