Harmonic, Statistical, and Topological Methods for Audio Classification

Justin Rogers

Advisor: Dr. Mark Panaggio

May 10, 2018

Abstract

A four-second audio file contains at least 64000 separate data points. If the sound is a musical note, there is a much more succinct description: "Saxophone playing B4." Our goal is to complete this classification algorithmically: can we give a computer a good ear for music?

To attain these results, we apply techniques from topological data analysis to construct *persistence diagrams*, which provide geometric descriptions of the frequency spectrum. This reduces file size by a factor of 600, with negligible information loss. Using the persistent approach, we introduce several novel algorithms for pitch classification. In particular, the "PersistFFT2" algorithm is significantly more accurate than the other algorithms, indicating an advantage of the topological approach.

To attain our final results, we use an ensemble learning technique to combine all our algorithms. This produced a classifier which exceeded each of its components: our final result was 90% pitch accuracy and 91% instrument accuracy on the test dataset.

Contents

1 Problem, dataset, and background

1.1 Motivation

If you hear two musical notes, it is easy to tell if they are identical. It is also easy to determine the type of instrument: brass, string, and so forth. With some training, the average person can even identify specific notes with high accuracy. This sort of listening is done on an intuitive level— can we accomplish the same result without intuition?

Suppose you are deaf, and you are given a book that contains 64000 numbers, representing regularly-spaced measurements of air pressure. You are told that these numbers represent a musical note, played on an unknown instrument. Given a calculator, can you determine that note and instrument?

This problem is the focus of our paper. Although it is a fairly specific problem, our methods work for general audio. For instance, this work might be adapted for problems in speech classification. The topological perspective from section 2.5 is still more general: it is applicable to any periodic phenomenon.

1.2 Main problem

Given an audio file that contains a recorded note from some instrument, how can we determine the fundamental frequency of that note? What algorithms might be useful for this, in terms of accuracy and computational efficiency?

1.3 Secondary problem

Given that same audio file, can we classify the instrument as one of these 11 possible families?

0. Bass
1. Brass
2. Flute
3. Guitar

4. Keyboard
5. Mallet
6. Organ
7. Reed
8. String
9. Synth Lead
10. Vocal

1.4 Dataset

We used the NSynth Dataset [4], published by TensorFlow Magenta. This is a very large dataset, containing 305000+ notes. Each note is a 4-second monophonic .wav file sampled at 16 kHz. The dataset spans 1000+ instruments, 5 different velocities, and all the pitches on a standard MIDI piano (21-108, or A0 to C8). This includes acoustic, electronic, and synthetic instruments.

As the dataset is extremely large, we used only a small portion of it: the set of 4096 examples labeled "Test" on the dataset's webpage. Some of the files in this dataset seem to be included erroneously: there are a handful of pathological notes that are officially labeled with bad pitches (below 21 or above 108). I removed these 36 pathologies and notified Magenta via their github. At this time I have not received a response, but for details see my full error report [8].

1.5 Digital audio background

The .wav format stores audio as a time series at a constant sample rate. We convert each note to a 1-by-64000 array of floats for analysis.

MIDI is an international standard for digital music and instruments. A note's **pitch** is given by an integer from 0 to 127: these correspond to standard piano keys. MIDI 69 is equal to A4, which is defined to be 440hz. MIDI 70 is A$^\sharp$4, and so forth. Knowing one frequency determines the rest: if we know the frequency of note n is ν, then the frequency of note (n+1) is $2^{1/12} * \nu$. This gives us the formula for the frequency of a note N:

$$freq(N) = 440 \cdot 2^{|N-69|/12}$$

This can easily be inverted. The inverse transformation gives us a method of converting arbitrary frequencies to "fractional MIDI notes," which formalizes the idea of a note halfway between adjacent piano keys. Most of the prediction algorithms we discuss will predict frequency, convert to fractional MIDI, and round to produce the final prediction.

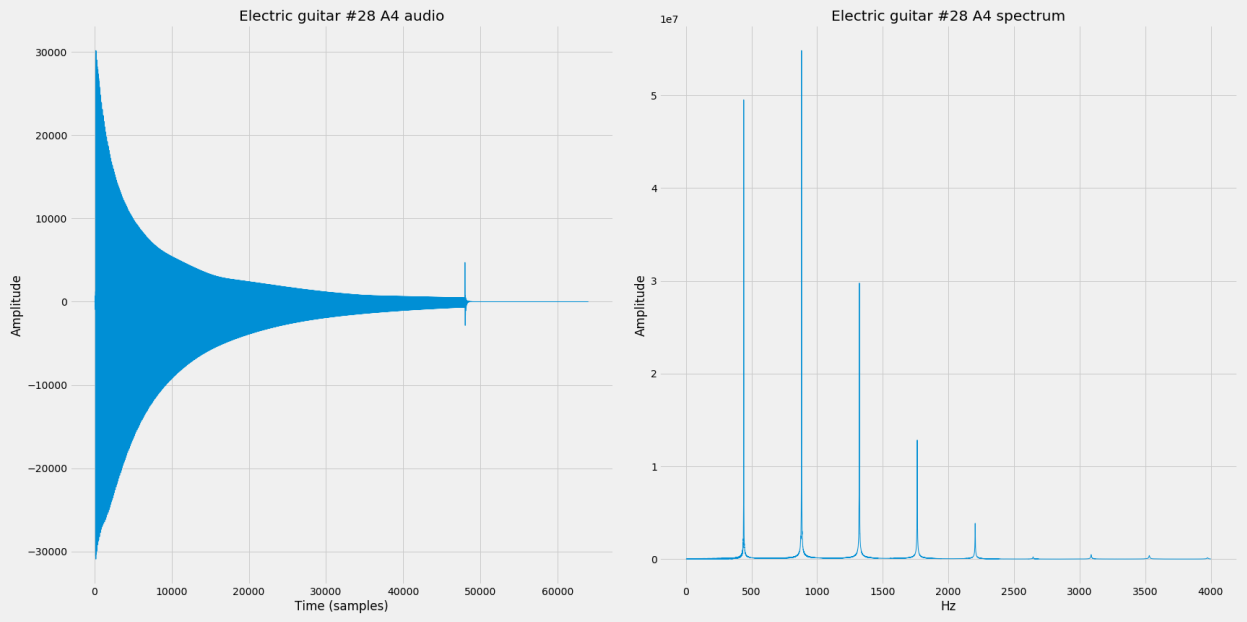
For each note, MIDI also includes a number from 0 to 127 called **velocity**, which is similar to a musical dynamic.[1] Notes in the NSynth dataset are drawn from the following list of velocities: (25, 50, 75, 100, 127).

1.6 Defining pitch: fundamental frequency

An individual note is composed of many constituent frequencies, but we generally perceive a note as having a single pitch: the lowest frequency component of the note. We define the frequency spectrum more precisely in section 2.1.

The **fundamental period** of a note is its period: the length of time it takes for a single cycle, ignoring sound damping or decay. From this, we can determine the **fundamental frequency** of a note: this is defined as the reciprocal of the fundamental period. We define the **pitch** of a note to be its fundamental frequency.

In a musical context, playing a single note of frequency ν will also create frequency components at frequencies that are integer multiples of ν. These frequency components are called **harmonics**. For instance, if we play a note at 440 Hz, we expect to see significant frequencies emerge at 880 Hz, 1320 Hz, and so forth. For an example, see figures 1, 2 and 3.



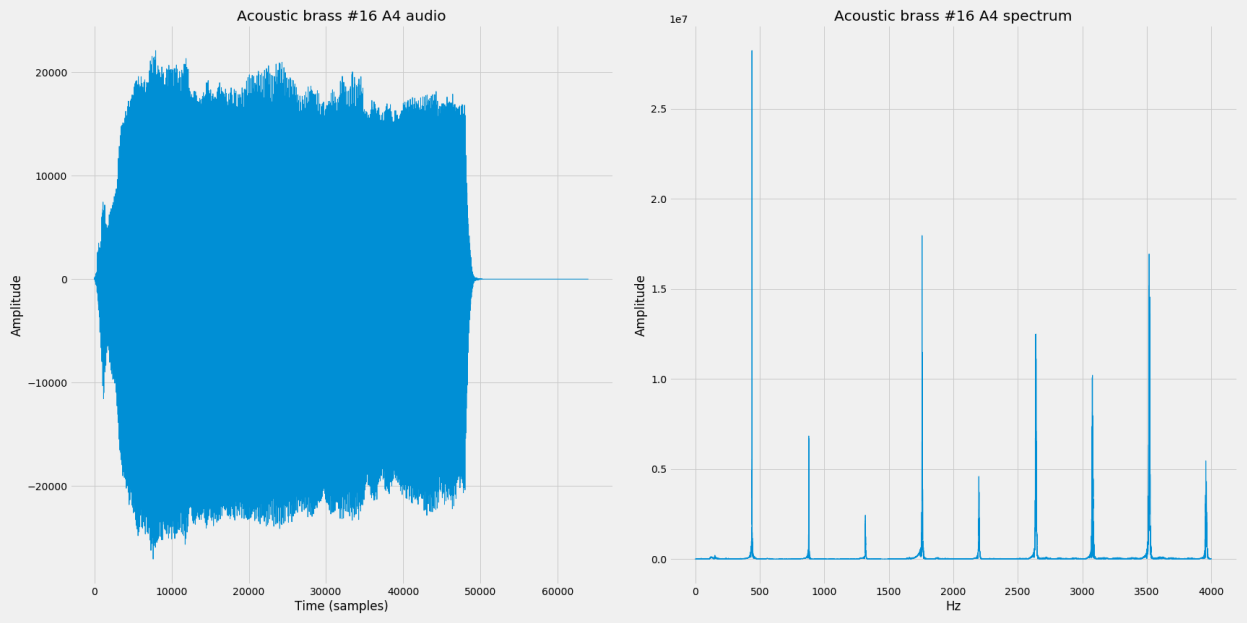
(a) Audio data (b) Spectrum

Figure 1: Signal and spectrum of an electric guitar playing A4 (440 Hz).

1.6.1 Difficulties in peak detection

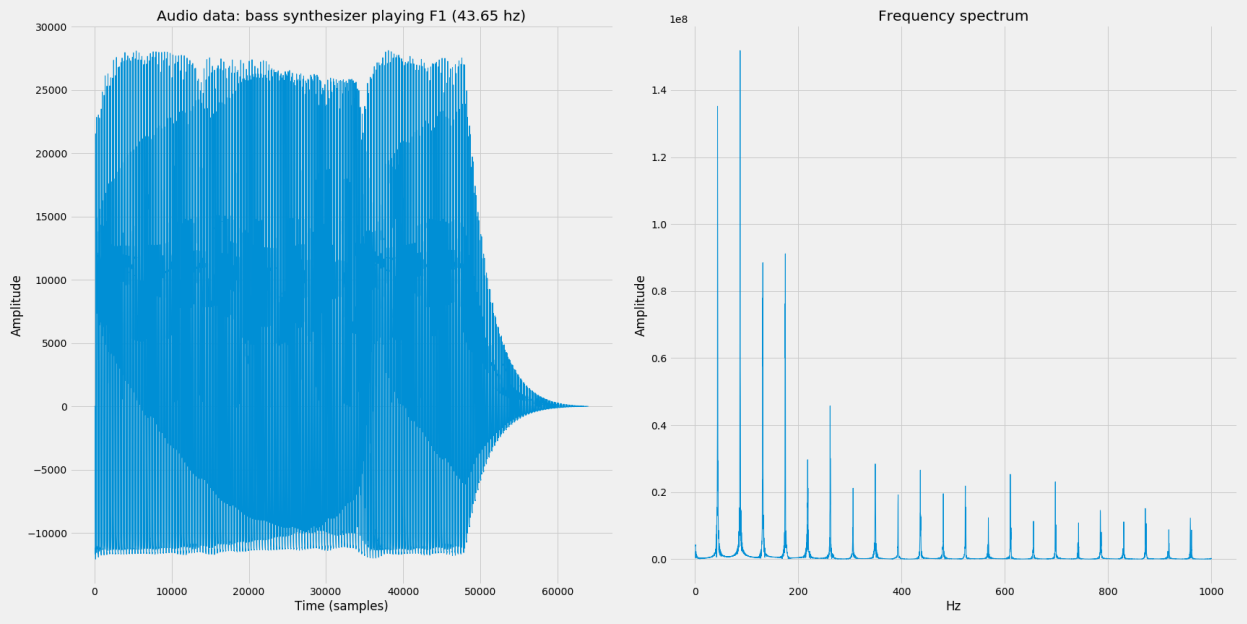
It would clearly be very beneficial to know the location of all the harmonics: this should allow us to determine the pitch. Visually, we see that the harmonics are usually located at

[1] E.g., fortissimo. This measures the sound level and power of the note: large velocities correspond to loud notes.



(a) Audio data (b) Spectrum

Figure 2: Signal and spectrum of an acoustic brass instrument playing A4 (440 Hz).



(a) Audio data (b) Spectrum

Figure 3: Signal and spectrum of a synthesized bass playing F1 (43.65 Hz).

steep peaks in the spectrum, so we need a way to find these peaks. This is a more difficult problem than it might seem: there is not a trivial way to define "peak" to fit our needs.

Suppose, for instance, that we want the five largest peaks. If we take the five largest elements of the spectrum, they might all be adjacent. As an example, we might get (439.5, 439.75, 440, 440.5, 440.75) as our five biggest points– but they're clearly part of the peak at 440 Hz. Our definition needs to account for this.

We might consider looking at the local maxima of the spectrum. This doesn't work very well either, since most of our spectra are rather erratic. Even if we apply a smoothing filter to the spectrum, there are still far too many local maxima. This suggests that we need a more nuanced definition: instead of "maximum" and "non-maximum", we need some notion of *scale*. If a local maximum is much larger than its neighbors, it is likely to be significant, but how can we quantify "much larger"?

After developing the necessary definitions, we will return to this problem in section 2.5 with a topological approach.

2 Mathematical background

We require two pieces of mathematical machinery for most of our algorithms: the discrete Fourier transform and persistent homology. Informally, the DFT converts our signal to its frequency spectrum, and we apply persistent homology to identify the significant peaks in that spectrum. These peaks generally correspond to musical harmonics, which we use to classify the note.

To define persistent homology, we also need to define simplicial complexes and simplicial homology. These topics are not used outside of the persistent case.

2.1 Discrete Fourier transform

Suppose we have a map $x_n : \mathbb{Z}_n \to \mathbb{C}$, i.e. a sequence of n complex numbers. The discrete Fourier transform is an operator $\mathscr{F}$ that converts this to another map or sequence, $X_k : \mathbb{Z}_n \to \mathbb{C}$. We may define this precisely:

$$\mathscr{F}(x_n) = X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n/N}.$$

It is equivalent to consider x_n and X_k as elements of $\mathbb{C}^n$. We see that $\mathscr{F} : \mathbb{C}^n \to \mathbb{C}^n$, so $\mathscr{F}$ is only a linear transformation. By considering the image of our basis vectors, we may represent $\mathscr{F}$ with an n-by-n matrix. This matrix is Hermitian, hence easily invertible.

This latter perspective is generally the more practical and more common viewpoint. The former perspective is beneficial for those who are already familiar with the continuous Fourier transform: the discrete case is closely analogous.

Computing the Fourier transform of a time series (x_n) gives us information about its constituent frequencies (X_k). The discrete Fourier transform approximates the continuous Fourier transform: given some frequency k, X_k tells us the relative importance of that frequency in our signal. The output is the *frequency spectrum* of that signal, and it describes all the frequencies that compose the original signal. By inverting the matrix of $\mathscr{F}$, we may reconstruct the original signal out of these frequencies.

The discrete Fourier transform can be accomplished efficiently through an O(n log n) algorithm known as the Fast Fourier Transform (FFT). [7] To work with the data, we take the absolute value (i.e., complex modulus) of the DFT. This is a standard procedure that discards phase information while retaining amplitude information.

We have already seen some examples of frequency spectra: figures 1, 2 and 3.

2.2 Simplicial complexes

2.2.1 Intuition

A 0-simplex is a single point, a 1-simplex is a line, a 2-simplex is a filled-in triangle, and a 3-simplex is a solid tetrahedron. These objects are convenient to work with, because they are characterized completely by their vertices. To describe a mathematical tetrahedron, one only needs to specify four vertices and say "consider this tetrahedron." There is a shorthand

notation for this: we write $\sigma = conv(v_0, v_1, v_2, v_3)$ to indicate that σ is the tetrahedron described by the vertices (v_0, v_1, v_2, v_3).

There is a degenerate case we would like to avoid: if all four vertices are coplanar, then σ would not be a tetrahedron. Similarly, if we want to describe a triangle by its vertices, we would like to ensure that the vertices are not collinear. To avoid these exceptional cases, we say that a set of vertices $(v_0, v_1, ...v_n)$ in $\mathbb{R}^n$ are in **general position** if no hyperplane[2] contains all n points. A simplex's **faces** can be thought of as "sub-simplices": for a tetrahedron, its faces are its vertices, its edges, its triangular faces, and the tetrahedron itself.

A **simplicial complex** is an object built out of simplices: for instance, we might take two triangles and glue them together along an edge. This contains information for each dimension: a simplicial complex is a set containing all of its constituent simplices, including all the vertices, edges, and other lower-dimensional simplices. Most reasonable topological spaces (e.g., anything arising in a practical context) are homeomorphic to some simplicial complex.

2.2.2 Details

An **n-simplex** is defined as the convex hull of n+1 points in general position. A **face** of a simplex $\sigma_1 = conv(v_0, v_1, ..., v_n)$ is the convex hull of a nonempty subset of the vertices v_i, e.g. $\sigma_2 = conv(v_1, ..., v_{n-1})$. A **simplicial complex** is a finite set of simplices K such that two conditions hold:

(1) If σ is in K, then every face of σ is in K.

(2) For $s_1, s_2 \in K$, if $s_1 \cap s_2$ is nonempty, it is a face of both s_1 and s_2.

2.3 Simplicial homology ($\mathbb{Z}/2\mathbb{Z}$ coefficients)

2.3.1 Intuition

Given a simplicial complex, we can use efficient algorithms to determine some topological information about each dimension: how many holes are there? For instance, a triangle with no interior has a single "one-dimensional hole", and an empty tetrahedron has a "two-dimensional hole" in its interior.

A zero-dimensional hole is a *gap* between components, so zero-dimensional homology describes the number of **connected components** in a space. Homology groups capture all this information: for a simplicial complex X, the nth homology group $H_n(X)$ is isomorphic to $\mathbb{Z}^k$, where k is the number of n-dimensional holes in X.

To determine this information, we look at the k-dimensional structure of X, and the way it attaches to the (k-1)-dimensional structure. This can be described by the **boundary map** ∂_k, which sends each simplex to its boundary. By analyzing the kernel of this map, we determine *absences* within the space, detecting the holes that correspond to abnormalities in the kernel of ∂_k.

It is worth noting that we can take coefficients in any ring, and the ordinary choice is $\mathbb{Z}$. Using $\mathbb{Z}$ would give us more information: in addition to keeping track of holes, it would also

[2] A hyperplane is an (n-1)-dimensional affine subspace of $\mathbb{R}^n$.

keep track of **torsion**, which gives us information about orientability. This would allow us to distinguish between, e.g., the circle and the Klein bottle. However, this mild advantage comes at a high computational cost, and so computational topology generally restricts to $\mathbb{Z}/2\mathbb{Z}$ coefficients.

2.3.2 Details

Fix a simplicial complex K and a dimension n. An **n-chain** is a sum of simplices s_i with coefficients in $\mathbb{Z}/2\mathbb{Z}$, written c = $\sum a_i s_i$. Under formal addition, n-chains form an abelian group known as the **nth chain group** $C_n(X)$.

For each n, we define a **boundary map**, ∂_n. The boundary of an n-simplex is defined to be the sum of all possible (n-1)-dimensional faces.

An n-cycle is an n-chain p such that $\partial_n(p) = 0$. An n-chain c is an **n-boundary** if there exists d such that $\partial_{n+1}(d) = c$. For a fixed n, n-cycles form the **cycle group** $Z_n(K)$. Boundaries form the **boundary group** $B_p(K)$.

We define the **nth homology group** to be: [2]

$$H_n(K) = Z_n(K)/B_n(K).$$

2.4 Persistent homology

2.4.1 Intuition

Suppose we have a nested collection of simplicial complexes: $\emptyset = K_0 \subseteq K_1 \subseteq ... \subseteq K_n = K$. We can view this as a simplicial complex with a "complexity dial": as we turn the dial to increase the complexity, we add more and more simplices. If the dial is turned all the way down, our complex is empty. If the dial is turned all the way up, we have the full complex K. This is an important example of a **filtered simplicial complex**.

Persistent homology is a tool that extends simplicial homology to these filtered spaces. In particular, it tells us when the *topological structure* of the space changes, as we turn our complexity dial. Perhaps a hole appears at complex K_3, and it gets filled in by complex K_{10}. Or a second component appears at K_4, but is absorbed by the main component by K_5. We are interested in the second case: zero-dimensional persistent homology, which tracks the change in connected components across a filtration.

Persistent homology is essentially a bookkeeping tool that records this type of information. When a new component appears, it is referred to as a **birth**. When a component disappears, it is referred to as a **death**. The **persistence** of a component is the length of time it exists: if a component is born at K_4 and dies at K_{12}, then its persistence is $12-4 = 8$.

2.4.2 Details

A filtered simplicial complex can be derived from a simplicial complex K and a map $f : K \to \mathbb{R}$, if f is nondecreasing along increasing chains of faces. See Edelsbrunner [2] for more details of this construction.

We begin with a filtered simplicial complex:

$$\emptyset = K_0 \subseteq K_1 \subseteq ... \subseteq K_n = K.$$

We take simplicial homology, and fix a dimension p. Each topological inclusion induces a homomorphism on homology.

$$0 = H_p(K_0) \to H_p(K_1) \to ... \to H_p(K_n) = H_p(K).$$

In general, for each i ≤ j, we have $f_p^{i,j} : H_p(K_i) \to H_p(K_j)$.

Definition 2.1. [2] The pth persistent homology groups are the images of the above maps.

$$H_p^{i,j} = im f_p^{i,j}, 0 \leq i \leq j \leq n.$$

2.5 Defining peaks through zero-dimensional graph persistence

2.5.1 Intuition

We can finally return to the problem of peak detection. Suppose we have the graph of some frequency spectrum. We slide a horizontal line down the graph, and we look only at the segments of the graph that are above the line. This constitutes a filtered simplicial complex, and we can look at its persistent homology: how do the components change, as we move the line? This is easiest to understand through an example, which is presented in figures 4–8.

This is an algorithmic procedure. Given a frequency spectrum, it outputs the location and persistence of each local maximum. Persistence can be interpreted as a measure of significance: low-persistence components correspond to noisy artifacts of the data, while high-persistence components correspond to harmonic components of the note. This output is called the **persistence diagram**: it is a list of points of the form (frequency, persistence). Each point represents a **persistent component**.

We define a **peak** to be a persistent component in a diagram. This ensures that each peak has one representative. Every local maximum is encoded within the persistence diagram, so this will also include many noisy artifacts within the data.

For each algorithm, we will reduce the size of each persistence diagram by setting a significance cutoff: e.g., every component below 5% of the maximum persistence is discarded. A **significant peak** is a persistent component within this new diagram.

2.5.2 Details

We regard the graph of a function as a formal graph: the image of the function is its vertex set, and then we add an edge between every two points adjacent in the domain, i.e., (n,f(n)),(n+1,f(n+1)). This is visually indistinguishable from the ordinary graph of the function, with a line segment between f(n) and f(n+1) for all valid n.

The standard filtration involves the sublevel sets of our function, where we consider preimages of the form $f^{-1}(-\infty, x)$. We modify this to use superlevel sets, i.e., $f^{-1}(x, \infty)$. This is equivalent, but computationally beneficial: we start scanning at the top, which is the most important part of the graph. Once we have found the persistent components to a sufficiently high degree of precision, we can halt our computation. If we scan bottom-to-top, we are beginning with the noisiest part of the graph, and we cannot halt the procedure until it finishes.

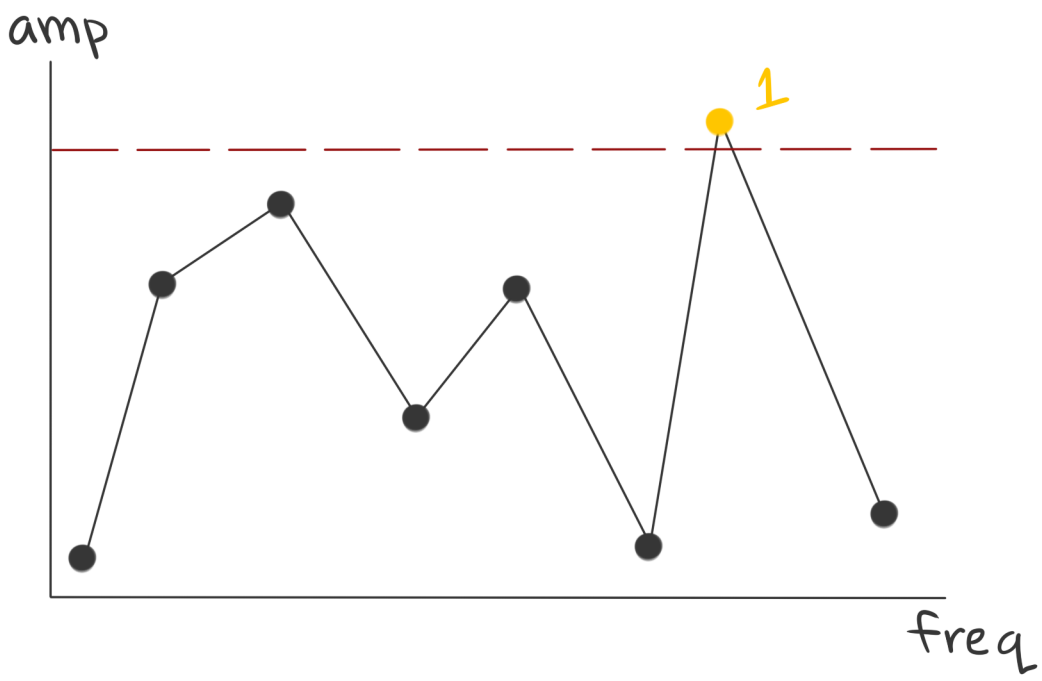


Figure 4: We begin sliding the dashed line down our graph. We consider only the highlighted subgraph above the line. At present, we only have a single component, labeled **1**.

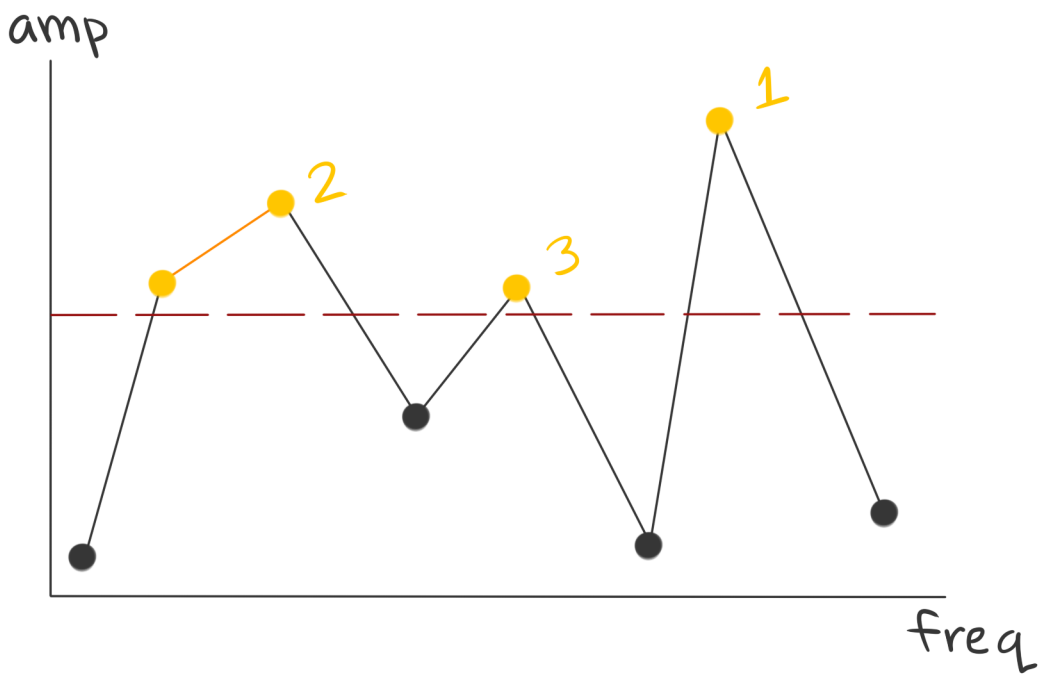


Figure 5: As we slide the bar down more, we get several component births: first **2**, then **3**. Note that component **2** has picked up a second, unlabeled point: since this point did not add or delete a component, it merely attaches to component **2**.

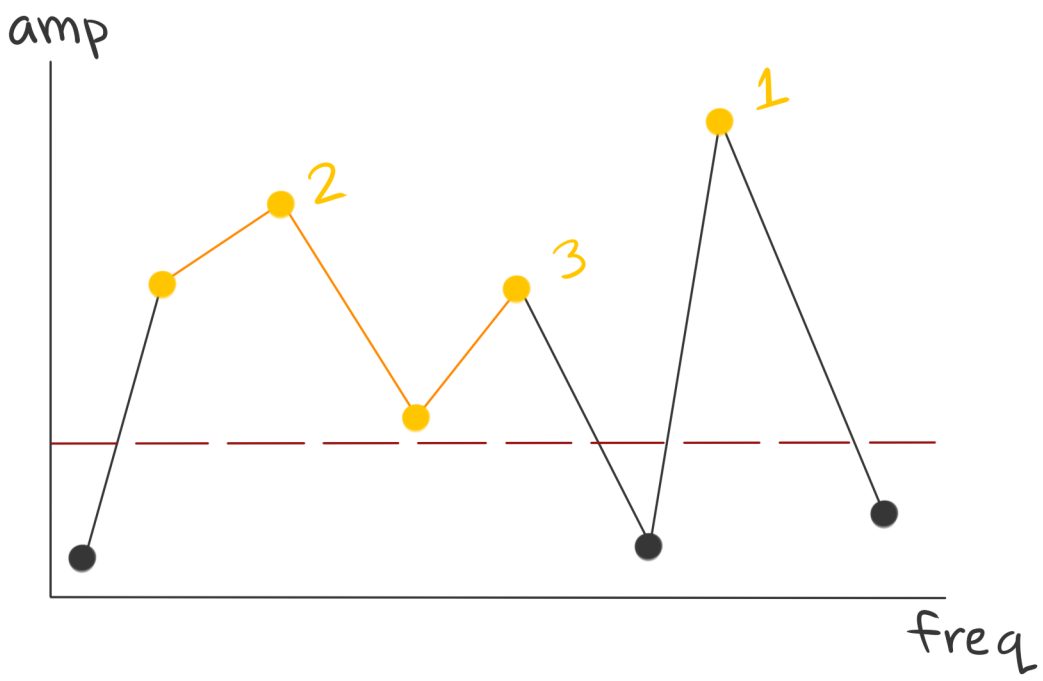


Figure 6: When two components collide, the younger component dies. The death of **3** is not labelled explicitly on this graph, but we still keep track of it. Only two components remain.

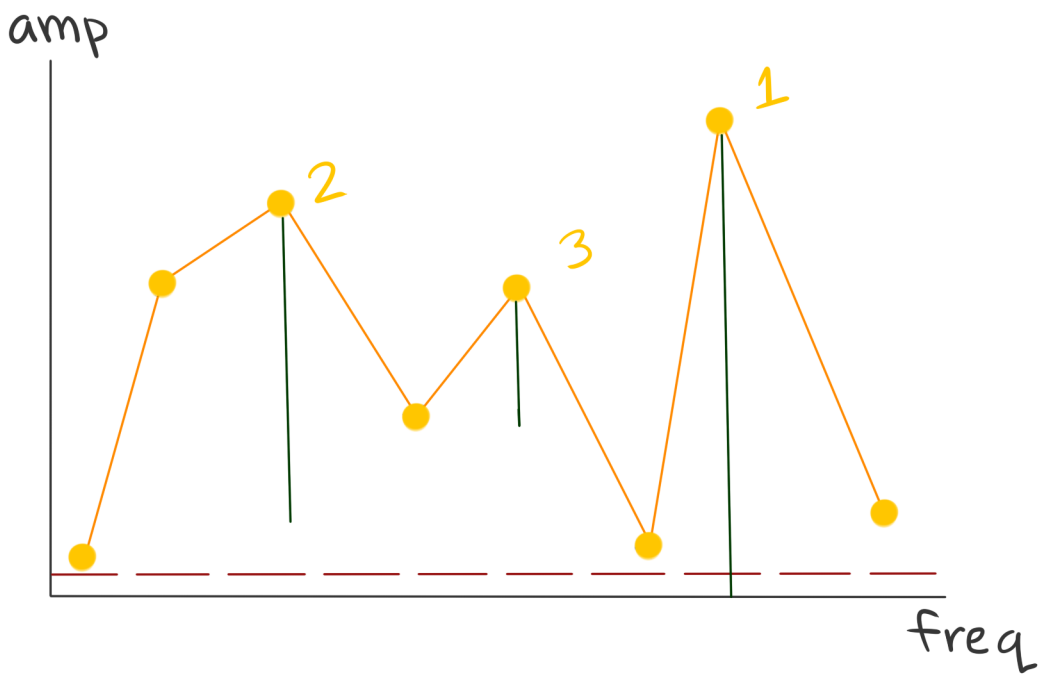


Figure 7: Once our filtration reaches the end of the graph, the first component has subsumed all others. Persistence is marked by green vertical bars beneath each component: they begin at the point of birth, and end at the point of death. For component **1**, we set its persistence equal to its amplitude, since it never dies.

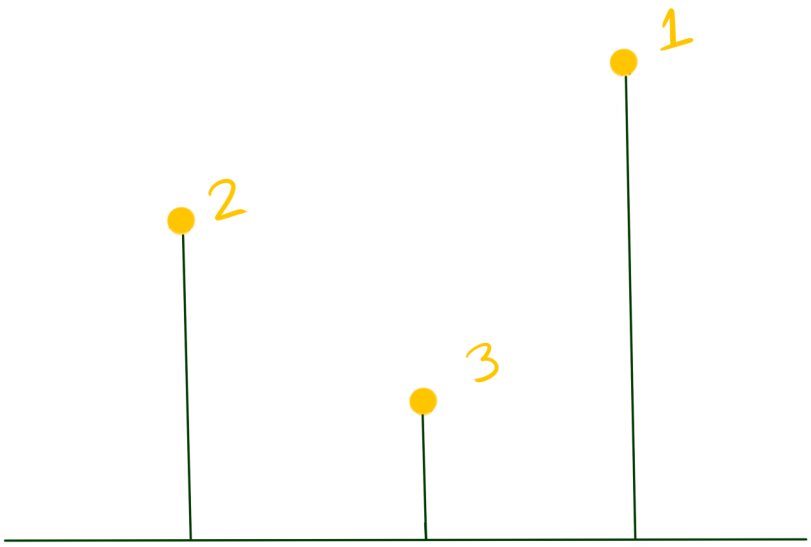


Figure 8: If we look at only the persistent components of the graph, we delete all the extraneous information. This object is one representation of the **persistence diagram**, which encodes all persistence information about the space. Note that every element corresponds to a peak in the initial graph.

The lowest valid frequency in our dataset is 26.75 Hz or MIDI 21: anything below this will convert to a MIDI note outside of our dataset. To avoid some pathological cases, we consider only the FFT from 26.75 Hz onwards: this ensures that every persistent component corresponds to a possible frequency in our dataset. We do not set an upper limit on frequency: the highest frequency in the dataset is 4186 Hz or MIDI 108, but we may still be interested in harmonics of greater frequency. For instance, if we see peaks at 3000 Hz and 6000 Hz, knowledge of the peak at 6000 makes the guess of 3000 more credible.

3 Frequency-domain classifiers

From this point forward, we make use of a signal's Fourier transform (abbreviated FFT)[3] and the persistence diagram of the FFT. The FFT captures information about the component frequencies of the sound. The persistence diagram captures information about spikes in the FFT, and, sufficiently filtered, gives us a very good approximation of the locations of harmonics.

For each of these algorithms, the FFT of our signal is a real array **A**. The persistence diagram of the FFT, notated **dgm**, is a set of pairs (frequency, persistence) corresponding to the 50 highest-persistence peaks in A.

Three graphs were generated for each algorithm: they are available in A.

3.1 Most significant frequency (NaiveFFT)

Our first algorithm is very simple. We take the FFT and predict the frequency of the largest element.

Accuracy: 64.6%.

3.2 Leftmost harmonic (PersistFFT1)

First, set a threshold $\lambda \in (0, 1)$. Let N be the maximum persistence of any component in dgm. We remove any components with persistence lower than $\lambda * N$. This discards insignificant components: an appropriate value for λ can be set by experimentation. This filtration process is used for several other algorithms in this section. For each algorithm, we tested all values in the sequence (.01, .02, ... , .40), and determined the best value of λ. If over 50 components remain after filtration, we keep the most significant 50: this sets a generous upper-bound on memory consumption.

For this algorithm, we found the best value was λ=**0.30**.

From the remaining significant components, find the one with lowest frequency. This frequency is our guess.

Accuracy: 81.6%.

3.3 Best harmonic (PersistFFT2)

As in the previous section, set a threshold $\lambda \in (0, 1)$ and filter out low-persistence components. For this algorithm, we found the best value was λ=**0.02**.

Take the frequencies of all remaining components, convert them to MIDI notes, and remove duplicates. This is our list of candidates for the final prediction.

We assign a certain amount of error to each candidate: a higher error indicates that the candidate predicts nonexistent peaks in the persistence diagram. A lower error indicates that it is a good match. Let $\nu(p)$ indicate the frequency of a point p in our persistence diagram, and G be a guess of frequency.

[3]This is slightly ambiguous terminology: FFT may refer to the algorithm for computing the spectrum, or the spectrum itself. This usage is common in the literature, since the meaning is generally clear in context.

$$Err(G) := \frac{\sum_{p \in dgm} pers(p) \cdot |G \cdot Round(\frac{\nu(p)}{G}) - \nu(p)|}{G \cdot \sum_{p \in dgm} pers(p)}.$$

$|G \cdot Round(\frac{\nu(p)}{G}) - \nu(p)|$ emerges as the frequency error in our prediction: if the fundamental frequency is really G, then we expect each harmonic to show up as an integer multiple of G. This term measures the distance of $\nu(p)$ from the nearest integer multiple of G. We take the absolute value and scale it by the persistence of p, since we should prioritize errors that occur in high-persistence components.

The terms in the denominator are corrective terms that normalize the error:

$$G \cdot \sum_{p \in dgm} pers(p).$$

The sum of total persistence varies greatly across our dataset, and the summation term ensures that different errors are in comparable units.

The G term penalizes lower-frequency guesses, since they have more explanatory power. If we did not penalize low-frequency guesses, note that the following holds for all naturals n:

$$Err(G/n) \leq Err(G).$$

This occurs because any integer multiple of G is also an integer multiple of G/n, and it results in many misclassifications.

Accuracy: 88.6%.

3.4 Spectral Harmonic Correlation (SHC)

Zahorian and Hu define this method in [9]. The idea is that, since we expect harmonics to be regularly spaced by the fundamental tone, we may use this to check candidate guesses. Given a guess, we calculate an associated score, measuring the extent to which the FFT has the predicted peaks. A higher score is better.

$$SHC(t, f) = \sum_{f'=-WL/2}^{WL/2} \prod_{r=1}^{N_H+1} S(t, rf + f').$$

"where S(t, f) is the magnitude spectrum for frame t at frequency f, WL is the spectral window length in frequency, and N_H is the number of harmonics." [9] I treat the entire note as a single frame, so "frame t" is just the audio file in question.

I believe that there is an error in this definition, since they define a number of harmonics to check (N_H), then check integral multiples of the frequency (f*2, f*3, ..., f*(N_H+2)). In addition to checking one harmonic too many, this incorporates no information about the fundamental tone of the guess. This seems questionable from a theoretical standpoint and produces worse results experimentally.

I have implemented a slightly different version, which produces more accurate predictions on my dataset. We take the product from r=0 to N_H: this includes the candidate fundamental as well as the next N_H harmonics.

$$SHC(t,f) = \sum_{f'=-WL/2}^{WL/2} \prod_{r=0}^{N_H} S(t, rf + f').$$

A problem can occur with this formula. It can attempt to access frequency data that we don't have: at a sampling rate of 16000 Hz, the FFT contains no data about frequencies greater than 8000 Hz. Zahorian and Hu did not apparently encounter this problem, since their paper is primarily concerned with a smaller range of frequencies.

We experimentally tested several ways to impute these out-of-range values: zero, one, median value, or mean value. Using the mean value was the most effective procedure. So, prior to computing the SHC function, we determine the largest frequency that it attempts to access. If that frequency is greater than 8000 Hz, we extend the FFT by appending its mean value repeatedly until we reach a sufficient length. This process is known as padding.

The original algorithm uses a default number of 3 harmonics, but this is insufficient for the variety of our dataset. We select the number of harmonics on a case-by-case basis: filter the diagram as in the previous two algorithms. Let **L** be the lowest frequency remaining in the diagram, and let **H** be the largest frequency remaining. Then set $H_n = Round(H/L)$: this gives the number of harmonics to look for.

Accuracy: 81.2%.

3.5 Persistent Spectral Harmonic Correlation (PSHC)

Since the FFT is memory-intensive to store,[4] it would be ideal to have a predictive workflow that iterates through data files, computes the FFT, computes the persistence diagram, and then deletes the stored FFT. We generalized the SHC algorithm to work on persistence diagrams, rather than the entire FFT.

Very few of the above details are changed. The PSHC algorithm essentially generates a very low-resolution approximation of the FFT by the points in the persistence diagram. For instance, instead of storing all the variance in the neighborhood of frequencies that map to A4 (427.5 to 452.8), we simply store the amplitude of the largest persistent component in that region. We then run an equivalent algorithm on this blocky FFT approximation.

For this algorithm's persistence threshold, we found the best value was λ=**0.06**.

Although this loses a large amount of data, the results seem to be at least as good as SHC. This is convenient, since PSHC requires much less computation.

Accuracy: 84.8%.

[4]Generally it is as large as the signal itself. For real input, we can compress it by half due to symmetry.

4 Time-domain classifiers

We used two algorithms that did not require the use of the frequency spectrum. These were both substantially less accurate than the frequency-based classifiers, but they do not require us to store the FFT or the persistence diagram.

4.1 Zero Crossings

This is a primitive way to approximate the fundamental frequency of a signal. For simple signals, sign changes occur twice per cycle: once when the signal is decreasing, and once when the signal is increasing. We count the number of decreasing (positive-to-negative) sign changes, and use that as an approximation of the frequency. This is not in general a very effective method, but it can be computed quickly. This method has been adapted from [3].

Accuracy: 4.2%.

4.1.1 Algorithm

Suppose we have a signal S=$(x_0, x_1, ..., x_n)$. Our goal is to count the number of positive points x_i such that x_{i+1} is nonpositive. Define a function to extract binary sign information:

$$P(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0. \end{cases}$$

Now, define two sequences L=$(P(x_0), P(x_1), ..., P(x_{n-1}))$ and R=$(P(x_1), P(x_2), ..., P(x_n))$: these are left and right shifts of the array signs. We define one more function:

$$Q(x, y) = \begin{cases} 1 & \text{if } x = 1 \text{ and } y = 0 \\ 0 & \text{else.} \end{cases}$$

Our final answer is then:

$$\sum_{k=0}^{n-1} Q(L_k, R_k).$$

This may seem unnecessarily convoluted, but an approach like this is desirable for efficiency. A motif in scientific computing is the usage of vectorized operations: whenever possible, we work with sequences or vectors as a whole, rather than iterating over the individual elements. Every process listed is implemented in a vectorized way, so it requires less time than the obvious iterative approach.

4.2 Autocorrelation

Autocorrelation attempts to determine the fundamental frequency of some signal by comparing two versions of the signal. The first version is fixed in time, the second is played at a delay. If we set the delay to be equal to the fundamental period, then the signals should be very similar.

To measure the similarity of signals, we define the autocorrelation of a signal S to be the convolution of S with S_{rev}, the signal played backwards in time. For a signal y and a lag l: [1]

$$AC_y(l) = \sum_{n \in Z} y(n)\, y(n-l).$$

Although this is listed as a time-domain method, efficient computation of the convolution uses the frequency domain, as convolution in time corresponds to multiplication in frequency.

We window the signal with some arbitrary window that avoids any transient effects (e.g., 1 second to 2 seconds). We find the first (l>0) local maximum of the autocorrelation function, and assume that this corresponds to the fundamental period of the note. We then predict pitch based on the fundamental period.

This method is in need of further optimization. Although it seemed promising, it was not very useful given our low bitrate and wide range of frequencies/instruments.

We might also try using several windows for the same frequency. One possibility is to implement a majority-vote classifier across various windows, which would use all of the information instead of only the selected window.

Autocorrelation is sometimes confused with a separate technique: instead of finding maxima of the autocorrelation function, one finds minima of an error function, such as the L_2 norm. This is computationally slow and offers no advantage for our problem.

Accuracy: 53.6%.

5 Aggregate classification methods

I implemented several classifiers from scikit-learn.[6]

5.1 Logistic regression

Logistic regression is a generalization of linear regression to classification problems. If we would like to use linear regression to predict a finite set of classes rather than a continuous target, we might wish to try predicting the probabilities of various classes. Probability is a continuously varying parameter, so it allows us to apply regression—but we run into problems by predicting probabilities that are less than 0 or greater than 1.

If we want to apply regression to probabilities, while ensuring that the sum of all probabilities is 1 and we have no negative probabilities, we may apply the **log-odds** (or **logit**) transformation. In the case of a binary decision problem, we have the following:

$$logit(p) = log(\frac{p}{1-p}).$$

Predicting the log-odds of each class is equivalent to predicting the log-likelihood of each class, which gives similar results to predicting actual likelihood. Logistic regression is a simple and powerful technique that is commonly used for classification problems. It is difficult to overfit to your data, since there are no free parameters.[5]

The high bias of this approach seemed incompatible with our nonlinear dataset: we produced only a marginal classification accuracy of <20%. Although this could certainly be improved, we shifted our approach to random forests, which are better suited to model complicated and nonlinear data.

5.2 Random forest classifiers

A **classification tree** is a classifier that partitions the decision space along simple boundaries to create many rectangular regions. We fit a simple model to each region, such as a constant prediction. [5] This is similar to a good algorithm for playing the game 20 Questions: with each new dimension (or question), try to split the remaining uncertainty in half. By the end, we know our approximate position in feature space, and we can make an educated guess about the final answer.

A **random forest classifier** creates a large collection of classification trees, fits them to different subsets of the data, and uses the majority vote of the trees to predict the final class. This ameliorates the risk of overfitting, since all of the trees are fit to slightly different data, but it is still important to do a training/test data split. This algorithm is easy to parallelize, which gives it a computational advantage over the similar method of boosting.

For reproducibility, we provide Python 3.6 code to construct our final random forests.

```
#Fit the pitch classifier to these columns of the data:
#['NaiveFFT', 'ZeroX', 'AutoCorr', 'PersistFFT1', 'PersistFFT2', 'PSHC',
  'peak_0_freq', 'peak_1_freq', 'peak_2_freq', 'peak_3_freq',
  'peak_4_freq']
```

```
rf_pitch=ExtraTreesClassifier(n_estimators=200, max_depth=17,
                              max_features=7, min_samples_split=5)
#Fit the instrument family classifier to the results of all 6 algorithms,
#as well as the entire persistence diagram.
#(peak_n_freq, peak_n_amp for 0<=n<=50)
rf_fam=ExtraTreesClassifier(n_estimators=200, max_depth=120,
                            max_features=88)
```

6 Conclusion

6.1 Final results

After tuning the hyperparameters with fivefold cross-validation, our final random forests attained 91% accuracy on the test dataset for instrument detection, and 90% accuracy for pitch detection. We are satisfied with these results: accuracy is largely bottlenecked by difficult examples within the dataset. Figures 9, 10, and 11 are three randomly selected examples where our random forest fails to produce the correct result.

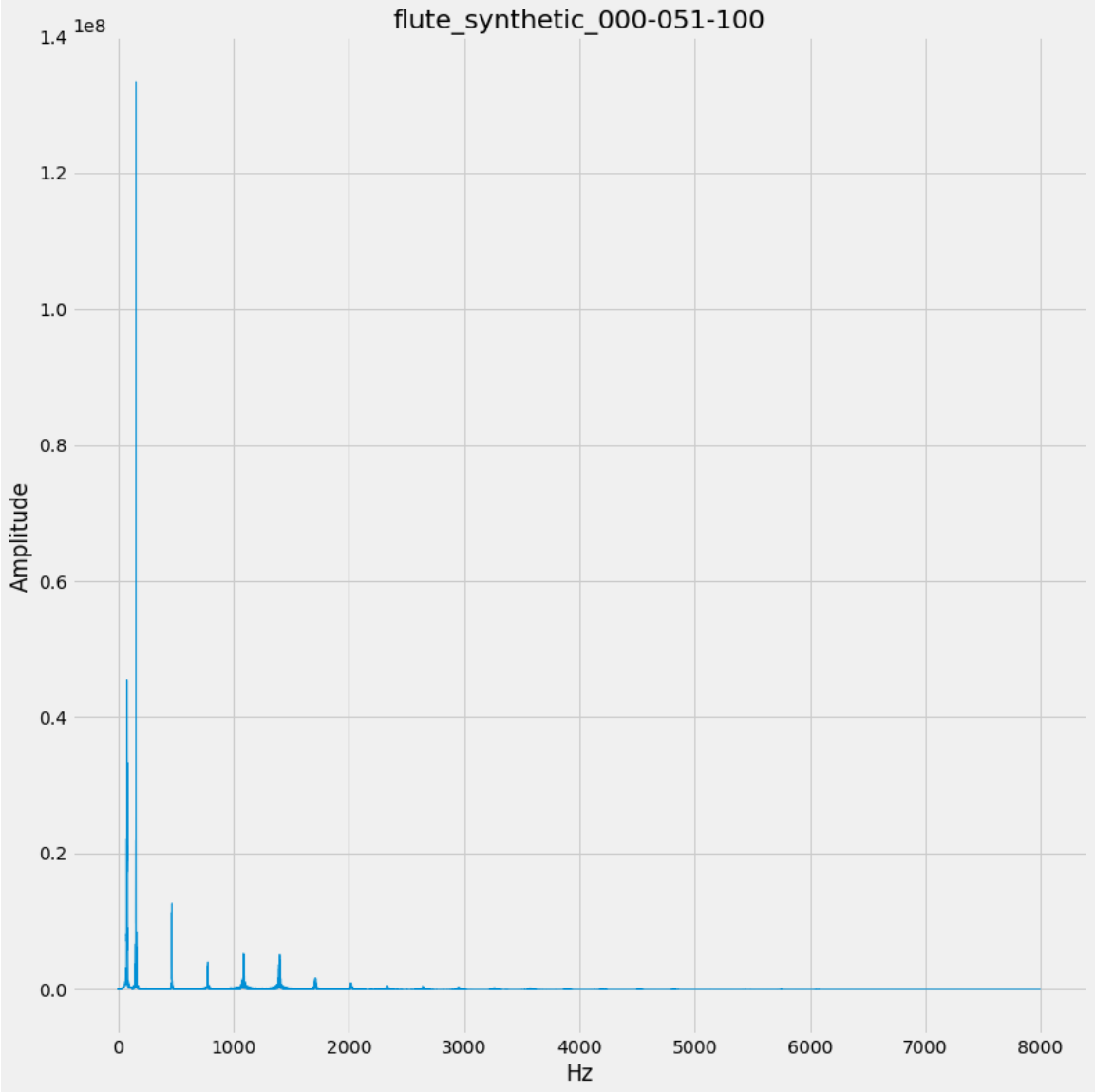


Figure 9: The spectrum of a misclassified flute: the correct frequency is the largest spike at 155.5 Hz.

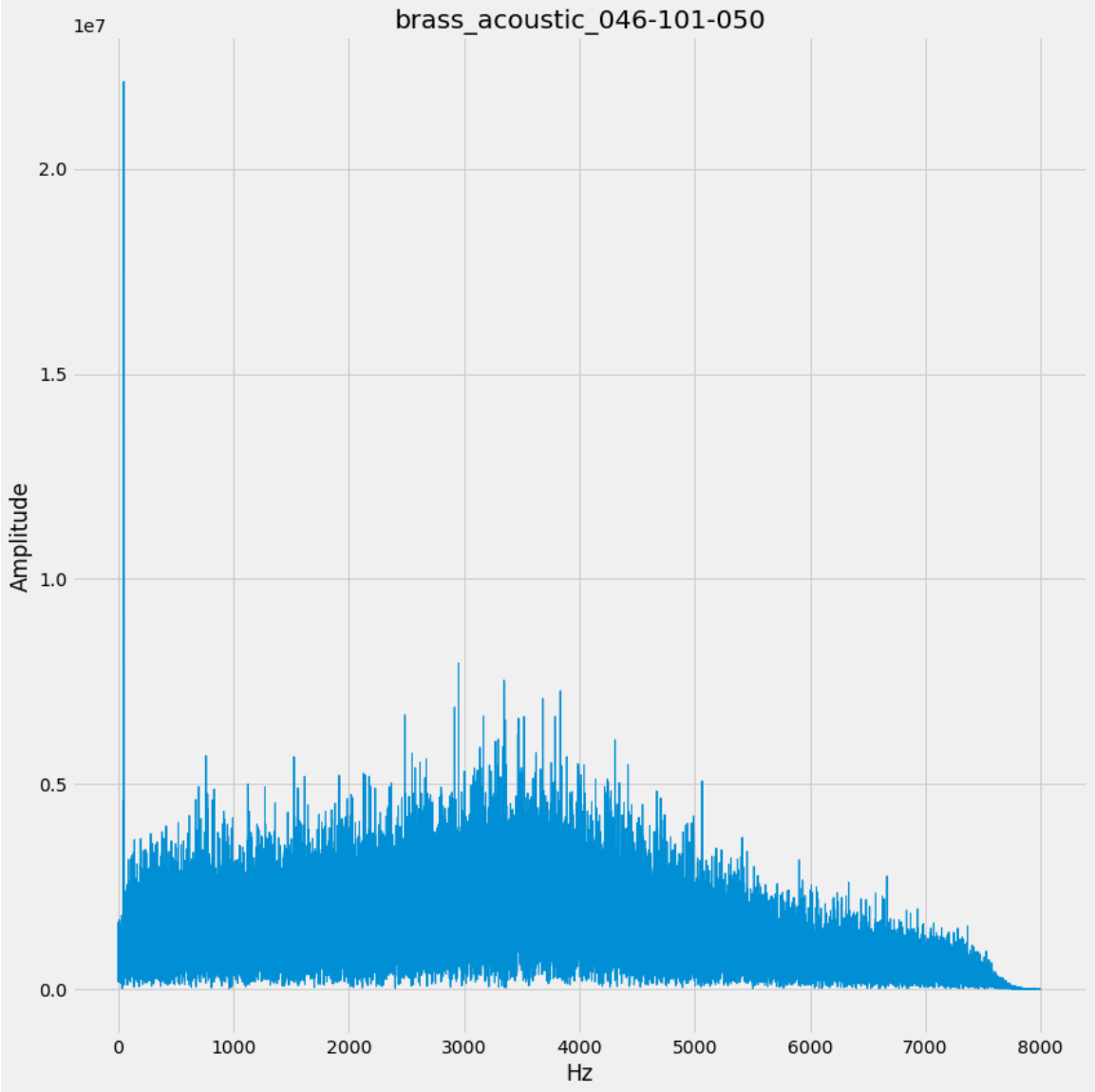


Figure 10: A misclassified brass instrument: the correct frequency is 2793 Hz.

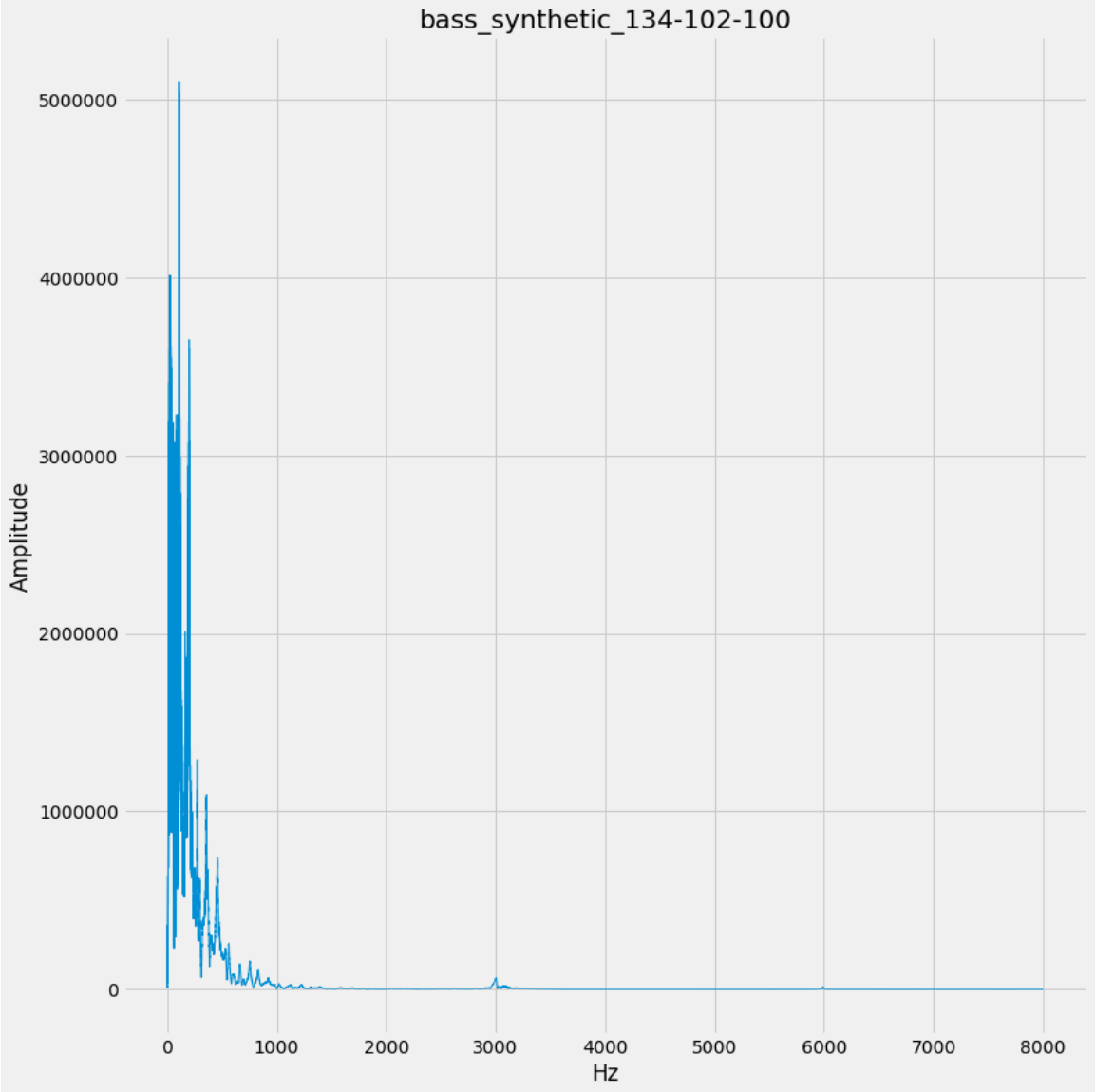


Figure 11: A misclassified brass instrument: the correct frequency is 2959.9 Hz.

6.2 Applications

6.2.1 Automatic music transcription

It would take much more work, but these ideas could potentially be extended to create software that automatically transcribes music. This would save time for many musicians and students: the process of music transcription is very arduous, and it is generally difficult to learn a song "by ear." With good automatic transcription, there would be no distinction between a musical recording and its score.

6.2.2 Speech recognition

Although speech recognition is a much more developed field, our methods may still be useful. We are not aware of any work in speech recognition which uses our topological characterization of peaks.

6.3 Future work

6.3.1 Chordal analysis

Polyphonic analysis is likely much more difficult than monophonic analysis, but it is an important problem: if you hear a chord, how can you reconstruct the notes?

6.3.2 Real-time work

We have not attempted to use these algorithms to analyze a continuous stream of real-time data. It would take some additional work, but some of our algorithms could be extended to cover the real-time case. In particular, computing persistence diagrams is generally cheap: nearly linear[5] in the number of samples.

6.3.3 Bottleneck distance

There is a natural metric defined between persistence diagrams: the **bottleneck distance**. This seems like a promising avenue of research.

6.3.4 Expanding autocorrelation

Although autocorrelation seemed promising at the outset, it was not ultimately useful for this problem. It might be beneficial to return to it using persistent peak-finding methods from section 2.5: we could find several peaks and determine the average spacing between these peaks, which should give us a higher-resolution approximation of the fundamental tone.

[5] "Almost linear" means that we make a linear number of almost-constant lookups to a union-find datastructure, which grows as the inverse Ackermann function.[2] For all practical purposes, this is linear.

7 References

[1] Wikipedia contributors. Autocorrelation, 2018.

[2] H. Edelsbrunner and J. Harer. Computational topology: An introduction, 2010.

[3] endolith. Frequency estimation methods in python, 2015.

[4] Jesse Engel, Cinjon Resnick, Adam Roberts, Sander Dieleman, Douglas Eck, Karen Simonyan, and Mohammad Norouzi. Neural audio synthesis of musical notes with wavenet autoencoders, 2017.

[5] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2008.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[7] Marıa Cristina Pereyra and Lesley A Ward. Harmonic analysis: from fourier to haar. *preprint*, 2012.

[8] Justin Rogers. Issue: Nsynth test dataset contains pitches outside the correct range, 2018.

[9] Stephen A Zahorian and Hongbing Hu. A spectral/temporal method for robust fundamental frequency tracking. *The Journal of the Acoustical Society of America*, 123(6):4559–4571, 2008.

My code: https://github.com/justinusjr/baconian-semiotics

A
Graphs

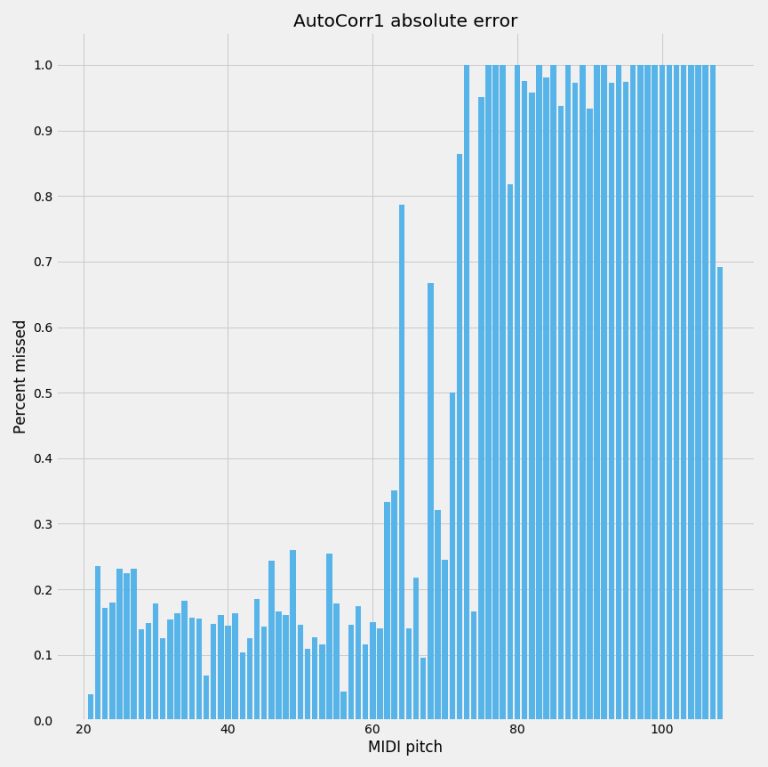
Three graphs are included for each classifier. They measure various forms of error across the entire dataset.

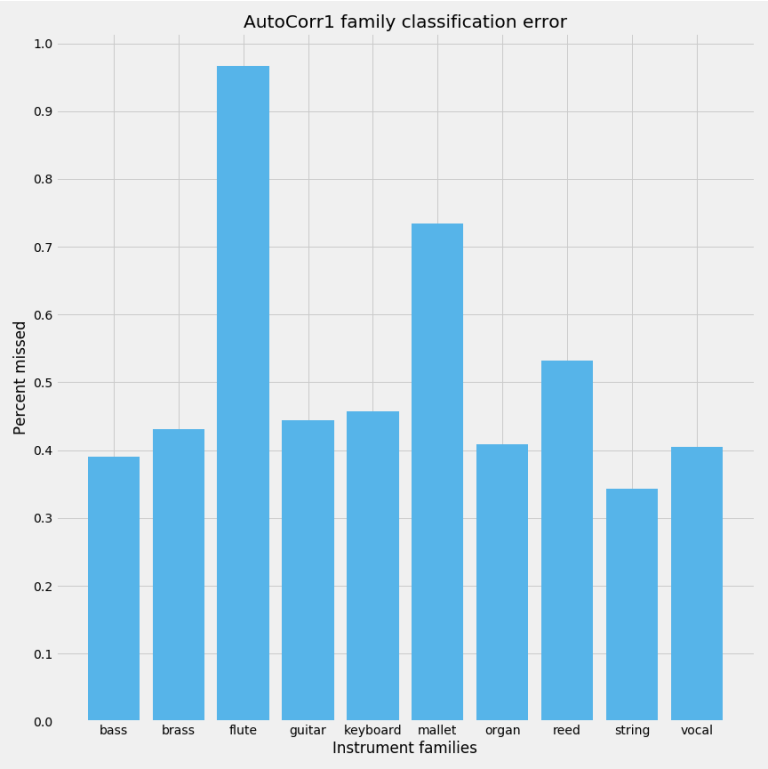
Absolute error measures accuracy on notes of a given pitch: e.g., we can see that autocorrelation misclassified every note of pitch 100.

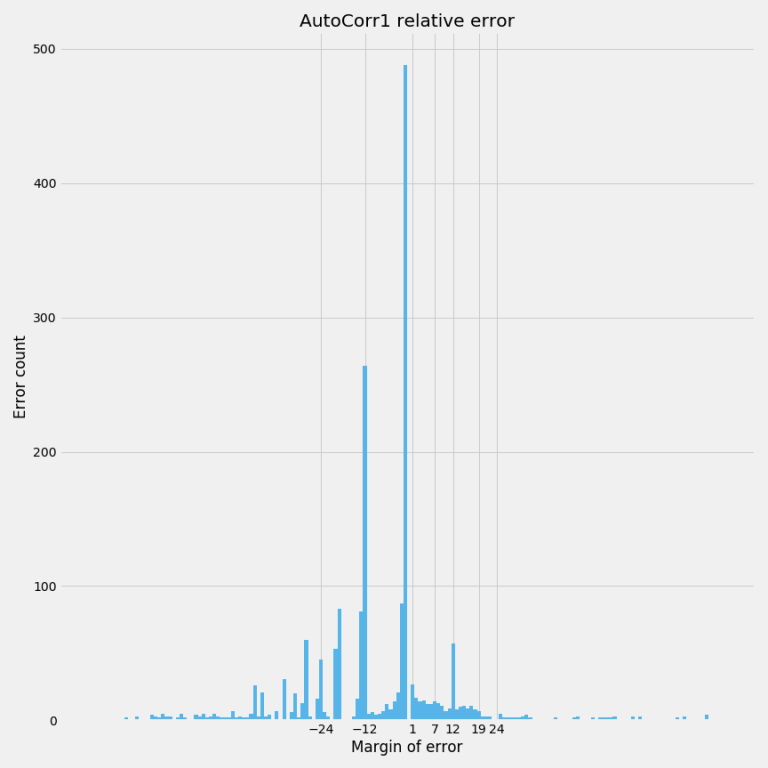
Family classification error is not a measurement of instrument classification. It measures pitch misclassifications within certain families: e.g., autocorrelation misclassifies the pitch for 95% of flutes.

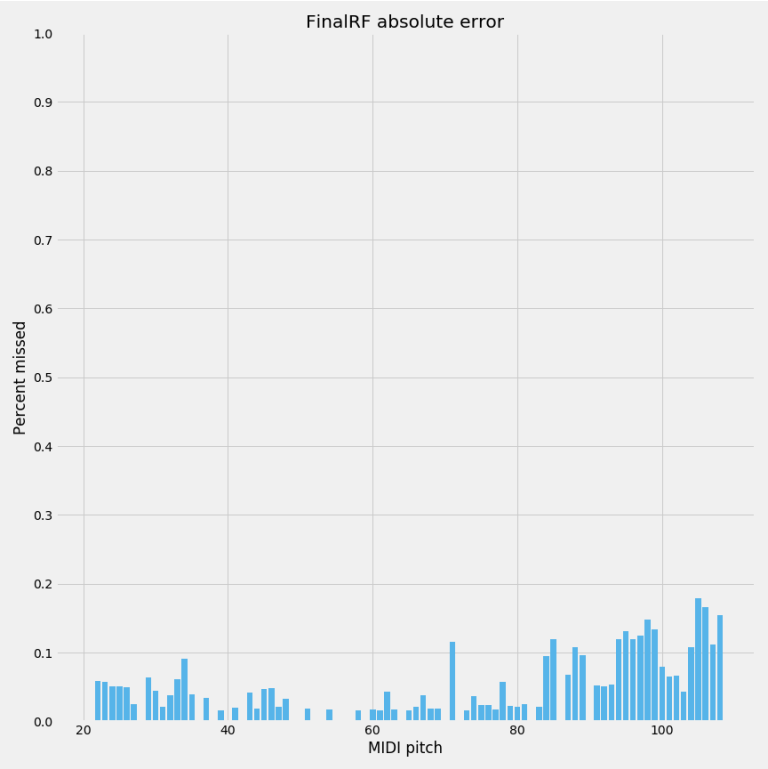
Relative error is a histogram of inaccuracy. We see that autocorrelation is often one note below the correct pitch, and it often guesses 12 notes too low: an octave error.

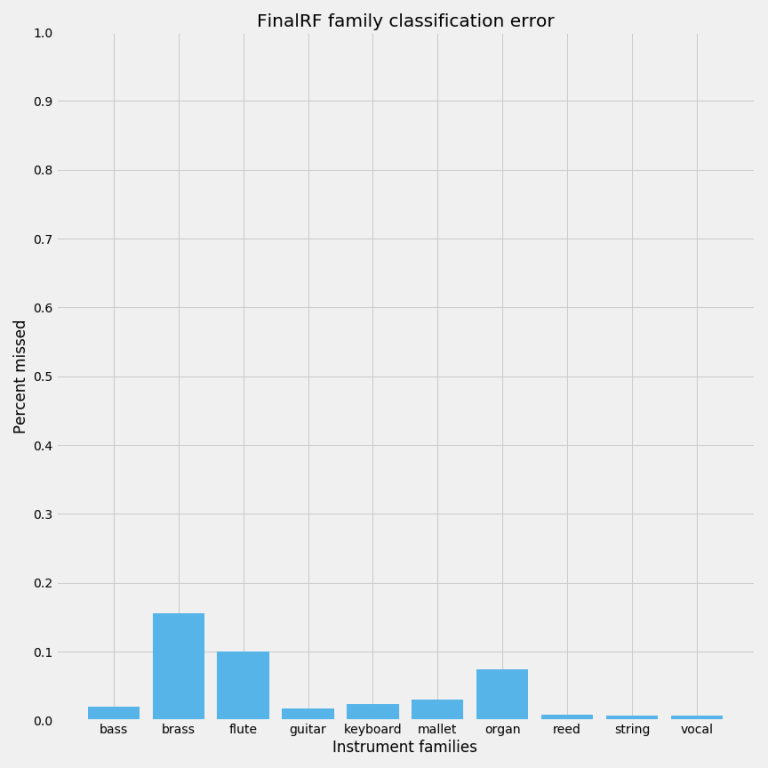
The FinalRF graphs were generated from the random forest. These graphs should be taken with a grain of salt: the forest was fit to part of this dataset, giving it an unrealistic 96% classification accuracy.

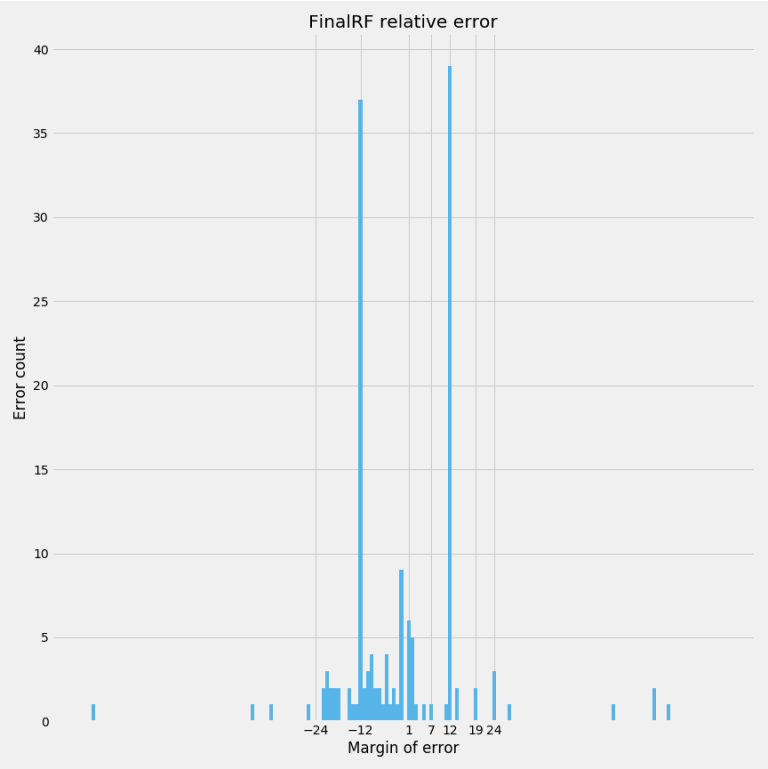


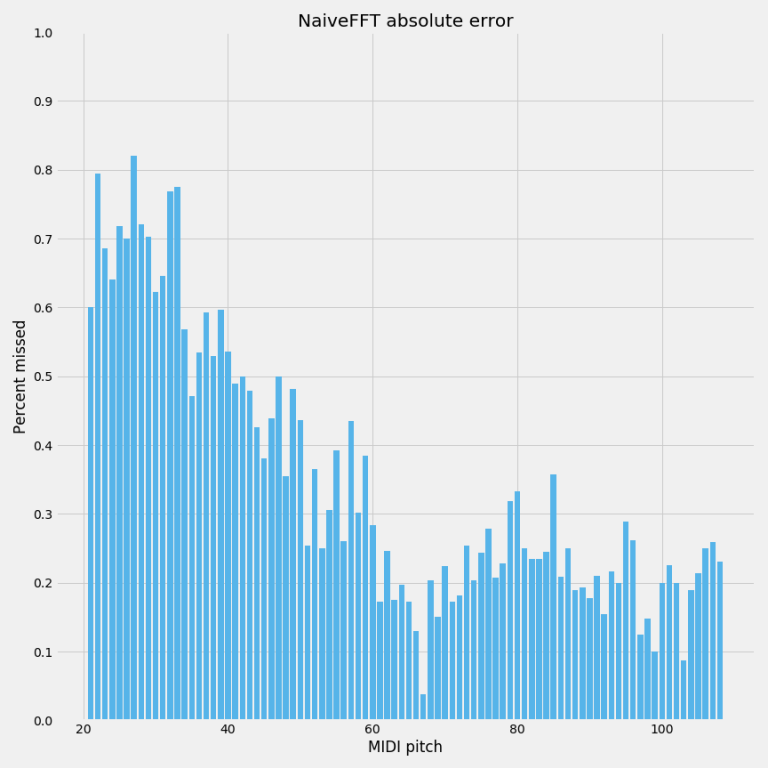

AutoCorr1 family classification error
Percent missed
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
bass
brass
flute
guitar
keyboard
mallet
organ
reed
string
vocal
Instrument families

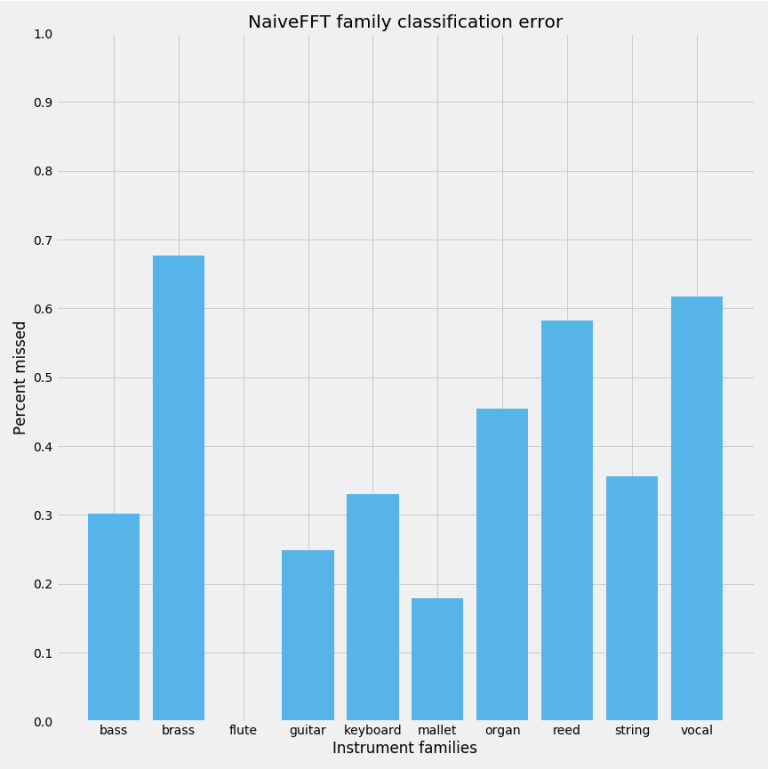

AutoCorr1 relative error
Error count
500
400
300
200
100
0
−24
−12
1
7
12
19
24
Margin of error

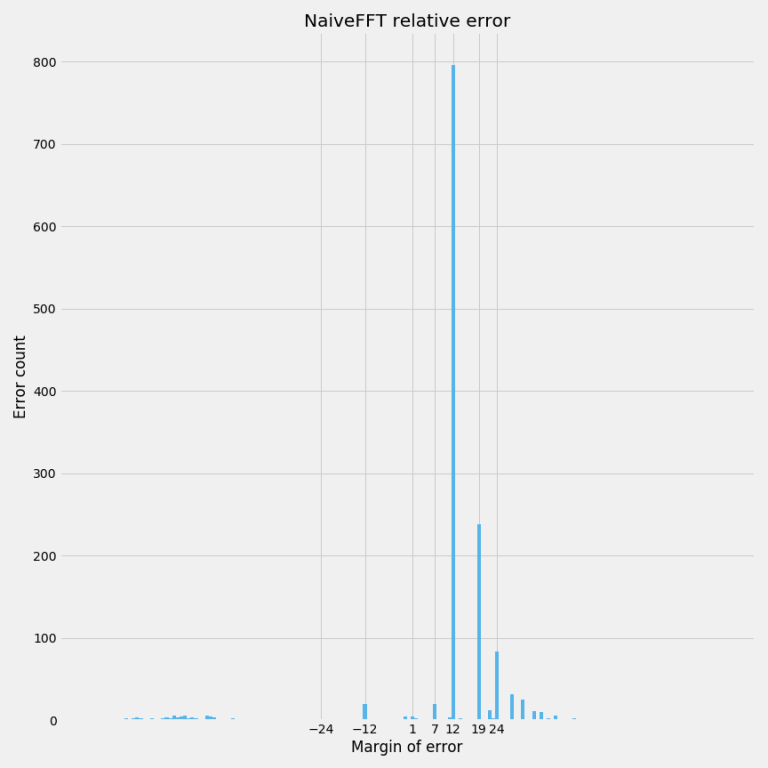

FinalRF absolute error
Percent missed
MIDI pitch
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
20
40
60
80
100

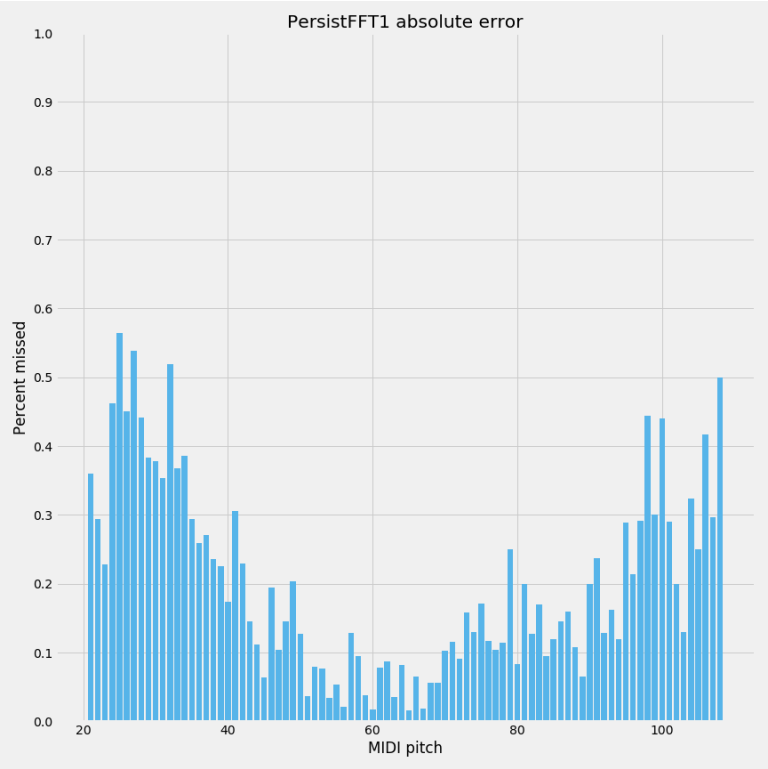

FinalRF family classification error
Percent missed
Instrument families
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
bass
brass
flute
guitar
keyboard
mallet
organ
reed
string
vocal

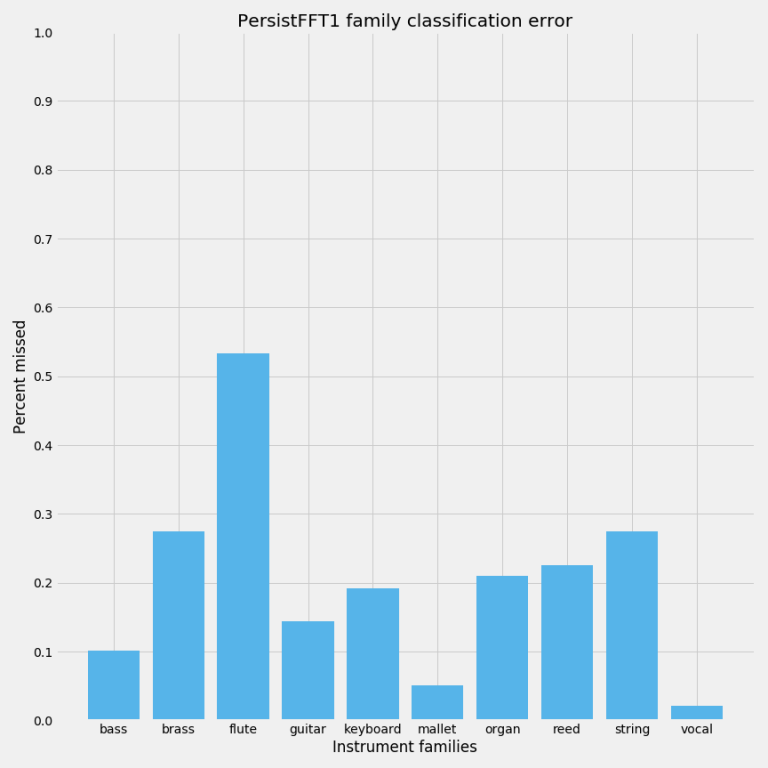

FinalRF relative error
Error count
Margin of error
0
5
10
15
20
25
30
35
40
−24
−12
1
7
12
19
24

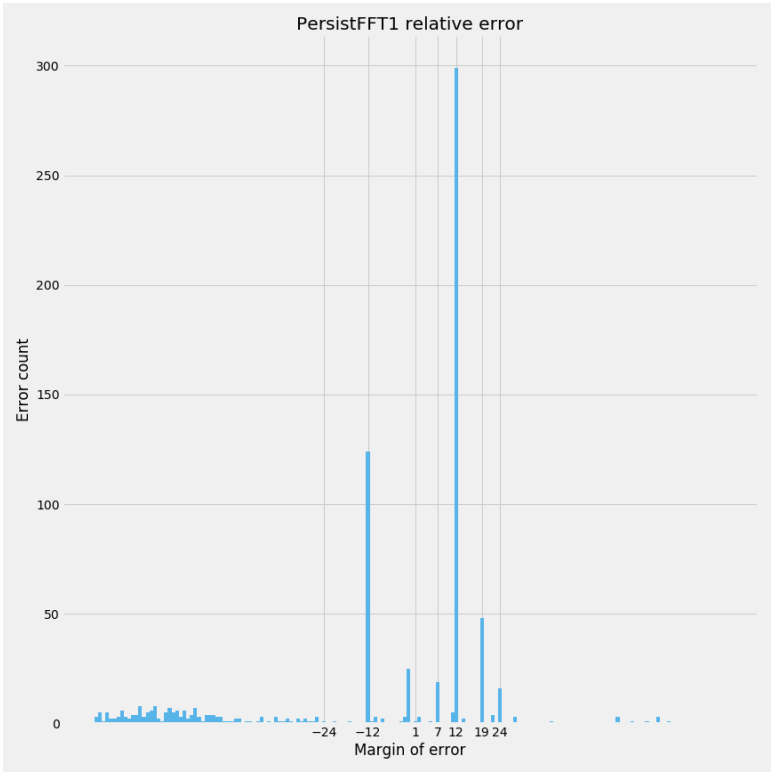

NaiveFFT absolute error
Percent missed
MIDI pitch
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
20
40
60
80
100

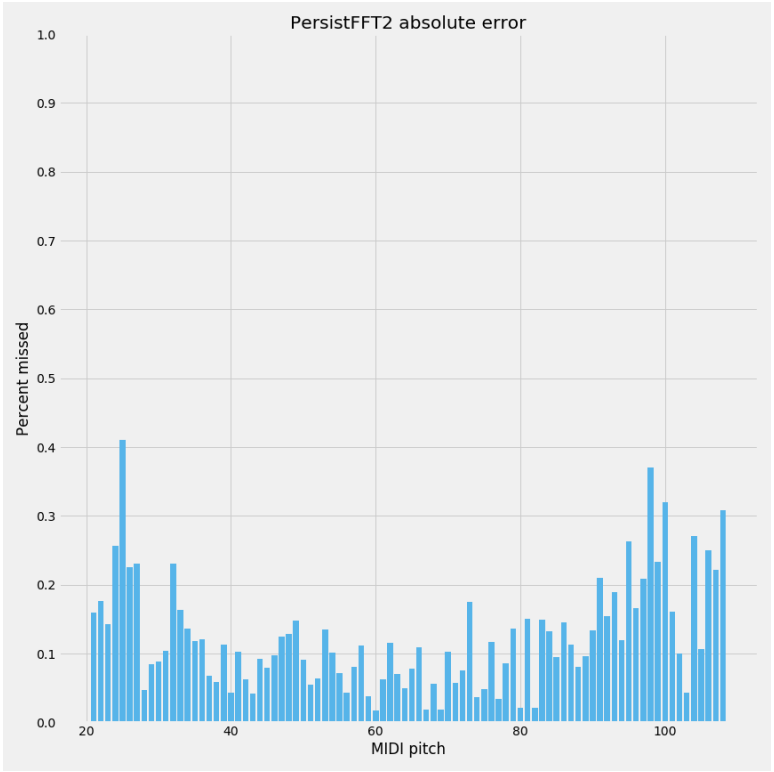

NaiveFFT family classification error
Percent missed
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
bass
brass
flute
guitar
keyboard
mallet
organ
reed
string
vocal
Instrument families

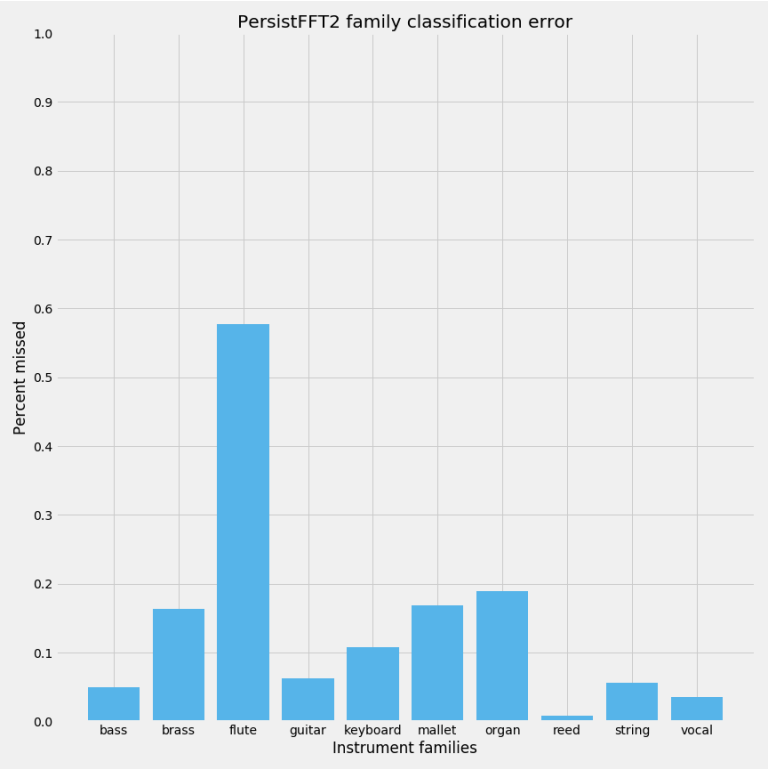

NaiveFFT relative error
Error count
800
700
600
500
400
300
200
100
0
−24
−12
1
7
12
19
24
Margin of error

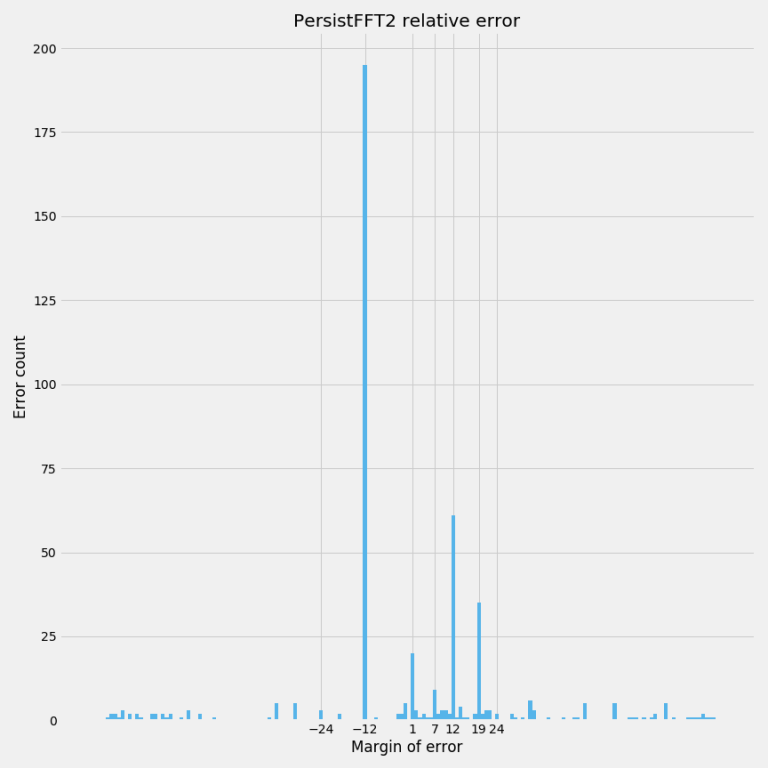

PersistFFT1 absolute error
Percent missed
MIDI pitch
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
20
40
60
80
100

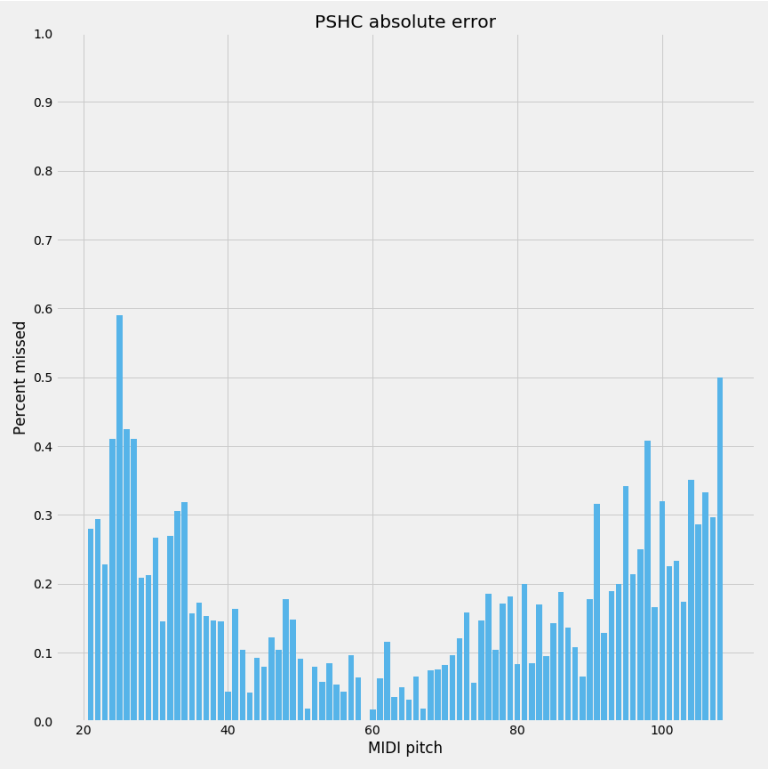

PersistFFT1 family classification error
Percent missed
Instrument families
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
bass
brass
flute
guitar
keyboard
mallet
organ
reed
string
vocal

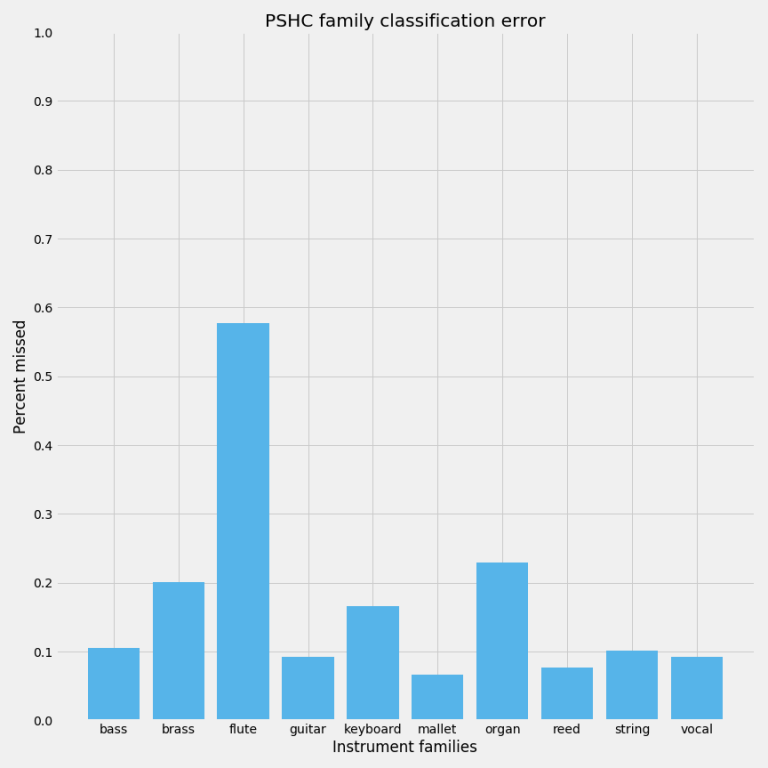

PersistFFT1 relative error
Error count
300
250
200
150
100
50
0
−24
−12
1
7
12
19
24
Margin of error

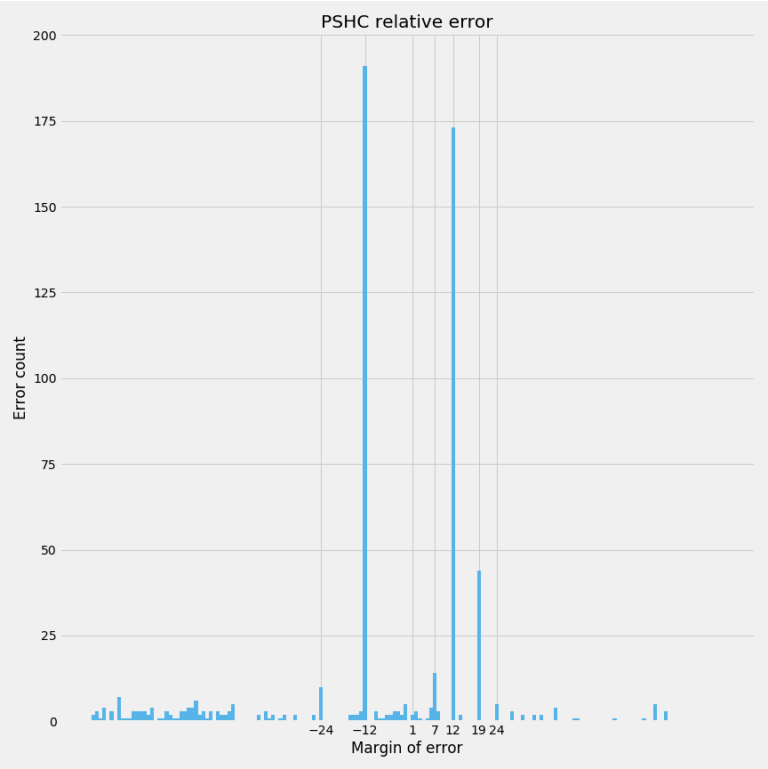

PersistFFT2 absolute error
Percent missed
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
20
40
60
80
100
MIDI pitch

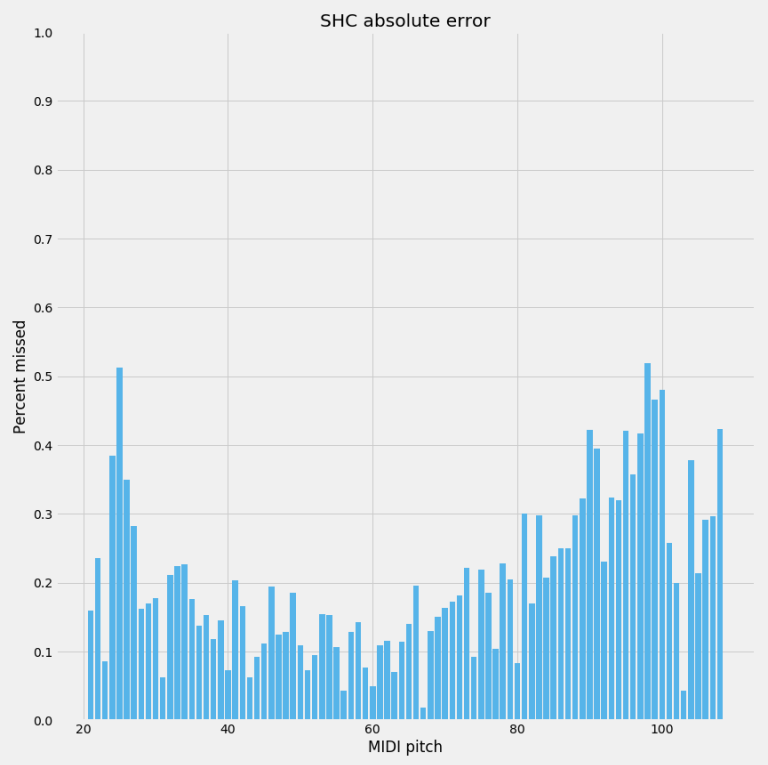

PersistFFT2 family classification error
Percent missed
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
bass
brass
flute
guitar
keyboard
mallet
organ
reed
string
vocal
Instrument families

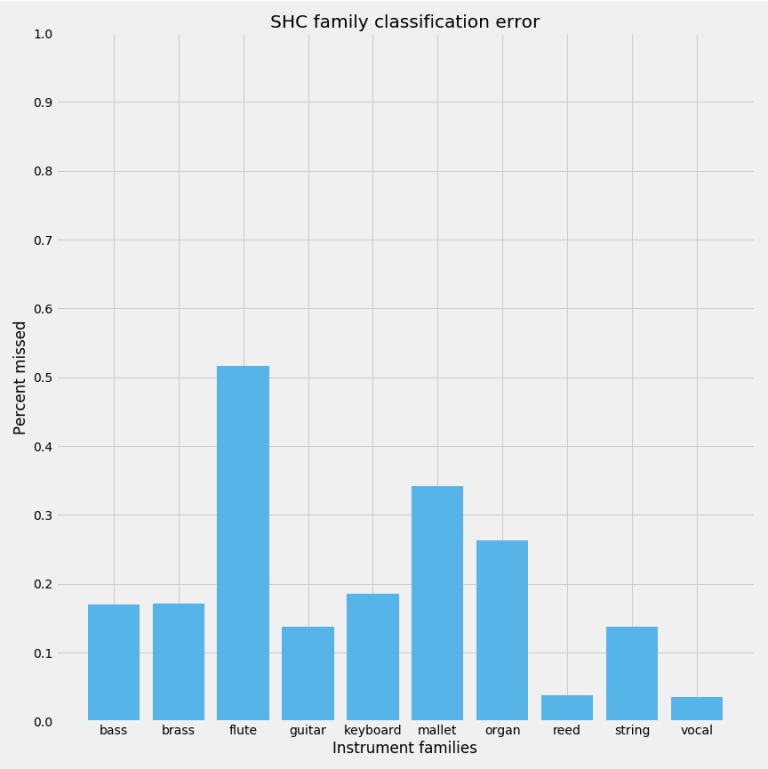

PersistFFT2 relative error
Error count
200
175
150
125
100
75
50
25
0
−24
−12
1
7
12
19
24
Margin of error

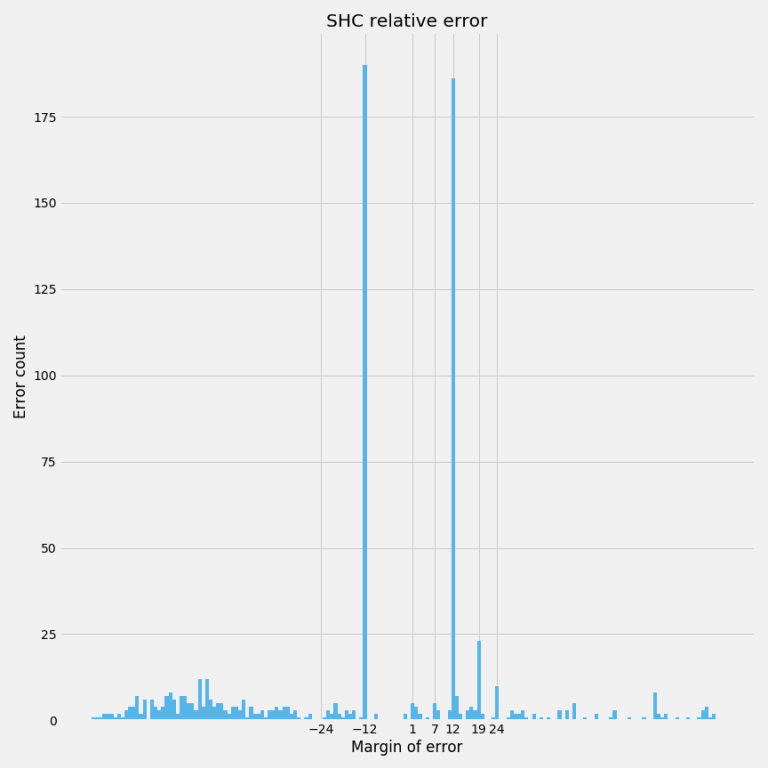

PSHC absolute error
Percent missed
MIDI pitch
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
20
40
60
80
100

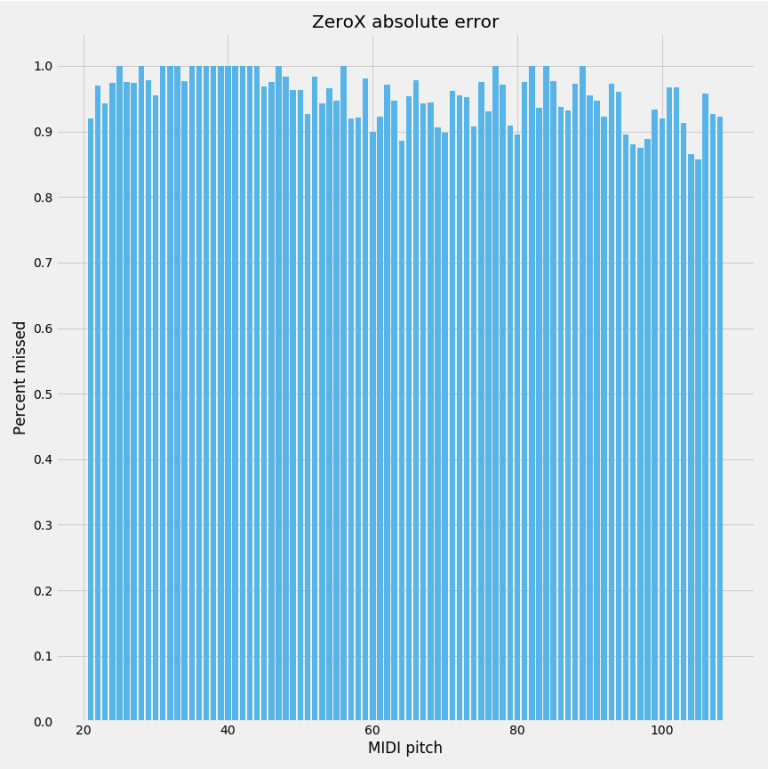

PSHC family classification error
Percent missed
Instrument families
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
bass
brass
flute
guitar
keyboard
mallet
organ
reed
string
vocal

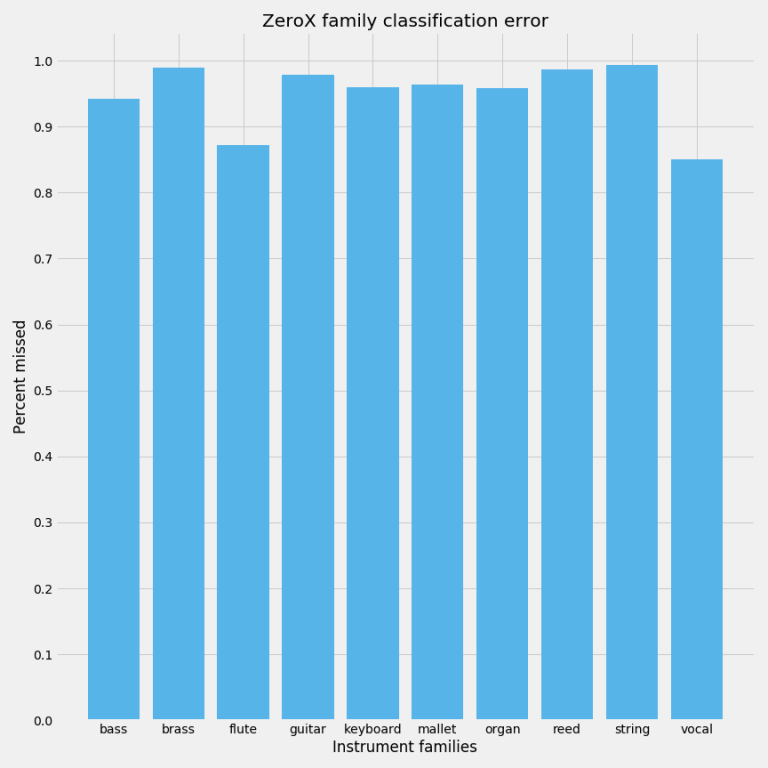

PSHC relative error
Error count
200
175
150
125
100
75
50
25
0
−24
−12
1
7
12
19
24
Margin of error

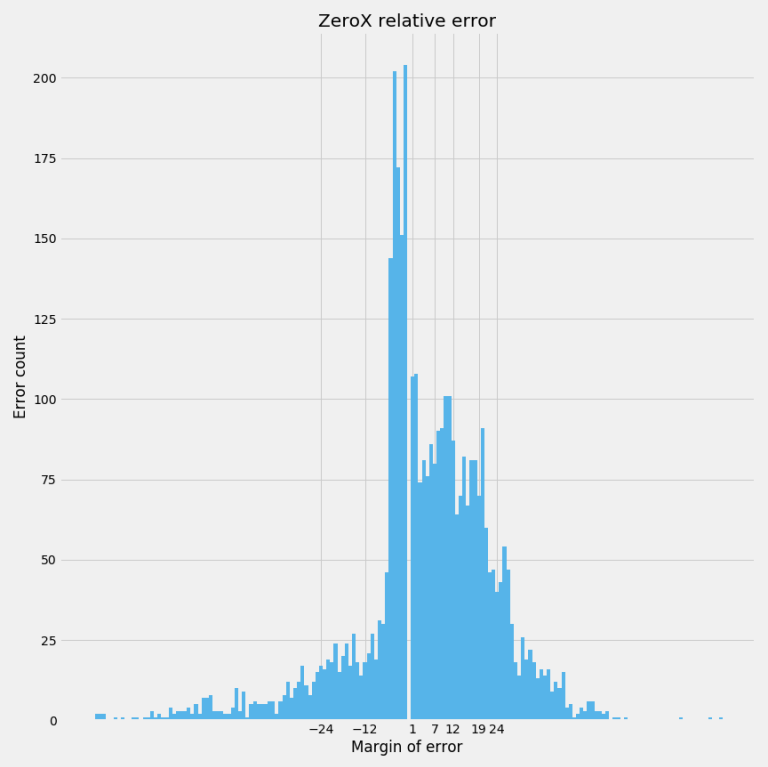

SHC absolute error
Percent missed
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
20
40
60
80
100
MIDI pitch


SHC family classification error
Percent missed
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
bass
brass
flute
guitar
keyboard
mallet
organ
reed
string
vocal
Instrument families


SHC relative error
Error count
175
150
125
100
75
50
25
0
−24
−12
1
7
12
19
24
Margin of error


ZeroX absolute error
Percent missed
MIDI pitch
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
20
40
60
80
100


ZeroX family classification error
Percent missed
Instrument families
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
bass
brass
flute
guitar
keyboard
mallet
organ
reed
string
vocal


ZeroX relative error
Error count
200
175
150
125
100
75
50
25
0
−24
−12
1
7
12
19
24
Margin of error