

Submitted by - Prathamesh Joshi
Kaggle Username - prathamesh0902
STATS 630 Homework 2

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

```
all_tokens_repeat = []
print('Reading data and tokenizing')
with open(file_name, 'r') as f:
    all_data = f.readlines()
    for each_word in all_data:
        all_tokens_repeat += self.tokenize(each_word)
```

```
idx = 0
for key, val in self.word_counts.items():
    self.word_to_index[key] = idx
    self.index_to_word[idx] = key
    idx += 1
```

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

```
new_probs = {}
table_lengths = {}
denom_sum = 0
for token, freq in tqdm(self.word_counts.items()):
    num_sum = np.power(freq, exp_power)
    denom_sum += num_sum
    new_probs[token] = num_sum

for token, freq in tqdm(self.word_counts.items()):
    new_probs[token] = freq / denom_sum
    table_lengths[token] = round(new_probs[token] * table_size)
```

```
self.negative_sampling_table = np.array(self.negative_sampling_table)
for token, length in tqdm(table_lengths.items()):
    temp_table = np.zeros((length,), dtype = int)
    temp_table.fill(self.word_to_index[token])
    self.negative_sampling_table = np.concatenate((self.negative_sampling_table, temp_table), axis=None)
pass
```

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

```
#Modification 2: Avoiding adding a negative example that has the same ID as the current context_word
while (len(results)<num_samples):
    ids = np.random.choice(self.negative_sampling_table,1)
    for id_cur in ids:
        if id_cur != cur_context_word_id:
            results.append(int(id_cur))
```

```
training_data.append((np.array([token_list[idx]]),np.array(context_list),np.array(predicted_labels)))
```

Output:

```
(array([27]),
 array([ 25,   3,  28, 328,  29,  72,  80, 624, 105, 127,   3, 158]),
 array([1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0])),
(array([28]),
 array([ 3,   1, 152,  22,  48, 140,   5, 121,  62, 405,  44,  29]),
 array([1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])),
```

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. Unlike in logistic regression where we initialized our β vector be zeros, here, we'll initialize the weights to have small non-zero values centered on zero and sampled from $(-init_range, init_range)$.

```
self.target_embeddings = torch.nn.Embedding(vocab_size,embedding_size)
self.context_embeddings = torch.nn.Embedding(vocab_size,embedding_size)

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

def init_emb(self, init_range):

    # Fill your two embeddings with random numbers uniformly sampled
    # between +/- init_range

    self.target_embeddings.weight.data.uniform_(-init_range, init_range)
    self.context_embeddings.weight.data.uniform_(-init_range, init_range)

    pass
```

Problem 5. Modify the forward function

```
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).
    """
    # code locator
    targets = self.target_embeddings(target_word_id)
    contexts = self.context_embeddings(context_word_ids)
    product = torch.mul(targets, contexts)

    sum_mat = product.sum(dim =2)

    sigmoid_mat = torch.sigmoid(sum_mat)


    return sigmoid_mat
```

Problem 6. Modify the cell containing the training loop to complete the required PyTorch training process.

- Completed – check code file

Problem 7. Check that your model actually works. We recommend running your code on the wiki-bios.10k.txt file for one epoch. After this much data, your model should know enough for common words that the nearest neighbors (words with the most similar vectors) to words like “January” will be month-related words. We’ve provided code at the end of the notebook to explore. Try a few examples and convince yourself that your model/code is working.

```
get_neighbors(model, corpus.word_to_index, "story")
```

100%  40445/40445 [

```
[{'word': 'storyline', 'score': 0.9274334907531738},
 {'word': 'pace', 'score': 0.9023187756538391},
 {'word': 'concept', 'score': 0.9005336165428162},
 {'word': 'plot', 'score': 0.8918889760971069},
 {'word': 'twist', 'score': 0.8911790251731873},
 {'word': 'ending', 'score': 0.8698703646659851},
 {'word': 'premise', 'score': 0.8635672330856323},
 {'word': 'novel', 'score': 0.8629588484764099},
 {'word': 'conclusion', 'score': 0.8614411950111389},
 {'word': 'mystery', 'score': 0.8613811731338501}]
```

```
get_neighbors(model, corpus.word_to_index, "happy")
```

```
100% 40445/40445 [
```

```
[{'word': 'satisfied', 'score': 0.958477795124054},  
{'word': 'thrilled', 'score': 0.9455586671829224},  
{'word': 'pleased', 'score': 0.9336206912994385},  
{'word': 'glad', 'score': 0.9326156377792358},  
{'word': 'surprised', 'score': 0.9276609420776367},  
{'word': 'enthralled', 'score': 0.9238246083259583},  
{'word': 'excited', 'score': 0.917981743812561},  
{'word': 'shocked', 'score': 0.9178008437156677},  
{'word': 'engrossed', 'score': 0.9165638089179993},  
{'word': 'sorry', 'score': 0.9156234264373779}]
```

Problem 8. Modify function load data to convert all words with less than min count occurrences into <UNK>. tokens. Modify your dataset generation code to avoid creating a training instance when the target word is <UNK>.

```
#Modification 1: remove stop words  
positive_words = [s for s in positive_words if s not in stop_word_id ]
```

- Removes Stop words as positive words

```
for i in range(len(all_tokens_repeat)):  
    if token_dict[all_tokens_repeat[i]] < min_token_freq:  
        all_tokens_repeat[i] = "<UNK>"
```

```
non_target_id = stop_word_id  
non_target_id.append(unk_index)
```

- Removes Stop words and <UNK>s as target words

```
for idx in tqdm(range(len(token_list))):  
    #Modification 3: Not a target for <UNK>  
    if (token_list[idx] not in non_target_id):
```

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

```

sum_of_counts = sum(self.word_counts.values())

prob_dict = {}
for token, freq in self.word_counts.items():
    z = self.word_counts[token] / sum_of_counts
    prob_dict[token] = (np.sqrt((z/0.001) + 1)) * (0.001/z)

```

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 $pk(w_i)$.

```

self.full_token_sequence_as_ids = []

for token in all_tokens_repeat:
    if token in self.word_counts.keys():
        if prob_dict[token] > np.random.binomial(1,0.5):
            self.full_token_sequence_as_ids.append(self.word_to_index[token])

```

Problem 11. Add tensorboard logging to your training loop so that you keep track of the sum of the losses for the past 100 steps and record the value with tensorboard

```

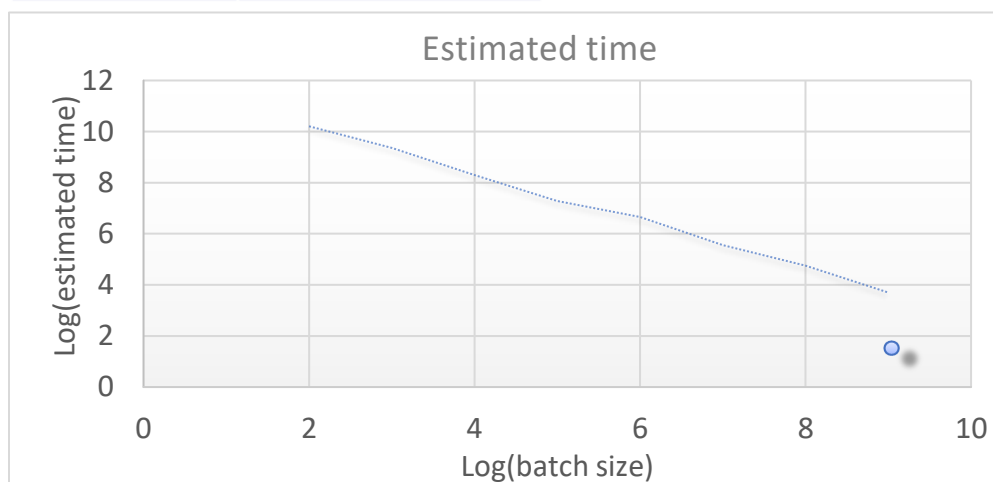
# TODO: Fill in all the training details here
outputs = model(target_ids, context_ids)
loss = criterion(torch.squeeze(outputs), labels.float())
writer.add_scalar("Loss/train", loss, epoch)
optimizer.zero_grad()
loss.backward()
optimizer.step()

# TODO: Based on the details in the Homework PDF, periodically
# report the running-sum of the loss to tensorboard. Be sure
# to reset the running sum after reporting it.
loss_sum += loss

```

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. For this, you'll set the parameter and then run the training long enough to get an estimate for both with tqdm wrapped around your batch iterator. You do not need to finish training for the full epoch. Make a plot where batch size is on the x-axis 10 and the tqdm-estimated time to finish one epoch is on the y-axis. (You may want to log-scale one or both of the axes). You can try other batch sizes too in this plot if you're curious. In your write up, describe what you see. What batch size would you choose to maximize speed? Side note: You might also want to watch your memory usage, as larger batches can sometimes dramatically increase memory.

Batch size	Estimated time
4	1183 sec
8	661 sec
16	314 sec
32	156 sec
64	101 sec
128	47 sec
256	27 sec
512	13 sec

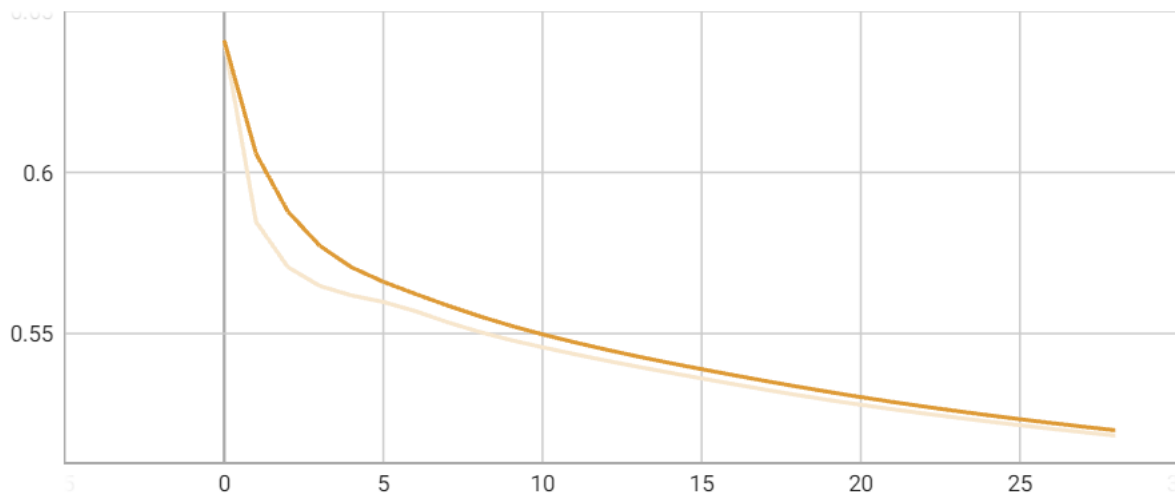


- The plot of $\log(\text{estimated time})$ vs $\log(\text{batch size})$ shows a decreasing linear plot

Problem 13. Train your model on at least one epoch worth of data. You are welcome to change the hyperparameters as you see fit for your final model (although batch size must be > 1). Record the full training process and save a picture of the tensorboard plot from your training run in your report. We need to see the plot. It will probably look something like Figure 1.

batch_size	2048
Epochs	30

Loss_sum/train



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.

```
NameError: name 'gensim' is not defined
```

```
AttributeError: 'Word2VecKeyedVectors' object has no attribute 'add_vectors'
```

- Not able to work with Gensim as it was not supported

Problem 15. Pick 10 target words and compute the most similar for each using Gensim's function. Qualitatively looking at the most similar words for each target word, do these predicted word 11 seem to be semantically similar to the target word? Describe what you see in 2-3 sentences. Hint: For maximum effect, try picking words across a range of frequencies (common, occasional, rare words).

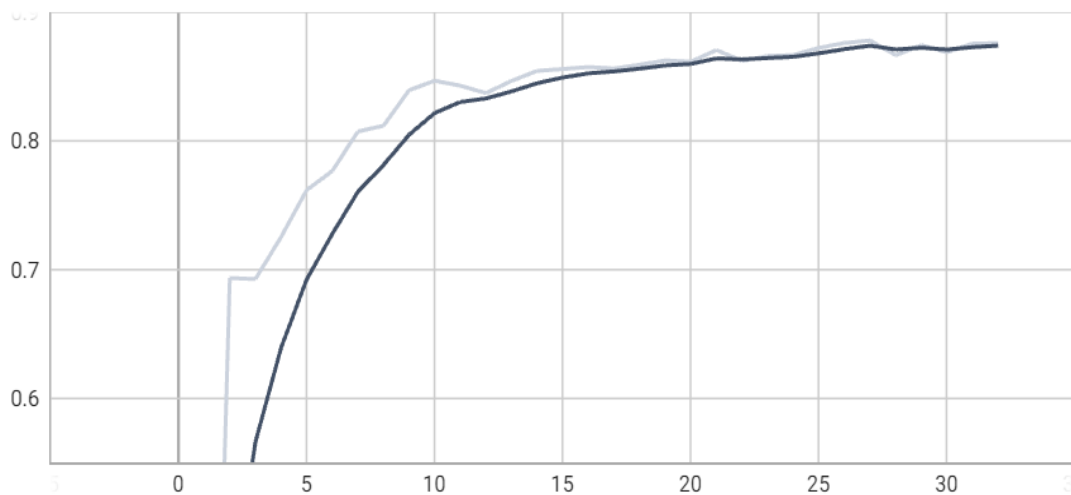
Problem 16. Given the analogy function, find five interesting word analogies with your word2vec model. For example, when representing each word by word vectors, we can generate the following equation, king - man + woman = queen. In other word, you can understand the equation as queen - woman = king - man, which mean the vectors similarity between queen and women is equal to king and man. What kinds of other analogies can you find? (NOTE: Any analogies shown in the class recording cannot be used for this problem.) What approaches worked and what approaches didn't? Write 2-3 seconds in a cell in the notebook.

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

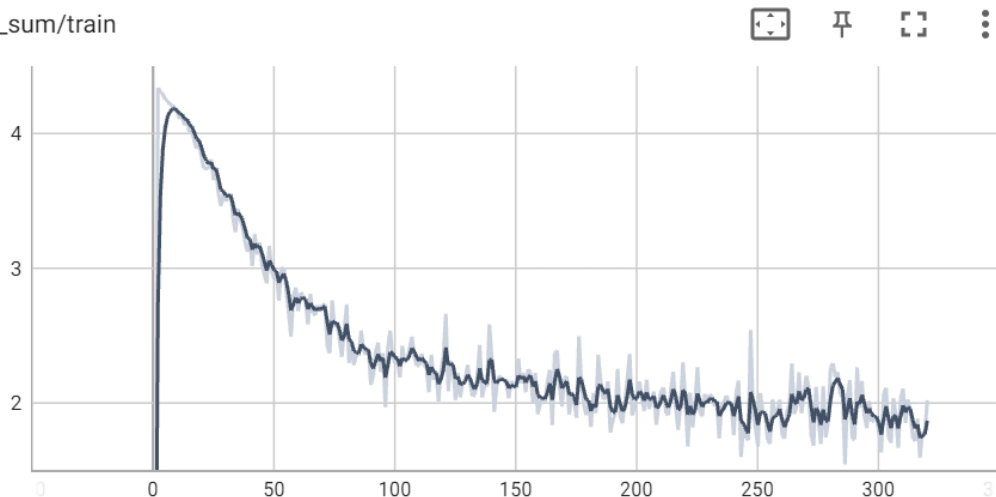
- Completed – check code file

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.

F1/train



Loss_sum/train

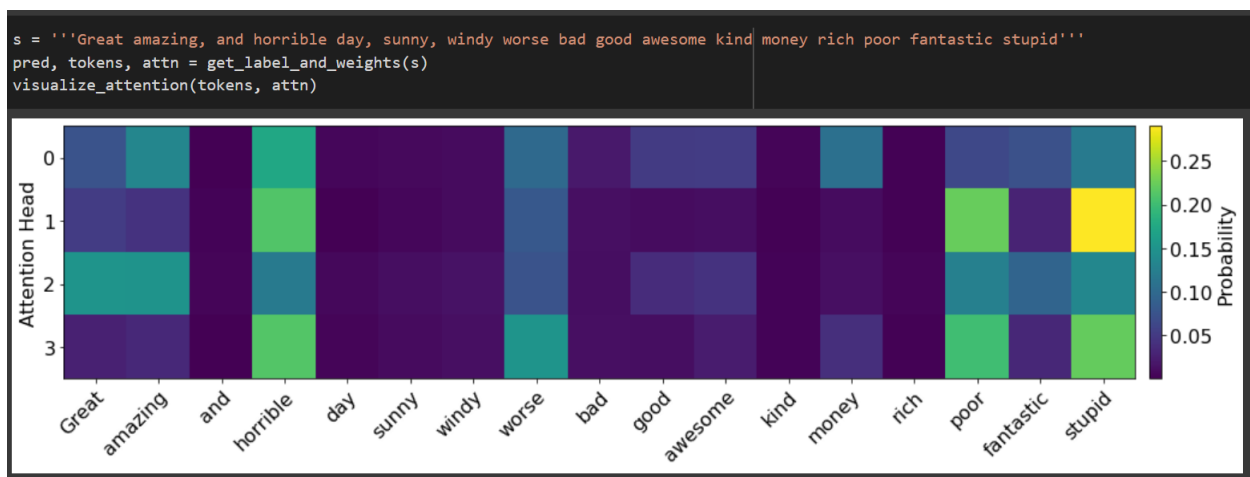


Problem 19. What if we didn't want to change the embeddings during the training process—i.e., we could keep our word meaning fixed and just train the network? This idea is known as freezing some parameters and can sometimes be a very helpful practice if you have a large set of pretrained vectors but your classifier's training data is small. In that setting, if you update the vectors during classifier-training, only some of the vectors get changed but the rest (for words not in the classifier training data) keep their old values from the word2vec pre-training, which can lead to the model not knowing how to interpret them as well. For this problem turn off gradient descent on the word vectors and retrain for one epoch. This training should be much faster since fewer parameters need updating. Show the plots of the loss and performance. In a few sentences, describe whether you think we should freeze our word vectors in this setting or not.

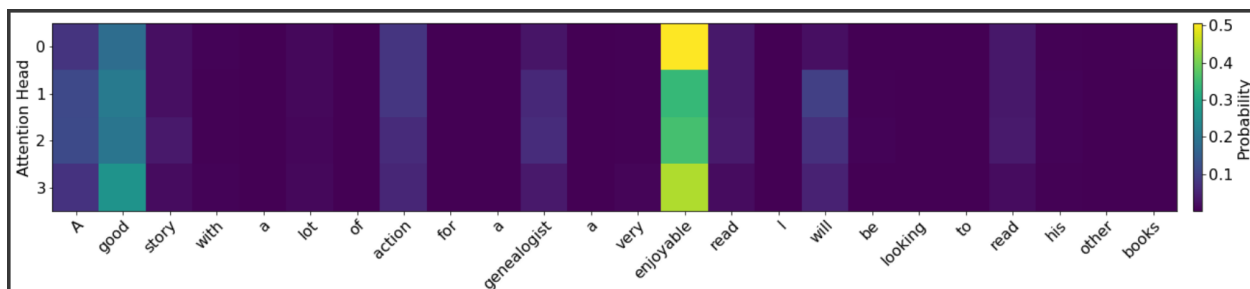
20. Predict the classifications for the test set and upload the scores to Kaggle. Be sure to include your Kaggle username in your report so we can give you credit.

- Score submitted in Kaggle competition, username – prathamesh0902

Problem 21. What is our attention doing? We’ve provided handy helpful function to visualize each head’s attention distributions across the input. Your tasks are: 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. 2. Using what you’ve observed from the visualizations, write a short paragraph describing what you think the attention heads are looking for. Describe any differences you see between heads and whether there are any patterns in terms of what they focus on. We encourage looking at many examples to get a sense of behavior (e.g., try randomly sampling text and visualizing the attention). 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. Show what the attention is looking at in this item. In a few sentences, describe whether the attention is looking at the right thing and whether the attention is a good explanation for why the classifier made the prediction for this item.

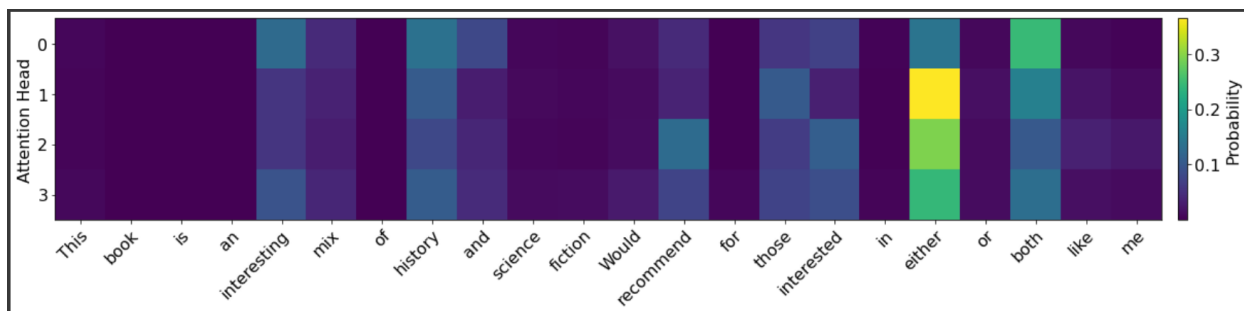


- The model with 4 Attention heads shows that mostly all the attention heads attend to highly positive and highly negative words.
- Head 1 and Head 3 attend to negative words more than Head 0 and Head 2.



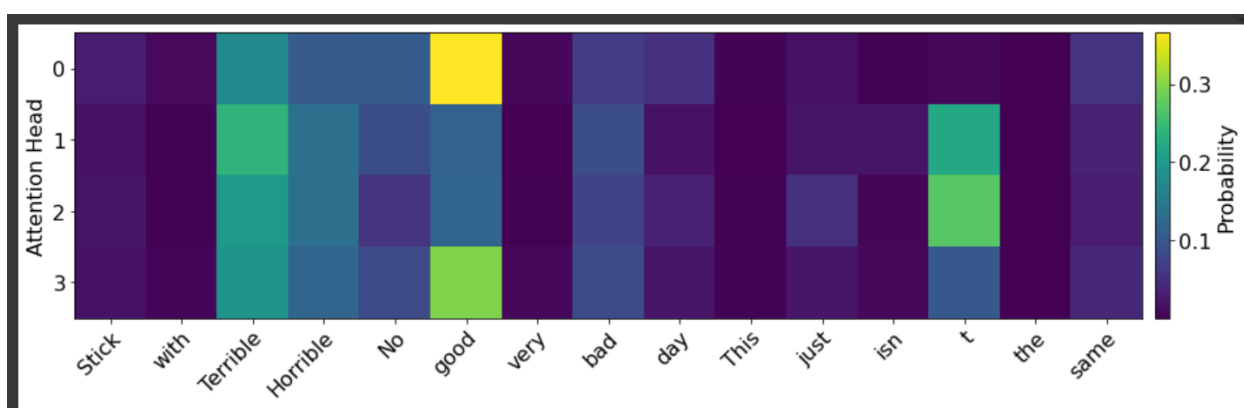
Positive Review 1

The attention heads attend only to positive words and are influenced even by presence of one of them.



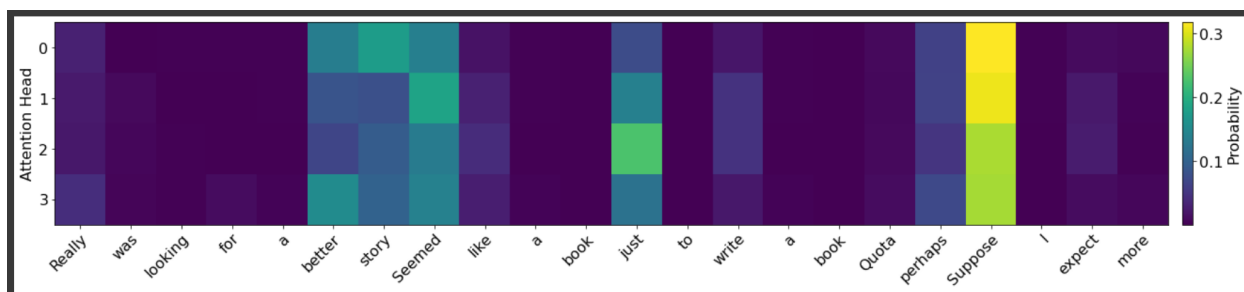
Positive Review 2

The attention is shifted from rather extreme positive words to words like 'either' and 'both' which show ambiguity



Negative Review 1

Rather than just identifying just negative words, the attention also identifies the positive words. This is likely due to the incorrect classification of word by the model



Negative Review 2

Here again the model is seemed confused in identifying the impact words.