Documentation for Project 1
Practical Machine Learning

Sîrbu Oana Adriana

Artificial Intelligence / 407

1. Description of my machine learning approaches

To access the training, validation and test data downloaded from the Kaggle platform I used Pandas library. Three different variables store the data of interest, each containing 4 columns: 'sentence1', 'sentence2', 'label', and 'guid'. My strategy involved merging text information from both 'sentence1' and 'sentence2' columns into a single column named 'combined_text', which will be used in subsequent steps. Example for the training data (for the validation and test data, same preprocessing steps were used):

```
train_data['combined_text'] = train_data['sentence1'] + " " + train_data['sentence2']
```

1.1 Chosen features set and data preprocessing

For this project, we were tasked with training diverse classifier models on a dataset containing Romanian sentences. Given that our data is in text format and recognizing the necessity to employ numerical input for training models—essential for processing, fitting, and making predictions—I made the decision to use a Bag-of-Words (BoW) feature in order to achieve the final goal.

The BoW feature extraction method enables the conversion of a text structure into its corresponding vector of numbers. For this particular task, I experimented with two existing BoW implementations: Count Vectorizer and TF-IDF Vectorizer. These implementations can be accesed by importing their classes from 'sklearn.feature_extraction.text' module. Given that with Count Vectorizer I achieved better results during my validation and testing phases, I will proceed to describe this approach and the tuning of its parameters.

Count Vectorizer essentially takes a set of text documents and turns it into a numerical format by keeping track of how often each word appears in the text. The final result is a matrix that shows the distribution of words across the given documents. In particular, in the context of our project, Count Vectorizer took into account all the sentences across the four labels and created a vocabulary containing each unique word in the entire available text. Additionally, it produced vectors highlighting the occurrences of each word from the vocabulary in all pairs of sentences within the training dataset.

Regarding the tuning of its parameters, I only focused on some of them, that I considered to be more important for my purposes: preprocessor, stop_words, n_gram range, max_df and min_df.

To handle the preprocessor, I created a custom function designed to 'clean' the input text documents, that would be given as argument: clean_text(text). The functionality of this function consists in removing

punctuation symbols (in the first line), special characters like lower Romanian guillemets (2nd line of the function), and convert all text to lower case (3rd line). This highly improved my results.

```
def clean_text(text):
    text_no_punct = ''.join(ch for ch in text if ch not in string.punctuation)
    text_no_special_char = text_no_punct.replace("„", "")
    cleaned_text = text_no_special_char.lower()
    return cleaned_text
```

Although it may have been used in the definition of my vectorizer, I applied it independently on the merged columns to facilitate debugging and easily observe the results.

```
train_data['clean_combined_text'] = train_data['combined_text'].apply(clean_text)
```

Therefore, to fit my Count Vectorizer I used the information contained in the 'clean_combined_text' columns. Next, various results illustrating the performance of the vectorizer for different values of its parameters are shown, justifying the final choice made for the best Kaggle submission.

Table 1. Parameters of Count Vectorizer Initialisation

Macro F1 score	**stop_words**	**ngram_range**	**max_df**	**min_df**
0.39	-	(1,1)	1	1
0.42	-	(1,2)	1	1
0.38	romanian_stop_words	(1,1)	1	1
0.39	romanian_stop_words	(1,2)	1	1
0.39	-	(1,1)	1	2
0.39	-	(1,1)	0.8	2
0.39	-	(1,1)	0.45	2
0.41	-	(1,2)	0.8	2
0.41	-	(1,2)	0.45	2
0.41	-	(1,2)	0.6	2
0.42	-	(1,2)	1	2

In the romanian_stop_words variable, as it name suggests, are stored common Romanian words that can be redundant for a sentence's meaning. I concluded that manually removing such words does not help much, but rather one should use the min_df and max_df arguments. By setting min_df argument to 2, one does not take into account words that appear only one across all the documents (I think of them as noise, as it may be only a small chance for that words being in the test set). Moreover, using this value reduced the computation time for predictions by four. This was part of the final version of Count Vectorizer. One can use a 0.8 value for

max_df (that I consider to remove the so called 'stop words' from the text, the ones that appear in more than 80% of the documents) to further improve the results.

N-grams are sequences of n adjacent words created from the entire text document. By setting the ngrams_range to (1,1) which is also the default value for this argument, one evaluates each word individually by placing it in the vocabulary. A value of (1,2) for this argument, in addition to single words, uses also bigrams (combinations of 2 neighbouring words). This improved the final results, and consequently, I used the (1,2) value for my final submission.

The above Macro F1 scores were determined by training one Logistic Regression model on the features obtained with this Count Vectorizer. More details about the implementation of such a model are found in the next subsection.

For the data preprocessing step, tests also included scaling the features in order to facilitate the faster converge of the optimization algorithms. For this purpose, a MaxAbsScaler() instance from the sklearn.preprocessing module was used. The snippet of code that implements this scaling is the following:

```
BoW_features_scaler = MaxAbsScaler()
BoW_vectorizer = CountVectorizer(ngram_range=(1,2), min_df=2)
train_vocab = train_data['clean_combined_text']
X_train_BoW = BoW_vectorizer.fit_transform(train_vocab)
X_train_BoW_scaled = BoW_features_scaler.fit_transform(X_train_BoW)
```

Apparently, after the scaling the evaluation scores went down a bit, therefore it was not used for the final training (it will appear in the code for documentation purposes).

1.2 Trained models and their performance

Different models were trained based on the previous features extraction and the other preprocessing steps.

1.2.1 Logistic Regression

Logistic Regression is a supervised machine learning algorithm that is mainly used for classification problems, both binary and multi-class ones. One can benefit of such flexibility by importing this model from the sklearn.linear_model module, which offers a multi_class argument that can be adjusted as needed. As the official documentation of scikit-learn presents, the default value for multi_class in 'auto', suggesting that the model automatically recognizes the task it faces and selects a proper model for training. If it detects that the task implies multi-class classification (as it the case in this task), the multi_class takes the value 'multinomial'. Results using the 'ovr' value (that stands for One-vs-Rest) are not presented (although this approach was also tested), as the evaluation scores were slightly lower. Behind the scenes, the 'multinomial' option uses a Softmax function in order to determine the probability of an instance to belong to each existing classes (4 in our case) and chooses the most likely output.

Below, one can see how an instance of this model looks like by manually selecting the 'multinomial' approach.

```
logisticr_model = LogisticRegression(max_iter=1000, random_state=1,
                                     multi_class='multinomial')
```

A Grid Search was performed in order to identify the best hyperparameters of a Logistic Regression model for this task. In the following table are some of the results obtained by using distinct values for the C,

penalty, solver and max_iter arguments. The second set of hyperparameters from the table proved to be the most effective, therefore they were selected to form the final configuration of the model.

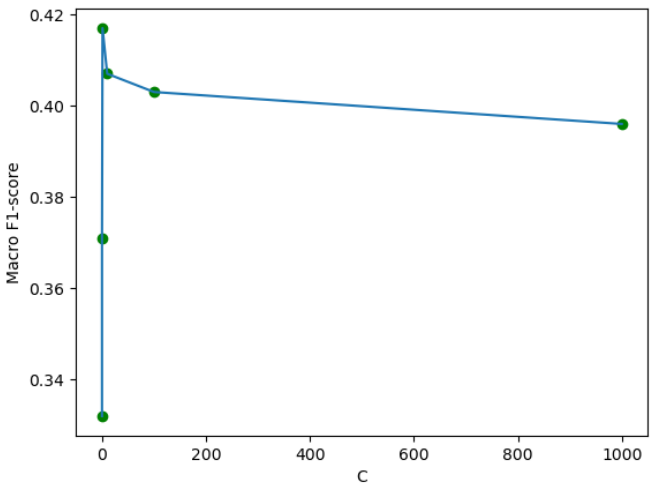
Table 2. Hyperparameters tuning for Logistic Regression

Macro F1 score	**C**	**penalty**	**solver**	**max_iter**
0.42	1	l2	lbfgs	500
0.42	1	l2	lbfgs	1000
0.33	1e-3	l2	lbfgs	1000
0.40	100	l2	lbfgs	1000
0.39	1	l2	sag	1000
0.38	1	l2	saga	1000

In the graph obtained by using the data from the below table, one can see how the regularization parameter C influences the macro F1-score, keeping the penalty='l2', solver='lbfgs' and max_iter=1000. A special attention is offered to this argument as it proved to highly affect the results.

Table 3. Macro F1 score as a function of C

C	1e-3	0.01	1	10	100	1000
Macro F1-score	0.332	0.371	0.417	0.407	0.403	0.396



Graph 1. Macro F1 score as a function of C

The classification report obtained by comparing the true validation labels with the ones predicted by the Logistic Regression model having the above best configuration (C=1) is the following:

Table 4. Classification report for the Logistic Regression model on validation data

	precision	**recall**	**f1-score**	**support**
0	0.11	0.04	0.06	74
1	0.58	0.19	0.29	72
2	0.54	0.71	0.61	1135
3	0.77	0.65	0.71	1178
accuracy			0.65	3059
macro avg	0.50	0.40	0.42	3059
weighted avg	0.67	0.65	0.65	3059

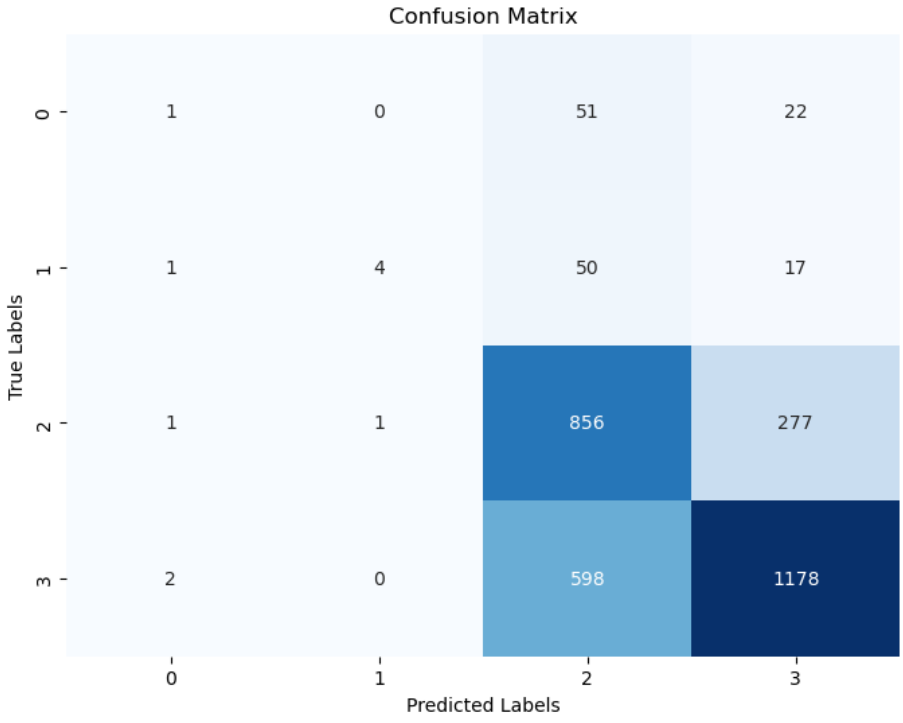


Fig 1. Confusion matrix for the Logistic Regression model on validation data

As we analyse both the classification report and the confusion matrix, we can observe that the model behaves poorly for 0 & 1 labels classification, due to the imbalance of the training dataset. Strategies to improve

these results are presented in the next section. Macro F1-score is the best scoring technique in this case, as it forces all the classes to contribute equally to the overall performance of the model (opposed to the accuracy scores).

1.2.2 Linear Support Vector Classification (LinearSVC)

SVCs are another type of supervised machine learning models that are widely used for classification tasks. For this project, as the second method, the LinearSVC model from the scikit-learn library was adopted. In addition to Logistic Regression, this model showed the best performance on the extracted features, therefore it receives here a special description. It is esentially the classic SVC which has a linear kernel, but it is characterized by liblinear's library optimization algorithm (compared to the default libsvm's one), as the official documentation states. In practice was observed that this model performed way faster than the original SVC. This is a consequence of the fact that it is particularly conceived to handle large number of samples more efficiently, as it is the case with all the text documents available in the training dataset.

Below it is illustrated a simple instance of this model, imported from sklearn.svm module.

```
linear_svc_model = LinearSVC(C=1, max_iter=1000, penalty='l2')
```

The C=1, max_iter=1000 and penalty='l2' are the default parameters of this model, but they are manually added here as an example. Of course, to increase the evaluation score, this parameters were tuned and the best version of this model was used for the second final submission. In the table below one can observe how varying this parameters affected the Macro F1-score. The penalty was always considered 'l2' and the max_iter was kept the same (the default value), because a bigger number of iterations would cause overfitting.

Table 5. Hyperparameters of LinearSVC

Macro F1-score	**C**	**class weights**
0.353	1e-3	no
0.386	0.01	no
0.396	0.1	no
0.396	1	no
0.395	10	no
0.387	1000	no
0.445	1e-3	yes
0.414	0.01	yes
0.396	1	yes

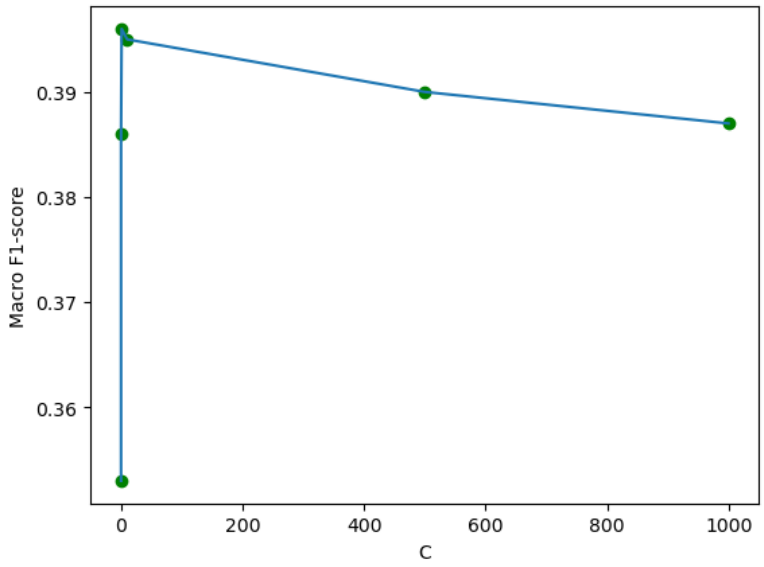
As 'C' is known to be the regularization parameter, the meaning of 'class weights' will be presented. Due to the fact that the task implies handling an an imbalanced dataset, the possibility to force the model to be more sensitive with data associated with the labels 0 and 1 is not to be neglected. Therefore, to teach the model the class distribution and to indicate which are the minority classes, one can compute their corresponding weights and pass them as arguments. These weightes were determined using the build-in 'compute_class_weight' method from 'sklearn.utils.class_weight'. The result is the following: weight for class 0 is 5.736, weight for class 1 is 11.146, weight for class 2 is 0.569, weight for class 3 is 0.505. It can be observed that the minority classes have higher weights. This implies that the model will offer them more importance, increasing the chance to perform better on them.

Below is the syntax for defining a LinearSVC model with the 'class_weight' parameter, assigning a dictionary where the label number serves as the key and the corresponding weight as its value.

```
linear_svc_model = LinearSVC(class_weight=dict(zip(np.unique(y), class_weights)))
```

Unfortunately, even though the results from Table 5 indicate a better performance when the 'class_weights' parameter is used, by running many different tests (including a special submission on Kaggle), it was concluded that the higher F1-scores associated with those implementations are caused by overfitting. The final parameters for this model are: C=1, penalty='l2', no class_weight and max_iter=1000.

For a better visualization, a graph displaying how the 'C' parameter influences the F1-score is provided.



Graph 2. Macro F1 score as a function of C

Moreover, the classification report obtained by making predictions using the LinearSVC model with the presented parameters can be analysedin Table 6.

A confusion matrix is also illustrated in Fig. 2, emphasizing the same poor performance on the labels 0 and 1 as seen earlier for the Logistic Regression model.

Table 6. Classification report for the LinearSVC model on validation data

	precision	recall	f1-score	support
0	0.10	0.05	0.07	74
1	0.39	0.17	0.23	72
2	0.52	0.68	0.59	1135
3	0.75	0.64	0.69	1178
accuracy			0.63	3059
macro avg	0.44	0.39	0.40	3059
weighted avg	0.64	0.63	0.63	3059

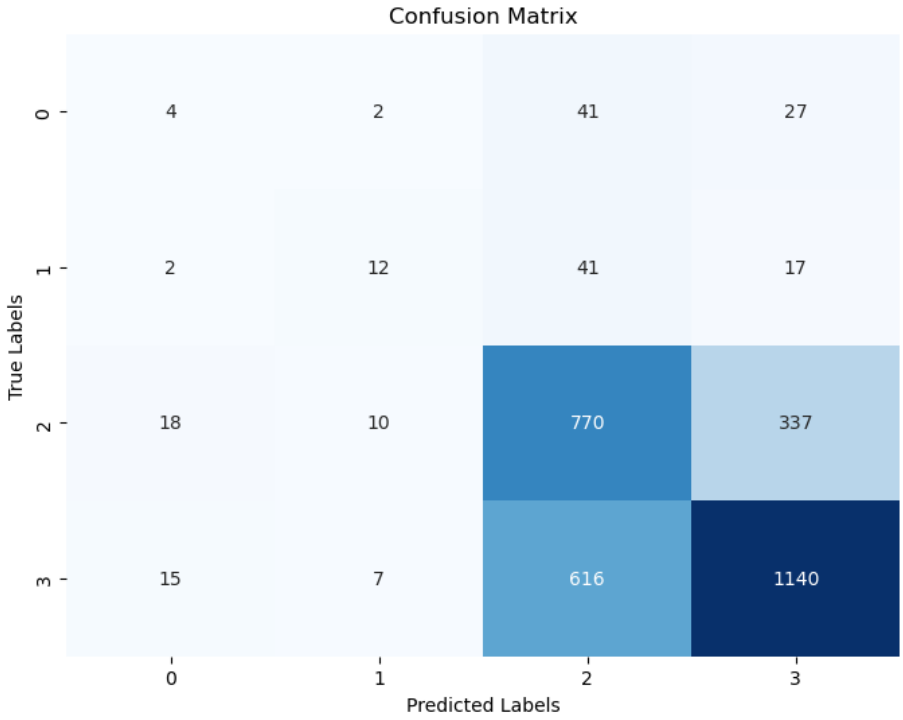


Fig 2. Confusion matrix for the LinearSVC model on validation data

2. Addressing the issue of imbalanced datasets

As can be noted from the previous section, the macro F1-scores were significantly affected by the inability to correctly predict data corresponding to labels 0 and 1. Multiple approaches have been tried in order to remove this limitation, but none of them were successful. Some of them have been presented earlier in this paper (see 'class_weight' parameter of LinearSVC for example).

This section focuses on presenting a new method used with the intention of significantly improve the results for the minority classes: ADASYN (Adaptive Synthetic) oversampling technique. However, this approach was not adopted for the final submissions, as it did not add real value to the analysis process.

A little bit more information about ADASYN: in comparison to some algorithms available in the scikit-learn library that are destined to copy data from the minority classes in order to generate more instances, ADASYN is designed to generate synthetic data for the most difficult examples to classify.

For this task, ADASYN was imported from the 'imbalanced-learn' library:

```
from imblearn.over_sampling import ADASYN
```

Before applying ADASYN, this was the distribution of the classes:

- Number of instances with label 0: 2592
- Number of instances with label 1: 1300
- Number of instances with label 2: 25722
- Number of instances with label 3: 28500

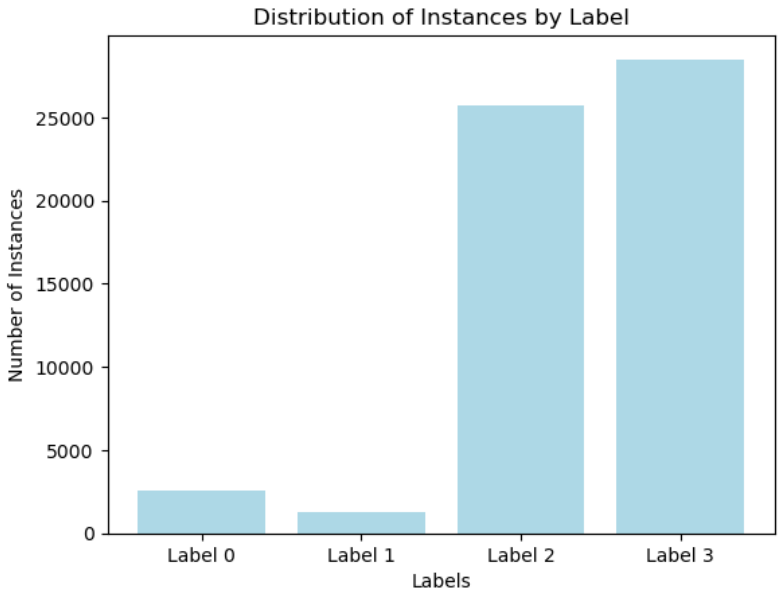


Fig 3. Distribution of instances by label

After applying the ADASYN algorithm, the new distribution for the classes is the following:

- Number of instances with label 0: 28412
- Number of instances with label 1: 28409
- Number of instances with label 2: 25722
- Number of instances with label 3: 28500

The instantiation was done with the following line of code:

```
adasyn = ADASYN(sampling_strategy='minority', random_state=1)
```

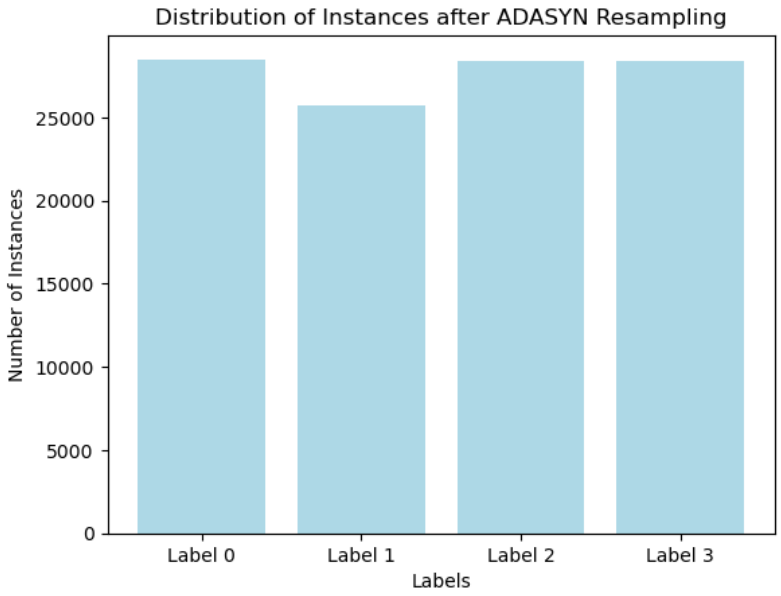


Fig 4. Distribution of instances by label after ADASYN resampling

The way one can use ADASYN is:

```
X_BoW_oversampled, y_oversampled = adasyn.fit_resample(X_train_BoW, y_train)
```

where X_train_BoW represents the training data after the features were extracted with the Count Vectorizer, and y_train contains the training labels.

On the new training data, the same Logistic Regression model that was described in the sub-section 1.2.1 was trained. The results can be analysed from the classification report:

Table 7. Classification report for the Logistic Regression model with ADASYN - 1st method

	precision	**recall**	**f1-score**	**support**
0	0.03	0.07	0.04	74
1	0.15	0.24	0.18	72
2	0.53	0.62	0.57	1135
3	0.75	0.61	0.68	1178
accuracy			0.59	3059
macro avg	0.37	0.38	0.37	3059
weighted avg	0.64	0.59	0.61	3059

As this method didn't improve any results, another approach was tested: a custom ADASYN instance was defined along with a RandomUnderSampler instance imported from 'imblearn.under_sampling'. The strategy implied oversampling the minority classes while undersampling the majority ones. A pipeline (from 'imblearn.pipeline') was created with this 2 instances and fitted also on the X_train_BoW data:

```
adasyn = ADASYN(sampling_strategy={0: 5 * int((train_data['label'] == 0).sum()),
                                   1: 10 * int((train_data['label'] == 1).sum())},
                                   random_state=1)
random_under_sampler = RandomUnderSampler(sampling_strategy=
                                   {label: int((train_data['label'] == label).sum() * 0.5)
                                   for label in [2, 3]}, random_state=1)
```

The new class distribution is:

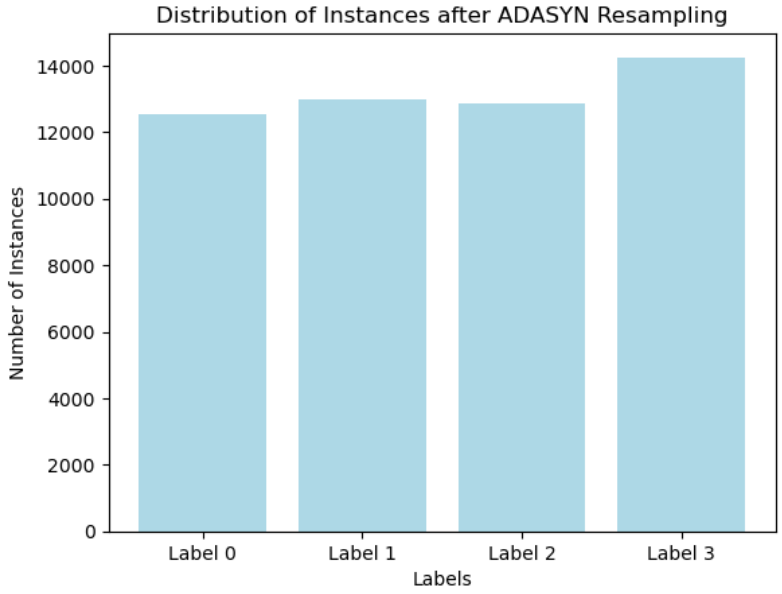


Fig 4. Distribution of instances by label after ADASYN resampling - part 2

The same Logistic Regression model was trained on the new X_BoW_pipeline variable (the result of the fitted pipeline on the original X_train_BoW) and, in this case, the new classification report is:

Table 8. Classification report for the Logistic Regression model with ADASYN - 2nd method

	precision	**recall**	**f1-score**	**support**
0	0.04	0.15	0.06	74
1	0.16	0.39	0.23	72

2	0.52	0.58	0.55	1135
3	0.76	0.58	0.66	1178
accuracy			0.57	3059
macro avg	0.37	0.43	0.38	3059
weighted avg	0.64	0.57	0.60	3059

As one can observe from the classification reports, both ADASYN approaches generated lower evaluation scores compared to the simpler option, when a Logistic Regression model was train on original data. They were tested with other models too, and the results were pretty similar. Therefore, even if it is an interesting approach and may help, in some cases, achieve better results, for this task it was not highly beneficial.

3. Other unsuccessful attempts

Different models were evaluated during this project. Brief results obtained by using them will be presented in this section.

3.1 Random Forest Classifier

Random forests are an ensemble of multiple decision trees trained on distinct parts of the same training dataset. It is used in supervised learning tasks and it is renowned for delivering good results to the classification problems.

This type of model can be imported from 'sklearn.ensemble' and initialized using:

```
randomf_model = RandomForestClassifier(n_estimators=50, random_state=1)
```

To find the best parameters of this ensemble, a Grid Search was run with the following parameters: 'n_estimators': [50, 100, 200] and 'max_depth': [None, 10, 20]. Based on its output, the final configuration of the ensemble is: RandomForestClassifier(n_estimators=100, max_depth=None, random_state=1).

The classification report of this ensemble trained on the extracted features with the Count Vectorizer from the training data and tested on the validation set is:

Table 9. Classification report for Random Forest Classifier

	precision	**recall**	**f1-score**	**support**
0	1.00	0.01	0.03	74
1	1.00	0.04	0.08	72
2	0.51	0.85	0.64	1135
3	0.84	0.56	0.67	1178

accuracy			0.64	3059
macro avg	0.84	0.36	0.35	3059
weighted avg	0.73	0.64	0.63	3059

And the corresponding confusion matrix is illustrated below:

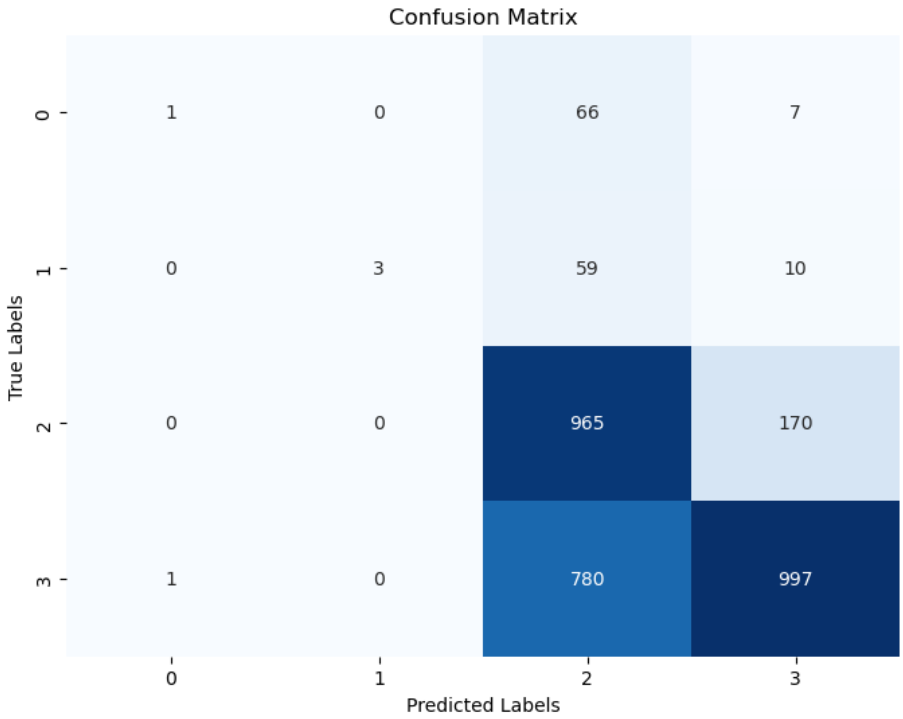


Fig 5. Confusion matrix for the Random Forest Classifier on validation data

The evaluation scores are lower even with the best parameters computed after tuning, consequently the Random Forest Classifier was not used for the final submissions.

3.2 Custom Ensemble

Another test involved creating a custom ensemble which combines the best Logistic Regression model and the best LinearSVC model. To do this, the VotingClassifier from the 'sklearn.ensemble' was imported. The implementation looks like:

```
custom_ensemble = VotingClassifier(estimators=[
('logistic_r', logisticr_model),
('linear_svc', linear_svc_model)],
                    voting='hard')
```

The results obtained on the validation set look promising, as can be seen in the classification report:

Table 10. Classification report for Custom Ensemble

	precision	recall	f1-score	support
0	0.09	0.05	0.07	74
1	0.43	0.21	0.28	72
2	0.52	0.72	0.61	1135
3	0.78	0.62	0.69	1178
accuracy			0.63	3059
macro avg	0.45	0.40	0.41	3059
weighted avg	0.66	0.63	0.63	3059

The associated confusion matrix is displayed in Fig 6.

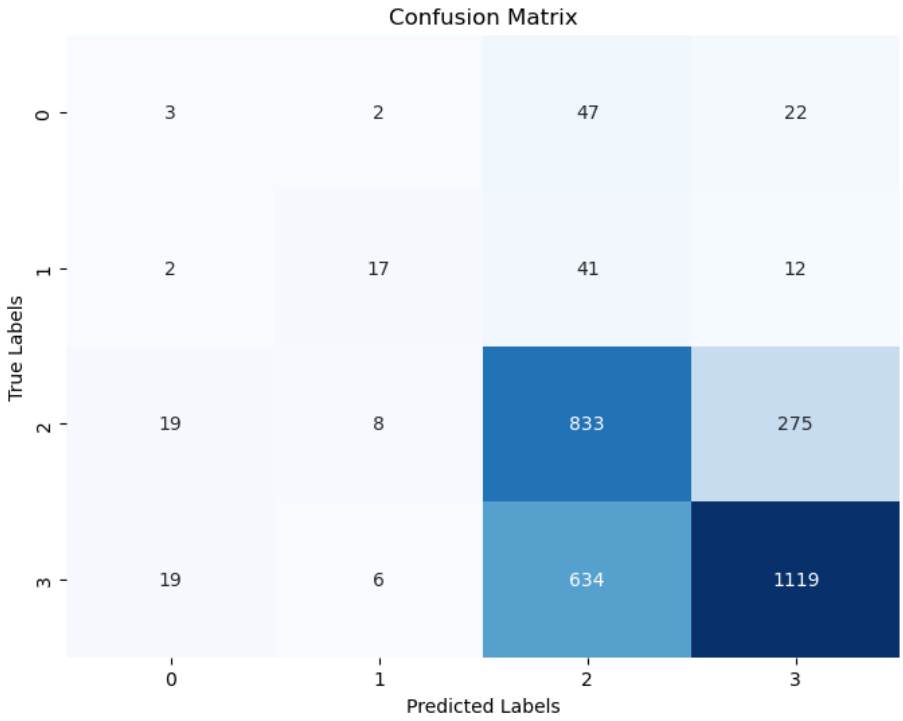


Fig 6. Confusion matrix for the Custom Ensemble on validation data

From this analysis, one can conclude that the ensemble of these 2 models performs slightly better on the label 1 instances and has comparable results with the independent models on the other labels' instances.

4. Conclusions

Using the preprocessing steps and feature extraction method described in the first section of this project, one can conclude that the Logistic Regression model performed the best, followed closely by the LinearSVC model. The hyperparameters for both models were fine-tuned and the results were presented in tables and graphs. Despite trying different techniques to tackle the imbalanced dataset issue, none proved to be entirely successful. Other approaches that work well with these particular datasets need to be explored in order to improve the quality of the predictions.