UNIVERSITÀ DI PISA

Data Mining II

Advanced Topics and Applications

Analysis of the

Ryerson Audio-Visual Database of Emotional Speech and Song

RAVDESS

Authors

Luca Coda-Giorgio
Marco Di Cristo

Data Science & Business Informatics

Academic Year 2022/2023

Contents

1 Introduction

In this study we provide a thorough analysis of a reworking of the RAVDESS (*Ryerson Audio-Visual Database of Emotional Speech and Song*) dataset, which contains audio-visual recordings of 24 actors saying two short statements in English. Each of the recordings is characterized by categorical attributes which refer not only to its format, vocalization (*song* or *speech*), the actors' ID and sex, the *filename*, but also the emotional expression the two statements (*neutral*, *calm*, *happy*, *sad*, *angry*, *fearful*, *disgust* and *surprised*) are sung or spoken with and the corresponding intensity (*normal*, *strong*).

2 Data Understanding and Preparation

2.1 Data Description

The dataset contains a total of 2452 instances, 1828 of which are part of the training set and the remaining 624 belonging to the test set, associated with 434 features, either numerical or categorical. The **numerical features** are basic synthesis measures extracted and transformed not only from the whole original time series audio data, but also from their four sequential and non-overlapping windows. Such features have been obtained applying the following transformations on the statistics listed below:

Transformations: *lag1* (Differencing), *zc* (Zero-Crossing Rate) , *mfcc* (Mel-Frequency Cepstral Coefficients), *sc* (Spectral Centroid), *stft* (STFT Chromagram)

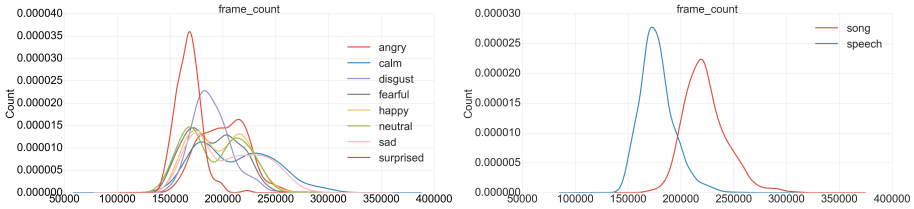
Statistics: *sum*, *mean*, *std*, *min*, *max*, *q01*, *q05*, *q25*, *q50*, *q75*, *q95*, *q99*, *kur*, *skew*

In addition, there are also ten **categorical features**: *modality*, *vocal_channel*, *emotion*, *emotional_intensity*, *statement*, *repetition*, *actor*, *sex*, *filename*, *frame_count*

2.2 Exploratory Data Analysis

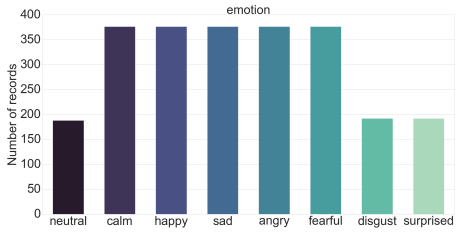
Our journey on this project started by performing an Exploratory Data Analysis. As mentioned before, initially we had 2 datasets: for this part, we merged the datasets in order to achieve a global analysis. After having briefly looked at our dataset, we begun to check for duplicated and missing values, that luckily were not present. Then, we checked for numerical variables with only 1 or 2 values. This kind of inspection will be useful in the next steps to perform dimensionality reduction. We figured out that there were 50 variables with only one value and 8 variables with only 2 values. We will see in the next paragraph how we dealt with them. Hereafter, we checked the values of other variables. In this phase, we noticed a very strange behavior regarding 3 variables, which is that the values of *lag1_q75*, *lag1_q25* and *q75_w1* were multiples of 0.00003. We decided to take note of this behavior, indicating that probably there is some kind of correlation.

The next part of our analysis concerns the distribution of the variables. Before that, we decided to switch to a data-transformation task, because we wanted to plot some data for the distribution analysis. Therefore, we checked the skewness of the variables. We set a minimum threshold of skewness of 0.9, and we found that there were 198 variables with a skewness greater than the threshold. In order to transform this variables, we initially thought of performing a classic log-transformation, but here we have to deal with negative values, therefore we would like to keep their original sign. So, after some research, we discovered that another transformation useful for treating skewed variables is the cubic-transformation. So we applied it and at the end we passed from 198 skewed features to 87



(a) *emotion*s' differences in terms of length

(b) *vocal_channel*'s differences in terms of length



(c) Number of records by *emotion*

Figure 1: Imbalance of *emotion* and peculiar differences in length

skewed features. We decided to maintain them, by considering the fact that we have also to perform feature selection in the next steps. Finally we can switch to the analysis of the variables' distributions.

We started with the categorical features. First of all, we noticed how both the *training* and *test* sets provided are sorted over all the categorical features, and since this could inhibit the tasks that will be performed in the next sections, for instance in classification where the classifiers could learn patterns in the data only due to the way the records are collected, we must shuffle the dataset.

Looking at the frequency distribution of the records by the categorical features' classes, we noticed how all of them were nearly perfectly balanced except *emotion* (Table 1, Figure 1c).
We also analyzed the differences in terms of length of the different *emotions* by plotting the *frame_count* variable's distribution grouped by *emotion* (Figure 1a) Here, we noticed that the emotions that followed a normal distribution were *emotion_angry*, where the majority of the values fall between 150000 and 200000 and *emotion_disgust*, where the majority of the values fall between 150000 and 240000.

For the other *emotion*s, except *emotion_surprised* it seemed that the distributions were more like the Bimodal Distribution, with two peeks. For *emotion_surprised* it is difficult to determine the similarity with some well-known distributions. After having noticed that the majority of the distributions are similar to a Bimodal Distribution, we started to think about a possible correlation with another variable. By plotting the same continuous variable (*frame_count*) grouped by the categorical variable *vocal_channel* (Figure 1b) it seems that most of the distributions of *frame_count* by *emotion* are bimodal, probably due to the fact that for *vocal_channel* we have a normal distribution for both the values. So this behaviour may explain the 'double-peek' observed.

After that we moved to the analysis of the distributions of the continuous features. Due to the high dimensionality of the dataset and the variability of the values, it seemed useless to illustrate the distribution of every single variable. Instead, we focused on the distribution of selected variables, grouped by two categorical ones: *vocal_channel* and *emotion*. We decide to show these features because during the classification task, they will be the variables that we will try to predict.

Table 1: Descriptive statistics: Categorical features

Feature	Classes	Frequency
vocal_channel	speech	0.587276
	song	0.412724
emotion	fearful, angry, happy, calm, sad	0.153344
	surprised, disgust	0.078303
	neutral	0.076672
emotional_intensity	normal	0.538336
	strong	0.461664
statement	“Kids are talking by the door”	0.500000
	“Dogs are sitting by the door”	0.500000
repetition	1st	0.500000
	2nd	0.500000
sex	M	0.508972
	F	0.491028

2.3 Dimensionality Reduction and Feature Selection

2.3.1 Filter Methods

As said before, after having performed the cubic-transformation on the skewed features, first we performed a correlation analysis to check if any variables were redundant with respect to each other. To do so, we started by computing the pairwise Pearson's correlation coefficients between all the continuous features, setting a threshold at 0.9 as the maximum tolerated absolute value. The results showed we had a total of 180 attributes which exceeded such cut-off, therefore we considered them linearly related.

To ensure we did not have any other non-linear relationships, We did the same with respect to the Spearman's ranking correlation coefficients maintaining the same cut-off as before and this time we discovered 193 correlated attributes. We then decided to remove the union of the sets of attributes resulting from both measures, achieving an only 178-feature vector.

Once we removed most of the redundant features, we also wanted to remove the irrelevant ones and we did so by exploiting the variance threshold strategy, checking all the features' variances and keeping only the ones that met at least a lower limit of 16%, therefore obtaining a further decrease in dimensionality by 55 features.

After that, we applied *sklearn*'s *MinMaxScaler* to normalize the features space in order to not have any of the remaining ones overwhelming the others. The dataset originated through all the steps described up to this point is the one we then proceeded to use for most of the unsupervised and supervised tasks in the next chapters. Before going forward, we also removed the *filename* feature, since it did not provide, in our opinion, any useful insights.

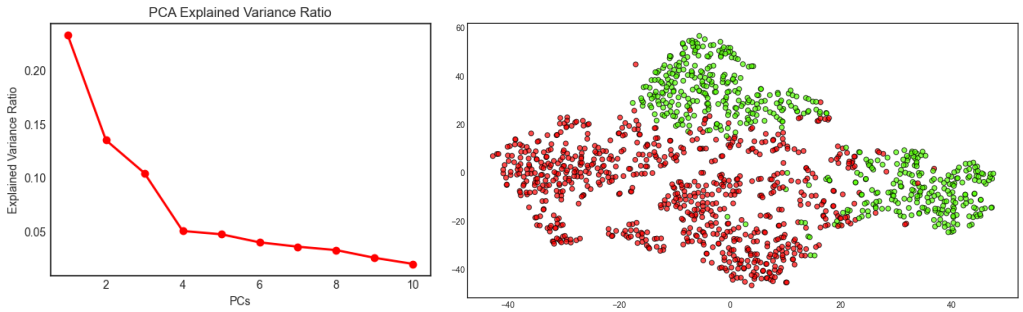
2.3.2 Wrapper Methods

To make sure we achieved the best performances over all the classification and regression tasks described next, we also wanted to extract from the latter even less potentially dimesionality-cursed datasets, therefore we also employed two wrapper approaches from the *sklearn* library: *Univariate Feature Selection, SelectKBest* for regression tasks and the *Recursive Feature Elimination, RFE* for classification tasks, both with the aim to select the 15 best features with respect to the corresponding targets (*zc_sum* and *emotion*).

In order to converge to the optimal number of attributes to select, in the case of RFE, as selector-estimator we used *DecisionTreeClassifier* from *sklearn* tuned with a *GridSearch* Cross Validation, while for the Univariate Feature Selection we passed as scoring function parameter the *f_regression* function from *sklearn.feature_selection*, since it performs univariate linear regression tests which return p-values and F-statistic.

2.3.3 Feature Projection

We also tried to perform two feature projection methods in order to evaluate if we could use less components and obtaining good results as well. We use the dataset with the cubic-transformation applied (the same of the previous dimensionality reduction steps). Firstly, we tried to apply a PCA, but instead of setting a number of components, we set the percentage of variance that we wanted to represent, i.e. 85% (Figure 2a). For evaluating the goodness of the PCA, we tried to run a Decision Tree in order to predict the binary variable *vocal_channel*. After that we also applied t-SNE's method, by setting the number of components to 2 (Figure 2b). For both cases, we obtained good results, where t-SNE performed better.



(a) PCA with 0.85 variance ratio (b) t-SNE projections grouped by *vocal_channel*

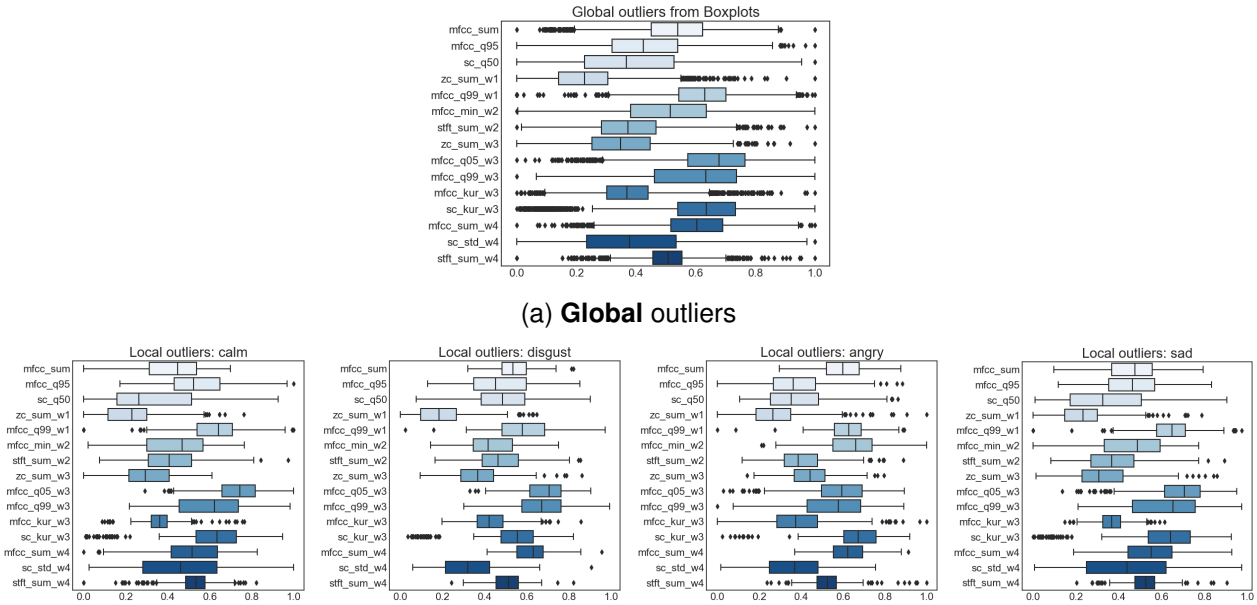
2.4 Anomaly Detection

Before proceeding with the classification and regression tasks that will be extensively depicted in the dedicated chapters, we employed 12 different anomaly detection algorithms to discover if any outliers were present in the data after having removed most of the possible noise in the latter, both in terms of features values but also in terms of anomalous records with respect to the others.

For this task, we decided to implement these techniques on the RFE-reduced dataset with respect to the *emotion* label, as explained in a previous section, therefore only honing in on the fields that will be crucial for the upcoming supervised tasks.

With this in mind, we exploited algorithms with a **local** strategy, focusing on each class of the variable *emotion*, such as **Boxplots** and **Automatic Boxplots** (Figure 3), while for most of the other algorithms we preferred a **global** approach, although we also checked for local outliers as mentioned before. In this case, we deployed the following methods: **Grubb's Test**, a Deviation-based approach using as **Smoothing Factor** the variance decrease of the features in the data after removing the outliers sequentially, **Elliptic Envelope**, **LOF**, **COF**, **HBOS**, **kNNs** (common outliers among several *k*s), **DBSCAN**, **ABOD**, **Isolation Forest** and finally its variant **Extended Isolation Forest**. For every algorithm, except DBSCAN, we selected the top 250 outliers based on the outlierness scores returned by each method, therefore not using a binary labeling.

After having applied all the different approaches, we collected all the sets of outliers and performed intersections between them to retrieve the set of common anomalies. Here, between the multiple methods, we didn't find any of them, so we decided to restrict these intersections to only sets caught by **LOF**, **Elliptic Envelope**, **kNNs**, **DBSCAN** and **Extended Isolation Forest**, resulting in 33 outliers.



(a) **Global** outliers

(b) **Local** outliers for *emotion* **Calm**

(c) **Local** outliers for *emotion* **Disgust**

(d) **Local** outliers for *emotion* **Angry**

(e) **Local** outliers for *emotion* **Sad**

Figure 3: Visualization of both **Global** and **Local** Outliers through Boxplots

Knowing each's index, we decided to remove them from the dataset, since, in our opinion, they almost certainly represented global anomalies (Figure 4).

Furthermore, in order to visualize the scatterplots portraying the clouds of anomalies resulting from each final method we selected, we chose to employ the *TSNE* class from *sklearn* on the dataset and therefore plot the data only based on its first two components (Figures 7).

2.4.1 Implementation and Results

First of all, we begun the anomaly detection process visualizing the Boxplots for each attribute with respect to every class of the *emotion* variable, leveraging also the results returned by an Automatic Boxplot algorithm. Such methods did not produce very promising results, since there were not any common outliers between all the attributes.

We then proceeded with the exploitation of the *HBOS* (Figure 5a) and *Elliptic Envelope* (Figure 5b, 7a) algorithms respectively from *pyod* and *sklearn*, setting the contamination parameter equal to 0.1 (in order to discover the top 1% outliers) and the *Grubb's Test* algorithm, which returned only 4 outliers that did not intersect with any other set stemming from the other approaches. To represent the first two histograms, we defined the number of bins using Sturge's Rule.

The most interesting results came from the implementation of the following methods. As distance-based approach we performed *kNN*s (*pyod*) with *k* values in the range (1,10) using the *minkowski* distance as metric parameter and we discovered a number of anomalies in the range (0,210) with respect to *k*. Since we could not properly choose the correct number of neighbors, this led us to employ *kNN*s with *k*s from 2 to 20 and same metric as before, but this time we averaged the outlierness scores for each record with respect to every number of neighbors. Once we retrieved such distances, we selected, as usual, the top 1% outliers (Figure 7b). In Figure 5c we represented the aforementioned distances (i.e. scores) with respect to each record.

The Local Outlier Factor *LOF* (Figure 7c) from *sklearn* was implemented in an unsupervised fashion, using as metric the *minkowski* distance as in the *kNN*s approach. The methology remained the same in terms of outliers selection, but this time each record was paired not only with the anomaly

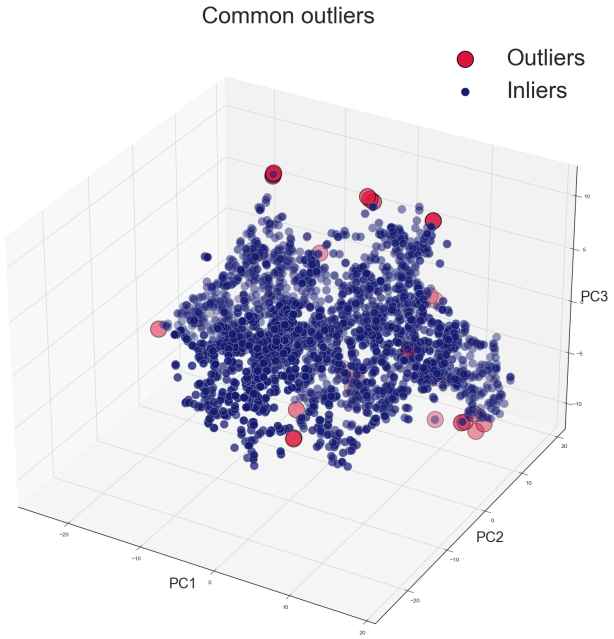
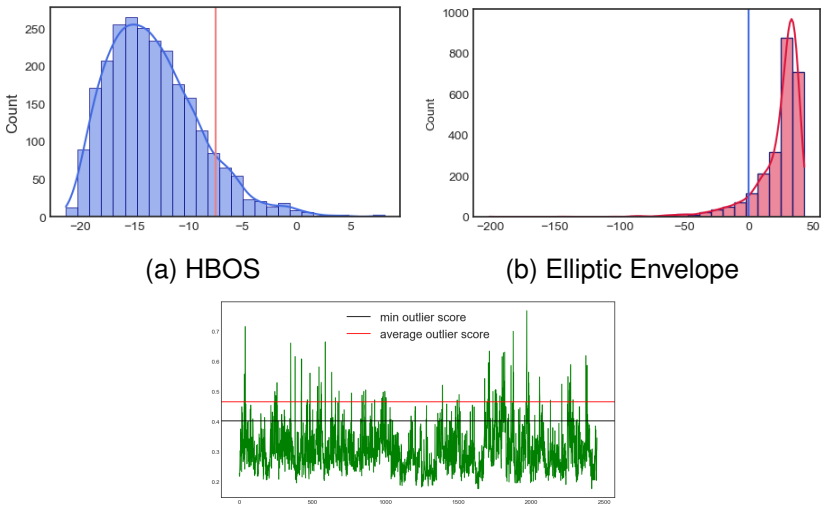


Figure 4: Common outliers between **LOF**, **Elliptic Envelope**, **kNNs**, **DBSCAN** and **Extended Isolation Forest**

label, but also with the proportional outlierness score returned by *LOF*.

As density-based approach we also employed *DBSCAN* (Figure 7d) from *sklearn*. To tune the *min_samples* and *eps* parameters, we used the elbow method as illustrated in Figure 6, in order to select the epsilon that best expressed the asymptotic behavior of the distance curve. Hence, we chose $k = 10$ and $eps = 0.43$, and with this configuration 70 outliers were retrieved.

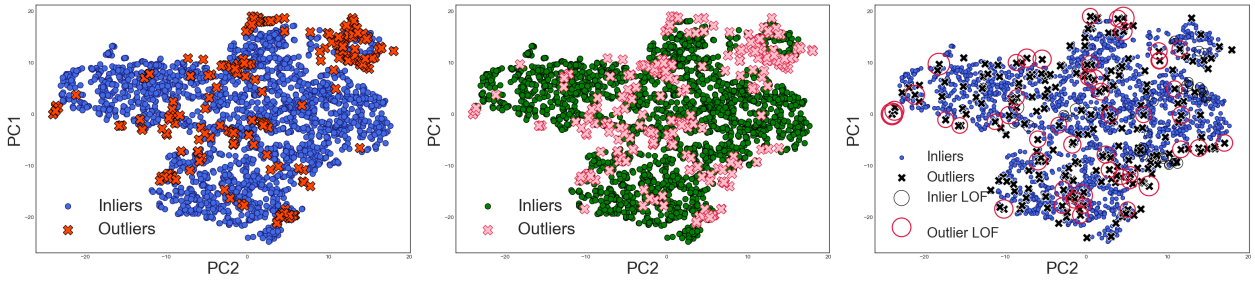
The last approaches we took for this task were model-based. We first implemented *IsolationForest* from *pyod*, which builds a forest of trees and for each selects only one random feature at a time on which to split the data, and as number of estimators we chose 1000 decision trees. Secondly we wanted to implement also the *Extended* version of the algorithm, which supports the selection on multiple features for each split. This time we set the same number of estimators but a maximum of 3 features for the splitting step. We finally chose the second variant as algorithm to make our final common-outliers labeling, since we thought that the isolation of the anomalous records in the data would have been more accurate and also more efficient.



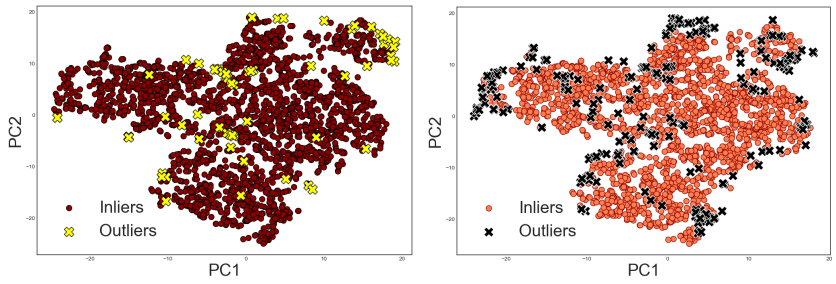
(a) HBOS (b) Elliptic Envelope

(c) kNNs

Figure 5: Results found through HBOS, Elliptic Envelope and kNNs

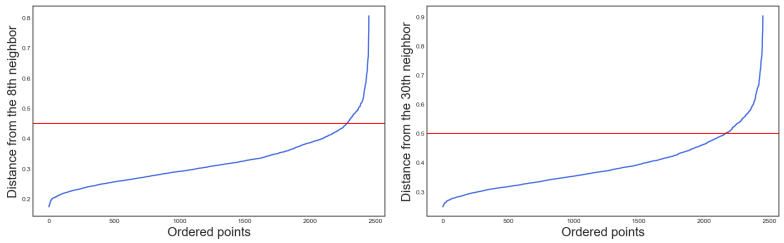


(a) Elliptic Envelope (b) kNNs (c) LOF



(d) DBSCAN (e) Extended Isolation Forest

Figure 7: Anomalies detected with different approaches and visualized with *TSNE*



(a) Distance of the records from the 8th neighbor (b) Distance of the records from the 30th neighbor

Figure 6: Elbow method to tune *k* and *eps* parameters for DBSCAN

2.5 Imbalanced Learning

2.5.1 Didactic objective

In order to do this task, we had to deal with the didactic request stated in the guidelines of the project. In this case, it asked us to perform some imbalance learning algorithms on a really imbalanced initial situation. Since there were not any variables so imbalanced, we decided to use *vocal_channel* as target variable by manually changing its distribution. So, initially, we have 59%/41% for the values *speech*/*song*. So, we resampled the dataset and we obtained a 94%/6% situation. In order to evaluate the performance of oversampling and undersampling algorithms, we predicted the 2 values by using a Decision Tree (Figure 8), and we used the result as a baseline for the next steps.

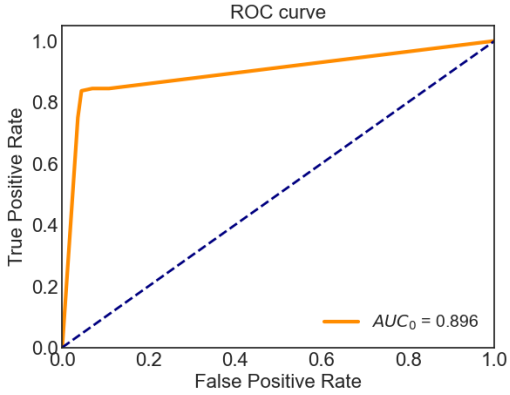


Figure 8: *Decision Tree*'s ROC Curve on Imbalanced data

2.5.2 Undersampling

Table 2: Undersampling's performance - Decision Tree

Undersampling Algorithm				
Algorithm	**Accuracy**	**F1 Score**	**AUC Baseline**	**AUC**
RandomUndersampling	0.88	[0.89 , 0.86]	0.89	0.88
Condensed Nearest Neighbour	0.92	[0.93 , 0.91]	0.89	0.93
Tomek Links	0.87	[0.89 , 0.83]	0.89	0.89
Edited Nearest Neighbours	0.93	[0.94 , 0.91]	0.89	0.93
Cluster Centroid	0.82	[0.82 , 0.83]	0.89	0.86

The first algorithms we tried are the undersampling ones. For each algorithm, we resampled the *training* set, we plotted the new resampled space with PCA, and then we followed the same steps that we performed for the baseline:

- Resample the *training* set
- Plot resampled space with PCA
- Predict *vocal_channel* with a decision tree
- Calculate the AUC
- Compare the AUC with the Baseline AUC

The results we obtained (Table 2) showed us that there were not really big benefits compared to the baseline prediction (the imbalanced one). All the algorithms' performances were almost the same in terms of AUC, except for the **Condensed Nearest Neighbour** and **Edited Nearest Neighbours** algorithms, where the AUC is slightly higher.

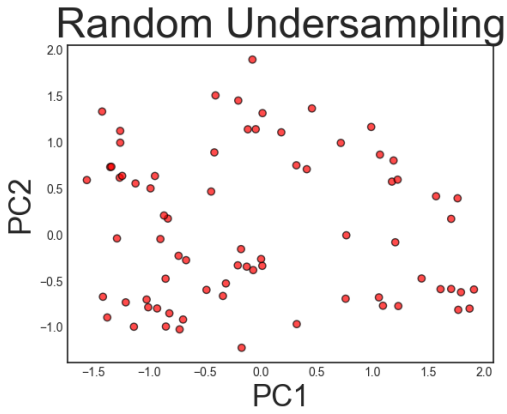


Figure 9: Random Undersampling

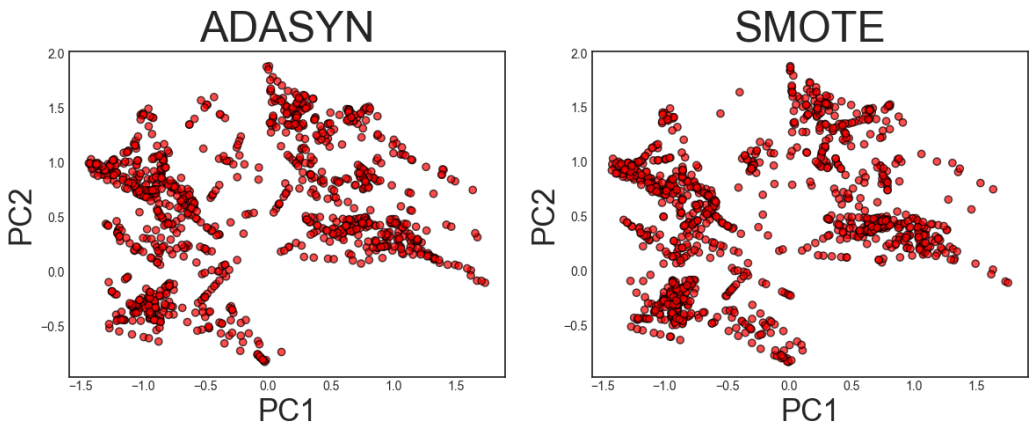
2.5.3 Oversampling

Table 3: Oversampling's performance - Decision Tree

Oversampling Algorithm				
Algorithm	**Accuracy**	**F1 Score**	**AUC Baseline**	**AUC**
RandomOversampling	0.90	[0.92 , 0.88]	0.89	0.89
SMOTE	0.90	[0.91 , 0.88]	0.89	0.90
ADASYN	0.90	[0.92 , 0.88]	0.89	0.90

Then, we moved on to try the oversampling algorithms. We decided to perform our usual pipeline to evaluate this kind of task: even in this case, we noticed that running the oversampling algorithms over our imbalanced dataset was still not useful enough: all the algorithm had the same results in terms of accuracy and AUC (with an negligible difference with the Random Oversampling).

Figure 10: PCA Visualization Oversampling



(a) ADASYN

(b) SMOTE

Comparison of Performances Metric											
	Logistic Regression	**Linear SVC**	**SVC**	**Random Forest**	**Bagging**	**Ada Boost**	**Normal NN**	**Dropout NN**	**GBoost**	**Hist GBoost**	**X GBoost**
Accuracy	0.48	0.46	0.47	0.46	0.43	0.35	0.49	0.48	0.47	0.46	0.45
F1-score	0.48	0.46	0.47	0.46	0.42	0.35	0.48	0.47	0.46	0.46	0.44
Precision	0.49	0.46	0.48	0.47	0.44	0.38	0.49	0.47	0.47	0.46	0.45
AUC	0.865	0.87	0.86	0.865	0.86	0.76	0.861	0.847	0.845	0.84	0.845
Recall	0.50	0.47	0.49	0.47	0.43	0.35	0.50	0.50	0.47	0.47	0.45

In conclusion, we can state that running imbalanced learning algorithms (both oversampling and undersampling) on our initial imbalanced situation, does not increment by much our performance on a simple classification task.

3 Classification and Regression

In this part, we performed some advanced classification tasks. Initially, by considering all the analyses conducted until now, we are faced with two decisions to make:

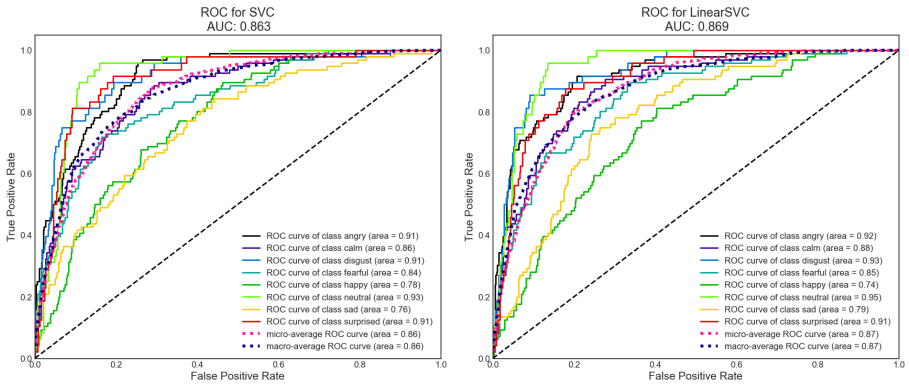
- Which dataset to use, between the datasets produced with the feature-selection tasks
- Determining the classification task

As far as what dataset to use, we decided to work with the dataset with the *filter methods* applied. Therefore we have 128 features and 2419 rows (1795 the *training* set and 624 the *test* set). Even if this was our first choice, we also tried to use other datasets for some classification algorithms as stated during the description of Wrapper Methods (*Gradient Boosting* and *Neural Networks*).

With regards to the classification task, we initially decided to use *vocal_channel* as variable to predict, but at the end we switched to ***emotion*** due to the fact that during the previous steps we already obtained good result with the prediction of *vocal_channel*. So, our classification task will be a multi-class prediction (with 8 classes) performed on the variable *emotion* (which values are: *neutral, calm, happy, sad, angry, fearful, disgust, surprised*)

3.1 Support Vector Machines

Figure 11: SVC and Linear SVC ROC AUC



(a) **SVC** ROC (b) **Linear SVC** ROC

Let's start by analysing the **Support Vector Machines** method. We performed both SVC and LinearSVC algorithms. Each of them obtained (more or less) the same performance towards the

classification task. In order to reach this results, we applied before an hyperparameter tuning phase, where we found some interesting insights:

On one hand, with regards to LinearSVC, we set the following parameters during the hyperparameter tuning phase:

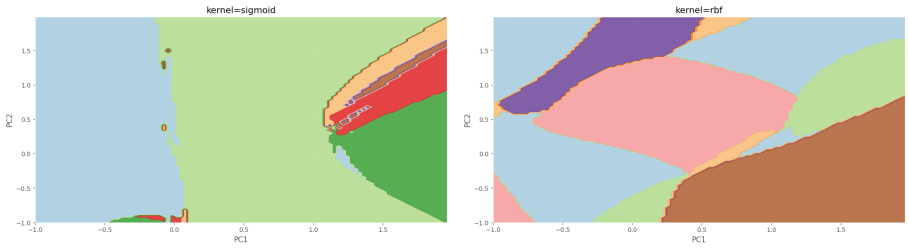
C = [50, 10, 1.0, 0.1, 0.01]
penalty = ['l1','l2']
loss=['hinge','squared_hinge']
multi_class=['ovr','crammer_singer']

These parameters were chosen by applying a GridSearch on a RepeatedStratifiedKFold (from *sklearn* library). Regarding the *C* parameter, we decided to use this parameter in order to evaluate the performances both for small values of C and higher values. We figured out that the best parameters were: **C**: 1.0, **loss**: squared_hinge, **multi_class**: ovr, **penalty**: l2. During the evaluation phase we reached an *accuracy* of 0.57. In addition, with the aim of plotting the ROC curve for each class, we used a probabilistic approach found online: *CalibratedClassifierCV*.

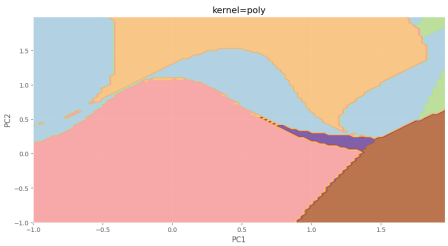
On the other hand, regarding **SVC**, we followed the same pipeline stated before. In this case, the parameters tuned were:

Kernel = [**'Poly'**, **'RBF'**, **'Sigmoid'**]
C = [50, 10, 1.0, 0.1, 0.01]
gamma = ['scale']

Here, the interesting parameter to analyze is the *Kernel*. In our case, the Kernel that showed the best performance during the GridSearch was ***Poly***, with which we had an accuracy of 0.61 during the evaluation phase. During the test phase, the algorithm behaved almost like all the others, but here we had a better performance in the prediction of *emotion_fearful*, with a precision of 0.60. Even if we have complex decision boundaries due to the number of values to predict (8), we believe that it could be interesting to show the decision boundaries' surface, for each tuned Kernel, approximated with a 2-D representation by using PCA feature projection on the axes (Figure 12).



(a) **Sigmoid** Surface (b) **RBF** Surface



(c) **Polynomial** Surface

Figure 12: Decision Boundaries Surface for each tuned Kernel

Comparison of Performance Metrics			
	SVR Linear	**SVR RBF**	**SVR Poly**
R-squared	0.989	0.992	0.989
RMSE	0.014	0.012	0.014

Table 4: Support Vector Regressor variants' performance measures

3.1.1 Regression with Support Vector Regressors

As a final implementation of the Support Vector models we presented in this chapter, we also performed a multiple regression task with *sklearn*'s *SVR* regressor, and we decided to choose, as continuous target variable, ***zc_sum***. Due to this report's space constraints, we will not repeat all the preprocessing steps as we did for classification, but we will limit ourselves to only describing the parameters we tuned and the results we achieved. In this case, for each SVR Kernel-based variant (**Linear**, **RBF** and **Poly**) we tuned the following parameters with a *GridSearchCV*: **C** [0.01, 120], **gamma, in the case of *RBF* Kernel**[0.001, 10] and **epsilon**[0.005, 2].

Once the tuning of each regressor was completed, for the **linear** Kernel variant we found as best parameters C=*25* and epsilon=*0.01*, while for the **RBF** regressor we set C=*70*, gamma=*0.003* and epsilon=*0.005*. Lastly, the regressor with **Poly** Kernel was defined with C=*90*, gamma=*0.01*, epsilon=*0.01* and the degree of the polynomial set to 3.

We then trained the models on the concatenation of the *train* and *validation* sets and evaluated their respective performance over the *test* set. The best performance we obtained referred to the Support Vector Regressor implemented with the **RBF** Kernel, not only in terms of *Goodness of fit*, **R-squared**, but also in terms of **RMSE**(*Root-mean-squared-error*.

The RBF SVR's R-squared and the residuals distribution are shown in Figure 13.

(a) Prediction Error (b) Residuals distribution

Figure 13: **SVR** with **RBF** Kernel

We also tried using the Univariate Feature Selection-reduced dataset to perform this task, and we obtained much worse results, so we chose to not further discuss them in this section.

3.2 Logistic Regression

Another classification algorithm we tried is the Logistic Regression. Initially we did not think that this algorithm would suit our classification task, due to the fact that literature often illustrates this classifier in a binary-classification context. However we obtained quite good results here also, with 4 values of *emotion* (*angry, calm, fearful and neutral*) with a precision greater than 0.55.

In the hyperparameter tuning phase, we chose to tune the following parameters:

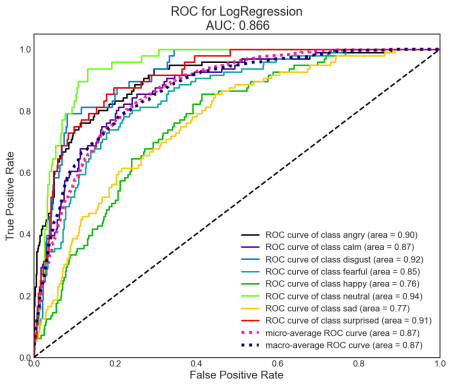
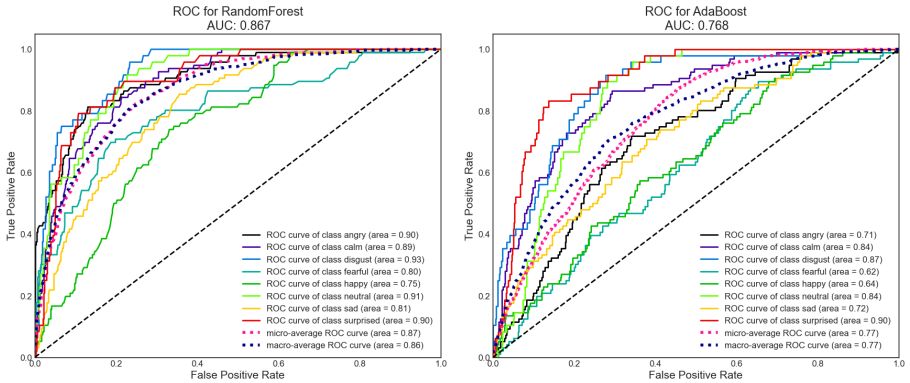


Figure 14: **Logistic Regression** ROC

solvers = ['newton-cg', 'lbfgs', 'liblinear'] **penalty** = ['l2'] **c_values** = [100, 10, 1.0, 0.1, 0.01, 0.001]

The best parameters found during the cross-validation were: **C**: 10, **penalty**: 'l2', **solver**: 'newton-cg' During the validation phase we noticed that also with **C**=1 and **C**=100 the results were good, much different from the ones achieved with 0.1 and 0.001 as **C** values.

3.3 Ensemble Methods: Bagging, AdaBoost and Random Forest



(a) **Random Forest** ROC (b) **AdaBoost** ROC

Figure 15: Ensemble Methods ROC

In the case of Ensemble Methods, instead, we implemented the following algorithms: **Bagging, AdaBoost and Random Forest**. The performances were almost the same for **Bagging** and **Random Forest** (Figure 15a). On the contrary, the **AdaBoost** performance (Figure 15b) was the worst compared to all the algorithms used during the classification task. For **Random Forest**, the parameters tuned were:

	n_estimator	criterion	class_weight	min_samples_leaf	min_samples_split	max_features
Random Forest	[10, 100, 1000]	['gini', 'entropy', 'log_loss']	['balanced', 'balanced_subsample']	[1, 5, 10, 20]	[2, 5, 10, 20]	['sqrt', 'log2']

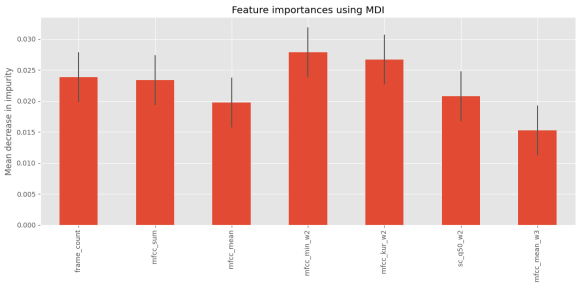
For the **Bagging** classifier, we decided to tune only the *n_estimator* parameter (with values: [10,100,1000]) and for the **AdaBoost** classifier the parameters: *n_estimator* (with the same values of Bagging) and *learning_rate* (with values:[0.0001, 0.001, 0.01, 0.1, 1.0].

Let's focus on the most interesting algorithm of this part, i.e. **Random Forest**. We tried to analyze what was going on during the computation of this method. After the validation phase, we chose to run

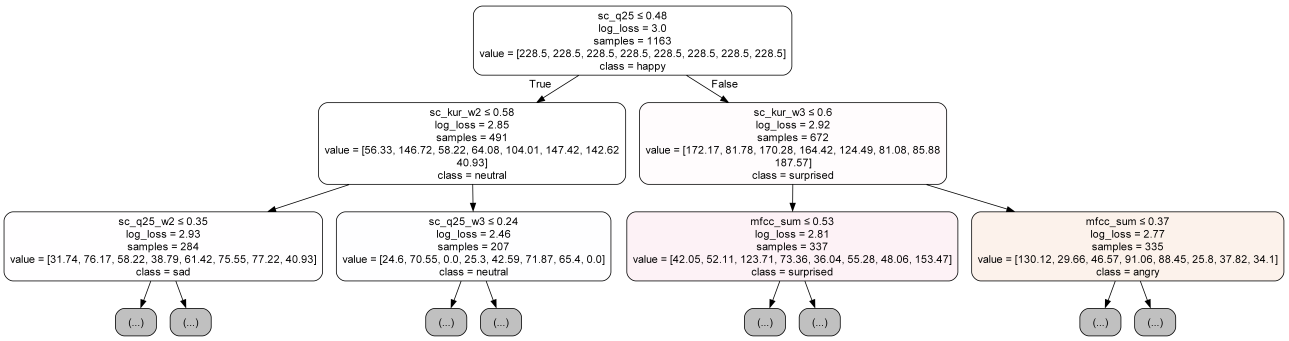
the model with these parameters:

class_weight= balanced_subsample, **criterion**= 'log_loss', **max_features**= 'sqrt',**min_samples_leaf**= 1, **min_samples_split**= 2, **n_estimators**= 1000

With this parameters we obtained an AUC value of 0.866. After that, we calculated the feature importance (Figure 16a), then we extracted the features with an importance greater than 0.015 derived from all the trees in order to try to limit the 'black-boxing' behavior due to 1000 estimators.



(a) Feature importance towards all the Random Forest



(b) A tree in the Random Forest

Figure 16: Focus on Random Forest

In addition, we chose a single tree (the 760th) with the aim of analyzing its structure (Figure 16b). As we expected, we did not find all the variables extracted with the feature importance, because of the high number of estimators chosen and the high depth of the tree (that we needed). Just for curiosity, we also tried to test our algorithm with 100 estimators, but we lost some points in the AUC (as expected by looking at our validation phase) and even here not all the feature importance's attributes were present.

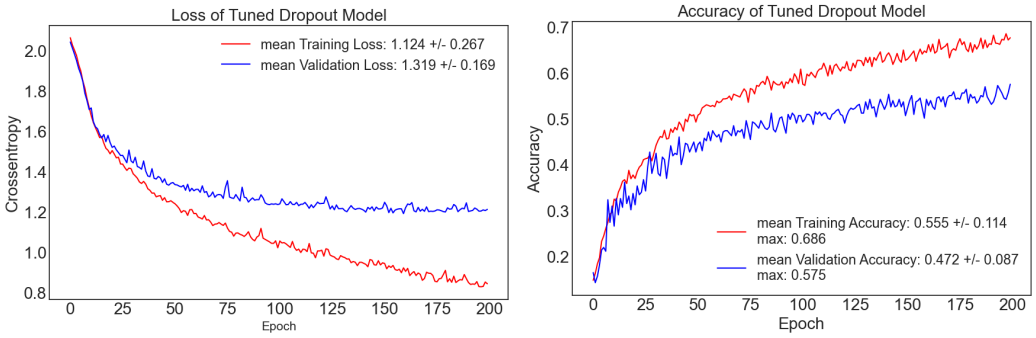
3.4 Neural Networks

In this section we will describe the methodology we used to build, tune and employ different variations of Neural Networks models through the use of the *tensorflow* and *keras* libraries and their respective modules, also by exploiting *scikeras KerasClassifier* wrappers to make such models compatible with all the tuning and evaluation pipelines provided by *sklearn*.

Here, the objective remained the same as for the other classification tasks, so we kept as target variable ***emotion***. As for the adopted dataset to both develop and employ the models, we not only made use of the one resulting by applying only the filter approaches described in the *feature selection* chapter, but also the RFE-reduced dataset. In the following, We will start with the discussion of the first, which consists of 2419 instances with 128 features, and after that we will move on to the second, that only consists of the top 15 features as explained in the corresponding section.

First of all, we split the dataset into two distinct sets of instances, the *training* and *test* sets, and after that we encoded the *emotion* variable using *LabelEncoder* from *sklearn* also binarizing all the categorical independent features, making them suitible for the task at hand, therefore applying another re-shuffling of the dataset to make sure the models didn't learn from any trivial patterns in the data. In the following step, we proceeded to split the *training* set in what we called *train* and *validation* sets with a holdout method, the first of which was made up by 80% of the *training* set and the second by the remaining records (20%). This way we made sure to not touch in any way the set of records we will then use to evaluate the models, the *test* set.

Since during the *exploratory analysis* phase at the beginning of this paper we discovered that *emotion* was not perfectly balanced, we decided to keep in mind this information and that it could be potentially useful to correct such behavior by employing the best oversampling algorithm we found in the *Imbalanced Learning* section, *ADASYN* (we will then decide not to apply it).



(a) Dropout ANN *loss* curve

(b) Dropout ANN *accuracy* curve

Figure 17: **Dropout (0.3) ANN**'s loss and accuracy curves

Topology and Configuration of the ANNs Moving to the actual creation of the models through functions (since the *scikeras* wrapper requires it), we decided to extend this process to obtain **three ANN variations** with different ad-hoc configurations: the first without any type of regularization (which will be called ***Normal***), then one with a ***Dropout*** and lastly incorporating an ***l2* regularization**. Once we performed a brief evaluation over the mentioned variations using simple structural configurations in terms of internal layers and activation functions, output activations, weights initializations and optimizers, we proceeded with the essential **hyper-parameter tuning** procedure, making use of Cross Validations through *sklearn*'s *GridSearchCV*. To evaluate the models, in terms of loss we used the *sparse_categorical_crossentropy* and as metric we used the global *accuracy*.

For each of the three variants, since passing the whole list of values for each possible parameter would have been too computationally expensive, we begun by tuning first the layers, using three sets of configurations with a different **number** of **internal layers** [2,3] and **internal nodes** [from 4 to 128 each], the **internal activation functions** [sigmoid, relu, tanh, softmax], the **batch size** [from 20 to 240] and the **epochs** [from 20 to 200]. Once we found consistent results across the trials, we chose to begin the tuning of the **optimizer** [adam, adagrad, sgd, rmsprop] with constant learning rate and the **weight initializations** [uniform, normal]. This time we found that the best results were obtained invariably with *adam* and *normal* weights across the variants.

As said before, we performed each of the previous steps for each variant, and especially in the case of both the regularized ones, we also tuned the **dropout** disconnected-units *percentage* [from 0.8 to 0.3], the *position* on which this occurred [after each internal node] and the **l2** coefficient. We also tried different combinations with progressively decreasing and increasing dropout after each layer.

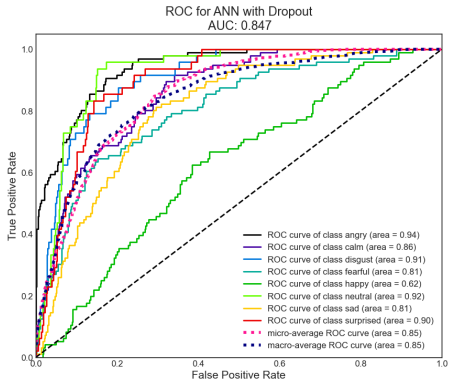


Figure 18: ROC AUC for **Dropout (0.3) ANN**, each class OVR

Results Out of the three models, we decided to keep only the *Normal* and *Dropout* variants since the l2 regularization didn't perform well enough looking at the cross validation results.

The parameters were consistent between the two we kept: layers *[64,32,8]*, internal activation function *relu*, *50* batch size, *200* epochs and *adam* with *normal* weights. As *dropout* we set as best value *0.3*.

The most important results were obtained varying output activation function, since we found that performing various runs with only the *train* and *validation* sets returned the most volatile performances. The results we illustrate in Table 5 refer to the best output layer activation function, the *sigmoid*, while in Figure 18 we show the ROC AUC curve for the best model, the **Dropout ANN**. We chose this ANN variation since we wanted to avoid the overfitting fenomena. In fact, the dropout 'layer' in our model randomly selects 30% of the input neurons, or nodes, and these will be ignored during training, therefore their contribution will be temporally disconnected on the forward pass and the weights in the backwards pass will not be updated.

In Figure 17 we show the tuned **Dropout (*0.3*) ANN** *loss* and *accuracy* curves.

As we can observe from the ROC Curve in Figure 18, **angry** (F1-score of 0.63) was the emotion that was most-correctly classified by the Dropout ANN (and also by the other), while instead the worst performance was obtained on class **sad** (F1-score of 0.25).

As previously said, in order to evaluate the different variants also on the RFE-reduced dataset, we performed the same preprocessing and tuning steps on such dataset and we observed that the results were much worse than the non-reduced one. Therefore, we decided to not include the models' performance measures in this section.

Comparison of Performance Metrics			
	Normal ANN	**Dropout ANN**	**L2 ANN**
Accuracy	0.493	0.481	0.350
F1-score	0.48	0.47	0.25
Precision	0.49	0.47	0.31
AUC	0.861	0.847	0.760
Recall	0.50	0.50	0.29

Table 5: Neural Network variants' performance measures

3.5 Gradient Boosting Machines

In this subsection we discuss about another classification task involving the deployment of Gradient Boosting Machines and its variants. As for the libraries, we used the *GradientBoostingClassifier* and

HistGradientBoostingClassifier from *sklearn* and the *XGBClassifier* from the *xgboost* library.

For this task, we implemented the same preprocessing steps as with *Neural Networks* and we performed the classifications on the same target variable, *emotion*, and the same datasets as before, so the one originated from only filter approaches during the feature selection, and also the one obtained through RFE dimensionality reduction.

Hyper-parameter tuning For the tuning of the parameters passed to the different algorithms, we chose to make use of *sklearn*'s *RandomizedSearchCV* cross validation function, setting the value 5 as folds. For the same complexity reasons as we explained in the previous chapter we tuned the parameters sequentially and as scoring measure to retrieve the best ones we used the accuracy of the model.

Starting from the **GradientBoostingClassifier**, we set the **criterion** to *friedman_mse*, we fixed the learning rate to 0.3, the **subsample** parameter to 0.8 and we tuned the **number of estimators** first, with values in the range [10,100], **min_samples_split** [20,120] and **max_depth** [3,12]. Once we found the best values for these parameters, we also tuned the **learning_rate** [0.01,1] and the **maximum_features** [7,20] and we tried to evaluate if any other parameter's value could be further tuned.

To tune the other variants, where possible, we kept the same settings as in the model we already described. For the **HistGradientBoostingClassifier**, we set the loss parameter to *categorical_crossentropy* and performed the tuning of the **l2** regularization coefficient [0.01, 0.2]. For the **XGBoost** (Extreme Gradient Boost) model, instead, we tuned the **gamma** regularizer and we set the **objective** function to *multi:softprob*.

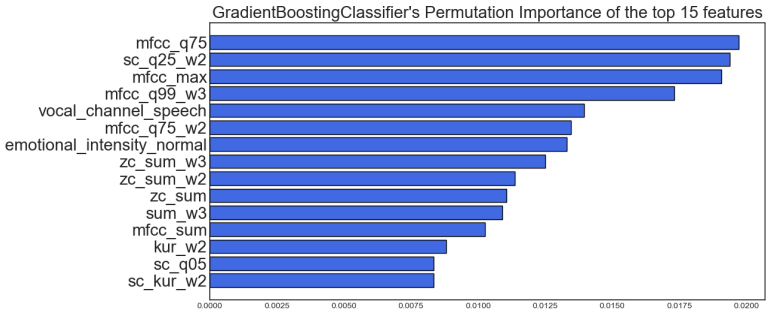


Figure 19: **GradientBoostingClassifier**'s top 15 most Important features

Results The final models we used to classify the records in the *test* set are configured like this: **GradientBoostingClassifier** with learning rate equal to *0.3*, *90* estimators, a depth of *7*, minimum samples per leaf of *8*, a minimum samples to split equal to *40* and *11* maximum features. The other models reflect this configuration, but in the case of **HistGradientBoostingClassifier** we also have a l2 regularization set to *0.01* and we removed the number of estimators, the minimum samples to split and the maximum number of features to select. For **XGBClassifier** we maintained the same l2 regularization coefficient, a *gamma* value equal to 0.2, while also setting, again, a total of 90 estimators.

The results shown in Table 6 show how the three models perform in a very similar way between all the evaluation measures we used to compare them, and they are also comparable with the results obtained with the *Neural Network* models.

Also, in Figure 20 we show the ROC AUC curve for the best model, the **GradientBoostingClassifier**. The performance in terms of the single classes was almost identical to the ANNs, with *angry* predicted with F1-score equal to 0.69 and *fearful* being the worst at only 0.35.

In terms of the features that most impacted the model, we retrieved them through the computation of each feature's permutation importance. These results are shown in Figure 19.

Afterall, the best of the three variations, in our opinion, seems to be the **GradientBoostingClassifier**, the model that didn't include any regularization coefficient to limit overfitting and complexity. This happens probably due to the fact that the configuration we used to build and train the model is complex enough without overly simplifying.

Comparison of Performance measures			
	GradientBoostingClassifier	**HistGradientBoostingClassifier**	**XGBClassifier**
Accuracy	0.473	0.462	0.454
F1-score	0.46	0.46	0.44
Precision	0.47	0.46	0.45
AUC	0.848	0.843	0.848
Recall	0.47	0.47	0.45

Table 6: Gradient Boosting Machines variants' performance measures

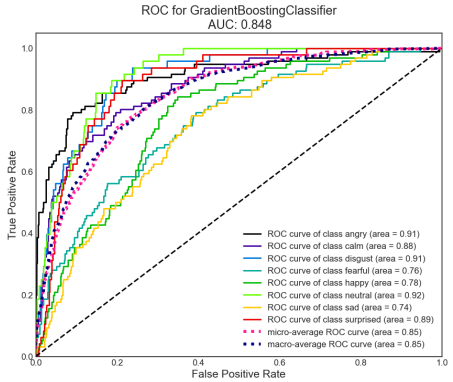


Figure 20: ROC AUC for **GradientBoostingClassifier**, each class OVR

Even in this case, we employed the classifier also on the RFE-reduced dataset, but the results were not aligned with the ones previously displayed.

3.5.1 Regression with Gradient Boosting

Comparison of Performance Metrics			
	GradientBoostingRegressor	**HistGradientBoostingRegressor**	**XGBRegressor**
R-squared	0.948	0.966	0.923
RMSE	0.029	0.024	0.036

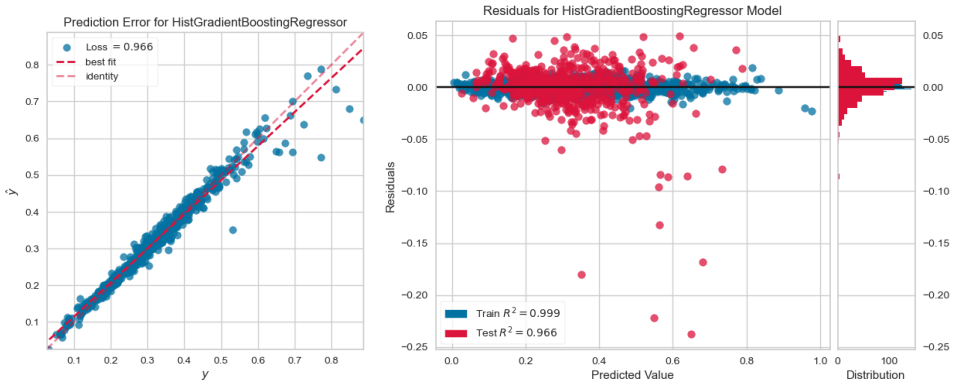
Table 7: Gradient Boosting Regressor variants' performance measures

To conclude this section, we also performed another regression task using the regressors corresponding one-to-one to the Grandient Boosting Machines variants defined and implemented in the previous subsection: the *GradientBoostingRegressor* and *HistGradientBoostingRegressor* from *sklearn* and the *XGBRegressor* from the *xgboost* library.

As target variable, we chose ***zc_sum***, as we did with the regression in the *Support Vector Machines* chapter, in order to compare the results obtained between both of them.

In this case, for each Gradient Boosted variant, we tuned every parameter which was also tuned in their classification counterpart, therefore we chose to not repeat the every step of the *Randomized-SearchCV*. The one important element is that this time we used the *neg_mean_absolute_error* as **scoring**.

Once the tunings were done, for the **GradientBoostingRegressor** we found as best parameters: learning rate=*0.08*, *80* estimators, max depth = *7*, min samples split=*170*,min samples leaf=*6*, *19* maximum features, subsample=*0.75*. As we did for the classifiers, we maintained these parameters for the variants, while removing or adding the same parameters as we did for the classifiers themselves (l2 regularization=*0.1* for the **Histogram-based** variant, while objective = *reg:squarederror* and lambda = *0.01* for the **Extreme** variant).



(a) Prediction Error (b) Residuals distribution

Figure 21: **Histogram-based Gradient Boosting Regressor**

We then trained the models and evaluated their performance, and the best we obtained came from the implementation of the **HistGradientBoostingRegressor**, not only in terms of **R-squared**, but also in terms of **RMSE**.

The Histogram-based Gradient Boosting Regressor's R-squared and the residuals distribution are shown Figure 21.

In terms of the features that most impacted the best regressor, we retrieved them through the computation of each feature's permutation importance. These results are shown in Figure 22.

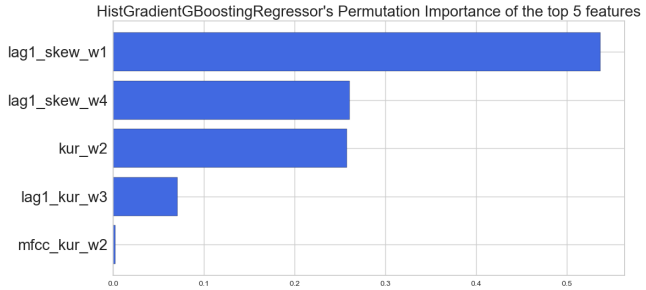


Figure 22: **HistGradientBoostingRegressor**'s top 5 most Important features

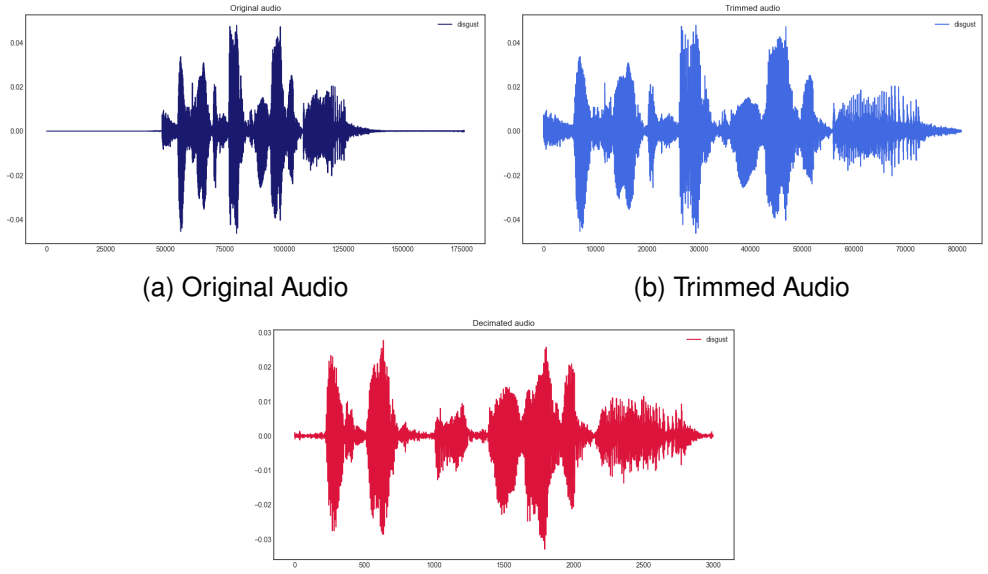
4 Time series

4.1 Data description

Now we switch to the second part of our project. For this part, we had to use the audio time series extracted from the original audio file. Once we have done some data import processes in order to create an initial dataset, we obtained a file with 2452 record (the same of the previous part) and 304305 columns, which represented the timestamps of the audio files, that contained also *NaN* values because the columns' dimensionality is set by the length of the longest time series instance. As before, the 2452 records are divided in 1828 for the *training* set and 624 for the *test* set. In addition, we also had a dataset with the categorical variables (the same provided in the first part), which will be used during our following classification and clustering tasks.

4.1.1 Data understanding and preparation

As soon as we started to work with this dataset, we had to deal with the very **high dimensionality** of the dataset and with the *NaN* values, which for some algorithms are problematic. We began to focus on the literature of audio pre-processing, and we decided to perform two key data preparation operations: The first one involved the employment of a *Trimming* methodology, which helped us to reduce the audio timestamps from 304 305 (Figure 23a) to 221 388 (Figure 23b). This technique helped us to deal with most of the null values. The second process applied in this task was the *Decimation*: this methodology was crucial in obtaining a smaller and task-adequate dataset, in fact, at the end of these preparations, we obtained a dataset with 3000 columns (Figure 23c).



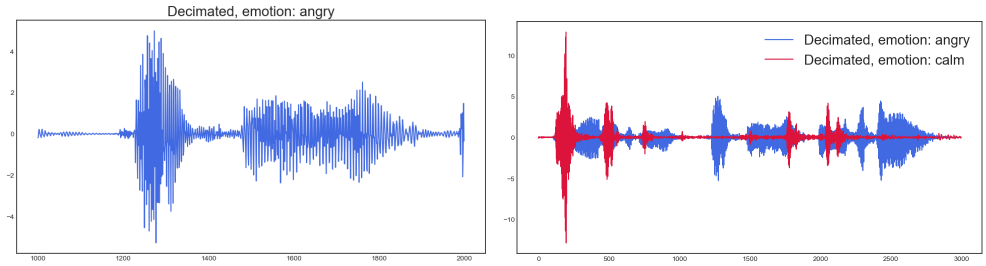
(a) Original Audio

(b) Trimmed Audio

(c) Decimated Audio

Figure 23: Time Series Pre-Processing

After having pre-processed the data, we started by doing an **exploratory analysis**. We tried to visualize a random time series (labeled with *angry* emotion) split in 3 parts (as the transformations on the windows *w1, w2, w3, w4* in the previous part of the project suggested us). We noticed that in the middle part of the time series the absolute values were higher in more timestamps (Figure 24a). Another analysis we performed was the comparison between the timestamps of two random time series associated to different emotions, *angry* and *calm* (Figure 24b).

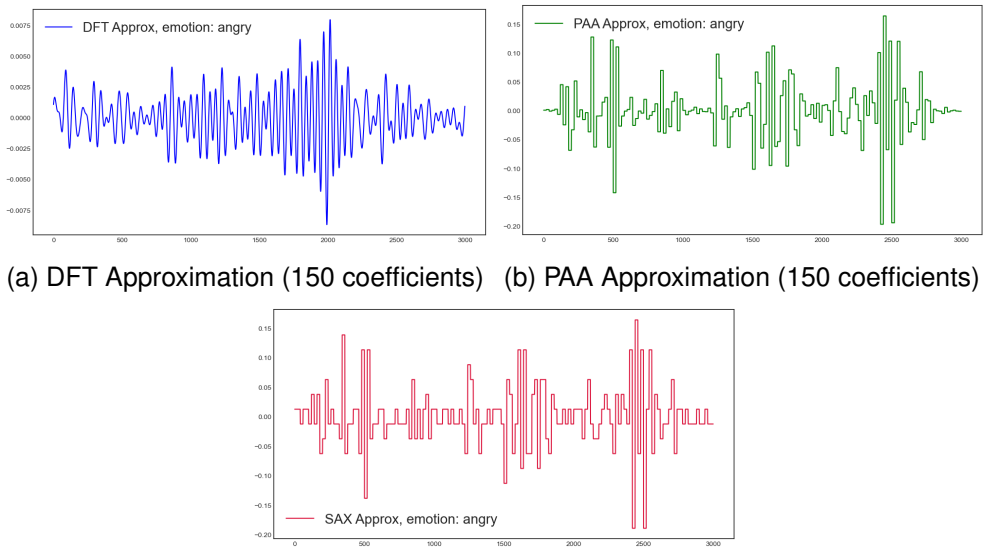


(a) Middle of a Time Series (*Angry*) (b) *Angry* vs *Calm* Time Series

Figure 24: Exploratory Analysis

The differences between *angry* and *calm* time series appear to be meaningful, in our opinion due to the semantics associated to the higher peaks and valleys in the audio data (representing the intensity of the signal). Even if the shown comparison initially led us to not perform any standardization with the aim to maintain such discrepancies, we then decided to apply a **TimeSeriesMeanVariance** scaler from *tslearn* to every time series, to prevent any type of dominance of some signals over others.

4.1.2 Approximation



(a) DFT Approximation (150 coefficients) (b) PAA Approximation (150 coefficients)

(c) SAX Approximation (100 symbols)

Figure 25: Time Series Approximations (*Angry*)

Before moving to Clustering and Classification tasks, we felt that was necessary to employ some Approximation algorithms on the time series data with the aim of reducing even more the dimensionality and to speed-up the calculation time. The approximations performed were:

1. **DFT**, with 150 coefficients (Figure 25a)
2. **PAA**, with 150 coefficients (Figure 25b)
3. **SAX**, with 100 symbols (Figure 25c)

The choice of coefficients was done by looking at the similarity with the original (decimated) audio. Depending on the the task to carry out, we will select different datasets (decimated/approximated) in order to achieve the best results.

4.2 Clustering

In this section we will explain all the different methodologies we applied in order to obtain the best possible results for the unsupervised task of clustering. As we did for the classifications tasks in the tabular-data analyses' section, we first shuffled the time series data in order to cut out all the potential biases that could derive from the collection and positioning of the data, since it was grouped by the categorical variables' values. As explained previously, in the following sections we will not only make use of the decimated dataset, but also the approximated ones (*DFT*, *PAA* and *SAX*).

In both of the cases explained in the dedicated subsections, we performed both a partitional **K-Means** clustering and a **hierarchical** clustering. Also, as reference for the decisions about the clusters to find, we kept in consideration the *emotion* classes, so we focalized our efforts on finding 8 clusters (one for each class). Since clustering the whole set of *training* records (1828) would have been too computationally expensive, we chose to select only half of them.

We chose to divide the clustering analysis in two subtasks:

- Clustering on exclusively the extracted coefficients from the approximation algorithms
- Clustering on the decimated and then approximated dataset representations with their original dimensionality (3000 timestamps)

As algorithms and libraries we used ***KMeans*** from *sklearn* for the coefficient-only clustering, ***TimeSeriesKmeans*** from *tslearn* for the full-represented approximations of the time series, and for both we deployed the ***AgglomerativeClustering*** from the *sklearn* library. For the latter, we also made use of the *scipy.spatial.distance* and *scipy.cluster.hierarchy* libraries to represent the resulting dendrograms.

4.2.1 Coefficients-Only

We tried applying clustering algorithms to the coefficients-only dataset that were extracted from the approximations. In these cases, we thought it was more sensible to use the non-timeseries-dedicated libraries, since we only clustered the coefficients' values.

In order to not congest the report with all the performance measures we computed for every set of coefficients combined with every variation we tried, we will only discuss the best in terms of silhouette score and Fowlkes Mallows score.

KMeans For each of the approximations, we performed a **KMeans** setting 8 as number of clusters to find and using KMeans++ as initialization for the centroids and using *Euclidean* as metric. We also tried the *Dynamic Time Warping* as distance matrix, but we will not show such results since in our opinion it doesn't semantically apply well when only using the coefficients.

We observed that, while some clusterings were better than other, the algorithms for the most part did not find 8 clusters. Therefore, we chose to reduce the number of clusters to 3, which should represent our understanding of the *emotions* grouping: quiet, medium and loud. Even in this case we were not able to find anything useful, so we decided to reset the number of clusters to 8 and analyse the results.

We found that the best KMeans with *Euclidean* distance's results came from the coefficients we obtained from the *Discrete Fourier Transform* (**DFT**) approximation on the time series, with the clustering's silhouette score = 0.8 and a Fowlkes Mallows score = 0.36. While these results appear relatively good, most of the clusters, except cluster 1, contain a very small number of records, some are even singletons. As a side note, the *DFT* coefficients were also the ones on which the KMeans with *DTW* performed the best.

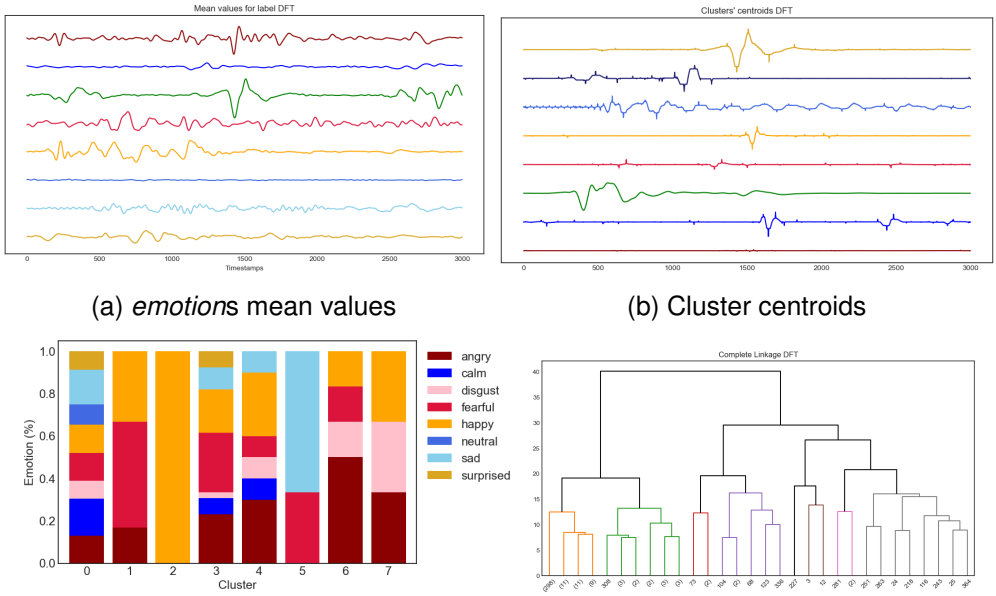
Hierachical Clustering We also performed a hierarchical clustering on the coefficients distinguishing two different linkages: for **Ward**'s method we used the **Euclidean** distance, while for the **Complete** linkage we used, as metric, the *precomputed* **DTW** matrix implemented from the library *dtaidistance*, since it was 200 times faster than the one from *tslearn*.

Also in this case, the best coefficients were the ones found with the **DFT** approximation, but the results, similar as the ones from the *KMeans*, were not very promising. Therefore, we proceeded our analyses with the full representations of the approximated time series.

4.2.2 Approximated Time Series

Here, we tried applying the clustering algorithms we defined above to the the decimated and approximated datasets with 3000 timestamps, but the main difference is that we used the **TimeSeriesKMeans** module from *tslearn*, while **AgglomerativeClustering** remained the same.

This time we wanted to understand if there were any centroids resulting from the clusterings which mirrored the mean values of the known *emotion* classes. In Figure 26a we represented such values using what will be the best approximation found also for this task, the **Discrete Fourier Transform**.



(a) *emotion*s mean values

(b) Cluster centroids

(c) Percentage of *emotion*s by cluster

(d) Dendrogram

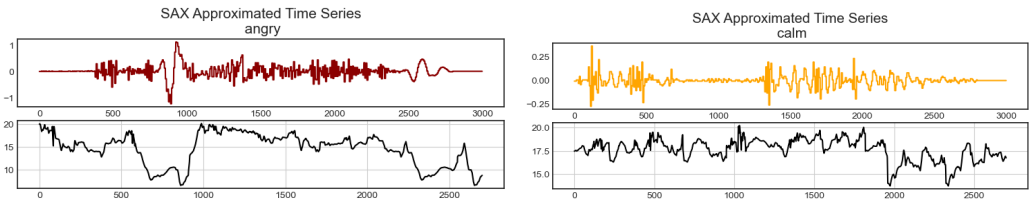
Figure 26: TimeSeriesKMeans and Hierarchical (**DTW**) Clustering results on *DFT* dataset

This algorithm performed quite poorly when applied using as metric the *Euclidean* distance, but gave better results when implemented making use of the **DTW** matrix. In fact, the best results were: clustering's silhouette score = 0.632 and a Fowlkes Mallows score = 0.3.

In this case the clusters were much more consistent in terms of number of records for each, showing progress from the coefficient-only clustering results.

In Figure 26c we show the percentages of each emotion found for every cluster, using red-toned values to show what we thought of being louder *emotions*, conversely blues for the quieter ones. Five clusters are found to have more loud noises, and this should show that there is some sort of separation between them, even if the clustering still doesn not provide enough insightful information.

In Figure 26b, instead, we display the centroids of the clusters we found using the *DFT* approximation and here the colors chosen are not explaining any information, in contrast with what we said before.



(a) MP for *angry* (b) MP for *calm*

Figure 27: Matrix Profile of two time series

Through **hierarchical clustering** with *Complete* Linkage we found relatively the same results as for the *TimeSeriesKMeans*, with the clusters being even less populated. In Figure 26d we represented the **dendrogram** we obtained, with silhouette score = 0.702 and a Fowlkes Mallows score = 0.353.

4.3 Motifs and Discords

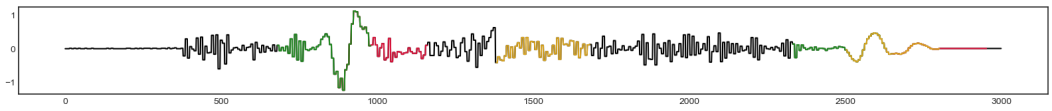
After the clustering analysis, we proceeded with Motifs and Discords detection, since we wanted to understand if there were frequent patterns in the time series and subsegments that, instead, stood apart from the other. To do so, we made use of the library *matrixprofile*, this way we were able to compute the segments that had the lowest and highest distance from the other (respectively motifs and discords). As metric we used the *Euclidean* distance, since the *DTW* seemed trivial for this task. Also, as preprocessing steps, we used the same as we did for all the other tasks explained before.

As for the implementation, we tried to use the decimated-only time series, but we couldn't extrapolate any insights, so we moved on to the SAX approximated dataset and we solved, in fact, this issue. For this specific task, we did not maintain the same number of coefficients as we did for the clustering analysis, but we increased the SAX representation's symbols to 200.

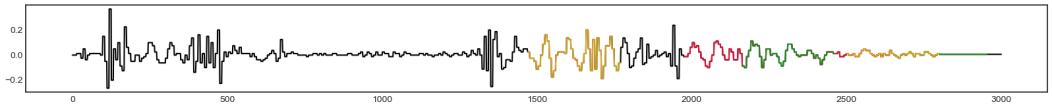
Since we had time series with different patterns based on the *emotion* label they were associated to (and so, they had potentially different motifs and discords), we decided to analyse the matrix profiles of one random time series for each emotion, and to visualize our findings we decided to select only two of them.

The matrix profiles were computed setting a window size of 300 timestamps, which gave us, in our opinion, the best outcomes. In Figure 27 we show the matrix profiles of time series referring to *angry* and *calm emotions*.

Motifs detection As for motif discovery, we set the maximum number of frequent patters to detect to 3, and in Figure 28 we can see the ones we found for each of the selected time series. Even if from the Figures we are not able to blatently expose the repeated patterns, they are clearly revealed when we closely observe them individually, since they repeat themselves but with different amplitudes or they are affected by some noise.



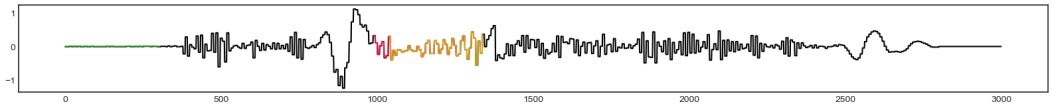
(a) Motifs for *angry*



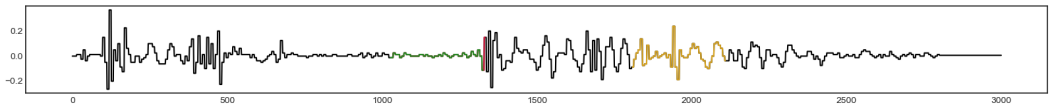
(b) Motifs for *calm*

Figure 28: Top 3 motifs of the two time series

Anomaly detection For this task, we chose to compute a maximum number of 4 top discords and as *Exclusion zone* we chose to remove the 5 nearest subsequences from each anomaly itself (Figure 29). While there are not clear visual discrepancies, we can notice how some of the anomalies detected are stationary fractions of the audio files, which have behaviors detached from segments characterized by large swings between peaks and valleys, in fact they are located immediately before or after these last.



(a) Discords for *angry*



(b) Discords for *calm*

Figure 29: Top 4 anomalies of the two time series

4.4 Classification

Table 8: Time Series Classifications Perfomances Metric

Comparison of Performances Metric						
	kNN_Euclidean	**kNN_DTW**	**CNN**	**LearningShapelets**	**kNN_Shapelets**	**kNN_DecisionTree**
Accuracy	0.79	0.96	0.94	0.92	0.94	0.93
F1-score	speech: 0.81 song: 0.79	speech: 0.97 song: 0.96	speech: 0.94 song: 0.93	speech: 0.93 song: 0.90	speech: 0.95 song: 0.94	speech: 0.94 song: 0.92
Precision	speech: 0.90 song: 0.71	speech: 0.99 song: 0.93	speech: 0.97 song: 0.90	speech: 0.91 song: 0.93	speech: 0.97 song: 0.92	speech: 0.94 song: 0.93
Recall	speech: 0.73 song: 0.89	speech: 0.95 song: 0.98	speech: 0.92 song: 0.96	speech: 0.95 song: 0.88	speech: 0.94 song: 0.95	speech: 0.95 song: 0.92

Now we move to the time series classification task. Here, our classification task, was to predict the binary variable *vocal_channel*. Before implementing all the methods, we tried some quick classifications with the various datasets extracted before in order to evaluate if the performances were good even with the approximations' coefficients, and so with less columns. However, the results were not so good (Figure 30), so we decided to perfom this task with the **only-decimated** version of the dataset.

```
Accuracy 0.5592948717948718
F1-score [0.57496136 0.54242928]
              precision    recall  f1-score   support

           0       0.65      0.52      0.57       360
           1       0.48      0.62      0.54       264

    accuracy                           0.56       624
   macro avg       0.57      0.57      0.56       624
weighted avg       0.58      0.56      0.56       624
```

Figure 30: An example of a classification with **SAX** Approximation

4.4.1 Time Series K-Nearest Neighbours and Convolutional Neural Networks

Let's illustrate now the results obtained by implementing 2 different classifiers: **Time Series K-Nearest Neighbours** and **Convolutional Neural Networks**. As we can see in the results' table (Figure 8) the kNN variant with DTW distance and the CNN are the best algorithms compared to all the methods we tried.

Time Series K-Nearest Neighbours In order to try to predict *vocal_channel* variable with KNN, we chose to adopt the time-series' KNN dedicated library: *KNeighborsTimeSeriesClassifier* from *tslearn*. Initially, we performed an hyperparameter tuning by implementing a *Cross-Validation* and a *RandomizedSearch*. We tuned the *n_neighbours* parameter in two steps: the first time by setting as fixed parameter *metric = Euclidean* and the second time *metric = DTW*. Regarding the Euclidean version, we discovered that the best *n_neighbours* was 3, while for the DTW was 5. During the test phase, we reached an accuracy of 0.79 (Figure 31a) for the Euclidean metric and an accuracy of 0.96 for the DTW metric (Figure 31b). These results are consistent with the literature, where DTW obtain higher results, even if the calculation time was dramatically higher.

```
Accuracy 0.7980769230769231
F1-score [0.80733945 0.78787879]
              precision    recall  f1-score   support

           0       0.90      0.73      0.81       360
           1       0.71      0.89      0.79       264

    accuracy                           0.80       624
   macro avg       0.80      0.81      0.80       624
weighted avg       0.82      0.80      0.80       624
```

(a) *Euclidean* KNN

```
Accuracy 0.9631410256410257
F1-score [0.96737589 0.95764273]
              precision    recall  f1-score   support

           0       0.99      0.95      0.97       360
           1       0.93      0.98      0.96       264

    accuracy                           0.96       624
   macro avg       0.96      0.97      0.96       624
weighted avg       0.96      0.96      0.96       624
```

(b) *DTW* KNN

Figure 31: **KNN** algorithms

Convolutional Neural Network We utilized the CNN as an additional classifier. To begin, we constructed the convolutional structure, consisting of three 1-dimensional convolutional layers. The Rectified Linear Unit (ReLU) was employed as the activation function, and each layer had 16, 32, and 64 output filters, respectively. Regarding the Kernel size, we established it as 8, 5, and 3. Subsequently, we incorporated a 1-dimensional Global Average Pooling layer and concluded the architecture with a Dense Layer employing the Sigmoid activation function. Throughout the training process, we aimed to optimize accuracy as the metric. For the loss function, we utilized sparse categorical cross-entropy, and an adaptive stochastic gradient descent optimizer: Adam. The learning rate for the optimizer remained constant at 0.001 for all epochs, which we set to 5.

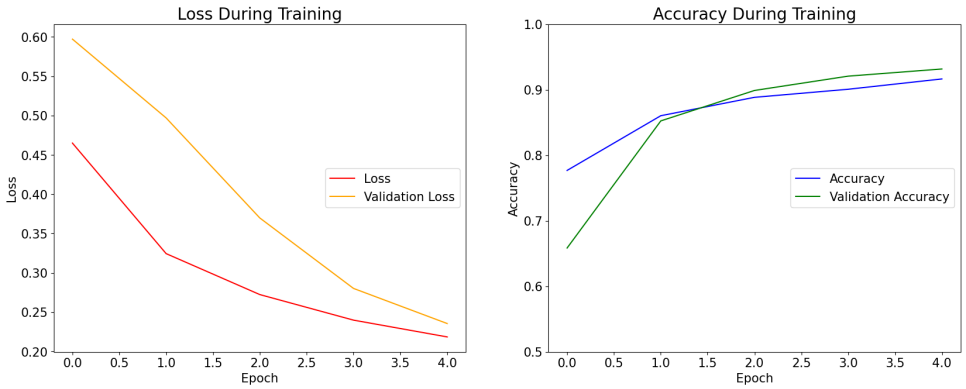


Figure 32: Loss and Accuracy during training

4.4.2 Shapelets Learning Approach

Another classification method that we performed was Decision Tree and K-NN ran on the calculated shapelets. In order to calculate the shapelets we used the learning-approach provided by the library *LearningShapelets* from *tslearn*. To derive the number and size of the shapelets we applied the *Grabocka* heuristic. On a 1828 timeseries dataset with 3000 time stamps and the 2 classes (*speech* and *song*) the heuristic identified as best parameters 6 shapelets with 300 time stamps. With respect to learning parameter, the optimizer was the Stochastic Gradient Descent (Adam) and the weight regularizer we selected was 1%. As batch_size we selected 1 and the number of epochs that we decide to set was 50.

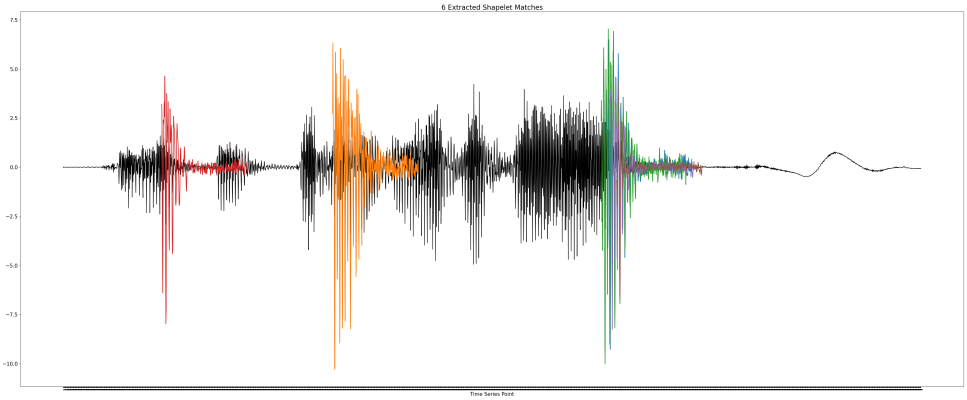


Figure 33: Shapelets retrieved

As we can see, the shapelets represent different points of the time series. The interesting thing is that the shapelets are across 3 different parts of the timestamps space, i.e at the beginning, in the middle and and the end. It suggests us that the timestamps in that specific point are more relevant.

After that, we transformed the *train* and the *test* set on the corresponding shapelet-transformed space using the model just trained and fitted and we ran an instance of the Decision Tree by setting the parameter *max_depth, min_samples_split and min_samples_leaf* respectively to 8, 200 and 35, obtaining an accuracy of 0.93. Another classifier we tried with the shapelets tranformation was K-NN, with 14 as *n_neighbours*. Here the accuracy was 0.94 (Table 8).

5 Explainability

In this section we will try some Explainability methods in order to understand the behavior of one of the classifiers we implemented in the previous chapters. We selected the Neural Network, due to the fact it is the most "black-boxed" between the classification algorithms we implemented. Here, we wanted to test how such classifier performed on the variable *vocal_channel*, so we conformed all the steps defined in the relative section (preprocessing and tuning) using as dataset the one obtained with a Univariate Feature Selection feature reduction applied to *vocal_channel*. Therefore, we employed a binary ANN classification on such target variable.

5.1 Prediction of a Song and a Speech record

5.1.1 Local Interpretable Model-Agnostic Explanations

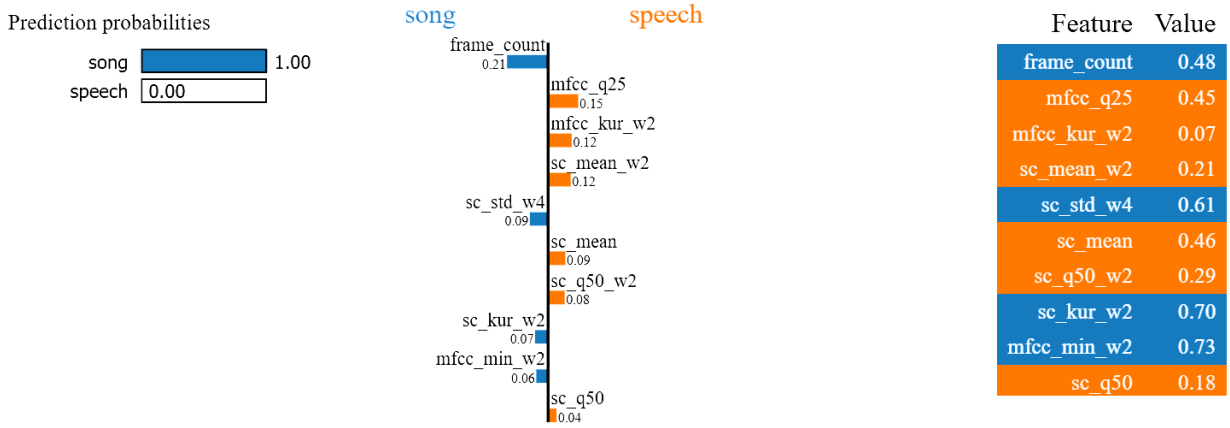


Figure 34: Local Interpretable Model-Agnostic Explanations for a record ***song***

The first technique we applied was LIME. It was initially ran on an instance of our dataset which had as value ***song***. The results of the prediction of the classifier were perfectly correct in this case. As we can see in Figure 34, this methodology gave us a feature importance table, that illustrate which is the most important features that "pushed" our prediction to one of the binary values. In this case, the most important feature useful to determine the value *song* is **frame_count**.

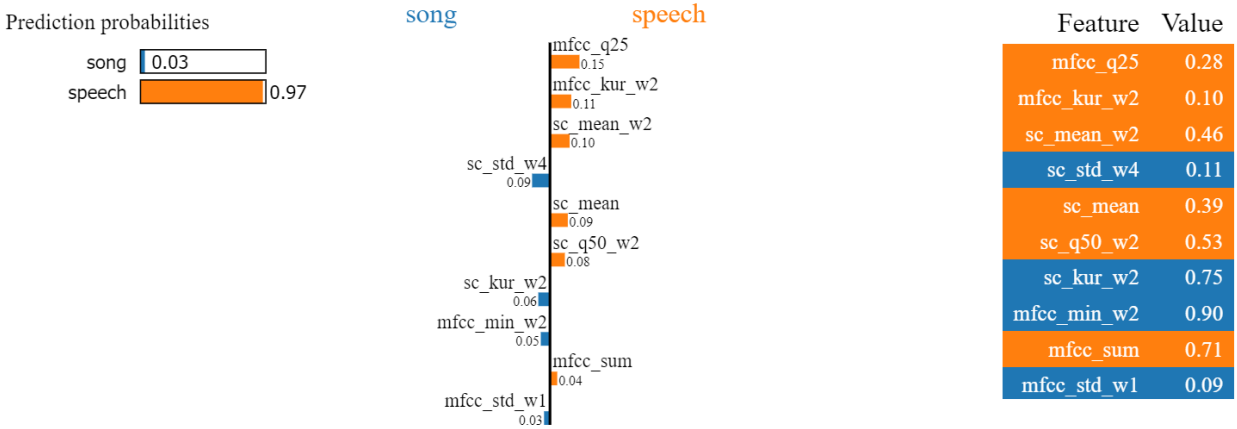


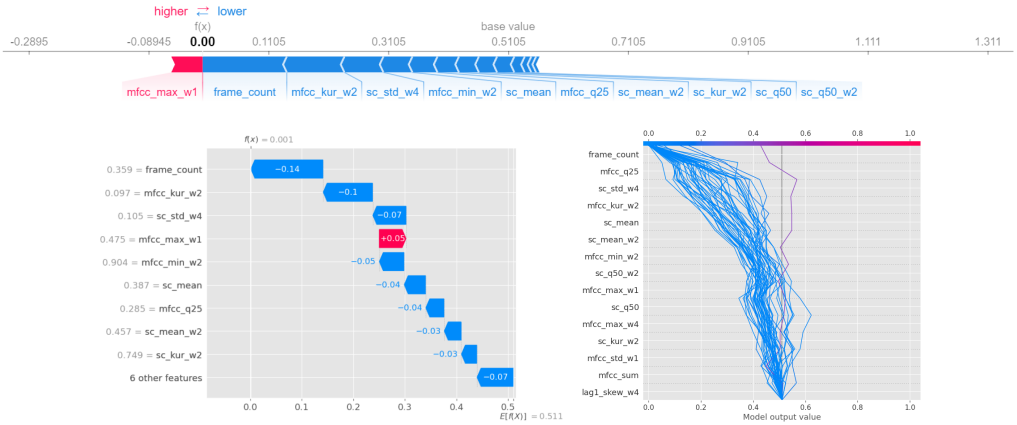
Figure 35: Local Interpretable Model-Agnostic Explanations for a record ***speech***

Next, we also ran the LIME method on an instance with value ***speech*** (Figure 35). For this record, the heaviest variable in terms of importance was **mfcc_q25**

It is nice to notice that *frame_count* and the multiple *mfcc*-derived features were also present during the permutation of the feature importance resulting from the *Random Forest* classifier. It seems that this type of features are really important, independently from the classification task.

5.1.2 Shapley Additive Explanations

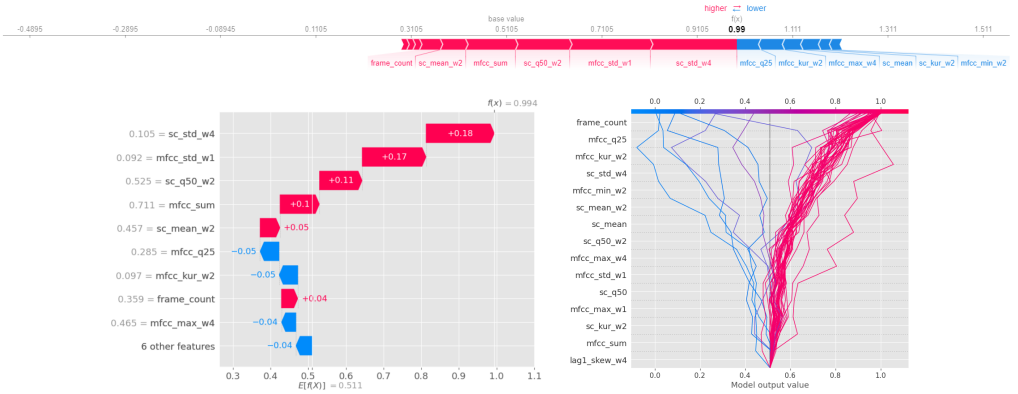
To ensure the explanations we found with the *LIME* method were trustable and consistent with other methods, we tried also employing the *SHAP* explainer, which, as can be seen in Figures 36 and 37, gave somewhat incoherent results with the ones obtained before.



(a) Single *song* record (b) First 50 *song* records

Figure 36: **SHAP** song

In fact, we can notice how, while *frame_count* remains the most important predictor for the class *song*, instead, this time the features *mfcc_kur_w2, mfcc_q25, sc_mean* drive the prediction to the class *speech*. These is in contrast with what we found using the *LIME* explainer, where these features weighted in favour of *song* predictions.



(a) Single *speech* record (b) First 50 *speech* records

Figure 37: **SHAP** speech

In Figures 36b and 37b, we show the **SHAP** decision plots for the first 50 records with label *song* and *speech* respectively. From these plots, we can conclude that *frame_count* remains the most important feature and while the *song* records are predicted correctly in most cases, the *speech* ones are more subject to misclassifications.