Categorizing News Reports Based on Description and Title

OVERVIEW

I wanted to create a model that could use a short description or title of a news article to categorize it. Having already made a model in class to categorize articles based on their contents, I wanted to see if it was possible to do it with a much smaller amount of words per entry. The total dictionary would likely be just as large, so one of the major objectives was to figure out what parameters would be needed to and how they would differ from the previous categorization models I had made.

The original plan was to create two separate bag of words dictionaries: one for the titles, and one for the descriptions. After doing so, I couldn't find any way available to peer inside the dictionary as they were combined into a 2x2 array that had a compressed dictionary of some kind. After searching the internet and not being able to find a way to see what was going on inside it, I decided to try a different method since I couldn't verify what was going on inside the data object and as such my conclusions would be in doubt. A simple way to fix this was to just combine the words from the title and the dictionary into a single string then create a bag of words based on those. In retrospect, I probably should have just done that to start off, since a description and a title are both likely to contain many of the same key words.

I ended up having to limit the size of the data set. The original file with all the data was around 100k entries and my computer just couldn't deal with that. Jupyter notebooks would freeze up and never come back or give me dead kernel errors, so I reduced the size of the project to just the first 20k entries in the data file in order to be able to run it multiple times in grid search. Another limit came in the form of parameters I couldn't run, but I will discuss that in the section named PARAMETER CHOICE.

The highest accuracy I was able to obtain was a 0.671 and I think that's pretty good for the limited data points. I believe it could have gone higher - perhaps even into the 80's - if I was able to run the full 100k entries.

THE DATA

News Category Dataset

[authors, category, date, headline, article_link, short_description]

short_description: She left her husband. He killed their children. Just another day in America.

headline: There Were 2 Mass Shootings In Texas Last Week, But Only 1 On TV

date:2018-05-26

link: https://www.huffingtonpost.com/entry/texas-amanda-painter-mass-shooting_us_5b081ab4e4b0802d69caad89

authors: Melissa Jeltsen

category: CRIME

Above is the name of the dataset I used, a list of the attributes recorded in the dataset, and an example of what one of the elements in that dataset contains. It was posted by Rishabh Misra on Kaggle.com under the title *News Category Dataset*. The data can be found by following this link HERE.

MODEL AND PARAMETER CHOICE

```
pipe = Pipeline([
('vect', TfidfVectorizer(min_df=10, max_df=0.1)),
('svc', LinearSVC(C=1))])

parameters = {
    'vect__min_df': [0, 10],
    'vect__max_df': [0.4, 0.3, 0.2],
    'svc__C': [2, 1, .1]
}

grid_search = GridSearchCV(pipe, parameters, cv=5, verbose=0)
```

Grid Search done with above parameters variable and 5-fold cross-validation.

I decided to use the LinearSVC instead of the SVC with Kernel = Linear because LinearSVC is better optimized for large amounts of data and would run faster than the SVC with Kernel = Linear. Since I needed all the help I could get to run a multitude of models of varying parameters I went with that.

Figuring out the range I wanted for the hyper-parameters took most of the actual work for this project. I knew I would have to severely limit the max_df since the dictionary was gigantic but each article only had 2 sentences (around 15-30 words give or take a few) associated with it. If I made max_df too high, nothing would be cut out, since it was likely even the most popular words for a class weren't appearing in any more than half of the elements of that class. This would result in a lot of trash words like "the", "in", and "a" being taken into consideration by the model.

The reverse was true of min_df. I didn't want to set it too high because likely there were some words important to the classification of each class that would barely appear, and I didn't want to set the limit high enough to cut those out. It turns out from testing it though that not cutting out any words at the lower end of the scale was the best way to do it (i.e. setting min_df = 0).

As for C, I went with 2, 1, and 0.1 to see if a better C would higher or lower than 1 by a large amount, but it came back as 1 being the selection for the model which is convenient.

When I first got to where I needed to choose the hyper-parameters, I ran the model manually over and over to learn what each one of the hyper-parameters did and test my theories about its effect on the accuracy score. Once I knew what the sweet spot was for them, I tested a few values in a small range to see exactly which ones were going to work the best.

I originally planned to test ngram_range with values of (1, 1) and (1, 2), but when I did so the model would never resolve so I eventually had to discard it to get the model to run on my machine.

MODEL RESULTS

pprint(grid_search.cv_results_)

Rank	Mean Accuracy	St. Dev.	Parameters		
			svc_C	**vect_max_df**	**vect_min_df**
3	0.65064	0.00769	2	0.4	0
16	0.60924	0.00688	2	0.4	10
5	0.64964	0.00887	2	0.3	0
17	0.60797	0.00693	2	0.3	10
6	0.64058	0.00654	2	0.2	0
18	0.59844	0.00799	2	0.2	10
2	0.65551	0.00697	1	0.4	0
7	0.62664	0.00764	1	0.4	10
1	0.65584	0.00751	1	0.3	0
8	0.62651	0.00754	1	0.3	10
4	0.65004	0.00516	1	0.2	0
13	0.61644	0.00694	1	0.2	10
11	0.61737	0.00941	0.1	0.4	0
10	0.61977	0.00648	0.1	0.4	10
11	0.61737	0.00901	0.1	0.3	0
9	0.62064	0.00668	0.1	0.3	10
15	0.61164	0.00797	0.1	0.2	0
14	0.61391	0.00691	0.1	0.2	10

print(grid_search.best_params_)

```
{'svc__C': 1, 'vect__max_df': 0.3, 'vect__min_df': 0}
```

print(accuracy_score(y_test, y_predicted))

```
0.671
```

It performed better than I assumed it was going to. There were 27 categories to decide on and it attained around a 67% success rate. That would be a significant rate even if it only had to choose between 2 categories (significant in that its better than random chance) and it managed to do it with 27 of them. Even though it wasn't good enough to make it very useful in real life but facing the limitations I had to shackle the model with just to get it to run without breaking down I feel like it did an impressive job. If I had been able to use all 100k data points I think it would have been a much better success rate. I'd also have liked to test different ngram_values to see if that could increase the rate of success but as stated earlier it was just too much work for my computer to be able to deal with in a decent amount of time if at all. Overall, I'm very happy with how it worked out and feel that I was able to do about as much as I could without putting my computer into a data-processing-coma.

Additional calculations and problem solving appear in the notebook pdf after the conclusion of the running of the model. I left it there for completeness sake and there are annotations all over the notebook to explain what I was doing and thinking for certain parts and to show the results of various steps in the creation of the model.

Lastly, thank you for teaching this class I enjoyed it and thanks TA's for the work you do behind the scenes. I hope yall enjoy the holiday break.