Homework 0, CS685 Spring 2022

This is due on February 4th, 2022, submitted via Gradescope as a PDF (File>Print>Save as PDF). 100 points total.

IMPORTANT: After copying this notebook to your Google Drive, please paste a link to it below. To get a publicly-accessible link, hit the *Share* button at the top right, then click "Get shareable link" and copy over the result. If you fail to do this, you will receive no credit for this homework!

LINK: paste your link here

How to do this problem set:

- Some questions require writing Python code and computing results, and the rest of them have written answers. For coding problems, you will have to fill out all code blocks that say `YOUR CODE HERE`.
- For text-based answers, you should replace the text that says "Write your answer here..." with your actual answer.
- This assignment is designed so that you can run all cells almost instantly. If it is taking longer than that, you have made a mistake in your code.

How to submit this problem set:

- Write all the answers in this Colab notebook. Once you are finished, generate a PDF via (File -> Print -> Save as PDF) and upload it to Gradescope.
- **Important:** check your PDF before you submit to Gradescope to make sure it exported correctly. If Colab gets confused about your syntax, it will sometimes terminate the PDF creation routine early.
- **Important:** on Gradescope, please make sure that you tag each page with the corresponding question(s). This makes it significantly easier for our graders to grade submissions. We may take off points for submissions that are not tagged.
- When creating your final version of the PDF to hand in, please do a fresh restart and execute every cell in order. Then you'll be sure it's actually right. One handy way to do this is by clicking `Runtime -> Run All` in the notebook menu.

Academic honesty

- We will audit the Colab notebooks from a set number of students, chosen at random. The audits will check that the code you wrote actually generates the answers in your PDF. If you turn in correct answers on your PDF without code that actually generates those answers, we will consider this a serious case of cheating. See the course page for honesty policies.

- We will also run automatic checks of Colab notebooks for plagiarism. Copying code from others is also considered a serious case of cheating.

Question 1.1 (10 points)

Let's begin with a quick probability review. In the task of language modeling, we're interested in computing the **joint** probability of some text. Say we have a sentence s with n words ($w_1, w_2, w_3, \ldots, w_n$) and we want to compute the joint probability $P(w_1, w_2, w_3, \ldots, w_n)$. Assume we are given a model that produces the conditional probability of the next word in a sentence given all preceding words: $P(w_i | w_1, w_2, \ldots, w_{i-1})$. How can we use this model to compute the joint probability of sentence s?

Write your answer here! Please include an equation(s) formatted in LaTeX in your answer. You can add LaTeX to your answer by wrapping it in $ signs; see the above cell for examples.

We know that
$P(w1, w2, w3..., wn) = P(w1|w2, w3, . . , wn - 1)P(w2|w3, . . . , wn - 2). . . P(wn)$

We're given that we know $P(wi|w1, w2, w3, . . . , wn - 1)$ for $i\epsilon n$, Hence we can calculate the equation.

Question 1.2 (10 points)

Why would we ever want to compute the joint probability of a sentence? Provide **two** different reasons why this probability might be useful to solve an NLP task.

Write your answer here! Please keep it brief (i.e., 2-3 sentences).

Probability of a Sentence: We want to compute probability of a sentence containing specific words happening. So we can have joint probability of all the words.

N-Gram Language Model: Task of predicting the next word given the previous words. We can use conditional probability for this.

Question 1.3 (5 points)

Here is a simple way to build a language model: for any prefix $w_1, w_2, \ldots, w_{i-1}$, retrieve all occurrences of that prefix in some huge text corpus (such as the Common Crawl) and keep count of the word w_i that follows each occurrence. I can then use this to estimate the conditional probability $P(w_i | w_1, w_2, \ldots, w_{i-1})$ for any prefix. Explain why this method is completely impractical!

Write your answer here! Please keep it brief (i.e., 2-3 sentences).

It becomes computationally cumbersome if we consider we have a huge corpus. Additionally we can't keep track of the long-distance dependincies.

Question 2.1 (5 points)

Let's switch over to coding! The below coding cell contains the opening paragraph of Daphne du Maurier's novel *Rebecca*. Write some code in this cell to compute the number of unique word **types** and total word **tokens** in this paragraph (watch the lecture videos if you're confused about what these terms mean!). Use a whitespace tokenizer to separate words (i.e., split the string on white space using Python's split function). Be sure that the cell's output is visible in the PDF file you turn in on Gradescope.

```
paragraph = '''Last night I dreamed I went to Manderley again. It seemed to me
that I was passing through the iron gates that led to the driveway.
The drive was just a narrow track now, its stony surface covered
with grass and weeds. Sometimes, when I thought I had lost it, it
would appear again, beneath a fallen tree or beyond a muddy pool
formed by the winter rains. The trees had thrown out new
low branches which stretched across my way. I came to the house
suddenly, and stood there with my heart beating fast and tears
filling my eyes.'''.lower() # lowercase normalization is often useful in NLP

types = 0
tokens = 0

# YOUR CODE HERE! POPULATE THE types AND tokens VARIABLES WITH THE CORRECT VALUES!
tokenized = paragraph.split()
types = len(list(set(tokenized)))
tokens = len(tokenized)
# DO NOT MODIFY THE BELOW LINE!
print('Number of word types: %d, number of word tokens:%d' % (types, tokens))
```

```
    Number of word types: 76, number of word tokens:100
```

Question 2.2 (5 points)

Now let's look at the most frequently used word **types** in this paragraph. Write some code in the below cell to print out the ten most frequently-occurring types. We have initialized a Counter object that you should use for this purpose. In general, Counters are very useful for text processing in Python.

```
from collections import Counter
c = Counter(tokenized)
# DO NOT MODIFY THE BELOW LINES!
for word, count in c.most_common()[:10]:
    print(word, count)
```

```
i 6
the 6
to 4
a 3
and 3
my 3
it 2
that 2
was 2
with 2
```

Question 2.3 (5 points)

What do you notice about these words and their linguistic functions (i.e., parts-of-speech)? These words are known as "stopwords" in NLP and are often removed from the text before any computational modeling is done. Why do you think that is?

Write your answer here! Please keep it brief (i.e., 2-3 sentences).

They're most frequent types of words belong to the specific POS categories. They usually doesn't contain a specific meaning for the task at hand and create noise.

Question 3.1 (10 points)

In *neural* language models, we represent words with low-dimensional vectors also called *embeddings*. We use these embeddings to compute a vector representation x of a given prefix, and then predict the probability of the next word conditioned on x. In the below cell, we use PyTorch, a machine learning framework, to explore this setup. We provide embeddings for the prefix "Alice talked to"; your job is to combine them into a single vector representation x using element-wise vector addition.

TIP: if you're finding the PyTorch coding problems difficult, you may want to run through the 60 minutes blitz tutorial!

```
import torch
torch.set_printoptions(sci_mode=False)
torch.manual_seed(0)

prefix = 'Alice talked to'
```

```
# spend some time understanding this code / reading relevant documentation!
# this is a toy problem with a 5 word vocabulary and 10-d embeddings
embeddings = torch.nn.Embedding(num_embeddings=5, embedding_dim=10)
vocab = {'Alice':0, 'talked':1, 'to':2, 'Bob':3, '.':4}

# we need to encode our prefix as integer indices (not words) that index
# into the embeddings matrix. the below line accomplishes this.
# note that PyTorch inputs are always Tensor objects, so we need
# to create a LongTensor out of our list of indices first.
indices = torch.LongTensor([vocab[w] for w in prefix.split()])
prefix_embs = embeddings(indices)
print('prefix embedding tensor size: ', prefix_embs.size())

# okay! we now have three embeddings corresponding to each of the three
# words in the prefix. write some code that adds them element-wise to obtain
# a representation of the prefix! store your answer in a variable named "x".

### YOUR CODE HERE!
x = torch.zeros(10)
for emb in prefix_embs:
  x += emb

### DO NOT MODIFY THE BELOW LINE
print('embedding sum: ', x)
```

```
prefix embedding tensor size:  torch.Size([3, 10])
embedding sum:  tensor([-0.1770, -2.3993, -0.4721,  2.6568,  2.7157, -0.1408,
         2.2783,  1.1165], grad_fn=<AddBackward0>)
```

Question 3.2 (5 points)

Modern language models do not use element-wise addition to combine the different word embeddings in the prefix into a single representation (a process called *composition*). What is a major issue with element-wise functions that makes them unsuitable for use as composition functions?

Write your answer here! Please keep it brief (i.e., 2-3 sentences).

Sentences are ordered. If we use element-wise addition we may end up with a final value but we would have lost the ordering of the sentence.

Question 3.3 (10 points)

One very important function in neural language models (and for basically every task we'll look at this semester) is the softmax, which is defined over an n-dimensional vector $< x_1, x_2, \ldots, x_n >$ as $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{1 \le j \le n} e^{x_j}}$. Let's say we have our prefix representation $\boldsymbol{x}$ from before. We can use the softmax function, along with a linear projection using a matrix W, to go from x to a probability distribution p over the next word: $p = \text{softmax}(W\,x)$.Let's explore this in the code cell below:

```
# remember, our goal is to produce a probability distribution over the
# next word, conditioned on the prefix representation x. This distribution
# is thus over the entire vocabulary (i.e., it is a 5-dimensional vector).
# take a look at the dimensionality of x, and you'll notice that it is a
# 10-dimensional vector. first, we need to **project** this representation
# down to 5-d. We'll do this using the below matrix:

W = torch.rand(10, 5)

# use this matrix to project x to a 5-d space, and then
# use the softmax function to convert it to a probability distribution.
# this will involve using PyTorch to compute a matrix/vector product.
# look through the documentation if you're confused (torch.nn.functional.softmax)
# please store your final probability distribution in the "probs" variable.

### YOUR CODE HERE
probs = torch.rand(5)
x = torch.unsqueeze(x, 1)
W = torch.transpose(W, 0, 1)
m = torch.nn.Softmax(dim=1)
props = m(torch.matmul(W, x))

### DO NOT MODIFY THE BELOW LINE!
print('probability distribution', probs)
```

```
    probability distribution tensor([0.3756, 0.5226, 0.5730, 0.6186, 0.6962])
```

Question 3.4 (15 points)

So far, we have looked at just a single prefix ("Alice talked to"). In practice, it is common for us to compute many prefixes in one computation, as this enables us to take advantage of GPU parallelism and also obtain better gradient approximations (we'll talk more about the latter point later). This is called *batching*, where each prefix is an example in a larger batch. Here, you'll redo the computations from the previous cells, but instead of having one prefix, you'll have a batch of two prefixes. The final output of this cell should be a 2x5 matrix that contains two probability distributions, one for each prefix. **NOTE: YOU WILL LOSE POINTS IF YOU USE ANY LOOPS IN YOUR ANSWER!** Your code should be completely vectorized (a few large computations is faster than many smaller ones).

```
# for this problem, we'll just copy our old prefix over three times
# to form a batch. in practice, each example in the batch would be different.
batch_indices = torch.cat(2 * [indices]).reshape((2, 3))
batch_embs = embeddings(batch_indices)
print('batch embedding tensor size: ', batch_embs.size())
```

```
# now, follow the same procedure as before:
# step 1: compose each example's embeddings into a single representation
# using element-wise addition. HINT: check out the "dim" argument of the torch.sum

# step 2: project each composed representation into a 5-d space using matrix W
# step 3: use the softmax function to obtain a 2x5 matrix with the probability dist

# please store this probability matrix in the "batch_probs" variable.

batch_probs = torch.rand(2,5)

W = torch.rand(10, 5)
comp = torch.sum(batch_embs, 1)
proj = torch.matmul(comp, W)
m = torch.nn.Softmax(dim=1)
batch_probs = m(proj)

### DO NOT MODIFY THE BELOW LINE
print("batch probability distributions:", batch_probs)
```

```
batch embedding tensor size:  torch.Size([2, 3, 10])
batch probability distributions: tensor([[    0.0370,     0.0704,     0.8874,
        [    0.0370,     0.0704,     0.8874,     0.0047,     0.0005]],
       grad_fn=<SoftmaxBackward0>)
```

Question 4 (20 points)

Choose one paper from EMNLP 2021 that you find interesting. A good way to do this is by scanning the titles and abstracts; there are hundreds of papers so take your time before selecting one! Then, write a summary in your own words of the paper you chose. Your summary should answer the following questions: what is its motivation? Why should anyone care about it? Were there things in the paper that you didn't understand at all? What were they? Fill out the below cell, and make sure to write 2-4 paragraphs for the summary to receive full credit!

Title of paper: Semantic Novelty Detection in Natural Language Descriptions

Authors: Nianzu Ma, Alexander Politowicz, Sahisnu Mazumder, Jiahua Chen†, Bing Liu†, Eric Robertson, Scott Grigsby

Conference name: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing

URL: https://aclanthology.org/2021.emnlp-main.66/

Your summary: Motivation: Novelty/Anomaly detection research so far mainly coarse-grained. Meaning it works in a topic or document level. It detects whether text belongs to that class or not. This paper authors introduce a new problem: fine-grained semantic novelty detec-tion. Whether text reflects real world or not. In this case main class becomes real-world.

Why should anyone care about it? This is a novel approach and introduces a new problem and solves it at the same time. Additionally it enables other research to be conducted on this topic.

Things I didn't Understand: Main thing I miss is how did we combine dependency parsing outpud with fixed embeddings. How did we get the input to feed into GAT's?