Urban Sound Classification with Support Vector Machines

Person codes: 10783942, 10561598, 10500756, 10583734, 10472602

April 2021

1 Introduction

This report describes the implementation of a classifier, which is capable of predicting different types of sound events occurring in audio excerpts. In particular, the goal is to classify common urban environment sounds such as a dog bark, a jackhammer sound or a car horn.
The work was divided into several parts in a modular fashion, and this report is organized as follows. Section 2 presents an overview of the dataset and shows the results of the preliminary studies of the dataset. Section 3.1 illustrates the processing of the audio excerpts, which are used to compute several features. The features are then stored inside a JSON file. Section 3.2 describes the training and validation procedure. Our classifier is implemented as a support vector machine (SVM) with 10-fold cross validation. In Section 3.3, we outline the adoption of parallelization procedures, which decrease dramatically the time needed for the feature computation, training and validation. Section 4 presents the obtained results. In section 5, we point out possible shortcomings and future improvements of our model.

2 Dataset overview and analysis

The dataset used for the Homework is the **Urbansound 8k** dataset, which has been proposed in [1]. It consists of 8732 labeled sound excerpts with short duration ($\leq 4s$) of urban sounds, and is subdivided into 10 folds. Each fold contains excerpts from 10 different classes:

1. Air conditioner
2. Car horn
3. Children playing
4. Dog bark
5. Drilling
6. Engine idling
7. Gunshot
8. Jackhammer
9. Siren
10. Street music

As a first step, the dataset was downloaded and studied in a Python dataset analysis notebook [2]. This dataset comes with a `.csv` metadata file, which contains exhaustive details about every audio excerpt. The `.csv` file was imported in a suitable data structure - a Dataframe from the Pandas library [3]. After that, the sample distributions were analyzed. The results are reported in the histograms in Figure 1 and in Tables 1 - 2, which show the number of samples for each fold and for each class. What catches the eye is that the `car_horn` and `gun_shot` classes are the least represented ones. However, these classes present the lowest intra-class variability, meaning that

less excerpts are sufficient to characterize the entire class. Therefore, it was not deemed necessary to perform data augmentation.

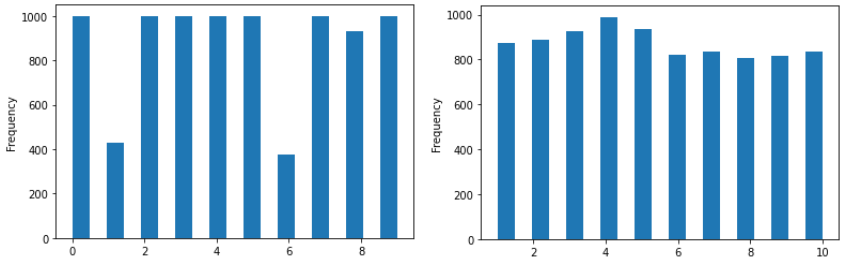


Figure 1: Histogram of audio samples per fold (**left**) and per class (**right**)

Fold n.	1	2	3	4	5	6	7	8	9	10
N. of samples	873	888	925	990	936	823	838	806	816	837

Table 1: Samples per fold

Class	**N. of samples**
air_conditioner	1000
car_horn	429
children_playing	1000
dog_bark	1000
drilling	1000
engine_idling	1000
gun_shot	374
jackhammer	1000
siren	929
street_music	1000

Table 2: Samples per class

3 Method

To improve code scalability, the implementation was organized into three distinct modules inside three Google Colaboratory [4] notebooks:

1. A dataset analysis notebook[1]
2. A feature computation and feature analysis notebook[2]
3. A support vectore machine (SVM) training and validation notebook[3]

The goal of the first notebook is to extract and plot the necessary metadata from the dataset, which are contained in the complementary `.csv` file. The second notebook computes the features upon which the model can be trained, and stores them in a JSON file. In the third notebook, a support vector machine (SVM) is trained and tested using the aforementioned features. The model is then validated.

[1] `https://github.com/magiwanders/CMLS_HW1/blob/master/src/0_DatasetAnalysis.ipynb`
[2] `https://github.com/magiwanders/CMLS_HW1/blob/master/src/1_FeatureAnalysisAndComputation.ipynb`
[3] `https://github.com/magiwanders/CMLS_HW1/blob/master/src/2_TrainingSVM.ipynb`

3.1 Feature extraction

The feature extraction was implemented in the feature computation and analysis Python notebook. For each audio excerpt, we extracted the Mel-Frequency Cepstral Coefficients (MFCC) using the `librosa` audio library [5] and the `pandas` data structure [3] library. The choice of MFCCs was motivated by the fact that the samples pertain to classes which present different noisy-harmonic character. Therefore, Mel-Frequency Cepstral Coefficients seemed as the best feature that could keep track of such detailed content. To compute the MFCCs, the following steps were followed: the spectrogram of the audio signal was computed by calculating the magnitude of the Short-time Fourier transform (STFT). Then, Mel filters were applied to the spectrogram and the logarithm was calculated. Finally, the Discrete Cosine Transform was applied. Table 3 shows all the parameters that were chosen for the MFCC computation.
After the computation of the MFCC, five feature statistics were extracted for each audio excerpt:

1. Minimum
2. Maximum
3. Mean
4. Median
5. Variance

Each MFCC has 25 coefficients; therefore, the combination of the five statistics resulted in 125 feature parameters extracted for every audio file. As previously mentioned, all the extracted features were then exported into a JSON file, in order to make the training procedure independent from the feature extraction part.

Parameter	**Value**
Sampling Frequency	22050 Hz
Window length	22.3 ms
Hop Size	50%
Mel filters	40
MFCCs	25
Min. frequency	0 Hz
Max. Frequency	22050 Hz

Table 3: Feature extraction parameters

3.2 Training and validation procedure

The training was developed in the SVM training notebook. The first step was to retrieve the Dataframe of the computed features, which had been previously stored in a JSON file named `extracted_features.json`. From here, the folds were divided iteratively in order to perform cross-validation: each fold was used only once as validation set, against the remaining 9 used as training set. For every fold iteration, the validation accuracy for each class and the confusion matrices were computed. Finally, a mean of the accuracies and confusion matrices was derived, in order to provide general performance value for the chosen classifier. The support vector machine was implemented with a `'poly'` kernel and a `OneVsRestClassifier` from the `sklearn` library [6]. The `OneVsRestClassifier` enables multi-class classification by dividing the problem into binary classification tasks between a selected class and the rest of the classes.

3.3 Remarks on performance and parallelization

Performance optimization was a key principle for the entire code base of the homework. Whenever possible, processing was parallelized by exploiting all available processing cores of the host machine, leading up to a ten fold improvement in terms of processing time. This was implemented

with the `p_tqdm` library [7], a wrapper of the `multiprocessing` and `tqdm` libraries [8][9], which also shows a progress bar for parallel computation tasks.
Empirical experimentation showed that the computational overhead of the parallelization process was better diluted for the parallelization of a very small number of very computationally long tasks, as opposed to a big number of relatively short tasks. This is mostly true for machines with a relatively limited number of processing cores, which is a reasonable assumption for most consumer desktops. Were the available number of cores much higher, say 32 or 64, the sheer number of concurrent threads would overweight the computation length in spreading the overhead computational cost. Therefore, it was therefore decided to split the parallelization as early as possible in the computational process, in order to obtain maximum performance. The only downside of this approach is purely aesthetic, and almost ascribable to a sub-optimal implementation of the `p_tqdm` library: since in this particular case all the parallelized tasks are roughly equivalent in required computational time, the progress bar will fill very quickly towards the end of the computation after a prolonged period of apparent inactivity. Although the lack of such feedback is certainly lacking in interaction design, it was judged a minor issue.
The computation thread was split twice: during feature extraction and during SVM training. Feature extraction was split into a maximum of 10 threads, each corresponding to the processing of all the samples within one folder. As mentioned before, this is only computationally convenient for host machines with a limited number of processing cores (theoretically at most 10, since 10 is the number of folders). All our machines satisfied this constraint. As was hinted before, limiting the number of threads would have been counter-productive for machines with much higher core count. The SVM training was similarly split into 10 threads, each for a different step of the cross-validation procedure. A scheme of the parallelization is shown in Figure 2.

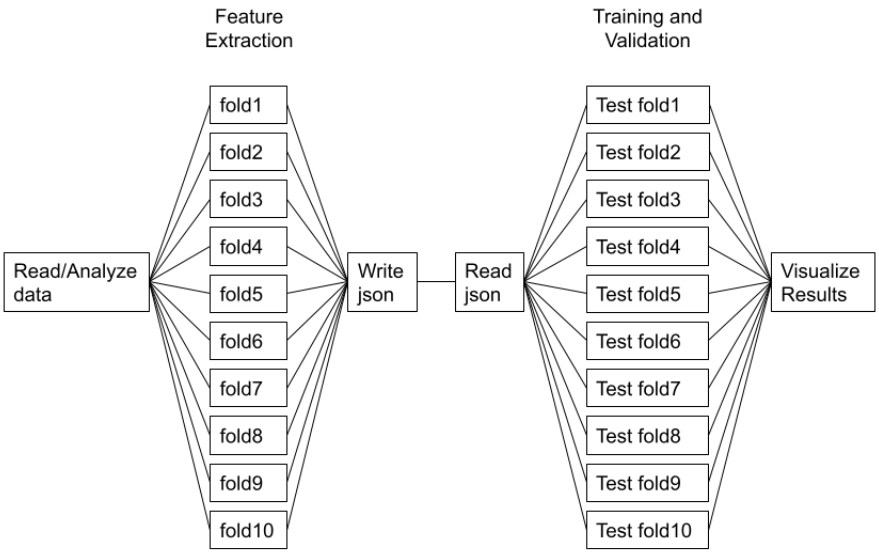


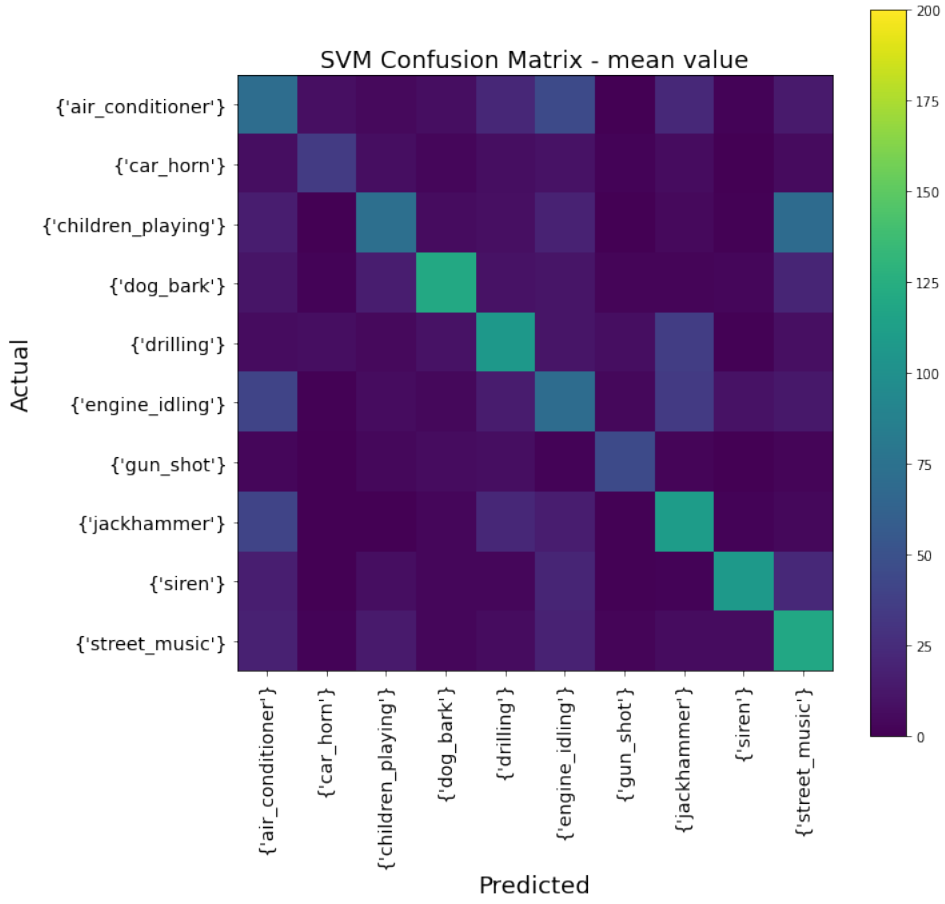
Figure 2: Simplified scheme of the code structure, with emphasis on visualizing parallelization

4 Obtained results

By looking at the confusion matrices and the resulting accuracy of every fold iteration, it can be seen that the model reached an averaged accuracy around 49.3%. By giving a first glance at the mean confusion matrix in Figure 3, it would seem that the classes `car_horn` and `gun_shot` have the worst results. However, it is quite the opposite: these two classes have the highest number of correctly predicted samples (True Positives) and the lowest amount of False Positives and False Negatives. This can be seen by the fact that the rows and columns corresponding to each of

these classes are quite dark. The seemingly dim color associated to the True Positives is simply due to the fact that these classes are the least represented ones inside the dataset. Other classes also reach a fairly good prediction accuracy. However, it can be observed that `engine_idling`, `air_conditioner`, `jackhammer` and `drilling` seem to have low inter-class variability and this leads to wrong separation of these classes.

Figure 3: Final Confusion Matrix



5 Conclusion and possible improvements

The results of the work led us to a number of conclusions.
The first one is that MFCCs are maybe not best suited for the distinction between different types of noisy-like sounds.
Another remark is that the dataset contains both foreground and background sounds. It is reasonable to assume that background sounds makes the classification task harder, because the sound event features are less prominent in these kind of excerpts.
For what concerning the MFCC feature statistics, we chose to use the minimum, maximum, mean, median and variance. It is possible that the computation of second-order statistics (e.g. first derivative, second derivative) may have increased the model accuracy.
A last observation is directed towards our choice of adopting an SVM model. Convolutional neural networks would have probably reached a higher accuracy; however, we chose a more traditional machine learning approach to abide by the program of the course, and to put into practice the concepts shown during class.

References

[1] J. Salamon, C. Jacoby, and J. P. Bello. A dataset and taxonomy for urban sound research. In *22nd ACM International Conference on Multimedia (ACM-MM'14)*, pages 1041–1044, Orlando, FL, USA, Nov. 2014.

[2] Urban sound classification with support vector machines. URL `https://github.com/magiwanders/CMLS_HW1`.

[3] `pandas`: Python data analysis library. URL `https://pandas.pydata.org/`.

[4] Google colaboratory. URL `https://colab.research.google.com`.

[5] `librosa`: audio and music processing in python. URL `https://librosa.org/`.

[6] `sklearn`: Machine learning in python. URL `https://scikit-learn.org`.

[7] `p_tdqm`: parallel processing with progress bars. URL `https://github.com/swansonk14/p_tqdm`.

[8] `pathos`: parallel graph management and execution in heterogeneous computing. URL `https://github.com/uqfoundation/pathos`.

[9] `tdqm`: a fast, extensible progress bar for python and cli. URL `https://github.com/tqdm/tqdm`.