Assignment 1

Problem Definition:

In this assignment, our objective is to perform text classification into five distinct classes using the Stanford Sentiment Treebank (SST) dataset. The task involves employing two different classification approaches: Naive Bayes and Logistic Regression. Additionally, there is a third part dedicated to evaluation, where we will generate a confusion matrix and derive performance metrics from it.

Dataset Description:

The dataset is sourced from the Stanford Sentiment Treebank (SST), a collection of movie reviews labelled with sentiment score (a real value) and some other annotations that are irrelevant to our use case.

Preprocessing:

The dataset consists of text samples that are scored on a scale from 0 to 1, representing the sentiment expressed within each sample. Your task is to perform sentiment classification by categorizing these text samples into five distinct classes.

You are required to map the scores as mentioned below:

- From 0 to 0.2 (0.2 included) will be class 0 “very negative”.
- From 0.2 to 0.4 (0.4 included) will be class 1 “negative”.
- From 0.4 to 0.6 (0.6 included) will be class 2 “neutral”.
- From 0.6 to 0.8 (0.8 included) will be class 3 “positive”.
- From 0.8 to 1.0 (1.0 included) will be class 4 “very positive”.

Part 1: Naïve Bayes

Algorithm Implementation:

In this part you should implement the algorithm from scratch using NumPy only without any extra libraries, you just need to exactly implement the algorithm attached below from chapter 4

function TRAIN NAIVE BAYES(D, C) **returns** log $P(c)$ and log $P(w|c)$

for each class $c \in C$ # Calculate $P(c)$ terms
  N_{doc} = number of documents in D
  N_c = number of documents from D in class c
  $logprior[c] \leftarrow \log \frac{N_c}{N_{doc}}$
  $V \leftarrow$ vocabulary of D
  $bigdoc[c] \leftarrow$ **append**(d) **for** d $\in$ D **with** class c
  for each word w in V # Calculate $P(w|c)$ terms
    $count(w,c) \leftarrow$ # of occurrences of w in $bigdoc[c]$
    $loglikelihood[w,c] \leftarrow \log \frac{count(w,c) + 1}{\sum_{w' \text{ in } V} (count(w',c) + 1)}$
return $logprior$, $loglikelihood$, V

function TEST NAIVE BAYES($testdoc$, $logprior$, $loglikelihood$, C, V) **returns** best c

for each class $c \in C$
  $sum[c] \leftarrow logprior[c]$
  for each position i in $testdoc$
    $word \leftarrow testdoc[i]$
    if $word \in V$
      $sum[c] \leftarrow sum[c] + loglikelihood[word,c]$
return $\text{argmax}_c\ sum[c]$

Figure 4.2 The naive Bayes algorithm, using add-1 smoothing. To use add-α smoothing instead, change the +1 to +α for loglikelihood counts in training.

Comparison with scikit learn:

After finishing your implementation, you should reproduce the same results using scikit learn library, the simplest way is to use a pipeline of CountVectorizer followed by MultinomialNB, but you will need to tweak some parameters so that the results are exactly the same.

Note: For comparison you will need your functions from part 3.

Part 2: Logistic Regression

Feature Representation:

In this part you should generate the features for each sentence, the word bi-grams features, recall Andrej Karpahthy Tutorial.

As an example, the sentence “I love this movie very much” has 5 word bi-gram features namly (‘I’, ‘love’), (‘love’, ‘this’) and so on. Each sentence should be represented with a vector of length equal to the

number of unique word bi-grams in the whole dataset with 1 at the corresponding index if the bi-gram exists and 0 otherwise, recall that this feature representation is sparse.

Algorithm Implementation:

In this part you should implement the algorithm from scratch using NumPy only without any extra libraries, you should follow the Reading from the course in details chapter 5

Comparison with scikit learn:

After Implementing you version of logistic regression you should compare it with scikit learn implementation, check this and this, try both and leave your comments, why one is better than the other and what is the one that is comparable to your implementation. When using this, you should change some of the default parameters to use it as logistic regression.

Note: Again, this part should be done after part 3 to compare the metrics.

Extra Notes:

- You have limited memory in Colab ~ 12Gb so
  - Take care of the data types, you shouldn't use a datatype with unnecessary precision. For example, don't use float32 if int16 is enough.
  - Also, most probably, you will have to do some memory optimization deleting some variables yourself in your training loop using *del* command in python.
- Don't forget to shuffle your data every epoch so that to enforce the stochasticity of the SGD.
- Your implementation should be a batched one, so your calculations should be vectorized, not as for loops.

Part 3: Confusion Matrix & Evaluation Metrics

In this Part you should

- Implement a function to generate the confusion matrix given *the predictions* and *the ground truth labels*.
- Compute the following metrics from the generated confusion matrix, the precision, recall and F1 score *per class* and *macro averaged*.
- You shouldn't use any libraries except NumPy in this part.
- After Implementing the previous requirements, you should compare it with scikit learn outputs, check this.

Grading Policy:

- This project should be done in ***teams of two***.
- We know that this assignment is a little bit tough, but the objective is to get your hands dirty.
- ***Deadline is due Friday, March 15, 2024.***
- Grading Scheme
  - Part 1 (35%)
    - Algorithm Implementation (30%)
      - Train Function (25%)
      - Test Function (5%)

    - Comparison with scikit learn (5%)
  - Part 2 (50%)
    - Feature Representation (10%)
    - Algorithm Implementation (35%)
    - Comparison with scikit learn (5%)
  - Part 3 (15%)