NLP + CLASSIFICATION

Miguel Ángel Matas Rubio Jesús Santiyán Reviriego

Miguel Ángel Roldán Mora

January 23, 2023

Abstract

A study is going to be carried out on a .csv database containing customer opinions on a series of food and drink products, evaluating these products in order to catalogue and process the data in the future.

Contents

1 Problem description

In this practice a study is going to be carried out on a database (Vectorization, Feature selection, Classification algorithm) in which the opinions of the clients about the products of the shop are stored, apart from saving the id, product, etc. The opinions are saved in numerical base from 1 to 5 and a comment in text, being a comment in text mode this may have typos so they must be corrected automatically by using algorithms, we must also maintain a more easy understanding so they will eliminate the contractions with which has the English and finally the data will be treated to remove unnecessary things such as words are duplicated, emoticons or special characters that should not be.

The source file provided to us is "products.csv", as it is a csv all the data come in the same cell but separated by ";", the first way to treat the file is going to be with the excel text editor, with which we need to separate all the data in different cells. Therefore it is necessary to take the file provided in our delivery as this first action is not done in the colab and without our source file the program will not work correctly[1].

2 Methods and materials

In the following, the classification processes will be specified, which also include vectorisation processes, as well as the preprocessing of the data, which will provide cleaner data for better classifications.

2.1 Preprocessing

As it has been said before, one of the functions to be carried out was the processing of the data in the database where, apart from making a lemmatization of all the terms in the database, some text corrections were also going to be made, where the changes to be made are as follows[1]:

- Remove useless characters: ! " $ % & / ( ) = _ ^ * ¡ @

```
[8]  df = df.applymap(lambda x: re.sub(r"[!_$%&/()=^*¡@]", "", x) if type(x) == str else x)
```

Figure 1: Remove chars

- Remove all capital

```
import pandas as pd
import re


df = pd.read_csv("products.csv", sep=';',encoding='latin1')

df.drop(['Unnamed: 10','Unnamed: 11','Unnamed: 12','Unnamed: 13','Unnamed: 14','Unnamed: 15','Unnamed: 16'], inplace = True,axis=1)
df.head()
df2 = df[['Summary','Text']]
df['Summary+Text'] = df["Summary"].str.cat(df["Text"], sep =' ')
df.head()
df.dropna(inplace = True)
df = df.apply(lambda x: x.str.lower() if(x.dtype == 'object') else x)
```

Figure 2: Data and lower

In figure 2, only the last line refers to removing capital letters, the rest of the code shown is used to read the csv, remove unnecessary columns and add a new "Summary+Text".

- Lemmatize all terms

```
[4]
24s
from nltk.stem import WordNetLemmatizer


#Lemmatization
lemmatizer = WordNetLemmatizer()
lem_all = []
for comment in df['Summary+Text'].values:
  lem_comment = []
  for token in comment.split():
    lem_comment.append(lemmatizer.lemmatize(token, pos='v'))
  lem_all.append(' '.join(lem_comment))

df['Summary+Text'] = lem_all
```

Figure 3: Lemmatizer

Optional preprocessing steps:

- Remove contractions

```
#contractions detection

contractions_list = [
    (r'ain\'t','am not'), (r'aren\'t','are not'),(r'can\'t','cannot'),(r'could\'ve','could have'),(r'couldn\'t','could not'),(r'didn\'t','did not'),
    (r'doesn\'t','does not'),(r'don\'t','do not'),(r'hadn\'t','had not'),(r'hasn\'t','has not'),(r'haven\'t','have not'),(r'he\'d','he would'),
    (r'he\'ll','he will'),(r'he\'s','he is'),(r'how\'d','how did'),(r'how\'ll','how will'),(r'how\'s','how is'),(r'I\'d','I would'),(r'I\'ll','I will'),
    (r'I\'m','I am'),(r'I\'ve','I have'),(r'isn\'t','is not'),(r'it\'s','it is'),(r'let\'s','let us'),(r'might\'ve','might have'),(r'mightn\'t','might not'),
    (r'must\'ve','must have'),(r'mustn\'t','must not'),(r'needn\'t','need not'),(r'o\'clock','of the clock'),(r'shan\'t','shall not'),(r'she\'d','she would'),
    (r'she\'ll','she will'),(r'she\'s','she is'),(r'should\'ve','should have'),(r'shouldn\'t','should not'),(r'so\'s','so is'),(r'that\'s','that is'),
    (r'there\'s','there is'),(r'they\'d','they would'),(r'they\'ll','they will'),(r'they\'re','they are'),(r'they\'ve','they have'),(r'wasn\'t','was not'),
    (r'we\'d','we would'),(r'we\'ll','we will'),(r'we\'re','we are'),(r'we\'ve','we have'),(r'weren\'t','were not'),(r'what\'ll','what will'),
    (r'what\'re','what are'),(r'what\'s','what is'),(r'what\'ve','what have'),(r'when\'s','when is'),(r'when\'ve','when have'),(r'where\'d','where did'),
    (r'where\'s','where is'),(r'where\'ve','where have'),(r'who\'d','who would'),(r'who\'ll','who will'),(r'who\'s','who is'),(r'who\'ve','who have'),
    (r'why\'s','why is'),(r'why\'ve','why have'),(r'will\'ve','will have'),(r'won\'t','will not'),(r'would\'ve','would have'),(r'wouldn\'t','would not'),
    (r'y\'all','you all'),(r'you\'d','you would'),(r'you\'ll','you will'),(r'you\'re','you are'),(r'you\'ve','you have')
]

contractions_list = [(re.compile(regex), not_contracted) for (regex, not_contracted) in contractions_list]

all = []

for comment in df['Summary+Text']:
  z = comment

  for (pattern, not_contracted) in contractions_list:

    (s, count) = re.subn(pattern, not_contracted, comment ,count = 0)
    if s is not comment:
        z = s
        comment = s


  all.append(z)


df['Summary+Text'] = all
```

Figure 4: contractions

- Remove repeated words

```
[6]  def Eli_eliminar_pal_contiguas(frase):
         e = []
         previous_word = ""
         for word in frase.split():
             if word != previous_word:
                 e.append(word)
             previous_word = word
         return " ".join(e)


     for comment in df['Summary+Text']:
       comment = Eli_eliminar_pal_contiguas(comment)
```

Figure 5: Remove repeated

- Remove or replace emoticons

```
def deEmoji(text):
    emoticons_list = [':)', ':(', ':D', ':P', ';)',':()',]
    for emoticon in emoticons_list:
        text = text.replace(emoticon, "")
    return text


for comment in df['Summary+Text']:

  comment = deEmoji(comment)
```

Figure 6: Remove emojis

2.2 TFIDF

TFIDF stands for "term frequency-inverse document frequency." It is a method for determining the importance of a word in a document within a set of documents. The importance is determined by the number of times a word appears in the document (term frequency) and how rarely the word appears in the overall set of documents (inverse document frequency). This results in a score for each word, which can be used to determine the most important words in a document or set of documents. It's commonly used in information retrieval and text mining tasks[2].

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(use_idf=True, smooth_idf=True)
tfidf = vectorizer.fit_transform(df['Summary+Text'])

print(tfidf.shape)
```

```
(50441, 40690)
```

Figure 7: TFIDF

When fitting the model, only the Summary+Text column is used as the rest of the dataset is irrelevant in these processes.

2.3 TFIDF + N-grams

An n-gram is a contiguous sequence of n items from a given sample of text or speech. The items can be phonemes, syllables, letters, words or base pairs, depending on the application. N-grams are often used in natural language processing and computational linguistics to identify the structure and meaning of text. They are also used in speech recognition, machine translation, and text generation. The most commonly used n-grams are unigrams (individual words), bigrams (word pairs), and trigrams (word triplets). The larger the value of n, the more context is included in the n-gram, but the less frequent the n-grams are in the text[3].

```
vectorizer = TfidfVectorizer(ngram_range=(1,3))
ngram = vectorizer.fit_transform(df['Summary+Text'])

print(ngram.shape)
```

```
(50441, 2603314)
```

Figure 8: N-grams

The use of n-grams is controlled by the parameter "ngram_range" with its first parameter being the minimum and the second parameter the maximum. In our case, we make use of the most common ones, which as mentioned before are unigrams, bigrams and trigrams.

2.4 TFIDF + N-grams + POS tagging

POS (part-of-speech) tagging is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition and its context. Tagging is used to help disambiguate grammatical and semantic roles in sentences. Common POS tags include noun, verb, adjective, adverb, and pronoun. POS tagging is a fundamental task in natural language processing (NLP) and is used in a wide range of NLP applications, such as syntactic parsing, semantic role labeling, and named entity recognition[4].

```
from nltk.tokenize import word_tokenize
df['tokens'] = [word_tokenize(sentence) for sentence in df['Summary+Text']]
df['tokens']
tagged = df['tokens'].apply(nltk.pos_tag)
pos_tag = []
sentence = ''
for comment in tagged:
  line = ''
  sentence = ''
  for word in comment:
    line = f'{word[0]}:{word[1]}'
    sentence += ' '+ ''.join(line)
  pos_tag.append(sentence)
df['post_tag'] = pos_tag
vectorizer = TfidfVectorizer(ngram_range=(1,3))
pos_tag = vectorizer.fit_transform(pos_tag)
```

Figure 9: POS tagging

3 Experiments and results

3.1 Selection and Hyperparameter

In figure 10, we select the best features by using the selectKBest and eliminating 70% of the features used by configuration. To indicate the parameter "k", 30% of the data of the selected configuration is used.

```
from sklearn.feature_selection import SelectKBest, chi2
def selection(x):
  X_clf=SelectKBest(score_func=chi2,k=int(x.shape[1]*0.3)).fit_transform(x,df['Score'])
  return X_clf
```

Figure 10: Feature Selection

In Figure 11, when performing the GridSearch we have not indicated any parameter for cross-validation as we prefer to leave it as default, i.e. cv=5.

```
from sklearn.model_selection import GridSearchCV
from sklearn import svm
from sklearn import model_selection, svm
X_train, X_test, y_train, y_test = model_selection.train_test_split
parameters = {'kernel':('linear',
'rbf'),
'C':[1, 10],
'gamma': [0.0001, 0.0005, 0.001,
0.005, 0.01, 0.1]}
svc = svm.SVC()
clf = GridSearchCV(svc, parameters)
results = clf.fit(X_train, y_train)
print(results.best_score_, results.best_params_)

0.7291407556159686 {'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}
```

Figure 11: Hyperparameter

These processes are necessary to carry out the following models.

3.2 TFIDF

We can see that the results shown have a large number of hits when the score is 5 and in the intermediate values this is not so accurate, but it is logical since the best and worst cataloguing is the easiest to predict. We can also see that the accuracy is 0.75 which indicates that with this model we get a large number of hits with the predicted values.

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(tfidf, (df['Score'].values), test_size=0.3, shuffle=True)

SVM = svm.SVC(C= 10, gamma= 0.1, kernel= 'rbf')

# Training
SVM.fit(X_train, y_train)

# Prediction
result = SVM.predict(X_test)

print(classification_report(y_test, result, labels=[1,2,3,4,5]))
cm = confusion_matrix(y_test, result, labels=[1,2,3,4,5])
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["1","2","3","4","5"])
disp.plot()
```

Figure 12: TFIDF Class

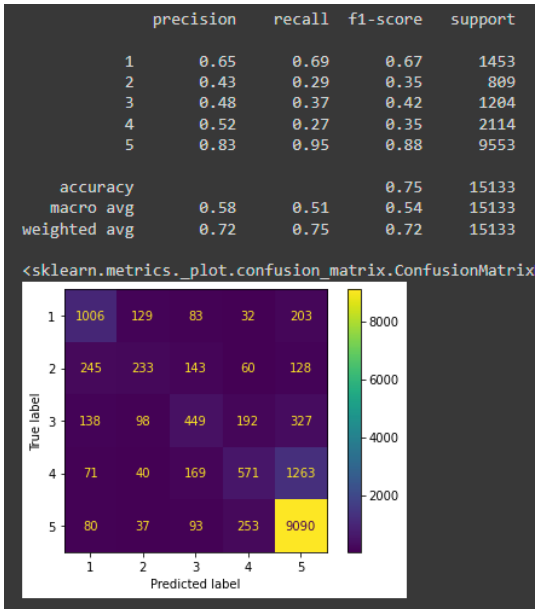

Figure 13: TFIDF Result

3.3 TFIDF + N-grams

Adding N-grams to the predictions, these gain a substantial improvement for the scenario of score 5, which presents almost no failures in its prediction, highlighting that the case of score 4 is the one that improves the most with respect to the model, for the case of the other scores these improve but nothing remarkable compared to the two values mentioned above. It is worth mentioning that the accuracy is 0.87 which indicates the great performance of this model.

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(ngram, df['Score'].values, test_size=0.3, shuffle=True)

SVM = svm.SVC(C= 10, gamma= 0.1, kernel= 'rbf')

# Training
SVM.fit(X_train, y_train)

# Prediction
result = SVM.predict(X_test)

print(classification_report(y_test, result, labels=[1,2,3,4,5]))
cm = confusion_matrix(y_test, result, labels=[1,2,3,4,5])
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["1","2","3","4","5"])
disp.plot()
```

Figure 14: N-grams Class

```
              precision    recall  f1-score   support

           1       0.69      0.80      0.74      1387
           2       0.68      0.36      0.47       857
           3       0.64      0.59      0.61      1251
           4       0.78      0.70      0.74      2181
           5       0.95      1.00      0.97      9457

    accuracy                           0.87     15133
   macro avg       0.75      0.69      0.71     15133
weighted avg       0.86      0.87      0.86     15133

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrix
```

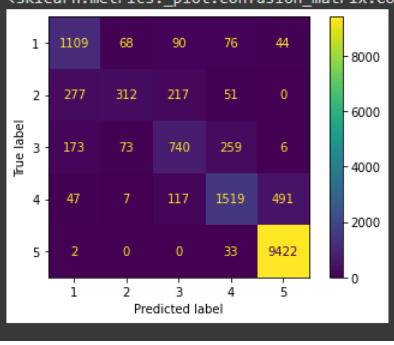

Figure 15: N-grams Result

3.4 TFIDF + N-grams + POS tagging

As the last model added is the POS Tagging, we can see that the accuracy is 0.83 as in the N-grams model, not being necessary to perform this step because there is no improvement and this is because in some scores this model improves but in others it causes less hits, valancing the overall data.

```python
X_train, X_test, y_train, y_test = model_selection.train_test_split(pos_tag, df['Score'].values, test_size=0.3, shuffle=True)

SVM = svm.SVC(C= 10, gamma= 0.1, kernel= 'rbf')

# Training
SVM.fit(X_train, y_train)

# Prediction
result = SVM.predict(X_test)

print(classification_report(y_test, result, labels=[1,2,3,4,5]))
cm = confusion_matrix(y_test, result, labels=[1,2,3,4,5])
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["1","2","3","4","5"])
disp.plot()
```

Figure 16: POS tagging Class

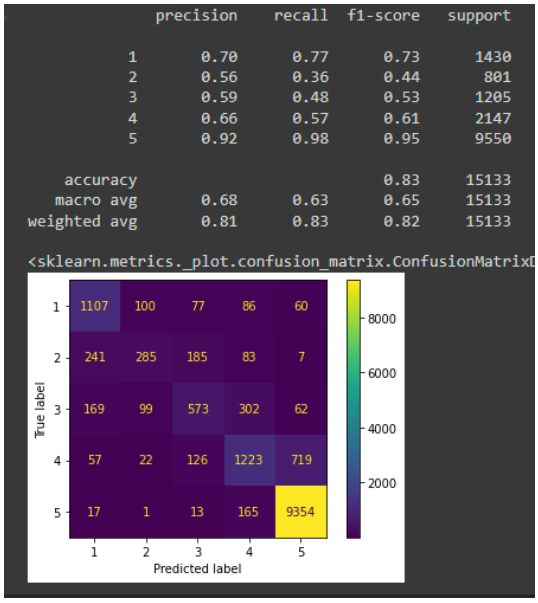

Figure 17: POS tagging Result

4 Conclusions

The data probably have a slightly unbalanced distribution with too many 'Score: 5' but still the results are decent and you can see how the model improves as more techniques are used.

As we have seen in the results section, the results of the N-Grams and POS tagging algorithms are practically identical, changing in some of the predictions but in the final percentage of hits giving the same result. We can see that all these algorithms and metrics of maching learning are quite correct when predicting the results from a plain text, being very good at seeing which are the most relevant words for the classification.

References

[1] JESÚS SERRANO GUERRERO. Nlp+classification project. URL https://campusvirtual.uclm.es/pluginfile.php/5043255/mod_resource/content/1/project.pdf.

[2] Li-Ping Jing, Hou-Kuan Huang, and Hong-Bo Shi. Improved feature selection approach tfidf in text mining. In *Proceedings. International Conference on Machine Learning and Cybernetics*, volume 2, pages 944–946. IEEE, 2002.

[3] Grigori Sidorov, Francisco Velasquez, Efstathios Stamatatos, Alexander Gelbukh, and Liliana Chanona-Hernández. Syntactic n-grams as machine learning features for natural language processing. *Expert Systems with Applications*, 41(3):853–860, 2014.

[4] Milan Straka and Jana Straková. Tokenizing, POS tagging, lemmatizing and parsing UD 2.0 with UDPipe. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 88–99, Vancouver, Canada, August 2017. Association for Computational Linguistics. doi:10.18653/v1/K17-3009. URL https://aclanthology.org/K17-3009.