Online Clustering Project Report 1

10/31/2024

Kennan Wu

Overview

The purpose of this project is to be able to cluster different audio signals in an online manner using ART2 clustering. As of now, the project features two main steps in achieving this. First, data preparation, which includes extracting features from the audio files, and creating five shuffled time series to work with. Second, the ART clustering algorithm itself, which will assess itself with a confusion matrix and accuracy percentage. Overall, with the most optimal set up so far, the range for the clustering algorithm was between 77% - 97%, with most accuracy percentages hovering around 90%.

Important Information

1. This code was written on a windows machine.
2. This code is optimized to run on the GPU using PyTorch, however if no GPU is detected, then it will run on the CPU, however the GPU greatly increases the processing speed.
3. To get the most accurate results, run the main.ipynb file.
4. .csv time series information is found in the output folder (this will populate itself after running the code)
5. For the whole dataset, look at all_features.csv for all possible features, and selected_features.csv, for features I deemed the most important
6. For each individual time series, ground_truth_{index}.csv contains the datapoints.

Getting the code to run

1. cd into the codebase root
2. if you are on a windows machine, run `` `conda env create --file=environments\environment-windows.yaml -n online-clustering-env` ``
3. if you are on macOS, run `` `conda env create --file=environments\environment-macos.yaml -n online-clustering-env` ``
4. In the main.ipynb file, select online-clustering-env as your kernel, and run the file

General Codebase overview

The codebase follows a fairly simple file structure. To maintain readability, hefty modules were written as classes in python files with four main classes. AudioProcessor, FeatureExtractor, DatasetHandler, and ART2Processor, all of which are in the *modules* folder. There are three different Jupyter Notebook files, main.ipynb, buffer.ipynb, and no_normalization.ipynb. Main.ipynb is the file with the most optimal setup I found during testing, where there is no smoothing, and normalized data. Buffer.ipynb uses an ART2 clustering algorithm that implements smoothing, where batches of data are added using the mean of that batch. Finally, no_normalization.ipynb uses no normalization to cluster data.

Processing the Audio

First, the audio needs to be loaded in and split into frames. To do this, the AudioProcessor class uses torchaudio to load in the mp3 files. It will take in a frame width in milliseconds, and a hop length ratio, which is the amount of overlap you want from the frames. The frame width and hop length gets converted into samples, and split into different frames. These frames are stored in a tensor, with other tensors accounting for class ids, and timestamps. These tensors can then be used for further processing.

Feature Extraction

Next, features need to be extracted from each frame for each audio. The FrameExtractor class takes in a n_fft, which in this case is set to be 1024. Using torchaudio, the MFCCs of each frame and the melspectrogram of each frame is extracted and stored in their own respective tensors. Heatmaps of these features are generated as well, helping with choosing which features to utilize (this is further explained in the Combining Features section).

Time Series Generation

The features, ids, and timestamps all need to be consolidated into a single form of data for better ease of use. The DatasetHandler will take all these tensors and create an all_features.csv file. Next, the features that are deemed to be not as differing as the rest are removed from the feature set, and a selected_features.csv file is created.

Now, the time series can be generated using the selected features we would like to cluster with. to do this, the selected_features.csv is randomized, and the first amount of elements to make up 2 minutes are chosen. A new csv file called ground_truth_0.csv is created. This process is done 5 times, creating 5 different csv files.

ART2 Clustering

Finally, we have two different options to cluster with, the first ART2clustering algorithm, takes in data points one at a time, and finds a cluster with the nearest mean to the datapoint. If that distance is less than a vigilance variable, the datapoint is added to the cluster, otherwise, this datapoint becomes a new cluster.

The second option is SmoothingART2clustering algorithm. Instead of taking a single point, the mean of batches of data are found and the distance of this mean to other means is calculated. The process is the same except now batches of data are added in at a time.

Once the data points are clustered, two graphs are made, the ground truth vs time graph, and the predicted classes vs time graph. These graphs allow for comparisons between predicted data, and the actual classification of that data. A confusion matrix and accuracy is also generated to tell us numerically how well the clustering algorithm did.

Results

Overview

Overall, a somewhat optimal solution was found for the clustering. First, there were several variables that were tweaked, including frame size, hop ratio, n_fft, and most importantly, the vigilance parameter. All of these parameters can be seen in the first cell of the main.ipynb file. These values for the parameters yielded the highest accuracies for me:

1. Frame size (ms): 150
2. Hop ratio: 0.1
3. n_fft: 1024
4. Vigilance: 1.4

Other methods were used as well, such as feature combination, normalization, and smoothing. All of these methods will be explained in much greater detail in the following sections, though here is a quick overview for each:

For feature combination, all features other than 2 and 12 of the MFCC coefficients were found to have the most variation between classes, thus they were chosen. For the MFSCs, most variation was found between coefficients 0-14, and 16-35. This helped reduce the noise of datapoints a bit, and increased the clustering efficiency.

For normalization, a simple max min strategy was used, and, as to be expected, clustering normalized data performed significantly better than non normalized data.

Finally, for smoothing, a new variable to change around called buffer size is introduced, which modifies how many data points should be buffered, before actually adding the data into the clusters. I wasn't able to get a significant improvement, in fact, the accuracy of the model decreased.

Selecting the MFCC and MFSC features

As stated in the general codebase overview, the MFCC and MFSC features were plotted in a heatmap. This heatmap allowed me to see which feature coefficients had more difference when compared to the other classes of data. These MFCC features also changed depending on the frame size, which I will talk about choosing the proper one in the next section.

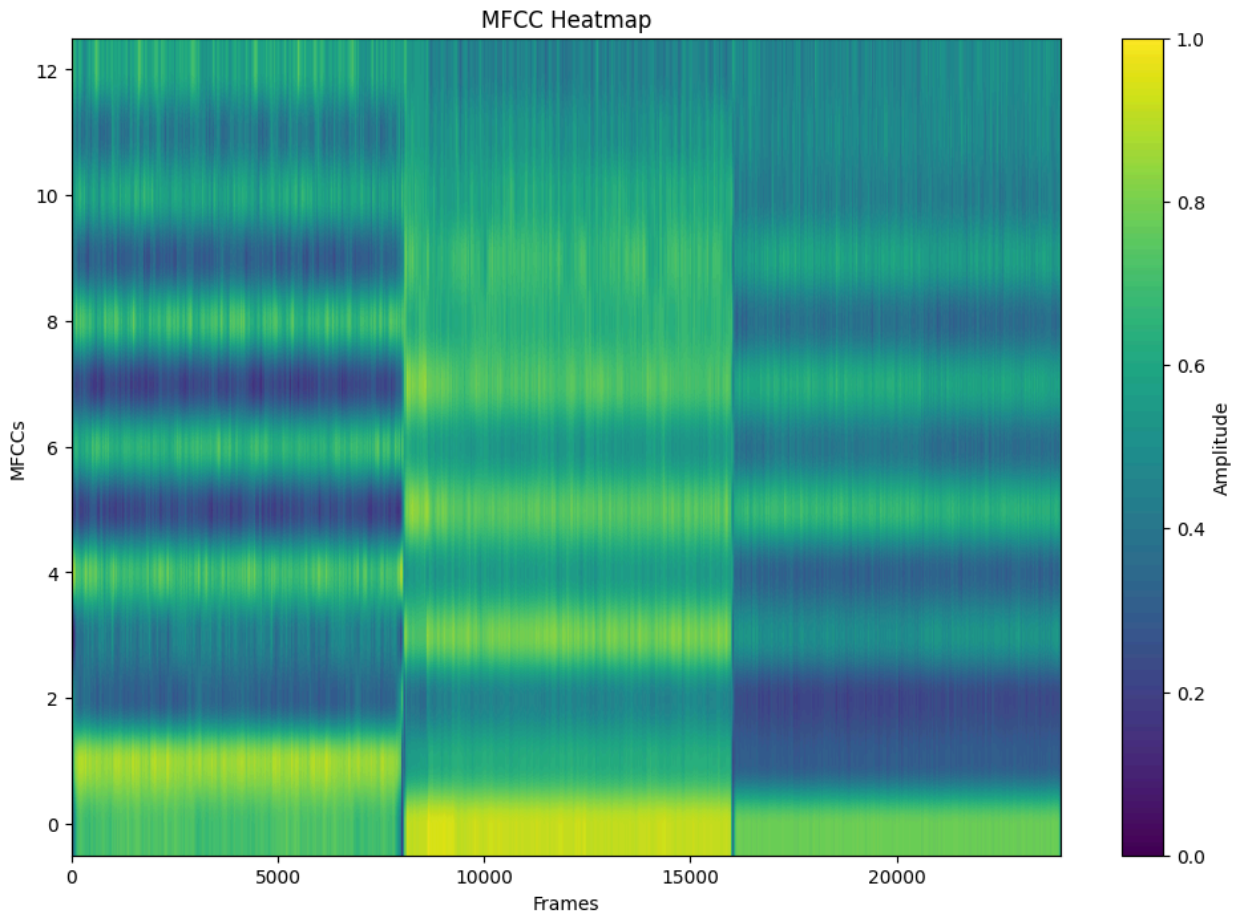

Figure 1.
The normalized MFCC heatmap for a 150ms frame size with 13 MFCC features.

As seen in Figure 1, we can see very distinguished boundaries at about frames 80,000 and 15,500. These boundaries are the changes in different audios. This is good because we want to find features that have the most variation between the differing classes. The goal of this heatmap is to look at which MFCC features have heavy differences between the classes. For example, looking at feature 2, we can see that for all classes, it is mostly the same color of blue, meaning this feature does not change that much depending on the class, and is a feature that will not help us categorize data. The same can be said for feature 12, where the feature is essentially the same color across the entire width.

On the contrary, a very distinct feature would be 5. In the first class it is a dark purple, in the next, a light green, and in the last a dark green. In each class there is a distinct color difference between the classes, and thus this feature is helpful for classification. Reducing the right features can have a fairly significant impact on the accuracy of the clustering. For example, running with 44 features leads to an overall accuracy of 63%, whereas 40 features leads to 93%.

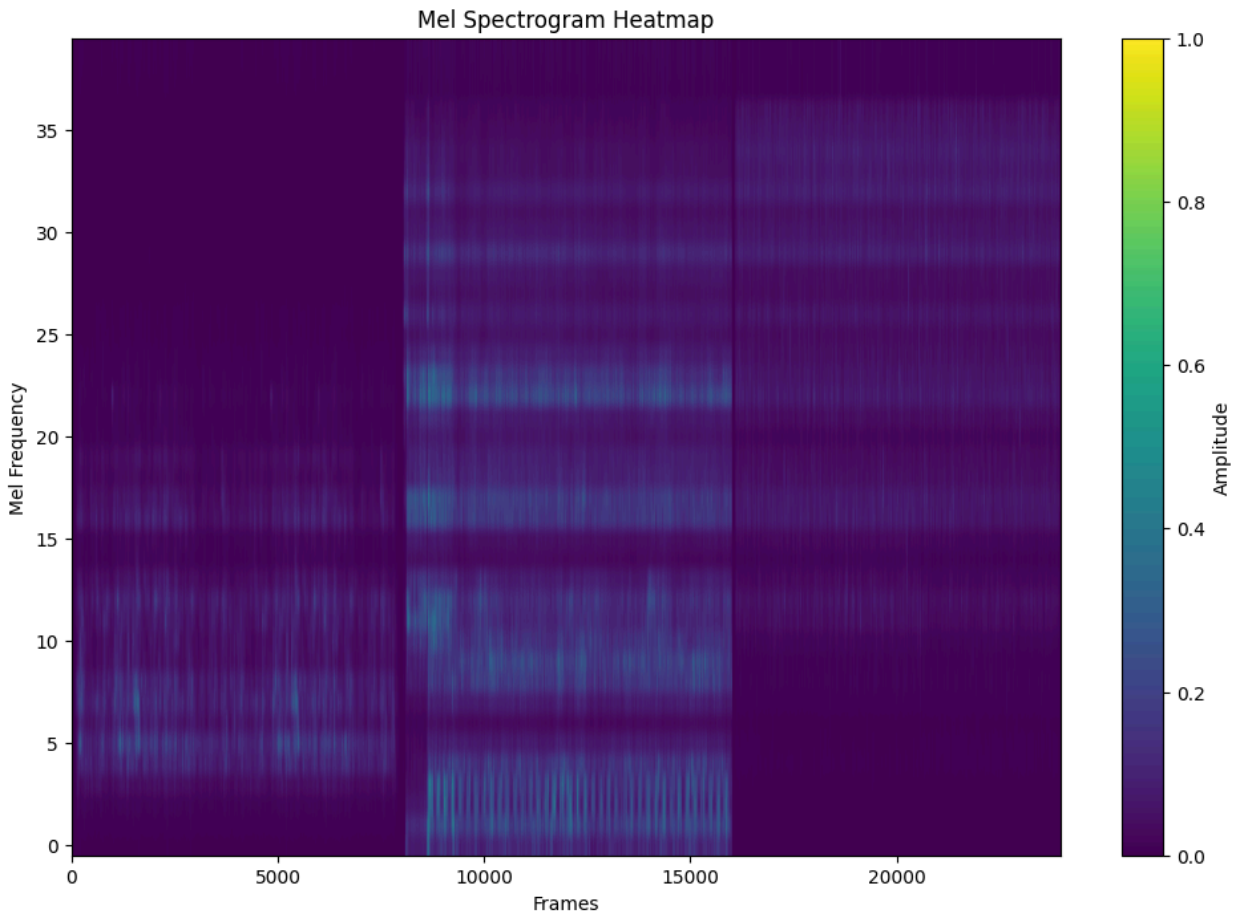

Figure 2.
The normalized heat map of the MFSCs for a 150 ms frame size 40 mels

Similar to the MFCCs, we want to use the features with the highest difference between the classes. In this case, the mels between 0 to 14 and 16 to 35 give the highest difference.

Frame Size Selection

Frame size also allows for variability in the clustering accuracy. To choose the right frame size, I looked at the MFCC and MFSC variations between smaller and larger frame size selections.

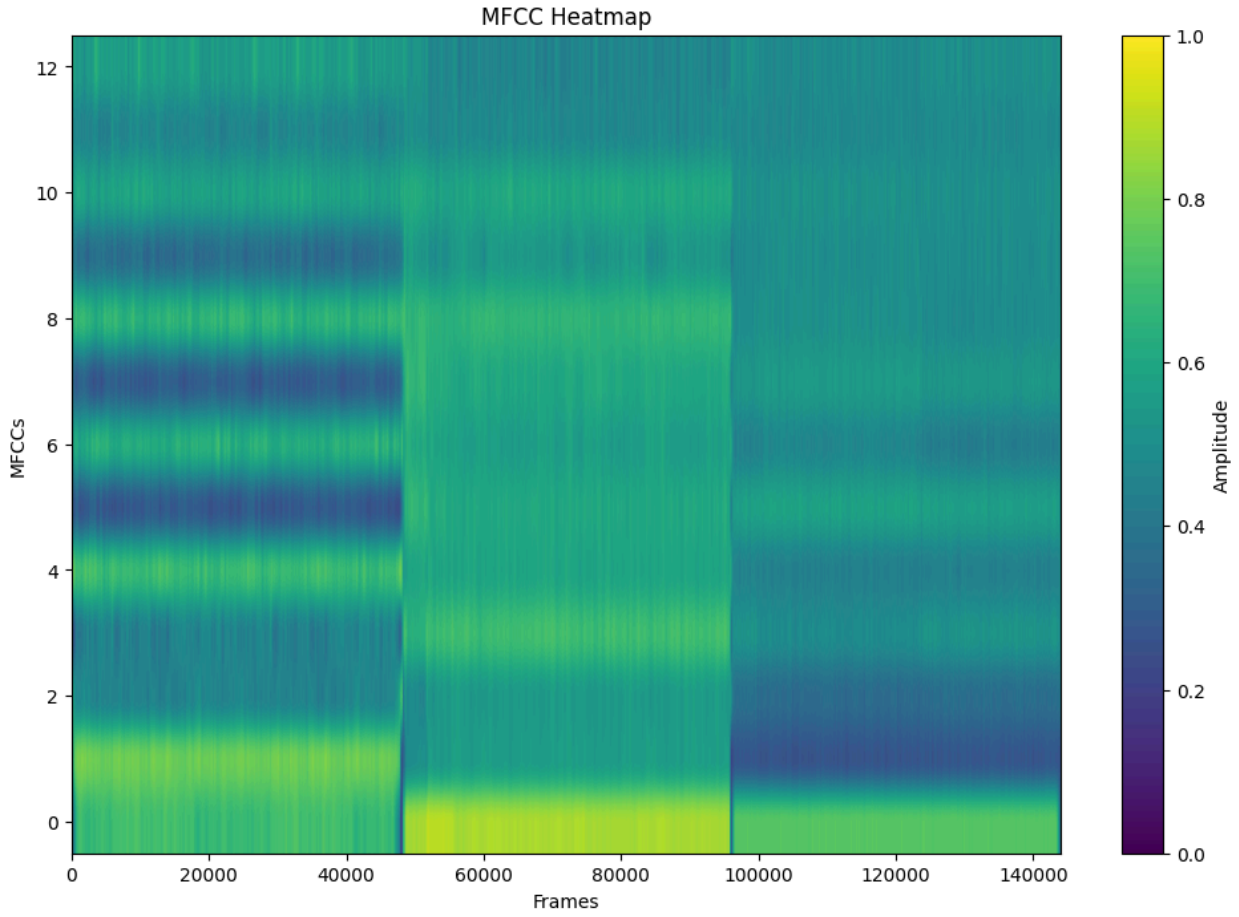

Figure 3.
The normalized MFCC heatmap for a 25 ms frame size with 13 MFCC features.

When comparing Figure 1, there is much less variability between classes, and features, in particular, the 2nd and 3rd class both sharing a deep shade of green. It can be seen that a decrease in frame size means a decrease in variability because there is now less audio to capture and thus less variability between small snippets of audio.

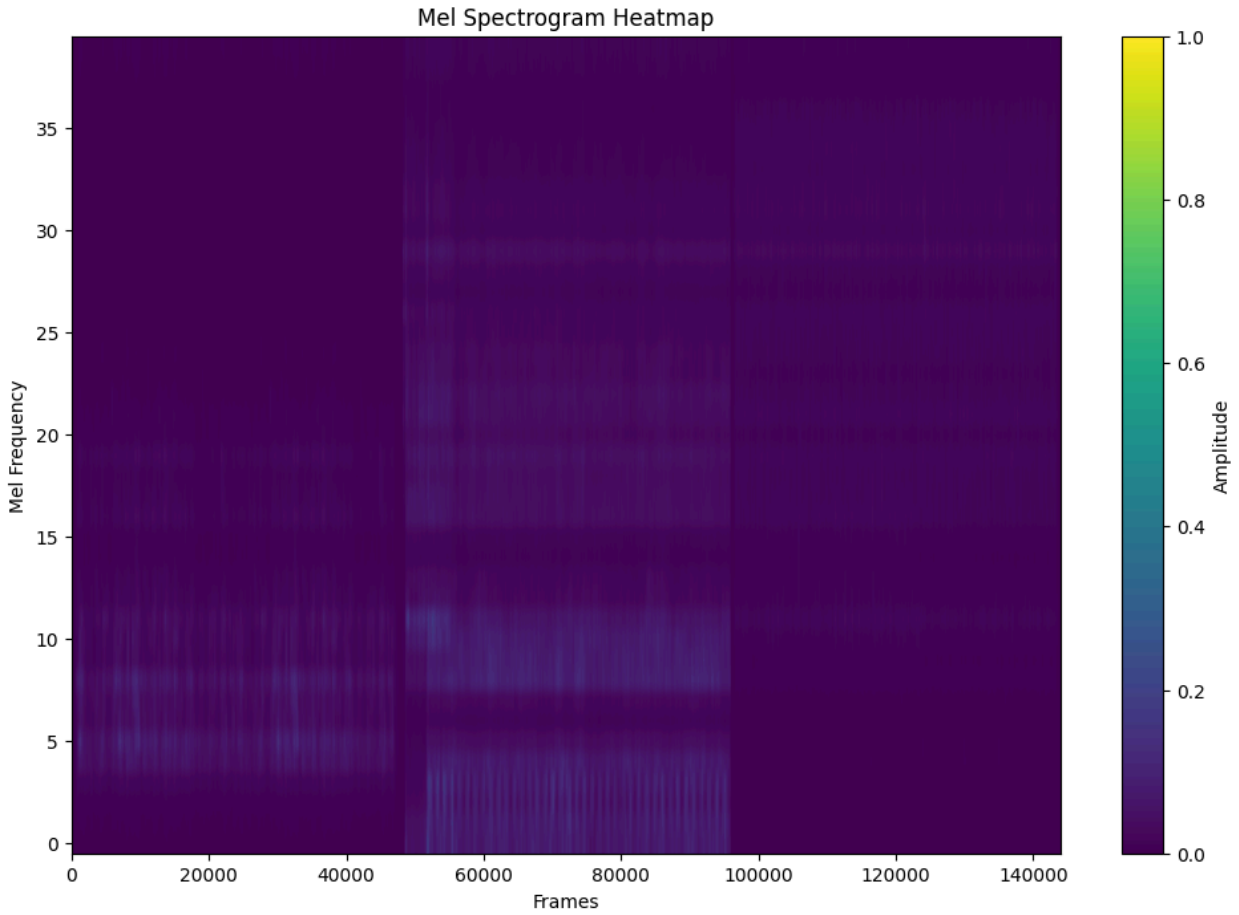

Figure 4.
The normalized heat map of the MFSCs for a 25 ms frame size 40 mels

The same can be said for comparisons between Figure 2 and Figure 4. Figure 4 now seems more like a block of purple, with almost no variability between classes.

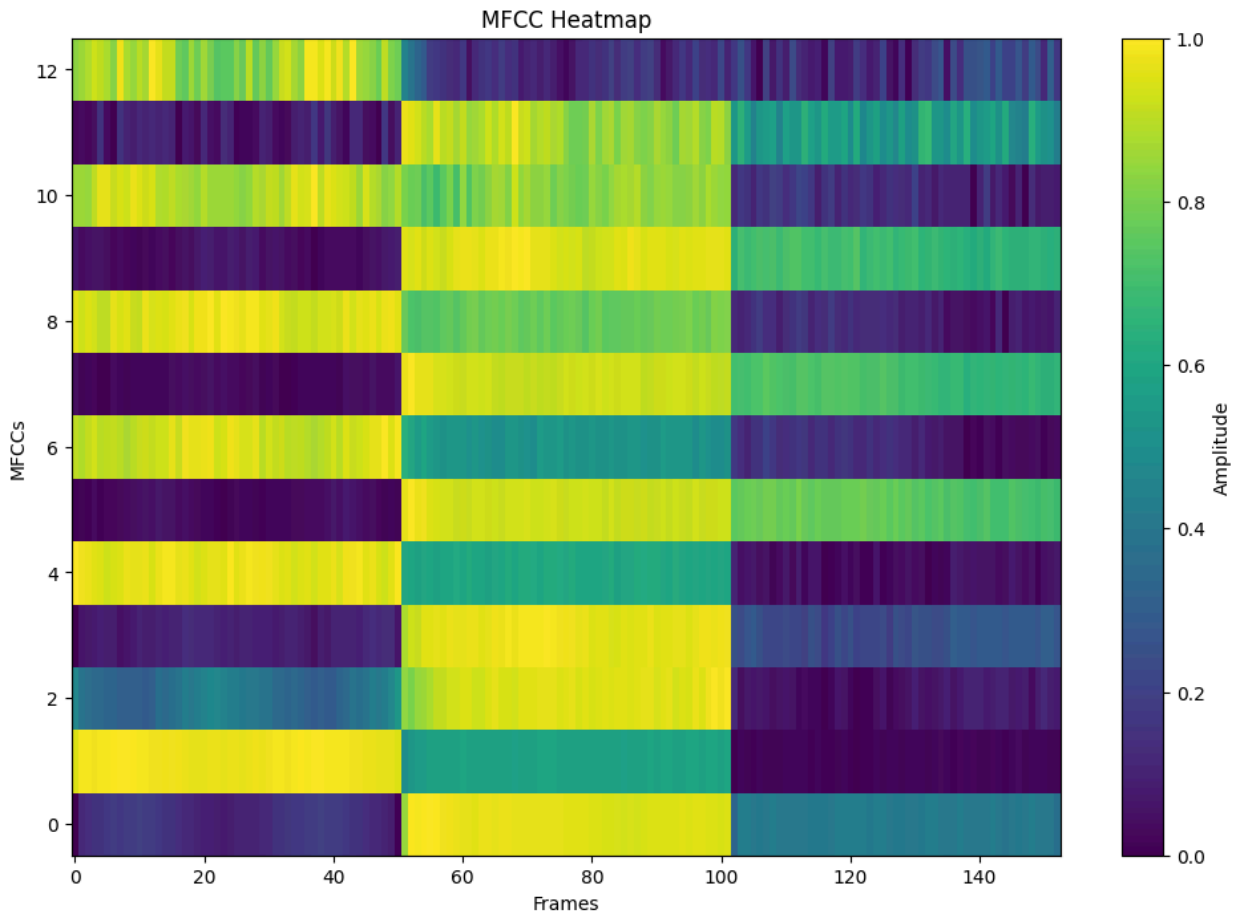

Figure 5.
The normalized MFCC heatmap for a 20,000 ms frame size with 13 MFCC features.

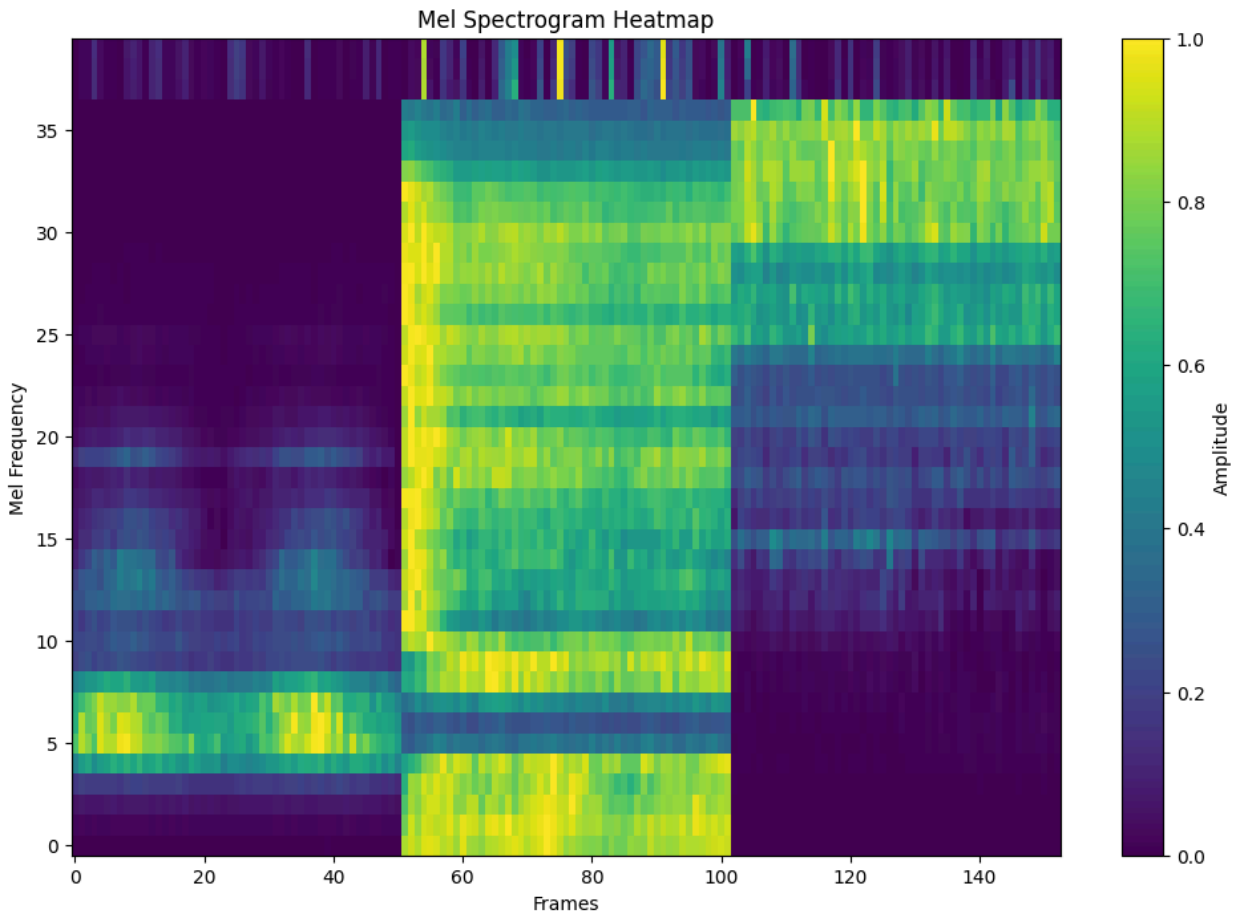

Figure 6.
The normalized heat map of the MFSCs for a 20,000 ms frame size 40 mels.

On the contrary, creating exceptionally large windows creates very high variability as seen in figures 5 and 6. However, this comes at a cost. High windows means much less data to work with, and larger chunks of data being fed to the actual classifier. Although at first this might make the accuracy increase at first, when adding multiple audios that could sound similar but are truly different classes, there would be a very large decrease in accuracy. Thus, a middle ground of 150 ms was chosen.

Normalization vs No Normalization

Normalization turns all data points to be between 0 and 1. For this codebase, min max normalization is used, where the smallest feature becomes 0, and the largest becomes 1. From my testing, No normalization has a very large impact on both accuracy and efficiency.

When values aren't normalized, performing numerical operations on them can become very tedious and space inefficient. For example, multiplying 1M by 1M is a much more computationally and storage intensive task than doing 1 times 1.

Non normalized values also cause features to not have a “weight”. When looking at MFCCs and MFSCs, a MFCC of 30 might have more weight than an MFSC of 100, but since the values aren’t normalized, we don’t know that, and using these plain numbers causes the distances between data points to be exceptionally large.

In my testing, because of the large MFCCs, my computer constantly ran out of storage when computing values and placing these large values into clusters. The code to run the non normalized values is in non_normalized.ipynb.

Choosing a Vigilance Parameter

The vigilance parameter is the most sensitive to increasing or decreasing the accuracy, since it is the direct number an ART2 clustering algorithm uses to determine whether to create a new cluster or not.

The best way to see how the vigilance parameter was changing was by looking at either how many clusters were being created, or how many data points were being placed in clusters. Generally, if there are too many data points in a single cluster, the vigilance parameter is too large, and if there are too many clusters, the vigilance parameter is too small.

```
tensor([787, 456,   1, 307, 590,   1, 146,   4, 380,   2,  71,  76,  94,   1,
         29,  15,  59,  39,   4, 322,   3,   1,   1,   1,  22,  90,  38,   2,
          1,  42,  68,  63,   1,   2,  31,   2,   5,   1,  23,  24, 135, 443,
          1,  64,   5,  86,   1,  29,   1,   6,  62,   2,  84, 149, 155,  76,
          1,   1,   1,   1,  37,  12,   1,  38,   2,   3,   1,  82, 492,   1,
        104,   1,   1,   1,  45,  22,   1,   4,   1,   1,  61,   3,  10,  45,
          3,   1, 101,  12,   1,  73,  28,   1,   9, 266,   1,   1,  10,  65,
         14,   7,   1,   1,   5,   1,  17,   2,   4,   1,  32,  17,   1,  16,
         24,   7,  41, 121,  43,  28,   3,  10,   2,   1,  67,   1,   5,   1,
          2,   1,   1,   8,   4,   1,   1,   8,  11,   7,   2,   1,   1,   9,
          5,   2,  76,   1,   2,   1,   1,  18,   2,   1,   2,   1,  14,   1,
          2,   1,   8,   1,   1,   1,   1,   1,   1,   1,   1,   3,   1,   5,
          1,   2,   1,   2,   3,   1,   1,   5,   1,   1,   1,   2,   1,   1,
          5,   2,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   6,  16,
```

Figure 7.
A tensor that holds the amount of data points in a cluster with vigilance of 0.2.

As seen in figure 7, each 1 represents a new cluster holding just a single elements. When the vigilance is too small, there will be too many new clusters.

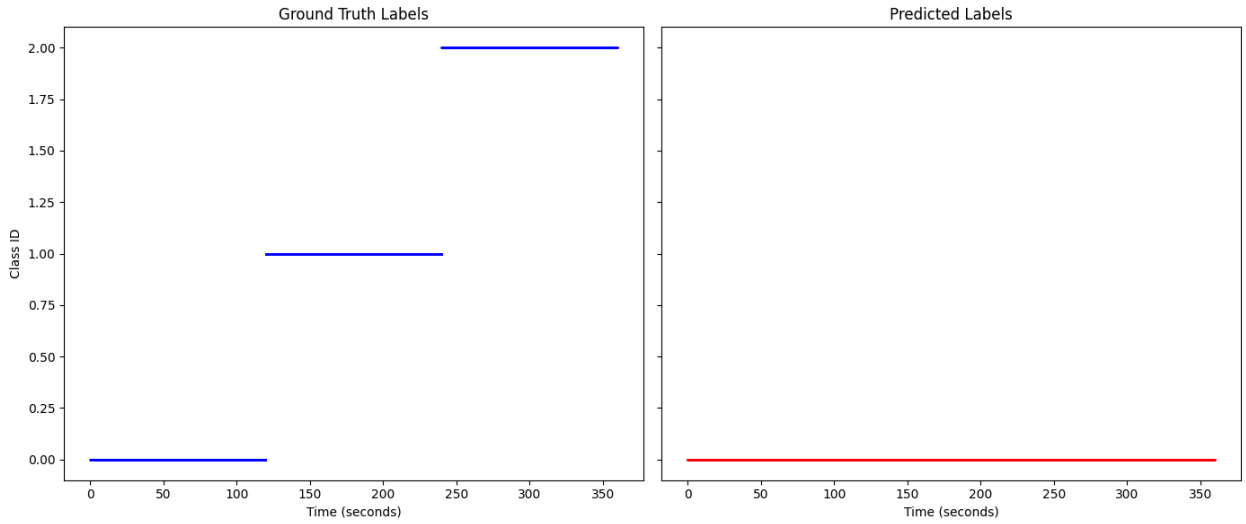

Figure 8.
The ground truth clusters, and the predicted label cluster with a vigilance of 10

As seen in the figure, a very large vigilance causes the data to be clustered all in a single cluster, thus causing the accuracy to be about 30 percent. For all these reasons, a single middle value of 1.5 was chosen.

Ground Truth vs Time and Predicted Clusters vs Time

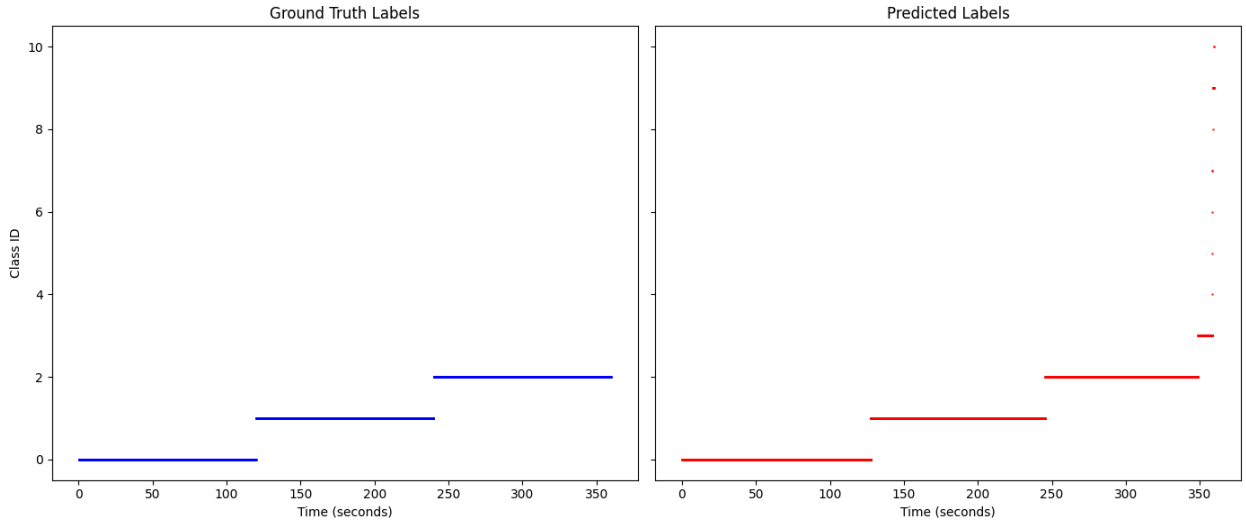

Figure 9.
The ground truth labels plotted against time and the predicted clusters plotted against time.

In figure 9, we see the ground truth labels being plotted as well as the predicted clusters. As seen in the predicted labels, some points are being labeled as different classes entirely from the 3 original classes due to the unknown variability of the nature of ART2. There are multiple

ways to fix this, however. The first method would be to increase the frame size, however increasing the frame size would cause less sensitivity between features, and thus having less features to classify in general. Another method would be to use smoothing, which is talked about in the following section.

```
Confusion matrix:
[[2039    0    0    0    0    0    0    0    0    0    0    0]
 [ 280 1708    0    0    0    0    0    0    0    0    0    0]
 [   0  262    3 1689    1    1    3    4    1    7    1    1]
 [   0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0]
 [   0    0    0    0    0    0    0    0    0    0    0    0]]
Accuracy: 90.60000000000001%
```

Figure 10.
The confusion matrix and accuracy for a time series with 150 ms frame size and 1.4 vigilance

The confusion matrix shown in figure 10 shows us how much certain classes of the ART2 clustering algorithm overlap with the ground truth labels. According to the size of the matrix, there are 11 total classes being created. The large number of classes being created could be caused by a couple outliers, such as the beginning of the audio, which is silence, could cause some samples to simply be classified as their own cluster.

Smoothing

Smoothing with the ART2 clustering algorithm adds in another variable we can modify which is the batch size. In my case, I worked with a batch size of 5. However, adding my batch size caused the accuracy of the clustering algorithm to drop by a substantial margin. This could be because there is a reduced sensitivity now to each data point. It would take more adjustments to the frame size and vigilance parameter to understand. Currently, the code for running the ART2 clustering algorithm with smoothing is on buffer.ipynb.

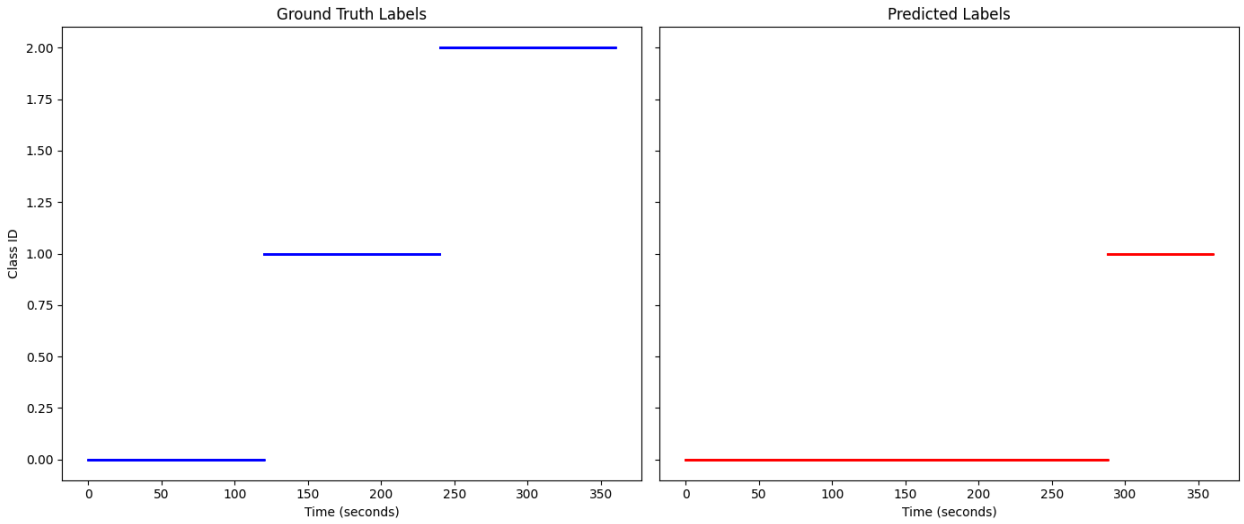

Figure 11.
Clustering with smoothing and vigilance of 0.2

Here, we can see that when clustering with smoothing, the amount of clusters created significantly reduces. However, this comes with its own drawbacks. Now that we have smoothing, individual data now has less information to work with, and distinguishing features that are meshed together causes more ambiguity between clusters. Because of this, we end up with classes that contain multiple ground truth classes, and clusters that are simply the outliers.

I believe the clustering with smoothing in theory is something that should be able to improve the accuracy as it should decrease the amount of outliers that are being placed in their own clusters. The only issue is when data points that are two different ground truth clusters are included in the same batch, the mean between the two of them are no longer that first class nor the second, but simply in between. This will create a class of its own, where it is a class of both ground truth 1 and ground truth 2.

Conclusion

Overall, a decent result was achieved with an around 90% accuracy cluster being created. To get to this result, there were several techniques that had to be employed. First was choosing the proper MFCC and MFSC features to use by looking at their heatmaps. Next was choosing a proper frame size selection by looking how the increase or decrease of frame size affected the heat maps of the MFCC and MFSC, and overall a frame size of 150 ms was chosen. Next normalizing the data allowed for more efficient handling of the data, and normalization of certain weights for the data. For the future, I would like to be able to have smoothing help out on increasing the accuracy.