Artificial Intelligence II Homework 4

Comments & Model Performance results on Question 1

Pavlos Spanoudakis (sdi1800184)

Basic Execution flow

1. We read the train & validation sets data from the input files, into `DataFrame`'s. The file paths can be modified on the notebook code cell #2. We also check whether all the samples have the expected format without missing values.

2. We create the Train and Validation sets, using `TweetDataset` objects. We tokenize each sample using `BertTokenizer`, for the pretrained `bert-base-uncased` model.

3. We will access the two sets in batches, using `DataLoader` objects.

4. We initialize the Model. Hyperparameters such as number of epochs, batch size, learning rate & Dropout probability can be modified on code cell #4. We use `CrossEntropyLoss` and the `Adam` optimizer.

5. We train the model: We use `list`s to store several performance stats during training, such as Loss and Accuracy on Train and Validation sets after each epoch.
   During each epoch:

   - For each batch given by the train set `BucketIterator`:
     - We make predictions on this batch
     - Extract the predicted labels & calculate the accuracy
     - Calculate & store the batch Loss
     - Perform backpropagation
   - After going through all the batches, we calculate the total accuracy & total Loss for the Train set in the epoch.
   - We perform the same actions on the validation set, this time without performing backpropagation of course.
   - During the final epoch, we store the predicted labels, as well as the model output (on both sets), to use in the final evaluation phase right after.

6. Displaying performance results: After the end of training we display:

   - The Confusion Matrices of the final model predictions on the Train and Validation sets.
   - The F1, Precision and Recall scores for the final model on both sets.
     We use the corresponding **scikit-learn** routines for calculating the scores. We use `average='micro'` for F1 score and `average='macro'` for Precision & Recall scores.
   - The ROC Curves for the Validation set predictions.
     We mirror the usage of `roc_curve` from **scikit-learn** for multiple classes, as demonstrated here. To create the curves, we apply the `softmax` function to the NN output vector, to convert it to possibility values that add-up to 1, and use the `softmax` output to create the curves. `roc_curve` applies generated possibility thresholds to create the curves, therefore if we provided

it with just the predicted labels, it would only apply 3 thresholds to each result, which is insufficient to create useful ROC curves.

Model Architecture

Different models performance comparison

Notes on all models:

- The performance results displayed below have been produced without using GPU acceleration.
- When not using GPU, the results can be reproduced using `SEED = 42`.
- Unfortunately, GPU-accelerated model results could not be fully reproduced.
- The execution of each model using only CPU takes about 4-5 minutes. In case this is inconvenient, GPU can be enabled in code cell #6, but note that the results for all models will differ from the ones presented below.
- All models use the **GloVe** pre-trained word embeddings from `glove.6B.100d.txt`.

1. This is the preselected model in the interactive notebook. For this model, we use:
    - Learning rate: 0.005
    - Batch Size: 256
    - # Epochs: 5
    - RNN: Bidirectional, 4-layer **GRU**, with `hidden_size = 32`
    - Dropout probability: 0.5

Comments/Observations on the models and their develpoment

- We notice that bidirectional models tend to perform better. They learn more quickly (in terms of epochs) and perform better in each seperate class, which affects the **Precision** and especially the **Recall** score.
- Skip connections did not seem to have any significant effect in the model performance.
- GRU models achieved slightly better performance, possibly due to their simplicity, which prevents overfitting in comparison to LSTM's.
- Decreasing the Dropout probability and/or the batch size usually resulted in higher tendency for overfitting.
- Using **GloVe** embeddings with vector size = 100 appeared to be just about right. Higher vector size would cause significant overfitting in most cases.
- Just like in the previous 2 homeworks, we see much better performance on `Neutral` and `Pro-Vaccine` tweets in all models, since the vast amount of train set tweets are labeled as such.
- However, the ROC curves show that while the model may have trouble classifying `Anti-Vaccine` tweets correctly, the possibility assigned to this class when the prediction is wrong is not small. In other

words, the model may be **misclassifying** the `Anti-Vaccine` tweets, but **it is not very "confident"** in these cases.

Development

- The notebook has been developed mostly in Google Colab and was based on the Projects 2 & 3 notebooks.
- It has been tested successfully in Google Colab & Kaggle environments, using GPU-accelerated runtimes.
    - **Note**: In Kaggle, Confusion Matrices code crashes because `ConfusionMatrixDisplay.from_predictions` appears to be unavailable.