M906 Programming (Summer Semester 2023)

Final Project: Aspect-Based Sentiment Analysis

Report

Kleopatra Karapanagiotou (It12200010)
Anastasia Karakitsou (It12200009)

Both team members contributed equally to this project.

Introduction

TASK DESCRIPTION

Aspect Based Sentiment Analysis (ABSA) is the task of mining and summarising opinions from text about specific entities (e.g. a restaurant). Typically, an ABSA system attempts to detect the main aspects that are discussed and estimate the polarity for each of them. In this setting one approach is to define an inventory (a closed set) of aspects for a domain and then build a system that can identify these aspects in text and decide about polarity. For the restaurant domain an aspect inventory has been defined in the context of SemEval conferences; e.g., the inventory contains FOOD#QUALITY, RESTAURANT#GENERAL, RESTAURANT#PRICES, etc. The goal of this assignment is to build a system that decides the polarity of a given aspect.

STRUCTURE

The attached .zip file contains:

1. A ‘readme.txt’ file that contains the instructions and order to run the code, along with description of each module.
2. A ‘src’ folder contains all modules required for this specific assignment (split.py, train.py, test.py,experiments.py, main.py, utils.py)
3. A ‘dataset’ folder, containing the .xml file used for this assignment
4. An ‘output_texts’ folder with all the .txt files of the results of each experiment.
5. A ‘requirements.txt’ files containing the libraries used.

Data

DATASET CREATION

The initial dataset and the split parts (created by running the module ‘split.py’) were in .xml format. In order to convert the parts into training and test samples, we merged useful information from each sentence (‘text’) tag along with each of the sentences’ opinions and polarity (Text, Category, Polarity).

CLEANING

We assume each aspect along with its sub-aspects as separate categories.

Issues we noticed with the data frame were the following:

1. Reference to two different sub-aspects in the same sentence were kept:

3	1004293	1004293:3	The food was lousy - too sweet or too salty and the portions tiny.	FOOD#QUALITY	negative
4	1004293	1004293:3	The food was lousy - too sweet or too salty and the portions tiny.	FOOD#STYLE_OPTIONS	negative

2. Duplicates with same text and same aspect have been removed:

9	1014458	1014458:2	The duck confit is always amazing and the foie gras terrine with figs was out of this world.	FOOD#QUALITY	positive
10	1014458	1014458:2	The duck confit is always amazing and the foie gras terrine with figs was out of this world.	FOOD#QUALITY	positive

3. NaN values at the column “Category” denote sentences that contain no opinion - these have been removed:

79	1074868	1074868:0	I went to this restaurant with a woman that I met recently.	NaN	NaN
80	1074868	1074868:1	She lives nearby but had never gone to this establishment thinking that it might be too touristy.	NaN	NaN

After cleaning, the remaining rows/sentences were reduced to 2300 from the initial 2799.

PREPROCESSING TEXT

- Since our task is Aspect-Based Sentiment Analysis, we wanted to incorporate the information of the “Category” column in the train and test set. That’s why we merged the columns [‘Text’] and [‘Category’], so that the vectors generated for each sentence contain the related ‘Aspect’ information as well.
- Tokenization: We used the tok-tok tokenizer from nltk library (https://www.nltk.org/_modules/nltk/tokenize/toktok.html) which is a simple, general tokenizer, where the input has one sentence per line; thus only the final period is tokenized.
- We decided to remove all stopwords of the main text of the column “Text”, as they do not contribute to the task of sentiment detection.
- We removed the html characters, the punctuation, and the special characters (excluding the ‘$’ symbol, because we consider this important in feature extraction (e.g ‘We paid $500 for a meal! That’s unbelievable!’).
- We applied stemming with Porter Stemmer in the text column.
- Considering that after removing stopwords, the token length of each sentence review was relatively short, we decided to use unigrams and bigrams as features.

VECTORIZING

There are many ways to represent words. In our task, we employed various vectorization techniques to represent words in different ways, each offering unique advantages.

We first used a **Count vectorizer.** The count vectorizer is used to convert a collection of text documents into a ‘bag of words’ (BOW), and then into a matrix of token counts. It provides a simple way to both tokenize a collection of text documents and build a vocabulary of known words. However, this type of vectorizer has some drawbacks. The dimensionality of count vectorizer output can be very large, especially when dealing with large corpora, because it creates a dimension for each unique word in the text. This can result in very high-dimensional data, which can be computationally intensive.. Also, the count vectorizer’s representation is not

as informative as other vectorizers, as it merely counts the frequency of each word's appearance in the text, without taking into account their importance.

Word importance is taken into account, however, via the **TF-IDF** vectorizer, which we used as our second vectorizer. This vectorizer not only considers the frequency of words in a single document (cf. TF = Term Frequency), but also the frequency of words across all documents in the corpus (cf. IDF = Inverse Document Frequency). By doing so, the TF-IDF vectorizer assigns more weight to words that are relatively unique to a specific document, enhancing their informativeness.

In addition to these methods, our third choice for word representation was the more advanced word embeddings, specifically the **Word2Vec** embeddings. Considering the relatively small size of our dataset, we opted for a smaller vector size, setting it to **100.** Word2Vec embeddings capture semantic relationships between words by representing them as dense vectors in a continuous space. This allows for a more nuanced understanding of word similarities and contextual relationships.

By leveraging a combination of the simpler choice of count vectorization, the richer TF-IDF vectors, and the semantically-oriented (average) Word2Vec embeddings, we aimed to capture different aspects of word representation.

Implementation

CLASSIFICATION ALGORITHMS
The classification algorithms used were the following:

- Logistic Regression (class_weight='balanced')
- Random Forest (class_weight='balanced')
- Support Vector Machines (class_weight='balanced')
- Multinomial Naive Bayes

Experiments

FEATURES
N-grams were extracted during vectorisation. We tested unigrams and bigrams with both vectorisers.

FEATURE REDUCTION
After conducting the initial experiments and finding the best classifier, features and vectoriser for our data (Logistic Regression, TF-IDF, unigrams), we experimented with max features=1000. Since the original average amount of unigram features per .xml part were approximately 2200, we decided to reduce the max features in half.

METRICS

Due to data imbalance, we considered accuracy as a non-representative metric, so we chose the F1 score of each class, along with the macro average F1 score for evaluation. Furthermore, F1 score provides a single score that balances both the concerns of precision and recall in one number.

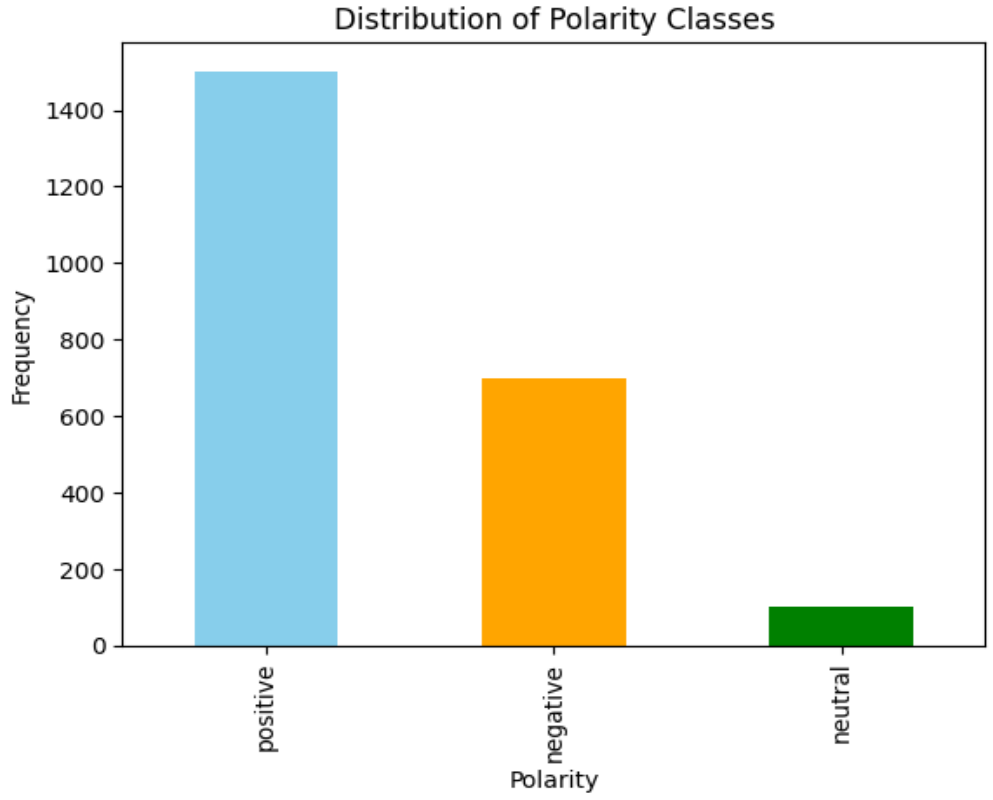

Figure 1: Data imbalance among the classes

Results-F1 (10-fold cross validation)

Below, the scores of the 10 Fold Cross Validation (experiments.py) are shown in the tables.

Logistic Regression

Features	F1 (avg)	F1 (neutral)	F1(positive)	F1 (negative)
All features, unigrams, countvectorizer (BOW)	75.42%	24.60%	82.65%	60.08%
All features, unigrams, tf-idf vectorizer	**76.19%**	**29.45%**	**82.60%**	**62.41%**
All features, bigrams, countvectorizer	69.96%	5.33%	80.16%	49.04%
All features, bigrams, tf-idf vectorizer	69.23%	16.95%	78.14%	49.54%
All features, Word2Vec	57.13%	10.46%	66.20%	35.70%
Max features=1000, unigrams, tf-idf vectorizer	74.64%	25.94%	81.16%	61.12%

Random Forest

Features	F1 (avg)	F1 (neutral)	F1(positive)	F1 (negative)
All features, unigrams, countvectorizer (BOW)	70.58%	4.04%	83.30%	45.10%
All features, unigrams, tf-idf vectorizer	70.62%	0.00%	82.81%	46.23%
All features, bigrams, countvectorizer (BOW)	62.16%	0.00%	80.73%	22.92%
All features, bigrams, tf-idf vectorizer	64.15%	0.00%	79.90%	32.94%
All features, Word2Vec	58.90%	0.00%	79.94%	11.56%
Max features=1000, unigrams, tf-idf vectorizer	**71.45%**	**0.00%**	**83.16%**	**47.98%**

SVM

Features	F1 (avg)	F1 (neutral)	F1(positive)	F1 (negative)
All features, unigrams, countvectorizer (BOW)	75.16%	23.18%	82.43%	59.79%
All features, unigrams, tf-idf vectorizer	**76.93%**	**17.59%**	**84.44%**	**61.99%**
All features, bigrams, countvectorizer (BOW)	67.51%	4.90%	79.38%	41.41%
All features, bigrams, tf-idf vectorizer	69.89%	2.14%	81.06%	47.35%
All features, Word2Vec	50.77%	1.47%	55.74%	38.07%
Max features=1000, unigrams, tf-idf vectorizer	76.80%	14.79%	84.10%	62.82%

Multinomial Naive Bayes

Features	F1 (avg)	F1 (neutral)	F1(positive)	F1 (negative)
All features, unigrams, countvectorizer (BOW)	**73.36%**	**0.00%**	**83.76%**	**53.18%**
All features, unigrams, tf-idf vectorizer	66.06%	0.00%	82.83%	30.86%
All features, bigrams, countvectorizer (BOW)	67.95%	0.00%	82.19%	39.50%
All features, bigrams, tf-idf vectorizer	60.26%	0.00%	81.32%	14.31%
All features, Word2Vec *	–	–	–	–
Max features=1000, unigrams, tf-idf vectorizer	69.53%	0.00%	83.28%	41.72%

* Multinomial Naive Bayes model has a limitation, as it is not compatible with the Word2Vec embeddings, because these embeddings take negative values. This is the error raised: *ValueError: Negative values in data passed to MultinomialNB (input X)*

Conclusions and Further Thoughts

From the above experiment settings several observations were made:

1. Unigrams tend to produce higher F1 scores in both vectorizers tested, compared to bigrams. Furthermore, during implementation it was observed that bigrams demanded additional computational complexity in processing pairs of consecutive words and that impacted the model's performance.So we can assume that bigrams do not contribute to our classification task, but rather add noise to it.
2. The classifiers that gave us comparatively better results were **SVM** and **Logistic Regression** with **unigrams** as extracted features, and **TF-IDF** vectorizer. For SVM we obtained an average F1 of **76.93%**, and for **Logistic Regression** an average F1 of **76.19%. However, Logistic Regression showed higher F1 scores for the three polarity classes when taken into account individually (neutral: 29.45%, positive: 82.60% and negative: 62.41%)**, which is important, considering the class imbalance of the dataset.
3. Averaging word2vec embeddings was a practice that placed a serious burden on the classifier's performance, since word order was not taken into consideration. It would be thus interesting to experiment with other sentence representations like **sBERT**, which derive semantically meaningful sentence embeddings that can be compared using cosine-similarity, as mentioned in the following StackExchange post: machine learning - What does average of word2vec vector mean? - Cross Validated (stackexchange.com)
4. The results of feature reduction in half did not indicate significant improvement in any of the classifiers tested. Small reduction or increase were observed, which suggests that approximately half of the features (unigrams) do not substantially contribute to our classification task.