Text Analytics Homework 3
Rachel Rosenberg
November 2019

Link to Github Repo:

https://github.com/rrosenb1/rrosenb1_msia490_2019/tree/homework3

Part 1: Dataset Stats

Dataset description:
This dataset is a large corpus of Amazon reviews for kitchen & home products. It comes from the Computer Science department at UCSD.
It contains 499815 unique reviews, among a subset of only users who have more than 5 reviews on the site overall (to clean the data).
The original data is labeled with a number of stars, 1-5. When cleaning, I assign "good" reviews as those with 3+ stars and 'bad' reviews as those with one or two stars.

There are two classes in the cleaned dataset: good review (1) and bad review (0).
Class distributions below:
Label = 0: (451905, 2)
Label = 1: (47910, 2)
There is some class imbalance; this will be addressed in the models.

There are 93.605 average words in each document in the corpus.

The above is output from code in the dataset_stats.py script.

Part 2: Logistic Regression

Logistic results:
Run locally

Model Name	Min_Df	Ngram_Range	Penalty	Accuracy	ROC
Logistic1	500	(1, 1)	l1	0.804	0.888
Logistic2	500	(1, 2)	l2	0.81	0.893
Logistic3	500	(1, 1)	l1	0.804	0.889
Logistic4	500	(1, 2)	l2	0.812	0.895
Logistic5	200	(1, 1)	l1	0.732	0.834
Logistic6	200	(1, 2)	l2	0.745	0.841
Logistic7	200	(1, 1)	l1	0.732.	0.834
Logistic8	200	(1, 2)	l2	0.746	0.842

For this experiment I varied the number of ngrams (1 or 2), the penalty or norm (L1 or L2) used i n regularization, and the Min_DF for TF-
IDF, which controls the number of features created. I used Accuracy and ROC score for my repo rting metrics - Accuracy because I want to know how the model performs as far as the misclassi fication rate, and ROC because it is a measure of how well the model performs on an unbalance d dataset, and this dataset was fairly unbalanced with more positive (3+ star) reviews than nega tive (1-2 star) reviews.

I took the following preprocessing steps:

* Labelling reviews with 3+ stars "positive" (1) and reviews with 1 or 2 stars "negative" (0).
* Removing all numbers from the text
* Removing all punctuation from the text
* Removing all stopwords (English) from the text
* Stemming all words

I used the following parameters identically for all models:

* Classes balanced. I did not want the model to predict based on the probability of a positive r eview, so I set class_weights to balanced in the Logistic Regression parameters.
* Latin encoding. All
* Max_DF of 0.3 (threshold for ignoring corpus-
specific stop words) - I set this value fairly low because I did not want to ignore too many words . Since this is a very varied dataset, there will be many words that are important to context and also are repeated many times.
* Binary = False (items are counted in the number that they appear). Since there will be some i mportant words that are used very few times, I didn't want them to be overshadowed (or consi dered equal to) words that are less important but used many times.

All models performed decently well, with accuracies ranging from 0.73 to 0.81 and ROC values r anging from 0.83 to 0.89. The best-
performing model on both metrics used L2 regularization, a feature size of 500, and 2 ngrams. This is likely because the model was given more information to work with (both with bigrams an d more features) and L2 regularization was used to cut the model to include only useful features.

Overall, the model could likely be improved upon if I used more features, more n-
grams, and a lower max_df, but all of these things would increase training time too much to be scalable.

Part 3: Support Vector Machine

SVM Results:
Run on Google Colab

Model Name	Min_Df	Ngram_Range	Kernel	Accuracy	ROC
SVM1	500	(1, 1)	lin	0.804	0.881
SVM2	500	(1, 2)	rbf	0.797	0.869
SVM3	500	(1, 1)	lin	0.803	0.885
SVM4	500	(1, 2)	rbf	0.800	0.871
SVM5	200	(1, 1)	lin	0.803	0.877
SVM6	200	(1, 2)	rbf	0.774	0.861
SVM7	200	(1, 1)	lin	0.807	0.878
SVM8	200	(1, 2)	rbf	0.813	0.866

For this experiment I varied:
- the number of ngrams (1 or 2),
- the kernel type used (either linear or RBF)
- the min_df_values parameter for TF-IDF, which controls the number of features created.

I used Accuracy and ROC for my reporting metrics because I want to know how the model perfo rms as far as the misclassification rate as well as on an unbalanced dataset.

I took the following preprocessing steps:
* Labelling reviews with 3+ stars "positive" (1) and reviews with 1 or 2 stars "negative" (0).
* Removing all numbers from the text
* Removing all punctuation from the text
* Removing all stopwords (English) from the text
* Stemming all words

All models performed decently well, with accuracies between 0.77 and 0.81 and ROC values bet ween 0.86 and 0.88. The best-performing model used:
- wordNgrams = 2 (able to use bigrams)
- Kernel type = RBF (based on mean absolute error)
- min_df_values = 200
. This model had an accuracy of 0.813 and an ROC score of 0.866.

This is likely because the model was given more information to work with (both with bigrams an d more features) and the decreased min_df. The RBF kernel type was also better here, though the choice of kernel type didn't make a huge difference either way. The most significant factor was the min_df_values value; since this dataset varies so much and spans so many different products, it is important to be able to create many features from the data.

Overall, the model could likely be improved upon if I used more features, more n-grams, and a lower max_df. I also used some less data to train my SVM models because the large models were getting hung up on both Google Colab and Deepdish; given better compute resources I could have gotten better SVMs.

Part 4: FastText Classifier

FastText Results:

Run locally

Model Name	Learn_Rt	wordNgrams	min_df	Accuracy
FT1	0.3	1	10	0.934
FT2	0.3	2	10	0.940
FT3	0.5	1	10	0.933
FT4	0.5	2	10	0.940
FT5	0.7	1	20	0.934
FT6	0.7	2	20	0.933
FT7	0.9	1	20	0.940
FT8	0.9	2	20	0.934

For this experiment I varied:

- the number of ngrams (1 or 2),
- the learning rate of the model (0.3, 0.5, 0.7, or 0.9), and
- the min_df_values parameter for TF-IDF, which controls the number of features created.

I used Accuracy for my reporting metric because I want to know how the model performs as far as the misclassification rate.

ROC Score and F1 were not key here because the dataset was balanced, so the model would no t be biased toward either class.

I took the following preprocessing steps:

* Labelling reviews with 3+ stars "positive" (1) and reviews with 1 or 2 stars "negative" (0).
* Removing all numbers from the text
* Removing all punctuation from the text
* Removing all stopwords (English) from the text
* Stemming all words

I used the following parameters identically for all models:

* Number of epochs: 25 - this sped up processing some and allowed for rapid iteration, which I valued greatly.
* Used the same dataset and sample size for each model -

All models performed decently well, with accuracies in the 90%s. The best-performing model on both metrics used:

- Bigrams (wordNgrams = 2)
- Learning_Rate = 0.3
- min_df_values = 10

. This model had an accuracy of 0.940.

This is likely because the model was given more information to work with (both with bigrams and more features) and the decreased learning rate.

Learning rate determines how much the model changes after each iteration; a low learning rate is more precise, but is slower. In this case, the low learning rate avoided large swings in learning direction and resulted in a more accurate model.

Overall, the model could likely be improved upon if I used more features, more n-grams, and a lower max_df, but the accuracy is already very high and the amount of increased performance possible requires further experimentation.

Part 5: CNN Classifier

CNN Results:

Run on Google Colab

Model Name	EmbedDim	Vocab_Size	Max_Len	Accuracy	ROC
CNN1	100	1000	50	0.841	0.856
CNN2	100	2000	50	0.852	0.868
CNN3	100	3000	50	0.859	0.880
CNN4	100	4000	50	0.873	0.871
CNN5	200	1000	80	0.893	0.894
CNN6	200	2000	80	0.898	0.871
CNN7	200	3000	80	0.901	0.878
CNN8	200	4000	80	0.912	0.889

For this experiment I varied:

- the dimension of the embedded vectors
- the vocabulary size used to train
- the max_length of the vector used to concatenate the input vector with the embedding vector.

I used Accuracy and ROC for my reporting metrics because I want to know how the model performs as far as the misclassification rate as well as on an unbalanced dataset.

I used a pre-trained embedding layer, trained from word2vec embeddings. For all models I used rectified linear units, L2 normalization, and the same dropout rate and mini-batch size. For all models I also use a binary cross-entropy loss function.

I took the following preprocessing steps:
* Labelling reviews with 3+ stars "positive" (1) and reviews with 1 or 2 stars "negative" (0).
* Removing all numbers from the text
* Removing all punctuation from the text
* Removing all stopwords (English) from the text
* Stemming all words
* Tokenizing each sentence
* Padding sequences so that their lengths would be uniform

All models performed decently well, with accuracies between 0.84 and 0.91 and ROC values bet ween 0.85 and 0.89. The best-performing model used:
- Embedded vector dimension of 200
- Vocabulary size of 4000
- Max_len of 80
. This model had an accuracy of 0.912 and an ROC score of 0.889.

This is likely because the model was given more information to work with in the better-performing models (by increasing dimensionality of embedded word vectors and vocabulary size).

Overall, the model could likely be improved upon if I used more features, more n-grams, and a higher vocabulary set to train on. I also used some less data to train my CNN models because the large models were getting hung up on both Google Colab and Deepdish; given better compute resources I could have gotten better CNNs.

Part 6: “Predict” Script

Script is able to take as input a text file of data to train on and subsequently trains a model and outputs predictions. To do this, it uses pre-saved model and cleaned data files. All preprocessing is done in the dataset_stats.py file. Functions cnn_runner and svm_runner in cnn.py and svm.py, respectively, are used to train the model with the best parameters, and subsequently pass the results back into their respective predict files and out into JSON.