Post-Training Data Addition to Decision Trees

Jeremy Swerdlow
Earlham College
Richmond, Indiana
jjswerd14@earlham.edu

April 25, 2018

Abstract

Decision trees are a widely used machine learning technique for classification problems.[13] Current classifiers in real world applications create new decision trees on a regular basis, instead of modifying existing trees. This process is time consuming, adversely impacting the overall performance of the classifiers.

This study focuses on exploring different ways to minimize the time taken for the decision tree to accommodate new training sets. We propose a two-phased algorithm, which modifies an existing decision tree based on the newly available data set. In the first phase, each record from the training set is compared against the existing decision tree, in order to decide whether a new child node should be created for this node or the value of a node in the tree should be updated. In the next phase, the algorithm adjusts the weights of each node based on newly added data to represent the dataset as a whole.

By using this method instead of rebuilding the entire tree, we still gain the benefit of this new data, but reduce the time cost to adding it. While the study is not yet complete, we expect the outcome to be an increase in the accuracy of the model as compared to the original, while experiencing a decrease in the time to make use of the new data.

1 Introduction

Machine learning, a field of computer science, is currently experiencing a growth in popularity. Jason Brownlee reports in his article *Machine Learning is Popular Right Now* that an example of this growth is seen in "Stanford's online machine learning course that had hundreds of thousands of people showing expressions of interest in the first year."[18] Companies, individuals, and researchers are developing products and projects that use machine learning. Machine learning algorithms generally solve three categories of problems: supervised, unsupervised, or reinforcement learning.[21] Of these, the area of focus for this research is in supervised learning.

Supervised learning models handle several types of problems, but one of the most common is classification. When given a set of data with labels, these models can classify unlabeled data into the same labels as the training set. While many supervised learning models can handle classification, such as neural networks and support vector machines, this research focuses on the decision tree.

Decision trees are an effective machine learning model for classification, that provide many benefits. The concept behind how they are created is easily understandable, and they can be trained relatively quickly. However, as with all machine learning algorithms, they rely on a corpus of data for us. This becomes an issue when using Big Data, which "can be described in terms of data management challenges that – due to increasing volume, velocity and variety of data – cannot be solved with traditional databases."[35] These collections range in size, with some exceeding terabytes of data.[33] When using such datasets, training decision tree models is time

consuming.

Decision tree algorithms follow a system process similar to the one presented in Figure 1. It consists of several individual processes, which can be split into three major phases: data gathering/preparation, training, and testing. When all of these have been completed, the model is ready for use in production.

However, this system does not allow for data collected after the completion of this process to be utilized. After the initial phase of gathering and preparation, the model is trained on the records, tested, and completed. It does not implement a method to update the tree. Instead, whenever the decision tree should be updated, a new model is trained from the combination of the old data and the new. As the training process takes a significant amount of time and resources, it is often pressed to delay recreating the model until a significant amount of new data is collected, or a certain amount of time has passed.[28]

It is on this aspect of the system this research focuses. We propose a method to add data to a preexisting tree. The expected result is an increase in the model's performance, while decreasing the time required to gain any insights available in the new data. To summarize, the presented method will allow for new data to be added to the model, without requiring the complete training process to reoccur.

2 Background

2.1 Machine Learning

The field of machine learning is a subset of artificial intelligence (AI), which has been a field of computer science since close to the creation of modern computers. In 1950, Alan Turing invented the Turing Test to determine whether a computer was intelligent.[5] The idea a computer could learn, grow, and adapt to a given situation has existed as long as the field of artificial intelligence, though the definition of an "intelligent" computer has changed over time. As computers become more complex, allowing them to solve more difficult problems, the criteria for "intelligent" has become more difficult. However, recent changes in the ways the problems were considered and in computational ability have contributed to giant leaps forward in the field, pushing the boundaries of intelligence further. In the 1990s, machine learning became a focal point of AI. This meant a shift from a focus on creating rules for computers to follow, to a focus on learning from past examples. Machine learning focuses on examining datasets and identifying patterns within them.

2.2 Decision Tree

As the focus of this research is the decision tree, it is important to discuss the history of the structure, as well as current versions in use. As mentioned in the introduction, decision trees are a type of supervised learning model. This requires that the data being used to train the model is labeled into different categories which the model will predict unlabeled data belong to. The structure of the decision tree is based off of the branches of a biological tree, such that at each split, the set of data is divided by some feature to maximize the ability to classify results by the labels associated with each training sample. In one of the centerpieces to the teaching of AI, Stuart J. Russell and Peter Norvig present pseudocode for a generic algorithm of how a decision tree can be built. This algorithm, shown in Fig. 2, works recursively, splitting the data on a specific feature before further splitting those subsets on other features.[4]

This generic algorithm has been extended and developed further in several ways, both to improve the basic design, and to create programs to construct and use decision trees.

2.3 Implementations of the Algorithm

Decision trees have been implemented in multiple programming languages, with different variations available. For the purposes of this research, Python was used for its relative ease of programming and the wide variety of available packages to help support the work.

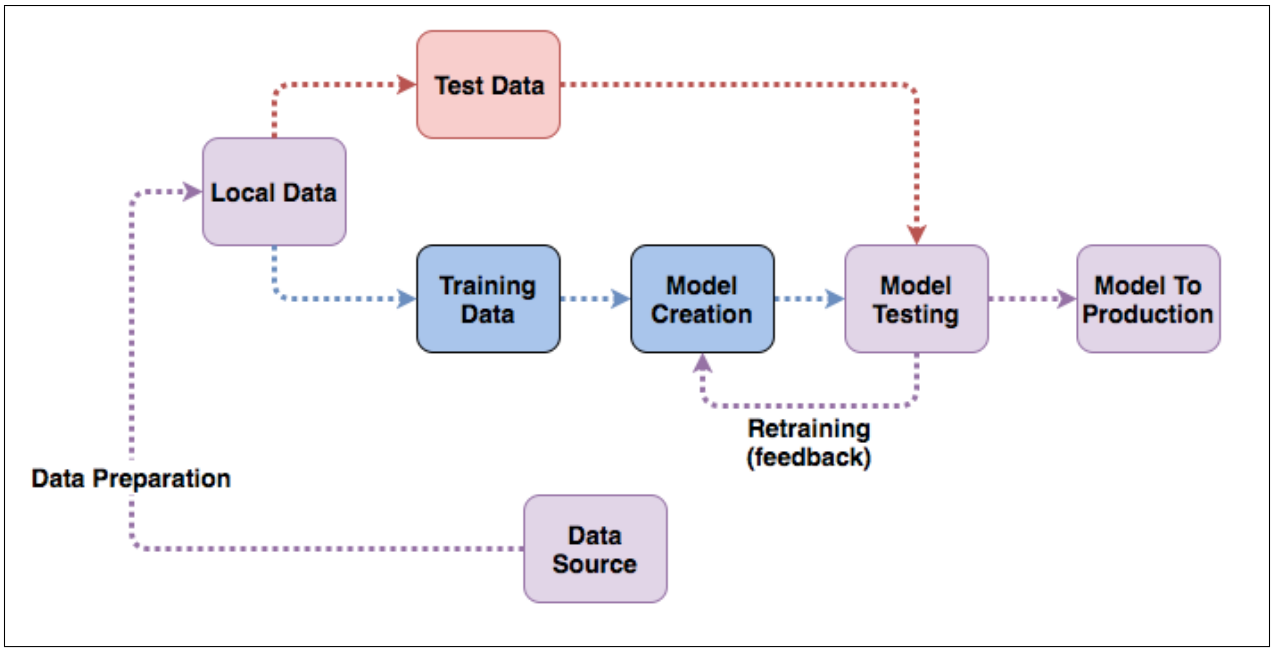

Figure 1: System diagram for the creation of supervised machine learning models

2.4 Improvements to the Algorithm

In addition to creating workable versions of the algorithm, developers have improved the ability and speed of creation for decision trees in various ways. One of the first improvements was the ID3 Algorithm. Developed by Ross Quinlan. It has been implemented in several languages. As Peng et al. describe in *An Implementation of ID3 Decision Tree Learning Algorithm*, it employs "a top-down, greedy search through the given sets to test each attribute at every tree node" in an effort to minimize the depth of the tree.[2] By trying to make as small a tree as possible, it becomes more general, and faster to predict labels, due to the fewer levels of node which must be traversed.

However, these improvements do not focus on what to do with the model after it has been created. It considers each tree to be discrete, and separate from other ones that will be created or have been created using similar data, or to classify the same set of labels.

This research challenges this idea, and considers trees as related when solving the same problem using similar or overlapping sets of data. Thus, the focus of this research is on updating preexisting trees, instead of creating new ones.

3 Methodology

With this focus of this project on updating preexisting trees, the work is broken down into developing, implementing, and testing an algorithm to solve this problem.

3.1 Algorithm Development

3.1.1 New System Flow

The first step to developing an algorithm was identifying where new data should come from, and how to bring it into the model. For example, if the decision tree is being used to classify mushrooms into categories of poisonous and edible, each time a new sample of a mushroom becomes available, the related data needs to be added. Due to this structure, there are now three types of data involved: those used to train the original decision tree, those which need to be added to a preexisting tree, and finally the collection used to test how successful the model is. For the purposes of this research, these are referred to as training, post-training, and testing data, respectively. In order to be able to use the post-training data on the preexisting tree, it must contain the same features as the data itself. To prepare the new data properly, the same

```
function DECISION-TREE-LEARNING(examples, attributes, parent_examples) returns
a tree

  if examples is empty then return PLURALITY-VALUE(parent_examples)
  else if all examples have the same classification then return the classification
  else if attributes is empty then return PLURALITY-VALUE(examples)
  else
      A ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
      tree ← a new decision tree with root test A
      for each value v_k of A do
          exs ← {e : e ∈ examples and e.A = v_k}
          subtree ← DECISION-TREE-LEARNING(exs, attributes − A, examples)
          add a branch to tree with label (A = v_k) and subtree subtree
      return tree
```

Figure 2: Pseudocode for the creation of a decision tree (*Artificial Intelligence*, 702).

process used on the data for the original tree can be reused. Given this new data set, the idea of adding data to a tree, instead of creating a tree from scratch, thus requires a new system diagram from the one displayed in Fig. 1. The new system diagram is shown in Fig. 3. The difference between the two is data now travels from the source directly to the model in production.

By streaming the post-training data from the source to the production model, while using the same preparation as applied to the training data, we can bypass the timely cost of creating the model, but still make use of the data. This way, the expensive process is only completed once, and then our new algorithm is used. With this system diagram created, we could move onto how the algorithm would edit the tree to add new data.

3.1.2 Algorithm Creation

The developed algorithm works similarly to the one presented by Russell and Norvig for creating decision trees. Its pseudocode is found in Fig. 4. Starting from the root node, the algorithm recursively moves through the tree, checking at each node for several cases, the first of which is if the node is a leaf node. In this case, the algorithm attempts to further split the data if possible, and if not, update the weight. For example, if there was a decision tree to classify animals, we may encounter a situation where there are similar examples all with the same class, such as dog. However, if a new sample is added with the class of cat, it may have similar values for its features to the class of dog. In this case, it may be that there is some feature we have not yet considered which may allow us to further distinguish between cat and dog, such as size.

In the case where a node is not a leaf node, there are two possible other occurrences. First, the algorithm checks to see if one of the branches to a child node matches the value of the feature of the data sample. If it does, it recursively calls to that node, again checking if it is a leaf and updating the leaf based on the result if it is. Going back to the example above, if we have a feature based on the number of legs, we may have values of 0, 2, and 4. If we have a new sample to add for a dog with 4 legs, when we reach this node in the tree, we would follow the branch which had 4 as its value.

The second possibility is that no child node matches the attribute. In this case, a new leaf node is created, with the label of the new data point being the label of the leaf. Again in the case of our decision tree for animals, imagine new data which includes a record for class of spider with 8 legs. When we reach the same node where

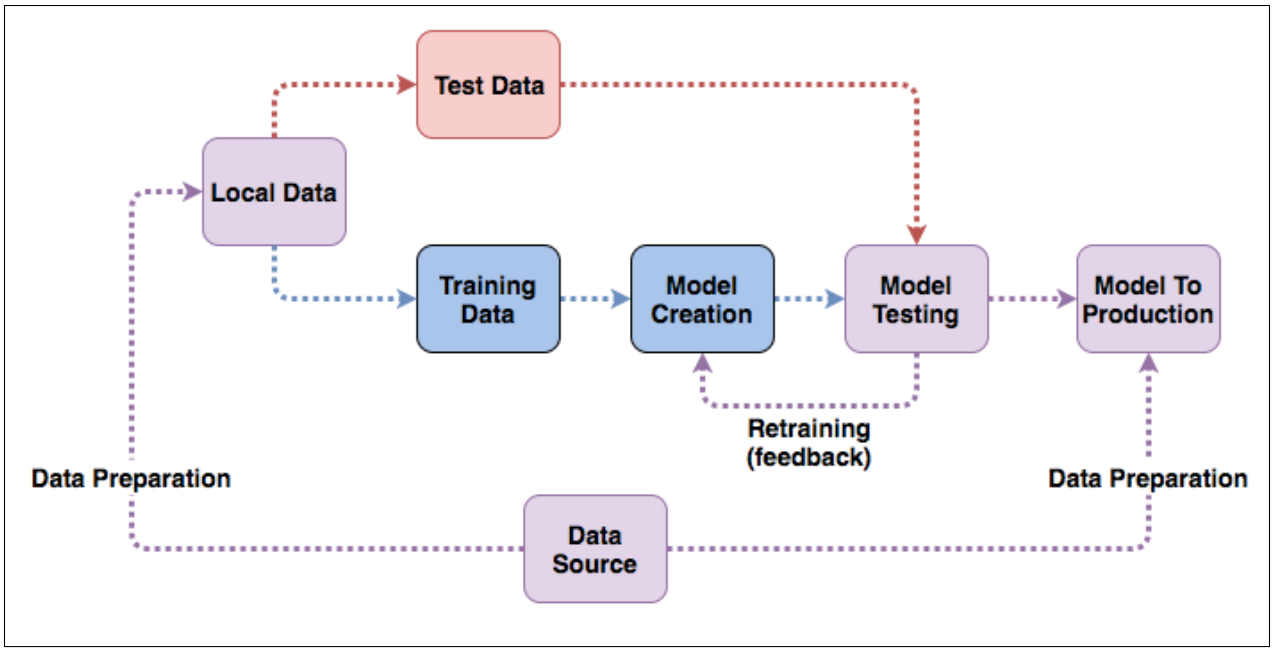

Figure 3: System diagram for the creation of decision trees with post-training data addition

the split is to 0, 2, and 4 legs, the algorithm creates a new leaf node, with the class spider as the value of the leaf, and 8 as the value of the branch to reach said leaf.

The process described above is repeated for each new point of data in our post-training dataset. While this means that there is some significance to the order in which data is encountered (there are possibly more children to compare against if nodes with attributes that are not included in the original tree are added first), once all samples have been added, the tree will be equivalent.

By checking for leaf nodes, and then covering the possibilities of the child existing or not, all possible outcomes from comparing a data sample versus the children of any node are accounted for.

With the algorithm developed, the next step was to implement a copy of the algorithm from the pseudocode for testing.

3.2 Algorithm Implementation

3.2.1 Python Implementation

From the pseudocode, an implementation of this algorithm was constructed. For the purposes of this research, Python 3 was chosen to create the implementation. Python 3 has an extensive collection of packages and environments which make the development and testing of code fast and easy. For the purposes of development and testing, the Anaconda Distribution was used to manage the resources.[26] Anaconda provides a simple interface through the command line for the installation and management of packages in Python. Of these, perhaps the most important to this research was the Jupyter Notebook.[25] Jupyter is an open-source project designed so developers can program interactively. This allows for users to visualize datasets, graph trees, and write documentation in the same location.

Several other packages were also used for development. While most of these come standard with Python 3, such as math, datetime, and copy, others played an important role.

For ease of access to the data, the Pandas package was used. It allows for data to be stored in DataFrames, similar to a SQL table. Their functionality allows for comparisons to be performed efficiently, providing an effective way to store data at each node of the tree. These DataFrames are the foundation of how data is stored in the implementation of the decision tree algorithm.

```
function ADD-ROW-TO-DECISION-TREE(examples, tree) returns a tree

    goal = tree.goal
    dec = tree.decision
    for each value row of examples do
        add row to tree.data
        if row[dec] in tree.branches then
            if tree.branches[row[dec]] is a leaf then
                if row[goal] is equal to leaf value then return tree
                else
                    exs ←{e : e ∈ tree.data and e[dec] = row[dec]}
                    tree.branches[row[dec]] = DECISION-TREE-
                LEARNING(exs, tree.remaining_attributes, tree.data)
            else
                tree.branches[row[dec]] = ADD-ROW-TO-DECISION-
TREE(row, tree.branches[row[dec]])
        else
            tree.branches[row[dec]] = row[goal]
    return tree
```

Figure 4: Pseudocode for adding new data to a preexisting decision tree

To create diagrams of the decision trees, the Python graphviz package was used. This package interfaces between Graphviz, a graph visualization software, and Python to create graphs, such as those in section 8.3 of Appendix A. The provides an API to create such diagrams, and can be displayed in-line with Jupyter notebooks.

Due to the relative ease of installing these packages through Anaconda, and the common practice of using Anaconda for machine learning and data science related projects, this group of technologies allowed for a quick version of the models to be created, visualized, and tested against.

4 Results

Following the completion of implementing the algorithm and decision tree method in Python, several tests were designed. The first set of tests examined whether or not the algorithm was successful at adding new data to the tree. These tests proved new data can successfully be added to the model, causing nodes which already existed to be updated, and new ones to be created. For these tests, the Pets dataset discussed in section 4.2.1 was used, as its small size improved visualization and speed. Following these tests to show the decision tree now contained information gained from the addition of new data, the next step was to test the model's performance.

For performance testing, two factors were important to examine: how well the model labeled data after new data was added in comparison to general practice, and what improvement in terms of time the new algorithm provides, in contrast to creating a new model.

The tests were performed on a 2014 MacBook

Air with 1.4 GHz Intel Core i5 processor and 8 GB 1600 MHz DDR3 memory. While the use of this specific computer will impact timing results, it is expected that these tests remain consistent across computers.

4.1 Performance Metrics

To test the performance of the model, a standard set of metrics to examine if the model was predicting accurately was needed. For the purpose of testing binary classifiers, researchers have previously created a series of metrics showing how well a model does to label samples as positive or negative. Four of these were used to test the models, and are detailed below. While each test individually has its faults, together the four provide a more complete set of measurements by which to test how well a classifier is performing. To best understand how these metrics work, it is important to first understand a specific characteristic of binary classification.

Within binary classification problems, you can consider one of the classes to be positive, and the other to be negative. From this, you can then create four groups of how the model predicted results: true positive, meaning that the model correctly predicted it as positive; true negative, the same as true positive but with a classification of negative; false positive, meaning the sample was predicted to be positive but was actually negative; and false negative, the same as false positive, but with an incorrect classification of negative. Using these four outcomes to labeling a sample, we can see how well the model does at predicting each individual class, as well as its overall success.

Additionally, it was important to see the time cost of adding new data with our algorithm, as the goal of this research was to avoid the time cost of training a new decision from the complete set of data.

4.1.1 Time

To measure how well the model performed in terms of time, Python's datetime package was used. On the computer used, this package provides time resolution to the microsecond, allowing for distinction between time costs to a granular level.

However, measuring at this level does raise some questions of computer inconsistency. To combat any issues of difference separate from the program itself, each test was run multiple times, and the results were averaged.

4.1.2 Accuracy

The first of the four performance metrics used to test the decision tree was accuracy. Within binary classification problems, accuracy is defined as the total number of correctly classified test samples, divided by the total number of test samples. To get this number, we refer again to the labels of true positive (TP), true negative (TN), false positive (FP), and false negative (FN). With these labels, accuracy is defined as below.

$$A = \frac{TP + TN}{TP + TN + FP + FN}$$

This measurement provides the user with the percentage of samples that the model correctly labeled.

4.1.3 Recall

The second metric used is recall. It measures how many of the test samples that were actually positive were correctly labeled as positive. Given previous definitions of TP, TN, FP, and FN, the following equation calculates recall:

$$R = \frac{TP}{TP + FN}$$

This metric is commonly used with precision, allowing for the creation of precision-recall curves.[24] A precision-recall curve compares how well these statistics do as different features of the model are changed.

4.1.4 Precision

Precision is related to recall, as it is another metric which considers the portion of true positives out of a larger collection. However, it differs from

recall, as it examines how many of the samples predicted as positive were actually positive. As such, the equation for calculating precision is as follows.

$$P = \frac{TP}{TP + FP}$$

4.1.5 F1-Score

The final metric used for testing the performance of the models is F1-score. It is more similar to accuracy, in that it determines the overall performance of the model. However, it does better to avoid issues in measuring performance that arise from biased data and other problems in the model or data than accuracy does.[32] It is defined as "the weighted average of Precision and Recall."[7] To calculate it, the below equation is used, where R and P are recall and precision as defined above.

$$F = 2 * \frac{R * P}{R + P}$$

These four metrics each provide different measurements of the model's success when used to classify the test samples. Collectively, they give the user a more complete understanding of where the model is succeeding or failing when predicting labels for the test data.

4.2 Datasets

As mentioned in the introduction, machine learning "is a branch of artificial intelligence based on the idea that systems can learn from data."[19] As such, machine learning models require datasets to be created. For the purpose of this research, a collection of smaller datasets were used. Each will be discussed briefly in the subsequent sections. In total, four datasets were gathered, three of which came from Kaggle, an online data management, storage, and distribution service.[11] Copies of each of these datasets can be found in the GitHub repository for this research, linked in Appendix A.

4.2.1 Pets

The first dataset was provided by David Barbella, PhD, Associate Professor of Computer Science at Earlham College. It contains 16 samples, with 4 features describing animals, and a label 'iscat'. Its small size and limited features made it an ideal pilot dataset for testing the algorithm, though this benefit made its results from the performance metrics vary significantly depending on which samples were placed into each of the sets via the random split.

4.2.2 Mushroom

The mushroom dataset contains 8124 total samples of mushrooms, with 22 features and a classification of whether each sample is poisonous or edible. Features include odor, habitat, and various attributes of the stems and caps. While this dataset was collected from Kaggle, it is originally from the UCI Machine Learning repository.

4.2.3 Kaggle 5-Day Data Challenge

As one of the leading sources of data for machine learning, Kaggle regularly hosts events in which people can participate globally. The final two datasets come from the sign-up survey for one such event.

Both surveys ask a series of questions related to the participant's background in statistics and machine learning, allowing for a dataset with features for each of their responses to be created. The surveys also ask whether the participant prefers dogs or cats, which is used as the label.

The combination of these four datasets provides variety in number of records, number of features, and strength of the relation between the features and the label. Together, they provide a variety of data from which the decision tree can be created and the proposed method can be tested.

4.3 Testing Design

From this collection of data and metrics, tests were designed. As the proposed method is intended to replace the process of remaking a decision tree from the combined corpus of the original data and newly collected samples, the tests compares the results of classifying samples done

by three different trees: the initial tree, the updated one, and the remade one. To create these trees, the data was separated into the three groups specified earlier: training, post-training, and testing. The split was done fractionally, with each group receiving a randomly selected portion equal to 0.6, 0.2, and 0.2 of the total data respectively. Additionally, the tests were run several times in an effort to reduce bias in the results. The values of Tables 1-12 of Appendix A show the results of these tests.

In each iteration of the test, the time required to create or update each of the three trees was recorded, using Python's datetime.now method. After the phase of adding the post-training data to the tree via the different methods was completed, the tree and the test data were used to calculate the metrics presented in 4.1.2-4.1.5.

To best compare how each version of the decision tree performed, it was necessary to use equivalent splits of the data for each model. This way, each model has the same data to use when training, and the same testing samples to label. This allows for any difference in score on a metric to be due to the design of the tree.

Additionally, to examine the impact of updated nodes versus adding new ones to the decision tree, three types of tests were run. In the first, data was split such that it was known a new node would need to be added to the tree. The second group of tests made sure no new nodes were added, but weights were still updated. In both of the above test styles, the data left after removing records pertinent to examining this feature were randomly split into the same fractions of 0.6, 0.2, and 0.2, with the same goal of eliminating bias.

Finally, the third set of tests were run with random splits of training, post-training, and testing data, because in real-world usage, the user would not necessarily know if a new node would be needed or not.

5 Discussion

The results from the tests as described above can be found in Tables 1-12 of Appendix A. In summary, when splitting the data into training, post-training, and testing data by batch, the updated tree generally out performs the initial tree, but does not do as well as the remade one in terms of performance metrics. Additionally, when handling data in batches, the time required to update the tree is substantially more than remaking it. In the case of the 5-Day Data Challenge 2nd dataset, updating the tree took on average nearly three times as long as remaking the tree. However, when the number of samples being added is smaller, updating the tree runs more quickly than remaking the tree, as can be seen from the results of these tests on the pets dataset. On average, updating the pets decision tree took only a third the time of remaking the tree. The increase of time cost based on the number of data is most likely due to the repeated recursive traversing of the tree for each new sample, instead of handling all that match a specific value for a feature at once.

These results show that while the algorithm successfully gained information from new data, due to its improved performance over the initial tree, the cost of adding each sample individually from a batch is more than remaking the tree. As the remade tree also outperforms the updated one, using this algorithm does not provide a benefit when adding data by batches.

However, when adding a small number of samples at a time, the algorithm outperforms remaking the tree in terms of time cost. This algorithm could potentially being effective in a pipeline, where new data is immediately added to the tree as it becomes available. There is still benefit to recreating the tree, as the retrained tree outperforms the updated one.

In conclusion, while this research presents a working algorithm for the addition of data to a decision tree after its initial creation, the algorithm does not perform as well as remaking a decision tree from the entire collection of data.

6 Related Work

As discussed earlier in this paper, while a significant amount of research has gone into decision

trees, little has gone into the consideration of decision trees as related to one another. However, for other supervised learning models, this practice is more common. Neural networks, which attempts to classify data using a structure similar to the neurons of a brain, have a variety known as recurrent neural networks, which consider samples to be related to one another, such that the predicted result of one set of features is considered and used to determine the next one.[29] They are used for similar problems to those of decision trees, such as pattern and image classification.[23][14] Many neural networks also allow for the internal weights to be saved, so that they can be reused as the starting values for training another network. This allows the new network to start off as similar to the old one, meaning fewer changes may be needed to reach an acceptable classifier versus randomly generated starting weights. This idea of neural networks being related to one another is the same to how our algorithm considers decision trees.

Despite the lack of research into this aspect of decision trees, they have been optimized in a variety of other ways. Efforts have been made towards parallelizing the computation for creating trees, towards reducing the time cost of making a tree, and some work has focused on improving a tree's predictive ability.[31]

One way the time cost has been improved is through gradient-boosted trees, which attempt to find the best tree, but may not always do so. The concept of boosting is covered well in Freund and Schapire's *A Short Introduction to Boosting*, which was published in 1999.[6] In Python, this method is implemented in various packages. Chen and Guestrin discuss one such package, XGBoost, in their presentational paper on it, *XGBoost: A Scalable Tree Boosting System.*[37][38]

Implementations of a variety of these approaches are available, such as the scikit-learn DecisionTreeClassifier, a Python-based implementation, which have been optimized to the specific language in which they are implemented.[30] The Spark distributed library offers a DecisionTree object, which can be trained using a distributed system, improving its training speed significantly.[12] Several research papers and documentations can be found in the bibliography for other versions, as well as specific applications of decision trees.

7 Future Work

This research has proven classification performance can be improved by adding data to a preexisting decision tree. However, the proposed algorithm has a substantial time cost associated when handling large batches of post-training data. Because of these outcomes, there are several areas into which this research could be extended.

One area of future work is to improve this algorithm. Despite its ability to update a tree, it has several downsides. It could be improved by updating by the batch, instead of by the sample. Then, the tree is traversed just once, instead of n times, where n is the number of new samples. There is also room for improvement in the types of data accepted. Currently, as this algorithm is built to work with the decision tree algorithm presented in *Artificial Intelligence: A Modern Approach*, it only handles categorical data. In most available implementations, continuous data is accepted, and even preferred. Python's scikit-learn package requires all data to be continuous, requiring the use of solutions such as the Pandas package usedummies method to convert categorical data into a series of features, with values of 0 or 1 marking if the row has that specific value for a categorical feature.

Another area that could be further explored is the extension of thinking of subsequent models as continuous instead of discrete to other machine learning methods. While neural networks have similar capabilities already, many other methods, including other forms of decision trees, do not.

While this research was unsuccessful in improving the time cost of using new data for decision trees, it has proved decision trees can be improved by adding new data, instead of by being remade, allowing for a wide variety of further research.

References

[1] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, "Advances in Optimizing Recurrent Networks," arXiv:1212.0901 [cs], Dec. 2012.

[2] W. Peng, J. Chen, and H. Zhou, "An Implementation of ID3 Decision Tree Learning Algorithm," From, 2009.

[3] Richard P. Lippmann, "An Introduction to Computing with Neural Nets," IEEE ASSP Magazine, Apr-1987.

[4] Stuart J. Russell and Peter Norvig, Artificial Intelligence: A Modern Approach. Prentice Hall.

[5] B. Marr, "A Short History of Machine Learning – Every Manager Should Read," Forbes. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2016/02/19/a-short-history-of-machine-learning-every-manager-should-read/. [Accessed: 09-Dec-2017].

[6] Yoav Freund and Robert E. Schapire, "A Short Introduction to Boosting," Journal of Japanese Society for Artificial Intelligence, vol. 14(5), pp. 771–780, Sep. 1999.

[7] "Accuracy, Precision, Recall & F1 Score: Interpretation of Performance Measures," Exsilio Blog, 09-Sep-2016. [Online]. Available: http://blog.exsilio.com/all/accuracy-precision-recall-f1-score-interpretation-of-performance-measures/. [Accessed: 14-Apr-2018].

[8] "Comparing supervised learning algorithms," Data School, 27-Feb-2015. [Online]. Available: http://www.dataschool.io/comparing-supervised-learning-algorithms/. [Accessed: 20-Nov-2017].

[9] Pierre Demartines and Jeanny Hérault, "Curvilinear Component Analysis: A Self-Organizing Neural Network for Nonlinear Mapping of Data Sets," IEEE TRANSACTIONS ON NEURAL NETWORKS, vol. 8, no. 1, pp. 148–154, Jan. 1997.

[10] Kenneth Sörensen and Gerrit K. Janssens, "Data mining with genetic algorithms on binary trees," European Journal of Operational Research, no. 151, pp. 253–264, 2003.

[11] "Datasets — Kaggle." [Online]. Available: https://www.kaggle.com/datasets. [Accessed: 14-Apr-2018].

[12] "Decision Trees - RDD-based API - Spark 2.2.0 Documentation." [Online]. Available: https://spark.apache.org/docs/2.2.0/mllib-decision-tree.html. [Accessed: 14-Apr-2018].

[13] P. Gupta, "Decision Trees in Machine Learning," Towards Data Science, 17-May-2017. [Online]. Available: https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052. [Accessed: 10-Apr-2018].

[14] Giorgio Giacinto and Fabio Roli, "Design of Effective Neural Network Ensembles for Image Classification Purposes." .

[15] "GitHub," GitHub. [Online]. Available: https://github.com. [Accessed: 04-Apr-2018].

[16] J. Ross Quinlan and Ronald L. Rivest, "Inferring Decision Trees Using the Minimum Description Length Principle*." Academic Press, Inc., 20-Jan-1988.

[17] Léon Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent."

[18] J. Brownlee, "Machine Learning is Popular Right Now," Machine Learning Mastery, 13-Dec-2013. .

[19] "Machine Learning: What it is and why it matters." [Online]. Available: https://www.sas.com/en_us/insights/analytics/machine-learning.html. [Accessed: 14-Apr-2018].

[20] Gavin Hackeling, Mastering Machine Learning with sci-kit learn. Packt Publishing, 2014.

[21] G. Machado, "ML Basics: supervised, unsupervised and reinforcement learning," Medium, 06-Oct-2016. .

[22] Dr. Yashal Singh and Alok Signh Chauhan, "Neural Networks in Data Mining," Journal of Theoretical and Applied Information Technology.

[23] Richard P. Lippmann, "Pattern Classification Using Neural Networks," IEEE Communications Magazine, Nov-1989.

[24] jamesdmccaffrey, "Precision and Recall with Binary Classification," James D. McCaffrey, 05-Nov-2014. .

[25] "Project Jupyter." [Online]. Available: http://www.jupyter.org. [Accessed: 12-Dec-2017].

[26] "Python :: Anaconda Cloud." [Online]. Available: https://anaconda.org/anaconda/python. [Accessed: 04-Apr-2018].

[27] Sebastian Raschka, Python Machine Learning. Packt Publishing, 2016.

[28] R. Littleton, Recreating ML models", 2017.
"We update every 28 days. Our update timeframe is based on the average duration between fills of a prescription drug claim. Ideally, we would update the algorithms as fast as meaningful improvements could be detected in algorithm performance (what is meaningful is something we would have to decide). For this model, one time per month makes sense a priori, but we have also seen that prediction quality goes down over the month. So it's a balancing act between computational resources, priorities and team capacity. I'm not sure if that dimension of the decision making is pertinent to what you are working on, but just in case."

[29] D. P. Mandic and J. A. Chambers, Recurrent neural networks for prediction: learning algorithms, architectures, and stability. Chichester ; New York: John Wiley, 2001.

[30] scikit-learn, scikit-learn decision tree. .

[31] Andrew Tulloch, "Speeding Up Decision Tree Making."

[32] B. A. Walther and J. L. Moore, "The concepts of bias, precision and accuracy, and their use in testing the performance of species richness estimators, with a literature review of estimator performance," Ecography, vol. 28, no. 6, pp. 815–829, Dec. 2005.

[33] "The GDELT Project." [Online]. Available: https://www.kaggle.com/gdelt/gdelt. [Accessed: 15-Apr-2018].

[34] Carlos Bort Escabias, "Tree Boosting Data Competitions with XGBoost." Universitat Politècnica de Catalunya - Universitat de Barcelona, 2017-2016.

[35] "What is Big Data? – Amazon Web Services (AWS)," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/big-data/what-is-big-data/. [Accessed: 25-Apr-2018].

[36] "Why accuracy alone is a bad measure for classification tasks, and what we can do about it - Tryolabs Blog." [Online]. Available: https://tryolabs.com/blog/2013/03/25/why-accuracy-alone-bad-measure-classification-tasks-and-what-we-can-do-about-it/. [Accessed: 20-Nov-2017].

[37] DMLC, XGBoost. .

[38] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," arXiv:1603.02754 [cs], pp. 785–794, 2016.

8 Appendix

8.1 GitHub

A full copy of the code, datasets, graphics, and other relevant files associated with this research can be found at https://github.com/JeremySwerdlow/senior_research.

8.2 Tables

	Initial	Updated	Remade
Accuracy	0.666667	0.666667	0.666667
Recall	0.666667	0.666667	0.666667
Precision	1.0	1.0	1.0
F1-Score	0.8	0.8	0.8
Time (sec)	0.145396	0.018312	0.095628

Table 1: Pets Dataset Random Split Results

	Initial	Updated	Remade
Accuracy	0.333333	0.333333	0.333333
Recall	0.333333	0.333333	0.333333
Precision	1.0	1.0	1.0
F1-Score	0.5	0.5	0.5
Time (sec)	0.103950	0.059968	0.098873

Table 2: Pets Dataset Updated Weights Results

	Initial	Updated	Remade
Accuracy	1.0	1.0	1.0
Recall	1.0	1.0	1.0
Precision	1.0	1.0	1.0
F1-Score	1.0	1.0	1.0
Time (sec)	0.179619	0.089466	0.189644

Table 3: Pets Dataset New Node Results

	Initial	Updated	Remade
Accuracy	0.668675	0.716867	0.710843
Recall	0.710526	0.719512	0.715152
Precision	0.907563	0.991597	0.991597
F1-Score	0.797048	0.833922	0.830986
Time (sec)	0.322284	1.735093	0.675836

Table 4: 5-Day Dataset Random Results

	Initial	Updated	Remade
Accuracy	0.732919	0.689441	0.689441
Recall	0.745223	0.733333	0.733333
Precision	0.975000	0.916667	0.916667
F1-Score	0.844765	0.814815	0.814815
Time (sec)	0.262060	1.201642	0.238624

Table 5: 5-Day Dataset Updated Weights Results

	Initial	Updated	Remade
Accuracy	0.745342	0.745342	0.745342
Recall	0.745342	0.745342	0.745342
Precision	1.0	1.0	1.0
F1-Score	0.854093	0.854093	0.854093
Time (sec)	0.352664	1.525556	2.035871

Table 6: 5-Day Dataset New Node Results

	Initial	Updated	Remade
Accuracy	0.736842	0.740351	0.754386
Recall	0.763838	0.760870	0.764286
Precision	0949541	0.963303	0.981651
F1-Score	0.846626	0.850202	0.859438
Time (sec)	2.876278	7.184069	2.595692

Table 7: 5-Day 2nd Dataset Random Split Results

	Initial	Updated	Remade
Accuracy	0.798561	0.791367	0.798561
Recall	0.800725	0.799270	0.800725
Precision	0.995495	0.986486	0.995495
F1-Score	0.887550	0.883065	0.887550
Time (sec)	1.623065	4.152574	1.687860

Table 8: 5-Day 2nd Dataset Updated Weights Results

	Initial	Updated	Remade
Accuracy	0.758993	0.758993	0.766187
Recall	0.768382	0.768382	0.768116
Precision	0.981221	0.981221	0.995305
F1-Score	0.861856	0.861856	0.867076
Time (sec)	2.274591	7.117846	1.970071

Table 9: 5-Day 2nd Dataset New Node Results

	Initial	Updated	Remade
Accuracy	1.0	1.0	1.0
Recall	1.0	1.0	1.0
Precision	1.0	1.0	1.0
F1-Score	1.0	1.0	1.0
Time (sec)	1.279615	17.855069	1.176098

Table 10: Mushroom Dataset Random Results

	Initial	Updated	Remade
Accuracy	1.0	1.0	1.0
Recall	1.0	1.0	1.0
Precision	1.0	1.0	1.0
F1-Score	1.0	1.0	1.0
Time (sec)	1.039368	1.230007	12.603490

Table 11: Mushroom Dataset Updated Weights Results

	Initial	Updated	Remade
Accuracy	0.998094	1.0	1.0
Recall	0.995868	1.0	1.0
Precision	1.0	1.0	1.0
F1-Score	0.99793	1.0	1.0
Time (sec)	1.283095	13.452872	1.143861

Table 12: Mushroom Dataset New Node Results

8.3 Example Graphs

Below is a collection of three graphs of decision trees generated from the Pets Dataset. The first graph depicts the initial decision tree, built from a portion of the dataset. In the second tree, additional data has been added, creating a new node. The final graph has further data added, such that the weights of nodes have been updated.

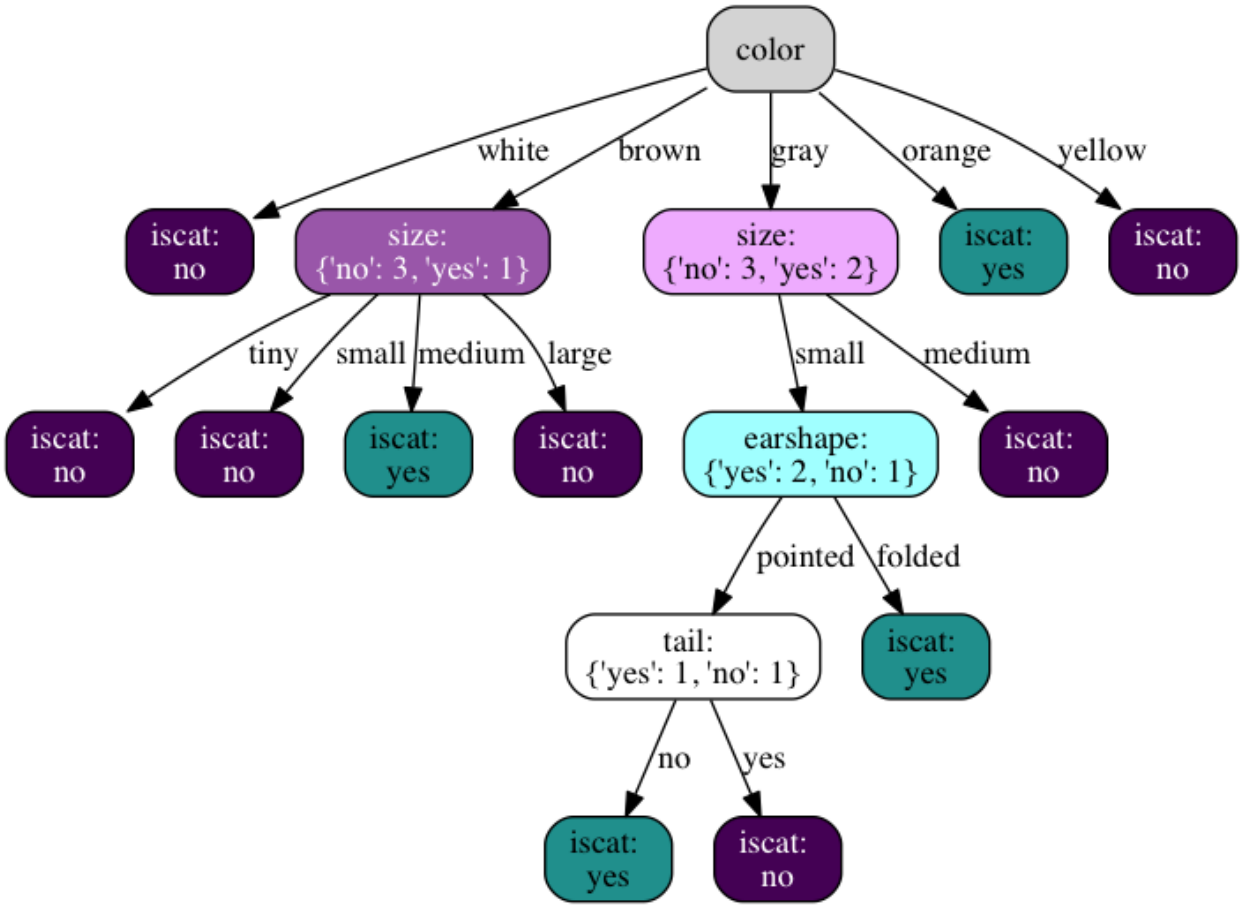

Figure 5: Decision tree from initial dataset

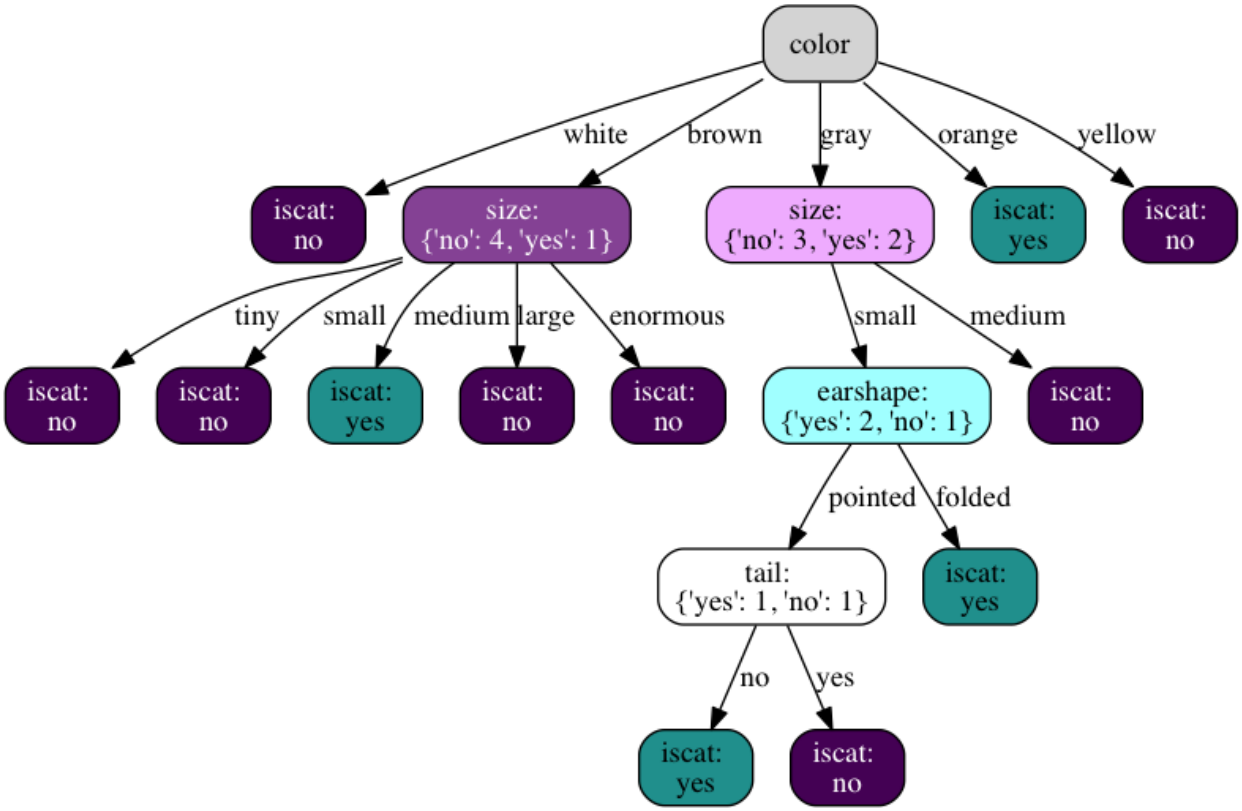

Figure 6: The above decision tree with new node added via algorithm

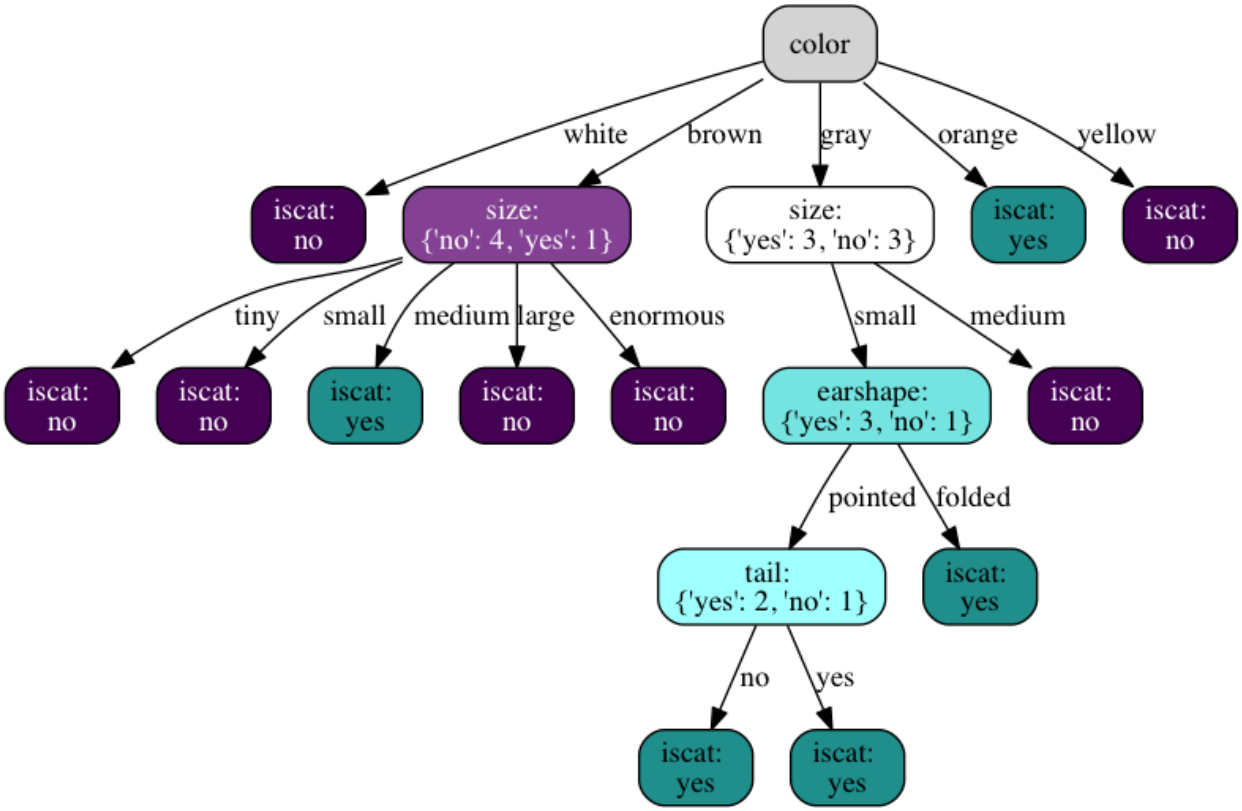

Figure 7: The final decision tree with updated weights and new node