assignment1

February 11, 2021

1 [COM6513] Assignment 1: Text Classification with Logistic Regression

1.0.1 Instructor: Nikos Aletras

The goal of this assignment is to develop and test two text classification systems:

- **Task 1:** sentiment analysis, in particular to predict the sentiment of movie reviews, i.e. positive or negative (binary classification).
- **Task 2:** topic classification, to predict whether a news article is about International issues, Sports or Business (multi-class classification).

For that purpose, you will implement:

- Text processing methods for extracting Bag-Of-Word features, using
  - n-grams (BOW), i.e. unigrams, bigrams and trigrams to obtain vector representations of documents where n=1,2,3 respectively. Two vector weighting schemes should be tested: (1) raw frequencies (**3 marks**); (2) tf.idf (**1 mark**).
  - character n-grams (BOCN). A character n-gram is a contiguous sequence of characters given a word, e.g. for n=2, 'coffee' is split into {'co', 'of', 'ff', 'fe', 'ee'}. Two vector weighting schemes should be tested: (1) raw frequencies (**3 marks**); (2) tf.idf (**1 mark**). **Tip: Note the large vocabulary size!**
  - a combination of the two vector spaces (n-grams and character n-grams) choosing your best performing wighting respectively (i.e. raw or tfidf). (**3 marks**) **Tip: you should merge the two representations**

- Binary Logistic Regression (LR) classifiers for Task 1 that will be able to accurately classify movie reviews trained with:
  - (1) BOW-count (raw frequencies)
  - (2) BOW-tfidf (tf.idf weighted)
  - (3) BOCN-count
  - (4) BOCN-tfidf
  - (5) BOW+BOCN (best performing weighting; raw or tfidf)

- Multiclass Logistic Regression classifiers for Task 2 that will be able to accurately classify news articles trained with:
  - (1) BOW-count (raw frequencies)
  - (2) BOW-tfidf (tf.idf weighted)
  - (3) BOCN-count
  - (4) BOCN-tfidf
  - (5) BOW+BOCN (best performing weighting; raw or tfidf)

- The Stochastic Gradient Descent (SGD) algorithm to estimate the parameters of your Logistic Regression models. Your SGD algorithm should:
  - Minimise the Binary Cross-entropy loss function for Task 1 (**3 marks**)
  - Minimise the Categorical Cross-entropy loss function for Task 2 (**3 marks**)
  - Use L2 regularisation (**2 marks**)
  - Perform multiple passes (epochs) over the training data (**1 mark**)
  - Randomise the order of training data after each pass (**1 mark**)
  - Stop training if the difference between the current and previous development loss is smaller than a threshold (**1 mark**)
  - After each epoch print the training and development loss (**1 mark**)

- Discuss how did you choose hyperparameters (e.g. learning rate and regularisation strength) for each LR model? You should use a table showing model performance using different set of hyperparameter values. (**5 marks; 2.5 for each task**). **Tip: Instead of using all possible combinations, you could perform a random sampling of combinations.**

- After training each LR model, plot the learning process (i.e. training and validation loss in each epoch) using a line plot. Does your model underfit, overfit or is it about right? Explain why. (**2 marks**).

- Identify and show the most important features (model interpretability) for each class (i.e. top-10 most positive and top-10 negative weights). Give the top 10 for each class and comment on whether they make sense (if they don't you might have a bug!). If you were to apply the classifier into a different domain such laptop reviews or restaurant reviews, do you think these features would generalise well? Can you propose what features the classifier could pick up as important in the new domain? (**3 marks; 1.5 for each task**)

- Provide well documented and commented code describing all of your choices. In general, you are free to make decisions about text processing (e.g. punctuation, numbers, vocabulary size) and hyperparameter values. We expect to see justifications and discussion for all of your choices (**5 marks**).

- Provide efficient solutions by using Numpy arrays when possible. Executing the whole notebook with your code should not take more than 10 minutes on a any standard computer (e.g. Intel Core i5 CPU, 8 or 16GB RAM) excluding hyperparameter tuning runs. You can find tips in Intro to Python for NLP (**2 marks**).

1.0.2 Data - Task 1

The data you will use for Task 1 are taken from here: http://www.cs.cornell.edu/people/pabo/movie-review-data/ and you can find it in the `./data_sentiment` folder in CSV format:

- `data_sentiment/train.csv`: contains 1,400 reviews, 700 positive (label: 1) and 700 negative (label: 0) to be used for training.
- `data_sentiment/dev.csv`: contains 200 reviews, 100 positive and 100 negative to be used for hyperparameter selection and monitoring the training process.
- `data_sentiment/test.csv`: contains 400 reviews, 200 positive and 200 negative to be used for testing.

1.0.3 Data - Task 2

The data you will use for Task 2 is a subset of the AG News Corpus and you can find it in the `./data_topic` folder in CSV format:

- `data_topic/train.csv`: contains 2,400 news, 800 for each class to be used for training.
- `data_topic/dev.csv`: contains 150 news articles, 50 for each class to be used for hyperparameter selection and monitoring the training process.
- `data_topic/test.csv`: contains 900 news articles, 300 for each class to be used for testing.

1.0.4 Submission Instructions

You should submit a Jupyter Notebook file (assignment1.ipynb) and an exported PDF version (you can do it from Jupyter: `File->Download as->PDF via Latex` or you can print it as PDF using your browser).

You are advised to follow the code structure given in this notebook by completing all given funtions. You can also write any auxilliary/helper functions (and arguments for the functions) that you might need but note that you can provide a full solution without any such functions. Similarly, you can just use only the packages imported below but you are free to use any functionality from the Python Standard Library, NumPy, SciPy (excluding built-in softmax funtcions) and Pandas. You are not allowed to use any third-party library such as Scikit-learn (apart from metric functions already provided), NLTK, Spacy, Keras etc.. You should mention if you've used Windows to write and test your code because we mostly use Unix based machines for marking (e.g. Ubuntu, MacOS).

There is no single correct answer on what your accuracy should be, but correct implementations usually achieve F1-scores around 80% or higher. The quality of the analysis of the results is as important as the accuracy itself.

This assignment will be marked out of 40. It is worth 40% of your final grade in the module.

The deadline for this assignment is **23:59 on Fri, 25 Feb 2021** and it needs to be submitted via Blackboard. Standard departmental penalties for lateness will be applied. We use a range of strategies to **detect unfair means**, including Turnitin which helps detect plagiarism. Use of unfair means would result in getting a failing grade.

In [ ]:
```
import pandas as pd
import numpy as np
from collections import Counter
import re
```

```
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import random

# fixing random seed for reproducibility
random.seed(123)
np.random.seed(123)
```

1.1 Task 1: Binary classification

1.2 Load Raw texts and labels into arrays

First, you need to load the training, development and test sets from their corresponding CSV files (tip: you can use Pandas dataframes).

In [ ]:

If you use Pandas you can see a sample of the data.

The next step is to put the raw texts into Python lists and their corresponding labels into NumPy arrays:

2 Vector Representations of Text

To train and test Logisitc Regression models, you first need to obtain vector representations for all documents given a vocabulary of features (unigrams, bigrams, trigrams).

2.1 Text Pre-Processing Pipeline

To obtain a vocabulary of features, you should: - tokenise all texts into a list of unigrams (tip: using a regular expression) - remove stop words (using the one provided or one of your preference) - compute bigrams, trigrams given the remaining unigrams (or character ngrams from the unigrams) - remove ngrams appearing in less than K documents - use the remaining to create a vocabulary of unigrams, bigrams and trigrams (or character n-grams). You can keep top N if you encounter memory issues.

```
In [ ]: stop_words = ['a','in','on','at','and','or',
                       'to', 'the', 'of', 'an', 'by',
                       'as', 'is', 'was', 'were', 'been', 'be',
                       'are','for', 'this', 'that', 'these', 'those', 'you', 'i',
                      'it', 'he', 'she', 'we', 'they' 'will', 'have', 'has',
                       'do', 'did', 'can', 'could', 'who', 'which', 'what',
                      'his', 'her', 'they', 'them', 'from', 'with', 'its']
```

2.1.1 N-gram extraction from a document

You first need to implement the `extract_ngrams` function. It takes as input: - `x_raw`: a string corresponding to the raw text of a document - `ngram_range`: a tuple of two integers denoting the type of ngrams you want to extract, e.g. (1,2) denotes extracting unigrams and bigrams. - `token_pattern`: a string to be used within a regular expression to extract all tokens. Note that

data is already tokenised so you could opt for a simple white space tokenisation. - `stop_words`: a list of stop words - `vocab`: a given vocabulary. It should be used to extract specific features. - `char_ngrams`: boolean. If true the function extracts character n-grams

and returns:

- 'x': a list of all extracted features.

See the examples below to see how this function should work.

```
In [ ]: def extract_ngrams(x_raw, ngram_range=(1,3), token_pattern=r'',
                           stop_words=[], vocab=set(), char_ngrams=False):



            return x
```

Note that it is OK to represent n-grams using lists instead of tuples: e.g. `['great', ['great', 'movie']]`

For extracting character n-grams the function should work as follows:

```
In [ ]: extract_ngrams("movie",
                       ngram_range=(2,4),
                       stop_words=[],
                       char_ngrams=True)
```

2.1.2 Create a vocabulary

The `get_vocab` function will be used to (1) create a vocabulary of ngrams; (2) count the document frequencies of ngrams; (3) their raw frequency. It takes as input: - `X_raw`: a list of strings each corresponding to the raw text of a document - `ngram_range`: a tuple of two integers denoting the type of ngrams you want to extract, e.g. (1,2) denotes extracting unigrams and bigrams. - `token_pattern`: a string to be used within a regular expression to extract all tokens. Note that data is already tokenised so you could opt for a simple white space tokenisation. - `stop_words`: a list of stop words - `min_df`: keep ngrams with a minimum document frequency. - `keep_topN`: keep top-N more frequent ngrams.

and returns:

- `vocab`: a set of the n-grams that will be used as features.
- `df`: a Counter (or dict) that contains ngrams as keys and their corresponding document frequency as values.
- `ngram_counts`: counts of each ngram in vocab

Hint: it should make use of the `extract_ngrams` function.

```
In [ ]: def get_vocab(X_raw, ngram_range=(1,3), token_pattern=r'',
                       min_df=0, keep_topN=0,
                       stop_words=[],char_ngrams=False):



            return vocab, df, ngram_counts
```

Now you should use `get_vocab` to create your vocabulary and get document and raw frequencies of n-grams:

```
In [ ]:
```

Then, you need to create 2 dictionaries: (1) vocabulary id -> word; and (2) word -> vocabulary id so you can use them for reference:

Now you should be able to extract n-grams for each text in the training, development and test sets:

2.2 Vectorise documents

Next, write a function `vectoriser` to obtain Bag-of-ngram representations for a list of documents. The function should take as input: - `X_ngram`: a list of texts (documents), where each text is represented as list of n-grams in the `vocab` - `vocab`: a set of n-grams to be used for representing the documents

and return: - `X_vec`: an array with dimensionality Nx|vocab| where N is the number of documents and |vocab| is the size of the vocabulary. Each element of the array should represent the frequency of a given n-gram in a document.

```
In [ ]: def vectorise(X_ngram, vocab):


            return X_vec
```

Finally, use `vectorise` to obtain document vectors for each document in the train, development and test set. You should extract both count and tf.idf vectors respectively:

Count vectors

TF.IDF vectors First compute `idfs` an array containing inverted document frequencies (Note: its elements should correspond to your `vocab`)

Then transform your count vectors to tf.idf vectors:

3 Binary Logistic Regression

After obtaining vector representations of the data, now you are ready to implement Binary Logistic Regression for classifying sentiment.

First, you need to implement the `sigmoid` function. It takes as input:

- `z`: a real number or an array of real numbers

and returns:

- `sig`: the sigmoid of `z`

```
In [ ]: def sigmoid(z):


            return sig
```

Then, implement the `predict_proba` function to obtain prediction probabilities. It takes as input:

- `X`: an array of inputs, i.e. documents represented by bag-of-ngram vectors ($N \times |vocab|$)
- `weights`: a 1-D array of the model's weights $(1, |vocab|)$

and returns:

- `preds_proba`: the prediction probabilities of X given the weights

```
In [ ]: def predict_proba(X, weights):



            return preds_proba
```

Then, implement the `predict_class` function to obtain the most probable class for each vector in an array of input vectors. It takes as input:

- `X`: an array of documents represented by bag-of-ngram vectors ($N \times |vocab|$)
- `weights`: a 1-D array of the model's weights $(1, |vocab|)$

and returns:

- `preds_class`: the predicted class for each x in X given the weights

```
In [ ]: def predict_class(X, weights):



            return preds_class
```

To learn the weights from data, we need to minimise the binary cross-entropy loss. Implement `binary_loss` that takes as input:

- `X`: input vectors
- `Y`: labels
- `weights`: model weights
- `alpha`: regularisation strength

and return:

- `l`: the loss score

```
In [ ]: def binary_loss(X, Y, weights, alpha=0.00001):
            '''
            Binary Cross-entropy Loss

            X:(len(X),len(vocab))
            Y: array len(Y)
            weights: array len(X)
```

```
        '''


        return l
```

Now, you can implement Stochastic Gradient Descent to learn the weights of your sentiment classifier. The `SGD` function takes as input:

- `X_tr`: array of training data (vectors)
- `Y_tr`: labels of `X_tr`
- `X_dev`: array of development (i.e. validation) data (vectors)
- `Y_dev`: labels of `X_dev`
- `lr`: learning rate
- `alpha`: regularisation strength
- `epochs`: number of full passes over the training data
- `tolerance`: stop training if the difference between the current and previous validation loss is smaller than a threshold
- `print_progress`: flag for printing the training progress (train/validation loss)

and returns:

- `weights`: the weights learned
- `training_loss_history`: an array with the average losses of the whole training set after each epoch
- `validation_loss_history`: an array with the average losses of the whole development set after each epoch

```
In [ ]: def SGD(X_tr, Y_tr, X_dev=[], Y_dev=[], lr=0.1,
                  alpha=0.00001, epochs=5,
                  tolerance=0.0001, print_progress=True):


            return weights, training_loss_history, validation_loss_history
```

3.1 Train and Evaluate Logistic Regression with Count vectors

First train the model using SGD:

Now plot the training and validation history per epoch for the best hyperparameter combination. Does your model underfit, overfit or is it about right? Explain why.

Explain here...

Evaluation Compute accuracy, precision, recall and F1-scores:

```
In [ ]: preds_te_count = predict_class(X_te_count, w_count)

        print('Accuracy:', accuracy_score(Y_te,preds_te_count))
        print('Precision:', precision_score(Y_te,preds_te_count))
```

```
print('Recall:', recall_score(Y_te,preds_te_count))
print('F1-Score:', f1_score(Y_te,preds_te_count))
```

Finally, print the top-10 words for the negative and positive class respectively.

```
In [ ]: top_neg = w_count.argsort()[:10]
        for i in top_neg:
            print(id2word[i])
```

```
In [ ]: top_pos = w_count.argsort()[::-1][:10]
        for i in top_pos:
            print(id2word[i])
```

If we were to apply the classifier we've learned into a different domain such laptop reviews or restaurant reviews, do you think these features would generalise well? Can you propose what features the classifier could pick up as important in the new domain?

Provide your answer here...

3.1.1 Discuss how did you choose model hyperparameters (e.g. learning rate and regularisation strength)? What is the relation between training epochs and learning rate? How the regularisation strength affects performance?

Enter your answer here...

3.2 Train and Evaluate Logistic Regression with TF.IDF vectors

Follow the same steps as above (i.e. evaluating count n-gram representations).

3.2.1 Now repeat the training and evaluation process for BOW-tfidf, BOCN-count, BOCN-tfidf, BOW+BOCN including hyperparameter tuning for each model...

3.3 Full Results

Add here your results:

LR	Precision	Recall	F1-Score
BOW-count			
BOW-tfidf			
BOCN-count			
BOCN-tfidf			
BOW+BOCN			

Please discuss why your best performing model is better than the rest.

4 Task 2: Multi-class Logistic Regression

Now you need to train a Multiclass Logistic Regression (MLR) Classifier by extending the Binary model you developed above. You will use the MLR model to perform topic classification on the

AG news dataset consisting of three classes:

- Class 1: World
- Class 2: Sports
- Class 3: Business

You need to follow the same process as in Task 1 for data processing and feature extraction by reusing the functions you wrote.

In [ ]:

Now you need to change `SGD` to support multiclass datasets. First you need to develop a `softmax` function. It takes as input:

- `z`: array of real numbers

and returns:

- `smax`: the softmax of `z`

```
In [ ]: def softmax(z):



            return smax
```

Then modify `predict_proba` and `predict_class` functions for the multiclass case:

```
In [ ]: def predict_proba(X, weights):


            return preds_proba
```

```
In [ ]: def predict_class(X, weights):



            return preds_class
```

Now you need to compute the categorical cross entropy loss (extending the binary loss to support multiple classes).

```
In [ ]: def categorical_loss(X, Y, weights, num_classes=5, alpha=0.00001):
            '''
            X:(len(X),len(vocab))
            Y: array len(Y)
            weights: array len(X)
            '''

            return l
```

Finally you need to modify SGD to support the categorical cross entropy loss:

```
In [ ]: def SGD(X_tr, Y_tr, X_dev=[], Y_dev=[], num_classes=5, lr=0.01, alpha=0.00001,
                epochs=5, tolerance=0.001, print_progress=True):




            return weights, training_loss_history, validation_loss_history
```

Now you are ready to train and evaluate you MLR following the same steps as in Task 1 for the different vector representations

Compute accuracy, precision, recall and F1-scores:

```
In [ ]: preds_te = predict_class(X_te_count, w_count.T)

        print('Accuracy:', accuracy_score(Y_te,preds_te))
        print('Precision:', precision_score(Y_te,preds_te,average='macro'))
        print('Recall:', recall_score(Y_te,preds_te,average='macro'))
        print('F1-Score:', f1_score(Y_te,preds_te,average='macro'))
```

4.0.1 Discuss how did you choose model hyperparameters (e.g. learning rate and regularisation strength)? What is the relation between training epochs and learning rate? How the regularisation strength affects performance?

Explain here...

4.0.2 Now repeat the training and evaluation process for BOW-tfidf, BOCN-count, BOCN-tfidf, BOW+BOCN including hyperparameter tuning...

4.1 Full Results

Add here your results:

LR	Precision	Recall	F1-Score
BOW-count			
BOW-tfidf			
BOCN-count			
BOCN-tfidf			
BOW+BOCN			

Please discuss why your best performing model is better than the rest.