Project Design

Topic Selection

For this project, I aimed to find a topic that was both interesting and actionable – especially in a business context. After exploring various topics, I eventually stumbled upon a dataset of NYT articles and comments that were left on these articles. This data was quite easy to use as it was well organized, and I began to explore possible questions I could answer. In the end I sought to categorize articles, based on their keywords, as being high-comment or low-comment. Ultimately I'd predict off of these keywords whether their corresponding article would be high or low comment.

A project of this nature would be quite easy to sell to newspapers: they can determine even before writing an article whether it would highly engage readers and encourage comments. Further, I hypothesized that an increase in comments would increase reader retention, as they would view the paper as not just something to read but more as a forum to engage with others.

Unlike my last project, which centered a lot around data cleaning and manipulation, I didn't need to do much with my data: I created a count for each article of how many comments it had received, and then created a target column with a binary classifier of high or low volume. After some EDA I decided a good threshold would be 100 comments as for most articles, if they started taking off with comments, they'd receive quite a high number. In the end this gave me a near 50/50 split, so it came with the added bonus of avoiding an imbalanced dataset.

Modeling

To manipulate my data into a format usable for modeling I needed to utilize Natural Language Processing (NLP). I began by preprocessing my keywords to remove stop words, but found that was unnecessary for my dataset as the keywords tend not to use filler words. So, I instead just tokenized them into individual words, and began modeling by using CountVectorizer.

After I had made vectors for each of my documents, I did a 70/30 train/test split for the articles and fed these vectors into Naïve Bayes. These vectors are quite sparse: with over 4,000 unique words and only ~6 keywords per article, there would be 6 '1's and 3,994 '0's for each document, and Naïve Bayes is the best classifier for this situation. Still, I suffered from the Curse of Dimensionality, and found that my data grossly overfit: the train score was around 89%, and the test score was around 70%.

As such, I began to use dimensionality reduction to make my data easier to feed into models. Here I tried two methods, LSA and LDA. LSA simply applies Singular Value Decomposition onto the vectors to reduce it from its original 4,000+ dimensions to a number of dimensions that I can choose. I played around with getting 5, 10, 15 and 20 dimensions, and fed these into a Naïve Bayes classifier to see which performed the best. They were quite close in outcome, but I settled upon 10 as my best case. The train/test split for these was consistent (there was no large gap), and was in the early/mid 70s.

I also tried LDA, which is a bit more complicated in nature but ultimately performs the same function, and found that it was not as useful for my dataset. Feeding these into a Naïve Bayes classifier gave me in the 50-60% accuracy, so I decided to stick to LSA.

At this juncture I also introduced a different group of vectors to try feeding into my models in the form of TFIDF. Instead of a word count like CountVectorizer provides, these provide a weighted number describing how often a word is used in each article relative to how often it's used across all of the articles. I tried feeding TFIDF straight into a Naïve Bayes, then tried feeding it into LSA and LDA and from there into Naïve Bayes, but each time found that it was a significantly worse predictor (in the 50-60% range).

After settling on 10 vectors from LSA in CountVectorizer, I ran different models to determine which produced the most accurate score when classifying these articles. While most of them had a very similar score, logistic regression and SVM performed the best at around a 77% score for both the train and the test. From there, I had to think about which metric is the most important for my product.

If news companies are trying to determine whether an article will receive a lot of comments, they'd care most about making sure that if my algorithm predicts that it will, that the article *actually* will get a lot of comments. As such, I cared most about precision/reducing the false positive rate. I graphed the ROC for my models and looked at their confusion matrixes at different thresholds, and found SVM was the most precise at the relevant thresholds. This gave me a very accurate classifier that I believe would sell well: near 80% of the time I say an article will be high-comment it will indeed be high-comment.

Unfortunately, dimensionality reduction comes at the cost of interpretability, so it was quite hard to see what words were highly correlated with being high-volume. One way to do this would be to put single words or word combinations into my predictor and see whether it predicts it to be high or low comment, but I decided to try my hand at unsupervised learning to determine what topics are high and low comment.

To this end, I ran LSA and LDA on both CountVectorizer and TFIDF, and looked at what topics were produced. LDA on Countvectorizer produced the cleanest separated topics; in the low comment articles there were often words centered around Culture (TV, crosswords, theater, etc.), Real Estate (housing, real-estate, location, etc.), and certain miscellaneous topics (women/girls/metoo all were common, education, rights, crimes). The high comment articles all produced the same topics each time: politics. It came in many forms: international relations (especially Russia and Syria), trade, department of defense, trump, and so forth.

I also did a quick foray into sentiment analysis to see if I could determine which articles had the happiest and angriest/saddest responses, but there was too much noise to accurately classify the comments in relation to their articles. Peoples responses might be sad because the article was sad or because they disliked the content or how the author wrote, for example. Ultimately, though, my supervised and unsupervised models provided clear insight into which articles were getting the most buzz in the NYT. These NLP projects seemed quite actionable, and I was satisfied with my results.

Tools

Python
Pandas
Scikitlearn
GridSearchCV
Gensim

CountVectorizer

Algorithms
Logistic Regression
SVM
Gradient Boosting
Random Forest
LSA
LDA

What I'd do differently next time

Going into this project I was uncertain what the pipeline would look like for NLP in supervised and unsupervised projects, so I spent a lot of time playing around with the tools trying to figure out the correct way to proceed. Ultimately I realized there was no 'correct' way, and that I needed to simply try each combination to see what worked. As a result I spent too much time paralyzed trying to figure out what was right instead of spending my time more fruitfully just trying out different things.

I also would have liked to done a word2vec vector for the high comment and low comment articles (1 each so 2 in total, the average of all the articles in the category) and then used cosine similarity to make a prediction for whether the article would be high or low comment. Unfortunately, I ran out of time before I could do this.