[SQUEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SCOTT ASATO CUTHBERT: Hello, everybody. We're going to be looking today at how we can use machine learning and artificial intelligence in order to learn more about musical behavior and music theory and just music in general. Our first task is going to be how do we transform music into formats that computers can learn from? Because a score alone is not something that can be used for artificial intelligence.

And so our first task is going to be understanding how feature extraction, the conversion of musical scores into intelligent numerical data, how feature extraction is able to make machine learning possible. So go ahead and load up Jupyter Notebook and follow along as we do some work with music21, first separate from the music21 machine learning toolkit, and then with music21 features in order to learn more about some genres and other ways of classifying music.

So go ahead and start Jupyter Notebook. And we're going to create the first step, which is to understand what features are and why we might want to extract them. So what is a feature? That's the first thing we're going to work on. Feature is something really, really, really basic. Let's say that we have a piece. We'll load from the music21 corpus, our favorite Bach chorale.

So you don't even need to know what comes up from this corpus. Parse bach.bwv66.6. And just to remind ourselves how it goes, here's the Bach. So great. Let's go ahead and choose a feature from this. Let's do the parts feature. And we'll call it numParts equals 4. And let's do the time signature numerator feature equals 4. There. We've now extracted manually two features from this chorale.

Feature extraction is the conversion of musical elements, or any elements. But musical feature extraction is the conversion of musical elements into single numbers or lists, arrays, vectors of numbers. It is thus the most boring thing in the world, as you've just seen. It is just a bunch of numbers. Feature extraction sucks out the lifeblood of music. It takes everything that's beautiful in music and turns it into a bunch of numbers. It is a life sucker. It is the zombie of the musical world.

But there are tons of amazing machine learning and artificial intelligence and deep learning algorithms in the world. As far as I know, they all have one thing in common. They only work on collections of single numbers or vectors of numbers. I'm not a machine learning expert, so maybe there are algorithms that work on two or multi-dimensional arrays of numbers. That would be amazing for machine vision or what we do. But nothing, as far as I know in the artificial intelligence world, works on things like this notation, or--

[AUDIO PLAYBACK]

-

[END PLAYBACK]

MICHAEL SCOTT ASATO CUTHBERT: Sound. So if we're going to work on doing artificial intelligence or predictions or anything super amazing on music notation, at least at present, we need to convert music to numbers. So that's what feature extraction is. Basic feature extraction is pretty simple. So in this case, let's just extract automatically the meter from this piece.

So we'll write something that does it. We'll import from the typing module because we like to do things like this. Turn Python, which is a weakly typed language, into slightly stronger types so that we can understand a little bit better what's going to go on. So we'll also import from music21 the meter and stream modules. Now let's write something called extract.

Extract meter, which takes in a variable score, which is a stream.Stream stream object. Maybe it should be a stream.Score object, but let's keep it flexible. And it returns a tuple of two integers. And we'll say here that returns a 2-tuple of the numerator and the denominator of the first time signature found in a score. Returns 0, comma, 0 if the score has no meter.

Why do we return 0, comma, 0 if there's no meter? In feature extraction, it's generally assumed that we want to return out of bound number rather than raising exception if there's a mistake. And the reason is you'd hate to be running extract meter on hundreds or thousands of pieces, and you're always getting 4, comma, 4, or 3, comma, 4 or 6, comma 8, or even something wacky like 11, comma, 32.

And it's all working successfully only to have your entire feature extraction, artificial intelligence pipeline fail if there's one piece that doesn't have a time signature. Like let's say it's Gregorian chant or some kind of trance meditation. So that's why we're going to be returning 0, comma, 0. On the other hand, maybe it's a bad assumption and we should raise exceptions when something's not right.

And then we should fix our work rather than hiding errors. But we'll get to that in just a bit. First meter. But first, let's see how we do this. So let's just remind ourselves we can take from anything get elements by class, meter.TimeSignature. And if we do not have any, the first meter will return 0, comma, 0, and otherwise will return the first meter's numerator and the first meter's denominator.

We're kind of hiding an error in our encoding. Is that the right thing to do? And I think this is something that, when we're talking about it-- and I would love to talk to anybody directly about ethics and morality and who artificial intelligence is working for, because I think there are often important considerations of artificial intelligence with moral and ethical consequences when we hide things that it can't do.

And so I want you-- if you're going to be somebody who's going to be working with artificial intelligence and the rest of your life and technology to think about such things, maybe we shouldn't be hiding our errors so much. OK. So we have this extract meter to get back to this. Oops. Doesn't matter if you put a parentheses around a tuple when you return it. But if you start with one, you have to do the other.

OK, so let's take this Bach chorale and let's run extract meter on the bach. 0, comma, 0. Oh, I know why. Because we forgot to do recurse. Flat would do just as well, but recurse is a little bit faster. Oh, by the way, something I didn't mention. You might have seen that we're sending in the class, actual class meter.TimeSignature.

We could also put in the string time signature. Doesn't matter either way there. But this actually doesn't get us the first meter. This gets us all the meters. And so if there are no meters that says if there's not a single one, return that, we'll call first meter the first of all meters. So of course, it's number zero.

Now that should be there. Now extract_meter bach. Fantastic. Returns 4, comma, 4. So why would we want to extract the meter as two numbers as part of feature extraction? We might do it as part of a series of features in order to make a prediction about a piece. Let's say we wanted to predict if a piece of music is a waltz or not. So we might first think, is this a waltz?

[MUSIC PLAYING]

Or is this a waltz?

[JOHN-PHILIP SOUSA, "STARS AND STRIPES FOREVER"]

Is this a waltz?

[MUSIC PLAYING]

So what makes something a waltz? Well, it probably has something to do with the meter. We fortunately already have a meter extraction from our time signature for the piece, and now we can use this to try to figure out if something is a waltz or not. So we would assume that the prevailing time signature in a piece and the opening time signature in a piece, if it's a waltz, would be 3, 4. And therefore, our extraction should be 3, comma, 4.

That is the worst assumption ever. The most famous waltz of all time, Johann Strauss' "Blue Danube" waltz, which you just heard, doesn't start in 3, 4. It starts with a slow introduction in 6, 8 time. I happen to have a random waltz on my computer. It's Johann Strauss' waltz called on the "Blue Danube." It's not in the music21 corpus, but it's instead on my hard drive on my desktop.

So I'll need to import the converter module from music21. I mean, it's just a wrapper around loading things and stuff. And now instead of using corpus.parse, I'll use converter.parse. I'll give the full filename to-- file path to my waltz. It's a music XML file. Oh, great. So the guy who's making this recording just goes online, grabs a random music XML copy of the "Blue Danube" from the internet, and this version skips the introduction completely.

So it has 3, 4 when it should have gotten 6, 8. So it's going to get the right answer, but for all the wrong reasons. Have you ever heard the expression GIGO? Garbage in equals garbage out. Let's not do that in our own work, what we're learning from in this tutorial. Let's instead make the world of artificial intelligence a better place and always check whether our training data is correct, whether this is actually a score of the "Blue Danube," rather than just the main theme of it. Let's check things like this before we use it.

Great. So it's in 3, 4. So our extract_meter function is going to return 3, 4. Let's try that. Sorry. Don't get the answer. Give the question. There we go. So now we have our first feature that might help identify a waltz. Its time signature. Now let's do something a little bit more advanced. Let's look at this piece and see, hm, is there anything that seems to happen pretty often?

Well, it looks like the last beat, beat three of the piece or offset two is more often higher than offset one. I don't know if that's true, but that looks like something that's a possibility. So we'll create a more sophisticated feature extractor which returns the proportion of all the measures where beat three is above beat one.

First, we'll make the assumption that beat one is always at offset zero and beat three is at offset two. These aren't very good assumptions, but maybe together with searching for meter, they'll two will make a good case for whether something is a waltz or not. So here we go. We're going to eventually write something-- proportion proportion_beat3_above_beat1, which is going to take a score. Stream.Stream.

It's going to return a float because it's a proportion. And I'll just say pass now, because in order to do write this, of course, we're going to need something that figures out for any given measure if beat three is above beat one for a single measure, for one measure. So we'll take an m, which will be a stream.Measure, and this is going to return true or false. Oops. It's going to return true or false.

Someday we have an incredible production crew that's going to edit out all these mistakes, but I don't have that yet. So we'll just say that measures notes get all the elements by class note, and that returns this stream iterator. For now, I'm going to say recast that stream iterator as another stream. You can do this with any of the get something by something classes in music21. Great. So within, this is now a stream of just nothing but notes.

And so I can say the first notes are those notes that appear on offset zero. And if there's no note there, return false. Great. Now we'll check beat_3_notes are those notes that are at offset two and that began at offset two. So that should work pretty well. And if there's no note that satisfies that condition, we're going to return false again. But what if there is more than one note at the beginning of the measure or at beat three?

So we have to make sure that the first note is the first of the beat three notes and the beat_3_note is the first of the beat_3_notes. The third-- the first. Now we'll go ahead and say if the first note's pitch is lower, then we're going to measure the pitch by pitch space or midi number is less than the beat three notes, pitch space will return false.

Otherwise, we're going to return true. Great. Well, let's be good. We'll make this a second thing. We'll cut and paste that there. Meanwhile we're going to-- let's write a little test on this. So we have blue. Let's get measure zero. So m0. Actually, let's measure one. So blue.measure 1. And we'll see, is_beat_3_above_beat_1. Oops.

[AUDIO SPED UP]

This is why we test things. Great. Is_beat_3_above_beat_1 in measure one? Actually, it is. Ah. Oh, I know what it is. We have a reverse of a sign. If this pitch is greater than or equal, then we return false. Actually, if it's-- yeah. The equal sign there. And now that's working. Measure two should be true. Also measure three should be false. Let's make sure that works.

Yep. And false. Great. So that seems to be working. Here's the irony, actually. Except for the fact that our label would be completely wrong, having every single thing that should be false be true and true be false probably wouldn't change our answers very much. But who knows. We call this, is_beat_3_above_beat_1 in one measure?

It better do what it says. OK, great. Now we're going to get the proportion that is the proportion of measures in this piece where beat three is above beat one. So we'll keep track of the total number of measures equals zero, the total-- we'll call them waltz-like measures. Those where beat three is above beat one. We'll call it zero. For every measure in the score, recurse getElementsByClass measure.

There's a reason why we'll use getElementsByClass measure instead of measure zero, one, two, three. Mostly to do with the fact that sometimes there's two measure threes or they're missing measure 11 or something like that, where there's measure 5A and 5B if there's a first ending or something like that. Anyhow. For every measure, we increment the total number of measures by one.

And now we look at one measure. And we say if that's true, then the total waltz-like measures plus equals one. And we're going to return the total waltz-like measures divided by the total measures. We're coding pretty fast, so we'll just put as a to-do later. What if there are no total measures? I always make sure that we don't do a division by zero measure error. But that can be a later thing.

So let's check this proportion of beat three above beat one on the "Blue Danube." 26%. Is that really true? It seems a little bit low. One, two down. Well, we'll check that later. Turns out it's not that bad. And bwv-- sorry. Bach. Call it this time. It's actually even a little bit more. Interesting. OK.

So in any case, this is a feature we've just defined. This particular feature figures out how often beat three is above beat one. Great. What we can do now is these are individual features. Let's create something that does change a whole bunch of features together. Well, not this one. A whole bunch of features. We'll get four features. So we'll say our feature extractor will take in a score, and it's going to return a tuple of a string, an int, an int, a float, and a Boolean.

OK. What the heck? What are these things going to be? So first, what we're going to do is we're going to take in score and we're going to-- well, we're going to say the filename of the score is going to be that first string. And you can get the filename from any music21 score by getting this filePath, which is a path lib, and getting the name, which gets rid of all the directories and stuff.

Great. So we'll get the filename of something and we'll get the-- next thing we're going to get these two ints, which are going to be the meter. So we'll say the meter feature is where I get that from extract. Extract_meter on that score. Next we're going to get this float, which is going to be the beat3_feature.

So we'll get the beat3_feature is the proportion-- why do I keep doing this? Beat_3_above_beat_1 for that score. Great. Super. Oops. I didn't hit Shift-Enter. Then we want to get the Boolean. What we're going to do is if we're training a data set, we're going to know whether or not something is a waltz. So we'll say that Boolean is going to be true or false on whether something is a waltz.

And so we could say is_waltz equals if true if Waltz is in the filename, or if lowercase waltz is in the filename. Filename. So then we know the filename, numerator and denominator, float of the proportion beat three, and whether or not filename is waltz. This last particular thing, we'll sometimes call this, instead of is_waltz, we're saying we'll use generic term ground_truth. We'll also see it called the class.

So whether it a belongs to the waltz class or not. So we can call it waltz or non-waltz or true/false. Something like that. Now we want to return all these things as a tuple. So tuples are created with a comma or with a parentheses. So we'll use the parentheses. The filename, the meter_feature's first argument, which is the numerator, meter_feature's second argument, which is the denominator, the beat3_feature, and the ground_truth.

So let's see. We'll run this feature-- let's call extractor. Let's call snippet properly. Feature extractor on the "Blue Danube." I still didn't spell it right. Why do I copy and paste? There we go. Oh, we can see "Blue Danube" waltz is in 3, 4. 26% has this beat proportion. And true. It is a waltz. Let's run our feature_extractor on bach.

We'll see filename 4, 4, 35. False. Great. So eventually we would want to have a computer system that can distinguish between a waltz and not by learning all this information. Unfortunately, I don't have enough waltzes in my data set to do this. So we're going to move to the next best thing. We're going to get a jig. So let's go ahead and grab a particular jig.

We have all of these jigs in this data set, which is really useful for people to know. It's called Ryan's Mammoth Collection. It's a 19th century thing. You can look it up. It's very famous. Hi. I wanted to jump in for a second and say something about this collection. I'm planning on not using again soon. Since I made this video, I've realized that within Ryan's Mammoth Collection, there is a lot of what are called minstrel songs.

That is, songs that were performed in the style of Negro melodies, but by white performers, often in blackface and with texts which are not included in this collection, thank goodness, but with texts that denigrated Black Americans and Black experience. So I'm going to try to change this example as soon as I can, and my apologies for what I didn't know then and what I know now and what I plan to continue to do to make computational musicology a more welcoming place.

So if we're going to be looking at the BloomingMeadowsJig. We'll load that blooming equals corpus.parse BloomingMeadowJig. Don't need the abc. You can leave it in however you want. Let's look at this. In case you haven't seen a jig in a while, it's kind of helpful for understanding the jig. Generally speaking, 6, 8. We've got these violin bow marks. We could use that as a feature extractor.

If it has violin markings, it's more likely to be a jig. If it doesn't, it's less likely. Lots of Bach pieces have violin bow marks. The Bach Violin Partitas, they're not all jigs. So it's not going to be perfect, but all these things kind help. But maybe that meter extractor 6, 8 is going to be a little bit helpful. So I happen to know that this little piece is a jig. It's kind of nice short piece, but there's a whole bunch of things. So we're going to get from Ryan's Mammoth Collection.

And we'll use a different thing that we've used before. Instead of corpus-parse, we're going to search everything for Ryan. I think everything in there is as part of this Ryan's Mammoth Collection. And we'll see that within Ryan, there's going to be quite a number of pieces. 1,063. You'll see the first time you use corpus search, it takes some time, but you can use-- after you do it once, it's often quite a bit faster. Super fast. Yeah.

There we go. OK. So we're going to have used that beat3_feature. Why not? I mean, we already have a feature. And we're going to use the meters feature. But let's use one more thing. I've heard that jigs tend to come in a few keys, a few key signatures. Maybe one flat, one sharp. They don't tend to come in five flats. That's kind of a hard thing for violins. So let's go ahead and create a sharp extractor.

So we'll call it-- let's call it get_sharps, because we'll just remind ourselves that a feature extractor is nothing more than something that takes in a score or some other piece of data and returns numbers-- in this case, an integer. We'll just put a try thing. We'll return the score. We're going to go through it, recurse, get all the elements-- use plural this time-- by class, key signature.

Get the first one and return the number of sharps. Remember, if it's two flats, sharps will be negative 2. And in case we can't find it, it might be an index error because there's zero. You can always, in your own code that you don't turn in and you don't put something there, you can just say accept all errors. Return 0. Let's double check this. Do we call it-- yeah, blooming. Sharps blooming. That's one. Let's remind ourselves. "Blue Danube" should also be one.

Great. And that Bach piece should be three. Super. OK. So that seems to work a little bit. Now let's write our feature extractor. Or our multi feature extractor. Never really sure if feature extractor just refers to something that gets one feature, like get_sharps or something that gets a whole bunch of features. People aren't really clear on that.

So we're going to return, again, the string. It's going to be the filename, two integers for the metered numerator and denominator. Pretty important. Float for that kind of silly beat three thing. But we'll add one more integer on whether or not-- yeah, on the number of sharps. And instead of returning a Boolean at the end, we're going to switch this last little thing to returning an integer for whether or not something is a jig. The ground truth.

And it's because most of the machine learning toolkits really like working with numbers, so might as well make our ground truth a number. Make a filename, the filePath.name. Again, as before. You don't really need that there, but it's going to be really helpful for figuring out why something's working or what particular pieces are wrong or right. I'm hoping I still had proportion somewhere in my clipboard, but OK.

So proportion_beat3_above_beat1. I always hit Shift-Enter too early. Sharps feature is going to be get_sharps from score. And our ground_truth. In this case, we said it's going to be an integer. So we'll say int jig in filename or jig in filename. Two things to note. One, Python int on true-- int on true returns one, int on false returns zero. In fact, in earlier versions of Python, true and false were just kind of aliases for one and zero.

Thankfully, that's a little bit better. The other thing is, again, we've learned about garbage in, garbage out. I don't actually know if every single thing in this collection that has the word jig in its title is a jig, and I definitely don't know that there's every piece in this collection that isn't a jig or that is a jig has jig in the title or that-- yeah.

That there might be jigs that don't have jig in the title, just as most rags in and ragtime and Scott Joplin have rag in their title, Maple Leaf Rag and so on. But some of them don't. The entertainer is the rag. It doesn't have rag in his title. So our ground_truth isn't exactly the world's gospel truth. This is something that people neglect all the time in machine learning. They keep training the computer to replicate truths that aren't necessarily true.

And you'll see this all over the place. I was just working with a master's student on a project where we were trying to train optical music recognition to produce the exact ground truth that we had found encoded. The thing is, the ground truth had lots of errors too. So that's something to know. Anyhow, getting back to what are we going to return. Take a second and see if you can figure this out based on what we've said the return value is and what we've defined here.

Great. So we're going to return the filename, meter_feature 0, meter_feature 1, beat3_feature, sharps_feature, and the ground_truth. Great. OK, so now we just need to iterate over everything. I want to show something. When you iterate over a corpus search object, there's this thing called a metadata bundle. That is to say, you haven't actually parsed 1,063 pieces.

It just knows about the existence of 1,063 pieces. So when we say for bundle in Ryan, we can print the bundle and we can see that we have all of these different entries for pieces. So what we'll go ahead and do is instead of getting-- but not actual pieces. So to get the pieces, we can say piece_parsed equals bundle. Actually, we call it-- instead of bundle, say metadata entry.

Since outputs end there. Metadata_entry in Ryan entry. Piece_parsed is the metadata_entry.parse. That's going to take some time. Then we're going to print out the feature_extraction on the piece_parsed. You're going to learn to use this little button in a second, the Stop button, because this is going to take some time if you hit Run. But you're going to have to do it in a little bit. Oops. Feature_extractor.

Used to call it feature_extraction. That and exact_meter. Extract_meter. Here we go. So we're going to put this back here. And let's go ahead and run this. Oops. Oh my goodness. I cannot spell anything today. OK. Here we go. Let's go ahead and running this. It shows that AdmiralsHornpipe is-- I don't know this piece, but it's in 2/2. Really seldom beat three was offset to higher than offset one.

Wow. That is really, really low. And it's in one sharp. Oh, good. Flats do work. And it is zero. It is not a jig. It is a hornpipe. I know we all know these differences between reels and hornpipes and jigs and all these things. Well, maybe some of us got our pirates in. OK, we can see here BannocksOBarleyMealJig, which is in 6/8. It has this feature. It's in two sharps and there's a one here, so it is a jig.

So that works pretty well. And we can scroll all the way through. Wow, there's a lot of pieces. OK, great. And some of them are jigs. Some of them are not. OK. So now we've extracted all this data. If we put all of this into some sort of file somewhere, we will have it all done. So I'm going to take a two second break and show how we move from feature extraction to machine learning.

OK, one of the most important things we're going to be thinking about when we think about machine learning is that we're going to be training the computer on a certain set of data, training our algorithms, training our neural networks or whatever, artificial intelligence and machine learning algorithm we choose to use on some portion of the data. And then we want to see, how good is the computer doing?

So we're going to hold out some portion of the data to be our test. To see, well, if the training data is somewhat like the test data, but the computer's seen the training data and it hasn't seen the test data, how well does it learn from the training data in order to make identifications in the test data? And there are lots of different ways of doing this.

Please take a machine learning class. Take an artificial intelligence class so you can learn about one hold out cross-validation or 1/10 hold out. Lots of different ways that we can do this better than what we're going to do. But instead, for now, to make things very simple, for whom this is going to be your first introduction to artificial intelligence and machine learning, what we're going to do is we're going to divide all of our training data into two parts, all of our data into half of it as the test data and half of it as the training data.

And I'm just going to go through and almost always do everything in, this one to test, this one training, this one test, this one training. It's just going to make things easier. But there are better ways to do it. Enough for now. Let's get to some artificial intelligence and machine learning. OK. Let's go ahead and open up two files for writing. I'm going to call them-- they're on my desktop.

Oh, now you know my secret, secret path. We'll call it train_data.tab. We're going to open this for writing, and we'll call it as training. If you're not familiar with Python file handles, please take a second to look this up because otherwise this won't make much sense. But essentially we're going to have two files so we're going to be able to write to, going back and forth, one versus the other.

There's going to be better ways of doing this, but this is how we're going to do it first. OK. We're going to need to say-- for a lot of various algorithms, we're going to need to be able to say what our column names are. So we'll say our columns are going to be a series of tab separated values. That's what these dot tab files are going to be.

A series of tab separated values that join these titles, filename, numerator-- doesn't matter what these are called-- denominator beat3 proportion. I'm not going to make myself type that again. And we'll say is_jig at the end for a ground truth. Fantastic. OK. Those are columns. And first I'm going to do is write to each of our different files, write our columns with a line break at the end.

Our testing is going to write columns with a line break at the end. Great. Now we want to describe what kind of value, what kind of value will a machine learning toolkit expect to get from filename, string, numerator, integer, beat3, float. Things like that. So we'll say our descriptions are going to be-- and these are pretty standard formats but they're not Python. So look at this. It's not str. It's string. And we'll say that our integers are discrete data with an i. Not discreet like indiscreet. OK, I won't make a joke.

But discrete like integer values. But beat3 data can be continuous. It can be 1.2, 1.21, so on. And then another discrete data for the number of sharps. And then for is_jib, it's also at 0, 1, so it's going to be discrete data. Fantastic. We're going to write thees to these files, the descriptions also. Boom. OK. Great.

We also tend to put these meta variables in, and they say, if anything is special, if anything is unlike the rest. Well, the ones that are special is the first thing. String is just a metadata. That is to say, it just means something so that we can more easily diagnose problems and look things up later. We don't actually need it. We're not going to study the filename. And then these are just normal things. Numerator, denominator, beat3, number of sharps.

And then at the very end, the final thing is our class. The thing that we're trying to learn. We're trying to learn, does this information, 98.11112, describe a jig or not? The class of objects that is a jig. Great. So these are our meta variables. And we're going to write them to, again, our-- do I still have that? No.

I'm going to write that to our training and to our testing. Great. Here. I can do this anyhow. We can cat that file. These magic things are stuff you can do in Jupyter that you can't do in normal Python. But it'll see train_data tab. Yep. We can see what we've written out there. OK, great. Now we're going to take this and we're going to write more things to it.

We're going to write into these two files information about from our feature extraction. So we're going to go back for each piece in Ryan. Enumerate Ryan. And enumerate says our first thing goes into piece and we'll call it i. Actually, We'll not call it piece. We'll call it metadata_entry. Great. And we'll say piece_parsed is our metadata_entry.parse.

Great. And our features are for feature_extractor on piece_parsed. Piece_parsed. And now we're going to make a string out of that feature. String equals tab separated version of the string value of each element for the element in features. OK. And we should call it f. Fantastic. And if i is even. No, if i is odd, we'll write it to the training data with a new line.

And otherwise we'll write it to the testing data with a new line. Great. Hopefully I didn't type anything wrong. We'll find out in a second. Great. It seems to be working. It's slow. One of the things I'm going to eventually learn is that everything's slower when I'm screen recording. So we could see that, yep, there's a bunch of things in here. Most important one, can we get a jig?

It has a one at the end if it's a jig, zero at the end if it's something else. These jigs are 6/8. Oh, look, there's a jig. It begins at 9/8. So maybe it's 8 that's more important. I don't know. We'll see if the computer can figure this out. We don't need to look at all 1,000 entries. Jupyter does a nice way of not putting that all together.

OK, great. So now we have something that we can feed into an artificial intelligence classification program. So we don't have one so we're going to write one of our own. So we're going to say neural network. Just kidding. No, that's not what this class is about. That's other classes. We're going to install one. So go ahead and install orange3, which is a really great package.

Oh, hey, I already have it installed. Yours probably takes a little bit longer than that. Great. Super. So now I have orange3. Orange3 names its stuff kind of funny. So I always-- because I've been using it for a long time, I just import it. It's not orange3. I just call it orange. Great. Now what we're going to do is we're going to load in our training data from an orange data table based on that information.

So a train_data.tab. I reverted myself to using camelcase things. I probably shouldn't, but I'm not going to rewrite everything now. So it doesn't look like much, but I have a lot in my sleeve. OK. So we're going to load those two. Great. We can look at train_data little bit. Yep. Whole bunch of stuff. These are, I think, matplotlib-ish type things. OK, great.

So what we want to do is create a bunch of learners, things that can classify and understand data and try to make guesses about things. Our first learner is kind of like the placebo of learners. It's called the majority learner. And there's other ways of doing this, but we'll use this one. And the majority learner is pretty cool. It just looks at the data.

It looks at what we put in the training data and it says, you know what? There's about three times as many things that are not jigs as there are jigs. So I can get 3 out of 4 right if I just learn to always vote with the majority. I don't know anything about music, but I'll just say-- I'll always say that's not a jig. And you know what? I'll usually be right.

So never classify things according to, hey, is this right or wrong? Did it get it right more than half the time? No. Classify how you're doing against, did it do better than the majority learner? You're going to now, if you don't know what this is, we're going to learn what a k nearest neighbors is, knnLearner. Classification. Classification. KnnLearner.

And this is a pretty standard artificial intelligence learning algorithm and so I'm not going to say too much about that. So we create a kind of learner thing, and our classifier is a learner that's going to work on a particular set of data, our training data. OK, good. So we create a learner, create a classifier. Hasn't actually learned anything yet, but it's going to be ready for doing that. OK.

Now we're going to check how often does the majority learner get things-- how many times does it get it correct and how many times does the k nearest neighbor's learner get things correct? And how many total things are there? We get that as a float. Still thinking too much how to think like Python too. So how many test data there are. Go and see. OK, 532 in the test data.

The training data does not need to be exactly the same size, or even close to the same size as the test data. But in this case, they are. Great. So now we're going to iterate through each of the rows of our test data not our training data. So we're going to know what row number we are on, i, and then test row. And then the majority learner is going to make a guess based on the data in the test row.

However, the majority learner is really stupid, so it's guess is always going to be, it's not a jig. Then k nearest neighbor is going to make a guess. We hope that k nearest neighbor is making a guess based on some intelligence that it has, or fake intelligence. I mean, it's called artificial intelligence for a reason. And then we get to see in this case what the actual answer is, what the class-- whether it's a jig or not. So that's our real answer.

So we know this but k nearest neighbor is not allowed to look at this. We really hope the computer's not peeking, otherwise all artificial intelligence is kind of screwed. OK. We'll just print out some test data that the row i is a little bit of information about our set. That should be good. Great. We'll print out that the answer to row i is jig, if realAnswer. I think that works. Else not jig.

And then we'll put a tab. We'll say k nearest neighbor got it right or wrong. Right if knn's-- if the k nearest neighbors guess is the same as the realAnswer, else wrong. Whew. That's some pretty tough stuff. Our thing can't even get that right. Good. OK. We'll print a blank line after that. If the majority guessed right, then the majority correct increment.

If k nearest neighbors guess is right-- so we're just checking at this point how good is this. k nearest neighbor's correct, it is going to be one bigger. Great. So we'll go ahead and let that run. Oops. Colons are very nice. Oops. I screwed something up. Oops. There we go. I have to cast this as a list because-- where is it? This data can't be sliced, but lists can.

OK. Is that going on? I probably need to reset my data, so let's go ahead and do that. It's going to be printing. Cool. Not jig. k nearest neighbors got it wrong. Not jig. k nearest neighbors got it right. Good. So, hey, we see a lot of got it right. I like that. I like that. OK. Here's the moment of truth. Without machine learning, there's a majority learner. Majority correct.

Always put things into print percentages. Always round your percentages to something reasonable. Nobody wants to see all those dangling digits. OK. So it got that number right. And we'll say with knn machine learning, we got the knn correct. How well do we do? Without machine learning, got it 77% right. k nearest neighbors got it 95% right. Wow.

OK, that's pretty awesome. That's a lot better than I thought when I thought I would do this. By the way, one of the things we can do. You probably want to keep track of your data and don't do this, but we'll just go ahead and check. Does it make a difference if we swap the training and the test data? Oops. Not training data. Train_data.

Do we get a fluke? And we can go ahead and run. 72 and 94. So yeah, still pretty good. Doesn't matter which one we do. I'm going to switch that back because I will totally make that mistake. But yeah. So we're doing really well with this. So this is how you can do artificial intelligence with machine learning.

But that's been a little bit hard. So fortunately, music21 has some tools to make a lot of these things even easier here. So music21 feature extraction will be the last topic that I'm going to talk about today. So we can go ahead and import from music21 the features module. Great. So within the features module, there are things called DataSets.

So let's create training_set, which will be a DataSet with a class label of is_jig. We call it ground truth or whatever. And we'll create an identical one for our test set. Testing_set. Great. So these are just things that are ready to add two important things. A set of data that is streamed, so a whole bunch of streams will be added to both the training and the test set, and also a set of feature extractors.

I'm not going to explain today how to write new feature extractors in the way music 21 expects them, but they're basically pretty similar to what we've just been writing with functions, except they're kind of expected to be classes. But we don't really have to because music21 has a whole bunch of feature extractors.

Some of them come from this amazing toolkit called jSymbolic, or they're re-interpretations of Corey McKay's jSymbolic toolkit, which is in Java, and they have numbers like D1 is the overall dynamic range features. Not all of them have been implemented I's are instrument features. You can probably see that the percentage of all notes being played by electric guitars, if you know something about jigs, is probably not going to be very helpful to classifying whether or not something is a jig.

But maybe the average melodic interval will tell you something like that, or how often there are melodic thirds or melodic fifths, or maybe other things might tell signs. So these are melody features, pitch features, and so on. There's rhythm features. That's going to be pretty important. R31, initial time signature feature. R32, compound or simple meter features, and so on.

So we could add a bunch of these features in, and we remember the names. Within the music21 documentation, there will be certain things that will tell you TripleMeterFeature set to 1 if the numerator of the initial time signature is 3 or 0 otherwise. And this shows you how it can work. So we can give a particular piece and it will tell us that bwv66.6 is 0, not in triple meter.

OK. We've already done something really, really great. And so I will tell you that if we use this very similar feature extractors to the ones we did before, we're going to get the same results. But one of the other things we can do is we can just use a whole bunch of pretty random ones. So we'll get these features by ID. And these are a bunch of features that I used to classify various pieces of-- I think it was 16th-- 15th century music. No, 16th century music.

So they're not exactly super tuned for what we're going to do. A lot of them are melodic features and so on. Let me see. Scroll over, see which ones haven't done pitch features. Pitch feature 8, 10, rhythm feature 15. Great. And we can see what they are. I'm already going to tell you, no surprise, these aren't going to work nearly as well as the ones we just made.

Note density feature isn't going to help us very much. I would have thought range might have told us something about a jig, but maybe it's not. Some people will tell you, hey, just throw lots of feature extractors in. It can't hurt. It kind of can. But because we often don't have enough data to make use of all these features. But let's give it a shot. We're going to add these feature extractors to our training set and to our testing set.

OK, so our training set and testing set is now ready to go where any stream we add to these sets, we're going to have these feature extractors added to them. So we'll say is for i, add metadata_entry in enumerate Ryan, again. Piece_parsed equals metadata_entry.parse.

If i is odd, then training or test will be-- so we only have to write things once, we'll set our training or test set will be the training set in this case. Otherwise our training or test will be our testing set. Great. Filename. Is the piece parsed? FilePath.name. Is_jig. Same thing we're doing before. Here we're not going to-- music21 smart enough. It doesn't need us to cast this to an integer. It'll figure out what to do itself. OK.

I'm going to say what row are we, is this a jig, and what's its filename. That's easy part. And then whatever data set we're working with, training or test, we're going to add the data from-- here's our stream, our score. What did we call it? ClassValue. What is the right answer? It's whether or not this is a jig. And then the identification for this will be the filename.

Boom. It's going to go through. It has 1,000 of these to run, so I'm not going to wait for all of it to happen. I'm going to stop for a second because we don't have all that time in the world. That's part of the reason why we also don't want to do everything. But that small training set, we're going to process, and we're going to take that training set and write it out to train2.tab.

No, we're not. You should do that because you have time to wait. But nobody wants to do that. We'll do the same thing with the testing_set.process and the testing_set. We're going to write that to test2.tab. I'm not doing that right now because they do in those old cooking shows, I've already done all this and I'm going to load it from what I've already done. So you know what I mean.

The old cooking shows where they put they put the raw turkey in the oven, and then they just immediately go to the other oven and the turkey-- sorry. I you all can read my mind, right? That's what you pay me to do, make you read my mind, not actually teach. That's fantastic. OK. Yeah. Anal enough. I'll make this line up so I don't make any typos. It's on my desktop. Test2.tab. That's why it broke.

OK. So again, we're going to do everything as before. Why do everything as before when you can just copy and paste. Because now we've already written everything out for-- yeah, this whole process and write thing takes care of everything we did before where we-- where is that?

Where we had to define-- we had to open our datas, we had to define our columns, we had to write that out, define our descriptions, write out all our meta things, extract what our feature extractors do, all that stuff of these little process, and then write did all that for us. So now we're actually back onto feature extraction is done. Let's do machine learning.

Anybody who tells you that the machine learning is the only hard part and that feature extraction is easy stuff is lying to you or doesn't know what they're talking about or whatever because feature extraction and doing it right is really, really hard. So we're going to basically copy and paste everything that we did before. Totals. Something like that.

Great. We're going to run this again and then we're going to print everything afterwards. Great. And that's going to go through and it's going to print a couple things. How are we doing? We've given it a lot more things to train on, so a lot more data. These are much bigger files because of all the different feature extractors, so it takes a while.

You can see in all this time, it's just done one row. Two rows. Are we going to wait for-- I think it's, like, an hour and a half. I ran this at 2:00 in the morning. Are we going to wait an hour and a half? We can all do that. No. I'm going to hit Stop for a second on the recording, and then I'm going to load up our results.

OK. Here are the results I got. Again, our majority learner got 77% right. And our k nearest neighbors super intelligent artificial intelligence only got 4% better. I mean, I don't even know if that meets the margin of error in a political poll or something. So all that extra time didn't actually improve anything. What's important is always to choose your-- choose your feature extractors well.

The fact that I didn't include that one simple extractor about the time signature made all the difference in the world, I think. OK. I just want to show that you can print all of the different features onto various charts and zoom in and stuff like that if people want to do that. But that'll be something for another time. This is what is available to you now.

Maybe not this kind of low level one, but using these feature extractors, simple ones like sharps and meter and-- what was that? Percentage of compound time, range features, things like that, in order to classify pieces is incredibly powerful. And when put together with the right machine learning and artificial intelligence algorithm, there's a lot of possibilities for the future of computational music analysis.

This lecture has run pretty darn long, so I'm going to say for a second little supplement on what are some of the things that I have learned through using computational artificial intelligence? What are some of the projects other people have done, and why the world state of the art black box deep learning or other sophisticated algorithms might not always be the best ones that we want to use when we're doing artificial intelligence and machine learning techniques on musical scores.

But we'll leave that for another time. Thank you so much for listening. Go ahead and play around with some data sets. You've learned how to use corpus search to be able to find various pieces. there's a music21, I think user's guide chapter and tutorial on how to do more interesting things with that. There's the paper written with Chris Ariza and Lisa Friedland on using feature extractors with music21.

And there's also some other music21 feature extractor papers that were written with MIT undergraduate UROPs on how we can use feature extraction to go beyond what we can do with midi, to really look at the nuance of musical scores in feature extraction. I'll be sending all those around. The meantime, please go back through the most interesting parts of this lecture to you, hit pause, and play around with your own data sets. Good luck in applying this to future projects.