CSOC'25 IG Assignment-3

Natural Language Processing

Presented By- Juhi Saxena
Roll No: 24095044

Summary of the Task

1. The task assigned was centred around a binary classification problem based on textual data processing, i.e assigning a label to a review based on its title and description whilst utilizing the *Amazon Reviews Dataset*

2. The primary aim was basic sequence modelling and assessing varied architectures based on their ability of context retention, semantic interpretation and analysis. The two architectures deeply explored in our case are RNNs and LSTM

Approach Implemented

Data Sampling and Trade off between Information and Computational resources

The original dataset consists of 3.6 million training samples in the training csv. This roughly amounts to 2 GB of data and processing such enormous amounts of data often requires heavy memory consumption and impractical amount of VRAM and RAM. A potential solution could have been breaking the data into chunks and batches and then process, but that amounts to hours of training and pre-processing which is beyond the technical limitations of an ordinary system with unavailiblity of a GPUs computational and processing prowess or even colab's free T4 GPU access(pertaining to time constraints of usage with respect to hyperparameter tuning.)
Thus the best possible approach was to sample a chunk of data for both the training and the testing datasets.

Data Sampling Method- Stratified Sampling

Stratified Data Sampling- Retaining the class proportion

Upon analysing the original dataset it was found out that the reviews primarily belong to two classes- class 1 and 2. Now the number of samples belonging to class 1 were found out to be- 1799880 and the number of samples falling under class 2 were found out to be-1799912. From this piece of information, we can easily conclude that the ratio of class 1 samples to class 2 samples is nearly 1:1, the dataset is pretty balanced.

Method of Execution

Stratified sampling refers to the extraction of a subset of the original dataset such that the class proportion is retained. This way the chances of class imbalance creeping in are nullified

Data Sampling-Approach continued

Firstly, we decide upon to extract 1.5 % of the training data as our sampled training data- 1.5% roughly contains 54,000 training samples. Now in order to retain the class proportion, 27,000 class 1 samples have been employed along with an equal number of class 2 samples. These two are independently and in a randomized manner extracted in two separated data frames which are subsequently concatenated and then reshuffled to form our final sampled training dataset. The same approach is applied to the test data.

Is an ordinary csv the best option?

- In our implementation, we have utilized the dataset format provided by the Hugging Face library, which is a de facto standard in NLP pipelines due to its seamless integration with transformer-based models.

Data Sampling- Approach Continued

What's so special about this format?

- Hugging Face provides a highly efficient and computation-friendly tabular dataset format that is optimized for machine learning tasks. One of its major advantages is that datasets can be seamlessly converted into Apache Arrow format, which allows for efficient compression and lightweight storage. This makes it ideal for exporting and deploying data to cloud-based ML environments.
- The Arrow format offers substantial benefits over traditional formats like CSV. It stores data in a columnar structure, enabling faster I/O, lower memory usage, and better support for large-scale, distributed processing.
- In our pipeline, we have leveraged BERT's pretrained tokenizer for preprocessing, and BERT's embeddings for downstream model training. The Hugging Face datasets format is the most compatible and efficient structure for integrating with models like BERTModel from the transformers library.

Data Pre-processing

- After sampling the dataset, we applied several standard NLP preprocessing steps, such as removing punctuation marks. However, we intentionally skipped lowercasing, as the tokenizer from the **"bert-base-uncased"** model (part of the BERT family) automatically handles lowercasing internally during tokenization.
- We initially considered incorporating *lemmatization* and *stop word removal*, but due to their computational overhead and the fact that omitting them did not significantly affect performance, we opted to exclude them from the pipeline. *Title and description have been combined to a single column(text_review).*
- Batched tokenization is done using the AutoTokenizer class of the HuggingFace. Besides, truncation has been applied to trim extremely long sequences to a standard length along with padding of short sequences to maximum length of 512 so as to ensure uniformity in sequence length. This is key to the usage of DataLoader class for batching and loading data for training as tensors of same length are required to be stacked.

RNN implementation

- The dataset is loaded using PyTorch's Dataset and DataLoader classes, with a batch size of 256. To ensure efficient GPU utilization, the pin_memory parameter is enabled for CUDA acceleration, and multi-threading is introduced by setting num_workers=2 to use two CPU cores during data loading. This batch-wise training strategy helps in preventing RAM exhaustion, ensuring efficient memory usage, maintaining a stable and even loss distribution across iterations
- The architecture utilized is a simple RNN. The first layer, i.e the embedding layer includes pretrained embeddings (768 dim) of the "bert-base-uncased"model. This layer is frozen and not a part of the training process. The next is a hidden layer of 128 neurons and the final output layer which is of single dimension. Sigmoid activation has not been applied in this definition since we calculate BCEWithLogits Loss while training.

Final Results- *RNN Architecture*

- Training is done for 5 epochs with a learning rate of 0.003 and using the Adam optimizer. The model along with the loaded data is moved to "CUDA"to account for GPU acceleration
- Final Accuracy obtained on the test data- 85.10
- Final F1-score obtained on the test data -0.848

Final Results- *LSTM Architecture* I

- Everything remains consistent with the RNN class, apart from the fact that we use an LSTM implementation with the same dimension for the hidden layer- 128
- Final accuracy obtained on the test dataset- 86.60
- Final F1-score obtained on the test dataset- 0.8542

Comparing the Two Architectures

Aspect	**RNN**	**LSTM**
Training Stability	Acceptable	More stable (less noisy)
Generalization	Decent	Better on unseen test data
Suitability for NLP	Decent, but outdated	Preferred for deep sequence models
Scalability	Harder to scale	Scales better to deeper layers or bidirectionality
F1-Score	0.848	0.8542
Accuracy	85.10%	86.60%

Tabela 1:

Why a slight difference?

- LSTM gives a better result slightly in our use case- LSTMs generally are empowered with a better context retention and critical capacity as against RNNs. Though a slight difference can be pertaining to the use of pre-trained embeddings. The RNN and LSTM architectures themselves do not compute the embeddings which could have brought in a significant difference in the accuracy scores pertaining to the long term context retention capacity of LSTMs and a way better semantic analysis than the RNNs.
- Yet the dominance of LSTMs can still be seen, even when using a single layer RNN to avoid vanishing gradient as much as possible, hence the conclusion- LSTMs are a better choice for text processing tasks.

-Resources Utilized- Resources on COPS IG(GitHub Repo)
-Kaggle
-ChatGPT for grammatical errors and theoretical questions.
Thank You!