A Real-Time Violin Gesture Detector

Paul V. Miller

April 28, 2026

1 Introduction

Can machine learning techniques be used to recognize the way a violinist is playing in real time? At first glance, this may seem to be a straightforward problem of audio classification that might only involve pitch detection, amplitude, or other conventional measurements. But the notion of "how a violinist is playing" encompasses a much broader, multidimensional set of musical phenomena, including articulation (e.g., pizzicato vs. bowed tones,) modes of sound production (such as tremolo or scratch noise,) or subtle variations in timbre. These distinctions are not just categorical, but often exist along continua that are affected by a performer's technique, the response of the instrument, and even the kind of strings or bow that is used.

From a signal-processing perspective, these gestures do not correspond to simple, isolated acoustic events but rather to evolving and fluctuating patterns in the spectral and temporal structure of sound. The same nominal gesture – for example, tremolo – may be realized in many different ways by different performers, or even by the same performer under varying conditions. For example, the tremolos in the finale of Beethoven's *Seventh Symphony* op. 92 (1811-12) are quite unlike the much more vigorous and nuanced ones in Luciano Berio's *Sequenza VI* for solo viola (1967). This variability makes rule-based or heuristic approaches difficult to generalize.

Machine learning offers a promising framework for addressing the problem, because it allows models to learn complex, high-dimensional relationships between acoustic features and perceptually meaningful gestural categories. By training on examples of different playing techniques, a model can associate patterns of spectral shape, temporal variability and harmonic structure with a performer's intentionality. The challenge becomes not just whether such a system can achieve a high accuracy of classification, but whether it can also generalize across performers, while operating in real time.

Measuring and classifying performance techniques can be useful to identify a specific player's idiosyncracies, stylistic fingerprints, or for pedagogical purposes. In this project, I used classification data to alter the trajectory of an electronic composition in real time, allowing for a sophisticated relationship between human and machine than what is typically possible. The larger purpose of the classification system ultimately serves as a novel link between the technical and artistic domains.

2 Background

2.1 MFCCs

Attempts to classify musical sound computationally are grounded in signal processing techniques that extract meaningful descriptors from audio signals efficiently. MFCCs (Mel-Frequency Cepstral Coefficients) are one of the key tools. The origin of today's research can be traced all the way back to the 1960s, when Bruce Bogert and others introduced the idea of the **cepstrum** to detect echos or delays in repeating structures of some kind of signal.[1] Shortly thereafter, Tukey and Cooley discovered the FFT algorithm, which sped up computing the DFT from $O(n^2)$ to $O(n \log n)$, making it far easier to compute energy distributions across the audio spectrum. The modern notion of MFCCs originated some 15 years later in the discipline of speech

[1] Bogert, Healy, and Tukey 1963. The word "cepstrum" is a witty anagram of "spectrum".

analysis, as a way to represent the perceptually relevant aspects of speech better than Linear Predictive Coding (LPC), the other dominant approach at the time.[2]

To produce MFCC coefficients, audio signals are broken down into frames (or "windows") of around 25ms (e.g., 1024 samples at 44.1kHz sampling rate = about 23ms). Windows overlap by a hop size (e.g., 512 samples). To improve FFT performance, the windows are multiplied by some sort of function to attenuate the edges (usually using a Hanning window) to prevent inevitable spectral leakage between frames. Then we compute FFT on each frame to produce a power spectrum. The next step involves applying 20 – 40 triangular filters spaced on the Mel scale, a perceptually-based arrangement. The filters vary both with respect to their frequency centers (which become more logarithmic at higher frequencies) and their widths (which are more narrow at low frequencies). Then we apply a logarithmic amplitude scaling, because human perception of amplitude is nonlinear. The final step is to apply a Discrete Cosine Transform (DCT) to the log energies. This can be expressed as:

$$c_n = \sum_{m=1}^{M} \log(E_m) \cos\left[\frac{\pi n}{M}\left(m - \frac{1}{2}\right)\right]$$

where c_n is the coefficient index, m is the Mel filter index, M is the number of Mel filters, and E_m is the energy in the mth Mel band.[3] We subtract $\frac{1}{2}$ to put the cosine sampling at the *center* of each Mel band, not the edge. The DCT is a form of compression in that it reduces dimensionality to a few cosine waves (usually 13) at different frequencies.[4] The DCT essentially *decorrelates* the bands, discarding some of the higher coefficients in the process.[5] Consequently MFCCs are a kind of "spectrum of a spectrum".

After Beth Logan's work demonstrated the viability of MFCCs across a wider range of audio applications, they were quickly adapted for sound and music analysis by researchers such as Geoffroy Peeters at IRCAM (Logan 2000; Peeters 2004). The field of MIR (music information retrieval) evolved rapidly around 2000 with the organization of the first international conference (http://ismir.net/). Whereas many early audio analysis algorithms were associated with MATLAB, C or C++, the spotlight has partly shifted towards the `librosa` library, which was written principally by researchers at Columbia University (McFee et al. 2015). To this day there is widespread interest in MFCCs both by academic researchers and commercial entities such as Spotify, iTunes, Pandora, and other popular consumer-facing applications.

2.2 Beyond MFCCs: spectral features

While MFCCs play a central role in spectral analysis, additional features such as spectral centroid, spectral flatness, spectral rolloff, harmonicity and zero crossing rate (ZCR) are useful tools for representing qualities of sound (Peeters 2004). Alongside MFCCs, these features enable systems to better distinguish broad categories such as pitched vs. noisy sounds, different instrument families, or vocalizations such as vowels and intonation. A system for recognizing violin playing should probably have these tools, in addition to the MFCC battery.

2.3 Real Time Systems

Frequently, these analytical metrics are used for offline, out-of-time audio analysis. Real-time interactive systems in computer music have increasingly sought to incorporate audio-based control, allowing performers to influence electronic music processes through the very sound they produce. Such systems require not only accurate classification abilities, but also low-latency processing times, and need to demonstrate consistency and robustness under changing acoustic conditions. The 2022 **Fluid Corpus Manipulation project** – nicknamed "FluCoMa" – provides a suite of objects for Max/MSP, targeting "techno-fluent aesthetic researchers" and composers (https://www.flucoma.org/).[6] This free open-source package, centered around researchers at the Queensgate campus of the University of Huddersfield, includes MFCCs, ML classifiers,

[2]Davis and Mermelstein 1980. While MFCCs are better for analysis, they are not easily invertable; LPC still excels at speech synthesis.

[3]Davis and Mermelstein 1980 essentially show exactly the same formula, though they define $log(E_m)$ earlier in the paper, and hard-code 20 filters into the formula (p. 359, formula 1).

[4]DCTs are also used in JPEG compression!

[5]The FluCoMa site shows many of these processes very clearly. See http://learn.flucoma.org/reference/melbands/.

[6]Tremblay et al. 2022.

and other tools for realtime use. The package provides many tools for real-time audio classification much like this project sets out to accomplish.

The present study offers a modest contribution to existing models by combining a feature-based representation of electric violin sound with supervised machine learning techniques, with particular emphasis on real-time operation and cross-performer generalization. By focusing on a small set of musically meaningful gestural categories (drawn from both standard and extended techniques,) and evaluating performance across different players, this work aims to build a bridge between analytical models of sound, and practical systems for interactive music performance.

3 Materials and Methods

3.1 Data Collection: General Procedures

We used a Zeta 5-string electric violin made by Steve Carlson in Bozeman, Montana. The bow was a carbon fiber CodaBow (Joule model), which is a bit on the heavy side for violin bow (61.7 grams), but very effective for five-string electric instruments. I used a standard Focusrite "Scarlett" 2i2 4th generation audio interface. Initial software development and analysis was completed on a Raspberry Pi 500 with 8 GB RAM. Later, more computationally intensive analysis, experiments and testing was conducted on an Apple M4 Macintosh with 24 GB RAM. The system deployed on an M5 laptop with 48 GB RAM.

First, I recorded 44 WAV files in Ableton Live for training and testing. Hannah Bedeck, a senior undergraduate violin performance major at Duquesne, recorded a second parallel set of 44 samples. These files were labeled and exported for supervised learning in separate paths, so they could be used more effectively in training. Together, we recorded over 37 minutes of audio.

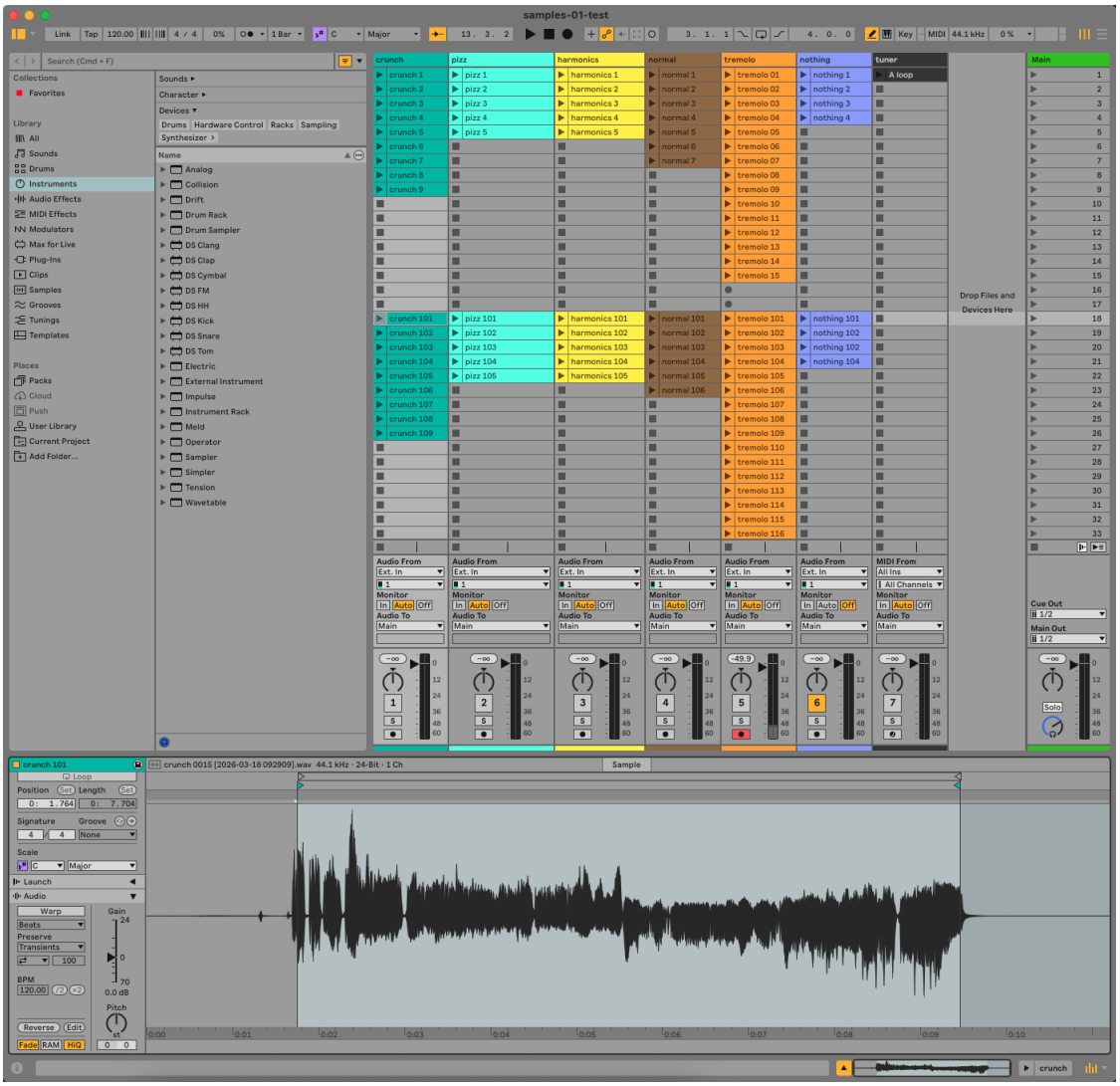


Figure 1: Ableton Live DAW with 88 recorded WAV files used in this study.
Top block: Paul Miller; Bottom block: Hannah Bedeck

Although analysis files were already carefully trimmed to ensure there was no silence at the beginning or end, each file was additionally trimmed by 1 second at both ends during the training procedure. I did not normalize the gain on the testing files, but rather set gain to the same value throughout the pipeline for the entire recording process. Since no microphones at all were used in recording the test data, there was no variation in room acoustic or player position relative to the microphone. All of the recordings were done in mono, 44.1kHz sampling rate, at 16 bits.

I decided to classify six different ways of playing the violin:

1. crunch: a loud, noisy high-bow-pressure crunch on the string (e.g., Helmut Lachenmann: *Pression* for solo violoncello (1969));

2. harmonics: smooth, glassy harmonics played across strings using harmonics (e.g., Salvatore Sciarrino: *Tre Notturni Brillanti* for solo viola (1975));

3. normal: ordinary playing of pitched notes;

4. nothing: total silence or nearly silence;

5. pizzicato: traditional pizzicato (plucking) of the strings – no bowing involved;

6. tremolo: fast alternation of bow stroke, at varying dynamics and varying speeds (e.g., Luciano Berio: *Sequenza VI* for solo viola (1967)).

3.2 Merge and Organize Features: Step 0

Because the WAV files were saved in two different folders, these datasets had to be potentially merged and unified before they could be processed. Step "0" ensured consistent class labeling and outputted an organized WAV dataset for subsequent processing.

3.3 Feature Extraction: Step 1

In step 1, I converted the recorded audio in the WAV files into numerical feature vectors, outputting a binary `.npz` file. Designing a feature vector for recognizing violin gestures is no easy task, and has significant consequences downstream. My study of audio systems was helpful in deciding what features to use. In Davis and Mermelstein's classic work on MFCCs, they employed 12-13 coefficients and 20 filters[7]. However, their research was mostly oriented towards speech recognition. Beth Logan used 13 coefficients during the early 2000s.[8]. Geoffroy Peeters used 12 MFCC coefficients; his research was specifically oriented towards music and sound processing, not speech[9]. The more recent openSMILE system (which is still actively maintained) was designed to extract features from both speech and musical sources using a wide array of audio processing techniques, including MFCCs of n-coefficients[10]. FluCoMa's `[fluid.mfcc~]` object defaults to 13 MFCC coefficients, but some of the help patches use more. The default number of MFCC coefficients in the `librosa` library is also 13, which is a good indication that this number is standard[11].

I started with a 57-dimensional feature vector. During the course of testing, I added two features to help improve classification. I hypothesized that they would increase the accuracy of tremolo and crunch detection. The two added features are indicated with an asterisk (*).

- MFCC-Based Features – 26 dimensions
  - MFCC Mean – 13 dimensions. These features represent the time-averaged MFCCs over the analysis window. They capture the spectral envelope – the overall timbral shape. Low-order coefficients reflect coarse properties such as brightness and spectral slope whereas high-order coefficients encode finer spectral detail. These features help distinguish timbre-based differences between gestures.
  - MFCC Standard Deviation – 13 dimensions. These features measure the temporal variability of the spectral envelope within the window. They quantify how much the spectral shape fluctuates over time, and are sensitive to instability, modulation and transient activity.
- Delta MFCC Features – 26 dimensions
  - Delta MFCC Mean – 13 dimensions. This is a time-averaged first derivative of MFCCs.[12] They capture systematic trends in spectral *change*, and indicate whether the spectral envelope is drifting over time. These measurements are useful for detecting gradual spectral motion, and evolving gestures such as glissando-like behavior.
  - Delta MFCC Standard Deviation – 13 dimensions. These features measure the variability of spectral change rates. They capture how rapidly and irregularly the spectrum is changing over time.

[7] Davis and Mermelstein 1980, p. 352.
[8] Logan 2000
[9] Peeters 2004
[10] Eyben, Wöllmer, and Schuller 2010
[11] McFee et al. 2015
[12] There is considerable discussion on the FluCoMa web site about these types of features, up to the second derivative!

- General Spectral Features – 5 dimensions
  - Spectral centroid. Measures the perceived "brightness" of the sound. This is the weighted average frequency across the entire spectrum. The centroid will increase if the higher frequencies have significant energy.
  - Spectral rolloff. This is the upper boundary of most of the energy in the spectrum. Rolloff measures the frequency below which most of the spectral energy lies. It counts up the energy in the spectral bins starting with the lowest, and simply stops when it reaches a certain threshold. Because of this, even a little energy in the upper frequencies can significantly affect rolloff. By default in `librosa`, the threshold is 85%; so, this is the frequency below which 85% of the spectral energy sits. The rolloff measurement is good at detecting harmonics, bow scratches and bow noise, because these ways of playing add lots of energy to the high frequency spectrum. This feature is especially good at detecting bow changes!
  - Spectral flatness. Measures noise vs. pitched structure. This is expressed as $\frac{geometric\,mean}{arithmetic\,mean}$ of the spectrum. A high value indicates the spectrum is more noise-like (i.e., the spectrum is more like broadband white noise,) whereas a low value indicates a more pitched sound (i.e., more like a sinusoidal wave).
  - Zero crossing rate (ZCR). The number of times the signal crosses the zero boundary. Computationally cheap and rough estimate for high frequency sound and high noise content. This value is high when there are attack transients or very noisy sounds.
  - * Harmonicity (autocorrelation-based). This feature measures the degree of periodicity in the window. High values indicate greater harmonic structure. Can help separate harmonic vs. noisy gestures, pitched vs. unpitched sounds, and stable tones vs. broadband noise. The function asks: how strongly does this signal resemble a time-shifted (τ) version of itself, at a plausible period? Autocorrelation generally tests all candidate periods and tries to find which one aligns itself best with the window. This was implemented using the `numpy correlate()` method, normalized relative to the zero-lag peak and evaluated over a plausible period range. At a sample rate of 44.1kHz, there will be 8820 samples in a 200ms window. Since autocorrelation is a quadratic operation $O(n^2)$, this means that in the very worst case there are about 77.8 million pairwise products for each frame. Because `numpy` is a well-optimized library, actual runtime will probably be substantially better; nevertheless, this is a potentially costly feature to compute.
    What do we gain by employing such a potentially costly feature? Autocorrelation buys us access to *periodicity*, which none of the other features capture well. Most of the other features are concerned with the **frequency domain**, but this one measures correlations in the **time domain**. The autocorrelation-based harmonicity measure is quite accurate even when the amplitude and timbre shift.
    Later in my analysis (table 2), I found that this feature was influential in separating different gesture classes.

- Energy and Temporal Features – 2 dimensions
  - RMS (in dBFS). Measures the overall signal energy.
  - * RMS micro-variance. Captures short timescale amplitude fluctuations. This can distinguish between silence (nothing) and active gestures; stable vs. modulating amplitude patterns (e.g., tremolo), and articulation differences.

The features I chose are principally designed to capture **three layers** of information – which are partially overlapping but largely complementary:

1. Spectral identity. MFCC mean and spectral features;
2. Temporal variability. MFCC standard deviation, and RMS micro-variance;
3. Spectral dynamics. Delta MFCC mean and std. "How is the sound changing over time"?

3.4 Model and Training Procedure: Step 2

3.4.1 Classifier

For processing, I decided on a window size of 200ms and a hop size of 50ms (so overlap was 75%). Peeters used a window size of 60ms and hop size of 20ms, but since I was hoping to catalog gestures in real time, I decided to use larger windows.

Since gesture classes are not separable by a single feature – and each class is a pattern across many features simultaneously – `LinearSVC` seemed to be the natural choice for a classifier. `LinearSVC` trains quickly and makes very fast inferences, which are very desirable properties for a real-time project. `LinearSVC` in `scikit-learn` handles multiple classes by learning a "one-versus-rest" strategy to separate each class from the others, so it returns easily interpretable results.

If we examine a 2-dimensional principal component analysis (PCA) of 1000 analysis windows, we see a notable lack of low-dimensional linear separability. However this sort of thing happens frequently in speech recognition, audio classification, or other high-dimensional feature systems. We can be confident this is the case, because the two PCA axes together account for only about 35% of the variance in the data – so there is quite a lot of structure that is **not** visible in the 2D plot. This suggests that low-dimensional projections *do not* reveal separability. Classification therefore relies on structure in higher-dimensional feature space that `LinearSVC` is designed to find.

3.4.2 Train-Test Strategy

In step 2, the training and testing split was at the file level, not the window level. This prevents temporal leakage between training and testing. Since there were plenty of audio files to choose from (and they were consistently labeled), this strategy seemed to make the most sense. The train/test split of files was 75/25. One file per class was guaranteed in the test set, and in combined Paul/Hannah trainings, I guaranteed a 50/50 split of performer WAV files. I performed three types of testing for a total of five train/test runs:

1. Within-performer evaluation: train/test on the same performer – 2 test runs
2. Cross-performer evaluation: train on one performer, test on the other – 2 test runs
3. Combined dataset evaluation: both performers mixed -- 1 test run

3.4.3 Evaluation Metrics

There were several ways I evaluated how the system performed.

1. In step 2, confusion matrices were produced to analyze systematic misclassification between gesture categories in train/test sessions;
2. A script was forked directly from the live detector algorithm that calculated average latency over the 50ms hops. This computed running averages for the average vectorization time and classification time;
3. Analysis was also run to determine which features in the 59-dimensional vector most strongly contributed to classification categories;
4. A 2D PCA graph allowed me to get a general idea of the principal features contributing to separability.

3.5 Producing a Reusable Model for Real-Time Use: Step 3

Once the classifier was evaluated and validated, a file needs to be produced for real-time use. The model trains on the full dataset and saves the trained model to a `.joblib` file. The file contains trained SVM weights, scaling (using `StandardScaler`) and class labels. This ensures that there is an artifact that can be usable in real time.

3.6 Real-Time Detection System: Step 4

3.6.1 Technical Background

This component comprised the final, and most complex part of the process. First, live audio samples must be collected and organized. The incoming audio stream through the Scarlett FocusRite interface is managed by the `sounddevice` library for Python (`https://python-sounddevice.readthedocs.io/en/0.5.3/`), which works with the PortAudio library and is in turn dependent on `numpy`. In order to use `sounddevice` properly I employed a standard operating procedure by connecting `InputStream()` to a callback function `audio_cb()` *inside* the `main()` function. This strategy is advantageous because collecting audio samples and organizing them into vectors needs to be done very quickly.[13] For this project I chose a standard sample rate of 44.1kHz at 16 bits (exactly the same as the WAV files that produced the dataset), and a conservative audio vector block of 1024 samples.[14]

I needed to implement a ring buffer to accumulate audio vectors (keeping the latest 200ms), and discarding older samples continuously.[15] In practice my buffer contains 8820 audio samples (0.2 sec. $\times$ 44,100 samples/sec. = 8820). The ring buffer is essential because it *decouples the real-time audio thread from the processing thread.*

3.6.2 System Design: The Analysis Frame and Voting Policies

Detecting violin gestures is significantly complicated by the fact that gestures vary temporally. While some ways of playing are best thought of as "states", others are structurally more "events". Specifically, crunch, harmonics, tremolo, normal and nothing are continuous, state-type events, whereas pizzicato is discrete. We do not want the classifier to retrigger "pizzicato" if it only happens once: rather, pizzicato is meaningful only at the moment that it happens. So in order to deal with this, I needed to build a hybridized control system that included both elements of a **finite state machine** ("FSM") and an **event-driven system**.

If the SVM detected a continuous event, then the system proceeded through a sliding window vote buffer, continued through a weighted voting transformer, and outputted a state prediction at a regular time interval. Two control values independently determined the temporal context and stability (`vote-state`) and the output granularity/system load (`state-hz`). They are by default set in the following manner: `vote-sec` = 1.75 seconds of voting buffer, and `state-hz` = produce output every 2Hz (0.5 sec).[16]

On the other hand, if the SVM identified a window as pizzicato, it split to another logic pathway. First, it checked to see if the decibel level was above the user-defined threshold (`db >= pizz_db`); second, it checked to see if the predicted margin was high enough (`margin >= pizz_margin`); third, it triggered a *refractory latch* (`refrac_sec`) so that pizzicato was not retriggered until we could be reasonably sure it wasn't happening again. I set this refractory latch quite high to ensure system stability (0.5 seconds), even though I could technically retrigger pizzicato much faster than that if I tried at the violin.

The prediction pipeline was built to be very robust at the expense of speed. My system makes two predictions: an "instantaneous" one ("predicted:") and a smoothed, slower and more accurate estimation ("state:"). Each *predicted* state contributes a vote for the *smoothed state* based on how confident the system is. So, a strong confidence is weighted more highly than a low confidence in voting. Additionally, certain predictions get weighted more than others. After exhaustively testing the system, I found that tremolo was underrepresented. So I lowered its confidence threshold *and* boosted its contribution during the vote

[13] Specifically, the audio callback function extracts the correct audio channel from the interface, converts data to float32, and pushes audio vectors to a queue so that data can be handed off for processing. Using a callback function like this is typical in audio processing because the thread must be very fast, it cannot block anything else, and it can't be asked to do heavy computation. So the whole pipeline looks like this: audio interface – InputStream() – audio_cb() – queue – main loop – ring buffer – feature extraction – SVM – state/event determination – OSC message – Max/MSP.

[14] The FluCoMa MFCC object also defaults to using audio vectors of 1024 samples.

[15] The openSMILE system also used a ring buffer in order to feed audio data into multiple analysis engines (Eyben, Wöllmer, and Schuller 2010).

[16] I used the Python `argparse` library throughout this project, to deal with the large number of variables that needed to be tuned during testing. This way I could tune settings far more safely and rapidly on the command line, without having to constantly go into the code and modify variables manually. One additional way this proved useful was that when moving the code from the development system (a Mac M4) to the deployment system (a MacBook Pro M5) the audio device number assignment changed quite unexpectedly! Later I fixed this somewhat heavy-handedly, by obliging the live detector script to default to any audio interface called "Scarlett".

aggregation in the pool. This is an example of **margin-weighted voting**. I also discovered that in the real-time system, the normal class of playing was frequently being classified as crunch. This was probably due to the change of bow stroke, which introduced a small amount of noise.[17] I tuned this classifier in the voting system so that if the `best_label` was crunch *and* the margin was less than a specific value, the system "fell back" to label the state as "normal". In my system testing, I found this fallback behavior significantly improved the system's accuracy. The system logic is summarized below.

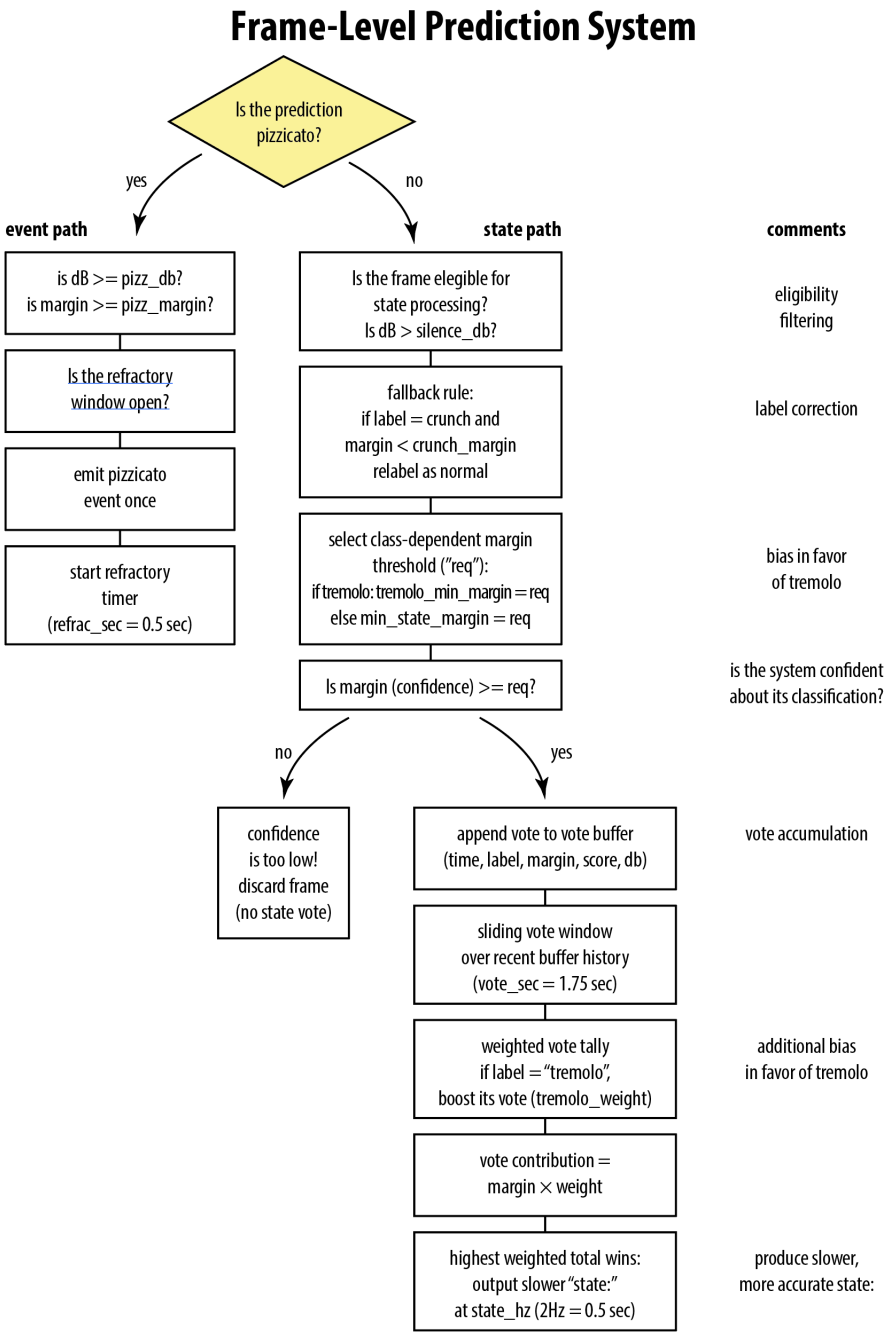


Figure 2: Prediction System Summary

[17]The audio analysis figures above vividly illustrate how noise unavoidably creeps into the system during changes of bow stroke (see section 4.1 below). However these misclassifications could also be the result of a *domain-shift* caused by a slightly different gain setting on the violin, than what was used to record the training samples.

Since my windows are 200ms (with a 50ms hop or overlap), and I set my sliding state window to 1.75 sec. The slower (but more accurate) *state* prediction system evaluated a maximum of 35 candidate frames (depending on the filtering). The voting window *does not* have to equal the output interval: by default the output interval outputs state predictions every 0.5 seconds. Sometimes the system evaluated fewer votes, because if a window was sufficiently quiet (`<= silence_db`, in my case -65 dB), it did not contribute a vote. If the SVM predicted pizzicato, that frame also did not count in the vote pool. If the system's confidence was too low – based on a *class-dependent margin threshold* – (`min_state_margin` or `tremolo_min_margin`), it did not contribute a vote. The structure of the sliding window vote aggregation is at first somewhat hard to grasp, so I represent it in a diagram below.

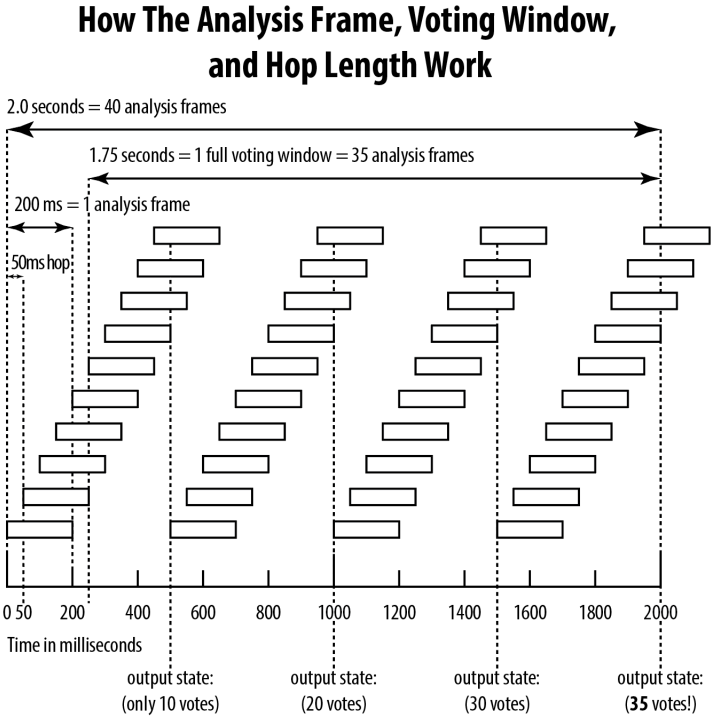


Figure 3: How state output is computed from system startup

3.7 System Integration

Once the system determined its predictions, it sent an OSC (Open Sound Control) message to port 8000 on the system's loopback address (`127.0.0.1`) so that Max/MSP could process it.[18] I used a standard Python library (`SimpleUDPClient` from `pythonosc.udp_client`) to do this. The OSC payload included the current predicted state, the laggy (but more accurate) state determined by voting, a separate message if pizzicato was detected, as well as confidence and margin estimations. Max/MSP unpacks the OSC payload using widely-available externals downloaded from CNMAT (https://cnmat.berkeley.edu/downloads). The composer or sound designer can then easily route control data as desired throughout a patch using Max's ubiquitous `[route]` object.

While Max/MSP can invoke or spawn a Python programming layer using externals, it is generally not a good idea to do so in time-critical applications because Python will run at the mercy of Max's notorious scheduler. However, Max can open an external shell that runs a shell script which, in turn, executes the Python live-detector script! While this is not an optimal architecture, it is easy to implement in the patcher using Max's global routing system `[; launchbrowser]`. Supplying a Max message that starts and stops the violin detection system gives a naive user the agency to control the system *from within the patch itself*,

[18] For more on OSC, see Wright and Freed 1997.

without having to know anything of how to use the command line.

4 Results

4.1 Raw Audio Analysis

In order to gain a better understanding of the raw audio files themselves, I produced graphs of several features using two random audio training files at a time. Because of the radically different scales of some features, several panels are required to capture the information meaningfully.

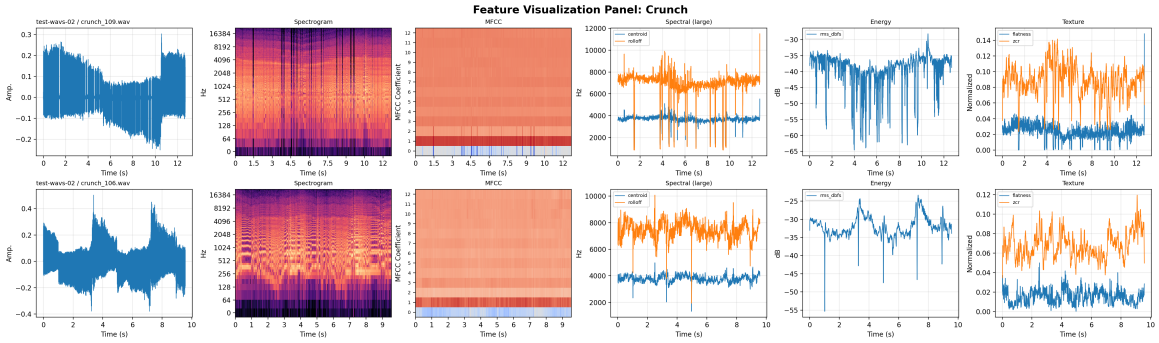


Figure 4: Crunch Analysis

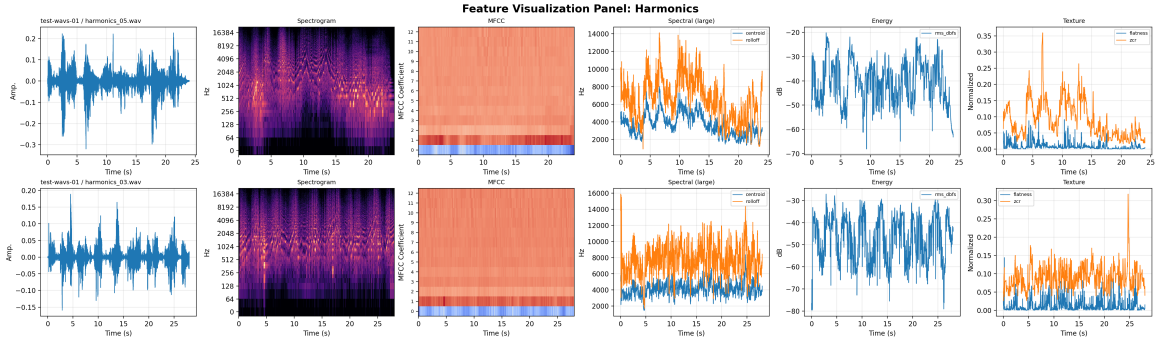


Figure 5: Harmonics Analysis

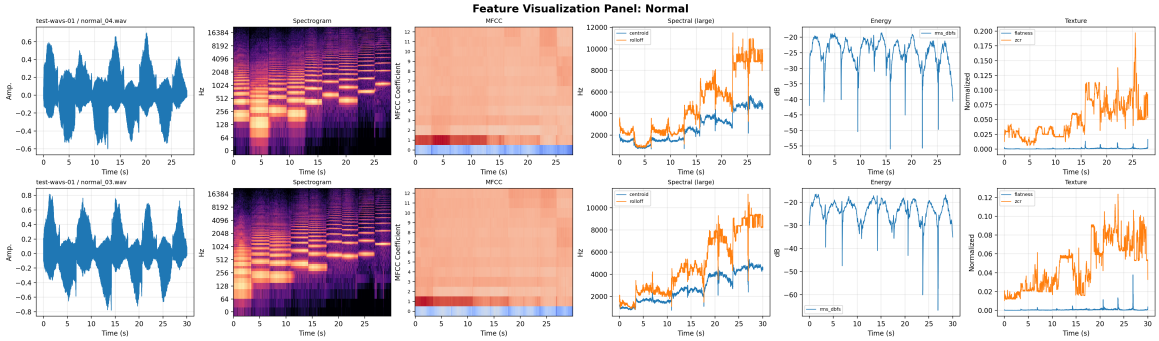


Figure 6: Normal Analysis

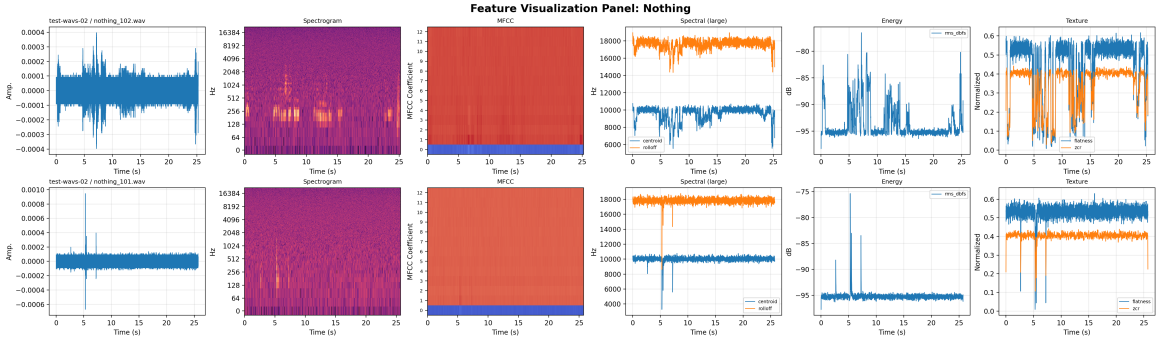


Figure 7: "Nothing" Analysis

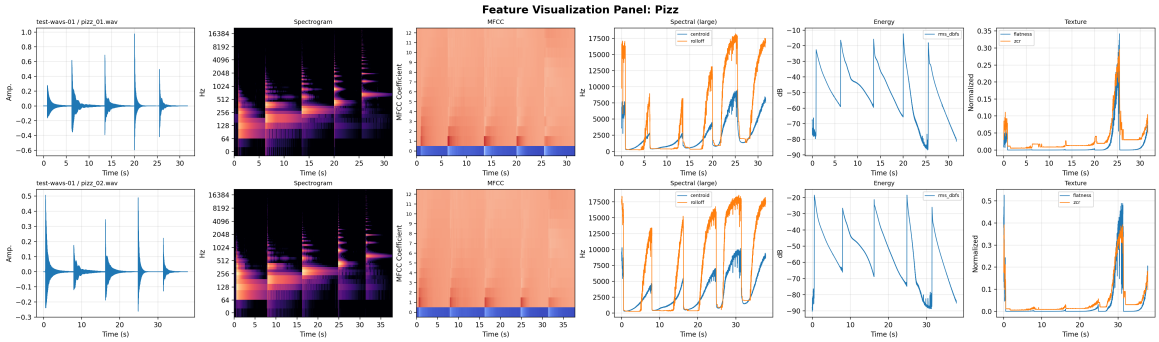


Figure 8: Pizzicato Analysis

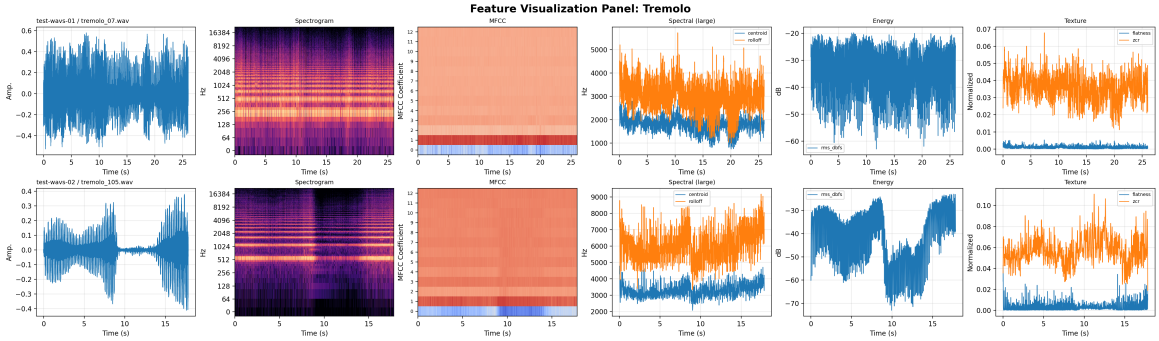


Figure 9: Tremolo Analysis

It is notable that bow changes frequently cause significant discontinuities in the spectral analyses, particularly for the normal and crunch classes. While bow changes introduce noise, they are inevitable in violin or viola playing.

4.2 2D PCA Analysis

I ran a PCA analysis on 1000 of the 44,578 analysis windows in the data. The loadings are shown in the text file below, and a graph is shown in Figure 10.

```
Loading: src/tests/train-test-03-paul-train-hannah-test/features-combined-harmonicity.npz
Keys: ['X', 'y', 'classes', 'file_index', 'win_index', 'sr', 'win_sec', 'hop_sec', 'ignore_sec', 'n_mfcc',
↪  'n_fft', 'hop_fft', 'roll_percent']
Shape X: (44578, 59)
```

```
Shape y: (44578,)

Variance ratios:
PC1: 0.23130
PC2: 0.12287
Total: 0.35417

Loadings for PC1:
--------------------------------------
mfcc_std_06               0.20179
dmfcc_std_01              0.20149
dmfcc_std_09              0.19858
dmfcc_std_06              0.19851
mfcc_std_01               0.19830
mfcc_std_09               0.19645
dmfcc_std_05              0.19226
dmfcc_std_11              0.19221
mfcc_std_05               0.18889
dmfcc_std_04              0.18556
mfcc_std_11               0.18549
mfcc_std_04               0.18346
dmfcc_std_08              0.18242
dmfcc_std_10              0.18241
mfcc_std_08               0.18231
mfcc_std_10               0.18083
mfcc_std_12               0.18022
dmfcc_std_12              0.17908
mfcc_std_07               0.17571
dmfcc_std_07              0.17350

Loadings for PC2:
--------------------------------------
spectral_centroid         0.31093
rms_dbfs                 -0.31075
spectral_rolloff          0.30982
mfcc_mean_00             -0.29400
zero_crossing_rate        0.29138
spectral_flatness         0.27923
mfcc_mean_08              0.23483
mfcc_mean_01             -0.19995
mfcc_mean_06              0.19609
harmonicity_autocorr     -0.19449
mfcc_mean_10              0.18809
mfcc_mean_09              0.18740
mfcc_mean_07              0.18492
mfcc_mean_02              0.16646
mfcc_mean_04              0.15050
mfcc_mean_05              0.13428
mfcc_mean_11              0.12140
mfcc_std_05               0.10169
mfcc_std_09               0.09438
mfcc_std_03               0.09029
```

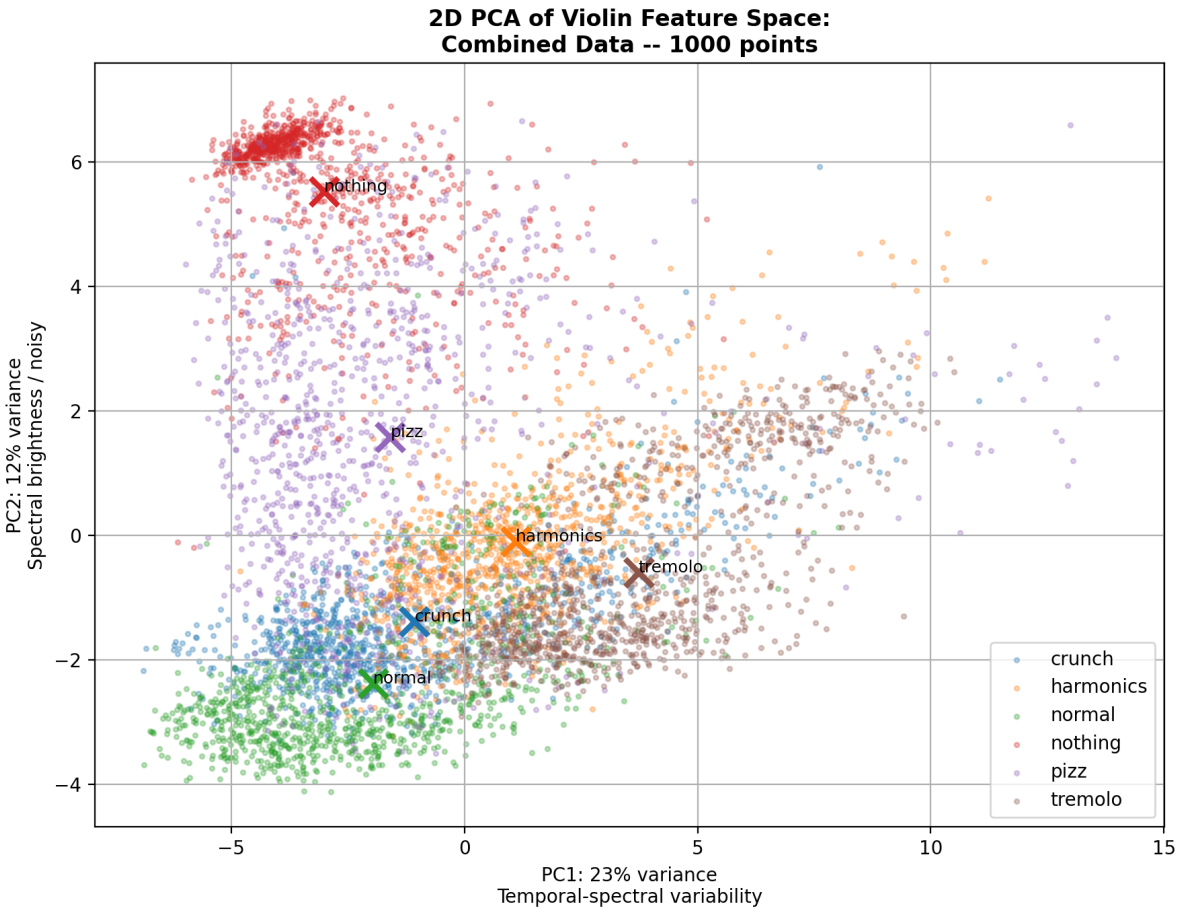


Figure 10: 2D PCA of 1000 analysis windows

4.3 Confusion Matrices

Given that we had two performers (myself and Ms. Bedeck), I ran all possible train/test combinations. The normalized confusion matrices are shown below.

The results from this analysis are quite telling. While within-performer accuracy is quite high (92.5% and 93.7%), cross-training caused a drop in classifier performance. Training on my data and then testing on Ms. Bedeck's recordings yielded slightly worse performance (86%) than the other way around (88%). This implies that the cross-performer evaluation shows lower, but comparable accuracy in both directions, indicating that the learned feature representation captures gesture characteristics that are largely invariant across performers.

The worst cross-performance training/testing the classifier reported was in the "nothing" category. This is principally because during my "silence" samples, there were slight perturbations of the string which were most often misclassified as pizzicato – whereas Ms. Bedeck's samples were almost completely silent. I accounted for these very slight pizzicato misreadings by implementing a noise level gate in the live detector.

Anecdotally, I noticed that Ms. Bedeck's tremolo speed was surprisingly fast and tight – more than I mustered – but she also has an understandably different body posture, connection to the instrument, and technique than mine. She is a younger person with a smaller body size, but practices much more diligently. It would be highly informative to collect performance data from other performers to gain more nuance.

Despite the slight cross-training asymmetries, the model mostly appears to have captured invariant features. When finally trained and tested on combined WAV files, performance was good overall, achieving an overall accuracy across classes of over 93.3%.

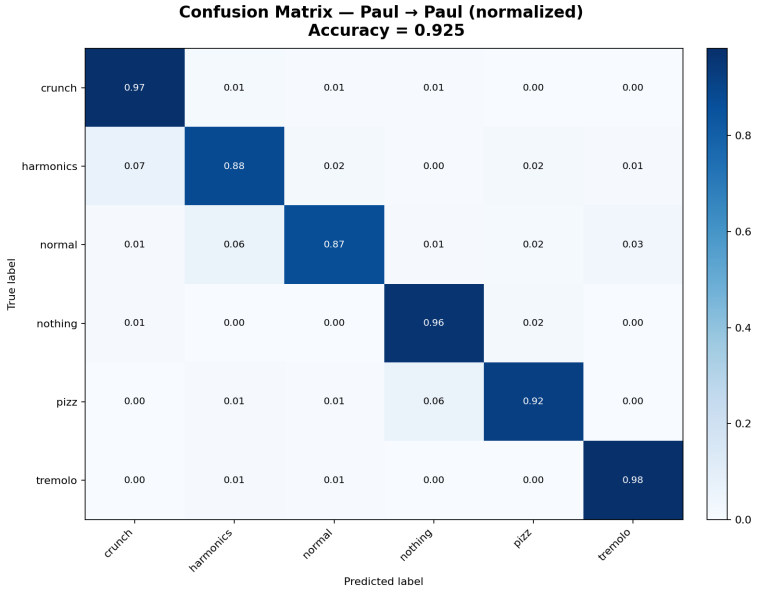


Figure 11: Confusion Matrices: Train on Paul, Test on Paul

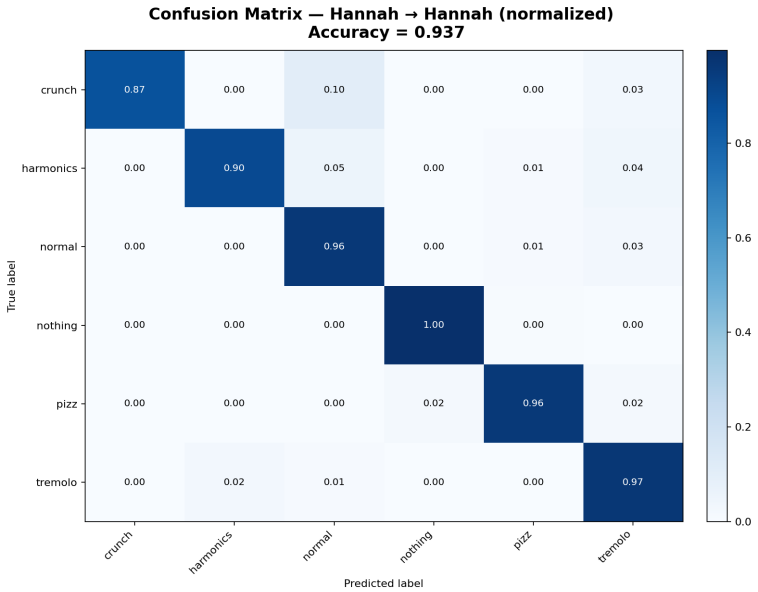


Figure 12: Confusion Matrices: Train on Hannah, Test on Hannah

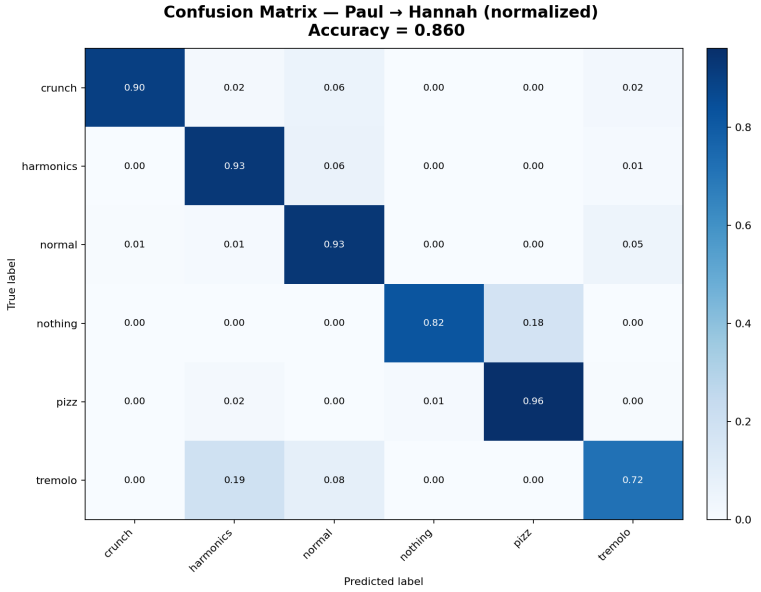


Figure 13: Confusion Matrices: Train on Paul, Test on Hannah

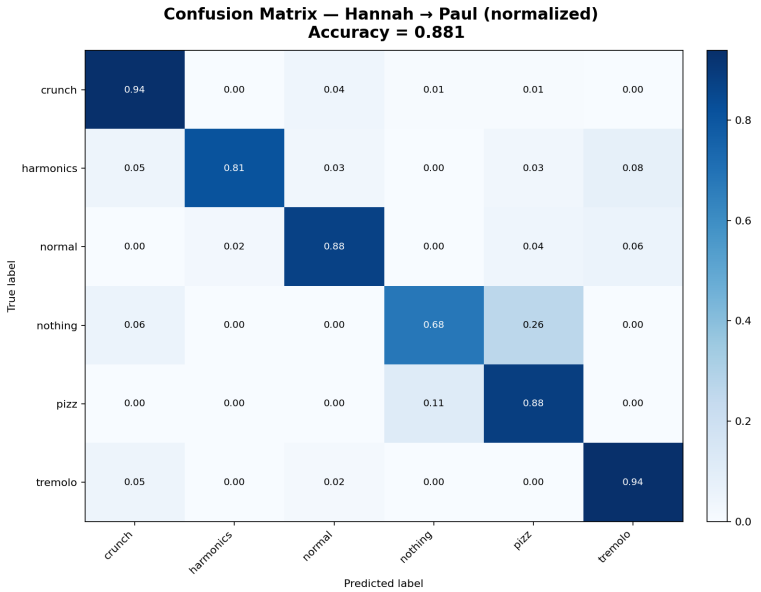


Figure 14: Confusion Matrices: Train on Hannah, Test on Paul

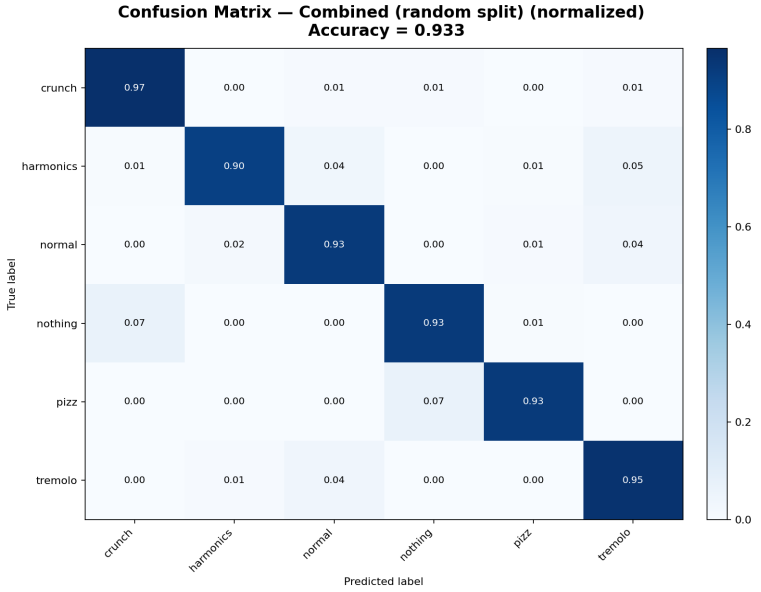


Figure 15: Confusion Matrices: Combined Train/Test split

4.4 System Latency

4.4.1 Background

The system's latency is affected by three primary factors:

1. **Buffer/window latency**. Since the analysis window is 200ms, in the worst case a gesture begins immediately after a window boundary and must wait nearly the full window duration before sufficient new data is available for analysis. In worst case, this contributes up to 200ms of delay. On average, this contribution is approximately half the window size, or 100ms.

2. **Hop latency.** The system's hop size is 50ms, meaning that feature extraction and classification occur at discrete 50ms intervals. In the worst case, the system must wait the full hop duration before processing a newly available window; so on average, this delay is approximately 25ms.

3. **Computational latency.** This is the time required to compute features and perform classification once a window is available. I estimated this to be 5-15 ms.

Combining these factors yields an **estimated average end-to-end latency** of approximately 140ms (100ms + 25ms + ~15ms), and an **estimated worst-case latency** of approximately 265ms under ideal scheduling conditions. If buffering and scheduling jitter creep in, this value could be higher. I surmised that the overall system latency would be dominated by buffering and hop scheduling, rather than computation.

By contrast, the system's "state" output operates over a much longer temporal horizon. Computed every 0.5 seconds, it incorporates 1.75 seconds of recent history and is therefore intentionally slower but considerably more stable and accurate.

A latency of even 100ms can feel sluggish in a performance context. One way to reduce this delay would be to decrease the window size (e.g., to 100ms) and hop size (e.g., to 25ms). However, this would likely reduce classification accuracy, particularly for sustained or temporally evolving gestures. Performance features such as tremolo and normal bowing require sufficient temporal context to stabilize. Inspection of waveform and feature visualizations above suggests that such gestures do in fact emerge over periods of time that are consistent with a 200ms estimate.

However, reducing the window and hop sizes would also increase the relative computational load. More frequent analysis cycles require more feature extraction and classification, placing greater demand on the

CPU. Although the classifier itself is quite efficient, I found that the feature extraction stage dominates computational cost (relative to classification). Moreover, the system is coupled to a Max/MSP patch, which may itself impose significant real-time processing demands. It is therefore advantageous to keep the analysis pipeline lightweight.

4.4.2 Measured

Empirical measurements over 500 analysis cycles showed an average computational latency of approximately **11ms**. This represents a small fraction of the total latency budget and was found to be stable across time. Further analysis revealed that feature extraction accounted for the vast majority of this cost (approximately **10.7ms** per cycle), while classification using the linear SVM required only about **0.35ms**. This confirms that spectral and statistical feature computation is the primary computational expense in the system, whereas classification itself is effectively negligible.

The average total latency in the system breaks down in the following way, *assuming uniform distribution of event onset within the analysis window*:

Cause	**Time (Average)**
Window lag	100ms (window duration / 2)
Hop lag	25ms (hop duration / 2)
Feature Extraction	10.7ms
Classification	0.35ms
Total:	**136ms**

Table 1: Overall measured system latency

4.5 Classifier Reports

I ran an analysis to determine which features most strongly contribute to the `LinearSVC`'s classification decisions. The results are shown below along with some interpretation.

Class	**Top Features**	**Interpretation**
Crunch	spectral_centroid mfcc_mean_00 mfcc_mean_01	Bright, high-energy broadband spectrum
Harmonics	zero_crossing_rate spectral_centroid harmonicity_autocorr	Harmonically structured spectrum, tonal spectrum with low noise
Normal	mfcc_mean_00 harmonicity_autocorr mfcc_std_01	Stable spectrum, strongly harmonic sustained tone
Nothing	mfcc_mean_00 spectral_flatness mfcc_mean_04	Low spectral activity, very little temporal variation
Pizzicato	rms_dbfs zero_crossing_rate rms_std_micro	Transient, percussive, high-energy impulse
Tremolo	spectral_centroid harmonicity_autocorr dmfcc_std_01	Harmonic signal with strong temporal modulation

Table 2: Features with the highest absolute model weights, for each gesture class (using LinearSVC)

4.6 Max/MSP Calibration Patch and User Features

A minimal Max/MSP patch serves to orient the user and provide minimal functionality. This patch is designed like a typical Max "help file" and invites the composer to mutate it.

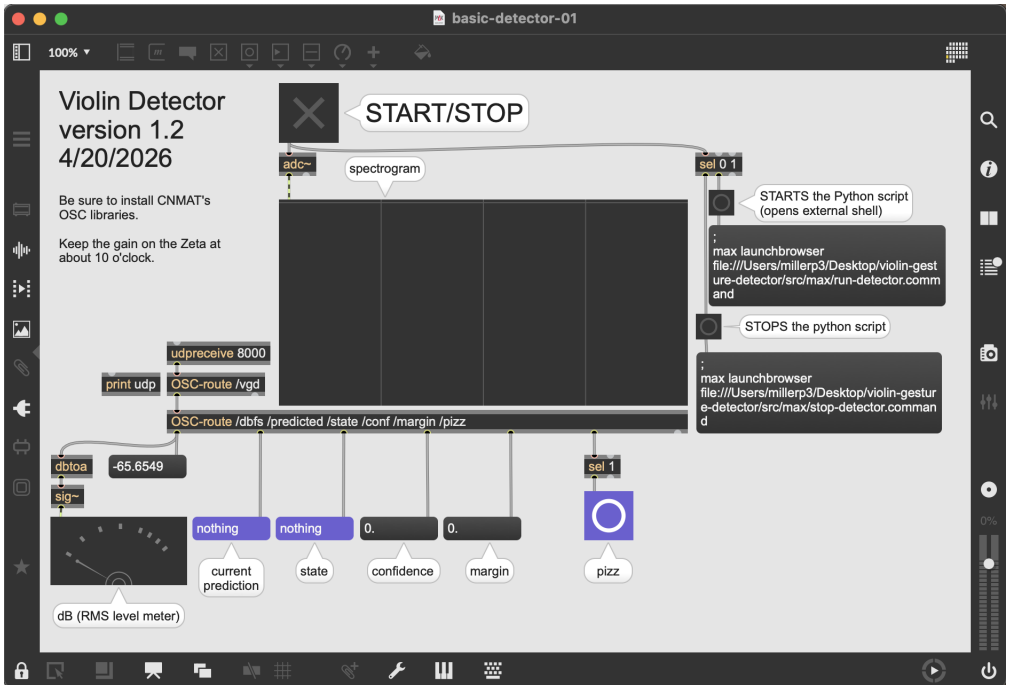


Figure 16: Minimal Max/MSP Calibration Patch

5 Discussion

5.1 System Behavior

In performing with this system, I found the lag time was noticeable but acceptable. In a future iteration, it might be worth analyzing accuracy gains vs. CPU usage when shorter analysis windows are used, under 200ms.

5.2 Limitations of the system and alternatives

The system is limited by number of gestures that it can recognize. In the future, it might be useful to capture additional performance data. To increase accuracy, it might even be possible to integrate some kind of sensor on the bow, to provide additional information about how the performer is interacting with the instrument.

As stated above, model training was done on 88 recordings from two different performers, which expanded to 44,578 windows. The system therefore has good statistical density but limited performer diversity. This is indicated by the fact that there is only 86% – 88% cross-performer accuracy. Adding more players would potentially expose the model to different bowing styles, expand the feature-space coverage, and reduce overfitting to individual technique. Although accuracy might increase slightly, robustness would significantly improve.

Is the linear SVM the best classifier for this sort of project? The linear SVM classifier works very fast at inference (which is critical for real-time applications), works well with high-dimensional features, is robust for correlated features, and produces results that are relatively easy to interpret. Its costs are that it offers only linear decision boundaries, and it cannot model very complex interactions unless the features themselves encode them. Given that it achieved better than 93% accuracy, it seems to be a fairly good approach.

A nonlinear SVM could model nonlinear class boundaries, and could improve performance when classes overlap in complex ways. However, it is much slower at inference, which can be problematic in a real-time application. It requires some tuning, it is often less interpretrable, and the memory usage grows significantly with the number of support vectors. Using it might provide some modest gains, but they would arguably not be dramatic.

On the other hand, logistic regression is very fast, and produces true probabilities. It is also somewhat simpler than SVM. But, it may perform slightly worse on tightly clustered data. Logistic regression would be a reasonable alternative to linear SVM, and might be a useful option to keep in mind for a future version.

Random forests would also be a possible approach. It can capture nonlinear relationships, is easy to scale up with features, and it's easy to use. But it is often slower than linear models. It could help with complex class boundaries, but it might not be ideal because of the real-time requirements.

Overall, I think the best way to improve the accuracy of the system in the near term is to explore training on more performers. Arguably the main limitation to the model is data diversity, not model capacity.

5.3 Musical and Artistic Implications

In a system like this – where there is lag and accuracy is still "only" 93% – any compositional system needs to be somewhat fault-tolerant and slow-paced. I built a simple demonstration of compositional possibilities. The core is a drone with a fundamental centered an octave below middle C (C3 = 130.81 Hz – the lowest string on the Zeta violin,) with 16 partials arranged in the harmonic series. The frequency values were sent through a resonant bandpass filterbank operating on pink noise, with very high Q values.

Crunch playing increases harmonic jitter and adds overdrive (clipping), making the drone less stable and more distorted. **Pizzicato** interpolates the mixer towards sending more pointlike events and less continuous "drone" to output. **Tremolo** increases the probability of those overlapping pointlike events. **Harmonics** increase the center frequency of a global bandpass filter, making the whole output significantly brighter, while scaling gain on the upstream bandpass filterbank to stabilize output amplitude. **Normal** playing increases the tendency of the system to regain equilibrium.

This is by no means a very nuanced compositional system, but was intended simply to showcase one very crude way of using the gesture detector's output to affect timbral qualities of a drone in a live musical system.

6 Conclusion

The violin detector system can reliably detect five different styles of violin playing (plus "silence") common in classical and extended techniques. The system demonstrates that real-time classification of violin performance gestures can be achieved using a relatively lightweight feature set, and a linear support vector machine. It operates with a carefully structured pipeline in which audio is transformed into short overlapping windows, mapped to a 59-dimensional feature space, and classified both at the immediate "frame" level and at a slower but more accurate "state" representation.

A central finding of the project is that system latency is not dominated by computation, but by buffering and seemingly unavoidable temporal analysis constraints. While feature extraction and classification required only about 11ms per cycle, the effective end-to-end latency is governed by window size and hop interval, resulting in an average response time on the order of 130-140ms. This demonstrates a quite purposeful design choice which traded responsiveness for accuracy, particularly for violin gestures that require temporal context to stabilize or emerge.

The resulting system, which communicates with Max/MSP via OSC, provides a flexible and extensible framework for gesture-driven musical interaction, opening up a sophisticated and more intertwined possibility for human/machine collaboration and interaction. The system could be extended to provide further details to components already implemented. For example, it would be possible to estimate the speed at which a violinist is performing tremolo. However, it would also probably be relatively straightforward to make the system sensitive to other kinds of playing, for example: *sul tasto* or *sul ponticello*, *col legno*, vibrato analysis, *spiccato* or *legato* bowing, or even bow pressure estimations. It would also be very interesting to detect the onsets and boundaries of gestures in some kind of gesture segmentation analysis. This could involve

a Recurrent Neural Network (RNN) that could process sequences by remembering past inputs, or even an LSTM ("Long Short-Term Memory") system that could introduce gates to control memory states.

The structures in the live audio analysis system presented here, are very similar in spirit to those in live video analysis systems in robotics or self-driving vehicles.[19] Whereas in audio we talk about windows of 150–300ms with a 50ms hop, with features encoded as MFCCs, RMS, harmonicity, spectral centroid, etc., in live video processing ("computer vision") we are often using 8–32 frames per window (250–1000ms), with a hop of 1–4 frames, and features defined as motion vectors, optical flow, or CNN embeddings. Parallel sensor systems such as LIDAR are also typically incorporated into the sensory bank as well as auditory sensors. Gesture classes in audio could be states such as tremolo, crunch or harmonics, or events like pizzicato. In video the states are frequently walking, running, standing, whereas events are jumping, clapping or falling. A key difference between audio and video detection systems is that video systems must include spatial variation elements such as position, scale and rotation – whereas in audio systems there are no such classifiers. So an audio detection system can be much simpler and more interpretable than a video system.

References

Bogert, Bruce P., Michael J. R. Healy, and John W. Tukey (1963). "The Quefrency Analysis of Time Series for Echos: Cepstrum, Pseudo-Autocovariance, Cross-Cepstrum and Sphe Cracking." In: *Proceedings of the Symposium on Time Series Analysis*. Ed. by Murray Rosenblatt. New York: John Wiley & Sons, pp. 209–243.

Cooley, James W. and John W. Tukey (1965). "An Algorithm for the Machine Calculation of Complex Fourier Series." In: *Mathematics of Computation* 19.90, pp. 297–301. DOI: 10.1090/S0025-5718-1965-0178586-1.

Davis, Steven and Paul Mermelstein (1980). "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences." In: *IEEE Transactions on Acoustics, Speech and Signal Processing* 28.4, pp. 357–366.

Eyben, Florian, Martin Wöllmer, and Björn Schuller (2010). "openSMILE: The Munich versatile and fast open-source audio feature extractor." In: *Proceedings of the ACM Multimedia Conference*.

Grey, John M. (1977). "Multidimensional perceptual scaling of musical timbres." In: *Journal of the Acoustical Society of America* 61.5, pp. 1270–1277.

Logan, Beth (2000). "Mel frequency cepstral coefficiens for music modeling." In: *Proceedings of the International Symposium on Music Information Retrieval (ISMIR)*.

McFee, Brian et al. (2015). "librosa: Audio and music signal analysis in Python." In: *Proceedings of the 14th Python in Science Conference (scipy 2015)*. Ed. by Kathryn Huff and James Bergstra, pp. 18–25. URL: https://brianmcfee.net/papers/scipy2015_librosa.pdf.

Müller, Meinard (2015). *Fundamentals of Music Processing: Audio, Analysis, Algorithms, Applications*. Springer.

Peeters, Geoffroy (2004). *A large set of audio features for sound description (similarity and classification) in the CUIDADO project*. Tech. rep. IRCAM.

Tremblay, Pierre Alexandre et al. (2022). *The Fluid Corpus Manipulation Toolbox (v.1)*. DOI: 10.5281/zenodo.6834643. URL: https://doi.org/10.5281/zenodo.6834643.

Tzanetakis, George and Perry Cook (2002). "Musical genre classification of audio signals." In: *IEEE Transactions on Speech and Audio Processing*. Vol. 10. 5, pp. 293–302.

Wright, Matthew and Adrian Freed (1997). *Open Sound Control: A New Protocol for Communicating with Sound Synthesizers*. Tech. rep. CNMAT, University of California, Berkeley. URL: https://opensoundcontrol.stanford.edu/files/1997-ICMC-OSC.pdf.

[19] A Pittsburgh company that started at CMU has had significant success fielding self-driving trucks using systems similar in spirit to the live violin detector: http://www.aurora.tech.

A Real-Time Violin Gesture Detector

paul v. miller // machine learning 2026

1.1 The Problem or Challenge

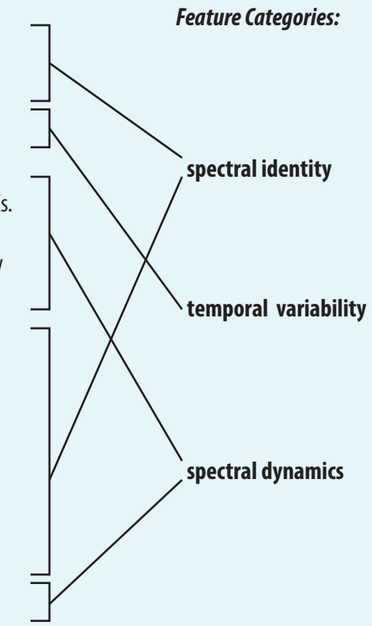
Can machine learning techniques be used to recognize the way a violinist is playing in real time? This is a complex problem because we cannot rely simply on pitch recognition, amplitude or other conventional measurements, but rather have to analyze a much broader, multidimensional set of musical phenomena such as articulation, modes of sound production, and subtle variations in timbre and spectrum.

1.2 System Overview

live audio from electric violin	→	feature vector	→	classifier (linear SVC)	→	OSC (open sound control)	→	Max/MSP patch
sounddevice library; sample rate = 44.1kHz; audio vector of 1024 samples; 5-string Zeta electric violin with a carbon fiber Joule Codabow; Focusrite 2i2 4th generation audio interface		59 dimensions; three categories: spectral identity, temporal variability and spectral dynamics		A low-latency efficient classifier to operate in higher-dimensional space		Protocol developed at Stanford university for sending performance data through networks or loopback; can send a much wider range of data than MIDI		Standard patching (visual coding) environment that makes it easy to use incoming data to affect processes in the audio or video domain

2. Feature Space – 59 dimensions

1. **MFCC-Based Features – 26 dimensions**
   a. MFCC Mean – 13 dimensions. These are time-averaged MEL-frequency cepstral coefficients over the analysis window (200ms with 50ms jumps).
   b. MFCC Standard Deviation – 13 dimensions. These measure the temporal variability of the spectral envelope within the window.
2. **Delta MFCC Features – 26 dimensions**
   a. Delta MFCC Mean – 13 dimensions. This is a time-averaged first derivative of ther MFCCs. These features capture systematic trends in spectral change over time.
   b. Delta MFCC Standard Deviation – 13 dimensions. These features measure the variability of spectral change rates.
3. **General Spectral Features – 5 dimensions**
   a. Spectral centroid – perceived brightness of sound
   b. Spectral rolloff – where does most of the spectral energy lie?
   c. Spectral flatness – broadband noise vs. sinusoidal wave?
   d. Zero crossing rate (ZCR) – how many times does the signal crosses the zero boundary?
   e. Harmonicity: autocorrelation-based – how much periodicity is in the signal (a time-domain measure – very costly calculation)
4. **Signal Energy and Temporal Features – 2 dimensions**
   a. RMS (in dBFS) – overall signal energy
   b. RMS micro-variance – short-term amplitude fluctuations (like tremolo)

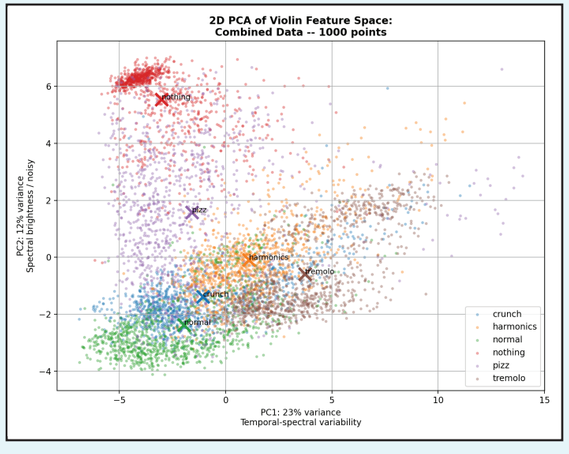


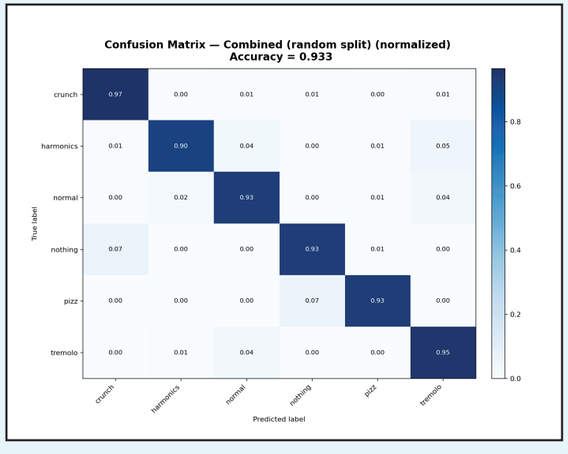
3. Classification and Performance

There were six categories of playing, corresponding to traditional and extended playing techniques:

crunch – pizzicato – harmonic glissandi – tremolo – "normal" – silence

We recorded 37 minutes of audio samples split evenly between two performers, and ran all train/test combinations. Although the 2D PCA analysis showed a lot of feature overlap, the linear SVM classifier performed well. The combined train/test confusion matrix shows **over 93% accuracy**. Training/testing included 88 audio files and 44,578 spectral windows (split 75/25 train/test). After tuning the live analysis model and implementing a tuned, rolling temporal voting system over 1.75 second intervals (35 windows,) I achieved high accuracy in real-time classification.

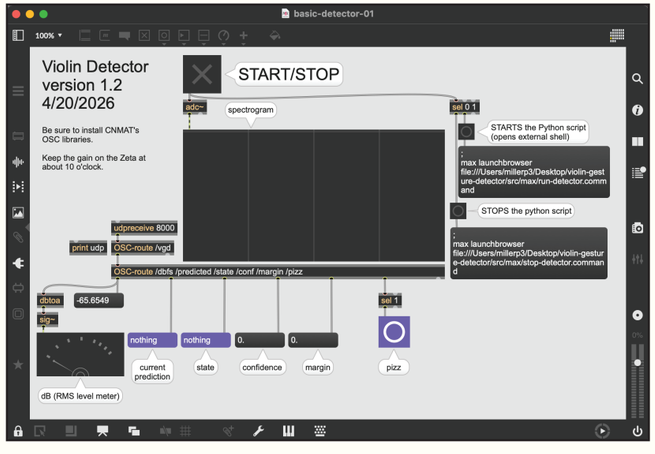




4. Outcomes and Results

4.1 User Interaction and Front End

A Max/MSP patch receives the system's predictions, and data can be easily routed using the OSC tags. I built a very basic minimal "help" patch for end users. This includes a built-in mechanism that allows naive users to start and stop the system by spawning a shell, which runs a simple shell script, which in turn executes the Python live detector script. Another shell script kills the Python process. The patch is shown below.



4.2 System Latency

As with any realtime system, latency is a significant concern. As the window duration is 200ms and the hop size is 50ms, lag will contribute on average to 125ms of latency. As measured over 500 analysis cycles, average computational latency was **11ms**. Feature extraction accounted for the vast majority of this cost while classification by the linear SVM cost very little:

Cause	Average Time
Window lag	100ms (window duration / 2)
Hop lag	25ms (hop duration / 2)
Feature Extraction	10.7ms
Classification	0.35ms
Total:	*136ms; worst-case 263ms*