NLP+CLASSIFICATION PROJECT

Francisco Jesús Díaz Pellejero†, Tamara Redondo Soto†
and Javier Villar Asensio†

Contributing authors: Fco.Jesus.Diaz@alu.uclm.es;
Tamara.Redondo@alu.uclm.es; Javier.Villar@alu.uclm.es;
†These authors contributed equally to this work.

Abstract

Use Machine Learning to analyze a collection about comments on purchases about edible and drinkable products, using Natural Language Processing highlighting into preprocessing, vectorization and classification algorithms.

1 Introduction

We have a file "products.csv" with different fields (Id, ProductId, ProfileName, HelpfulnessNumerator, HelpfulnessDenominator, Score, Time, Summary and Text), which gives us information about the comments on these products.

We have to transform the natural language of these comments into a formal language, that computers can process, using the preprocessing in the Summary+Text field, eliminating useless characters, contractions, emoticons, etc.

Give meaning by vectorizing the text with different methods, select the best features with SelectKBest and finally classify the algorithm using the selected best features in two different ways.

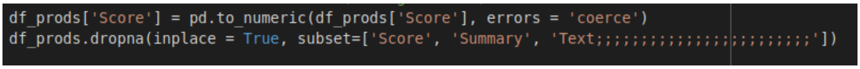
2 Methods and materials

2.1 Preprocessing

We are using pandas' library dataframes as the storage of our dataset. Text prepocessing is the first step in the process of building a model, first all capital letters will be changed using Lower Casing and we remove the blank rows. Also replace useless characters like:

!,”, (), /, =,’ ∗, @

and use lematization with verbs, adjectives, adverbs and nouns to reduce the words to an existing term in the language. Null values in the fields **Summary**, **Text** and **Score** were removed reducing the number of rows to 9000. There were too many of them that would cause problems.



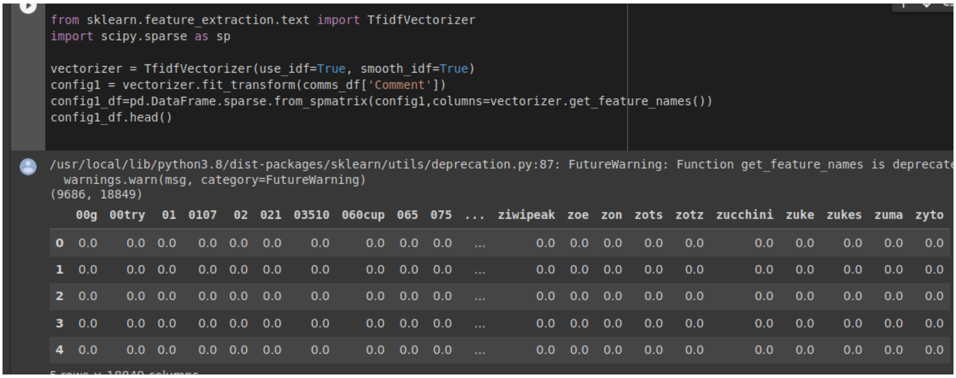
```
df_prods['Score'] = pd.to_numeric(df_prods['Score'], errors = 'coerce')
df_prods.dropna(inplace = True, subset=['Score', 'Summary', 'Text;;;;;;;;;;;;;;;;;;;;;;;;'])
```

2.2 Vectorization

Vectorization is the classification of text into a vector, we have used 4 different options:

2.2.1 TFIDF

TF-IDF (Term Frequency-Inverse Document Frequency) is how we measure in machine learning the importance of strings (words, phrases, lemmas, etc) in a document or a collection of documents. The library *sklearn* provides us a vectorizing algorithm *TfidfVectorizer* whic allows us to fit and transform our dataframe into a vector:



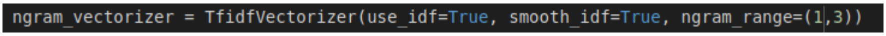
```
from sklearn.feature_extraction.text import TfidfVectorizer
import scipy.sparse as sp

vectorizer = TfidfVectorizer(use_idf=True, smooth_idf=True)
config1 = vectorizer.fit_transform(comms_df['Comment'])
config1_df=pd.DataFrame.sparse.from_spmatrix(config1,columns=vectorizer.get_feature_names())
config1_df.head()
```

```
/usr/local/lib/python3.8/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning: Function get_feature_names is deprecate
  warnings.warn(msg, category=FutureWarning)
(9686, 18849)
   00g  00try  01  0107  02  021  03510  060cup  065  075  ...  ziwipeak  zoe  zon  zots  zotz  zucchini  zuke  zukes  zuma  zyto
0  0.0    0.0 0.0   0.0 0.0  0.0    0.0     0.0  0.0  0.0  ...       0.0  0.0  0.0   0.0   0.0       0.0   0.0    0.0   0.0   0.0
1  0.0    0.0 0.0   0.0 0.0  0.0    0.0     0.0  0.0  0.0  ...       0.0  0.0  0.0   0.0   0.0       0.0   0.0    0.0   0.0   0.0
2  0.0    0.0 0.0   0.0 0.0  0.0    0.0     0.0  0.0  0.0  ...       0.0  0.0  0.0   0.0   0.0       0.0   0.0    0.0   0.0   0.0
3  0.0    0.0 0.0   0.0 0.0  0.0    0.0     0.0  0.0  0.0  ...       0.0  0.0  0.0   0.0   0.0       0.0   0.0    0.0   0.0   0.0
4  0.0    0.0 0.0   0.0 0.0  0.0    0.0     0.0  0.0  0.0  ...       0.0  0.0  0.0   0.0   0.0       0.0   0.0    0.0   0.0   0.0
```

2.2.2 TFIDF + Ngram

Ngram is an algorithm that binds words in a sentence that are consecutive and correlated, so it creates more meaningful subsections when vectorizing.

TfidfVectorizer has a parameter in which you can indicate wether you want to use the Ngram algorithm and the range of words you want to pick for each vector index, we have considered that three is a good maximun amount of words that might have a different meaning. For example; **"come up with"** is a phrasal verb with its own meaning and **"come", "up", "with"** are different words with different meanings when they are not consecutive.

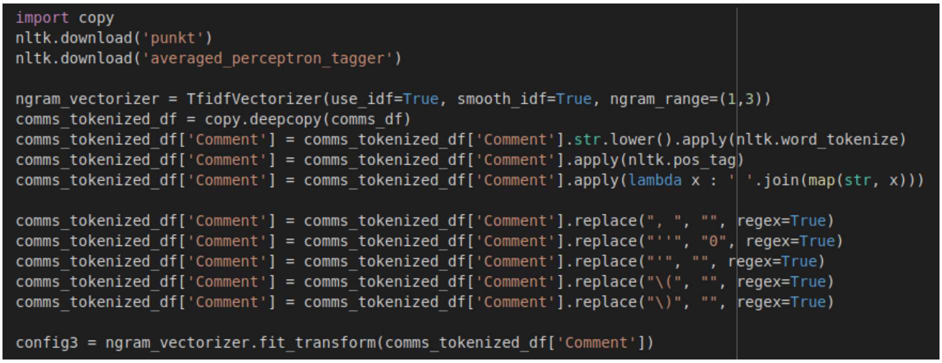


```
ngram_vectorizer = TfidfVectorizer(use_idf=True, smooth_idf=True, ngram_range=(1,3))
```

2.2.3 TFIDF + Ngram + POS tagging

POS tagging is an algorithm to classify words into their cathegorical type (verb, noun, adverb, etc). *nltk* library can be used to POS tag words, we tokenize the words of our dataset and then we apply to every word the algorithm to be tagged.

Once tagged, we remove the special characters that were added when tagging and do the same thing than the previous configuration, this way we have applied the three techniques.

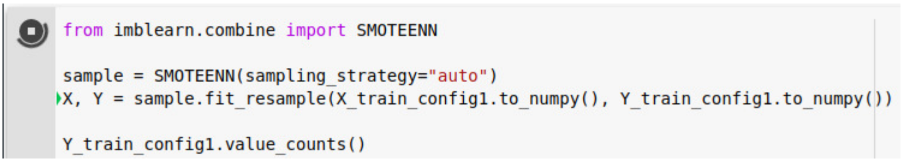


```
import copy
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

ngram_vectorizer = TfidfVectorizer(use_idf=True, smooth_idf=True, ngram_range=(1,3))
comms_tokenized_df = copy.deepcopy(comms_df)
comms_tokenized_df['Comment'] = comms_tokenized_df['Comment'].str.lower().apply(nltk.word_tokenize)
comms_tokenized_df['Comment'] = comms_tokenized_df['Comment'].apply(nltk.pos_tag)
comms_tokenized_df['Comment'] = comms_tokenized_df['Comment'].apply(lambda x : ' '.join(map(str, x)))

comms_tokenized_df['Comment'] = comms_tokenized_df['Comment'].replace(", ", "", regex=True)
comms_tokenized_df['Comment'] = comms_tokenized_df['Comment'].replace("''", "0", regex=True)
comms_tokenized_df['Comment'] = comms_tokenized_df['Comment'].replace("'", "", regex=True)
comms_tokenized_df['Comment'] = comms_tokenized_df['Comment'].replace("\(", "", regex=True)
comms_tokenized_df['Comment'] = comms_tokenized_df['Comment'].replace("\)", "", regex=True)

config3 = ngram_vectorizer.fit_transform(comms_tokenized_df['Comment'])
```

2.3 Feature Selection

Now we have vectorized every word of the dataset, but we have a problem, most of the words are not useful words or are not even correctly spelled and the amount of words recognised (or tokens because some of them are not even words) is too large.
With **feature selection** we can get rid of the percentage of features we do not want/need, we are using just a 30% of the features.
We are using de *SelectKBest* method being **K** the number of features we want. Once set up the algorithm, we fit the model to our dataset; the vectorized dataset is the input (**X**) and the wanted value (**Y**) is the score of the users in the original dataset. And finally we transform the result, this creates the dataset with only the features we got when fitting.
The procedure is exactly the same with each of the 3 configurations of vectorization.

We have tried to unbalance the results of the reduction so we can have better results in the classification, as the original dataset has a majority of 5 stars results, this might cause a too optimistic prediction model but, as we are working with google colab, we have RAM memory constraints and we have not enough space to do so. Despite that there is the code we run:

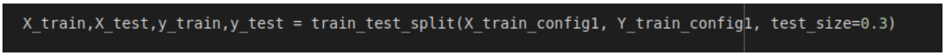


```
from imblearn.combine import SMOTEENN

sample = SMOTEENN(sampling_strategy="auto")
X, Y = sample.fit_resample(X_train_config1.to_numpy(), Y_train_config1.to_numpy())

Y_train_config1.value_counts()
```

2.4 Classifications

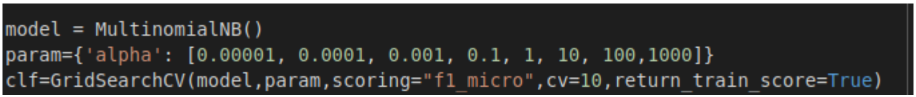
The methods of classification used were **NaiveBayes** and **TreeClassifier** and the procedure is exactly the same with each of the three vectorizations so we show just one of them.

First of all, we split the dataset, 70% will be used for training and the 30% left will be the test.

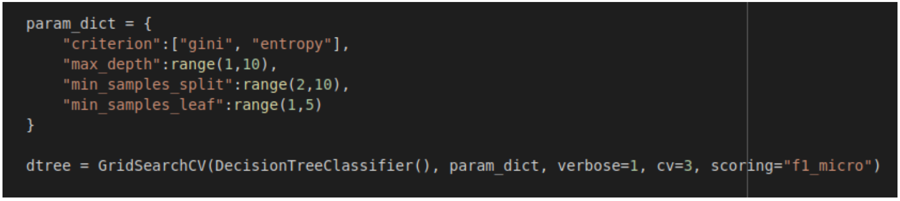


```
X_train,X_test,y_train,y_test = train_test_split(X_train_config1, Y_train_config1, test_size=0.3)
```

Now we set the configuration of the models. We are using the *GridSearchCV* in which we have to indicate the model (multinomial NB/tree classifier), its hyperparameters; in NB we only have *alpha* which is a smoothing factor and in the tree classifier some settings for the creation of the tree and the cross validation number.



```
model = MultinomialNB()
param={'alpha': [0.00001, 0.0001, 0.001, 0.1, 1, 10, 100,1000]}
clf=GridSearchCV(model,param,scoring="f1_micro",cv=10,return_train_score=True)
```



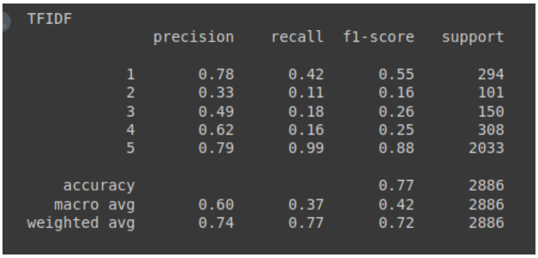
```
param_dict = {
    "criterion":["gini", "entropy"],
    "max_depth":range(1,10),
    "min_samples_split":range(2,10),
    "min_samples_leaf":range(1,5)
}

dtree = GridSearchCV(DecisionTreeClassifier(), param_dict, verbose=1, cv=3, scoring="f1_micro")
```

Now we are ready to fit and predict.

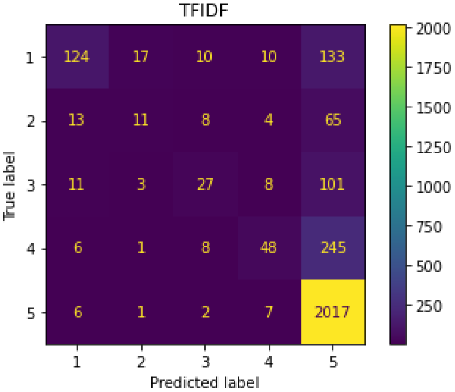
3 Experiments and results

Now we are showing the results and a final comparison. We will show every configuration with both classifications.

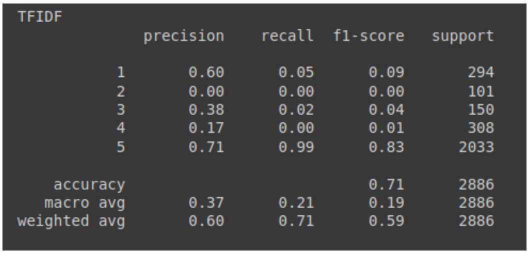
3.1 TFIDF



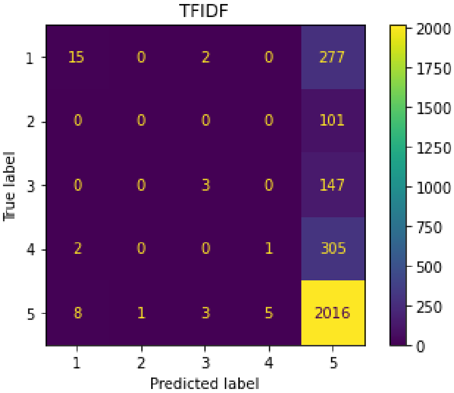
```
TFIDF
              precision    recall  f1-score   support

           1       0.78      0.42      0.55       294
           2       0.33      0.11      0.16       101
           3       0.49      0.18      0.26       150
           4       0.62      0.16      0.25       308
           5       0.79      0.99      0.88      2033

    accuracy                           0.77      2886
   macro avg       0.60      0.37      0.42      2886
weighted avg       0.74      0.77      0.72      2886
```



	1	2	3	4	5
1	124	17	10	10	133
2	13	11	8	4	65
3	11	3	27	8	101
4	6	1	8	48	245
5	6	1	2	7	2017

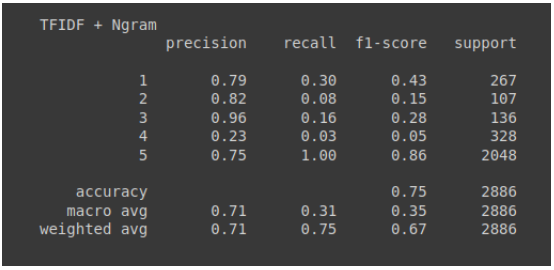


```
TFIDF
              precision    recall  f1-score   support

           1       0.60      0.05      0.09       294
           2       0.00      0.00      0.00       101
           3       0.38      0.02      0.04       150
           4       0.17      0.00      0.01       308
           5       0.71      0.99      0.83      2033

    accuracy                           0.71      2886
   macro avg       0.37      0.21      0.19      2886
weighted avg       0.60      0.71      0.59      2886
```

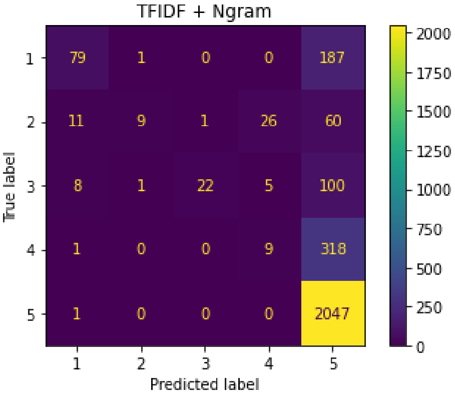


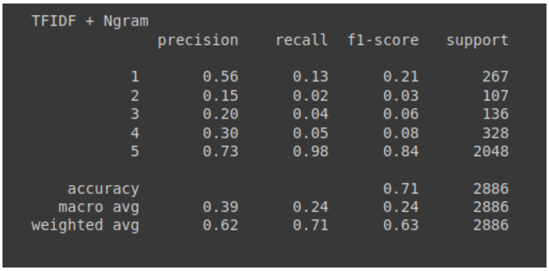
	1	2	3	4	5
1	15	0	2	0	277
2	0	0	0	0	101
3	0	0	3	0	147
4	2	0	0	1	305
5	8	1	3	5	2016

3.2 TFIDF + Ngram

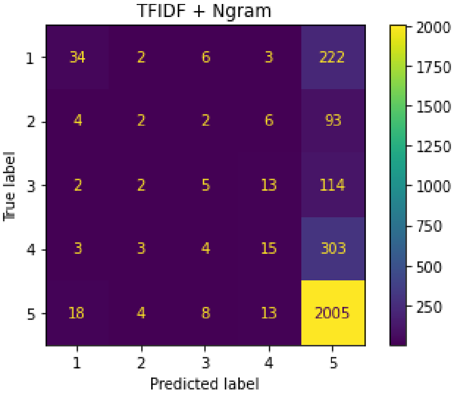


```
TFIDF + Ngram
              precision    recall  f1-score   support

           1       0.79      0.30      0.43       267
           2       0.82      0.08      0.15       107
           3       0.96      0.16      0.28       136
           4       0.23      0.03      0.05       328
           5       0.75      1.00      0.86      2048

    accuracy                           0.75      2886
   macro avg       0.71      0.31      0.35      2886
weighted avg       0.71      0.75      0.67      2886
```

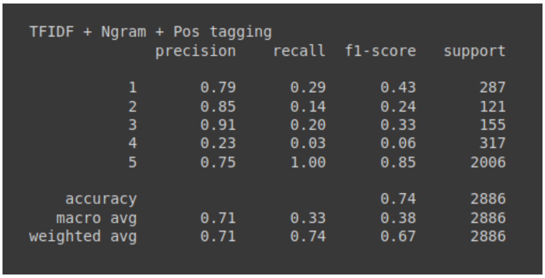




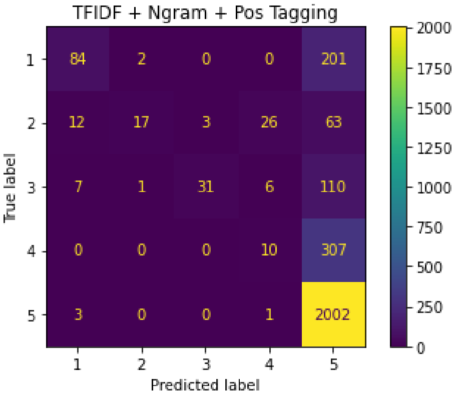
```
TFIDF + Ngram
              precision    recall  f1-score   support

           1       0.56      0.13      0.21       267
           2       0.15      0.02      0.03       107
           3       0.20      0.04      0.06       136
           4       0.30      0.05      0.08       328
           5       0.73      0.98      0.84      2048

    accuracy                           0.71      2886
   macro avg       0.39      0.24      0.24      2886
weighted avg       0.62      0.71      0.63      2886
```



3.3 TFIDF + Ngram + POS tagging

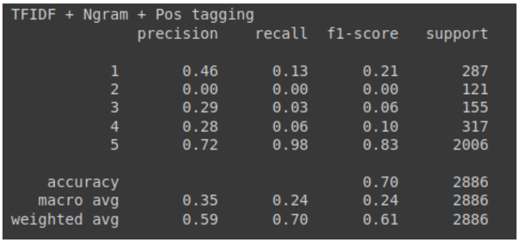


```
TFIDF + Ngram + Pos tagging
              precision    recall  f1-score   support

           1       0.79      0.29      0.43       287
           2       0.85      0.14      0.24       121
           3       0.91      0.20      0.33       155
           4       0.23      0.03      0.06       317
           5       0.75      1.00      0.85      2006

    accuracy                           0.74      2886
   macro avg       0.71      0.33      0.38      2886
weighted avg       0.71      0.74      0.67      2886
```

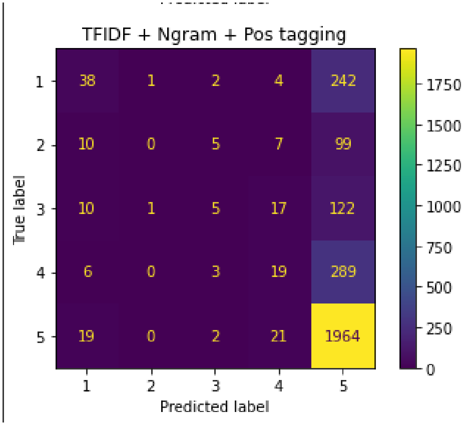


TFIDF + Ngram + Pos Tagging

True label \ Predicted label	1	2	3	4	5
1	84	2	0	0	201
2	12	17	3	26	63
3	7	1	31	6	110
4	0	0	0	10	307
5	3	0	0	1	2002



```
TFIDF + Ngram + Pos tagging
              precision    recall  f1-score   support

           1       0.46      0.13      0.21       287
           2       0.00      0.00      0.00       121
           3       0.29      0.03      0.06       155
           4       0.28      0.06      0.10       317
           5       0.72      0.98      0.83      2006

    accuracy                           0.70      2886
   macro avg       0.35      0.24      0.24      2886
weighted avg       0.59      0.70      0.61      2886
```

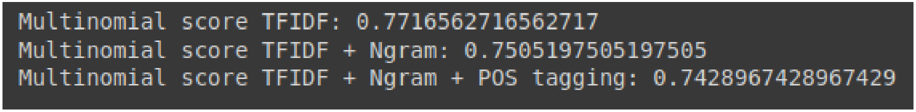


TFIDF + Ngram + Pos tagging

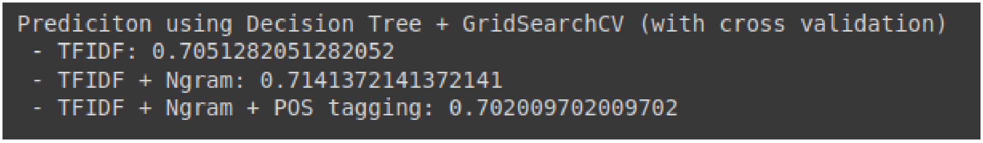
True label \ Predicted label	1	2	3	4	5
1	38	1	2	4	242
2	10	0	5	7	99
3	10	1	5	17	122
4	6	0	3	19	289
5	19	0	2	21	1964

3.4 Results Comparison

As we can see the results are quite similar, being the best in multinomial NB the first configuration and in the tree classifier the second configuration



```
Multinomial score TFIDF: 0.7716562716562717
Multinomial score TFIDF + Ngram: 0.7505197505197505
Multinomial score TFIDF + Ngram + POS tagging: 0.7428967428967429
```



```
Prediciton using Decision Tree + GridSearchCV (with cross validation)
 - TFIDF: 0.7051282051282052
 - TFIDF + Ngram: 0.7141372141372141
 - TFIDF + Ngram + POS tagging: 0.702009702009702
```

4 Conclusions

Seeing the results we see that the effort of adding more prepocessing to the dataset may be worthless due to the little difference in the results, actually the best configuration is the simplest one.

Also we know that the predictions are too optimistic, there are some reasons for that; the models used may be more susceptible to have this kind of behaviour, maybe other classifier would have been more adequate but their computational time would be too long (we are working with Google Colab and the dataset is large so better algorithms would be hard to use with this environment) and we could not unbalance the dataset (due to the same reason).

Any other dataset with higher entropy of stars given by the users would have been better so the fitting the model would learn more of every prediction in the range of the results (0-5).

Nonetheless, this may be useful to detect/predict five stars rank products.