CSC 558 Data Mining and Predictive Analytics II, Spring 2020

Dr. Dale E. Parson, Assignment 2, Classification of audio data samples from assignment 1 for predicting numeric white-noise amplification level for the signals' generators.[1] We will also investigate discretizing the white-noise target attribute (class) and other non-target attributes.

DUE By 11:59 PM on Wednesday March 4, 2020 via make turnitin on acad. The standard 10% per day deduction for late assignments applies.

There will be one in-class work session for this assignment. You may attend in person or on-line. I encourage attending the work session in person. Start early and come prepared to ask questions.

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/whitenoise558sp2020.problem.zip  whitenoise558sp2020.problem.zip
unzip  whitenoise558sp2020.problem.zip
cd  ./whitenoise558sp2020
```

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 and you took csc458, 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 **whitenoise558sp2020** directory:

README.txt	Your answers to Q1 through Q16 below go here, in the required format.
csc558wn10Ksp2020.arff	The handout ARFF file for assignment 2, **wn** means white noise.
makefile	Files needed to **make turnitin** to get your solution to me.
checkfiles.sh	
makelib	

ALL OF YOUR ANSWERS FOR Q1 through Q16 BELOW MUST GO INTO THE README.txt file supplied as part of assignment handout directory **whitenoise558sp2020**. You will lose an automatic 20% of the assignment if you do not adhere to this requirement.

1. Open **csc558wn10Ksp2020.arff** in Weka's Preprocess tab. This is the same dataset used for assignment 1, with AddExpression's derived attributes already in place, and with **tosc** and **tid** removed; tagged numeric attribute **tnoign**, which is the gain on the white-noise generator, is the class (a.k.a. target attribute) of assignment 2. Where assignment 1 had a nominal attribute as the class, this assignment has **tnoign** as a numeric class attribute.

Here are the attributes in csc558wn10Ksp2020.arff.

---

[1] See Assn1AudioOverview http://faculty.kutztown.edu/parson/spring2020/CSC558Audio1_2020.html and in-class discussion on the Zoom archives.

centroid **Raw** spectral centroid extracted from the audio .wav file.
rms **Raw** root-mean-squared measure of signal strength extracted from the audio .wav file.
roll25 **Raw** frequency where 25% of the energy rolls off, extracted from the audio .wav file.
roll50 **Raw** frequency where 50% of the energy rolls off, extracted from the audio .wav file.
roll75 **Raw** frequency where 75% of the energy rolls off, extracted from the audio .wav file.
amplbin1 through **amplbin19** Normalized amplitudes of 1st through 19th overtones of the fundamental. Filter **RemoveUseless** has removed **amplbin0** because of its constant value of 1.0.

Raw indicates an attribute that you normalized in assignment 1 to the reference fundamental frequency or amplitude. Attributes **centrfreq**, **roll25freq**, **roll50freq**, **roll75freq**, **nc**, **n25**, **n50**, **n75**, and **normrms** are Derived Attributes we created in assignment 1. Even though they are redundant with attributes from which they derive, they turn out to be useful for fine-tuning classifiers. We are keeping them for now. There are 34 attributes in the ARFF data of this assignment.

tnoign Target white noise signal gain passed to the audio generator in the range [0.0, 1.0]. Except for the five **tnoign**=0.0 samples that we will remove, the signal generator for this dataset generates **tnoign** in the range [0.1, 0.25). Note the Weka Preprocess statistics for **tnoign** below.

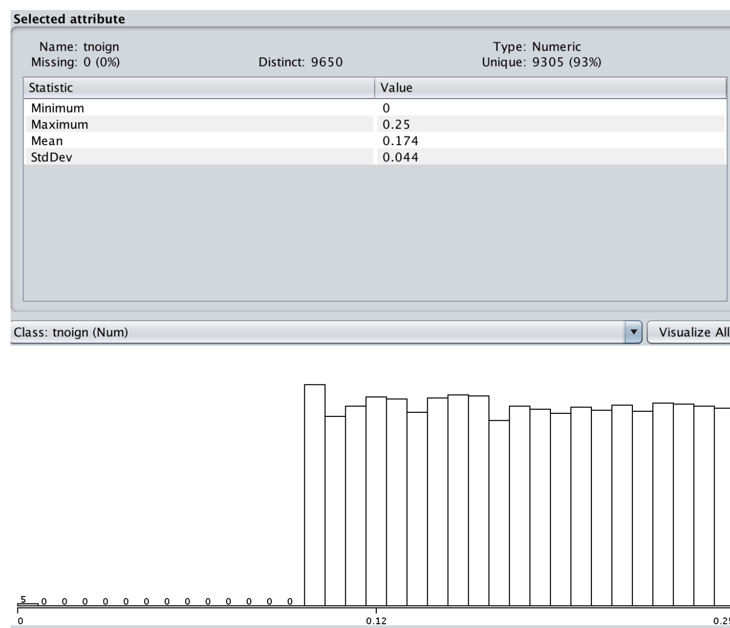


Figure 1: Class attribute tnoign in the handout dataset.

Since this assignment is about predicting white noise gain tagged as attribute **tnoign**, it is important to review the definition of white noise. As linked from Assn1AudioOverview, "White noise is a random signal having equal intensity at different frequencies, giving it a constant power spectral density…In discrete time, white noise is a discrete signal whose samples are regarded as a sequence of serially uncorrelated random variables with zero mean and finite variance.[2]" This white noise signal is distinct from the Sine, Triangle, Square, Sawtooth, and Pulse wave signals that were the focus of assignment 1, added into the composite signal with a random gain in the range [0.5, 0.75]. The dataset of assignments 1 and 2 add white noise with a random gain in the range [0.1, 0.25] to each signal-record in the dataset, with 5 exceptions that you will remove in step 2 below. Compare the frequency domain plot of the noiseless

[2] Wikipedia page on white noise https://en.wikipedia.org/wiki/White_noise , quotation checked for accuracy.

1000 Hz training sine wave of assignment 1[3] with the 1001 Hz sine wave with a **tnoign**= 0.139453694281 used as a noise-bearing training instance in assignment 1[4]. Both peak at about 1000 Hz, but the signal without white noise loses most of its strength after that. The signal with **tnoign**= 0.139453694281 white noise falls off considerably less, maintaining an almost constant signal strength all the way to the Nyquist frequency of 22050 Hz. The contribution of white noise at each frequency is random and seemingly small, but the net contribution of white noise is to add signal strength evenly across the frequency spectrum. We are trying to determine that contribution based strictly on audio data in the WAV files in this assignment. Important points to note include the following.

- Most of the frequency spectrum in the range [0, 22050] Hz lies above the non-noise signal generation (sine, triangle, etc.) fundamental frequency of [100, 2000] Hz. White noise spans the [0, 22050] Hz range. While the non-sine waves contribute harmonics that push measures such as centroid and the rolloff frequencies higher than the fundamental frequency, white noise pushes these measures even further up the frequency spectrum because it spans the [0, 22050] Hz range.

- White noise contributes additional power beyond the non-noise signals across the wave + white noise signal. Attribute **rms** is the measure of power across the time-varying, time-domain signal. Unlike the normalized fundamental frequency of **amplbin0**, which represents only the strongest frequency component of a signal, **rms** integrates signal strength across the frequency spectrum.

The five **tnoign**=0.0 samples illustrated in Figure 1 are outliers in relation to the other 10,000 instances.

2. Use Weka's Unsupervised -> Instance -> RemoveWithValues Preprocess filter to remove the five outlying instances with **tnoign**=0.0. Use the attributeIndex to select tnoign, use the splitPoint to select a value for this attribute above which OR below which instances will be discarded, using invertSelection if necessary to change the direction of the split. Successful application of RemoveWithValues to tnoign results in 10,000 instances with tnoign in the range [0.1, 0.25], which is the range of white noise gain for the signal generator. **SAVE THIS 10000-INSTANCE DATASET OVER TOP OF csc558wn10Ksp2020.arff, replacing the original csc558wn10Ksp2020.arff file**.

Q1: What is your exact RemoveWithValues command line from the top of Weka's Preprocess tab?

3. Run Classify -> Functions -> LinearRegression on this 10,000-instance dataset, for which you should get approximately the following results.

THIS IS THE MODEL:
tnoign =
0.5689 * centroid +
9.3174 * rms +
-0.155 * roll25 +
0.0823 * roll50 +
0.0776 * roll75 +
-0.0417 * amplbin1 +
-0.0911 * amplbin2 +
0.0149 * amplbin3 +
-0.0528 * amplbin4 +
0.0117 * amplbin5 +

[3] http://faculty.kutztown.edu/parson/spring2020/lazy1_SinOsc_1000_0.9_0.0_0.FREQ.png
[4] http://faculty.kutztown.edu/parson/spring2020/lazy1_SinOsc_1001_0.500235007566_0.139453694281_615143.FREQ.png

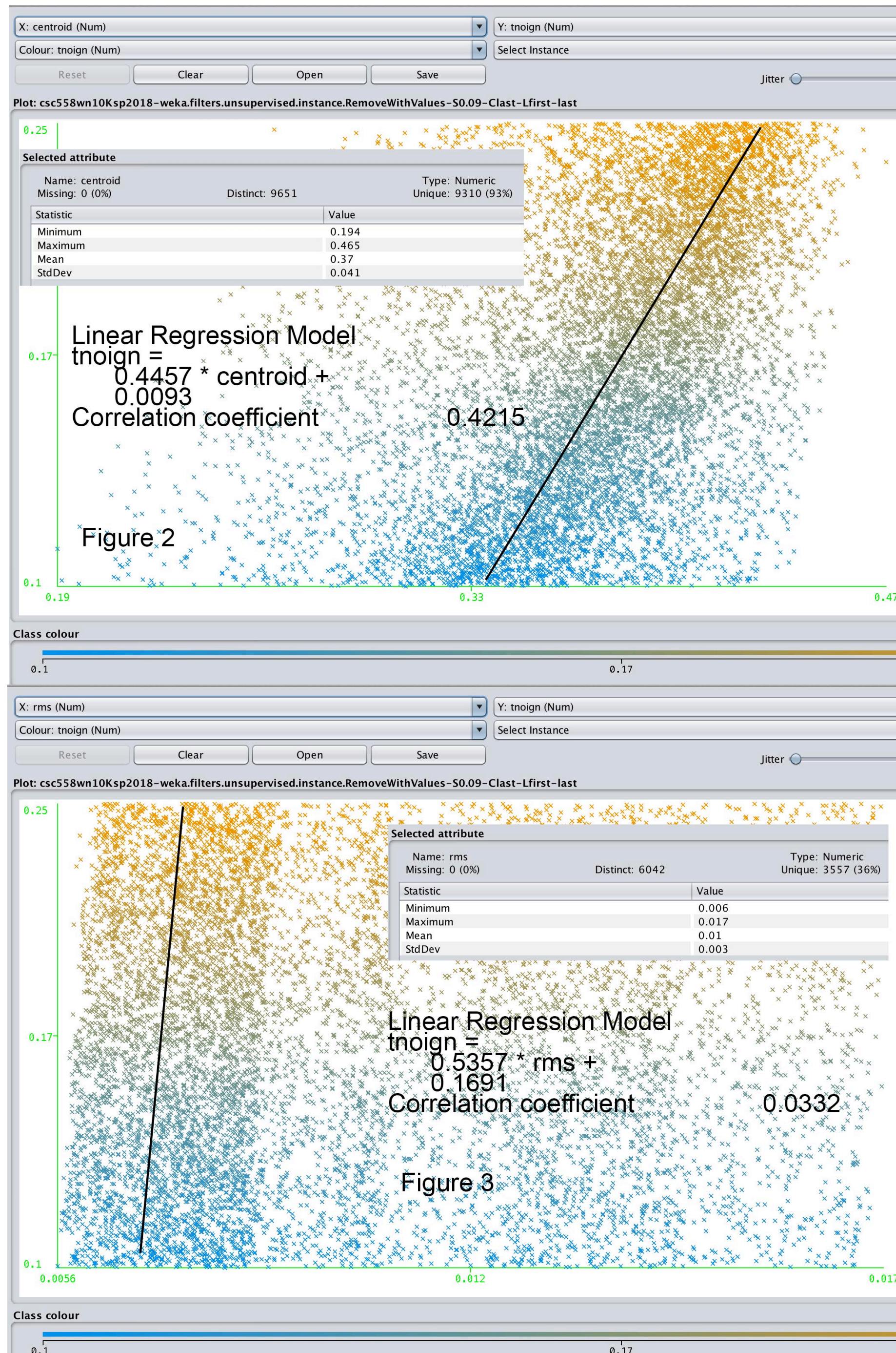
-0.0172 * amplbin6 +
 0.0692 * amplbin7 +
-0.0745 * amplbin8 +
 0.1115 * amplbin9 +
 0.0514 * amplbin10 +
 0.074 * amplbin11 +
 0.0254 * amplbin12 +
 0.0166 * amplbin13 +
 0.0525 * amplbin14 +
 0.0734 * amplbin16 +
 0.1784 * amplbin18 +
 0.1485 * amplbin19 +
 0   * centrfreq +
-0   * roll25freq +
 0   * roll50freq +
 0.0191 * n25 +
-0.0075 * n50 +
 0.0033 * n75 +
 0.4098 * normrms +
-0.4479

THESE ARE THE RESULTS:

Correlation coefficient	0.7964
Mean absolute error	0.0205
Root mean squared error	0.0263
Relative absolute error	54.5234 %
Root relative squared error	60.4761 %
Total Number of Instances	10000

I have **highlighted using bold-underline** the two strongest contributing attributes from LinearRegression's perspective, and I have **highlighted using bold** other contributors with an absolute value for the coefficient of at least 0.1. Note that negative coefficients are still important contributors in predicting a target numeric attribute; the sign means simply that they have a negative correlation.

The following two Figures show the correlation between pairs centroid<->tnoign and rms<-> tnoign in Weka graphical form, and via running LinearRegression using on these pairs of attributes, using one pair at a time.



Q2: Given the fact that **centroid** is more closely correlated with **tnoign** than **rms** is correlated with **tnoign**, illustrated by both the approximate slopes of these graphs (we will discuss in class) and their individual correlation coefficients, why does **rms** have a coefficient **9.3174** that is much higher than **centroid's** coefficient of **0.5688** in the complete LinearRegression model? The answer lies within Figures 2 & 3.

Q3: Why does signal **centroid** correlate positively with white noise gain **tnoign**?

Q4: Run Classify -> Trees -> M5P model tree on this 10,000-instance dataset, and record the Results (not the Model) for Q4. How do the M5P Results (correlation coefficient and error measures) compare with those of LinearRegression for this dataset?

Correlation coefficient ?
Mean absolute error ?
Root mean squared error ?
Relative absolute error ? %
Root relative squared error ? %
Total Number of Instances 10000

Q5. Run the instance-based (lazy) classifier IBk repeatedly with its default configuration parameters, increasing the KNN (number of nearest neighbors) parameter on each run until its performance begins to degrade, inspecting correlation coefficient for its peak. What value of KNN gives the most accurate result? Shows its Results.

KNN = N
Correlation coefficient n.n?
Mean absolute error n.n?
Root mean squared error n.n ?
Relative absolute error n.n? %
Root relative squared error n.n? %
Total Number of Instances 10000

Q6: Run the instance-based (lazy) classifier IBk one more time with its KNN as determined in Q5, then run it again after changing the nearest neighbor search algorithm from LinearNNSearch to KDTree with default parameters, and run it again using BallTree instead of KDTree. What change in behavior or performance do you notice compared to using the default LinearNNSearch nearest neighbor search algorithm?

In preparation for the next steps, run Preprocess filter Unsupervised -> Attribute -> Discretize on the target attribute **tnoign**, making sure to set the **ignoreClass** configuration parameter to **true**. Leave the useEqualFrequency parameter at false, leave bins at 10, and check **tnoign** before and after using the filter to make sure its distribution histograms look similar, and that it is not numeric after discretization. Do NOT discretize any numeric attributes other than **tnoign**. Check in the Preprocess tab to make sure no other attributes are discretized.

Q7: Now, run Preprocess filter Unsupervised -> Attribute -> Discretize on all remaining attributes with useEqualFrequency parameter at the default false and bins at 10. Inspect some of them in the Preprocess tab. Run classifiers rule **OneR**, tree **J48**, **BayesNet**, and instance (lazy) classifier **IBk** with the KNN parameter found in Q5 and nearest neighbor search algorithm of KDTree, and give their Results as outlined below, preceding each Result with the name of its classifier.

Correctly Classified Instances N N.N %
Incorrectly Classified Instances N N.N %
Kappa statistic N.N
Mean absolute error N.N
Root mean squared error N.N
Relative absolute error N.N %
Root relative squared error N.N %
Total Number of Instances 10000

Q8: Execute Preprocess -> Undo once, then check to make sure that only class **tnoign** is still Discretized. All other attributes except **tnoign** should be numeric. Now, run Preprocess filter **Supervised** -> Attribute -> Discretize on all remaining attributes (not **tnoign**). Inspect some of them in the Preprocess tab. Run classifiers rule **OneR**, tree **J48**, **BayesNet**, and instance (lazy) classifier **IBk** with the KNN parameter found in Q5 and nearest neighbor search algorithm of KDTree, and give their Results as in Q7, preceding each Result with the name of its classifier. Which classifiers became BETTER as measured by “Correctly Classified Instances” when compared with Q7, and which became WORSE. Just write BETTER or WORSE behind their classifier names.

Q9: Execute Preprocess -> Undo once, then check to make sure that only class **tnoign** is still Discretized. All other attributes except **tnoign** should be numeric. Run classifiers rule **OneR**, tree **J48**, **BayesNet**, and instance (lazy) classifier **IBk** with the KNN parameter found in Q5 and nearest neighbor search algorithm of KDTree, and give their Results as in Q8, preceding each Result with the name of its classifier. Which classifiers became BETTER as measured by “Correctly Classified Instances” when compared with Q8, and which became WORSE. Just write BETTER or WORSE behind their classifier names.

In general, increasing the resolution of the non-target attributes by keeping them numeric may help accuracy of prediction, since discretized non-target attributes only approximate the precision found in numeric non-target attributes. Unfortunately, precise numeric attributes may be harder for some classifiers to analyze. Bayesian analysis, for example, does its own discretization of numeric non-target attributes; this discretization may be better or worse than the Supervised Weka discretization filter at correlating non-target attributes to the target class.

Q10. Try using ensemble meta-classifier **Bagging**, using your most accurate classifier (in terms of Correctly Classified Instances) configuration from Q9 as its base classifier. What base classifier did you select, and does it improve performance over Q9 in terms of Correctly Classified Instances by more than 2% of 100% correct of the non-bagged Result of Q9? Show your Result as before. All attributes except the target **tnoign** should be numeric at this point.

Q11. Try using ensemble meta-classifier **AdaBoostM1**, using your most accurate classifier configuration form Q9 as its base classifier. What base classifier did you select, and does it improve performance over Q9 in terms of Correctly Classified Instances by more than 2% of 100% correct of the non-boosted Result of Q9? Show your Result as before. All attributes except the target **tnoign** should be numeric at this point.

Q12. What accounts for any performance improvements in terms of Correctly Classified Instances in Q10 and Q11 over Q9 results?

In preparation for the final steps, you must copy your modified, 10,000-instance csc558wn10Ksp2020.arff back into the project directory on acad and run **make train** to create a 100-instance training set and a 9900-instance test set as follows.

```
$ make train
'echo "making 100 training instances in csc558wnTrain100sp2020.arff"
making 100 training instances in csc558wnTrain100sp2020.arff
bash -c "echo '@relation csc558wnTrain100sp2020' > csc558wnTrain100sp2020.arff"
bash -c "grep @ csc558wn10Ksp2020.arff | grep -v  @relation >> csc558wnTrain100sp2020.arff"
bash -c "grep ^[0-9] csc558wn10Ksp2020.arff | head -100 >> csc558wnTrain100sp2020.arff"
echo "making 9900 test instances in csc558wnTest9900sp2020.arff"
making 9900 test instances in csc558wnTest9900sp2020.arff
bash -c "echo '@relation csc558wnTest9900sp2020' > csc558wnTest9900sp2020.arff"
```

```
bash -c "grep @ csc558wn10Ksp2020.arff | grep -v  @relation >> csc558wnTest9900sp2020.arff"
bash -c "grep ^[0-9] csc558wn10Ksp2020.arff | tail -9900 >> csc558wnTest9900sp2020.arff"
```

This places the first 100 instances of csc558wn10Ksp2020.arff into csc558wnTrain100sp2020.arff and the remaining 9900 instances into csc558wnTest9900sp2020.arff. You could do this by hand in a text editor, but using **make train** is a lot less time consuming and less error prone.

Q13. After bringing files csc558wnTrain100sp2020.arff and csc558wnTest9900sp2020.arff back onto your Weka machine, load csc558wnTrain100sp2020.arff in the Preprocess tab as the training set, and set csc558wnTest9900sp2020.arff to be the supplied test set in the Classify tab. Run M5P and record its Results here. How many rules (linear formulas) does M5P generate?

Next, load your file csc558wn10Ksp2020.arff into Weka, run the Unsupervised -> Instance -> Randomize filter on it one time, with the default seed of 42, to shuffle the order of the instances. **Save** this as csc558wn10Ksp2020.arff, **copy** it back into the project directory on acad, and run make train again. Now, the first 100 instances in csc558wnTrain100sp2020.arff and the remaining 9900 instances in csc558wnTest9900sp2020.arff have been randomized with respect to order. Bring these new training and test files onto your Weka machine.

Q14. After bringing randomized files csc558wnTrain100sp2020.arff and csc558wnTest9900sp2020.arff back onto your Weka machine, load csc558wnTrain100sp2020.arff in the Preprocess tab as the training set, and set csc558wnTest9900sp2020.arff to be the supplied test set in the Classify tab. Run M5P and record its Results here. How many rules (linear formulas) does M5P generate?

Q15. Before I removed **tosc** from your handout data, the instances were in the following order by **tosc** values. They remained in this order until you Randomized instance order. Note the five initial, 0-noise instances that you have deleted at the start of the current assignment in the above command output:

```
$ grep Osc csc558lazyraw10005sp2018.arff | cut -d, -f2 |uniq -c
      1 'PulseOsc'
      1 'SawOsc'
      1 'SinOsc'
      1 'SqrOsc'
      1 'TriOsc'
   2000 'SinOsc'
   2000 'TriOsc'
   2000 'PulseOsc'
   2000 'SawOsc'
   2000 'SqrOsc'
```

What accounts for the improvement in going from Q13 to Q14? Note that before Randomization, instances in file csc558wnTrain100sp2020.arff were in the same order as they are in the above csc558lazyraw10005sp2018.arff file.

Q16. Can you improve performance of M5P further by bagging it? Give Results showing improvement, or explain why this attempt at improvement fails. Make sure to use the randomized training and set files csc558wnTrain100sp2020.arff and csc558wnTest9900sp2020.arff, with M5P as the base classifier.

Each of Q1 through Q16 is worth 6% of the project, with the remaining 4% going for having a correctly Randomized csc558wn10Ksp2020.arff file in the project directory. Use **make turnitin** by the deadline to avoid a penalty.