Decision Tree Data Parsing: Missing & Numerical Data

Maximiliano Rios, Jessie Anker, and Seongho Lee
{riosm, ankerj, leese}@dickinson.edu

Abstract

All decision tree models require some kind of dynamic data parsing, especially when data can be represented in a wide variety of forms. Dr. John MacCormick, associate professor at Dickinson College, created a basic implementation of a Java-based decision tree. This application dynamically parses and processes nominal data. However, the majority of ARFF files include data forms such as nominal, numerical, real, and missing data, denoted as a question mark. In this paper, we investigate and implement various methodologies of missing and numerical data handling. Our focus for missing data handling will be *purification by skipping* and *purification by imputing*. For numerical data, our focus will be the bucketing and categorization of continuous numerical values.

Introduction

The purpose of this project is to build a tree that can successfully and accurately handle these different dataforms. Thus, the final decision tree model should be able to dynamically parse nominal, numerical, real, and missing datasets.

Two concepts that are relevant to decision trees are underfitting and overfitting. Underfitting is when the decision tree model is not complex enough to capture the relationship between the dataset features and the target variable (What is Underfitting, 2020). Underfitting is indicated through having a high error rate even on training data. Overfitting is when the model has become too accurate to the training set so that it performs poorly on the testing dataset (What is Underfitting, 2020). This is indicated by a very low training loss and a very high testing loss.

The starting point for this project was Dr. John MacCormick's Java-based decision tree application. His standard decision tree takes an .arff file as input and outputs a decision tree and its training error. The program is able to parse the .arff file and process the data by creating vectors out of the lines of data. The program then calculates the entropy and expected entropy of each attribute. The decision tree is created through recursively generating nodes by splitting on the attribute value with the lowest expected entropy (Russell & Norvig, 698). Hence, the decision tree is able to handle

Purification by imputing required the replacement of the values by an educated

nominal data with a discrete number of attribute values.

In order to handle numeric data and missing data, we must implement additional functionalities to Dr. John MacCormick's Java-based decision tree.

First, we researched ways to handle numeric data. Numerical data might not seem difficult to handle, but because it is continuous, data that has an infinite number of possibilities, it is difficult to make a definitive, efficient decision tree. Through brainstorming and research we found the method of bucketing data. At first we considered having a fixed bucket size. For example, bucket 1 would capture data from 0-10, bucket 2 handles data from 10-20, bucket 3 handles data from 20-30, and so on until the buckets cover the full range of data. However, this method would not be a viable solution. For instance if there is a very wide range of values within a large dataset data, our hardware would not have the computational power or space in order to achieve this. Furthermore, if there is a small range of values over a large dataset, the program will overfit and all of the values could be contained within one bucket. It is difficult to process data that has not already been processed. In order to bypass this problem we decided to create a fixed number of buckets instead of fixed bucket sizes. Fixed number of buckets is the method we explored in our experiment.

To better understand the implementation process for handling missing data, we reviewed three possible solutions: *purification by skipping*, *purification by imputing*, and adapting a learning algorithm to be robust to missing data. Assuming performance would increase in the order listed, we began implementation on the skipping method by simply removing all data lines that contained the appropriate missing value character ("?"). guess based on the surrounding values. In this implementation, we provided a utility function that would provide a "most common value" from the attribute value set that the missing data value is associated with.

Methodology

Purification by Skipping The implementation of method 1 resolves the missing data values by removing data lines that contains the missing data character. Verification should be made during the parsing of instance values to skip all lines that apply. While this method is simple to produce, we expected the result to be erroneous and lackluster. However, in the results section of the application, we recorded an unexpected performance.

Purification by Imputing The implementation of method 2 resolves the missing data values by replacing the values based off of a utility function. The utility function that we chose was a "most common value" algorithm that would record these types of values and update a reference list. We expected that results would vary depending on the prevalence of attribute set values, resulting in variations in success depending on the data sets.

Purifications Testing Testing was performed by calculating the difference in the error rates

of methods 1 and 2 on the same training and testing datasets. Datasets were broken into a 50/50, train-to-test ratio and formatted to allow reading into the application. For these tests, all missing data characters were represented by a question mark.

Bucketing This method handles numerical data values by creating buckets in which the numeric attribute values can be grouped. The bucketing method is based off of the range of data read into the program. This is through creating a list for each numeric attribute and then sort each list. One divides the list of values equally into the number of buckets by identifying the range of values each bucket will cover. Then replace each individual value with the corresponding bucket name that the value falls under. Essentially converting numerical data into nominal data.

Different Number of Bucket Testing We will test the effect of different bucket sizes on the accuracy of decision tree predictions. Note that the number of instances has to be greater than 125 when running this experiment because we are making 125 buckets.

1. Split the data 50% training data 50% testing data.
2. Create 3 buckets for the data.
3. Divide all numerical data equally into each one of those buckets.
4. Construct the decision tree based on the training data.
5. Finally, compute the error rate of the training and testing data and record the training and test error rate.
6. Repeat steps 2-5 for each of the following bucket size values (5, 10, 15, 20, 25, 50, 75, 100 125).

Hypothesis For the MAGIC dataset, we believe that the best bucketing results will be 125 because the size of the dataset is very large and this is the largest number of buckets. For the Breast.w dataset, we believe the best
bucketing results will be 10 for the same reasoning.

Results

Missing Data Implementation For testing the performance of the missing data implementation, we recorded the error rate for the tests under two data sets, the SOYBEAN.ARFF and AUTOS.ARFF datasets. These datasets appropriately contain nominal data and nominal & numerical data. In order to test these datasets, we split the given data by either a 50/50 or 90/10 training-to-testing ratio. If reviewing the two datasets, note that both range in the amount of data provided, and both must be split in order to provide training and testing data for the decision tree. This occurrence may have caused an unpredicted spike in the error rate for the output of the program. We predict that method 2 would perform better because we are keeping a wider variety of data intact. However, due to unpredictability of the results from the datasets in past experiments, we could not provide accurate numerical predictions for the difference between the two methods.

Outputs While we hypothesized that our outputs would correlate to how much data would remain intact inside of the instance sets, this was not the case. Method 1, *purification by skipping*, proved to come out ahead with method 2 having a higher error rate of more than 4% to 25% in some test cases. Another difference that we saw was

Test Sets	**Split**	**Error Rate**
Method 1	50/50	0.6259541984732825
Method 1	90/10	0.6470588235294118
Method 2	50/50	0.6929824561403509
Method 2	90/10	0.8142857142857143

Figure 1: SOYBEAN.ARFF test examples error rate results based on dataset splits.

Test Sets	**Split**	**Error Rate**
Method 1	50/50	0.8850574712643678
Method 1	90/10	1.0
Method 2	50/50	0.9223300970873787
Method 2	90/10	0.9130434782608695

Figure 2: AUTOS.ARFF test examples error rate results based on dataset splits.

Number of buckets	Training Error	Testing Error
3	0.011904761904761904	0.5273775216138329
5	0.002976190476190476	0.3659942363112392
10	0.0	0.27089337175792505
15	0.0	0.42363112391930835
20	0.0	0.3314121037463977
25	0.0	0.3631123919308357
50	0.0	0.2478386167146974
75	0.0	0.2478386167146974
100	0.0	0.2478386167146974
125	0.0	0.2478386167146974

Figure 3: BREAST.W.ARFF test example error rate results based on fixed bucket sizes.

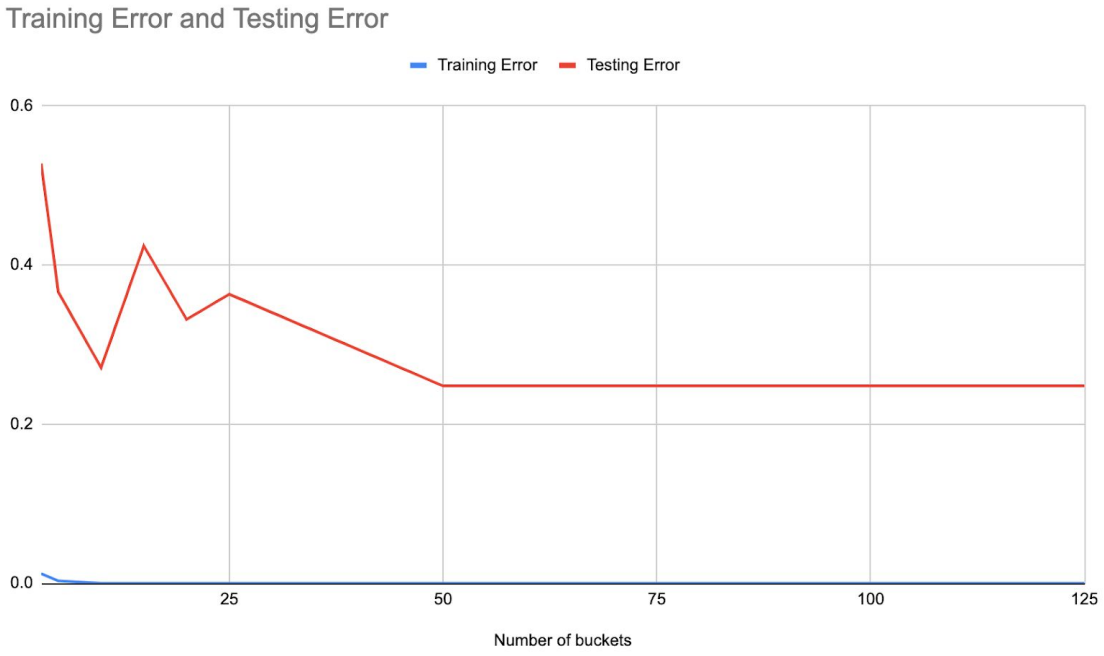

Figure 4: BREAST.W.ARFF displaying movement with increase in bucket sizes.

(*Continuation of output*) that the 90/10 ratio did not provide an improved performance over the 50/50 ratio. This may have been because of the size of the data sets or the reliability of the replaced missing values. In addition, the error rates were incredibly high, with the error rate of method 1 in one case being 1.0, never reaching an accurate conclusion on the branch.

Breast.w Dataset Description

- All of the attributes are continuous besides the class attribute
- All of the attributes are integers ranging from 1-10

For figures 3 and 4, the buckets are the most effective at 50 buckets. However, what was notable was that the range of values were from 1 to 10, leading to an assumption that 10 buckets, 1 for each value, would be the best choice for this dataset. A point to note in figure 7 is the drop in error rate at point 10, where a local minimum occurs that draws near to the error rate of 50 buckets.

Number of buckets	Training Error	Testing Error
3	0.11524710830704521	0.22860147213459517
5	0.026182965299684544	0.27392218717139855
10	0.0014721345951629863	0.26393270241850686
15	2.103049421661409E-4	0.2719242902208202
20	0.0	0.27917981072555204
25	0.0	0.2817034700315457
50	0.0	0.2929547844374343
75	0.0	0.3173501577287066
100	0.0	0.31514195583596216
125	0.0	0.3117770767613039

Figure 5: MAGIC.ARFF test example error rate results based on fixed bucket sizes.

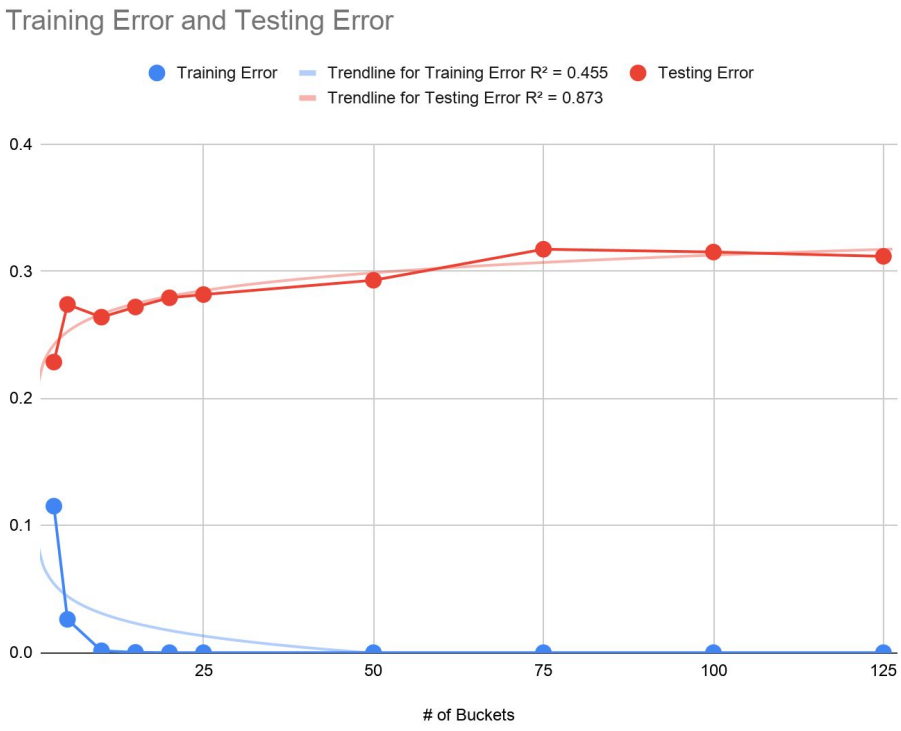

Figure 6: MAGIC.ARFF displaying movement with increase in bucket sizes.

MAGIC Dataset Description

- Number of Instances: 19020
- Number of Attributes: 11 (including the class)
- Missing Attribute Values: None
- Class Distribution:
- g = gamma (signal): 12332
- h = hadron (background): 6688
- All of the Attributes were continuous besides the class attribute

For figures 5 and 6, the training error steadily decreases as the number of buckets increases. However, the testing error increases as the number of buckets increases. It seems as though the most optimal number of buckets is at 10 buckets for the MAGIC dataset.

Conclusion

Critical Analysis The results of the purification by skipping and imputing tests proved unexpected due to the size of our data and the effectiveness of the guessing algorithm, in this case, "most common value". As seen in figures 1 and 2, method 1 performed much better than method 2, in all but 1 case. We believe that the difference in these tests are a result of small datasets and improper division of training and test values. We predict that as the dataset grows and missing data values become more prevalent, the performance of method 2 will overtake method 1.

In addition, the error rates for the tests were found to be high. As seen in figure 2, method 1 performs with a 100% fail rate on the AUTOS.ARFF dataset. We believe this is due to the datasets rather than a fault in the code. However, reviews and revisions are being made to confirm this prediction. As of now, the imputing process runs at $O(m * n^2)$, with m representing the number of lines in the dataset, and n representing the number of attributes in the dataset.

The results of the bucket test methodology on the breast.w dataset (breast cancer research) indicated that 50 buckets were the most optimal. This contradicts our hypothesis because we predicted that 10 buckets would lead to the least amount of testing and training error. This is because there was a discrete amount of data, the values were integers ranging from 1-10. This dataset was more of a test because we knew the expected outcome. It did in a way confirm our prediction because at number of buckets = 10 there was a local minimum for testing error. Further, research needs to be done in order to fully understand why the decision tree performed better at 50 buckets than 10 buckets.

The results of the bucket test methodology for the MAGIC dataset indicated that 10 buckets were the most optimal. However, this did not follow our predictions which was the greatest number of buckets, 125, would bring the least amount of testing and training error. At 125 buckets the error rate did decrease, but the testing error was less when the number of buckets equalled 10. This suggests that there is a possibility of a global minimum testing error after 125 buckets. We were not able to successfully test for this result because our program was not efficient enough to run 125+

buckets in reasonable time. This could be due to merge sorting each list of attribute values and also when assigning buckets to the value, the algorithm loops through the number of buckets and tests if the value is within that bucket range. Thus, the algorithm is $O(n_i * nlgn + n_i * i * b) \sim O(n_i * i * b)$ where n_i is the number of numeric attributes, n is the number of values at each attribute, i is the number of instances, and b is the number of buckets.

Errors and Struggles Gathering testable data that was easily formatted in order to successfully parse the data was a lot more difficult than anticipated. Browsing possible datasets, reformatting data and importing into eclipse all took a lot of time.

While access to testable data was easily accessible, measuring the reliability and size of these datasets proved difficult. Because our tests relied so heavily on the effectiveness on the division of training and test data, we would need to review the data more carefully and follow appropriate division methodologies.

Making the number of buckets flexible also took some time and thought. It was not efficient to comment out the sections of code when trying to create a different number of buckets. Instead we added a global variable as a reference to the number of buckets and based our code off of that. Hence, our code was more flexible and adaptable to further testing and improvement.

Resources

De Pontes Pereira, Renato. "ARFF Data Sets." *Github*, 12 Dec. 2012, github.com/renatopp/arff-datasets/tree/master/classification.

Fox, Emily. "Handling Missing Data." University of Washington, 2018, courses.cs.washington.edu/courses/cse416/18sp/slides/S6_missing-data-annotated.pdf.

Russell, S. J., & Norvig, P. (2010). Artificial intelligence: A modern approach (3rd ed.). Upper Saddle River, NJ: Prentice Hall.

Taylor, Jonathan, and Susan Holmes. "Statistics 202: Data Mining ." Stanford University, 19 Oct. 2012, statweb.stanford.edu/~jtaylo/courses/stats202/restricted/notes/trees.pdf.

What is Underfitting: DataRobot Artificial Intelligence Wiki. (n.d.). Retrieved December 02, 2020, datarobot.com/wiki/underfitting.