SI 630: Homework 2 – Word Embeddings and Attention

Yu Yan (kuminia)

Kaggle username: kkkyyy1

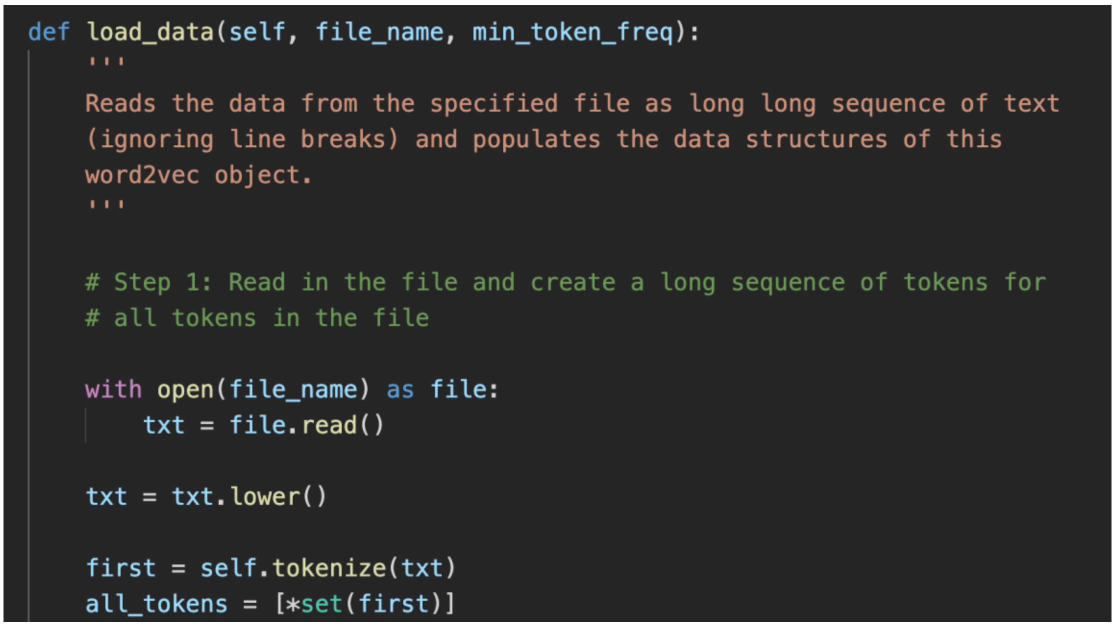
February 19, 2023

Task 1: Word2vec

(I will change the order of problem to report, because I have already done the subsampling part.)

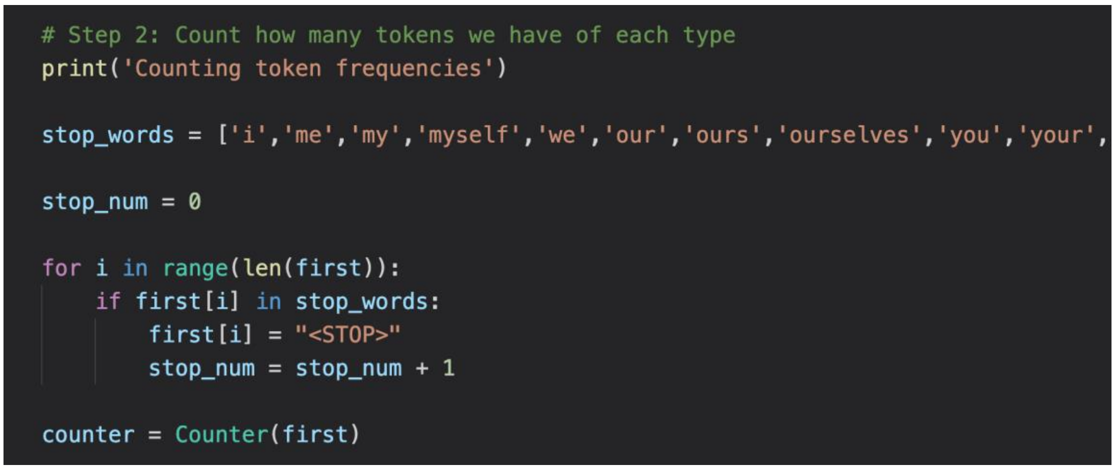
Problem 1. Modify function load data in the Corpus class to read in the text data and fill in the id to word, word to id, and full token sequence as ids fields. You can safely skip the rare word removal and subsampling for now.

I first lower the text in the file, and then use tokenize to read those text, and create a list to save the unique tokens called all_tokens.



```
def load_data(self, file_name, min_token_freq):
    '''
    Reads the data from the specified file as long long sequence of text
    (ignoring line breaks) and populates the data structures of this
    word2vec object.
    '''

    # Step 1: Read in the file and create a long sequence of tokens for
    # all tokens in the file

    with open(file_name) as file:
        txt = file.read()

    txt = txt.lower()

    first = self.tokenize(txt)
    all_tokens = [*set(first)]
```

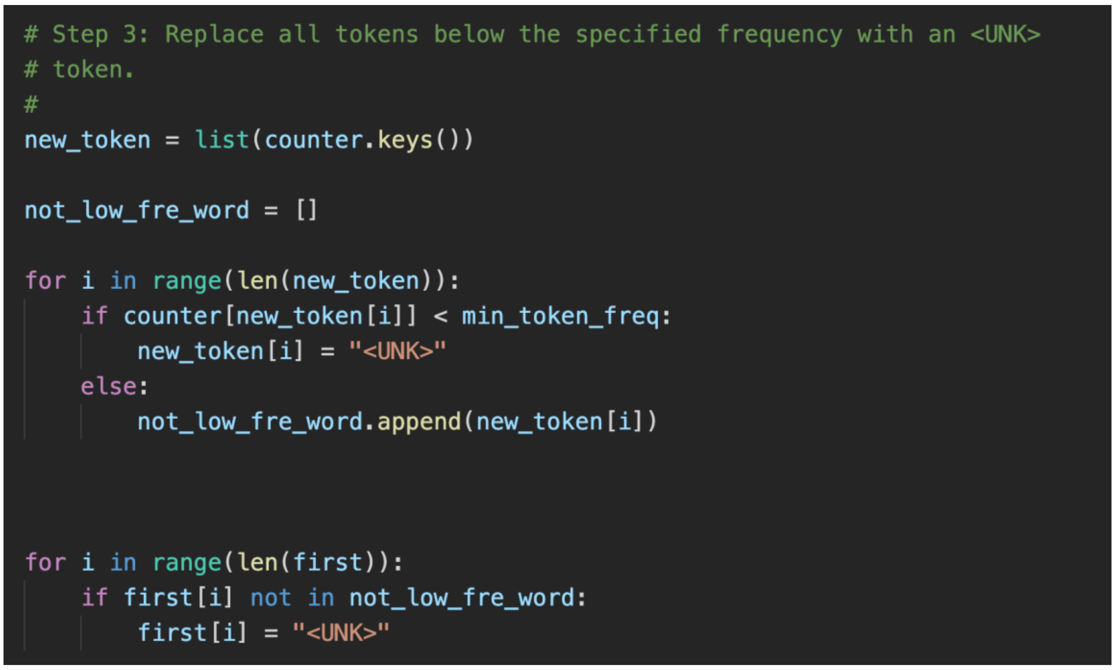
Then, I define a list called stop_words and add some words I think should be stop_words to this list, and change the stop_words into <STOP>. By this way, I can better do the next step. Then, I count the tokens using Counter.



```python
# Step 2: Count how many tokens we have of each type
print('Counting token frequencies')

stop_words = ['i','me','my','myself','we','our','ours','ourselves','you','your',

stop_num = 0

for i in range(len(first)):
    if first[i] in stop_words:
        first[i] = "<STOP>"
        stop_num = stop_num + 1

counter = Counter(first)
```

Problem 8. Modify function load data to convert all words with less than min count occurrences into <UNK> tokens.

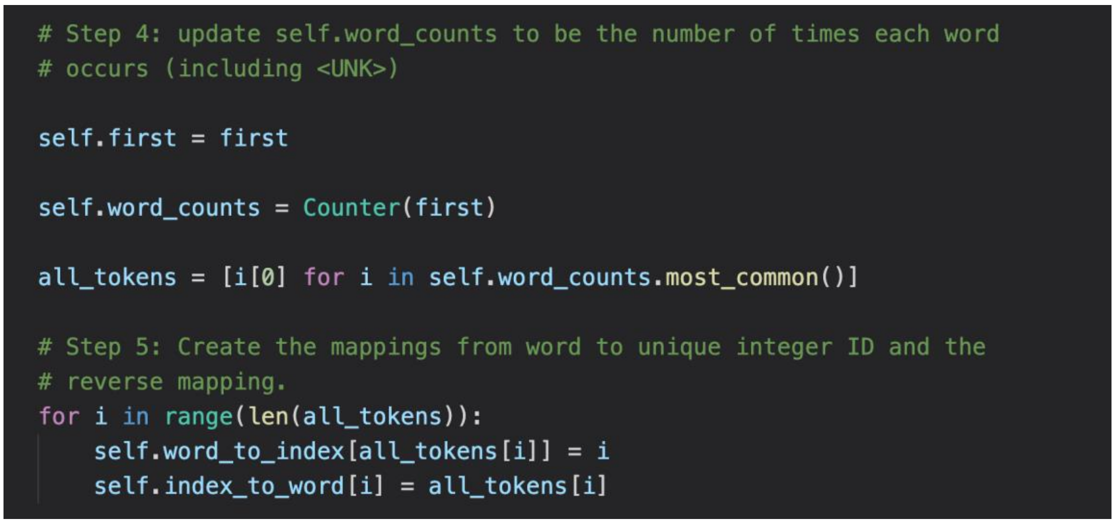
I use two for loops to replace all tokens below the min_token_freq with an <UNK> token, and all texts are in the variable called first.



```python
# Step 3: Replace all tokens below the specified frequency with an <UNK>
# token.
#
new_token = list(counter.keys())

not_low_fre_word = []

for i in range(len(new_token)):
    if counter[new_token[i]] < min_token_freq:
        new_token[i] = "<UNK>"
    else:
        not_low_fre_word.append(new_token[i])



for i in range(len(first)):
    if first[i] not in not_low_fre_word:
        first[i] = "<UNK>"
```

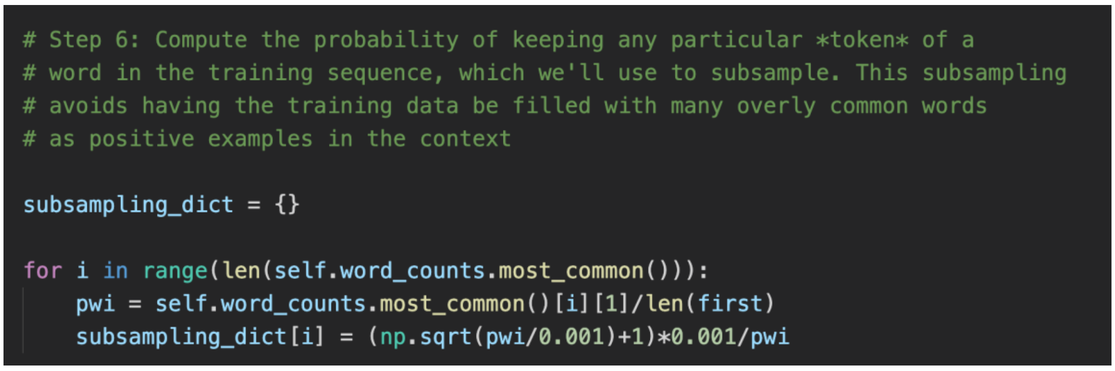
Problem 9. Modify function load data to compute the probability $p_k(w_i)$ of being kept during subsampling for each word w_i.

Before calculate the probability, I first use self. to define some variables.



```
# Step 4: update self.word_counts to be the number of times each word
# occurs (including <UNK>)

self.first = first

self.word_counts = Counter(first)

all_tokens = [i[0] for i in self.word_counts.most_common()]

# Step 5: Create the mappings from word to unique integer ID and the
# reverse mapping.
for i in range(len(all_tokens)):
    self.word_to_index[all_tokens[i]] = i
    self.index_to_word[i] = all_tokens[i]
```

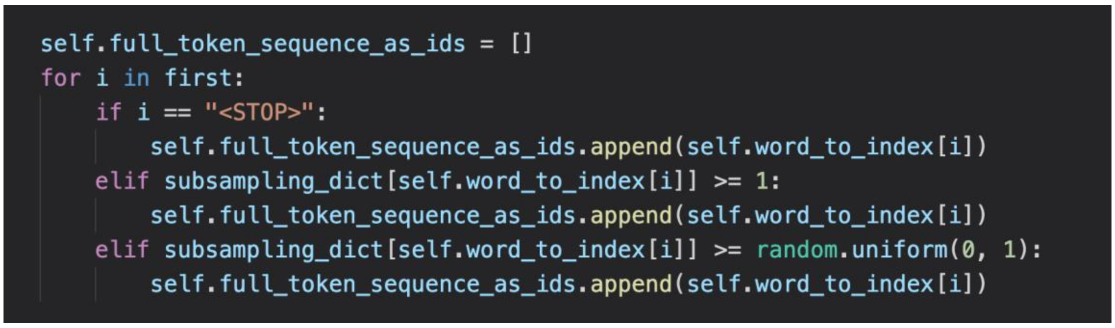
Then, I use for loop to get the probability of each unique tokens with the probability that being kept in a dictionary.



```
# Step 6: Compute the probability of keeping any particular *token* of a
# word in the training sequence, which we'll use to subsample. This subsampling
# avoids having the training data be filled with many overly common words
# as positive examples in the context

subsampling_dict = {}

for i in range(len(self.word_counts.most_common())):
    pwi = self.word_counts.most_common()[i][1]/len(first)
    subsampling_dict[i] = (np.sqrt(pwi/0.001)+1)*0.001/pwi
```

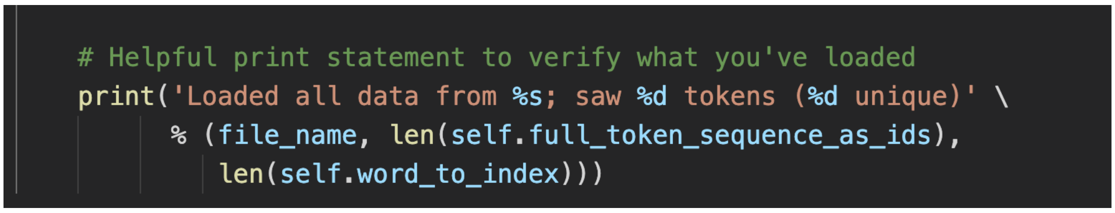
Problem 10. Modify function load data so that after the initial full token sequence as ids is constructed, tokens are subsampled (i.e., removed) according to their probability of being kept p_k (w_i).

In this step, I will not delete any stop words, which I have already defined its name “<STOP>”. Besides, I use random.uniform() function to get a number between 0 and 1 with the same probability, and use the list called full_token_sequnence_as_ids to append the words that has the probability to be kept more than the probability that I get from random.uniform() function.



```python
self.full_token_sequence_as_ids = []
for i in first:
    if i == "<STOP>":
        self.full_token_sequence_as_ids.append(self.word_to_index[i])
    elif subsampling_dict[self.word_to_index[i]] >= 1:
        self.full_token_sequence_as_ids.append(self.word_to_index[i])
    elif subsampling_dict[self.word_to_index[i]] >= random.uniform(0, 1):
        self.full_token_sequence_as_ids.append(self.word_to_index[i])
```

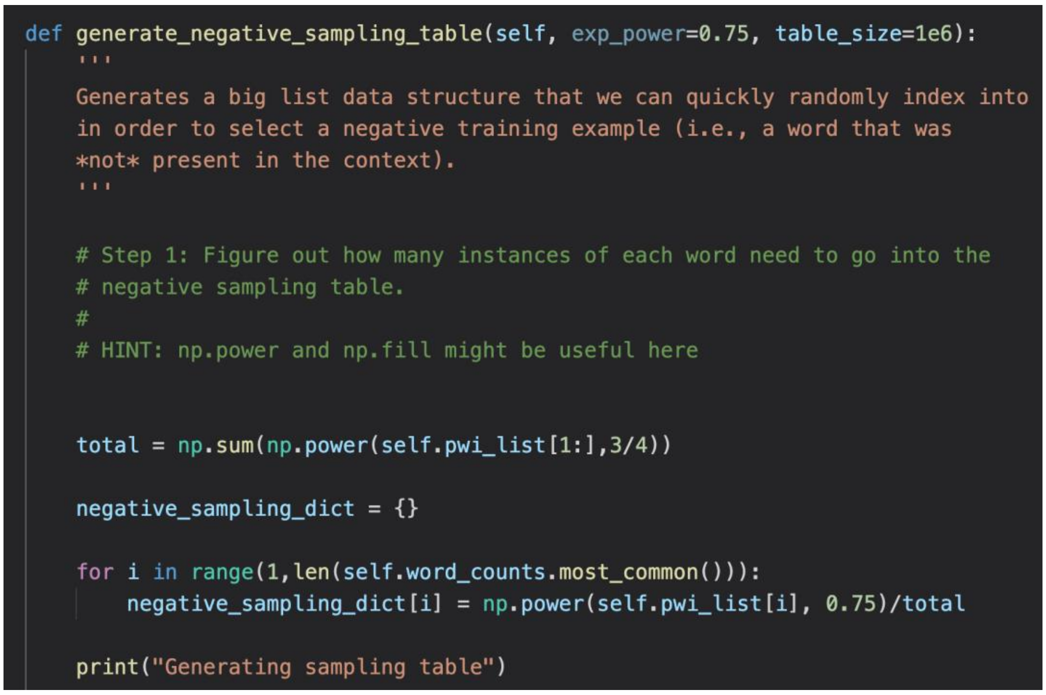
Then, I use the print statement to report what I get.



```python
    # Helpful print statement to verify what you've loaded
    print('Loaded all data from %s; saw %d tokens (%d unique)' \
          % (file_name, len(self.full_token_sequence_as_ids),
             len(self.word_to_index)))
```

Problem 2. Modify function generate negative sampling table to create the negative sampling table.

In this part, I first create a value called total, which is the sum of pwi_list (I define it before), and I only use [1:], because I don't want to have the stop words in negative sampling table and stop words are in the first position ([0]), which I have already checked. Then, I use a for loop get each unique token's probability in negative sampling.



```python
def generate_negative_sampling_table(self, exp_power=0.75, table_size=1e6):
    '''
    Generates a big list data structure that we can quickly randomly index into
    in order to select a negative training example (i.e., a word that was
    *not* present in the context).
    '''

    # Step 1: Figure out how many instances of each word need to go into the
    # negative sampling table.
    #
    # HINT: np.power and np.fill might be useful here


    total = np.sum(np.power(self.pwi_list[1:],3/4))

    negative_sampling_dict = {}

    for i in range(1,len(self.word_counts.most_common())):
        negative_sampling_dict[i] = np.power(self.pwi_list[i], 0.75)/total

    print("Generating sampling table")
```

I use this code to get the table with table_size as its size.

```python
self.negative_sampling_table = np.arange(table_size,dtype=int)
```

Finally, I use the code below to generate the negative sampling table.

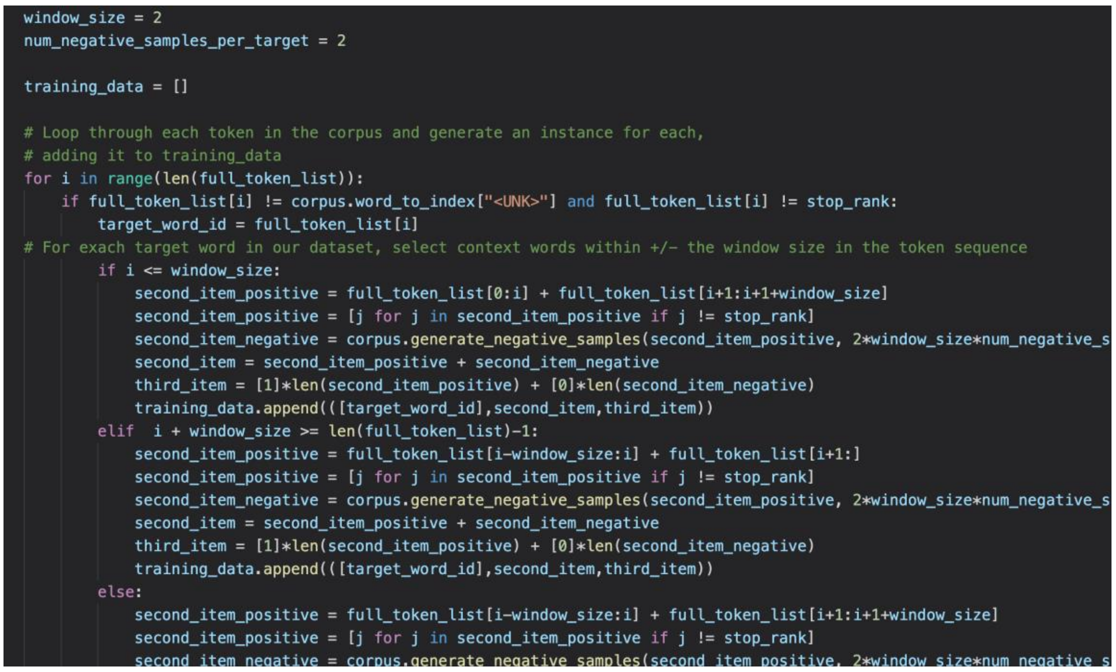
```python
last = 0
for i in negative_sampling_dict:
    size = round(negative_sampling_dict[i]*table_size)
    self.negative_sampling_table[last:last+size].fill(i)
    last = size+last
self.negative_sampling_table = self.negative_sampling_table[0:last]
```

Problem 3. Generate the list of training instances according to the specifications in the code.

Before I get the training instances, I first define the full_token_list.

```python
full_token_list = corpus.full_token_sequence_as_ids
```

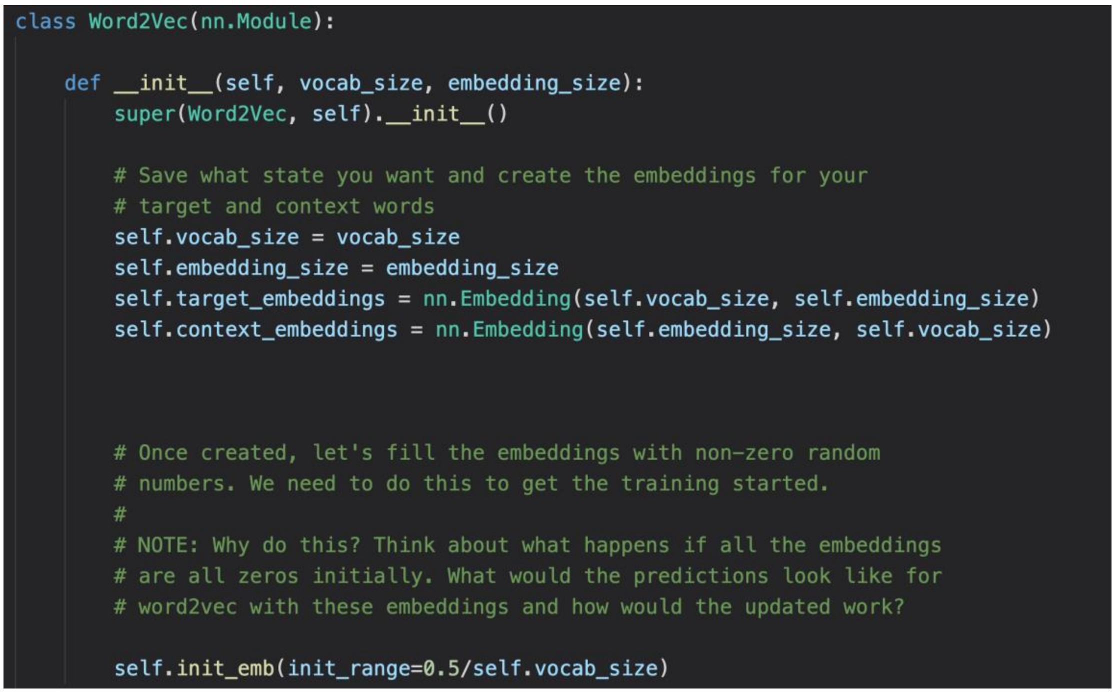
Then, I set the window_size equal to 2, and num_negative_samples_per_target equal to 2. And implement for loop to loop all of the word in full_token_list. When the full_token_list[i] equal to the number of "<UNK>" or the number of "<STOP >", I will not do anything. On the contrary, I will judge whether the word is at the first of the list or the last of the list, and deal with the two situations separately. And I will also delete the stop words, when they are in the context list. Finally, I append the tuple into training_data.



```
window_size = 2
num_negative_samples_per_target = 2

training_data = []

# Loop through each token in the corpus and generate an instance for each,
# adding it to training_data
for i in range(len(full_token_list)):
    if full_token_list[i] != corpus.word_to_index["<UNK>"] and full_token_list[i] != stop_rank:
        target_word_id = full_token_list[i]
# For exach target word in our dataset, select context words within +/- the window size in the token sequence
        if i <= window_size:
            second_item_positive = full_token_list[0:i] + full_token_list[i+1:i+1+window_size]
            second_item_positive = [j for j in second_item_positive if j != stop_rank]
            second_item_negative = corpus.generate_negative_samples(second_item_positive, 2*window_size*num_negative_s
            second_item = second_item_positive + second_item_negative
            third_item = [1]*len(second_item_positive) + [0]*len(second_item_negative)
            training_data.append(([target_word_id],second_item,third_item))
        elif  i + window_size >= len(full_token_list)-1:
            second_item_positive = full_token_list[i-window_size:i] + full_token_list[i+1:]
            second_item_positive = [j for j in second_item_positive if j != stop_rank]
            second_item_negative = corpus.generate_negative_samples(second_item_positive, 2*window_size*num_negative_s
            second_item = second_item_positive + second_item_negative
            third_item = [1]*len(second_item_positive) + [0]*len(second_item_negative)
            training_data.append(([target_word_id],second_item,third_item))
        else:
            second_item_positive = full_token_list[i-window_size:i] + full_token_list[i+1:i+1+window_size]
            second_item_positive = [j for j in second_item_positive if j != stop_rank]
            second_item_negative = corpus.generate_negative_samples(second_item_positive, 2*window_size*num_negative_s
```

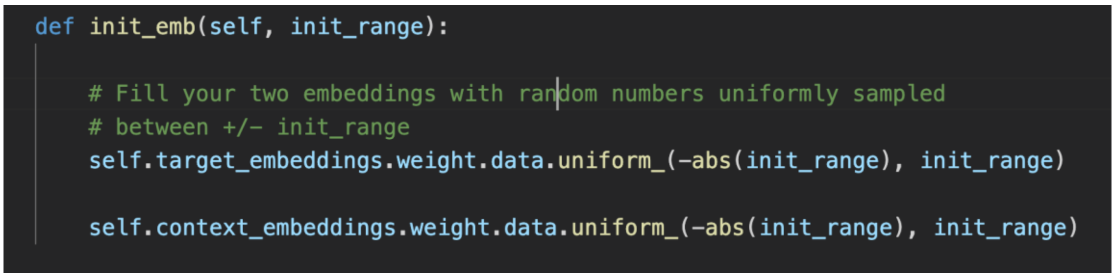
Problem 4. Modify the init weights function to initialize the values in the two Embedding objects based on the size of the vocabulary |V | and the size of the embeddings.

I first define those variables as below.



```
class Word2Vec(nn.Module):

    def __init__(self, vocab_size, embedding_size):
        super(Word2Vec, self).__init__()

        # Save what state you want and create the embeddings for your
        # target and context words
        self.vocab_size = vocab_size
        self.embedding_size = embedding_size
        self.target_embeddings = nn.Embedding(self.vocab_size, self.embedding_size)
        self.context_embeddings = nn.Embedding(self.embedding_size, self.vocab_size)



        # Once created, let's fill the embeddings with non-zero random
        # numbers. We need to do this to get the training started.
        #
        # NOTE: Why do this? Think about what happens if all the embeddings
        # are all zeros initially. What would the predictions look like for
        # word2vec with these embeddings and how would the updated work?

        self.init_emb(init_range=0.5/self.vocab_size)
```

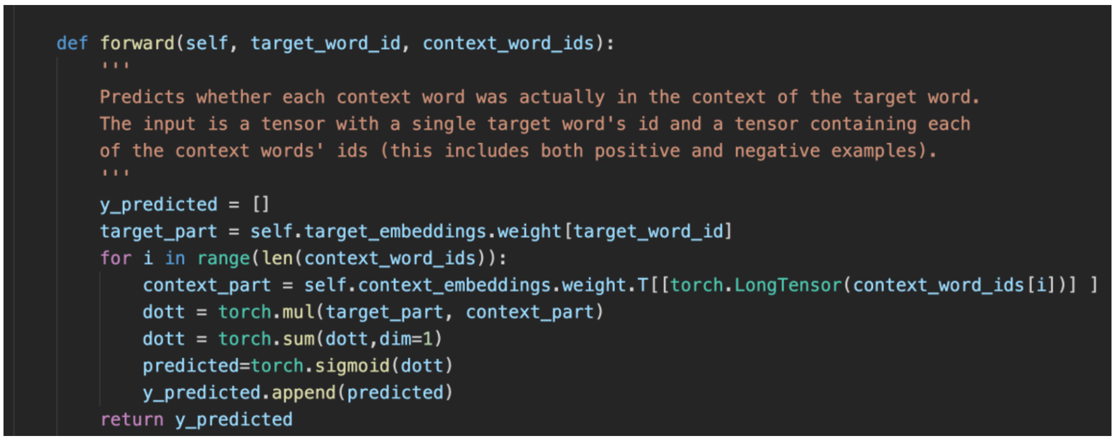
Next, I use this function to get the target_embeddings and context_embeddings.



```python
def init_emb(self, init_range):

    # Fill your two embeddings with random numbers uniformly sampled
    # between +/- init_range
    self.target_embeddings.weight.data.uniform_(-abs(init_range), init_range)

    self.context_embeddings.weight.data.uniform_(-abs(init_range), init_range)
```

Problem 5. Modify the forward function

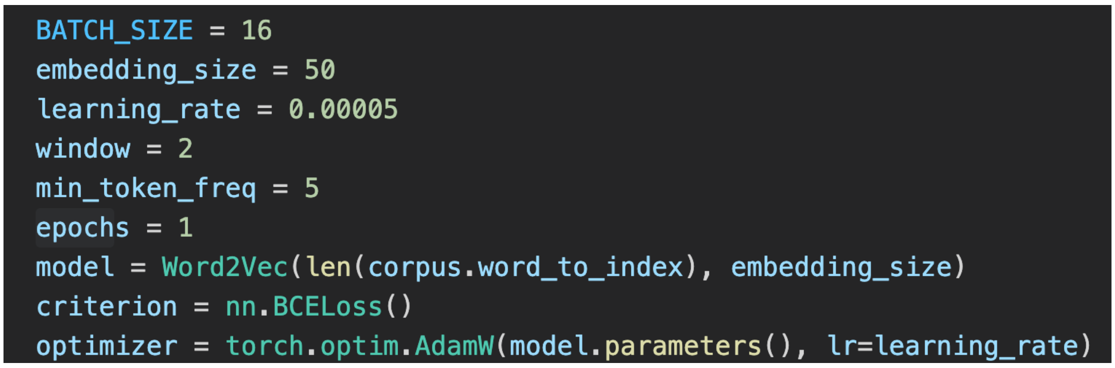
For my forward function part, I use a y_predicted list to record the result, and return the y_predicted. I also use mul and sum to get the dot product for those vectors, and using sigmoid function to get the 0 or 1 result.



```python
    def forward(self, target_word_id, context_word_ids):
        '''
        Predicts whether each context word was actually in the context of the target word.
        The input is a tensor with a single target word's id and a tensor containing each
        of the context words' ids (this includes both positive and negative examples).
        '''
        y_predicted = []
        target_part = self.target_embeddings.weight[target_word_id]
        for i in range(len(context_word_ids)):
            context_part = self.context_embeddings.weight.T[[torch.LongTensor(context_word_ids[i])] ]
            dott = torch.mul(target_part, context_part)
            dott = torch.sum(dott,dim=1)
            predicted=torch.sigmoid(dott)
            y_predicted.append(predicted)
        return y_predicted
```

Problem 6. Modify the cell containing the training loop to complete the required PyTorch training process. The notebook describes in more details all the steps

I first set those variables.



```python
BATCH_SIZE = 16
embedding_size = 50
learning_rate = 0.00005
window = 2
min_token_freq = 5
epochs = 1
model = Word2Vec(len(corpus.word_to_index), embedding_size)
criterion = nn.BCELoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
```

And I use Dataloader to get the make the training_data into batch.

```
dataloader = DataLoader(training_data, batch_size=BATCH_SIZE, shuffle=True)
```

This is the code for the training process, I also use tqdm to keep track of how fast things are and how much longer training will take.

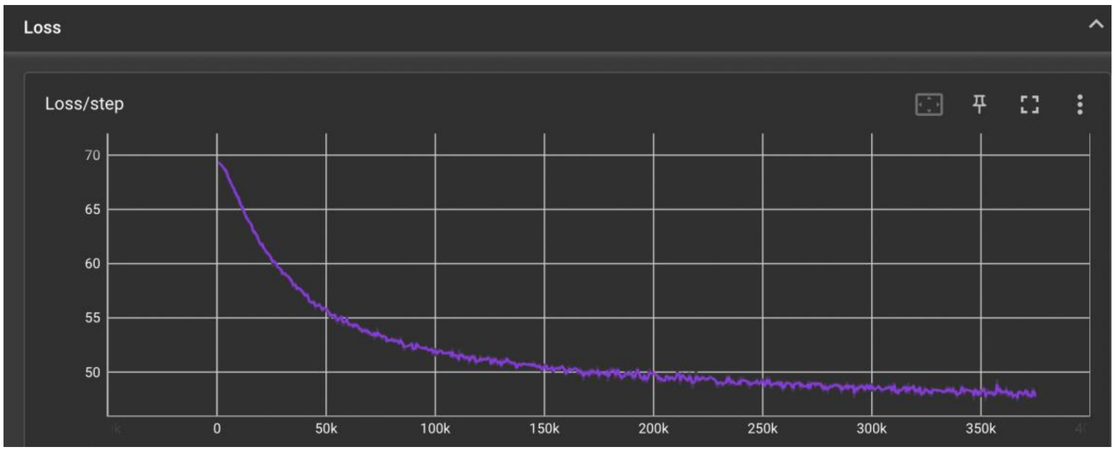
```
writer = SummaryWriter()

for epoch in range(epochs):

    loss_sum = 0

    # TODO: use your DataLoader to iterate over the data
    for step, data in enumerate(tqdm(dataloader)):

        # NOTE: since you created the data as a tuple of three np.array instances,
        # these have now been converted to Tensor objects for us
        target_ids, context_ids, labels = data


        # TODO: Fill in all the training details here

        y_predicted = model(target_ids,context_ids)

        loss = criterion(torch.stack(y_predicted), torch.stack(labels).float())

        loss_sum = loss_sum + loss.item()

        optimizer.zero_grad()

        loss.backward()

        optimizer.step()
```

I will record the loss every 100 steps.

```
    if (step+1)%100 == 0:
        writer.add_scalar("Loss/step", loss_sum, step+1)
        loss_sum = 0
```

Problem 7. Check that your model actually works.

Yes, it actually works.



Problem 12. Try batch sizes of 2, 8, 32, 64, 128, 256, 512 to see how fast each step (one batch worth of updates) is and the total estimated time.

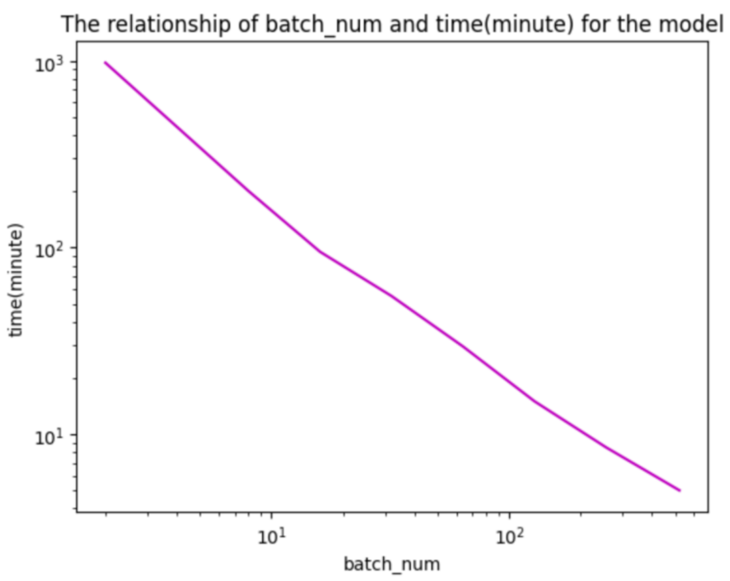
I also add the batch size 16 in this try. I just run the model with different batch sizes, and record the time as minutes, then I shut down the running and clear the output of each different batch sizes model.

```
batch_num = [2,8,16,32,64,128,256,521]
time = [978,201,95,55,29.5,15,8.5,5]
```

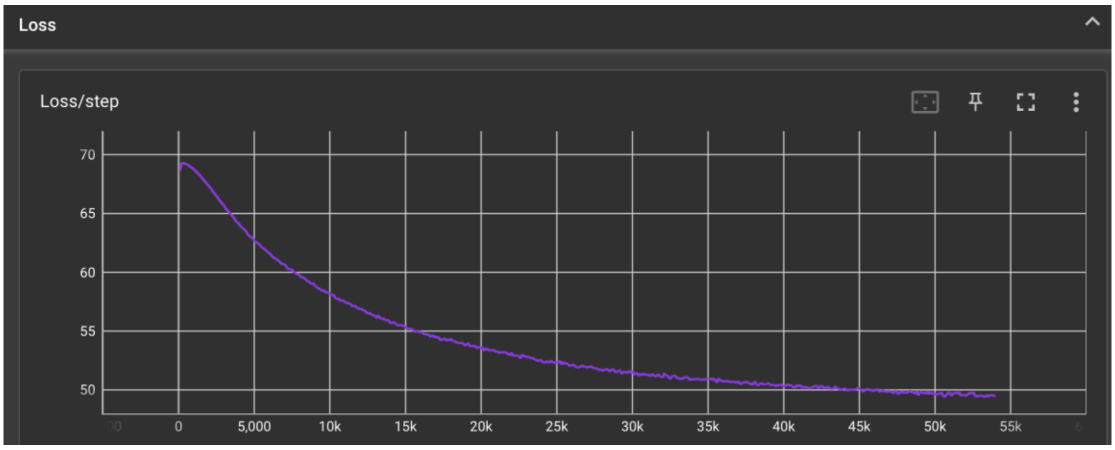
I use matplotlib to make this plot, and make each axis with log scale.

```
plt.figure()
plt.plot(batch_num,time,"-m")
plt.xlabel("batch_num")
plt.ylabel("time(minute)")
plt.xscale("log")
plt.yscale("log")
plt.title("The relationship of batch_num and time(minute) with log-scale on each axis")
```

This is the final result.



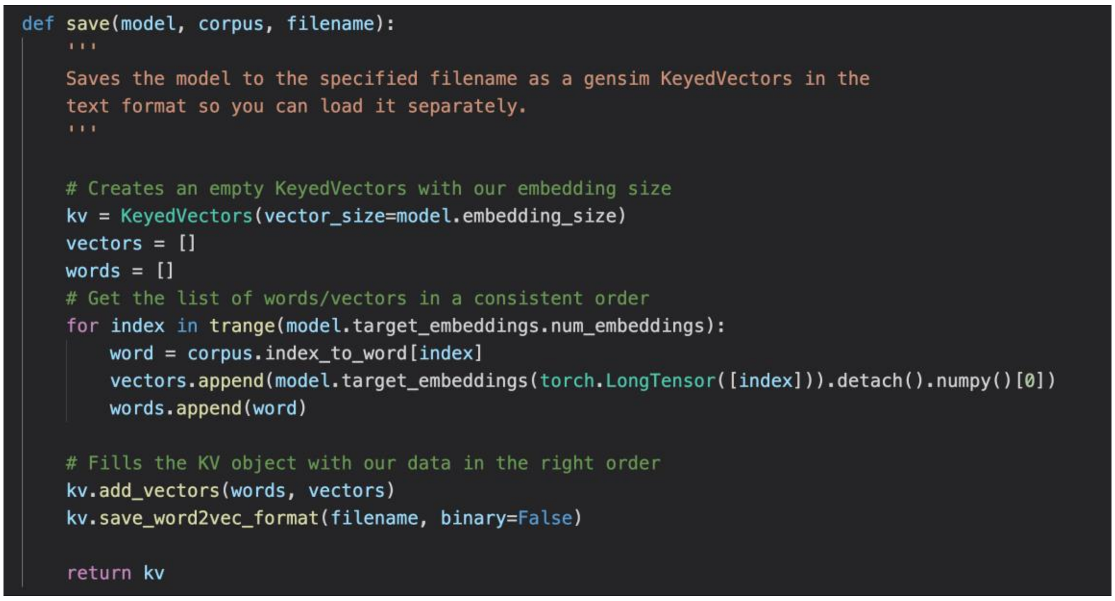
Problem 13. Train your model on at least one epoch worth of data.



Task 3: Qualitative Evaluation of Word Similarities

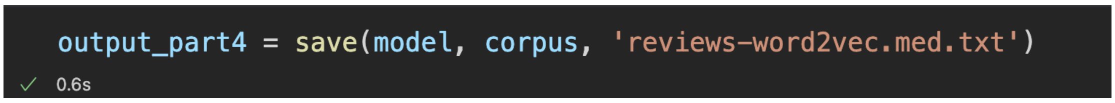
Problem 14. Load the model (vectors) you saved in Task 2 by using the Jupyter notebook provided (or code that does something similar) that uses the Gensim package to read the vectors. Gensim has a number of useful utilities for working with pretrained vectors.

I use this function to save the model.



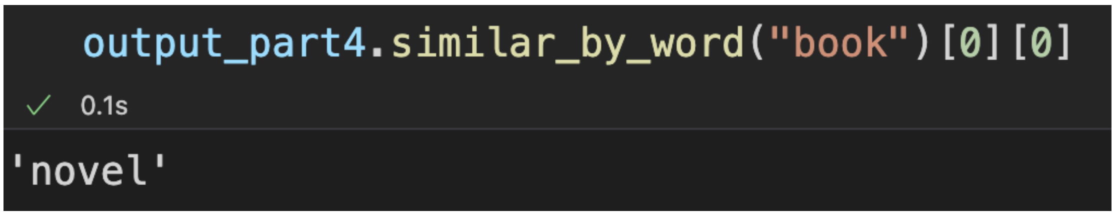
```python
def save(model, corpus, filename):
    '''
    Saves the model to the specified filename as a gensim KeyedVectors in the
    text format so you can load it separately.
    '''

    # Creates an empty KeyedVectors with our embedding size
    kv = KeyedVectors(vector_size=model.embedding_size)
    vectors = []
    words = []
    # Get the list of words/vectors in a consistent order
    for index in trange(model.target_embeddings.num_embeddings):
        word = corpus.index_to_word[index]
        vectors.append(model.target_embeddings(torch.LongTensor([index])).detach().numpy()[0])
        words.append(word)

    # Fills the KV object with our data in the right order
    kv.add_vectors(words, vectors)
    kv.save_word2vec_format(filename, binary=False)

    return kv
```

I use output_part4 to record the result.

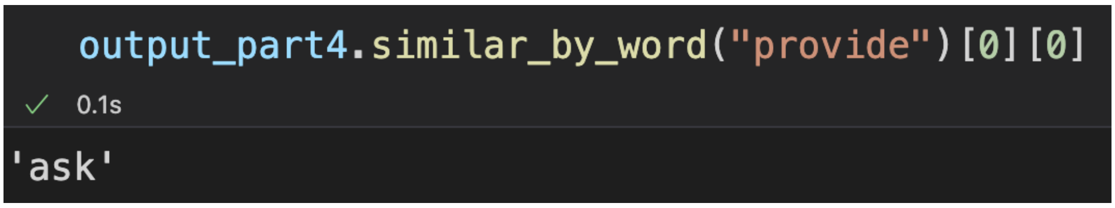


```python
output_part4 = save(model, corpus, 'reviews-word2vec.med.txt')
```
```
✓ 0.6s
```

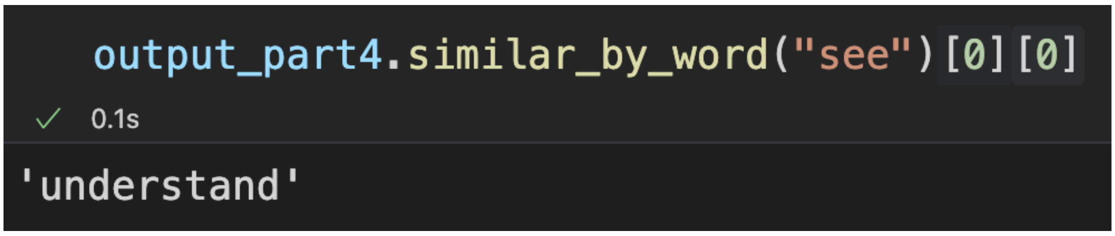
Problem 15. Pick 10 target words and compute the most similar for each using Gensim's function.



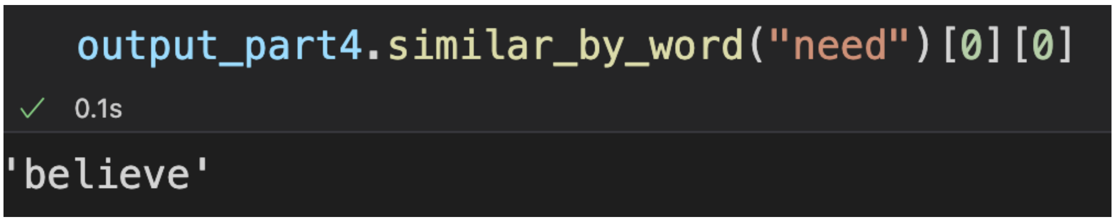
```python
output_part4.similar_by_word("book")[0][0]
```
```
✓ 0.1s
'novel'
```



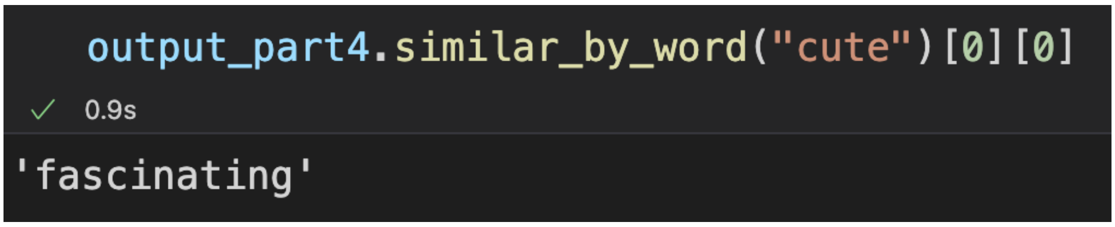
```python
output_part4.similar_by_word("provide")[0][0]
```
```
✓ 0.1s
'ask'
```



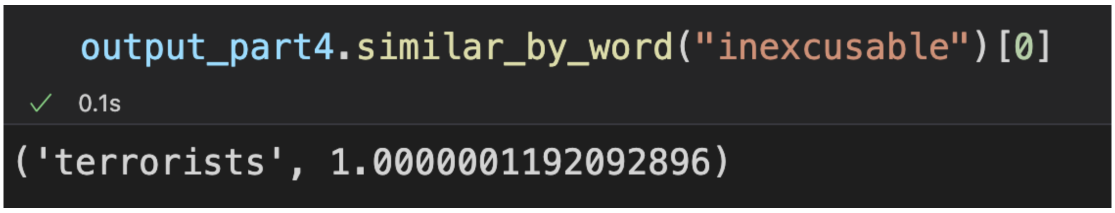
```python
output_part4.similar_by_word("see")[0][0]
```
```
✓ 0.1s
'understand'
```



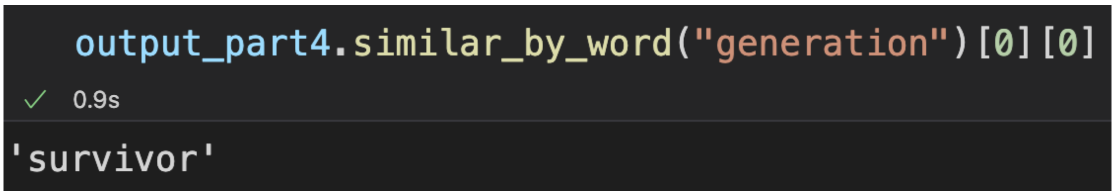
```
output_part4.similar_by_word("need")[0][0]
✓ 0.1s
'believe'
```



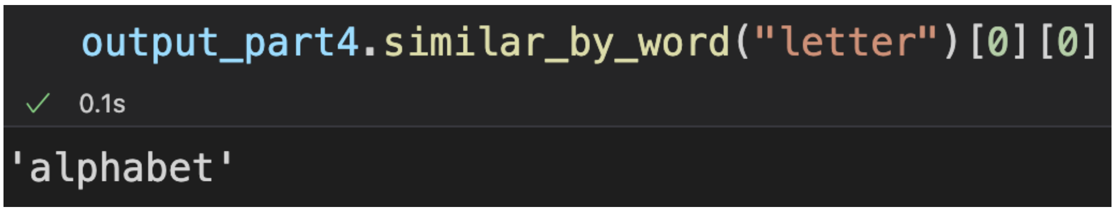
```
output_part4.similar_by_word("cute")[0][0]
✓ 0.9s
'fascinating'
```



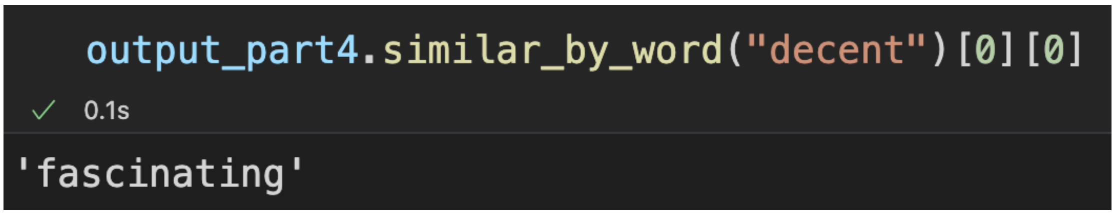
```
output_part4.similar_by_word("inexcusable")[0]
✓ 0.1s
('terrorists', 1.0000001192092896)
```



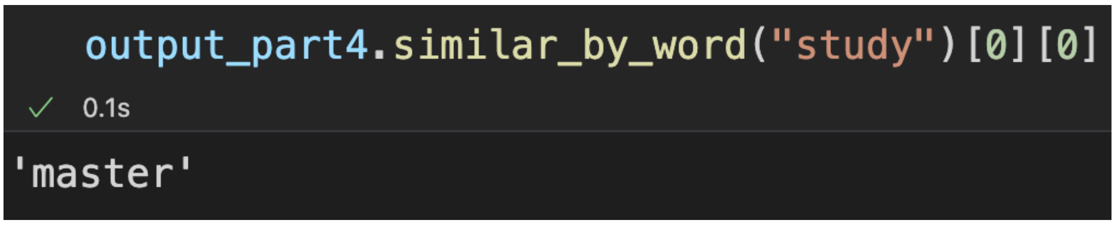
```
output_part4.similar_by_word("generation")[0][0]
✓ 0.9s
'survivor'
```



```
output_part4.similar_by_word("letter")[0][0]
✓ 0.1s
'alphabet'
```



```
output_part4.similar_by_word("decent")[0][0]
✓ 0.1s
'fascinating'
```

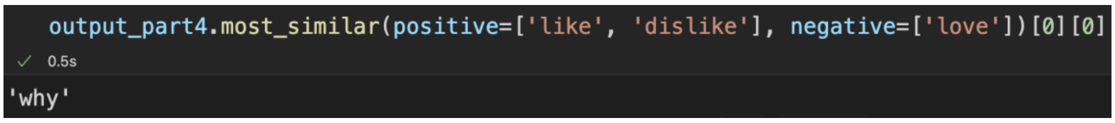


```
output_part4.similar_by_word("study")[0][0]
✓ 0.1s
'master'
```

Most of them are semantically similar to the target word, for example, alphabet really likes letter in some situations, and cute also likes fascinating. Inexcusable doesn't like terrorists, one is adjective and the other is noun, but they also have same meanings in some sentences.

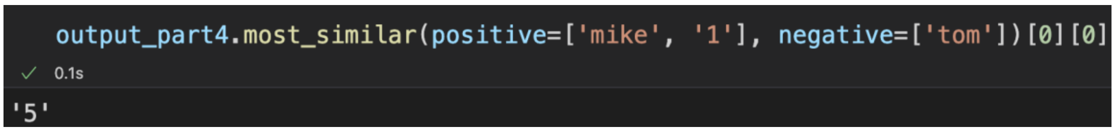
Problem 16. Given the analogy function, find five interesting word analogies with your word2vec model.

The first is like - love = dislike – hate. But the output is not very successful, I think that is because like sometimes can be different meanings, not just love.



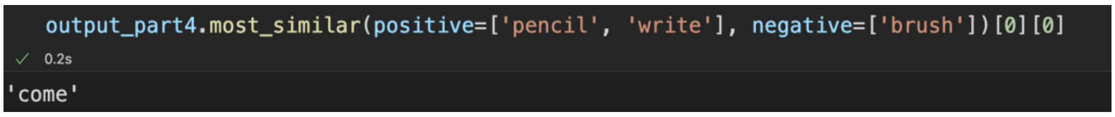
```
output_part4.most_similar(positive=['like', 'dislike'], negative=['love'])[0][0]
✓ 0.5s
'why'
```

The second is mike - 1 = tom – (a number), which can be represented like name + age = another name + age. The output is very good.



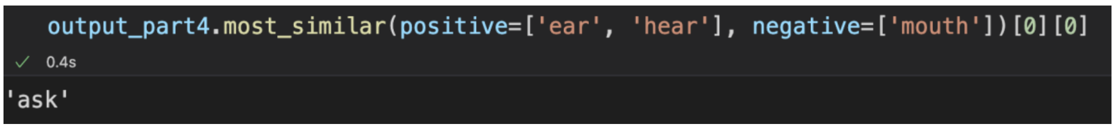
```
output_part4.most_similar(positive=['mike', '1'], negative=['tom'])[0][0]
✓ 0.1s
'5'
```

The third is pencil + write = brush + paint, the output is not very good.



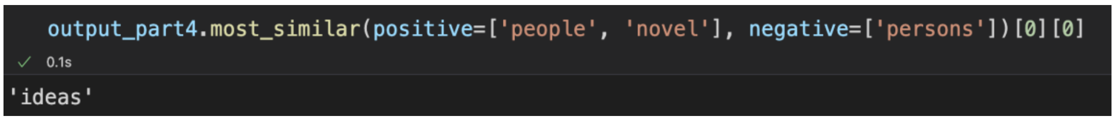
```
output_part4.most_similar(positive=['pencil', 'write'], negative=['brush'])[0][0]
✓ 0.2s
'come'
```

The fourth is ear + hear = mouse + (eat), the output is “ask”, although it is not “eat”, but “ask” is also very acceptable.



```
output_part4.most_similar(positive=['ear', 'hear'], negative=['mouth'])[0][0]
✓ 0.4s
'ask'
```

The fifth is people + novel= persons + (book), the output is “ideas”, although it is not “book” or “books”, but “ideas” is also acceptable.

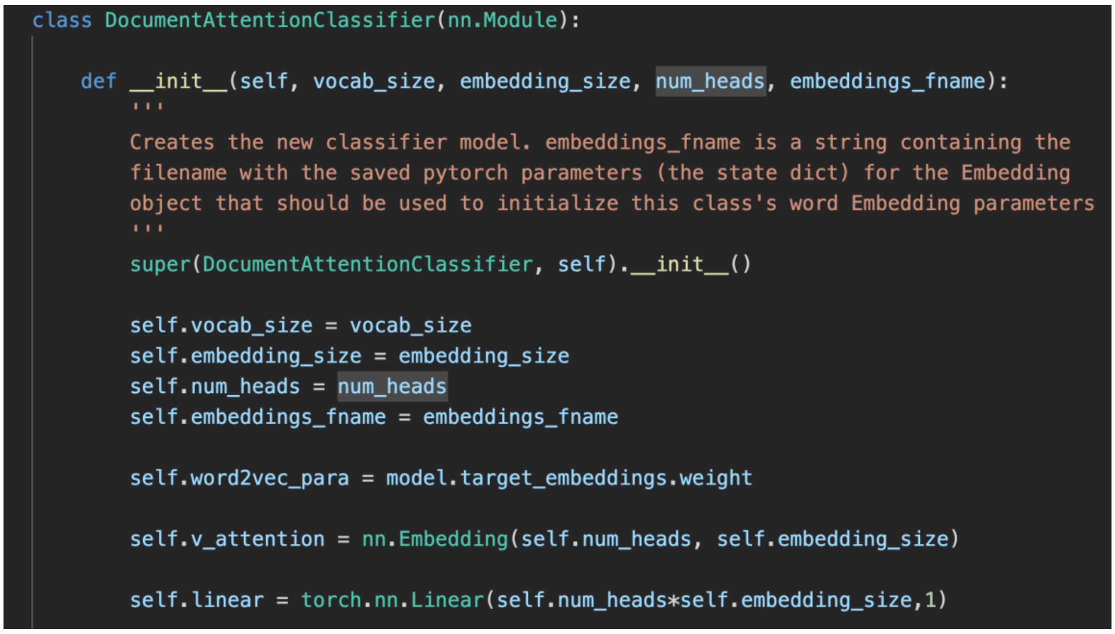


```
output_part4.most_similar(positive=['people', 'novel'], negative=['persons'])[0][0]
✓ 0.1s
'ideas'
```

Task 4: Using Word Vectors with Attention for Classification

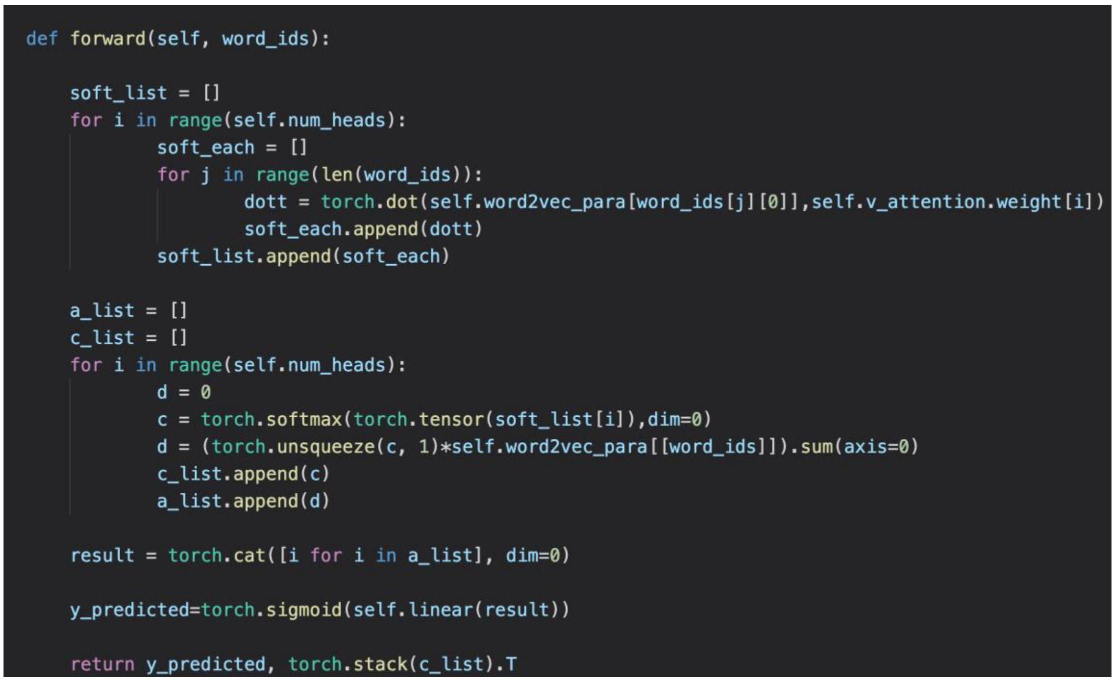
Problem 17. Build your attention-based classification model like the above. Your new model should use tensorboard to track the loss and F1

I first define the variables, the variables v_attention and linear will be changed.



```
class DocumentAttentionClassifier(nn.Module):

    def __init__(self, vocab_size, embedding_size, num_heads, embeddings_fname):
        '''
        Creates the new classifier model. embeddings_fname is a string containing the
        filename with the saved pytorch parameters (the state dict) for the Embedding
        object that should be used to initialize this class's word Embedding parameters
        '''
        super(DocumentAttentionClassifier, self).__init__()

        self.vocab_size = vocab_size
        self.embedding_size = embedding_size
        self.num_heads = num_heads
        self.embeddings_fname = embeddings_fname

        self.word2vec_para = model.target_embeddings.weight

        self.v_attention = nn.Embedding(self.num_heads, self.embedding_size)

        self.linear = torch.nn.Linear(self.num_heads*self.embedding_size,1)
```

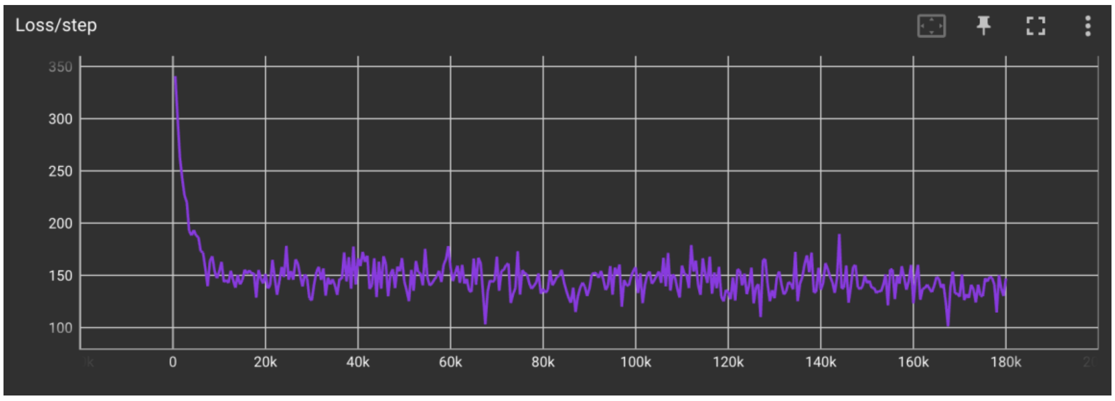
This is the forward function.

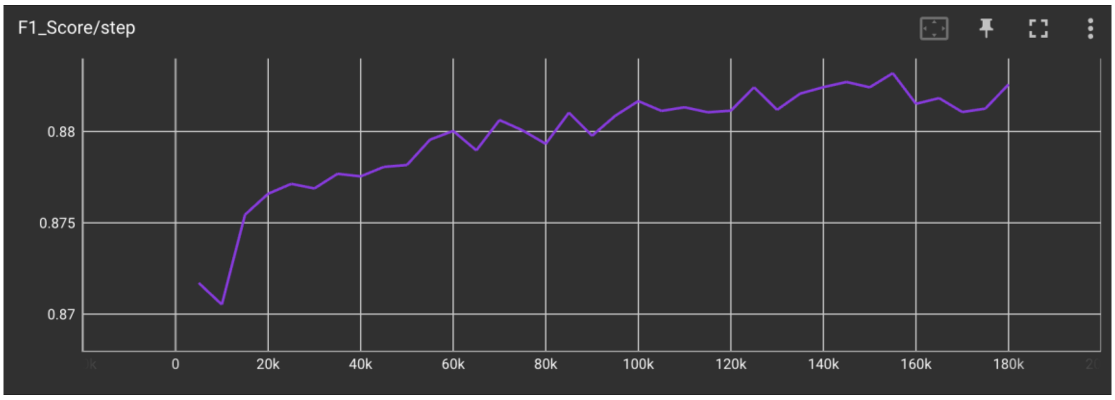


```
def forward(self, word_ids):

    soft_list = []
    for i in range(self.num_heads):
            soft_each = []
            for j in range(len(word_ids)):
                    dott = torch.dot(self.word2vec_para[word_ids[j][0]],self.v_attention.weight[i])
                    soft_each.append(dott)
            soft_list.append(soft_each)

    a_list = []
    c_list = []
    for i in range(self.num_heads):
            d = 0
            c = torch.softmax(torch.tensor(soft_list[i]),dim=0)
            d = (torch.unsqueeze(c, 1)*self.word2vec_para[[word_ids]]).sum(axis=0)
            c_list.append(c)
            a_list.append(d)

    result = torch.cat([i for i in a_list], dim=0)

    y_predicted=torch.sigmoid(self.linear(result))

    return y_predicted, torch.stack(c_list).T
```

Problem 18. Train the classifier for at least one epoch on the provided training data.

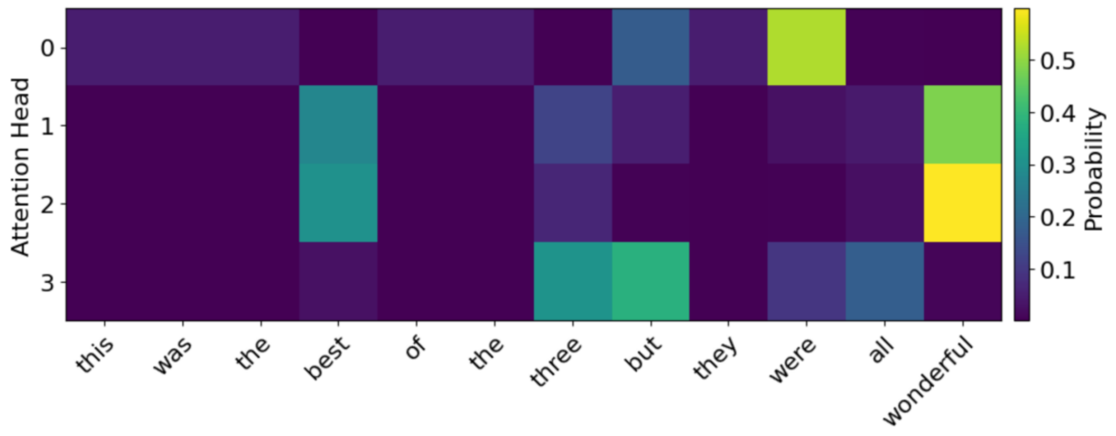
You should include a plot for the loss and the F1. Longer training is encouraged but not required.

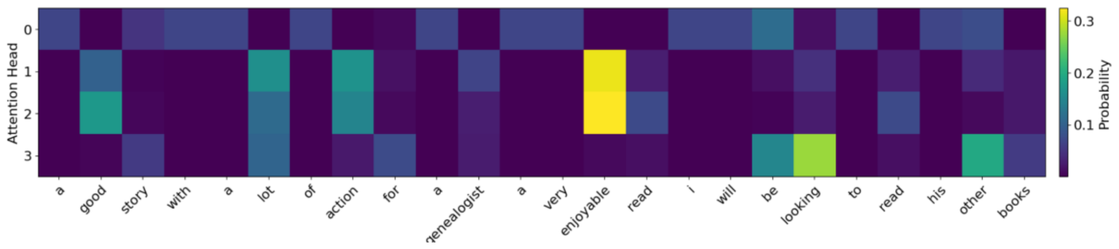


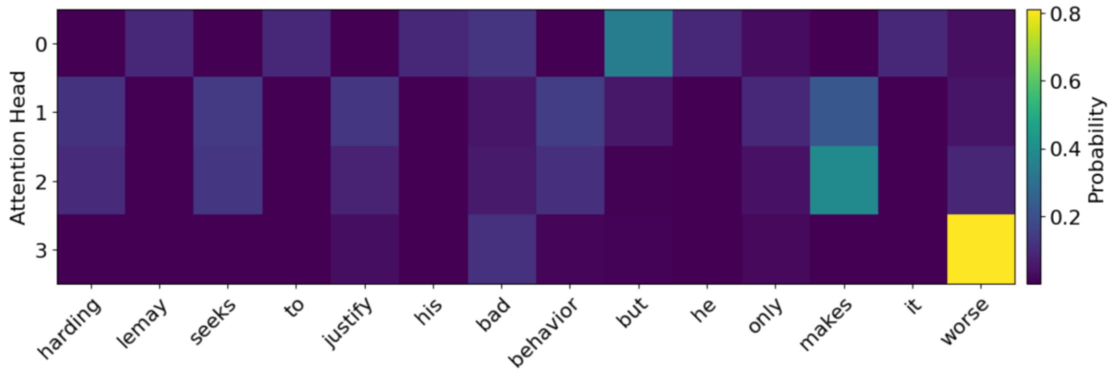


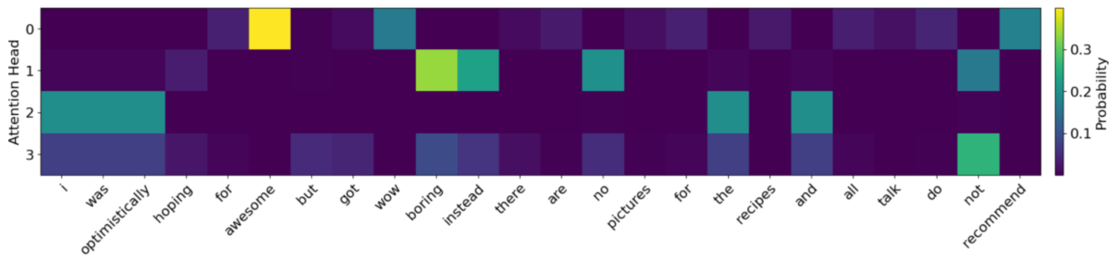
Problem 21.

1. Generate at least four "interesting" attention plots from text in the dev data, at least two for each class, and describe why you think the plots are interesting.









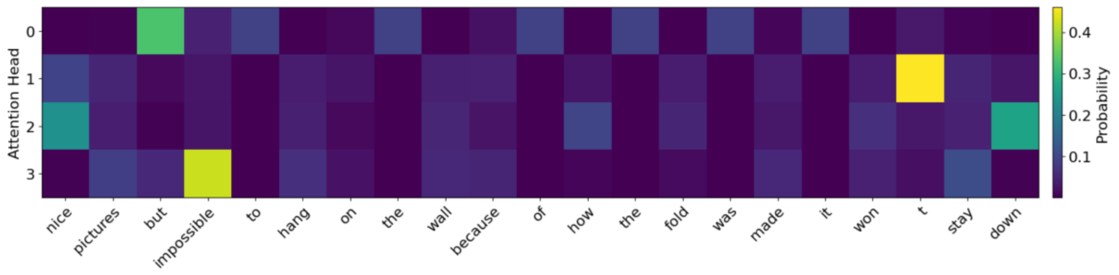
The first two are 1, and the rest are 0.

I really like those plots. The brightest parts of these images are adjectives or negative words (not) and turning words (but), so we can easily find some patterns in them. Besides, the difference between 0 and 1 will be very big. In sentences classified as 0, negative words or adjectives expressing negative will be very brightest, while in sentences classified as 1, positive words will be very brightest. So, from these four simple examples, we can already see that 1 represents positive emotion, 0 represents negative emotions, so I like these four pictures very much.

2. Using what you've observed from the visualizations, write a short paragraph describing what you think the attention heads are looking for.

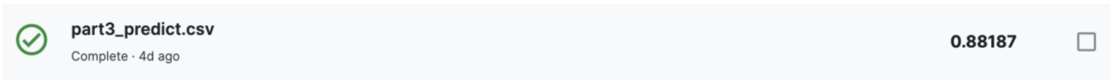
The attentions are looking for the most emotional words, generally adjectives; In addition, if the sentence contains negative words or turning words, the attention will pay more attention to these words. Therefore, when the sentence contains the negative words or turning words, it is more likely to get 0, however, when it just has the positive words, it will give 1.

3. Try to fool the classifier by either writing an example that the model predicts incorrectly or directly looking for a mistake on the dev data.



The attention is looking at adjective and turning words in this item, however, this sentence is very complex, it contains negative words, positive words and turning words. Although, the attention goes in the right path, but it still can't get the right result. I think that is because the model doesn't have enough input to understand more complex sentimental sentence.

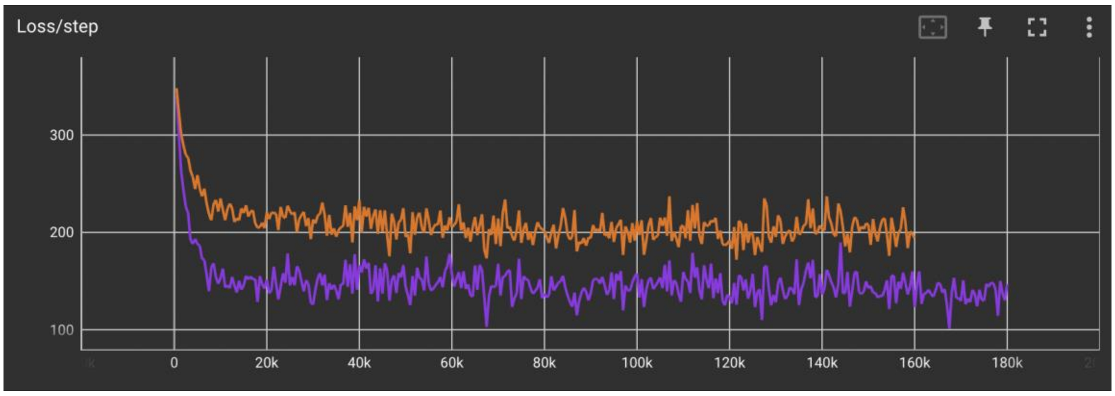
Problem 20. Predict the classifications for the test set and upload the scores to Kaggle. My final result is 0.88187.

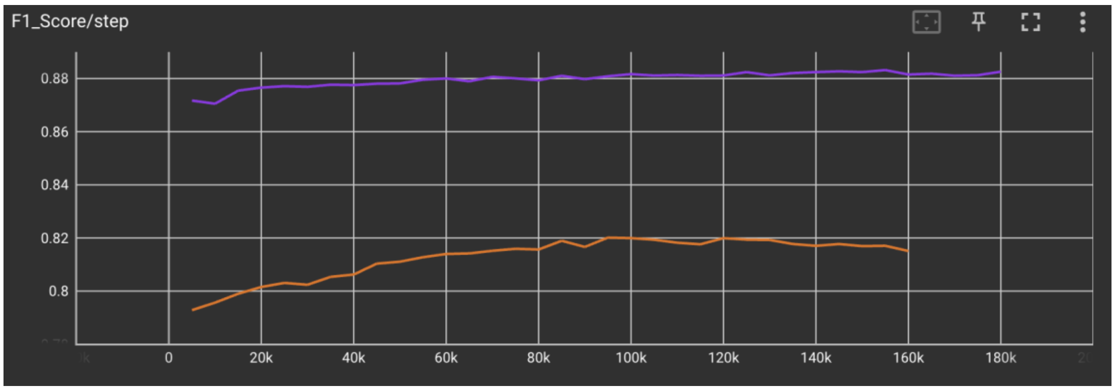


part3_predict.csv
Complete · 4d ago
0.88187

Problem 19. In a few sentences, describe whether you think we should freeze our word vectors in this setting or not.

We shouldn't freeze our word vectors. As the figures show below, the orange line represents the change of loss and F1_score with freeze, and the purple represents the change of loss and F1 score without freeze. The purple line has good results for both loss and F1 score, therefore, we should not freeze our word vectors in this setting.


Loss/step
300
200
100
0
20k
40k
60k
80k
100k
120k
140k
160k
180k


F1_Score/step
0.88
0.86
0.84
0.82
0.8
0
20k
40k
60k
80k
100k
120k
140k
160k
180k