CSC 558 Data Mining and Predictive Analytics II, Spring 2020

Dr. Dale E. Parson, Assignment 3, Classification of time series data consisting of four-field MIDI[1] noteon/noteoff messages (Musical Instrument Digital Interface) similar in nature to a time series of network packets.[2]

DUE By 11:59 PM on Wednesday April 15, 2020 via make turnitin on acad. The standard 10% per day deduction for late assignments applies.

We will have some on-line work time via Zoom during class time.

Perform the following steps to set up for this project. Start out in your login directory on csit (a.k.a. acad).

```
cd  $HOME
mkdir  DataMine  # This may already be there.
cd  ./DataMine
cp  ~parson/DataMine/midi558spring2020.problem.zip  midi558spring2020.problem.zip
unzip  midi558spring2020.problem.zip
cd  ./midi558spring2020
```

This is the directory from which you must run **make turnitin** by the project deadline to avoid a 10% per day late penalty. If you run out of file space in your account, you can remove prior projects. See assignment 1's handout for instructions on removing old projects to recover file space, increasing Weka's available memory, and transferring files to/from acad.

You will see the following files in this **midi558spring2020** directory:

README.txt	Your answers to Q1-Q15 questions go into here, in the required format.
spring2020concert.arff	The primary handout ARFF file for assignment 3.
spring2020notenorm.arff	An auxiliary ARFF file to analyze.
BIGspring2020notenorm.arff	Another auxiliary ARFF file to analyze.
pypardata/	My Python library for generating time-lagged attributes.
makefile	Files needed to **make turnitin** to get your solution to me.
checkfiles.sh	
makelib	
maketimelag	

ALL OF YOUR ANSWERS FOR Q1 through Q15 BELOW MUST GO INTO THE README.txt file supplied as part of assignment handout directory **midi558spring2020**. You will lose an automatic 20% of the assignment if you do not adhere to this requirement.

BACKGROUND:

[1] http://midi.teragonaudio.com/

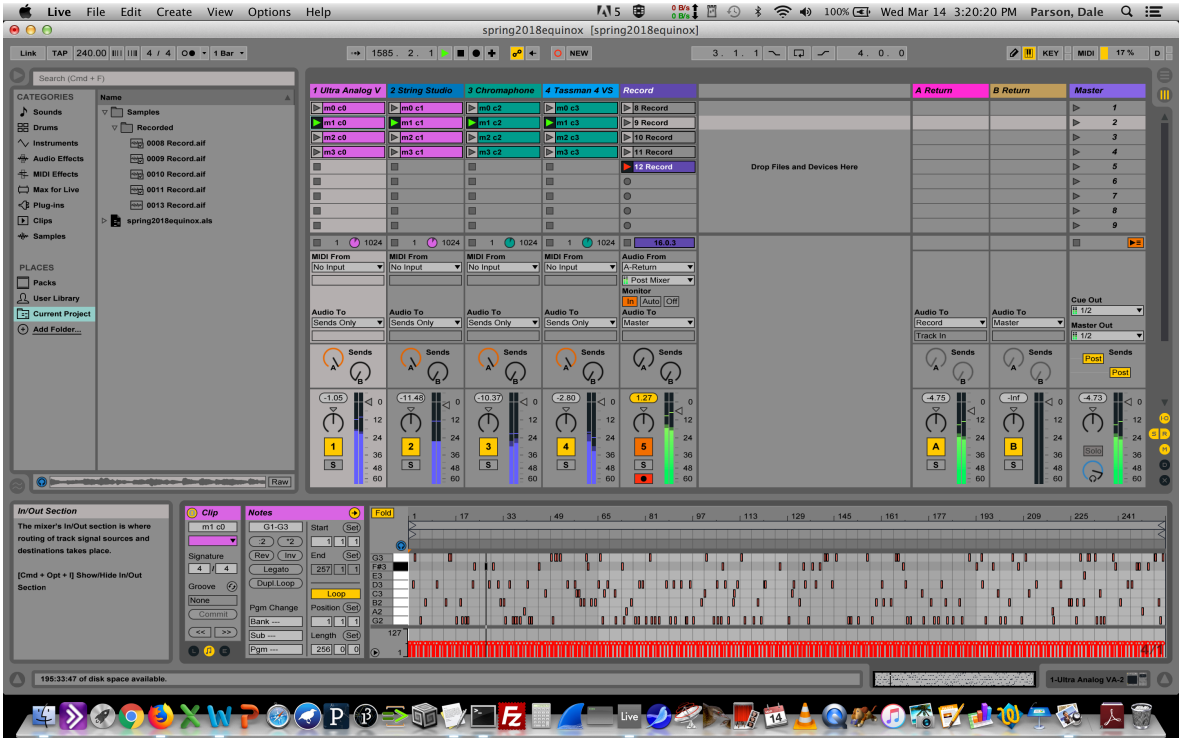
[2] This project derives from research into real-time analysis of MIDI data streams that I did in 2005-2006 http://faculty.kutztown.edu/parson/pubs/dafx06_dparson_final.pdf and work that students and I did in real-time algorithmic music generation in 2009-2010 http://faculty.kutztown.edu/parson/pubs/ICMC2010DaleParsonScrabbleMidi.pdf .

The dataset in spring2020concert.arff consists of 4 movements[3] of a generated musical score for backing tracks over which I have performed live music on a webcast, with 4 instruments in each movement, where an instrument takes the form of a MIDI channel[4]. The MIDI data packets we will analyze contain 4 fields each as in the following table.

MIDI command	MIDI channel	Packet's data1	Packet's data2
noteon or **noteoff**	0, 1, 2, or 3	note number range [0,127]	notevelocity [0,127]

Our project uses only two MIDI command types out of many available, **noteon** for sounding a note, and **noteoff** for silencing one previously sounded. The ARFF files use only **noteon**, since there is a one-to-one correspondence between **noteon** and **noteoff** data, varying only by note length (time from **noteon** until **noteoff**), a property that we are not analyzing. A numeric MIDI **channel** in the range [0,15] maps to an instrument in a synthesizer; our design uses only the first four [0,3]. A noteon packet's data1 field maps to a **note number** in the range [0,127], giving pitch similar to 128 keys on a keyboard. A noteon packet's data2 field maps to a **note velocity** in the range [0,127], giving volume (amplitude) of the note.

Open **spring2020concert.arff** in Weka's Preprocess tab. This dataset is a score for a 4-movement musical accompaniment piece that I used in a March 24, 2018 webcast[5]. My Jython generator that generates this score also generates 16 MIDI sequence files (sequences of these MIDI data packets), not included in your handout, that I used within a software synthesis framework in the performance. There are 16 MIDI files for 4 movements X 4 channels. Here is a screen shot of the software synthesis framework I will use during the performance, called Ableton Live. The channel-mapped instruments run in 4 vertical columns, and the 4 movements run in 4 horizontal rows.



[3] A **movement** is an interval of time during which each instrument plays in 1 consistent **mode**, and they play more-or-less together.
[4] Four short recordings of backing-track sounds synthesized for movements [0,3]:
http://faculty.kutztown.edu/parson/spring2018/csc558sp2018movement0.mp3
http://faculty.kutztown.edu/parson/spring2018/csc558sp2018movement1.mp3
http://faculty.kutztown.edu/parson/spring2018/csc558sp2018movement2.mp3
http://faculty.kutztown.edu/parson/spring2018/csc558sp2018movement3.mp3
[5] http://radio.electro-music.com

Here are the attributes in spring2020concert.arff, one per **noteon** message (a.k.a. data packet).

movement	**Tagged** numeric movement in which this note sounds in the range [0,3].
channel	MIDI instrument (channel) in which this note sounds in the range [0,3].
command	MIDI command, always **noteon** for this dataset.
notenum	MIDI note pitch in the range [0,127]; notenum % 12 == 0 is a C note.
velocity	MIDI note volume in the range [0,127].
tick	MIDI timing of this note in beats within its movement, starting at 0.
ttonic	**Tagged** data identifying the movement/channel's primary note ("do" in its scale), [0,11].
tmode	**Tagged** data identifying the movement/channel's mode, i.e., its musical scale.

The attributes described as **Tagged** are meta-data that, while conceptually part of the score, are not part of the per-channel MIDI data streams that we want to analyze. The attributes described as "MIDI" are part of those MIDI data streams.

MIDI notes do not represent audio data samples as in assignments 1 and 2. They represent commands sent to electronic musical instruments. You can think of them as domain-oriented network packets, which is what they are. Each MIDI packet in this dataset represents one **noteon** command. There are a few things you need to understand about *notes* and *modes* (a.k.a. *scales*) before beginning analysis.

0	1	2	3	4	5	6	7	8	9	10	11
C	C# / Db	D	D# / Eb	E	F	F# / Gb	G	G# / Ab	A	A# / Bb	B

There are twelve notes in a Western chromatic scale, after which this sequence repeats at the next higher octave. These are the white & black keys on a piano. If a note is divisible by 12 (note MODULO 12 == 0), it is a C note, although we don't care about its name for this assignment. Note number MODULO 12 gives its name, and note number / 12 gives its octave. Whereas the doubling of a harmonic frequency in assignments 1 and 2 gave the same perceived note at an octave higher, here adding 12 to a note's number gives the same note at an octave higher, i.e., a doubling of its primary harmonic frequency component.

Our primary goal in assignment 3 is to extract the musical **mode**, also known as its scale, for the composite key of a (**movement**, **channel**). Different musicians on different channels can play different modes within the same temporal movement. We can extract **mode** directly by mapping from (**movement**, **channel**) ordered pairs in the data, just like reading a musical score. However, a main goal is to extract musical mode from non-tagged attributes, primarily **channel** and **notenum** attribute-values. Each instance carries only a single **notenum**, but a **mode** is composed of multiple notes (actually note intervals) as the following Jython data from the project generator illustrates.

```
# Major modes (major 3rd)
__IonianMode__ = [0, 2, 4, 5, 7, 9, 11, 12]     # a.k.a. major scale
__LydianMode__ = [0, 2, 4, 6, 7, 9, 11, 12]     # fourth is sharp
__MixolydianMode__ = [0, 2, 4, 5, 7, 9, 10, 12]   # seventh is flat
# Minor modes (minor 3rd)
__AeolianMode__ = [0, 2, 3, 5, 7, 8, 10, 12]    # natural minor scale
__DorianMode__ = [0, 2, 3, 5, 7, 9, 10, 12]     # sixth is sharp
__Phrygian__ = [0, 1, 3, 5, 7, 8, 10, 12]       # 2nd is flat
# Locrian has Minor third, also known as a diminished mode because of flat 5th
__LocrianMode__ = [0, 1, 3, 5, 6, 8, 10, 12]        # 2nd is flat, 5th is flat
# Chromatic is not a mode, it is just all the notes.
__Chromatic__ = [i for i in range(0, 13) # actual notes are [0, 12]
```

The values [0,12] in those 8 lists are intervals between notes in a mode; the lists include 12 as the octave partner of the tonic at 0, but otherwise 12 can be considered as 0 with respect to notenum % 12. However, the actual **mode** in the MIDI data depends both on the **tonic** note of a movement-channel pair and the **intervals** of other notes **in relation to that tonic**. For example, if the tonic note is 62, which is the D above middle C with a note number of 60 (62 % 12 = 2), then the notes in the Aeolian mode (see above) start at the tonic, i.e., [2, 4, 5, 7, 9, 10, 12, 14]; these intervals are normalized to the range [0,11] by using MODULO 12 (% 12), and offset to show the tonic at the leftmost position.

My Jython generator uses two statistical curves in generating individual notes, the **Gaussian**[6] (a.k.a. Normal) curve with the peak at the **tonic** note, or the **uniform**[7] curve that scatters notes uniformly. It uses only one of these curves for a given **movement-channel** of notes. The **Gaussian** distribution is more predictable and sounds more consonant to the listener when used with a consonant mode.

To find the target attribute **tmode** from only the non-tagged MIDI attributes, we must do the following steps.

1. Find the actual **tonic** from then MIDI data, as the most frequently occurring interval (normalized via %12), in each **movement-channel** pair. I am guaranteeing that a given **channel** does not change **tonic** during a **movement**. We can confirm our investigation to find **tonic**, temporarily, by consulting tagged attribute **ttonic**.
2. Translate each **notenum** in each instance to normalized attribute **notenormalized**, which is its distance MODULO 12 from its **tonic** of the previous step. This step is similar the normalization of the fundamental frequency of assignments 1 and 2 into amplbin1; here we are normalizing the tonic for each **movement-channel** pair into tonic *interval* (distance) of 0.
3. Collapse **notenormalized** attributes from multiple instances across time into multiple *time-lagged attributes* within each instance in order to infer the actual **mode**. It is not possible with high accuracy to infer multiple-interval modes appearing on the previous page with only one interval (in **notenormalized**) worth of data. We need multiple notes that are adjacent in time in order to infer the mode being played by analyzing only non-tagged data in the musical data stream. That statement is true for human listeners and for our analysis. I am guaranteeing that a given channel does not change **mode** during a movement.

These are the main points you need to know about the application domain.

PLEASE PUT YOUR ANSWERS INTO **README.txt** as before. Save over top of the original file **spring2020concert.arff** only when you see the instruction **SAVE**. You can save work-in-progress in other files, but please do not turn them in or over-write **spring2020concert.arff** by accident. Questions Q1 through Q15 are worth 6.66% each.

PREP STEP A: Run Weka's Preprocess filter **StringToNominal** for first-last to translate string attributes, then run filter **RemoveUseless** and **SAVE** this file over top of **spring2020concert.arff**.

Q1. What attributes did **RemoveUseless** remove and why?

PREP STEP B:
B.a. Temporarily remove all attributes except **movement** & **ttonic**.
B.b. Temporarily run unsupervised attribute filter **NumericToNominal** only on **movement**, inspect in Preprocess.
B.c. Next, temporarily run filter **NumericToNominal** only on **ttonic**, inspect in Preprocess.

[6] http://www.muelaner.com/wp-content/uploads/2013/07/Standard_deviation_diagram.png
[7] https://bgsu.instructure.com/courses/901773/files/32348049/preview?verifier=KafdUjjLOHIHGljYCDzfKONmzorGoVo7uTgoKP7Z

B.d. Use a classifier (rule, tree, statistical, or other) to find a 100% Correctly Classified Instances with Kappa statistic = 1 to find the **movement** -> **ttonic** mapping.

Q2. Paste the rule, tree, or other structure here. Identify the classifier. Paste Correctly Classified Instances through Root relative squared error as well.

Q3. Find three other rules, trees, formulas, tables, or other structures that give Correctly Classified Instances = 100% with Kappa=1 and paste them here. Identify the classifiers.

PREP STEP C: Execute UNDO once to make **ttonic** numeric while leaving **movement** as nominal. Run LinearRegression & M5P.

Q4. Paste their formula/tree and results (correlation coefficient and error measures) here. Interpret their formulas in light of Q2 and Q3 classifiers. Do they agree with those previous results? If so or not, how do you interpret the formula of LinearRegression and the tree/formula(s) of M5P to agree or disagree with the classifiers of Q2 and Q3?

PREP STEP D: Load **spring2020concert.arff** that you saved after RemoveUseless. **Remove** attribute **ttonic** and **SAVE** this file over top of **spring2020concert.arff**. Now run J48, NaiveBayes, and BayesNet to classify **tmode** as the target attribute.

Q5. Which classifier is the most accurate, the second most accurate, and the third most accurate of these three, in terms of Correctly Classified Instances and the error measures? PASTE the classifier's structure (tree, table, or typed copy of the BayesNet's graph's tables) AND accuracy/error measures **ONLY** for the most accurate of the three.

COMMENT 1: Observe the importance of attribute **movement** in achieving this degree of accuracy. Attribute **movement** is a tagged attribute that is not part of the musical data. Using it to infer **tmode** is like reading the musical score to determine the **mode** used in each movement.

Now we need to determine the tonic from the note musical data and verify that we achieve the same movement -> tonic correlation as in Q2-Q3.

We need to find some "MODULO 12" interval values in Weka. Weka filter NumericTransform with java.lang.Math.floorMod does not work because there is no way to pass two arguments from Weka. We will use Weka AddExpression filters that incorporate the floor() function to find the integer MODULO. Here is an example AddExpression expression and a Python function.

```
noteinterval = aN - (floor(aN / 12) * 12)  # AddExpression for Weka attribute aN, N is attribute number
# The above line gives "aN % 12", a.k.a. "aN MODULO 12".

>>> def modulo(number, divisor):
...     quotient = floor(number / divisor) # floor discards any fraction
...     result = number - (quotient * divisor)
...     return result
```

PREP STEP E: Use AddExpression to create a new attribute **noteinterval** derived from **notenum** as in: **notenum** MODULO 12, for 12 steps in a scale. This will give a range of values [0,11], with peaks for the **ttonic** values of Q2-Q3. BE CAREFUL NOT TO INCLUDE EXTRA SPACES IN AddExpression's created-attribute name, because Weka includes such spaces in names. Also, BE CAREFUL WITH SPELLING! I misspelled partitionedticks as partionedticks in a later step, wasting some time.

Q6. Correlate **movement** -> **noteinterval**, with **noteinterval** as the target attribute, using temporary filtering techniques identical to Q2, i.e. with only **movement** and **noteinterval** in the dataset. PASTE the resulting classification structure and accuracy/error measures here. Does the model (rule, decision tree, or table) agree with your results in Q2? Does the accuracy agree with your results in Q2? Explain any differences in accuracy/error measures compared to Q2. Hint: Inspect the names of the two attributes used in Q2 versus the names of the two attributes used in Q6.

PREP STEP F: Execute **Undo** as necessary to get back to the state where you had just created derived attribute **noteinterval** via AddExpression; do NOT remove **noteinterval**. Use AddExpression to create a new attribute **tonic** that is an identical per-instance copy of the value in **movement**. Reorder the attributes to put the **tonic** copy of **movement** immediately after **movement** in the attribute list, without changing any other relative ordering. I have found that the next step will not work if **tonic** is the last, target attribute. Use unsupervised attribute filter **MathExpression** to over-write attribute **tonic** (which currently holds a **movement** number) with the per-movement tonic found in step Q2. You will can use a nested ifelse

ifelse(CONDITION1,RESULT1,ifelse(CONDITION2,RESULT2,RESULT3))

expression that is conditional on the **movement** number to map **movement** number to its **tonic** that you found in Q2. Here is some poor documentation; wer are NO doing the Math Discretization section, but it does show an ifelse example:

https://waikato.github.io/weka-wiki/using_the_mathexpression_filter/

I have not been able to get MathExpression's "==" comparator to work, nor the logical "&&" operator, so I used a "<" comparison in a nested **ifelse** expression to do this mapping. MathExpression actually mutates (changes) all attributes by default, so make sure to set its **ignoreRange** ranges to exclude all attributes except **tonic**. MathExpression uses the symbol A to refer to the current attribute being mutated. You do NOT need to supply the attribute's number, since it is always just the current attribute being mutated.

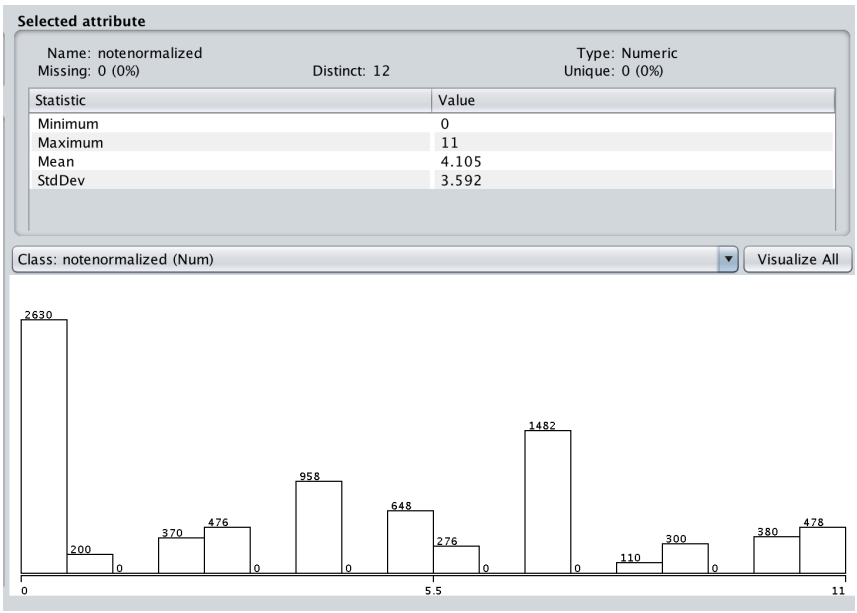
Q7. Show your MathExpression command line here.

Q8. Correlate **movement** -> **tonic**, with **tonic** as the target attribute, using temporary filtering techniques similar to Q2, i.e. with only **movement** and **tonic** in the dataset. PASTE the resulting classification structure and accuracy/error measures here. Does this result agree with your result in Q2? Explain any differences in accuracy/error measures compared to Q2.

PREP STEP G: Execute Undo as necessary to get back to the state where you had just configured derived attribute **tonic** via **MathExpression**. Create a new **AddExpression** derived attribute **notenormalized** according to the formula:

notenormalized = (noteinterval - tonic) MODULO 12

This attribute should give a Preprocess histogram similar to this one. Weka's new (2020) bug of not keeping track of the proper class attribute may color the following histogram, but the shape and numbers on columns should be identical to the following image.



Remove **notenum** and **noteinterval**, since we now have their normalized derived attributes in **tonic** and **notenormalized**. Remove **velocity** since target attribute **tmode** does not depend on it.

Reorder the remaining 6 attributes so that **movement** is attribute 1 (a1), **channel** is attribute 2, and **tmode** is the last, target attribute. Keep the relative order of the other attributes unchanged. SAVE THIS DATA as tmp.arff, then re-load tmp.arff, to work around the new Weka bug in tracking the correct class attribute, which is now **tmode**.

Q9. Run the most accurate classifier of Q5 to find whether its accuracy has changed. Record the classifier structure and its results here, and explain any difference or lack of difference from Q5.

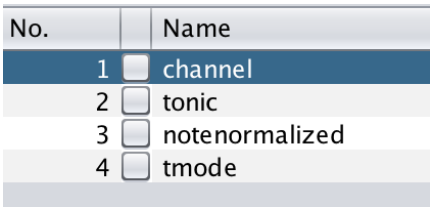
Q10. Temporarily remove attribute **movement** and re-run this classifier. Record its classifier structure and its accuracy results using the same classifier as Q9. What happens to accuracy, and how does the classifier structure change? Relate your answer back to COMMENT 1.

PREP STEP H: UNDO to restore **movement** (**movement** and **channel** are the attributes 1 and 2, and **tmode** is the last). Next, use AddExpression to create a new derived attribute **partitionedticks** according to this formula:

partitionedticks = (movement * 20000) + (channel * 2000) + tick

We are doing this because there are up to 512 notes per **movement-channel** pair, and the value in attribute **tick** re-starts at 0 at the start of each movement, as necessitated by the music synthesizers. We want to infer the **tmode** without using any other tagged attributes, so we need to separate MIDI data streams into 16 distinct virtual-temporal regions, 1 for each movement-channel pair. We want to avoid mingling MIDI musical data from different movements together and from different channels within a movement together. **Reorder** the attributes to put **partitionedticks** just after attribute **tick**, restoring **tmode** as the last attribute, and keeping the relative order of the other attributes the same. **SAVE** this dataset as **OneNoteAtATime.arff**. Re-load **OneNoteAtATime.arff** to gets past Weka's class attribute identifying bug.

Temporarily **Remove** all but the following 4 attributes. We are removing **movement**, **tick**, and **partitionedticks** because they serve as part of the *musical score* in relating non-note data to **tmode**.



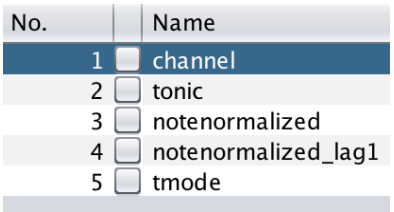
Q11. Run the classifier of Q10 and confirm to yourself that the Correctly Classified Instances and Kappa are within 10% of their values in Q10. Record these two values as part of your answer for Q11 (tag them with **BEFORE** in README.txt). Now copy **OneNoteAtATime.arff** into your project directory on acad and run **make timelag**, which runs this command line :

chmod +x ./maketimelag && ./maketimelag 10 python

This command creates the following 10 files.

TimeLag1.arff
TimeLag2.arff
TimeLag3.arff
TimeLag4.arff
TimeLag5.arff
TimeLag6.arff
TimeLag7.arff
TimeLag8.arff
TimeLag9.arff
TimeLag10.arff

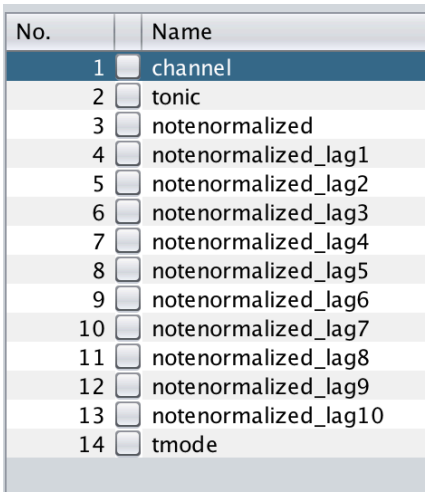
For each instance in OneNoteAtATime.arff, this command creates TimeLag1.arff by copying **notenormalized** for each instance's temporally preceding instance into the subsequent notenormalized_lag1 attribute, as temporally ordered by **partitionedticks**. Copy all 10 lag ARFF files back to your machine. Load **TimeLag1.arff** into Weka, **Reorder** to put **tmode** last, SAVE and RE-LOAD **TimeLag1.arff**. Remove all but the above 4 attributes + notenormalized_lag1 (**KEEP channel, tonic, notenormalized, notenormalized_lag1, and tmode**) and rerun the classifier of Q11, and record Correctly Classified Instances and Kappa values with **AFTER**. Did they improve, degrade, or stay the same compared to the BEFORE values? Explain why.



Classifier? **BEFORE**
Correctly Classified Instances N N %
Kappa statistic N
Classifier? **AFTER**
Correctly Classified Instances N N %

Kappa statistic N

PREP for Q12. The makefile also creates TimeLag2.arff through TimeLag10.arff with additional levels of time lagging, i.e., 2-time lags through 10-time lags, from their predecessors. Inspect TimeLag10.arff in Weka to see the lagged attributes. Open **TimeLag10.arff**, Reorder the attributes to put **tmode** last, SAVE and RE-LOAD **TimeLag10.arff**, and **Remove movement**, **tick**, and **partitionedticks** as before, so that we can try to infer tmode from channel-temporally associated note information. Keep these attributes:



No.	Name
1	channel
2	tonic
3	notenormalized
4	notenormalized_lag1
5	notenormalized_lag2
6	notenormalized_lag3
7	notenormalized_lag4
8	notenormalized_lag5
9	notenormalized_lag6
10	notenormalized_lag7
11	notenormalized_lag8
12	notenormalized_lag9
13	notenormalized_lag10
14	tmode

Q12. From among the classifiers that we have used in previous projects (or pick one that is new to you!), run a classifier that gives at least 98% Correctly Classified Instances on this TimeLag10.arff dataset. **Tell me the exact classifier you used, including any configuration parameter changes**. If you can get it to 99.5%, I will award 10 bonus points. In addition to identifying the classifier, give the following measures, and state why you think it classifies well. Here are two of mine.

```
Correctly Classified Instances        8220               98.9408 %
Incorrectly Classified Instances        88                1.0592 %
Kappa statistic                          0.9869
Mean absolute error                      0.0034
Root mean squared error                  0.0465
Relative absolute error                  1.6624 %
Root relative squared error             14.6324 %
=== Confusion Matrix ===

    a    b    c    d    e    f    g    h   <-- classified as
 2884    2    0    3    0    0    0    1 |    a = Ionian
    1  757    4    0    0    0    0    6 |    b = Mixolydian
    0    4 1301    0    0    0    2   15 |    c = Lydian
    1    0    0 1023    0    0    0    0 |    d = Aeolian
    0    0    0    0  256    0    0    0 |    e = Dorian
    0    0    0    0    0  512    0    0 |    f = Phrygian
    7    0    3    0    0    0  502    0 |    g = Locrian
    0   24   13    0    2    0    0  985 |    h = Chromatic

Correctly Classified Instances        8221               98.9528 %
Incorrectly Classified Instances        87                1.0472 %
Kappa statistic                          0.987
Mean absolute error                      0.0035
```

```
Root mean squared error                  0.0462
Relative absolute error                  1.7144 %
Root relative squared error             14.5592 %
=== Confusion Matrix ===
    a    b    c    d    e    f    g    h   <-- classified as
 2884    2    0    3    0    0    0    1 |    a = Ionian
    1  757    4    0    0    0    0    6 |    b = Mixolydian
    0    4 1302    0    0    0    1   15 |    c = Lydian
    1    0    0 1023    0    0    0    0 |    d = Aeolian
    0    0    0    0  256    0    0    0 |    e = Dorian
    0    0    0    0    0  512    0    0 |    f = Phrygian
    7    0    3    0    0    0  502    0 |    g = Locrian
    0   24   13    0    2    0    0  985 |    h = Chromatic
```

Prep for Q13. Load **spring2020notenorm.arff** into Weka and explore its data using the Preprocess tab and Edit window. It consists of only 16 instances, one for each movement-channel pair. The notenormalized0 through notenormalized11 attributes are simply **histogram counters** for exactly how many times that particular notenormalized value in the range [0,11] appears in that movement-channel. After running **StringToNominal**, **SAVE** over top of **spring2020notenorm.arff**. Then try running any classifier you like. It should be possible to hit 100% Correctly Classified Instances easily, but it is not. There are only 16 instances, and the movement-channel pair alone should be enough to correctly classify **tmode**. Furthermore, if I run Simple K-means clustering on Weka's Cluster tab (K-means is **not** a classifier – TRY IT!), I get exactly the right association of movement-channel pairs to tmode. Below are 4 rows from a 16-cluster K-means clustering of this dataset. The rows are CL = cluster number, MV = movement, CH = channel, and TM = tmode.

CL	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MV	2	0	2	3	2	1	3	1	1	0	3	2	0	3	1	0
CH	3	0	1	3	2	3	0	0	2	1	1	0	3	2	1	2
TM	Chromatic	Ionian	Locrian	Lydian	Chromatic	Phrygian	Ionian	Aeolian	Dorian	Ionian	Ionian	Ionian	Lydian	Mixolydian	Aeolian	Mixolydian

Clustered Instances	
0	1 (6%)
1	1 (6%)
2	1 (6%)
3	1 (6%)
4	1 (6%)
5	1 (6%)
6	1 (6%)
7	1 (6%)
8	1 (6%)
9	1 (6%)
10	1 (6%)
11	1 (6%)
12	1 (6%)
13	1 (6%)
14	1 (6%)
15	1 (6%)

```
__movementModeNames__ = [   # Jython tables ordered by movement, channel, that generate the notes
   ["Ionian", "Ionian", "Mixolydian", "Lydian"],
   ["Aeolian", "Aeolian", "Dorian", "Phrygian"],
   # Give the lead instrument Ionia in the dissonant section for added tension.
   ["Ionian", "Locrian", "Chromatic", "Chromatic"],
   ["Ionian", "Ionian", "Mixolydian", "Lydian"],
]   # reprise first movement in fourth movement
```

Q13. Why does classification work so poorly on this 16-element dataset, even though these attributes are 100% predictive for **tmode**?

Q14. Find some combination of classifier (I used one that we previously used) and Classify Test Options in Weka that give this result on these 16 instances. Tell me the classifier, the Test Option, and explain the reason for the improvement. (HINT: If you can't find it, investigate file BIGspring2020notenorm.arff, which simply repeats each instance in spring2020notenorm.arff 1024 times. **HOWEVER, your answer must be for spring2020notenorm.arff, NOT BIGspring2020notenorm.arff**.

Correctly Classified Instances	16	100 %
Incorrectly Classified Instances	0	0 %
Kappa statistic	1	
Mean absolute error	0	
Root mean squared error	0	
Relative absolute error	0 %	
Root relative squared error	0 %	
Total Number of Instances	16	

Q15. You get these points for a correctly SAVED and turned in **spring2020concert.arff** and **OneNoteAtATime.arff**.