CSE842 HW1

Javen W. Zamojcin
zamojci1@msu.edu

2024, September 19th

1 Naïve bayes classifier

1.1 How to Run: Requirements

The program for this part was implemented using `Python 3.10`. All libraries used and their respective versions may be found in the included `requirements.txt` file.

The movie reviews dataset is accessed through the `nltk.corpus` library, which requires manually downloading the dataset file. This may be done through the Python console:

```
import nltk; nltk.download('movie_reviews')
```

1.2 How to Run: Parameters

There are several parameter options for running the program in this part.

1. You may manually train, specifying the two training folds and saving the resulting model parameters:

```
zamojci1_hw1p1.py train fold[1-10] fold[1-10]
```

2. You may manually test, loading the previously trained model parameters and specifying the one testing fold:

```
zamojci1_hw1p1.py test fold[1-10]
```

3. Or you may allow the program to automatically train and test the model, loading all 10 folds, running 3-fold cross-validation, then outputting the averaged run scores:

```
zamojci1_hw1p1.py all
```

1.3 Design Decisions

Because of the the project requirement to allow CLI parameters specifying the exact folds to train and test on, I built my data architecture around loading the pre-split data fold files based on fold index parameters. This felt awkward to do, and I would have preferred to have split/folded the dataset myself.

For example, `train fold1 fold2` would load and train on the 400 documents `['pos/cv[000-199]_xxxx', 'neg/cv[000-199]_xxxx']`.

I also wasn't sure how manually specifying the training and testing folds would work with k-folds cross-validation and "reporting the average results", so I created the additional "all" run branch which performs the complete k-folds cross-validation across the entire dataset and reports the averaged scores.

As for saving the trained model parameters, I simply pickled the Pandas dataframe containing the computed set of word probabilities given a class (positive or negative), as well as a separate dataframe containing the computed probabilities of the two classes.

For computing the word probabilities, I took advantage of the NLTK dataset procedures; filtering the target words for the given fold indices and class type (categories), creating the frequency distributions, then dividing each word count by the total sum of word count in the filtered corpus. I additionally used Laplacian smoothing as some words would occur in one class and not the other (zeroing the sentence probability chains). The class probabilities were computed as the count of documents filtered by category (positive, negative) divided by the total count of documents in the given fold.

For testing the documents in a given fold, I first load the pickled dataframes containing class and word probabilities computed from the training step. Then for each document, I retrieve its respective list of words, and calculate the sentence probability by index matching each word to the logarithmic word probabilities dataframe and summing the column vector. I make a class prediction for each document by choosing the largest sum of the logarithmic class probability and sentence probability.

1.4 Results

From training and testing using 3-fold cross-validation across the entire set of folds, I received the following averaged evaluation metrics:

Accuracy	0.811111
Recall	0.795556
Precision	0.821628
F1	0.808273

Table 1: P1 Evaluation Metrics

2 NLTK and Sklearn

2.1 How to Run: Requirements

The movie reviews dataset for this part is accessed through local files, which are included under the provided `movie_reviews/` directory.

Additionally, the NLTK library tokenizer may require the following manual download through the Python console:

```
import nltk; nltk.download('punkt_tab')
```

2.2 How to Run: Parameters

All three models may be trained and evaluated through a single script `zamojci1_hw1p2_all.py`, specifying the model type as a single argument:

1. Naive Bayes: `nb`
2. SVM Bag-of-Words: `svm-bow`
3. SVM TF-IDF: `svm-tf`

2.3 Design Decisions

All three models share the same approach of first loading the dataset, performing k-folds cross-validation, vectorizing and transforming the fold training data, fitting the given model, making model predictions for the fold testing data, and evaluating the predictions against the true classes, with the one exception being that the SVM-BoW model does not use the TF-IDF transformer on the training data and only vectorizes the word frequencies.

This approach allows for reusing most of the pipeline code and avoiding code redundancy. To my understanding, the SKLearn CountVectorizer I used in all three pipelines essentially produces a bag-of-words from the provided data, so this is why the only difference between the two SVM models is the use of the TF-IDF transformer, which converts from the bag-of-words raw frequencies.

The initial set of model parameters I settled on (these will be expanded upon in the Extra part) were: `max_features=3000, k_folds=3, min_df=2`. As for the k-folds decision, I found that increasing this value past 3 would indeed slightly improve the metric results at the expense of having a much longer run-time. So for the scope of this project and experimenting with other parameters I settled for k=3. For the max-features decision, the tutorial claimed that the more features the higher your model accuracy would be, however I found that accuracy would actually decrease and be more computationally expensive if I removed the max-features parameter. Using the 3000 most frequent features seems to be adequate. And for the min-df parameter, I found that any value higher or lower than 2 would start to hurt the performance. My understanding of this is that while rare words can provide useful information for predicting class probability, one-offs might be less useful.

2.4 Results

Using the parameters `max_features=3000, k_folds=3, min_df=2`, I received the following averaged evaluation metrics across the three models:

Model	Accuracy	Precision	Recall	F1
NB	0.790999	0.824207	0.741275	0.779955
SVM-BoW	0.713991	0.760305	0.625095	0.685921
SVM-TF	0.826993	0.821694	0.835999	0.828549

Table 2: P2 Model Evaluation Metrics

3 Extra

For the extra work section, I decided to expand upon and experiment with the model parameters, specifically with the CountVectorizer. For all of these experiments, I use the SVM-TF model as a baseline as it clearly had the best previous performance.

Experiment	1	2	3	4	5
Accuracy	0.834997	0.772502	0.829497	0.834997	0.828996
Recall	0.845034	0.766211	0.832053	0.845034	0.855065
Precision	0.828979	0.776547	0.828012	0.828979	0.813075
F1	0.836693	0.771008	0.829932	0.836693	0.833353

Table 3: P3 Experiment Evaluation Metrics

1. The first experiment was the removal of stop words. The documentation for CountVectorizer recommended carefully choosing your method for removing stop words and that there were known issues with their 'English' stop word set. Alternatively, I chose to use the parameter `max_df=0.70`. My understanding of this is that words in the 70th percentile and above in terms of frequency

are removed (i.e. the most common words which do not contribute much information). This significantly improved the evaluation results. I also tried using `max_df=0.85` but this decreased the performance.

2. The second experiment was to adjust the `ngram_range` parameter, changing the lower and upper boundary of the range of n-values for different word n-grams extracted. Setting this parameter to `(2, 2)` means only bigrams will be extracted, which resulted in much worse performance.

3. I adjusted the `ngram_range` parameter again but with the value `(1, 2)`, allowing for both unigrams and bigrams to be extracted. This performed better than the bigrams only, but still worse than unigram only extraction (default).

4. For the 4th experiment I changed the `analyzer` parameter from the default `word` value to `char`, which changed the extracted features to be made of character n-grams instead of word n-grams. Interestingly, this had no affect on performance.

5. For this last experiment I set `max_features=None` and `min_df=1` in addition to `max_df=0.7`, meaning only the 70th percentile and above are removed, and every other word remains. This performed worse than the baseline with the exception of the recall score which did slightly better.