Faculty of Science
Master of Statistics and Data Science
Academic year: 2020 - 2021

Collecting and Analyzing Big Data [S0K17a]

Text Classification Summary

by

Khachatur Papikyan r0825613

For the text classification assignment multiple methods along with their different configurations have been tried and evaluated. The list includes probabilistic models, tree-based ensembles and others. In particular, Logistic Regression, AdaBoost, Random Forests, Gradient Boosting, Extreme Gradient Boosting (i.e. XGBoost) and K-nearest neighbours were attempted both with their default settings and custom configurations. The final report is based on Logistic Regression and XGBoost, which have yielded my personal best results according to their accuracy scores on the holdout test set submitted on the course Kaggle site. It is worth mentioning that in the context of current assignment a multiclass classification problem is being addressed, because every observation belongs to only one label, otherwise the problem would have emerged into multilabel classification, where each movie could belong to multiple genres.

1) Logistic Regression

Before applying the model, first the distribution of the genres is visualised for descriptive purposes. It is clear that certain labels such as "drama" and "comedy" are dominating in the dataset by their counts. But in the seeding stage of modeling this fact was omitted and the model was built without touching the initial distribution of genres shown in Figure 1. As expected based on this distribution, the probabilistic model such as Logistic Regression trained on this data would mostly classify observations as "drama" or "comedy" and suffer from recognizing the other genres. In order to escape from this type of overfitting, a custom upsampling technique is defined to make the observations equally distributed across the genres. In essence, the dataset is augmented by randomly sampling N observations from the less frequent genres with replacement, where N is the difference between the count of the most frequent genre and the count of each of the less frequent genres, and adding those samples to the dataset. Figure 2 shows the distribution of genres after upsampling.

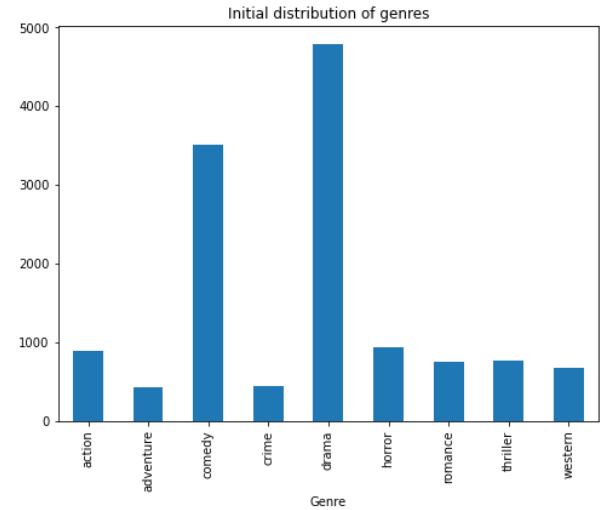

Figure 1: Initial distribution of genres

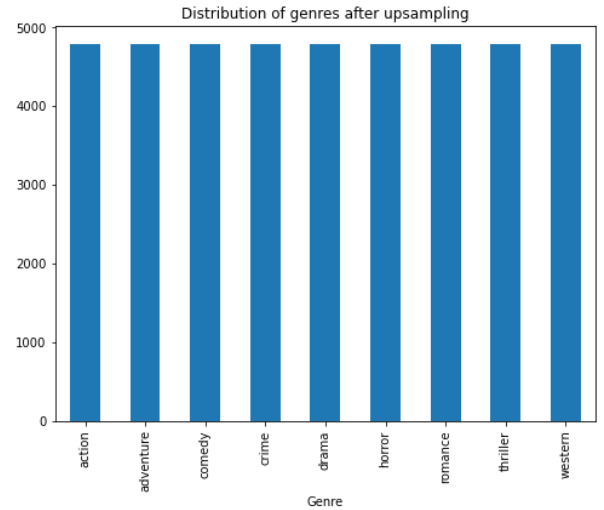

Figure 2: Distribution of genres after upsampling

After upsampling, two custom functions are defined for cleaning the dataset and stemming. Both of the functions are important from the point of making the texts more comparable across the observations for the machine. After the cleaning phase punctuations, enumerations and any other non-alphabetic symbols are removed, also the text is converted to lowercase. Through the stemming phase the words are reduced to their stems. Code snippets from Figure 3 and 4 demonstrate the codes for defining those custom functions.

```
# Defining a function for the Plot text cleaning
def clean_text(text):
    # remove backslash-apostrophe
    text = re.sub("\'", "", text)
    # remove everything except alphabets
    text = re.sub("[^a-zA-Z]"," ",text)
    # remove whitespaces
    text = ' '.join(text.split())
    # convert text to lowercase
    text = text.lower()
    return text
```

Figure 3: the clean_text() function

```
# Defining a function for stemming the Plot text
def stem_text(sentence):
    new_sentence = ''
    for word in word_tokenize(sentence):
        new_sentence = new_sentence + ' ' + eng_stemmer.stem(word)
    return new_sentence
```

Figure 4: the stem_text() function

As soon as the text is cleaned and stemmed, the training data is further randomly split into 80% training and 20% validation sets in order to have an opportunity of checking the performance of the model before submitting a final check on the holdout unlabeled test set on Kaggle. Different proportions apart from the 80/20 split were tried as well, but 80-20 seemed to be an optimal one.

The modeling is done through pipeline, which includes a CountVectorizer for making a feature matrix from the text, a TfidfTransformer for taking into account the term frequencies and inverse document frequencies of the text vectors in the feature matrix and a Logistic Regression classifier as 3 consecutive steps. Different parametrization options are defined in a dictionary of parameters for the 3 components of the pipeline. Multiple models (in total 144) are fitted through grid search by using different combinations of parameters and the best performing model along with its parameters is retrieved. The whole model building process is demonstrated through the code snippet in Figure 5. Certain parameters, such as max_iter=1000 or stop_words='english', were predefined directly in the pipeline in order to save the running time by decreasing the number of different models to fit, because they were already checked with grid search to be the optimal ones.

```
# Creating a Pipeline with vectorizer, tf_transformer and Logistic Regression classifier
pipeline_LogReg = Pipeline([
                    ('vectorizer' , CountVectorizer(lowercase = True, max_features = None, stop_words='english')),
                    ('tf_transfomer', TfidfTransformer()),
                    ('classifier' , LogisticRegression(max_iter=1000, random_state=20))
                   ])

# Creating a dictionary object with different parametrizations for the Pipeline to try
parameters_LogReg = {'vectorizer__max_df' : [.1, .4, .8, 10],
                     'vectorizer__min_df' : [.00005, .001, 5],
                     'vectorizer__ngram_range' : [(1, 1), (1, 2)],
                     'classifier__solver' : ['liblinear', 'saga']
                }

# Initializing GridSearchCV object
grid_search = GridSearchCV(pipeline_LogReg,
                           parameters_LogReg,
                           n_jobs = -1,
                           cv = 3,
                           #scoring = 'f1',
                           verbose = 1)

# Fitting the models and finding the best combination of parameters for the pipeline
grid_search.fit(Train['Plot'],
                Train['Genre'])
```

Figure 5: Modeling process

The best accuracy score from 3-fold cross validation is 0.9, with max_df=0.8, min_df=5e-05 and ngram_range=(1, 2) for CountVectorizer and solver='saga' (this point is not surprising because ‘saga’ is said to be more stable and works faster with large datasets according to the documentation) for Logistic Regression classifier.

Training accuracy is 97% and validation accuracy is almost 93%, which signifies a slight overfitting, but not very significant.

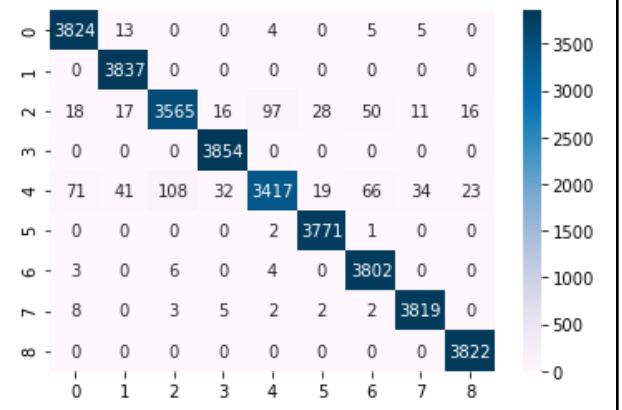

Figure 6: Confusion matrix of the best estimator on training set

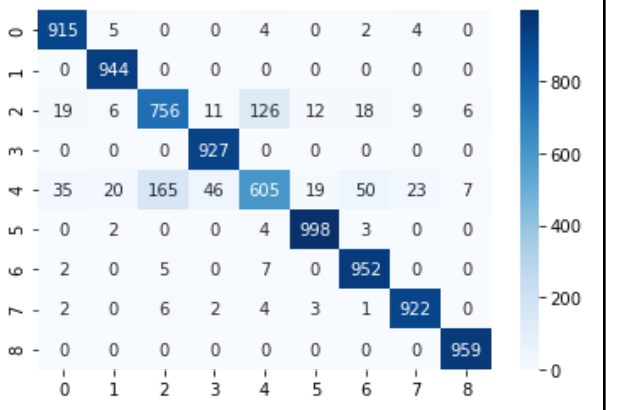

Figure 7: Confusion matrix of the best estimator on validation set

Confusion matrices from Figure 6 and 7 suggest that after upsampling the model may sometimes have difficulties with classifying comedies and dramas, which were the 2 most frequent genres of the initial training data before upsampling.

Finally, before making predictions on the holdout unlabeled test, the dataset is cleaned and stemmed with the above mentioned clean_text() and stem_text() functions and then the predictions are made. According to the submissions on Kaggle the model shows 61% accuracy on the holdout unlabeled test set.

2) Extreme Gradient Boosting (XGBoost)

XGBoost is known to be one of the state-of-the-art tree-based modeling techniques and is said to dominate many Kaggle competitions on a variety of datasets due to its stability. The full process includes the following steps:

1. Loading the training and the holdout unlabeled test sets.
2. Cleaning and stemming the texts for both of the datasets (This time NO upsampling was done on the training set, because this way it was possible to achieve higher accuracy score in Kaggle submissions).
3. Randomly splitting the training set into 80% training and 20% validation sets.
4. Modeling (CountVectorizer, TfidfTransformer and XGBoost Classifier respectively in the pipeline) and grid searching with 10-fold cross validation for the best combination of hyperparameters (Note that only the final version of the grid searched optimal parameters is included in the notebook, because the algorithm itself is heavy and it takes hours to run with increasing number of parameters' combinations).
5. Checking the predictions on training and validation sets.
6. Making predictions on the holdout unlabeled test set.

The best accuracy score of the model after 10-fold cross-validation is 0.57 (in fact, almost the same was even with 5-fold CV), with max_df=0.4, min_df=5e-05 and ngram_range=(1, 1) for CountVectorizer and directly predefined early_stopping_rounds=5 for XGBoost classifier.

Training accuracy is 96% and validation accuracy is around 58%, which means that the model is clearly still highly overfitting. But the good point here is that on the holdout unlabeled test set as well the accuracy after submission on Kaggle is 56-57%, consequently with this modeling approach the results from validation set and final test set are close to each other and more stable in that sense.