1.

Before talking about the code itself, let’s look at a sample of the data used by the authors to train the models.

This is a positive book review in our database. The data is given to us as a dictionary of word counts and a label. We can see that the authors use bigrams and exclude some stopwords.

Ideally, we want to look for a way to find domain-independent words, words that can be found in all contexts, and that are helpful to split or cluster the data. Words like gorgeous, must-have.

Also, we want to find domain-specific words that also help us split the data, like fan, artwork.

We found that the authors rely heavily on word frequency to make these decisions. Below we can see the top 20 words by frequency in the dataset. Many of these words carry little discriminative power. We don’t see words easily related to sentiment until the end of the list.

2/3

The dataset consists of 2000 labeled reviews, 1000 positive and 1000 negative, per item, as well as about 5000 unlabeled reviews.

The repo didn’t provide a method to process the data, just a link to the data source. The paper describes their feature selection process as: choose the 5000 most common words across all datasets. The fact that we don’t know their tokenization process means we couldn’t exactly replicate their experiments.

We store the data as a bag of words in numpy arrays. This is the format that almost all of the algorithms in their repo follow accept. We save the data as Source Labeled Data, Source Labels, Target Labeled Data, Target Labels, Unlabeled Data. From 4 items, we end up with 12 combinations.

4. We use the hyperparameters provided in the repo. The code here is as provided by the authors with comments by us.

In here they start with 500 domain-independent features and 100 clusters. The common features are chosen by looking at the 500 top words by frequency in source and target, and then reducing this number by looking at the intersection. The compliment of these words are the domain-specific features.

This doesn’t seem like the best way to do things for a couple reasons: As we saw, there are many common words with little discriminating power. We require picking many common words to be able to include enough relevant words.

After this, the algorithm builds a co-occurrence matrix, they build a Laplacian and from here they obtain the top 100 eigenvectors from which to obtain the new features. Again, Since they chose a high number of domain-independent features, they need to choose a high number of clusters to find enough relevant clusters. This could have been done more efficiently. The paper for this model proposes a few ways to do this such as using mutual information to choose the features.

Finally, the eigenvectors are concatenated to the original features, and we have our augmented dataset

Most feature selection models in the repo follow a similar flow: Pick pivot features, use an algorithm to cluster/discriminate (SCL uses correlations between the features, for example), augment the original features.

5.

Finally we run all combinations and obtain accuracies. Their code doesn’t provide much more than accuracies, but we can save the models to do further analysis later.

Here, we see our results compared to theirs. We get results of similar magnitude. Our results underperform some of their combinations, outperform some of theirs. We think that this is because we don’t have the exact same initial data.

Also, we think that feature selection might have had a very important role in determining the scores in their paper. For models like SCL, the choice of pivots is even more influential (they pick pivots as the top 10 words by frequencies). Some of the models have their own way of choosing features in a smart way (like using TF-iDF). So we think that you need to be careful taking their results at face value.