Machine Learning - Milestone 3
Natural Language Processing

Raúl Barba Rojas[1,1*], Diego Guerrero Del Pozo[1,1†] and Marvin Schmidt[1,1†]

[1*]Department of Computer Science, Universidad de Castilla-La Mancha, Paseo de la Universidad, 4., Ciudad Real, 13071, Castilla-La Mancha, Spain.

*Corresponding author(s). E-mail(s): Raul.Barba@alu.uclm.es;
Contributing authors: Diego.Guerrero@alu.uclm.es;
Marvin.Schmidt@alu.uclm.es;
†These authors contributed equally to this work.

1 Introduction

In E-Commerce applications, customer reviews provide valuable information to both the seller and potential other customers. A review, in our case consisting of a title, a general text section and a rating from one to five stars, may for example be used to measure the quality of a product or the general reception of the product being sold. Star ratings given by customers represent this very metric, the overall opinion about the bought item.

Within this exercise, we aim to solve the classification problem of predicting the star rating of multiple reviews from the title and general text section only. To do this, we'll be utilizing Natural Language Processing techniques as taught within the laboratory section. The dataset used for both the training and testing process originates from `products.csv`, containing 50808 reviews entries of drinkable and edible products. Besides the already mentioned features, the various other features such as `ProductID` need to be discarded.

The suggested workflow defined within the task definition follows the schematic presented during the laboratory session. First, the dataset needs to be preprocessed and prepared for usage. Following this step, a vectorization and feature selection step are required. Finally, the main part is to train a classifier, including parameter tuning. This report presents notable considerations of each step, providing code snippets wherever applicable.

2 Preprocessing

During the preprocessing, the raw dataset is transformed using various steps that benefit the quality of the input to any upcoming steps. Within the task description, both mandatory and optional steps are listed. Our goal is to produce an output dataset using the input CSV by executing a selection of these steps in a defined order. As presented in the laboratory session, the solution to this problem is mostly achievable using the functionality provided by the NLTK library. In this section, we mention each performed preprocessing step of our notebook file[1], as well as further noteworthy details about them.

As mentioned in the introduction, the raw input dataset consists of 50808 rows each defining 9 features using the CSV structure. Listing 1 shows an example entry within the dataset.

Listing 1 Example review within products.csv.

```
"5,B006K2ZZ7K,A1UQRSCLF8GW1T,""Michael D. Bigham """"M. Wassir""""""",
0,0,5,1350777600,Great taffy,""Great taffy at a great price.
There was a wide assortment of yummy taffy.  Delivery was very quick.
If your a taffy lover, this is a deal.""";;;;;;;;;;;;;;;;;;;;;;;;;
```

We observe the following fields.

1. `Id` (number)
2. `ProductId` (String)
3. `UserId` (String)
4. `ProfileName` (String)
5. `HelpfulnessNumerator` (int)
6. `HelpfulnessDenominator` (int)
7. `Score` (int in range 1..5)
8. `Time` (int)
9. `Summary` (String)
10. `Text` (String)

Before continuing with any of the preprocessing steps, the first step is to **extract the three relevant fields**, `Score`, `Summary` and `Text` from each row. Despite the usage of CSV, this step is not trivial due to errors in the present CSV structure. The reason for this is that the sections containing the separator are not properly escaped. Furthermore, we need to remove the misplaced quotation marks and semicolons. To solve these issues, the structure of the malformed CSV was inspected to find the following assumptions.

1. Quotation marks and semicolons have no relevance for this CSV structure, only commas are used for separation.
2. A valid row must contain exactly nine commas or separator characters.
3. If a row contains more than nine commas, it contains at least one “rogue comma”.
4. “Rogue commas” may only occur in `ProfileName`, `Summary` and `Text`. Therefore, if illegal commas were used, fields 5-10 cannot be distinguished unambiguously.

[1] Refers to document: ”`Task03_NLP_Preprocessing_Barba_Guerrero_Schmidt.ipynb`”

First, **all quotation marks and semicolons are removed**. Then, using observations 3 and 4, each row is filtered to remove all commas after the ninth one. Therefore, if a rogue comma occurs in the summary, the contents after the comma become part of the text, which shouldn't reduce the quality of the dataset by a significant amount. If a rogue comma occurs only within the text, its removal has no impact on the quality. If a comma occurs in the profile name, all fields, especially the score, are rendered unusable. This has only been noticed within the steps following the preprocessing and is therefore not handled in this part.

Using this technique, we restore the CSV structure and continue by eliminating all features except for the score, the summary and the text. The processing steps can now be executed sequentially. Trivial steps, such as the **removal of emoji characters**, will be omitted.

First, **HTML-like structures** are eliminated from the summary and text fields, as their removal is not handled by the upcoming steps. This step was added retroactively and deviates from the task description, as it's not asked for. However, it does increase the quality of the dataset greatly, as any content within the tags would be rendered useless otherwise (see Listing 2).

Listing 2 Removal of HTML tags using regex.

```
1  html_tag = re.compile('<.*?>')
2
3  def cleanhtml(line):
4    output_line = re.sub(html_tag, ' ', str(line))
5    return output_line
```

After filtering out special characters and emoji, a **spelling correction step** is implemented. This step can be implemented in essentially two lines of code thanks to the `TextBlob` library shown in the lab sessions (see Listing 3). However, it turned out to require more computational power than expected, taking more than 20 hours of runtime for processing the entire dataset. Keeping in mind the limits of the free Google Colab plan, we decided to not include a spelling correction step for the final preprocessed dataset. Upon the theoretical completion of the spelling correction, the contents of the strings are transposed to lower case.

Listing 3 Spelling correction of text within a `String` object.

```
1  def correct_spelling(line_to_correct):
2    text_blob = TextBlob(line_to_correct)
3    return str(text_blob.correct())
```

The contents of each string are then checked for **repeated terms** using Regex statements. To do this, the example code shown within the laboratory was adjusted slightly (see Listing 4).

Listing 4 Removal of duplicate words within a string.

```
def remove_word_duplicates(line):
  output_line = re
  expression = re.compile(r'\b(\w+)\s+\1\b')
  sub_term = r'\1'

  output_line = expression.sub(sub_term, line)

  if output_line != line: # if term was removed = check for further duplicates
      return remove_word_duplicates(output_line)
  else:
      return output_line
```

Next, we want to **expand so-called word contractions** such as "**aren't**" to "**are not**". We can assume that texts within reviews are filled with common contractions, as well as slang abbreviations such as "'**ol**" instead of "**old**". This step uses an extended list of contractions[2] instead of the suggested one linked in the task description, since it further improves the process quality at the cost of an additional minute of processing. In total, 148 contractions are listed within the resource. Listing 5 shows the implementation of this step.

Listing 5 Expansion of all word contractions within a string.

```
(...)
    (r'you\'re', 'you are'),
    (r'you\'ve', 'you have')
]

contractions_regex = [(re.compile(contracted_word), uncontracted_word)
    ↪ for (contracted_word, uncontracted_word) in contractions]

# removes exactly one contraction
def remove_word_contraction(line):
  # check term for each possible contraction (iterates list of contractions)
  for (contracted_word, uncontracted_word) in contractions_regex:
    (s, count) = re.subn(contracted_word, uncontracted_word, line)
    # return result if substitution was successful
    if count > 0:
      return s
  # no substitution was necessary
  return line

# removes all contractions
def remove_word_contractions(line):
  output_line = remove_word_contraction(line)
  # if term was removed we need to check for further duplicates
  if (line != output_line):
    return remove_word_contractions(output_line)
  # if no term removed, no contractions left
  else:
    return output_line
```

[2] Refers to dataset "`contractions.csv`", available at https://www.kaggle.com/datasets/ishivinal/contractions?resource=download.

Finally, each word within each string undergoes a **lemmatization step**. Similarly to the spelling correction, the implementation of this process is achievable using very few lines thanks to the `WordNetLemmatizer` of the NLTK library (see Listing 6).

Listing 6 Lemmatization of all words within a string.

```
lemmatizer = WordNetLemmatizer()

def lemmatize_words_in_line(lemmatizer_instance, line):
  lemmatized_words = [lemmatizer.lemmatize(word) for word in line.split
      ↪ (" ")]
  return " ".join(lemmatized_words)
```

The results of the preprocessing are saved to an intermediary CSV file "`products_preprocessed.csv`" and uploaded to the project GitHub to allow for easy usage within the upcoming steps.

3 Vectorization

After a proper preprocessing of the given `csv` file, there was still a need to clean part of the dataset. For example, there were reviews whose `Score` was not a value between 1 and 5 (see Figure 1), and there were numbers in several reviews which were counted as words, an undesirable behaviour.

SCORE
0
47
0
8
0
...
6
RN
0
0
10

Fig. 1 Dataset values with illegal `Score` values.

Once created a `Review` field composed of `Summary` + `Text`, we can apply vectorization to the input dataset. However, due to memory and time issues, only a subset of 1000 reviews could be used, otherwise, the number of features obtained per review made a dataframe big enough to occupy more than the given 4 GB. For vectorization, 4 different configurations were followed.

1. **TFIDF**: the simplest configuration, which only needs a `TfidfVectorizer` that obtains a total of 5728 different words in the input reviews.
2. **TFIDF + N-grams**: we tried to use both 2 and 3 as values for N, but ended up choosing 2 in order to avoid longer execution times for the POS tagger, and bigger csv files. In this case we obtain more than 43000 2-grams.

3. **TFIDF + N-grams + POS tagging**: several options were considered for the use of the POS tagger, such as including relative probabilities for each n-gram, but our final decision was to include the number of adjectives, since we consider that it takes into account people doing either reviews with a very low or very big score.
4. **TFIDF + N-grams + POS tagging + Other features**: in this case we included the number of words and sentences per review, following a similar philosophy as the previous example, that is to say, people angry at their purchased product will tend to write longer reviews.

Finally, after applying vectorization to each of the four previous configurations, we are ready to apply feature reduction to each of them.

4 Feature Selection

For feature reduction, we used `SelectKBest` with a *chi-square* score function to reduce the number of features by 70% and remove the less important ones. After this, we can include all this information in a dataframe to be exported and used in the following phase. One thing has to be taken into account: when using the number of adjectives, words or sentences, this column needs to be normalized in order to have better models.

This feature reduction was applied to each configuration, leaving us with a dataframe at the end. The following example corresponds to the configuration `TFIDF + N-grams + POS tagging + Other features` (see Figure 2).

	0	1	2	3	4	5	6	7	8	9	...	12901	12902	12903	12904	12905	12906	num_words	num_sentences	num_adjectives	score
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	0.044084	0.074074	0.08	5
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	0.023202	0.037037	0.04	1
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	0.097448	0.259259	0.08	4
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	0.033643	0.074074	0.02	2
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	0.017401	0.111111	0.08	5
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
995	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	0.049884	0.074074	0.10	1
996	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	0.085847	0.148148	0.10	2
997	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	0.015081	0.000000	0.08	5
998	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	0.044084	0.111111	0.04	4
999	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	0.039443	0.148148	0.08	5

1000 rows × 12911 columns

Fig. 2 Visualization of the dataset after Feature Selection.

5 Classification Algorithm

Last but not least, we had to apply machine learning algorithms to the different configurations mentioned above. Specifically, we wanted to test two algorithms for each configuration (the same algorithms will be used for the sake of comparison). For that reason, we decided to use two algorithms:

1. **SVM**: essentially, we are using it because it is both simple and effective, as it can adapt to different kinds of data (with different kernels).
2. **XGBoost**: we decided to use it because it is a machine learning algorithm that is well known for giving excellent results overall in machine learning competitions, such as the ones that are hosted in Kaggle. We wanted to see its effectiveness on NLP, which is the main reason why we decided to apply it.

An important consideration is that there is a big difference between the previous algorithms: XGBoost uses GPU to accelerate the training of the model, while SVM does not require further optimizations. Figure 3 shows the table containing the obtained results after applying the algorithms.

Configuration	Model	Precision Score	Recall Score	F1 Score
Conf.1: TFIDF	SVM default	0.733333	0.733333	0.733333
	SVM tuned	0.74	0.74	0.74
	XGBoost default	0.666667	0.666667	0.666667
	XGBoost tuned	0.673333	0.673333	0.673333
Conf.2: TFIDF with ngrams	SVM default	0.8	0.8	0.8
	SVM tuned	0.826667	0.826667	0.826667
	XGBoost default	0.66	0.66	0.66
	XGBoost tuned	0.67	0.67	0.67
Conf.3: TFIDF with ngrams and pos tagging	SVM default	0.8	0.8	0.8
	SVM tuned	0.83	0.83	0.83
	XGBoost default	0.67	0.67	0.67
	XGBoost tuned	0.686667	0.686667	0.686667
Conf.4: TFIDF with ngrams, pos tagging and extra features	SVM default	0.806667	0.806667	0.806667
	SVM tuned	0.826667	0.826667	0.826667
	XGBoost default	0.663333	0.663333	0.663333
	XGBoost tuned	0.673333	0.673333	0.673333

Fig. 3 Obtained results using different configurations.

There are multiple conclusions that can be drawn from it.

1. SVM looks generally better for this specific task when compared to XGBoost, as all the SVM models are better than the XGBoost models (even when they are tuned).
2. Models will rarely predict label 2, most likely due to the imbalance in the data (it is really unlikely that someone will rate a product with 2 stars, probably 1 or 5 are the most common).
3. A two-phase classification could be used for further improving the results (probably it would not improve too much, but it could improve the results slightly). The procedure would be as follows.
   A model would decide whether it is 5 rating or not (binary classification), for those predicted as no-5 rating, another model would predict which rating they should have. This kind of procedure (that we developed in the second project of this course) generally slightly improves the results when we have unbalanced data, as it is the case.

Additionally, the results presented in figure 3 show how there are 2 models that achieve the highest scores.

1. SVM model on configuration 2 (tuned).
2. SVM model on configuration 4 (tuned).

If we take a look at the confusion matrices, which can be found in the Google Colab, they are pretty similar (overall, they are really good, but both models tend to never predict labels 2 or 3, which can be understandable due to the unbalanced nature of the data). Because of that reason, we conclude that the best model is SVM on configuration 2 (tuned), because it is simpler and it gives the same results (there is not a real difference between them), so it maximizes the performance while lowering the effort in applying NLP techniques (only requires vectorization and ngrams).