Popular Truth

Michael Sun

May 2019

1 Introduction

In this project I investigate different approaches to predicting the popularity of a news article, given article metadata and content. Apart from applying supervised learning on it, I am interested in which features, either those prepared by the dataset creator or additional ones I engineer, carry the most predictive power.

In a 2015 CS229 project, He Ren and Quan Yang used a random forest to achieve 69% accuracy in classifying articles as either popular (> 1400) shares or not. I aim to outshine the baseline set by my predecessors by engineering new features on the extracted article content itself, combining what I've learned in CS229 and previous NLP techniques I'm familiar with.

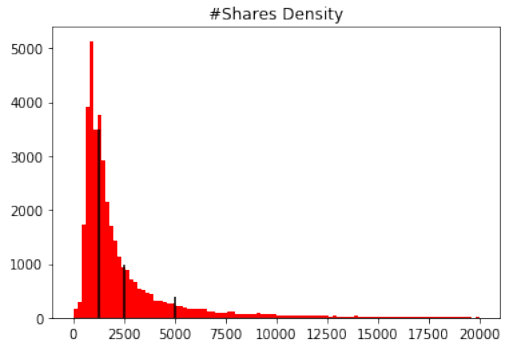
2 Collecting News

I obtained the dataset of article urls and metadata from UCI's Online News Popularity Dataset. To create a robust classifier, I used beautifulsoup to scrape all the articles of its author and content.

3 Taking in the News

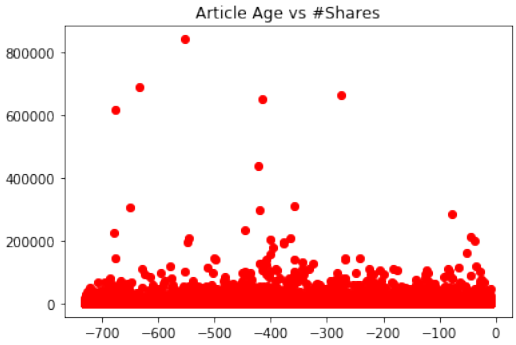
3.1 Shares distribution

The overall shares distribution is a skewed Gaussian or a power law curve. There wouldn't be much point in a classification task if the distribution was just Gaussian noise, but like other forms of media news articles produce network effects that spikes no. shares. Admittedly, this power law may be more pronounced in the more sensationalist websites of today, but nonetheless this motivates the distinction between an article that takes off or doesn't (hence classification).



3.2 Time vs Shares

At first thought, it doesn't seem like we can take the share number verbatim, as we may worry a) the article's longevity and b) the overall increase of visitors to the news site may skew the observations.It turns out after looking at the data that the average number of shares, taken in chronological batches, was relatively constant. For our purposes, we'll assume a) and b) either cancel out or aren't too big of a hindrance, treat the age of the article as a feature, and proceed without normalizing the number of shares.



4 Naive Bayes Classification

To get a feel how useful vocabularies will be for my task, I first tokenized all sentences of all articles, and removed stopwords. For experimentation purposes, I only take the first 1000 articles.

After constructing a good vocabulary set, I built a frequency matrix for the articles. At first, I thought only [nouns, proper nouns] is enough (i.e. seeing "Kardashian" $\Rightarrow$ high shares), but after getting only 0.60 accuracy, decided to systematically optimize a good part-of-speech filter.

Starting with an empty POS feature set, I applied forward search, picking the POS in combination with the existing feature set produces the highest delta score, and append it to the feature set until no feature improves the score. The result is [plural nouns, past-participle verb, present-participle verb], with a score of 0.69 (on only 1000 articles!), which is a good sign.

(P.S.) The most predictive words for Naive Bayes are, in order, ['shootings', 'subscribers', 'subjects', 'prosecutors', 'clues'].

5 Kernalized Linear Regression

Using the Naive Bayes log-probability for an article, I can feed it as a (hopefully useful) feature in linear regression.

I applied the linear and sigmoid kernels on support vector regression, quickly obtaining an accuracy of 0.65 on a small sample, which easily puts shame to the reported 0.52 accuracy of my predecessors using the same method, so the new feature does indeed make a significant difference.

After doing this, I realized the most telling of vocabularies aren't words, but bigrams (Donald Trump) or n-grams (United States of America). At that point I thought I needed to expand the feature set (even if it may become very sparse) and find ways to reduce dimension later.

6 Working with the Article Itself

6.1 Title

We begin with the most promising predictor - the title (duh)! Word2Vec is a form of unsupervised learning that maps a set of words to continuous vectors in a way words that appeared in similar contexts (sentences, documents, etc.) have similar features have higher dot products. By taking an average of the word vectors in a title, we can approximate a "title" vector. The resulting title vector is appended as fifty new numerical features.

(Later, I wondered if the TF-IDF matrix itself of the title can be useful, so I applied a similar approach as working with the content (see below); I added in bigrams and trigrams to form a vocabulary to get an incredibly sparse TF-IDF matrix, but after doing LSA this actually worsened the accuracy, so I decided to leave this part out. You can see this tangent in the code).

6.2 Keywords

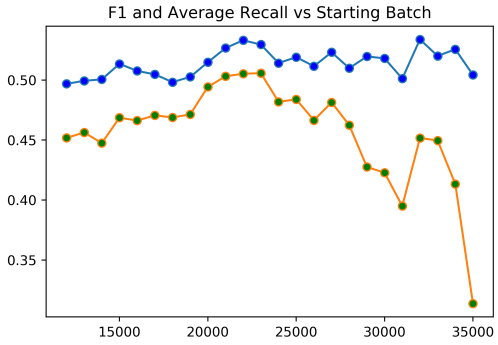
Although it would've made sense to apply the same approach as the title, I also recognize that many articles have "filler" keywords, and I want to make

sure common ones like "world" or "society" are weighted less than words like "crime", so similar to the title, I applied a TF-IDF transformation.

6.3 Content

Again, we form a vocabulary of all n-grams up to hex-grams (we can push further, but six consecutive words seem like a reasonable cutoff for phrases) so we can apply tfidf. But wait. That's a lot of features!

It turns out, past a certain point in the past, the content isn't even relevant anymore. To find when increasing the vocabulary set becomes counterproductive, I plot classification metrics against the starting batch from which content is used; this peak turns out to be around 22000.



Next, to reduce the dimension, we have two approaches - a) use Lasso Regression with a strong regularization constant to drive trivial feature coefficients (words/n-grams) to zero, or b) use LSA to capture the variance of the tf-idf's of the existing vocabulary set. There seems to be a trade-off (bias-variance) between dimension reduction via the two approaches, so we experiment to find the what combination of each.

(I chose Lasso because it works well with large, sparse features, and TruncatedSVD because of its effectiveness with tf-idf matrices for documents.)

After a param search over C (reg constant), alpha (cutoff threshold for coef elimination), and reduced dimension d, I obtain values of $\{C, \alpha, d\} = \{6.5, 0.4, 1500\}$, where C produced the highest F1 score via Lasso regression on the training data, α reduced dimension by a factor of 5, and d captured $> 70\%$ of the variance.

With 1610 numerical features ready (61 of them being the original metadata), it's time to put it all together.

7 Results

Method	Eval Acc.	Eval Prec.	Test Acc.	Test Prec.
(1) Lasso Regression	0.67	0.63	?	?
(2) Support Vector Classifier	0.65	0.65	?	?
(3) Random Forests	0.68	0.64	?	?
(4) Neural Networks	0.66	0.61	?	?

I have yet to use the test set, as I want to tune this on the validation set a bit further after the course ends.

1. This set a highly effective baseline. Somewhat surprisingly, this outperformed Ridge Regression, implying more features could have been eliminated.

2. This had a recall and specificity of 0.65 too; it's seems to have "converged" to a very stable point. Getting a high precision is especially hard with articles, and it seems finding a separation boundary outperforms all other models.

3. This was the previous project's best result. I had issues surpassing it as my engineered features don't fit well with random forests.

4. Parameters were tuned from the heuristic that the max features considered for each split decreases precision, and min samples needed for a split should decrease as variance is lowered with more trees. As it turns out, random forests aren't very effective in dealing with all the engineered continuous features.

8 Conclusion

The goal of this project was to predict the binary popularity label of an article from its metadata and content. I engineered new features by extracting the word2vec of the title, the TF-IDF-weighted vectors of the keywords, and the LSA-reduced TF-IDF counts obtained after a Lasso feature selection on the vocabulary set for the content. With common sense and experimentation, my engineered features are (in a weak sense) optimal considering I didn't resort to any more powerful methods. Each approach I used cleanly outperforms the metadata-only counterpart of the project that inspired mine, with the exception of random forests, which as discussed above doesn't work well with my engineered features. It seems likely 0.69 is a fundamental limit to this task's performance, as a) the shares distribution is continuous and a lot of articles fall close to 1400 shares, and b) there is an inherent degree of noise and unpredictability in article readership (i.e. what happens in the first hour).

9 Takeaways / Future Work

Nonetheless, I got to experience the uncertainty and excitement involved in applying machine learning to a relevant problem in industry. Text is incredibly rich in information, but pruning out the noise required me to think critically and resourcefully, finding the most relevant features I need (Table 2) and reducing features/dimensions when necessary (Lasso and LSA). I also got deeper domain insights, like the relevance of keywords, importance of titles, and usefulness of word vectors from the content. Finally, I learned how to combine a heterogeneous set of features together, accounting for variable interactions/dependencies, and the bias-variance trade-offs between different models (i.e. regression with no feature engineering vs random forests on "split"able features).

Ultimately, the goal is to dig deeper into the article extract the best predictors of popularity, but first I am curious how much predictive power is in the url and first sentence of the article. Afterwards, I want to investigate a more recurrent approach by ordering the features (i.e. url =¿ keywords =¿ title =¿ first sentence =¿ first paragraph =¿ etc.) and having the algorithm halt once it has "enough information" (this is similar to how humans read articles). Via rewards, the agent learns to prioritize which features are more important and which to look at first.