Unsupervised Classification of Articles

Joseph Harter

University of Colorado Boulder

DTSA-5011 Summer 1 2025

June 3, 2025

Repo: https://github.com/joe-harter/dtsa-5510-kaggle

Abstract

In this paper I'll explore multiple ways to classify articles into discrete categories. I'll use non-negative matrix factorization as an unsupervised approach and k-nearest neighbors as a supervised approach. The data is provided from a kaggle challenge[1]

Keywords: non-negative matrix factorization, k-nearest neighbors, NLP

[1] *BBC News Classification*. (n.d.). Kaggle. https://www.kaggle.com/competitions/learn-ai-bbc/data

Unsupervised Classification of Articles

I'll show an unsupervised and supervised way of classifying articles and compare the performance of both using confusion matrices. This paper gives a good "no code" summary of the work I did. But the related code can be found in the repository mentioned above. The files "eda.ipynb" hosts all exploratory analysis. "Model.ipynb" hosts all code for the unsupervised model while the "supervised_model.ipynb" is for the KNN model.

The Data

The datasets came in two files. Both were a list of articles:

Name	Shape	Columns
BBC News Train.csv	(1490, 3)	ArticleId, Text, Category
BBC News Test.csv	(735, 2)	ArticleId, Text

Table 1

As you can see the test dataset is missing a category. The kaggle competition is expecting the competitor to upload a list of categories correlating with those articles and then they will provide feedback on what percent was correct. However, we will never know what the actual intended categories were so I ignored the test set for this specific project as I needed to know the category in order to know how accurate my models were. So our full dataset will just be 1490 articles with 2 different columns because I didn't make use of the ArticleId.

The distribution of categories for all of the articles can be seen in table 2. There are no null values in the dataset, and the text appeared to be properly cleaned up. That is there were no

accidentally joined words as determined by looking for hapexes[2]. You can also see the distributions of word counts in figures 1 and 2.

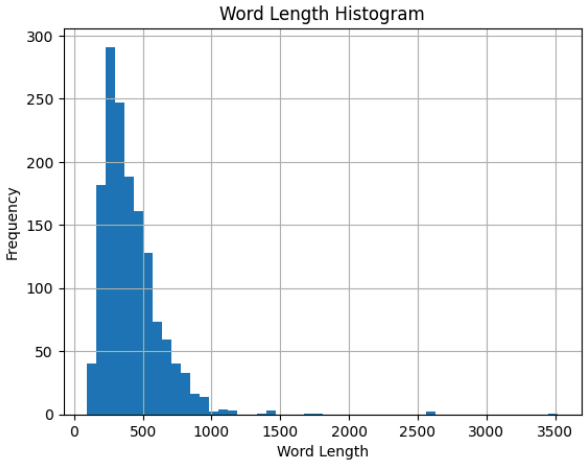

Figure 1

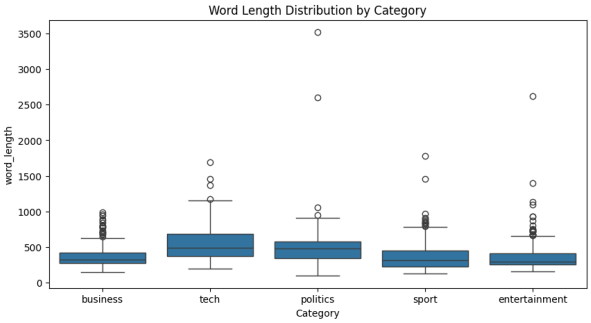

Figure 2

A Note on Preprocessing

This dataset was in really good shape. I didn't see any evidence of HTML or accidentally joined words as mentioned above. So I only removed stop words[3] though it might not have been necessary as it seems like the chosen Tfidf class already handled it but just for consistency I

[2] Cynic, A. (2017, September 7). *A First Exercise in Natural Language Processing with Python: Counting Hapaxes*. The Cat's Whisker. https://catswhisker.xyz/log/2017/9/7/a_first_excercise_in_natural_language_processing_with_python_counting_hapaxes/

[3] GeeksforGeeks. (2024, January 3). *Removing stop words with NLTK in Python*. GeeksforGeeks. https://www.geeksforgeeks.org/removing-stop-words-nltk-python/

applied it myself. In my attempt to improve the model I also apply lemmatization[4] but I'll go into the details of that later on.

Unsupervised Model

The first model to solve this problem was an unsupervised model using non-negative matrix factorization.[5] First I split the "training" set into a train and test set with 20% of the articles landing in the test set. I then used TfidfVectorizer which returns a statistical value for each word (feature actually) in each article. For my initial test I used the following hyperparameters for the vectorizer:

- `max_df` - .95
- `min_df` - 2
- `max_features` - 1000

I then used the NMF class from sklearn to fit this vector and generate and classify each article with 1 of 5 "categories". These categories are unlabeled of course, so I borrowed the `label_permute_compare` method from an earlier lab to find the most likely mapping from unsupervised category to actual category. This resulted in an 89% accuracy score on the split training set.

The test set had the same accuracy of 89% so that shows we weren't overfitting on the training data. See the confusion matrix in figure 3. We perfectly predicted the sports articles and did fairly well on all of the others. But let's look at tuning our hyperparameters to see if that's something that can improve our accuracy.

---

[4] GeeksforGeeks. (2024a, January 2). *Python | Lemmatization with NLTK*. GeeksforGeeks. https://www.geeksforgeeks.org/python-lemmatization-with-nltk/

[5] GeeksforGeeks. (2025, May 20). *NonNegative matrix factorization*. GeeksforGeeks. https://www.geeksforgeeks.org/non-negative-matrix-factorization/

I looped through a range of values for all 3 hyperparameters (see “Optimum Hyperparameters” in the GitHub Repo) and discovered that the best combination for this set of data was to to have a max_df of .8, a min_df of 4 and a max_features of 1500. This resulted in a 92.6% accuracy on the test set. See “tfidf_results.csv” for a full accounting of results.

Further Optimization

I then sought additional ways to improve the performance. There were two options that came to mind. I could lemmatize the words, which would then do a better job of finding just how unique each document really was. I could also convert my words to bigrams and vectorize those. This would show us how often certain combinations of words are together in the documents rather than just the individual words.

Non-Optimized

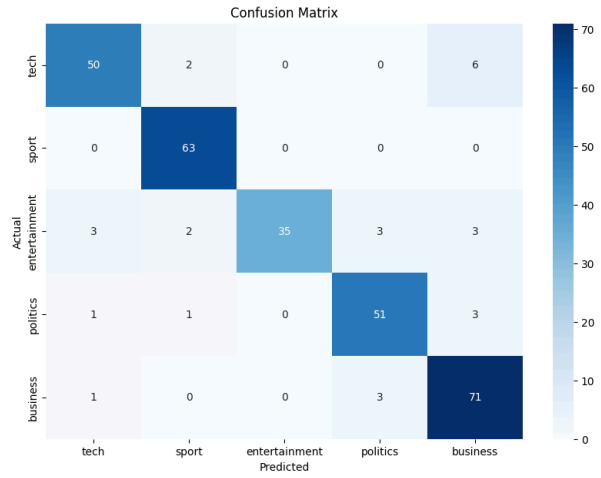

Figure 3

Optimized

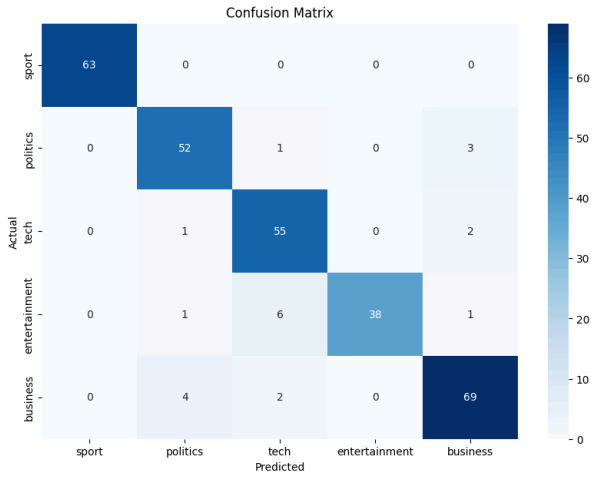

Figure 4

Unsupervised Conclusion

Unfortunately lemmatization and using bigrams did not result in a more accurate model. Bigrams actually led to a very inaccurate model. When trying to hyperparameter tune I only saw at best an accuracy of 71%. The final confusion matrix (figure 4) shows a general improvement.

Supervised Model

To compare to a supervised model I thought a KNN solution would be appropriate with the number of nearest neighbors set to 1. I used the hyperparameters that we determined to be optimal from the unsupervised model for consistency sake. This resulted in a 95% accuracy. See figure 5 for a confusion matrix.

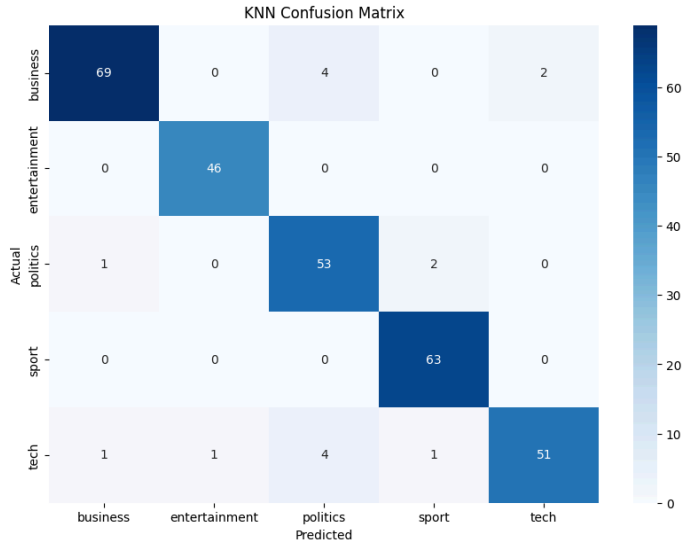

Figure 5

Results

It's impressive what matrix factorization can do when we don't have labels. It wasn't as accurate as our KNN model, but it turned out to be useful and if we're in a situation where we don't have labels then it has increased my confidence in such a method. I will note that I'm slightly uncomfortable permuting the different categories to guess at which labels should apply to an unsupervised model. Our model may not categorize the data by category a human concept of "category" and could result in different totals and would therefore not be comparable to a supervised method. This specific example seemed to work, however.