CS598 - Project 3: Movie Review Sentiment Analysis

Author: Gunther Correia Bacellar, NetID: gunther6 Date: 11/25/2020

Introduction

In this project, I analyzed different ML models to predict the sentiment (either positive or negative) of movie reviews using a vocabulary size of up to 1,000 terms. I used an IMDB movie review dataset with 50,000 observations. In order to build a unique vocabulary, I worked with training dataset of 25,000 observations from split 1, and used the same vocabulary to train and test models and calculate the AUC for 5 different splits. The objective is to find the best model able to predict sentiment from a 25,000 training dataset with AUC superior to 0.96 for each of 5 training/test datasets folds using the same vocabulary, limited to up to 1,000 terms.

All processing time reported below was based on a Surface Book 3 machine with Intel Quad-Core 10th Gen i7-1065G7 CPU @ 1.50GHz, 32GB running Windows 10 64-bit and it is related to train and test

Preprocessing

Step 0: This step of creating the vocabulary is explored in the HTML report, but I want to share the AUC impact of some transformations I analyzed before finding the best vocabulary. I used Lasso to pre-selected the vocabulary (up to 1,000 words) and Ridge logistic regression with standard parameters with different transformations to create a benchmark and understand the impact of each transformation below (AUC values based on CV using the 5 folds):

The improvement from using single words to N-gram terms for the vocabulary is significative. 4-grams had the best AUC value, followed by 2-grams. I decided to adopt the 2-grams instead of 4-grams vocabulary for three reasons: 1-) a model using 2-grams is easier to explain than a model with 4-grams, 2-) the AUC difference between 2-grams and 4-grams is minimal (0.00002) and 3-) the time to process 5 folds (train/test) is less than half for a 2-grams vocabulary.

Number of grams per term	CV AUC	Processing Time
5	0.96291	1.9 min
4	0.96292	1.5 min
3	0.96283	1.0 min
2	0.96289	48.1 s
1	0.95351	32.3 s

Table 1: CV AUC of different transformations for the vocabulary

After deciding by 2-grams vocabulary, I also tested the CV AUC impact of other transformations:

- Adding Stemming: AUC of 0.90209
- Stop words: Removing stop words using tidytext package: AUC of 0.95104

These two transformations didn't improve the CV AUC value, so I decided not to implement them. With that, I defined my AUC benchmark of 0.96289 using Ridge logistic regression before any tuning.

Step 1: With the vocabulary created and defined, I started my analysis reading the vocabulary from myvocab.txt with 1000 terms, then I loaded the training and test dataset to test different models. The same vocabulary were used to test all 5 folds.

Step 2: In order to preprocessing and prepare the train/test datasets, I pre-selected some algorithms to analyze against my benchmark: Ridge logistic regression, Naïve Bayes, XGBoost, Random Forest, SVM and LDA.

Step 3: Before modeling the training data, I lowercased the reviews, tokenized them and created a DTM matrix using 2-grams. DTM matrix was used for the Ridge Regression, Naïve Bayes, Random Forest, LDA and SVM models. In addition, I created a xgb.DMatrix for the XGBoost model.

Step 4: In my last preprocessing tentative, I tested the effect of tf-idf transformation using fit_transform (text2vec). This step decreased my CV AUC from 0.96289 (benchmark) to 0.96022, so I decided not to adopt it.

Models pre-analyzed

I first analyzed 8 models using standard parameters to decide which ones I would focus on the tuning process. The table 1 show the CV AUC (5-fold) of all models initially considered.

As expected, Naïve Bayes and LDA models didn't performed well as these models assume some independence between terms, what is not the case for the review texts.

From these 8 models, I decided to concentrate the tuning in the 4 models (in yellow) with highest AUC (in yellow)

Model	CV AUC	Processing Time
Ridge Logistic Regression	0.96289	3.28 s
XGBoost (gblinear)	0.96276	3.89 s
XGBoost (gbtree)	0.95464	19.7 min
Random Forest	0.95378	4.4 min
SVM (linear kernel)	0.90319	1.13 h
SVM (radial kernel)	0.90072	1.28 h
LDA	0.89881	8.8 min
Naives Bayes	0.81051	52.6 s

Table 2: CV AUC of all models tested before tuning parameters

Model Validation: Ridge logistic regression

With a good vocabulary of 985 terms in hands, I used cv.glmnet to find the lambda.min and tune the Ridge model, what increased the CV AUC increased from 0.96289 (with lambda = 0) to 0.96672 (with lambda.min). The reason I didn't use lambda.min in the beginning of my preprocessing tests is the big difference between the processing time with and without calculate lambda.min, from 3.3s to 1.8 min. When I had to run hundreds of simulations, the time saved helped to better analyze my preprocessing.

I also used grid search to find manually a single value of lambda for all 5 folds. I was able to find a single value of lambda (0.07) for all 5 folds that produced a AUC value of 0.96677, slightly better than applying cv.glmnet (0.96672), with the advantage of reducing the processing time from 1.8 min (cv.glmnet) back to around 2s. So, my new best CV AUC for the 5 folds using Ridge regression was 0.96677 for vocabulary using 25,000 observations from train dataset (fold 1). Also the AUC of all 5 folds were also larger than 0.96.

To help reduce the processing time to run CV for Ridge regression, I used the R package doParallel for parallel processing (only in the tunning stage), helping to reduce the processing time in around 30%.

Model Validation: XGBoost

I also decided to analyze XGBoost. Using all standard values and a minimal parametrization, I started with a CV AUC (all 5 folds) benchmark of 0.95464 for booster gbtree. With the experience acquired in project 1, I created a search grid of 7 XGBoost parameters: nrounds, eta, max.depth, colsample_bytree, gamma, subsample and min_child_weight. Instead of only do two wave of search grid per parameter as in project 1, I did many, starting with a sparse range to one less sparse until find some type of convergence. Each wave of search grid used 8-10 values per parameter of each of 7 parameters. As tree models using XGBoost take longer times, I let the computer running overnight for each wave of search grid (6h per wave). After some waves of search grid I found a model with eta = 0.045, nrounds = 4400, colsample_bytree = 0.008, subsample = 0.8, max.depth = 9, gamma = 0.355 and min_child_weight = 0.466 that generated a CV AUC of 0.96658, very close to Ridge regession. Also, running gbtree booster with the tuned parameters reduced the processing time a lot for the 5 folds from 19.7 min to 1.9 min.

After finding the best XGBoost with gbtree booster, I tried XGboost with gblinear booster. This algorithm was much faster than gbtree , with only 2 main parameters to tune (nrounds and eta). To allow reproducibility, I used nthread = 1 and set.seed(1234). The total search grid took me less than 30 min to find the tuned parameters nrounds=16 and eta=0.0788, producing a CV AUC of 0.96717, better than gbtree booster and Ridge regression.

Model Validation: Random Forest

For RandomForest I tried three different R libraries: randomforest, ranger and xgboost. XGBoost has an experimental parameter "num_parallel_tree", that when used with colsample_bytree < 1, subsample < 1, eta = 1 and round = 1 allows to run pure RandomForest models through XGBoost. The three libraries returned approximately the same values, but RandomForest through XGBoost was 40% faster than the other two libraries.

Using the same approach for search grid, I found the parameters subsample= 0.844, colsample_bytree= 0.0078 and num_parallel_tree= 45, with a CV AUC of 0.96683, better than Ridge regession and XGBoost gbtree, but slightly lower than XGboost gblinear.

Interpretability

Inspired by the “Visualizing ML Models with LIME” article, I tried to measure the impact of all 985 terms in the vocabulary to understand how significant they are to contribute to the model and predict accurately a sentiment, as well as to explain better the model. The approach I used was to exclude one term per time and measure the AUC impact (difference in the AUC before and after excluding a single term) for each of 985 terms, and use a scale of 1e-6 for the delta AUC. This allowed me not only to rank all vocabulary terms in the model, but also measure them.

With this approach, see below the terms that better explain and less explain the model.

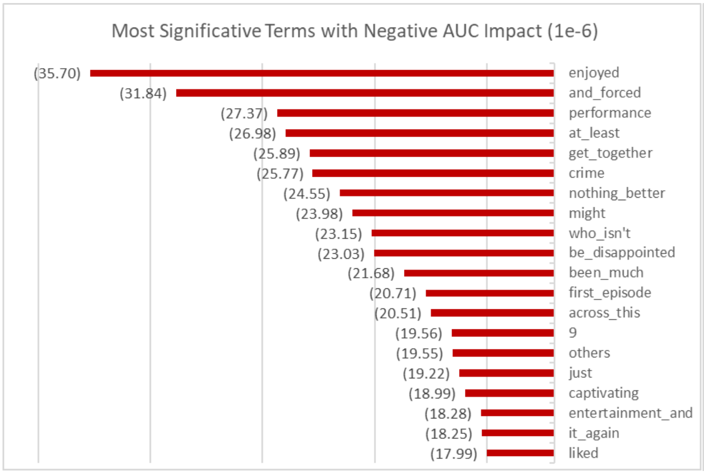


Figure 1: The 20 out of 985 terms that less contribute to accuracy of the model

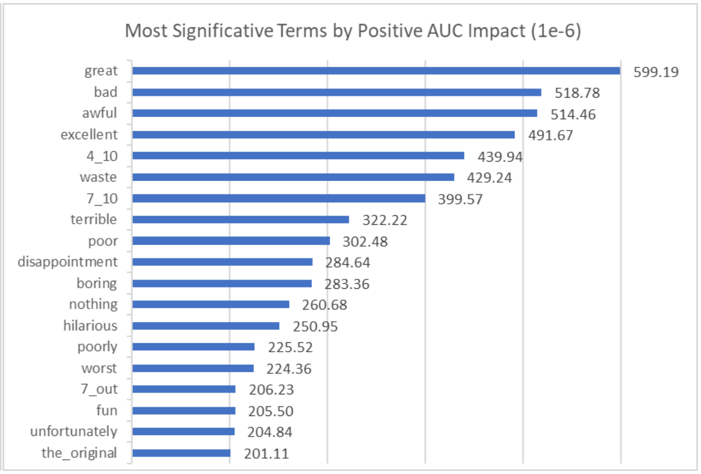


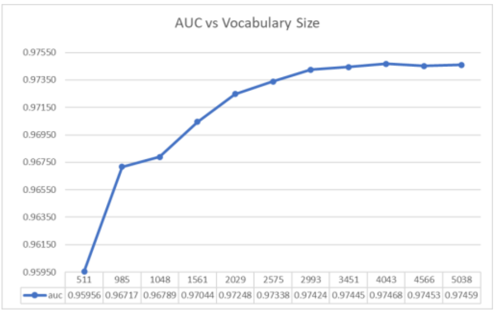
Figure 2: The 20 out of 985 terms that most contribute to accuracy of model

This means that terms such as “great”, “bad”, “awful”, “excellent” provide an accurate classification by the model, while terms such as “enjoyed”, “and_enforced” and “performance”, “at_lest” confuse the model a bit more, creating more room for misclassifications. From the 985 terms, 157 had a negative value and 828 had a positive value. You can see in figures 1 and 2 the top extremes in terms of positive and negative AUC impact (delta AUC).

Improving AUC value even more

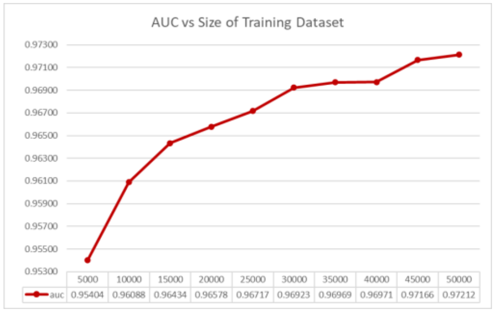
It is possible to improve even more the best CV AUC value found: 0.96717 (XGBoost gblinear). Some alternatives:

1) **Increasing the size of vocabulary:** Adding more terms to vocabulary improves the AUC value. For example by using a vocabulary size of 4,403 terms, the AUC jumps to 0.97. Figure 3 shows this trend for CV AUC values using XGBoost for different vocabulary sizes.



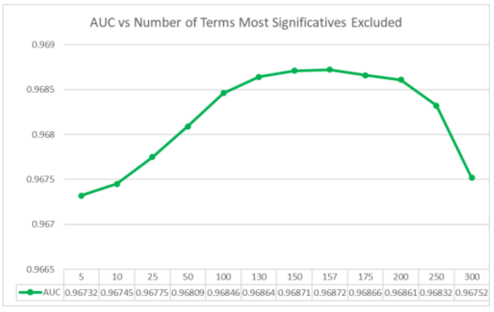
	511	985	1048	1561	2029	2575	2993	3451	4043	4566	5038
auc	0.95956	0.96717	0.96789	0.97044	0.97248	0.97338	0.97424	0.97445	0.97468	0.97453	0.97459

Figure 3: Effect of vocabulary size in the AUC



	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000
auc	0.95404	0.96088	0.96434	0.96578	0.96717	0.96923	0.96969	0.96971	0.97166	0.97212

Figure 4: Effect of size of training dataset in the AUC



	5	10	25	50	100	130	150	157	175	200	250	300
AUC	0.96732	0.96745	0.96775	0.96809	0.96846	0.96864	0.96871	0.96872	0.96866	0.96861	0.96832	0.96752

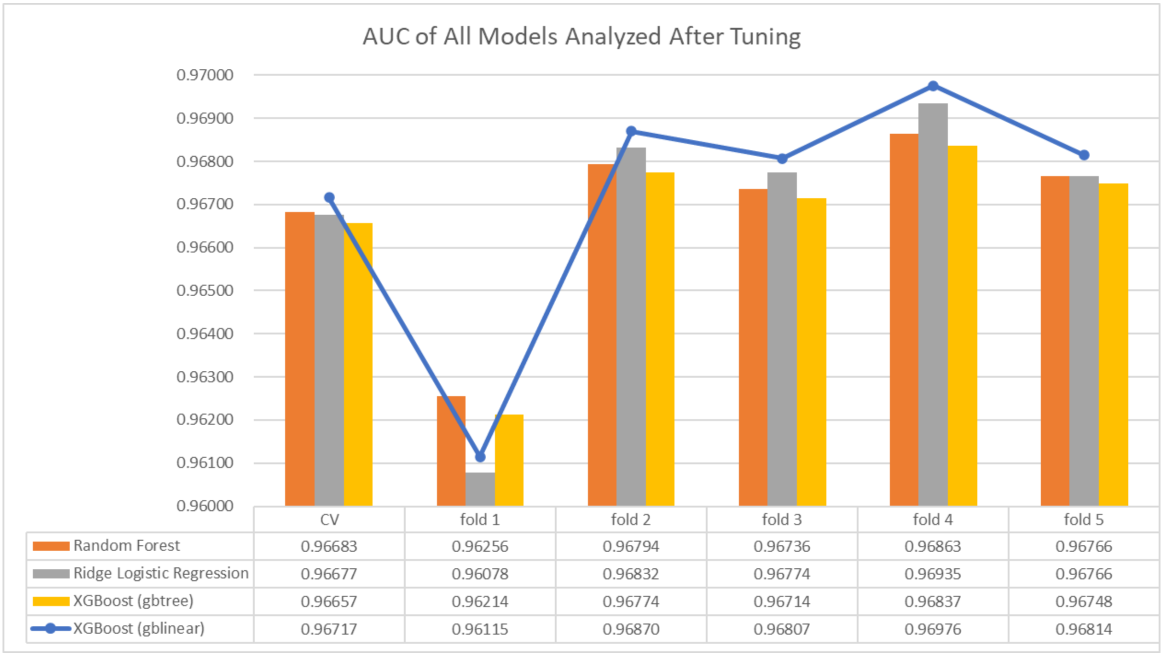
Figure 5: Effect of exclusion of vocab terms in the AUC

2) **Using a training dataset bigger than 25K observations:** Figure 4 shows the graph of CV AUC using XGBoost with a vocabulary of 985 words and different sizes of training dataset from 5,000 to 50,000 observations. For example, by training the XGBoost gblinear model with all 50K observations the CV AUC jumped to 0.97212,
3) **Excluding from the vocabulary terms that less contribute to the accuracy:** as mentioned in the interpretability session, 828 terms contributed positively for an AUC of 0.96717 (XGBoost) and 157 contributed negatively for the AUC. Figure 5 shows the graph of CV AUC after excluding from 985-term vocabulary the top N terms that less contribute to AUC. The effect is positive until the 157th term. For example, by excluding all 157 terms that less contribute to AUC (negative effects) would have generated a CV AUC of 0.96872.

I didn't used any of these alternatives. Alternative 1 is a requirement. Alternative 2 would mean to use the 25K test data to train the model with 50K observations, what would be a type of "cheating". Alternative 3 also use information from test dataset to test and create an index, what would be also consider a type of "cheating". So, I keep my best model with only the training dataset from fold 1. Choosing a different fold other than fold 1 didn't produce any significant change in the AUC value.

Conclusion

I learned a lot in this project, spending most of the time tuning parameters, testing different models and understanding what impacted more the AUC metric. Figure 6 shows the graph of the 4 models analyzed and tuned in this project and in blue the best model: XGBoost with gblinear booster. In the future, I plan to automate the grid search and tunning process using R packages such as MLR and also install XGBoost R package optimized for GPU to accelerate the tuning process.



	CV	fold 1	fold 2	fold 3	fold 4	fold 5
Random Forest	0.96683	0.96256	0.96794	0.96736	0.96863	0.96766
Ridge Logistic Regression	0.96677	0.96078	0.96832	0.96774	0.96935	0.96766
XGBoost (gbtree)	0.96657	0.96214	0.96774	0.96714	0.96837	0.96748
XGBoost (gblinear)	0.96717	0.96115	0.96870	0.96807	0.96976	0.96814

Figure 6: Summary of all models analyzed and tuned in this project. In blue the model with highest CV AUC.