Project 1 – PML

Stan Eduard-George (407)

The first project for Practical Machine Learning consisted of a Kaggle challenge of classification. The classification needed to be done over a set of two Romanian sentences. The evaluation method is the macro F1 score.

Data preprocessing

Data preprocessing is the same for both methods implemented, so I will start with it. First, I chose to store data in Pandas data frames, finding it a versatile way of representing and manipulating features. After storing the features and labels into data frames, I created a new feature representing the combined sentences, separated with a blank space. The next step was to separate the **X** and **y** variables as follows:

```
X = df['combined']

y = df['label']
```

Now, it is clear that the only feature I will work with is the one with combined sentences. Afterwards, I applied CountVectorizer, a way of counting all the occurrences of a certain word in a specific sentence. I also changed some of the hyperparameters so that the code would be more memory and time efficient. Here is the line of code that I am going to explain:

```
vectorizer = CountVectorizer(token_pattern="\w+", min_df=2, max_df=0.35, ngram_range=(1,2))
```

* token_pattern="\w+" - changed from the default r"(?u)\b\w\w+\b". I have changed this parameter because the default one includes 416 other symbols, single letters, and digits. Some of them are displayed below.

'ë', 'ń', 'ø', 'θ', 'B', 'v', ' ⌋ ', 'ρ', 'ь', 'Ă', ',', '橋', 'ч', 'ñ', ',', 'ट 'è', ' ・ ', 'y', '刀', '妃', '商', 'な', ',', '' 'º', '된', 'さ', ',', 'ש '發', '¼', '²', 'Ω', '밟', 'ʒ', 'ī', '(ϖ)', 'の', '2', 'π', 'b', 'ス', 'ç', 't', '震', 'ѡ', '德', 'ŋ', 'ズ', 'д', 'β', 'χ', 'ज', 'ा'

* min_df=2 - the vectorizer considers only the instances (one word or a group of words) that appear more than two times in the whole list of sentences. (this action truncates the BoW by around 77%, from 1480284 instances to 347826).

* max_df=0.35 - the vectorizer removes the instances that appear in more than 35% of the sentences. (It only reduces 8 instances in total, but it's always good to remove these misleading occurrences. This parameter also gives the best results).

* ngram_range=(1,2) - the vectorizer counts single words and groups of two words.

A detailed study on how to choose the optimal ngram_range was realized using Logistic regression:

ngrams = (1,1)

	max_features	recall	precision	f1 score
0	1000	0.353907	0.422829	0.353061
1	1500	0.378696	0.480780	0.389003
2	2000	0.394235	0.498808	0.407762
3	2500	0.386661	0.464475	0.395956
4	3000	0.395621	0.472302	0.408615
5	3500	0.391389	0.450991	0.402868
6	4000	0.395184	0.458481	0.408238
7	4500	0.398516	0.459608	0.410061
8	5000	0.403117	0.452271	0.413827

ngrams = (1,2)

	max_features	recall	precision	f1 score
0	1000	0.349225	0.430400	0.349119
1	1500	0.376029	0.486845	0.386146
2	2000	0.374178	0.435928	0.380729
3	2500	0.386895	0.438375	0.394220
4	3000	0.393723	0.445310	0.402578
5	3500	0.388774	0.432330	0.396576
6	4000	0.402765	0.439050	0.409822
7	4500	0.397930	0.444518	0.407704
8	5000	0.400460	0.439486	0.408899

ngrams = (1,3)

	max_features	recall	precision	f1 score
0	1000	0.350289	0.407213	0.349299
1	1500	0.372852	0.476816	0.381397
2	2000	0.382654	0.454444	0.392289
3	2500	0.382856	0.432587	0.389588

4	3000	0.397387	0.454157	0.406531
5	3500	0.389065	0.435522	0.397006
6	4000	0.406177	0.456683	0.417564
7	4500	0.408248	0.447513	0.416734
8	5000	0.404490	0.440648	0.412214

ngrams = (1,1)

	max_features	recall	precision	f1 score
0	20000	0.374433	0.427330	0.386291
1	40000	0.374199	0.435340	0.387950
2	60000	0.367911	0.434070	0.380485
3	80000	0.372217	0.438371	0.384420
4	100000	0.368525	0.436945	0.380082
5	120000	0.368586	0.436806	0.380162
6	140000	0.370287	0.438360	0.381561
7	160000	0.373155	0.440839	0.384364

ngrams = (1,2)

	max_features	recall	precision	f1 score
0	20000	0.401002	0.455549	0.413690
1	40000	0.397924	0.444217	0.410483
2	60000	0.396964	0.471114	0.414180
3	80000	0.396332	0.463063	0.411778
4	100000	0.402213	0.479819	0.419759
5	120000	0.404209	0.495292	0.423732
6	140000	0.400250	0.476251	0.415782
7	160000	0.401277	0.483351	0.418014

ngrams = (1,3)

	max_features	recall	precision	f1 score
0	20000	0.400787	0.448780	0.411675
1	40000	0.401102	0.465694	0.417268

2	60000	0.390821	0.438916	0.402186
3	80000	0.399752	0.474545	0.416355
4	100000	0.397631	0.475447	0.414111
5	120000	0.407106	0.479699	0.423702
6	140000	0.400883	0.481597	0.417393
7	160000	0.400733	0.475716	0.416247

Then, I fit the vectorizer with the sentences in 'X' and assign the transformed ‘X’ to 'X_vectorized'.

For memory reduction, I chose to redistribute the memory for the data stored in X_vectorizer, and y from int64 to int16, respectively int8. The reason I didn't use 'int8' for X_vectorizer is because the maximum number of occurrences of a word in a sentence is ... 502.

Method 1

The first model I chose to implement was logistic regression. It turned out that it gives good predictions for this dataset. The only hyperparameter changed is the maximum number of iterations so that the model could converge (max_iter=1000). There’s no significant difference in the f1 score if the max_iter parameter is lower:

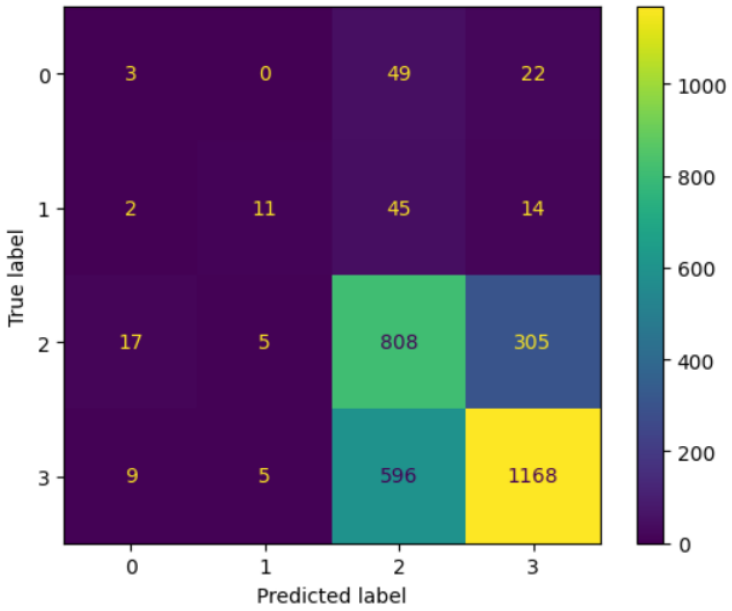
Max_iter	f1-score
300	0.40
500	0.40
1000	0.40

Next, I vectorized the test set on the vectorizer fitted before, and then I predicted the labels. Lastly, I exported the predicted data + guid to a csv file.

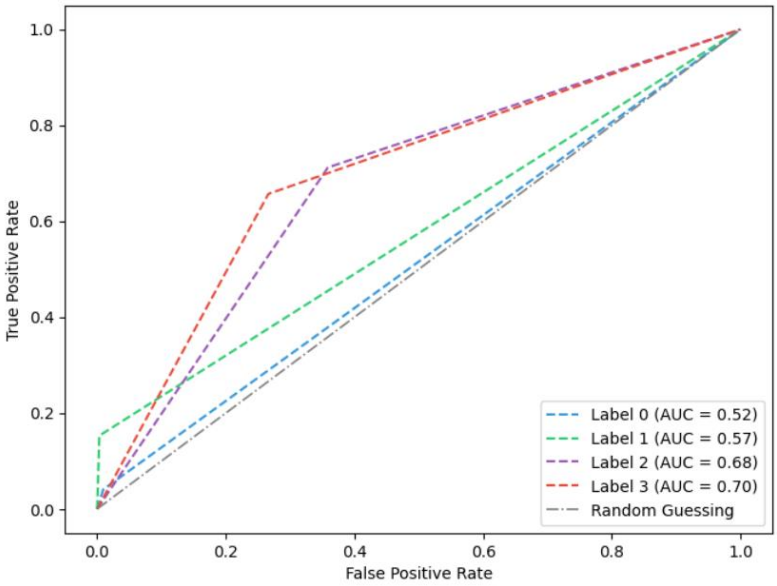
Classification report on the given validation set – Method 1

	precision	recall	f1-score	support
0	0.10	0.04	0.06	74
1	0.52	0.15	0.24	72
2	0.54	0.71	0.61	1135
3	0.77	0.66	0.71	1778
accuracy			0.65	3059
macro avg	0.48	0.39	0.40	3059
weighted avg	0.66	0.65	0.65	3059

Confusion matrix – Method 1



ROC curve – Method 1



Method 2

For the second method, there's a little bit more data preprocessing. To be more precise, I used SMOTE to oversample data. The reason I chose to use SMOTE is because the data is imbalanced. Here is the number of sentences associated with their label 0:2666, 1:1372, 2:26857, 3:30278. Clearly, from here we can see that labels 0 and 1 are more than 10 times fewer than the two others. So, I decided to increase the number of 0 and 1 labeled sentences by 0.3 of the length of the majority.

The model chosen for this second method is linear support vector machine (LinearSVC). The reason I didn't use the usual SVC is because it took too much time to compute.

LinearSVC(max_iter=300, C=0.1, dual='auto', random_state=42)

The parameters were chosen empirically with the help of the following study:

0: max_iter=300

1: max_iter=500

2: max_iter=1000

3: max_iter=300

And so on...

F1 scores for C (LinearSVC)

	C	f1 score	f1 score SMOTE 03	f1 score SMOTE 05	f1 score SMOTE 08
0	0.1	0.239062	0.410665	0.411497	0.406122
1	0.1	0.239062	0.410665	0.411497	0.406122
2	0.1	0.239147	0.410665	0.411497	0.406122
3	1.0	0.238437	0.398708	0.400484	0.393146
4	1.0	0.238364	0.398803	0.400351	0.393020
5	1.0	0.238608	0.398751	0.400331	0.393247
6	10.0	0.237798	0.390940	0.391226	0.383718
7	10.0	0.237242	0.391125	0.389882	0.384207
8	10.0	0.237558	0.392139	0.387138	0.383749
9	100.0	0.238594	0.392698	0.387623	0.384519
10	100.0	0.238353	0.382285	0.386067	0.383433
11	100.0	0.237794	0.383623	0.385774	0.384248
12	1000.0	0.236673	0.386959	0.383845	0.378507
13	1000.0	0.237531	0.377160	0.383731	0.380405
14	1000.0	0.238510	0.373649	0.383875	0.378231

The study revealed that using SMOTE made a significant difference for LinearSVC model. Therefore, for other models it didn't make any improvement. This table also shows the F1 scores for the methods that use SMOTE for 0.3, 0.5, and 0.8 increase of 0 and 1 labeled sentences.

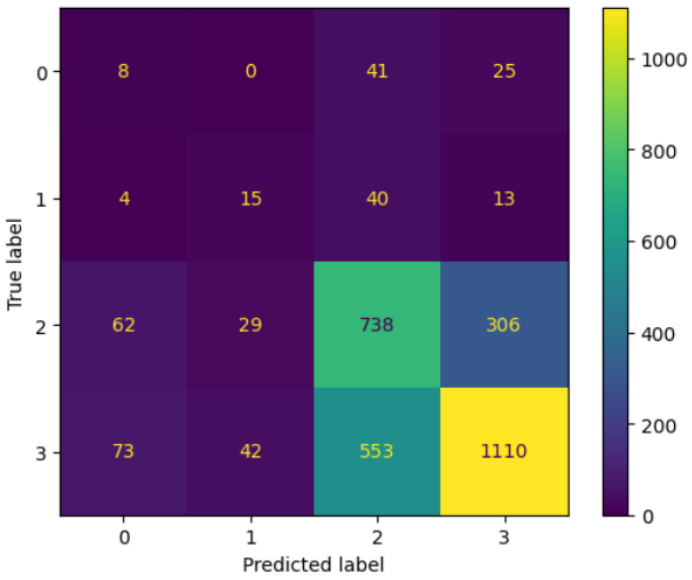
The reason I chose max_iter=300 for the final model instead of 500 or 1000 was because, even though the model didn’t converge, the results for C=0.1 were the same. A smaller maximum number of iterations means faster code.

Next, I vectorized the test set on the vectorizer fitted before, and then I predicted the labels. Lastly, I exported the predicted data + guid to a csv file.

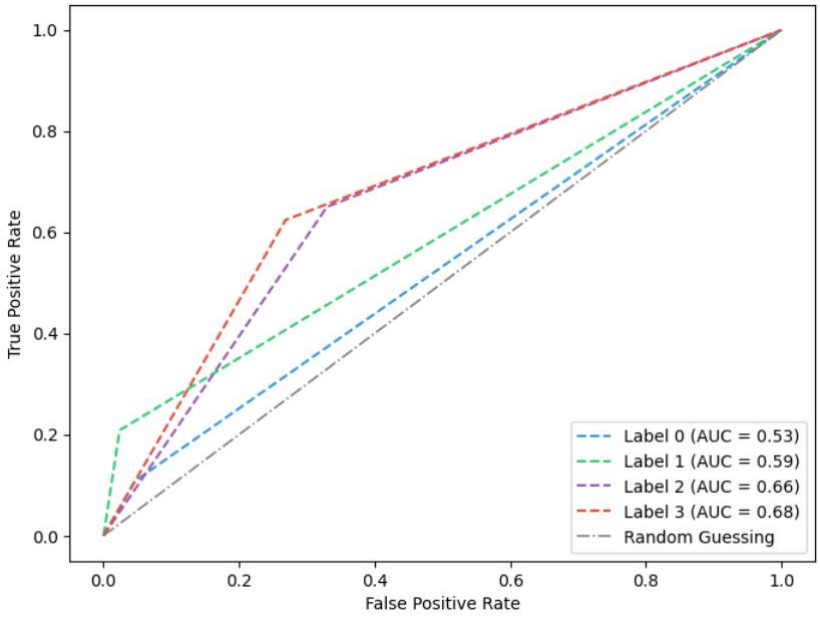
Classification report on the given validation set – Method 2

	precision	recall	f1-score	support
0	0.05	0.11	0.07	74
1	0.17	0.21	0.19	72
2	0.54	0.65	0.59	1135
3	0.76	0.62	0.69	1778
accuracy			0.61	3059
macro avg	0.38	0.40	0.38	3059
weighted avg	0.65	0.61	0.62	3059

Confusion matrix – Method 2



ROC curves – Method 2



Conclusion

As we can see from the confusion matrices and the ROC curves, the models perform similarly over the validation set. There is no surprise, since data preprocessing is done with CountVectorizer in both cases. However, using SMOTE made a small difference regarding the classification of 0 and 1 labels, with the price of misclassifying the other two labels. The ROC curves reveal that the AUC scores are 0.52, 0.57, 0.68, 0.70 for method 1, and 0.54, 0.59, 0.66, 0.68 for method 2. Overall, the results are not the best, and could be improved with the right approach.

*Even though the results are as they are, this project taught me many essential skills that will be useful in my career. I'll wait for your feedback on what I could have done better. Thank you!

Failed attempts

Before implementing the two models presented above, I've had plenty of failed tries to achieve a high F1 score. Let me list some of them:

- Number of punctuation signs for sentence1, sentence2 and the combined sentences;

- Length of each sentence in sentence1, sentence2 and the combined sentences;
- A word relevance score calculated with the help of a TfidfVectorizer. The vectorizer outputs the relative scores for each instance (one word or a sequence of two words) in a sentence. What I did here was to compute the mean of these scores and assign it to each sentence in all three features mentioned above;
- Jaccard similarity score that was computed in the following way: create a set for sentence1 and sentence2 features. The actual score will be represented by the division between the lengths of the intersection and the union between the sets;
- Cosine similarity score, computed over the outputs of CountVectorizer applied on sentence1 and sentence2;
- Length similarity which means length of sentence1 over length of sntence2, and vice-versa;
- Topic distributions, where I've computed the predominant topic in each of the combined sentences. For this, I used topic modeling with Latent Dirichlet Allocation (LDA). The actual values added to the feature were the most related topics of the combined sentences.
- Topic similarity score. It should calculate the similarity between the topics in sentence1 and sentence2, but it doesn't. This feature could be improved.

Heatmap for the unused features