By: Marwan Harajli
Subject: Springboard Capstone Project 1 Final Report
Last Updated: 01/11/2019

Urban Noise Classification

1- Introduction:

In this report we assess the feasibility of using supervised learning to classify urban sounds. More specifically, given labeled .wav files each representing one of ten sounds/classes (Air conditioner, car horn, children playing, dog barking, drilling, engine idling, gun shot, jackhammer, siren, and street music), we investigate training classification algorithms that can label unseen data.

This project presents itself as an educational opportunity in tackling unstructured data. Since the data is unstructured, useful features need to be found and extracted. Moreover, due to the nature of the unstructured data (being audio data), the project also presents an opportunity to delve into audio processing. The latter has a wide array of applications that range from speech processing to recommending music to radio stations. With that, the project yields the potential to be a good data science exercise that also opens the door to the exciting field of audio processing.

In what follows we discuss the obstacles met in obtaining and handling the data (the sounds), cleaning the data, and extracting features from the data. Finally, using the extracted features, classification algorithms are trained, and their performance assessed.

2- The Data:

This project uses the UrbanSound8k dataset that can be found and downloaded through this link. The data consists of 8732 labeled (with one of 10 labels) .wav files with durations <= 4 seconds. The data is provided in 10 folds, with a recommendation to keep the files in their respective folds when performing cross validation.

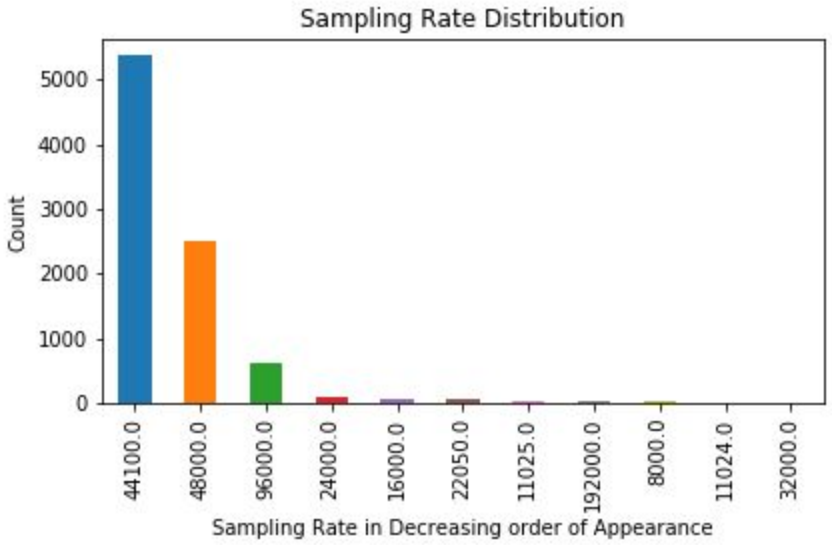
The reason behind this recommendation comes from the fact that some sound files are different slices of the same audio file. This means means that 2 siren sound clips for example, could be 2 different segments (that do not overlap) form one original longer audio file. If these two segments were placed in different folds, performance scores could be inaccurately inflated (specifically when one of these two is in a test fold). To avoid this scenario, audio files from the same original source file are placed in the same fold.

3- Data Wrangling and Cleaning:

Having .wav files is not very useful when it comes to training classification algorithms. To start with, we need to convert these .wav files into numpy array representations. While there are many libraries that are capable of converting .wav files to numpy arrays, not all of them have the capability of reading 24 bit depth audio (which is the case with some of the audio in this dataset). We consequently used the *read* function of the soundfile library.

Given a .wav file, the *read* function returns a numpy representation of the sound along with an integer representing the sampling rate (SR). The sampling rate denotes the number of samples the array provides per second. Consequently, the duration of the audio file can be found as: SR x LengthOfArray. In Figure 1, we show the distribution of sampling rate in the dataset.

Figure 1: Sampling Rate Distribution



As seen in Figure 1, sampling rates vary over the data. Note the sampling rates on the horizontal axis are listed in order of appearance in the data rather than numerical value.

An array representation of a sound with a higher sampling rate involves using longer arrays. It can thus be beneficial to use smaller sampling rates to reduce memory and processing demands. Using very small sampling rates however is also unfavorable as the arrays become too coarse to well represent the sounds. Finally, making all the arrays have the same sampling rate adds a level of homogeneity that is favorable.

For the purposes mentioned above, the librosa library is used to resample every loaded array into a 22050 SR representation using the *resample* function.

The read_audio function that can be found in the urbanNoises module, performs all the steps mentioned above. Given a file name, the function reads the file as a numpy array, resamples it to 22050 SR representation, and changes it to mono if it is stereo.

Figure 2 shows the variation in duration over the sound files. The majority of the files have 4 s duration with a small number having smaller durations. This instigates the question regarding the possibility of deleting these files with smaller duration to avoid arrays of different lengths. To answer that question we check that no one class is majorly affected by doing so. Consequently, we check the proportion of sounds that have duration less than 4 seconds for every individual class.

Taking a closer look at Figure 3, we see that a relatively large proportion of audio files for dog barks, car horns and gun shots have duration less than 4 seconds. With that, deleting all audios with durations different than 4.0 seconds becomes problematic.

Figure 2: Duration Distribution

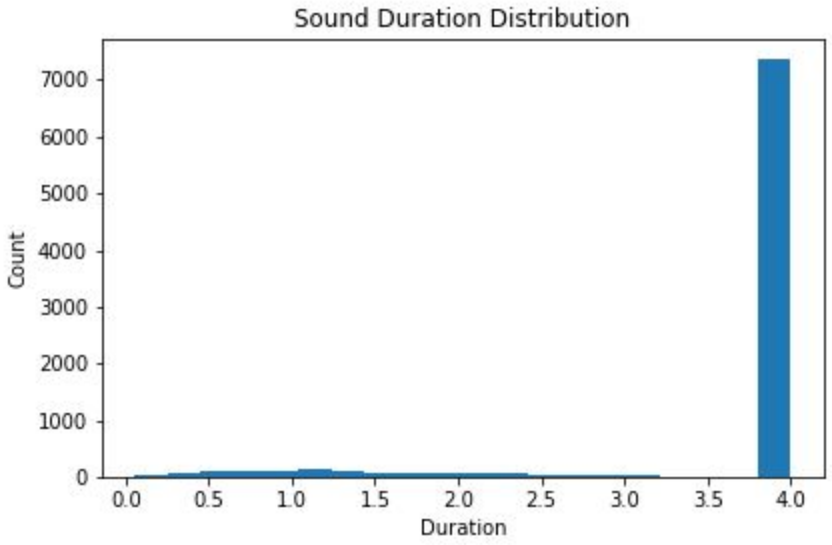
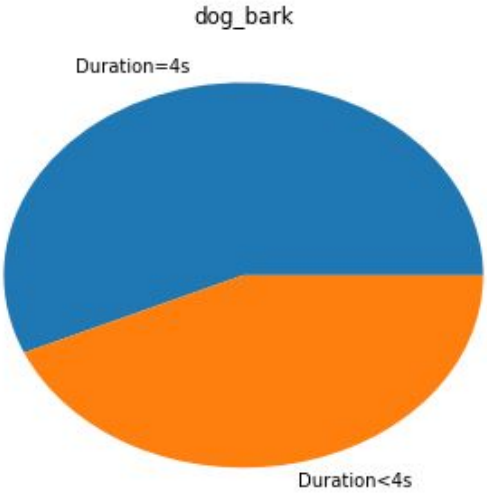


Figure 3: Proportions of sounds with duration less than 4 seconds for dog_bark, car_horn, and gun_shot



In summary, every .wav file is transformed into a mono numpy array representation with sampling rate=22050. Deleting sounds with duration less than 4s is infeasible since that eliminates large portions of

entire classes. With these numpy arrays, we are now ready to train classification algorithms. As they stand however, these arrays are:

1. Large in size (up to 4*22050 features)
2. Vary in size (depending on the duration of the sound).

In section 4 we look into engineering features that reduce feature dimensionality and are duration independent.

All code generating figures in this section can be found in the urban_noises_data_wrangling notebook.

4- Feature Engineering:

As it stands, every sound is represented as an array. Each sample in the array corresponds to air pressure at a certain time. This can be seen for sounds from different classes in figure 4. As mentioned in section 3, we would like to transform each sound to a lower dimensional space where the transformation itself is duration independent. We look at a few different features we can extract from these arrays.

4.1- Mel-scaled Spectrograms:

Spectrograms explain the frequency content of a sound and its variation with time. It is usually shown as heat plot where the x-axis is time, the y-axis is the frequency, and the color at any (x,y) is the intensity of the y frequency at time x.

Mel-scaled spectrograms show the same information as above while scaling the y-axis with the mel scale (see this for an explanation of the mel-scale). Figure 5 shows the mel-scaled spectrograms for the same sounds we saw in Figure 4. We also note that in figure 5, the intensities are shown on a log scale.

4.2- Mel-frequency Cepstral Coefficients:

If we take the discrete cosine transform of every time bin of a log-scaled mel-spectrogram, and do the same for all time bins, we obtain Mel-Frequency cepstral coefficient. Figure 6 shows these coefficients for the sounds in Figure 4.

4.3- Chroma of a short-time Fourier transform:

Chromagrams show the power distribution of a sound over the 12 pitches (C, C#, D, D#, E, F, F#, G, G#, A, A#, B), and its variation with time. Figure 7 shows the chromagrams for the sounds shown in Figure 4.

4.4- Tonnetz Features:

A link is provided that explains Tonnetz Features. Figure 8 shows the tonnetz features of the sounds shown in Figure 4.

Figure 4: Air Pressure Variation with Time for Various Sounds.

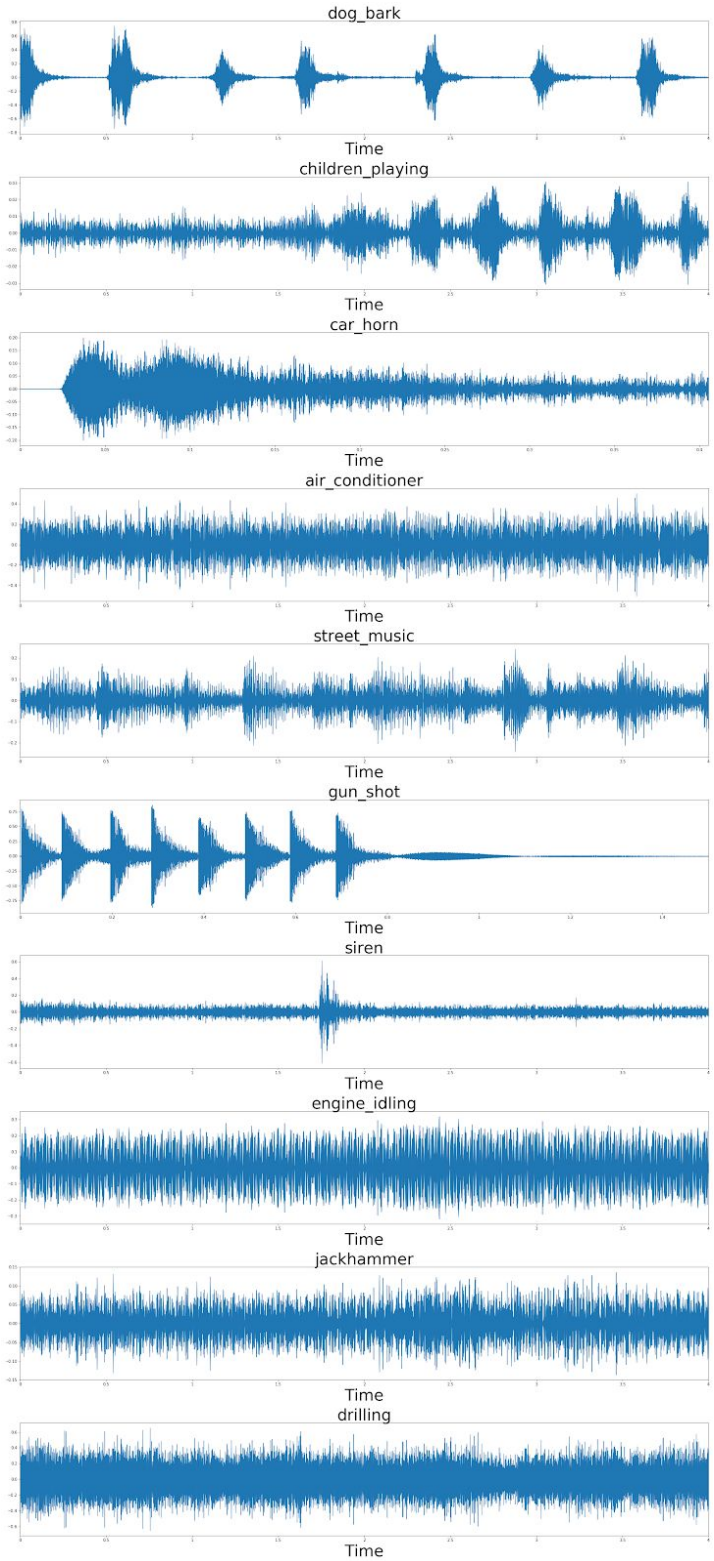


Figure 5: Mel-Spectrograms for Various Sounds.

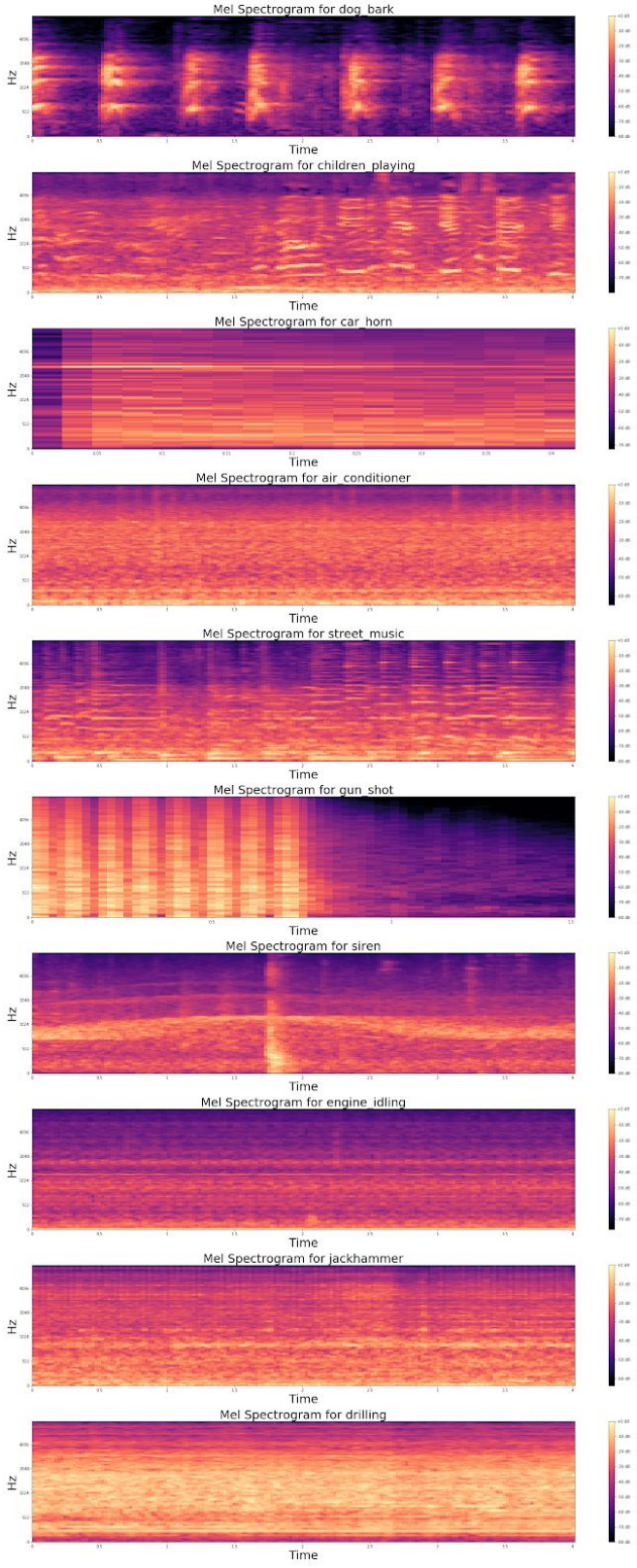


Figure 6: Mel-Frequency Cepstral Coefficients for Various Sounds.

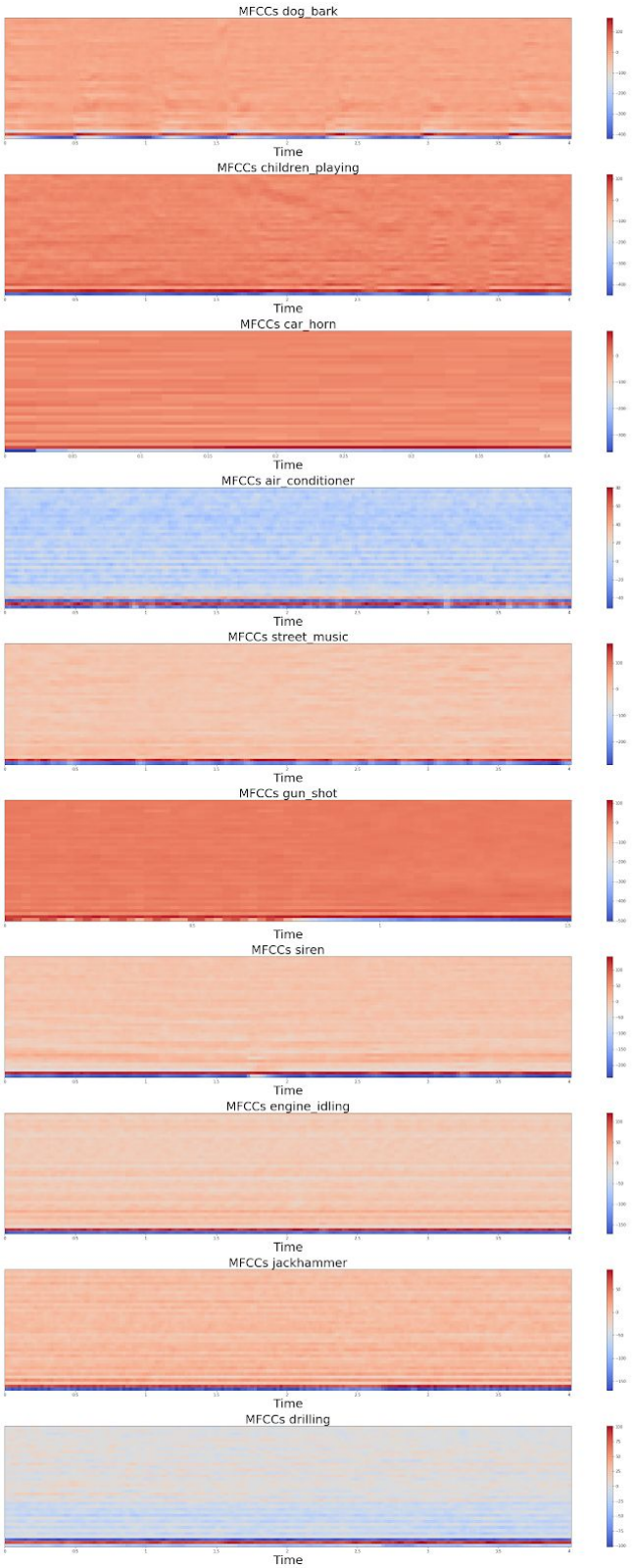


Figure 7: Chromagrams for Various Sounds.

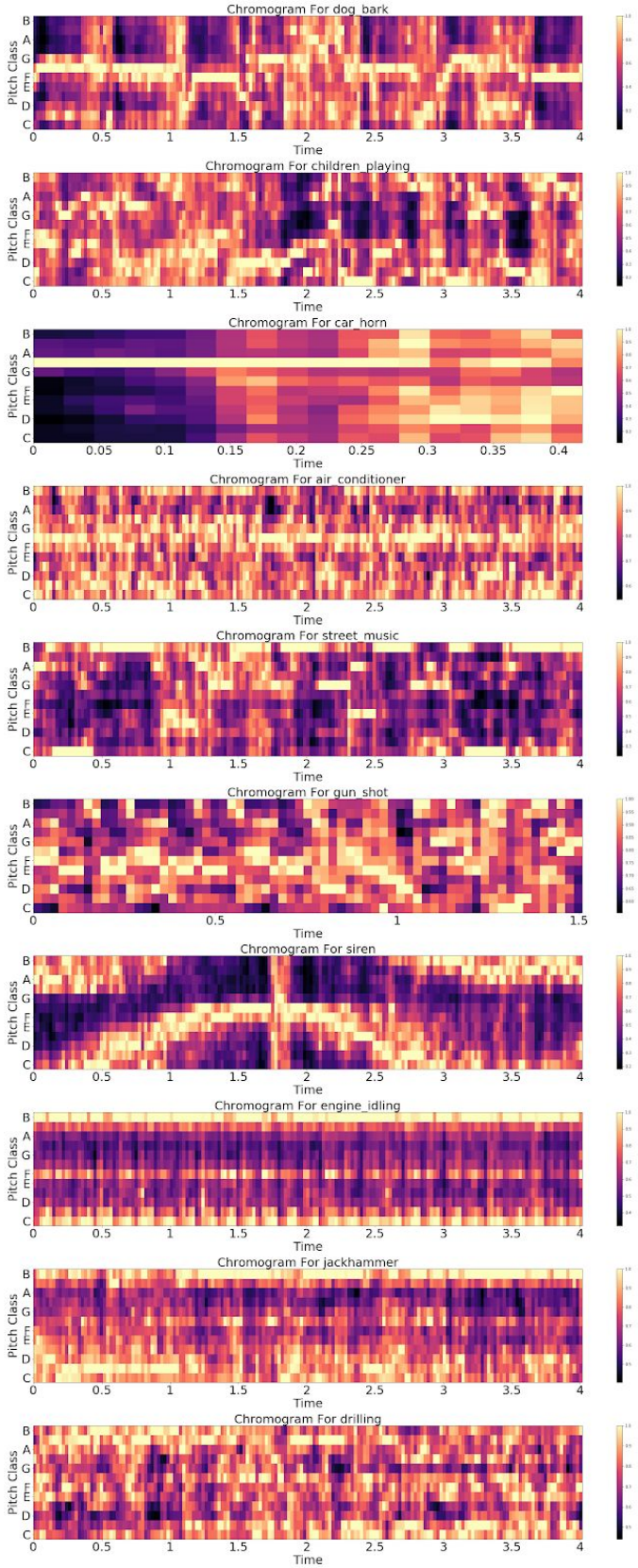
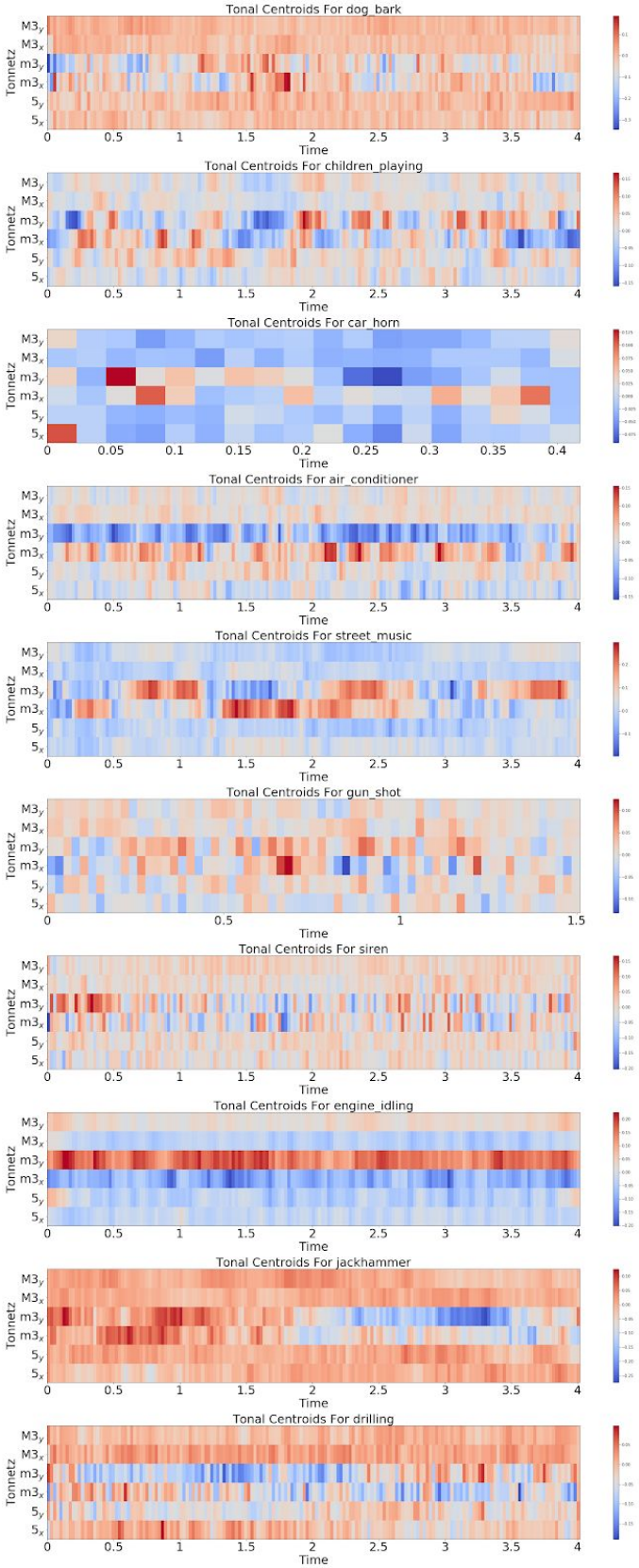


Figure 8: Tonnetz Features for Various Sounds.



4.5- Using these Features:

Above, we describe four features we can extract from the array representation of a sound, namely:

1. Mel-scaled spectrogram.
2. Mel-frequency cepstral coefficients.
3. Chroma of a short-time Fourier transform.
4. Tonnetz coefficients.

While each of these features reduces the dimensionality of a sound, its dimensionality is still dependent of the sound's duration. This can be seen in all the preceding figures where gun_shot features have an x-axis up to 1.5s for example.

To make these features time independent, one possibility is to temporally average. For the melspectrogram, for example, that would mean averaging over the rows.

More formally given features fi (i in [1,2,3,4]) that are transformations that take an array s (the sound) of size n, and return a matrix Mi of size mi*ki (n,mi,ki are non-zero positive integers).

$$f_i : R^n \rightarrow R^{m_i \times k_i}$$
$$s \rightarrow M_i$$

ki is dependent on n (i.e. the duration of the sound), while mi is dependent on the transformation itself but independent of the duration (for example how many mel bins are chosen for the mel-scaled power spectrogram). Temporally averaging each Mi gives an mi*1 vector.

$$f_i^{'} : R^n \rightarrow R^{m_i}$$
$$s \rightarrow M_i^{'}$$

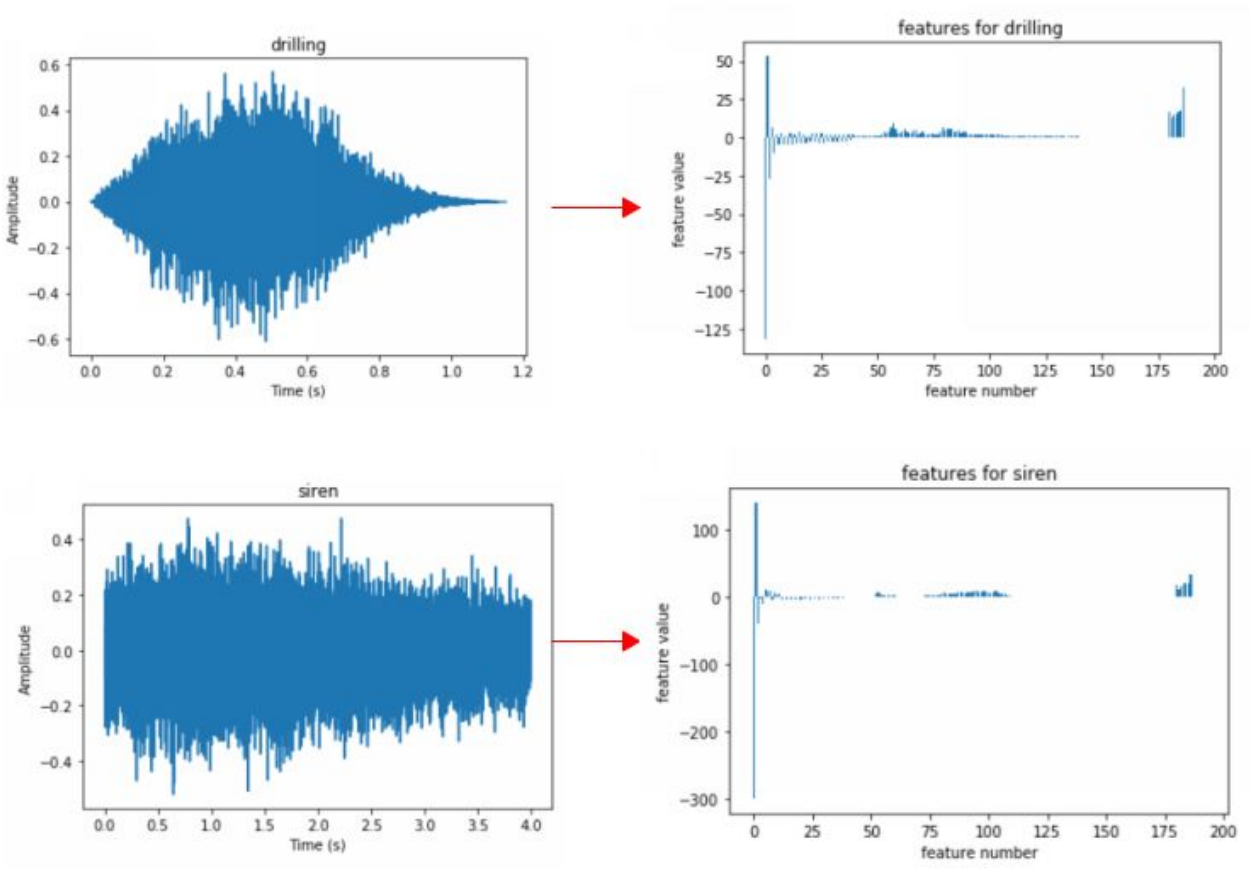
Finally, stacking these 4 $M_i^{'}$ vectors above each other, each sound obtains a feature of length:

$$length = \sum_{i=1}^{4} m_i$$

With this strategy, sounds with different durations produce features of identical lengths. Figure 9 shows the features extracted from two different sounds.

The write_features function in urbanNoises.py extracts the temporally averaged features as described above, and pickles them using the python pickle library. All plots in this section can be found in the FeatureEngineering notebook.

Figure 9: Feature Extraction for a Drilling sound and a Siren sound



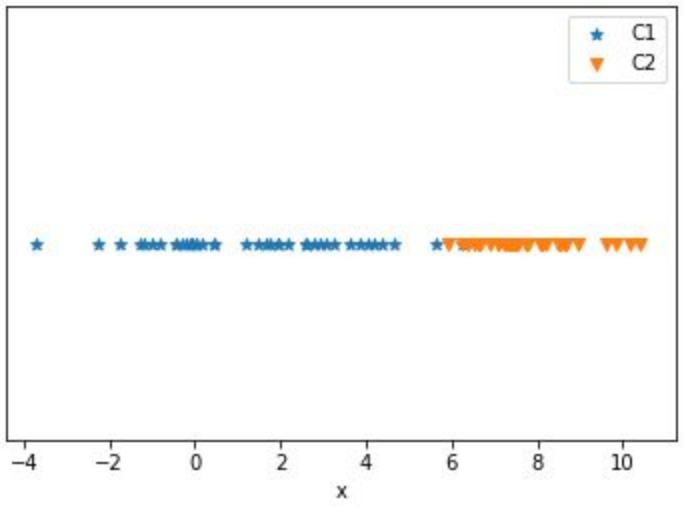
5- Assessment of Features:

In this section we assess the usability of the extracted features. To perform this assessment we propose a probabilistic framework for predicting classes and see how well it performs with the features we have extracted.

5.1 Probabilistic Framework:

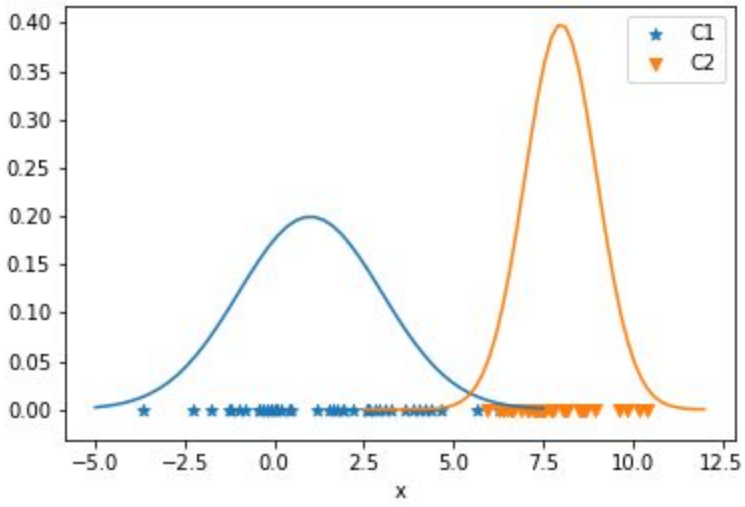
Assume we have a scenario where two classes C1 and C2 exist. These classes are to be predicted using a single feature x. A set of x values and their corresponding classes are given as shown in Figure 10.

Figure 10: Data Obtained



These points are in fact generated assuming each class has its feature distributed according to a distinct normal distribution as shown in Figure 11. C1 has x normally distributed with mean 1 and std 2 while C2 has x normally distributed with mean 8 and std 1.

Figure 11: Distribution of feature for each class

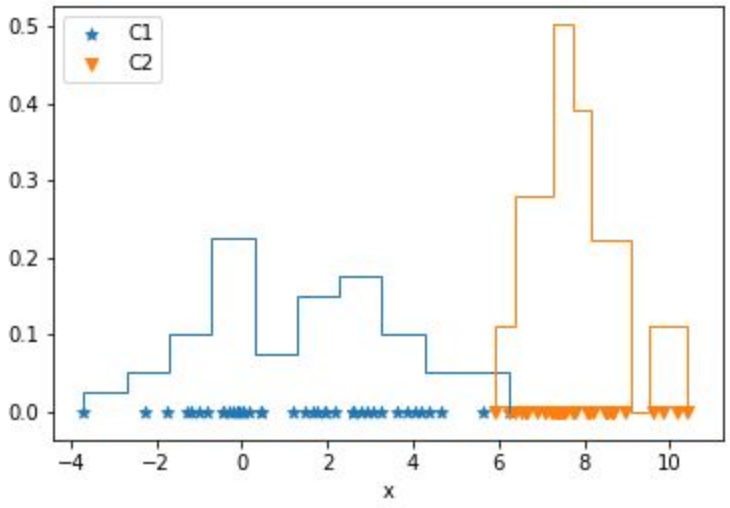


Now given a new observation, say x= 7 we wish to predict what class this observation belongs to. A likelihood estimator for (Ci|Observation) is pdf(observation|Ci) where pdf is short for probability density function. Thus evaluating pdf(observation|C1) and pdf(observation|C2) and comparing them we can conclude the most likely class this observation belongs to; in this case C2 (since pdf(x=7|C1)=0.0022 and pdf(x=7|C2)=0.242).

Up to this point however we have assumed we know the underlying distribution for the feature for each class. This is not true in general (as is the case in the urban noises dataset); rather we need to use the data at hand to produce estimates for pdf(x|Ci).

This can be done using a normed histogram to get an estimate of the pdf as shown in Figure 12. With this estimate, we see that once again x=7 is classified as C2 since the estimate for pdf(x=7|C2) is greater than the estimate for pdf(x=7|C1).

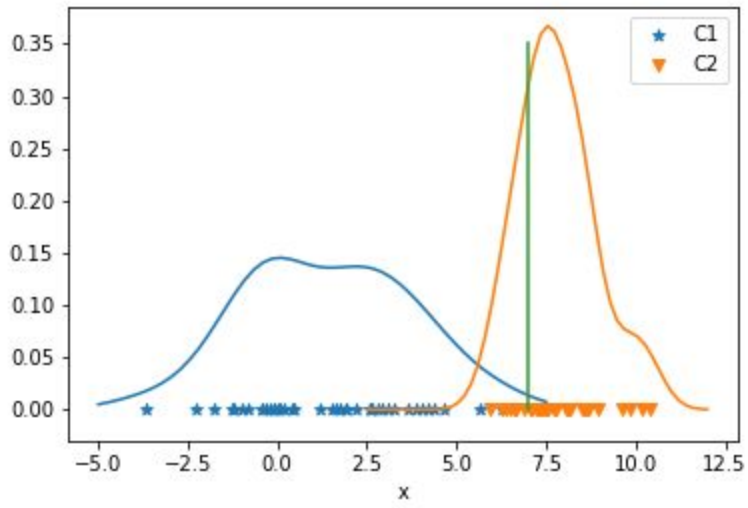
Figure 12: Estimate of Distribution of feature for each class using histograms



Histograms can be computationally tough to generate for high dimensional features however. Consequently we try estimating pdfs using kernel density estimators (using the scipy library with the stats.gauusian_kde method).

Figure 13 shows the estimates for the pdfs using kernel density estimators. The figure also shows the line x=7 to show that the estimate for pdf(x=7|C2) is higher than the estimate for pdf(x=7|C1).

Figure 13: Estimate of Distribution of feature for each class using KDE



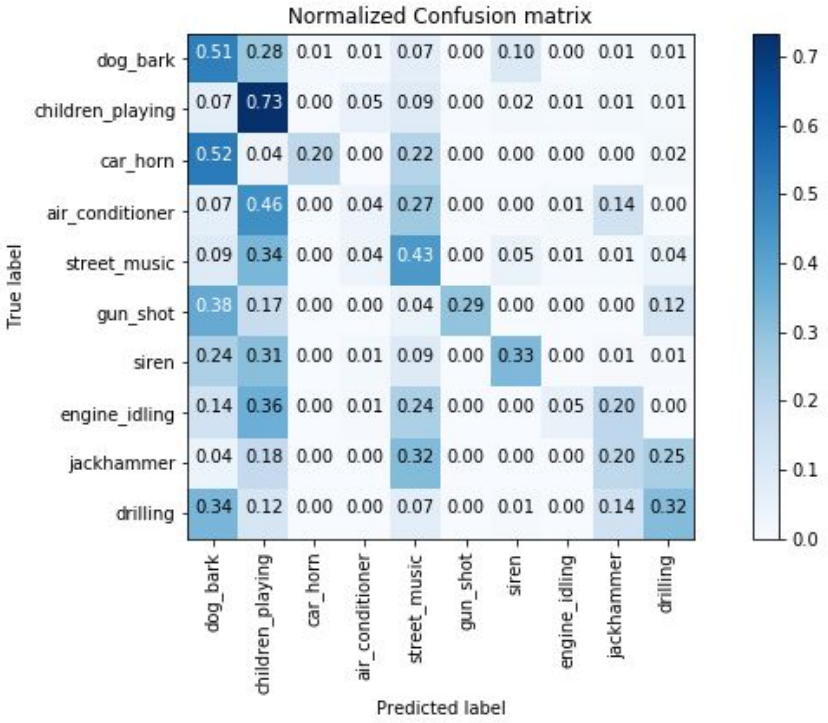
5.2 Application to Sound Extracted Features:

In our dataset, each labeled sound is transformed into a feature vector of 186 dimensions. We split the data into a training set (using 7 of the 10 folds) and a testing set (using 3 of the 10 folds). Using kernel density estimators, we obtain pdf estimates for the features over each class using the training data. With that we can estimate pdf(feature|Ci) for every i and for every feature in the test set. Finally, for each sound in the testing set, the class Ci that maximizes the estimate for pdf(features of sound|Ci) is assigned as the predicted class of the sound.

The above procedure yielded a 31% accuracy rate in predicting the right class for the sounds in the testing set. Figure 14 summarizes the results obtained with a confusion matrix.

We note that while 31% is not a great score, a simple algorithm that is very quickly executed yield results that are better than a coin toss. This could indicate that more sophisticated classification models can do well. We investigate that in the next section.

Figure 14: Confusion Matrix for Probabilistic Classification



6- Sound Classification:

In this section, we train various classification models and compare their performance. In this report we try:

1. Support Vector Machines.
2. K-Nearest Neighbors.
3. Random Forests.
4. Simple Neural Network.

6.1 Hyper-Parameter Tuning:

Some of the algorithms in the next sections involve hyper-parameters that need to be chosen. Hyper parameter tuning is undertaken to chose these parameter. Recalling that our data is pre-folded into 10 folds, standard practice would involve the following:

1. Set one of the folds as a test set and the other 9 as a training set.
2. On the 9 folds, we gridsearch over the parameters and cross-validate over the folds to pick the optimal hyper-parameters. (i.e. for every choice of hyper-parameters we split the 9 folds into testing and training 9 times and assess the chosen hyper-parameters).
3. Using the obtained optimal hyper-parameters, we train the model on the 9 folds and test on the test fold.
4. Repeat this process till each fold has been considered as a test fold.

This process yields the optimal hyper parameters as well as how well the algorithm generalizes. On our dataset however, some classification models can take a long time using a personal computer, consequently we try this variation for those models:

1. Set one of the folds as a test set, and the others as a training set.
2. We take a subset of each of the 9 folds, we gridsearch over the parameters and cross-validate over the reduced folds to pick the optimal hyper-parameters. (i.e. for every choice of hyper-parameters we split the 9 reduced folds into testing and training 9 times and assess the chosen hyper-parameters).
3. Using the obtained optimal hyper-parameters, we train the model on the 9 full folds and test on the test fold.
4. Repeat this process till each fold has been considered as a test fold.

The hyper-parameter tuning and results seen in section 6 can be found in the Hyper Parameter Tuning notebook.

6.2- Support Vector Machines:

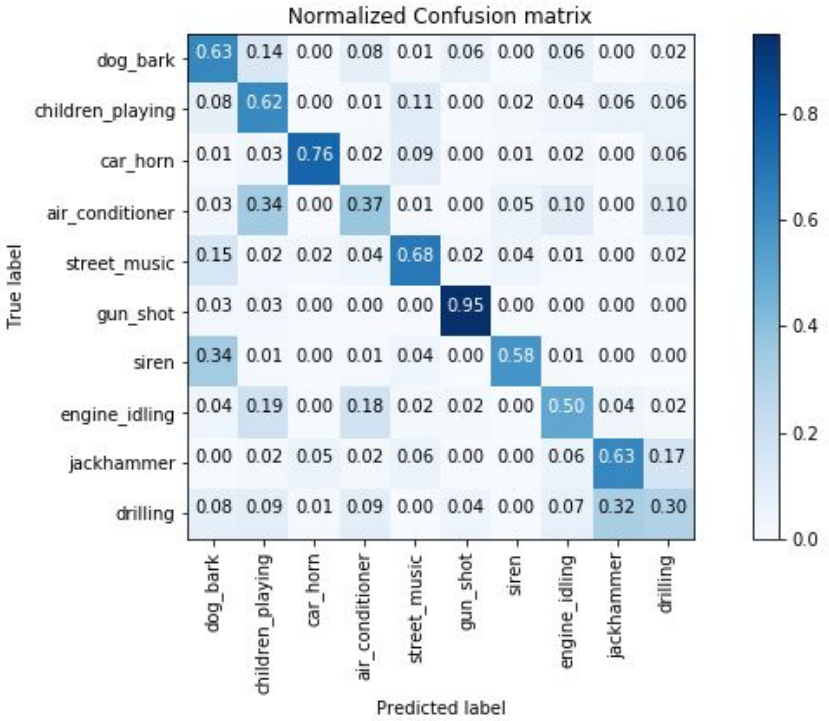
We implement SVM classification using the sklearn's SVC class. There are a few parameters that need to be chosen for this algorithm namely:

- C: Penalty parameter of the error term.
- The choice of kernel: 'linear', 'poly', 'rbf', 'sigmoid', or 'precomputed'.
- Gamma: Kernel coefficient for 'poly', 'rbf', and 'sigmoid'.

Moreover, SVM is sensitive to scaling of the data. We thus use sklearn's MinMaxScaler to scale all training features between -1, and 1.

On average, the accuracy score for SVM is found to be: 0.59. The performance of the model is also further described in the confusion matrix shown in Figure 15. This confusion matrix was obtained taking one of the folds as a test set. As can be seen in Figure 15, the model misclassifies air_conditioner and jackhammer as drilling (which can make sense due to the nature of the noise). The model performs very well in identifying siren sounds.

Figure 15: Confusion Matrix Using SVM.



6.3- K-Nearest Neighbors:

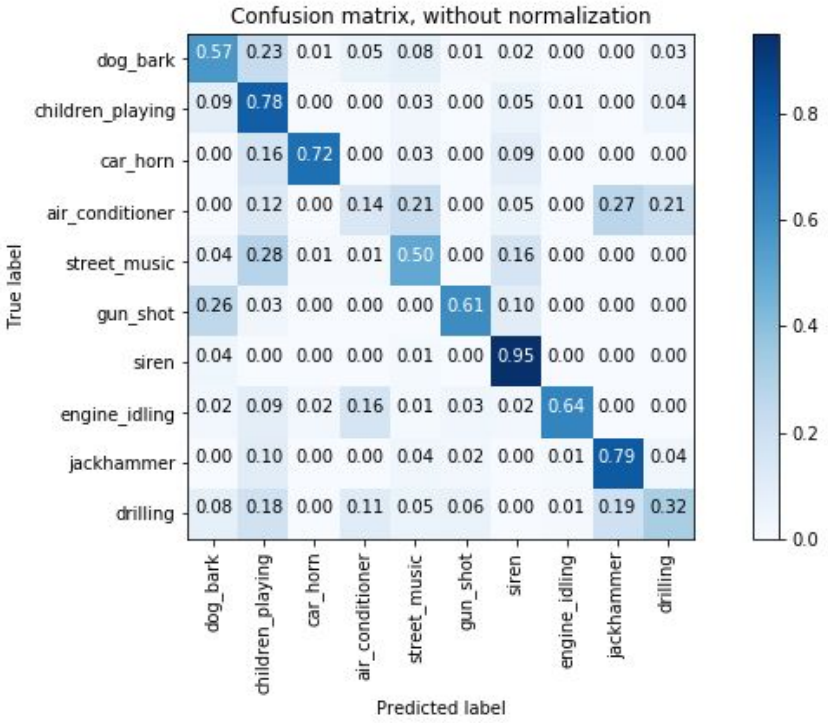
We implement k-NN classification using the sklearn's KNeighborsClassifier class. There is just one parameter that needs to be chosen for this algorithm namely:

- n_neighbors: the number of closest neighbors to consider.

Moreover, since it utilizes the Euclidean metric to measure nearness between samples, k-NN is sensitive to scaling of the data. We thus use sklearn's MinMaxScaler to scale all training features between -1, and 1.

On average, the accuracy score for k-NN is found to be: 0.49. The performance of the model is also further described in the confusion matrix shown in Figure 16. This confusion matrix was obtained taking one of the folds as a test set. As can be seen in Figure 15, the model misclassifies air_conditioner and jackhammer as drilling (which can make sense due to the nature of the noise). The model performs very well in identifying siren sounds. While k-NN generally does not perform as well as SVM, it does perform better for certain sounds like sirens.

Figure 16: Confusion Matrix Using k-NN.



6.4- Random Forests:

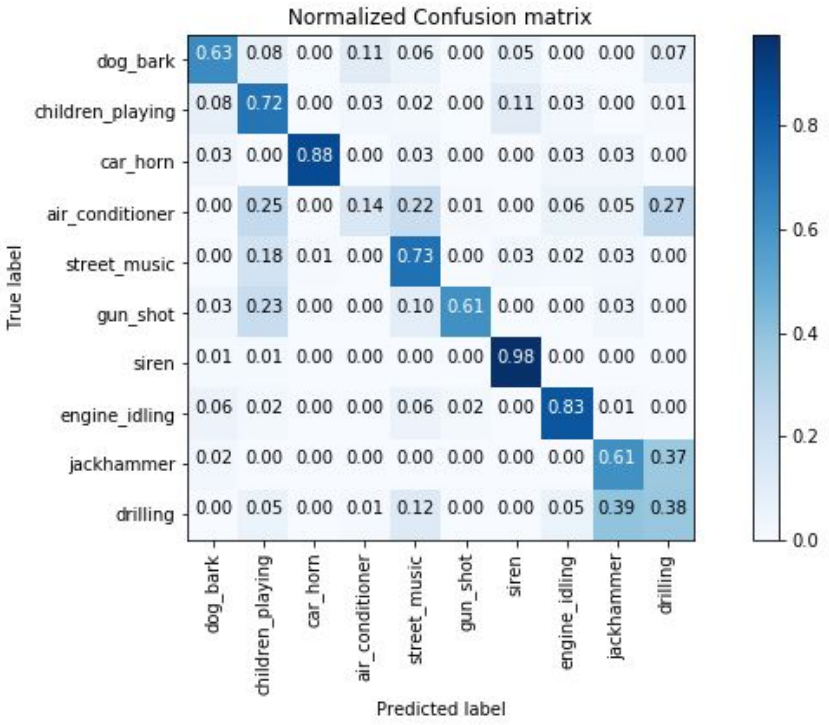
We implement Random Forest classification using the sklearn's RandomForestClassifier class. The parameters that need to be chosen for this algorithm are:

- n_estimators: The number of trees in the forest.
- max_features: The number of randomly chosen features to consider when looking for a split.

No scaling is needed for the Random Forest classifier.

On average, the accuracy score for KNN is found to be: 0.55. The performance of the model is also further described in the confusion matrix shown in Figure 17.

Figure 17: Confusion Matrix Using Random Forest.



6.5- Simple Neural Network:

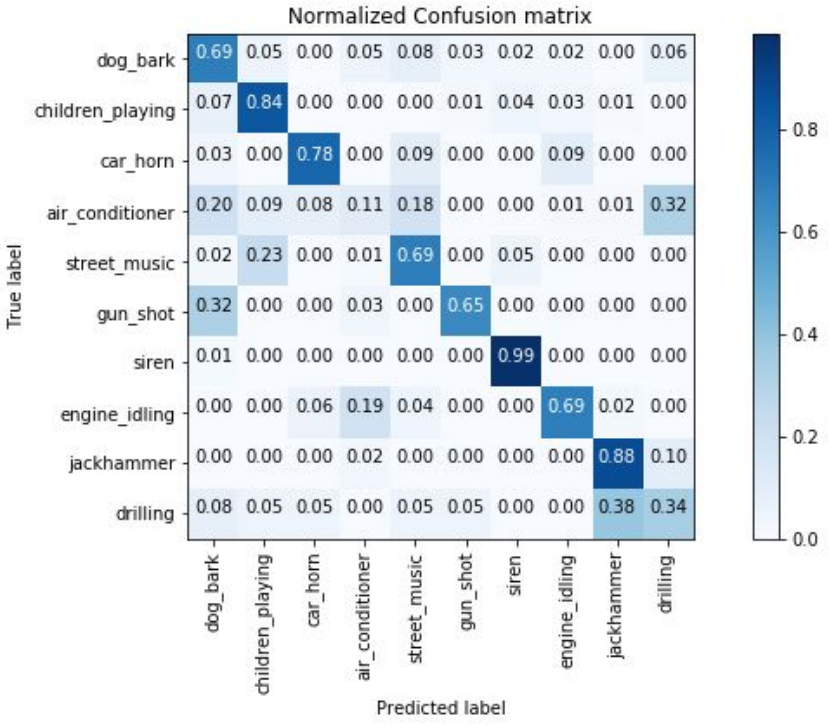
We implement a simple Neural Network classification using the sklearn's MLPClassifier class. The parameters that need to be chosen for this algorithm are:

- Number of Hidden Layers:we stick to 3
- Number of Neurons per hidden layer.
- Activation Function: logistic, tanh, or relu.
- Solver: we stick to 'adam'.

Stochastic gradient descent is sensitive to scale, we thus use sklearn's MinMaxScaler to scale all training features between -1, and 1.

On average, the accuracy score for KNN is found to be: 0.56. The performance of the model is also further described in the confusion matrix shown in Figure 18.

Figure 18: Confusion MLP.



7- Conclusions and Future Work:

In this report, we investigated using supervised learning to classify sounds. Specifically, we used the UrbanSound8k dataset, extracted features from every sound, and used these features to train various classification algorithms.

In general, classification models were able to label unseen sounds with an accuracy of about 60%. Different models varied in performance, as well as in ability to label certain sounds. For example, k-NN proved to be better in labeling siren sounds than SVM, while SVM had a higher accuracy rate in general.

Below we summarizes future work that may help enhance the performance of classification models:

1. In our study we temporally averaged, removing any temporal variation information from or features. We could include features for every row of every spectrogram,chromogram, etc. that explain the variation in that row. (We could record for example the coefficient of a fitted polynomial).

2. In our study we used subsets of folds to perform hyper-parameter tuning. The grid used in the parameter gridsearch was also coarse. We recommend the use of finer grids that are assessed using cross-validation on the full folds.

3. In this study, we used MinMaxScaler when scaling the data. Trying different scalers should be considered.

4. Viewing that different models showed different capabilities for certain sounds, ensemble methods that include all the models discussed above could increase performance.

5. Extract more features!

This report, along with all supporting code can be found on:

https://github.com/harajlim/Urban_Noises