CSCI 544- HOMEWORK 2
Asmita Chotani

NOTE: The Kaggle link shared in the homework document had most of the code written. To try something more and add onto my knowledge, along with dataloaders I have also showed results based on the word embedding array converted to tensors i.e the Xtrain, Xtest, Ytrain and Ytest datasets were converted into individual tensors and then passed on to the .forward(), and loss() functions

I observed that using dataloader performed better than using individual tensors, even though it took lesser time to process the epochs. It is not feasible to run with individual tensors for the data because not only does it take a long time, it also does not give the accuracy on the level of dataloaders. I also observed that the models performed better when the data was preprocessed. For reference I have submitted code for both the methods.

1. Dataset Generation

Like HW1, the ratings were categorized into classes 1,2, and 3. Initially, the index was reset to make the dataset accessible and the null values in "review body" was treated and replaced with a string(' '). Next two different datasets were created from it.

The first dataset "wv_data" was not preprocessed or cleaned. This dataset was used for the Word2Vec model and creating word embeddings for the future parts(different models being tested).
The second dataset "clean_data" was cleaned and preprocessed like HW1. This dataset was used to test the Simple Models using TFIDF features. The same hyperparameter values were considered which were obtained in HW1.

2. Word Embedding

(a)Try to check semantic similarities of the generated vectors using three examples of your own.

Exploring the genism model, I came across few functions that can be used to check for similarities- those were "most_similar" and "similarity", "cosine similarity" (I considered "most_similar").

E.g.:-

most_similar('Excellent') gave all the words similar to "excellent" and the probabilities
[('excellent', 0.6091997027397156), ('definition_redistributional', 0.575360119342804), ('Exceptional', 0.5664600729942322), ('flexible_hou_MORE', 0.5228071212768555), ('EXCELLENT',

0.521685779094696), ('Decent', 0.5081128478050232), ('Superb', 0.502091646194458), ('Terrific', 0.4998748004436493), ('Satisfactory', 0.4908524453639984), ('+_Bens', 0.48303356766700745)]

most_similar(positive=['she','father'], negative=['him']) gave all the words that were similar to she+father-him.
[('mother', 0.7119966745376587), ('husband', 0.6427904963493347), ('daughter', 0.6421711444854736), ('sister', 0.6059560179710388), ('eldest_daughter', 0.5988065004348755), ('grandmother', 0.5926641225814819), ('mom', 0.5790745615959167), ('aunt', 0.5724061131477356), ('niece', 0.5554639101028442), ('granddaughter', 0.5517464876174927)]

(b)Check the semantic similarities for the same two examples in part (a). What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better?

For part b I create my own word2vec model by passing the processed(splitting reviews based on space) review data to the Word2Vec function of genism.model.

For comparison purpose, cosine similarity was determined.

1. 'he' and 'she'
   my model: 0.8114579
   pretrained model: 0.6129949

   My model had better similarity than the pretrained model, the reason would be that they were more eminent in our corpus.

2. 'excellent' and 'outstanding'
   my model: 0.7313928
   pretrained model: 0.5567487

   My model had better similarity than the pretrained model, the reason would be that they were more eminent in our corpus. This makes sense as the review of items will 50% of the time contain positive reviews which would have synonyms of 'excellent'.

3. 'him' and 'father'-'she'
   my model: -0.7035967
   pretrained model: 0.0734347

   My model had better similarity than the pretrained model, the reason would be that they were more eminent in our corpus. This makes sense

as the review s of items can contain sentences like “my father bought this”, “I bought for my father as I wanted to buy a gift for him” “she liked it”

4. 'queen' and ‘king’- ‘woman’
   my model: error- "Key 'queen' not present"
   pretrained model: 0.2858241

   My model gave an error due to the absence of the keywords.

5. ‘Google’ and ‘Gmail’
   my model: error - "Key 'Gmail' not present"
   pretrained model: 0.68005306

   My model gave an error due to the absence of the keywords.

The model I created gave error for some of the examples, the error occurred because the words did not occur in my corpus which was created using the review data.
I feel the google corpus is larger and hence has more chances of words occurring

3. Simple models

Word Embeddings were created for the reviews in training and testing data after splitting the “wv_data” dataset. For embedding creation, a list was created for each review, splitting the review by space, and storing the separate words. Next, presence of each word was considered checked in the pretrained google W2V model and if present the value respective to that word was extracted and added to the NumPy array. As mentioned in Piazza, the words which did not occur in the pre trained model were not considered. To Normalize the values, the extracted values were later divided by the total length of the NumPy array. Such NumPy arrays of each review have been added to a list, which is being used as the input in the Perceptron, SVM and simple FNN as a NumPy array. Since no specific length was mentioned for this part, no limitations were applied and hence the array was created as it is (no padding with zeros or truncating).

(a)Report your accuracy values on the testing split for these models similar to HW1, i.e., for each of perceptron and SVM models, report two accuracy values Word2Vec and TF-IDF features.

a. Perceptron

TFIDF: 0.61
W2V: 0.49

b. SVM

TFIDF: 0.68
W2V: 0.63

(b)What do you conclude from comparing performances for the models trained using the two different feature types (TF-IDF and your trained

The TFIDF model performed better than the W2V model for both the simple models. One reason behind it could be that the W2V model has been trained by google corpus and the TFIDF model has been trained on our own dataset which was processed and cleaned.

4. Feedforward Neural Networks

(a)Report accuracy values on the testing split for your MLP.

The accuracy of the Feed Forward Network using the 80% 20% split was 64.08.

(b) To generate the input features, concatenate the first 10 Word2Vec vectors for each review as the input feature ($x = [W^T, ..., W^T]$) and train the neural network. Report the accuracy value on the testing split for your MLP model. What do you conclude by comparing accuracy values you obtain with those obtained in the "'Simple Models" section.

Similar to the word Embeddings in Part 3, new embeddings were created, but this time the 1st ten words of each review were considered only and were also concatenated, post that if the length of the NumPy vector was less than 10, it was padded with zeroes. In case of a null review, which was initially replaced with ' ', it was padded with 10 zeroes.

The accuracy of the Feed Forward Network by concatenating the 1st ten vectors for each reviews using the 80% 20% split was 54.94. Since we are only vectorizing only starting 10 words in concatenated feed forward, the simple feed forward performs better

As compared to the Simple Models, MLP performed better than both Perceptron and SVM.

5. Recurrent Neural Networks

Similar to the word Embeddings in Part 3, new embeddings were created, but this time the review length of 20 was considered. So, if the length of the review exceeded 20 words, only the first 20 were considered in the of each review and if the review length was less than 20, it was padded with zeros. Same as part 4, in case of a null review, which was initially replaced with ' ', it was padded with 10 zeroes.

(a)Train a simple RNN for sentiment analysis. What do you conclude by comparing accuracy values you obtain with those obtained with feedforward neural network models.

The accuracy of the Recurrent Neural Network using the 80% 20% split was 56.37.

Generally, RNN performs better than FNN. But in or case since we are only considering a review length of 20, and initial FNN is considering the whole review, the simple FNN is performing better. On the other hand, when compared to FNN using concatenated vectors (1st ten vectors), simple RNN(review length 20) performs better.

(b)Repeat part (a) by considering a gated recurrent unit cell.

The accuracy of the Gated Recurrent Neural Network using the 80% 20% split was 60.86.

(c)Repeat part (a) by considering an LSTM unit cell.

The accuracy of the Recurrent Neural Network using the 80% 20% split was 57.94.

(d)What do you conclude by comparing accuracy values you obtain by GRU, LSTM, and simple RNN.

With respect to RNN, GRU performed better than RNN and for the same number of epochs LSTM performed better than GRU. GRU uses 3 gates controlling the information, which improves the semantic similarity calculation.

The value of LSTM submitted is lower than GRU in this case because LSTM takes a lot of memory and due to limited resources, I could perform limited epochs only.

CSCI 544- HOMEWORK 2
Asmita Chotani

ACCURACIES

(The accuracies obtained in the jupyter notebook. Although I reached around 60 for LSTM, I do not have the model saved and the jupter notebook was changed.)

I have submitted the code with outputs for Individual tensors as well as the dataloaders with done after prreprocessing.

	USING Not processed data for W2V		USING processed data for W2V
MODEL	INDIVIDUAL TENSORS	DATALOADERS	DATALOADERS
FNN	64.08	62.125	64.25
FNN CONCAT	54.94	54.6	56.78
RNN	56.37	60.56	64.35
GRU	60.86	61.35	64.80
LSTM	57.94	58.38	61.88