Dynamic Visualization of Scikit Learn Random Forest Classification Models

ADAM WOJTULEWSKI, University of Florida, United States

ABSTRACT

Comparing individual decision trees that make up Random Forest Models using existing visualization libraries, which rely on static visualizations, can prove be a tedious and difficult task. Of the problems with current approaches, a primarily concern involved with using these tools is that they have a difficult time adjusting to deep trees with a large number of nodes and provide minimal information in assessing the performance of testing data on subgroups of nodes in trees. To address the usability and flexibility problems of existing visualization libraries, this paper introduces a new type of dynamic visualization that affords both studying individual decision trees and comparing different decision trees generated by Random Forest Models. To study individual and groups of nodes of a single decision tree, this library allows users to zoom into nodes of interest using a bounded box technique and allows users to study datapoints contained in a set of nodes by remembering clicked nodes and subsequently generating tables and figures based on the nodes stored. To differentiate data between different trees, users may combine this functionality with a tab switching functionality to instantly compare information between two different trees that make up the trained Random Forest classifier. Through this visualization application, it is easier to determine shared trends among different decision trees making up the same Random Forest Model, allowing to better generate rulesets for custom models and to eliminate irrelevant Decision Trees to prevent them from hindering the performance of the decision-making of the Random-Forest Classifier aggregator.

CCS Concepts: • **Artificial Intelligence → Machine Learning → Random Forest**; Visualization; Interactive Display; Data preprocessing

KEYWORDS: Random Forest Classification Dynamic Visualization, Dynamic Visualization, Random Forest Visualizer

1 INTRODUCTION

Traditional computer science approaches involve programmers developing solutions with a specific end goal in mind that adheres to all possible combinations. In fact, it is often pertinent to provide both proofs of correctness and of time complexity to show that an algorithm will provide the expected output. In the past few years though, a subset of computer science, Artificial Intelligence, has become popular with a different approach, developing a solution that has the capabilities of self-teaching and adapting to situations that the programmers may not have originally anticipated. One type of artificial intelligence is machine learning, which has a general goal of trying to teach a computer how to make human-like decisions in order to solve prediction-oriented problems. Due to its frequently impressive predictive performance, the application of machine learning has pervaded the computer science world, with many companies eager to capitalize on its expected

benefits. Unlike traditional programming approaches though, due to their tendency of relying on advanced mathematics techniques, such as linear algebra and calculus, where numerous calculations are performed behind the scenes within a program, there is typically a lack of transparency associated with these approaches.

While most machine learning approaches provide limited explanations surrounding their methodology for the average researcher, thus being labeled 'black-box methodologies', one example of a machine learning technique that is easier to interpret is the decision tree model. A decision tree is a hierarchical data structure that attempts to divide data with a set of rules, splitting into subtrees depending on whether certain conditions are met.

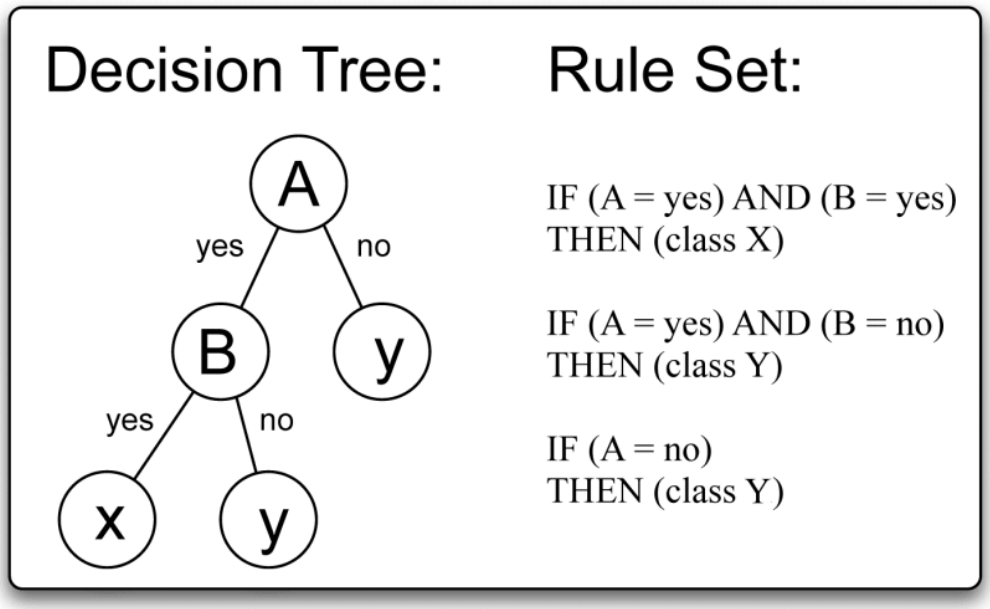


Figure 1: Provides an explanation of how a decision tree works [1].

For a binary classification tree, shown in Figure 1, a node splits into a left subtree if a condition is met and to the right subtree if a condition is not met (or vice versa depending on the implementation). By observing the non-leaf nodes of a tree, it is possible to build a ruleset that is used to get to each leaf node at the bottom of the tree, helping to identify potential groups of interest, an important task in many fields such as cancer research and targeted marketing. While a Decision Tree is intuitive, they are often subject to overfitting and consequently, the behavior of a single DT is best evaluated by comparing its result against that of several DTs. This combination of several decision trees, where a prediction vector y for some input matrix X (each primarily stored as a DataFrame in Python) is based on the aggregated result of several decision trees is in itself a separate machine learning model known as a Random Forest Model.

Random Forest Models are frequently called upon and used because of their tendency to provide more generalizable behavior for testing datasets. For models with good performance, information like feature importance can be quite beneficial to isolate and identify factors that most

influence predictions, which could point to areas of future research and discovery. Returning to the earlier example of cancer research, being able to identify that some feature is directly correlated to some desired output, like patient outcome, could help researchers to better treat patients, meaning more lives could be saved. However, sometimes, there can exist situations where certain factors cannot have an impact on the prediction of the model, or a model produces one or several poorly behaving decision trees as part of its aggregation. Whenever this is the case, researchers may opt to update the model by removing these trees, with the intent of producing a better behaving and hopefully better performing model.

With this information in mind, it becomes clear that high quality tools for determining the quality of decision trees could be quite useful. Unfortunately, the current state of visualization code for decision trees is quite lacking, making the process of analyzing decision trees largely tedious and inefficient. With the current visualization tools that exist for decision trees, being able to quickly tell how well a tree is performing from its visualization is not easy and this is especially true for large, dense, and deep trees, as a lot of the information of the nodes becomes quite cluttered and difficult to see. Moreover, the information given can be quite confusing for some learning about this model.

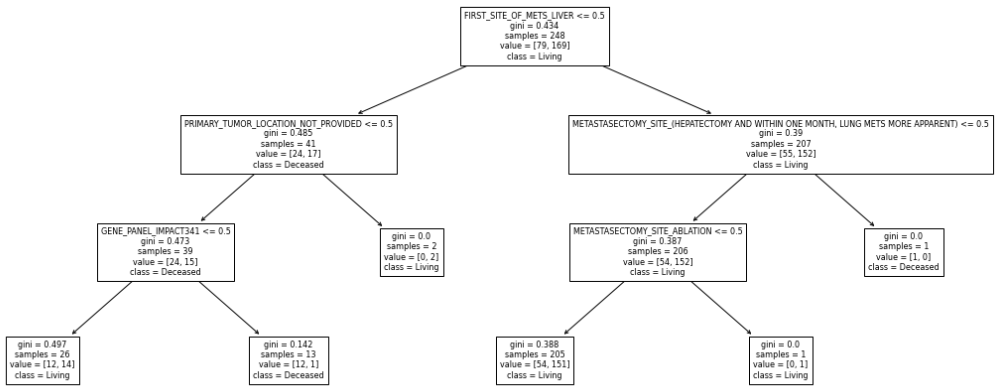


Figure 2: Shows a decision tree visualization for the overall survival of colorectal cancer patients made with matplotlib.pyplot.

Taking a look at this example of a visualization for the matplotlib.pyplot package in Figure 2, which is perhaps the most basic tree visualization choice, the information portrayed is not immediately clear for most casual users and the text for this tree, which only has a maximum depth of 4, is quite small, meaning it could be difficult for some individuals to see clearly.

Now we observe a tree that has a maximum depth of 16 in Figure 3.

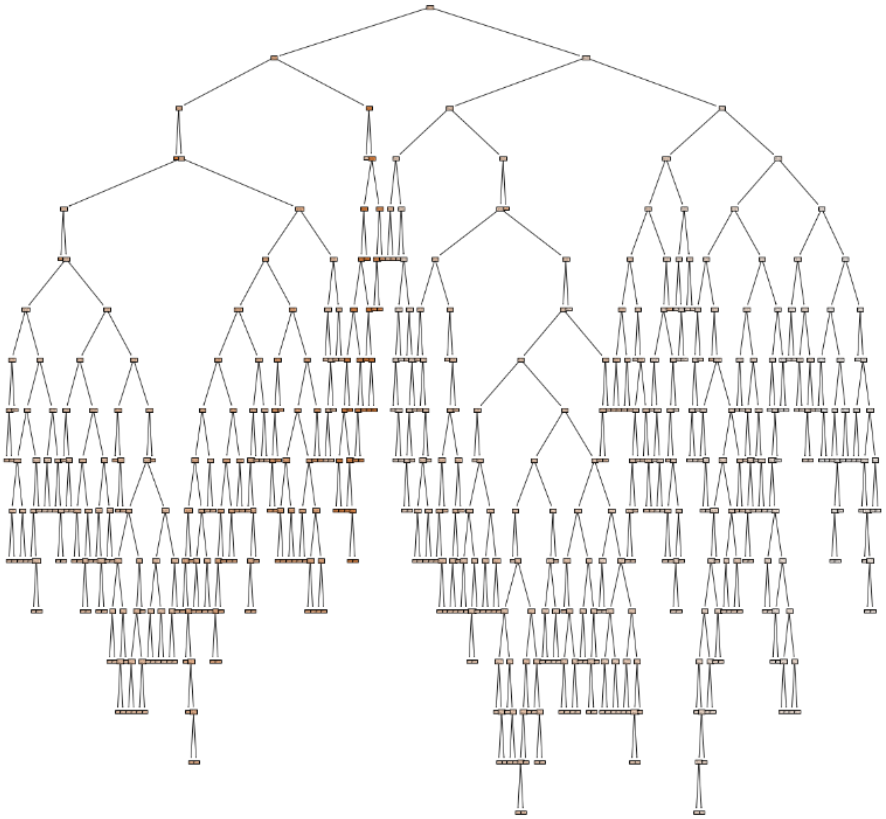


Figure 3: Reveals a decision tree visualization with a depth of 16 that was made with matplotlib.pyplot, a tool that does not provide a zoom option, rendering a largely unusable image.

Here, besides the general shape of the tree, which hardly reveals any interesting information, nothing significant can be learned from the tree. Hovering over the nodes provides no information and besides being able to zoom into the image by double clicking on the image (when generated in Jupyter Lab or Jupyter Notebooks), which is restricted in its scope, there is a limited interactivity with this image. Considering interactive and colorful libraries are increasingly used both in the workforce and in research environments, the narrow capabilities of this library and other similar decision tree libraries pales in comparison to visualization libraries like Seaborn, Plotly, and Dash, which can be used for other types of figures.

To address the usability and flexibility problems of existing visualization libraries, this paper introduces a new type of dynamic visualization that affords both studying individual decision trees and comparing different decision trees generated by Random Forest Models using an interface built primarily in Dash. In short, this library allows users to interact and view a set of nodes and it generates figures and tables dynamically based on clicked nodes to allow users to study a set of nodes and evaluate the performance of a decision tree with respect to this set of nodes; each of these will be thoroughly detailed in the paper below. Through this visualization application, it is easier to determine the characteristics and features of a single decision tree and

the shared trends among different decision trees making up the same Random Forest Model, important outcomes that could allow researchers to make improved models in the future.

2 RELATED WORK

Naturally, given the increased popularity of AI approaches and a push to better understand their methodologies, it is no surprise that there is a great deal of literature which can be consulted with respect to both random forest and decision tree models and the development of visualization libraries for each model.

There are three steps in the machine learning pipeline, which roughly include data pre-processing, model training, and then model validation, steps which occur before, during, and after model building respectively [2]. Irrespective of the step, visualization methods can be very beneficial towards researchers, providing them with additional context that may not be evident with initial approaches. Therefore, it comes as no surprise that the development of visualization software has been undertaken with respect to all three categories.

Regardless of which step of the pipeline is being addressed with the visualization software, a set of common guidelines should be followed in order to generate software that is beneficial for researchers. Hoque et al are vocal proponents of this idea as they first detailed the essential capabilities and the human concerns that should be considered by all Human-Centered Computing developers and then supported their claims by highlighting four examples of Human-Computing Artificial Terms (HCAI) tools that abode by these points [3]. They established these capabilities by citing Ben Shneiderman, who argued that Human-Computing AI tools should 1) **amplify** or strengthen existing abilities of researchers, 2) **augment** the number of abilities researchers have, 3) **empower** researchers to solve tasks that were previously unsolvable, or 4) **enhance** the performance of existing tools [4,5]. After clarifying the meaning of each bold word and providing examples of how tools solved these, concerns with respect to AI models were identified with the intent of emphasizing what users prioritize when applying these tools to their work. From the list of these rules, the three most relevant to this paper include transparency, explainability, and understandability, as these all strongly motivate the development of accompanying software to provide additional insight to models. With these considerations in mind, the paper delves into the importance of visualization techniques and outlines exactly how they help researchers. In short, visualization techniques for models are open-ended and data-driven and help to externalize data, allowing for users to better understand both the information contained within the dataset and the processes that act on the dataset, allowing researchers to make more informed subsequent decisions with respect to the ML pipeline on a current dataset. With respect to these outcomes and the considerations discussed previously, HCAI tools like TimeFork, HaLLMark, Outcome-Explorer, and uxSense provided substantial benefits to end-users, justifying the importance of abiding by these guidelines for users.

With these general considerations in mind, this paper will now begin to look at different random forest and decision tree visualization tools to describe the ways in which the dynamic program developed in this paper differs.

Currently, there exist many libraries designed with the intention of analyzing and learning from random forest and decision trees. One of the most popular libraries is the Treeinterpreter library in Python, a library which seeks to extract information about tree-based models like decision trees and random forests and interpret their predictions [6]. Experts use Treeinterpreter to understand how each feature contributes to a prediction by decomposing the prediction into bias. Running the predict function within this library yields tree distinct outputs: 1) the predicted values from the model, the average prediction of the model (the bias term), and a 2D array where each row corresponds to a prediction, and each column represents the contribution of a specific feature to that prediction. The last of these outputs is especially significant, as it allows experts to efficiently determine feature importance in a model, which is the primary use case for this library. The library unfortunately does not support dynamic visualizations of data, relying on numerical and textual representations of trees, and thus it is often used alongside libraries like SHAP (SHapley Additive exPlanations) and LIME (Local Interpretable Model-agnostic Explanations). That being said, these libraries are still primarily used to visualize feature significance as opposed to the trees themselves and do not allow close inspection of tree nodes; therefore, their goals differ from the library developed in this paper.

Since interpretable visualizations in terms of only feature importance already exist and are readily available by these popular tools, it was decided that tree visualization would be the focus on this paper. Currently, there exist two main choices in terms of tree visualization, those being matplotlib.pyplot and dtreeviz.

As discussed above, the matplotlib.pyplot library provides a very bare and simple choice of visualization for trees. It provides information such as the splitting value, gini, number of samples found therein, the distribution of the class, and the overall class based on the majority value for the dataset, information which can be found below in Figure 4.

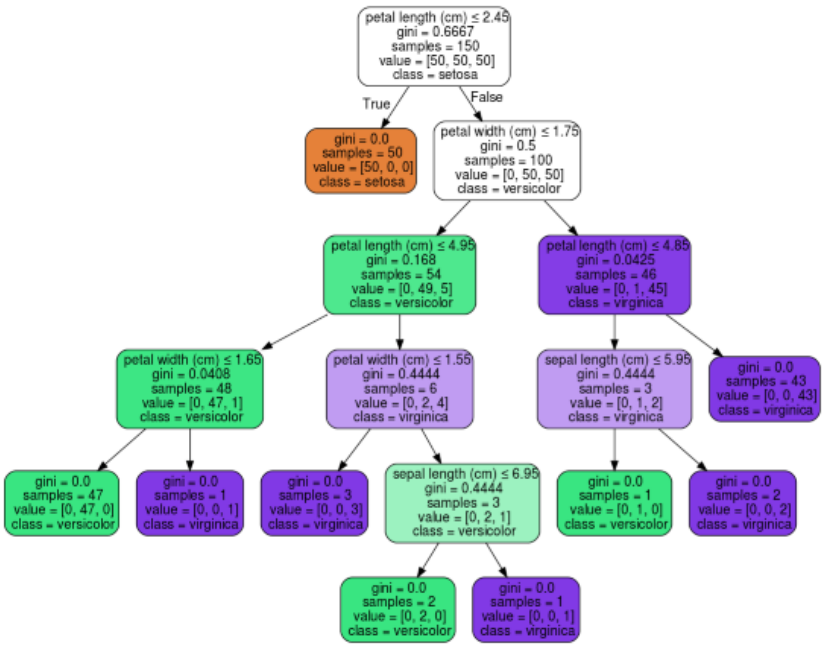


Figure 4: Reveals the Random Forest Classification for the Iris dataset in matplotlib.pyplot.

As discussed above, for smaller decision trees, visualization is frequently sufficient as it provides the general shape of the model and can be exported, but as the size of the tree increases, it becomes increasingly tedious to view node values and it becomes less insightful. While zoom is available, this action is completed through a double click action over a certain point and can only be completed once, with a subsequent double click reverting back to the previous state. Moreover, while the distribution of binary values is shown, if a user is interested in learning about the actual data values stored in the node, they would have to manually search for these nodes and then perform subsequent operations on them, requiring cognitive and physical load on the part of the user. Without immediate access to the datapoints stored in a node to fully examine their values, the feedback given from this visualization loses some of its value. Additionally, for leaf nodes of trees that have a substantial depth value, users must trace back on the node's path back to the root to understand the complete ruleset followed in order to get that leaf node.

Compared to the matplotlib.pyplot library, the dtreeviz library offers several benefits that make it preferred to researchers over the previously described library. Considering that the library was designed with the intention of visualizing trees, this makes sense. Dtreeviz creates visually appealing and detailed static visualizations which are highly customizable. Since the gini coefficient (certainty score) and the three additional values (samples, value, and class) cost space and do not greatly help with interpretation of the model's information, the library removes these labels found in the previous figure and instead offers histograms which better reflect the distribution of the items within nodes. With purposeful coloring corresponding to the respective target value, it becomes much easier to identify how the target values are split up among the nodes. All of these changes yield figures that resemble Figure 5.

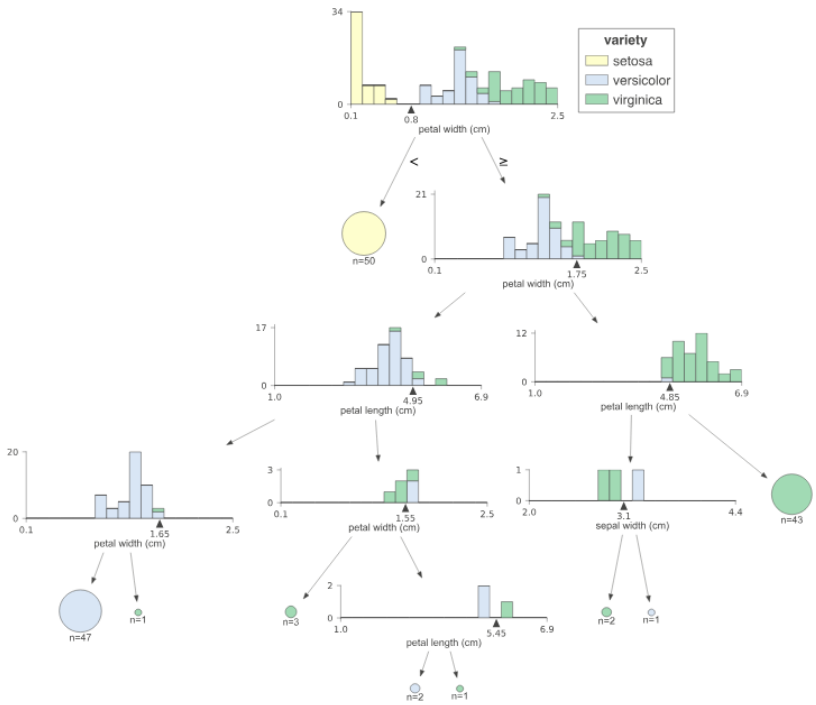


Figure 5: Reveals the Random Forest Classification for the Iris dataset in dtreeviz.

While the visualization does a significantly better job in displaying the distribution of target values with respect to the splitting value, the same criticisms for this value hold true as before. If a user is interested in learning about the actual data values stored in the node, they would have to apply selective constraints on DataFrame objects to find these values, requiring a cognitive and physical load and a knowledge of how to perform these operations on the initial DataFrame. Likewise, users must trace back on the node's path back to the root to understand the complete ruleset followed in order to get that leaf node and this becomes more difficult with the increased number of graphs and figures that make up the visualization. Finally, just like the matplotlib.pyplot library, for deep trees, the figure becomes increasingly difficult to read and the limited zooming options can make reading the figure a daunting task. As far as static visualization libraries are concerned, it is difficult to improve upon what dtreeviz offers, but dynamic visualizations can address many of the concerns presented above, which motivates the choice within the paper to pursue a dynamic visualization option.

While the visualization techniques referenced up until this point were all static, there are dynamic visualization libraries that have been made. Therefore, it is important to remember that the sole fact that the tree visualization software is dynamic is not in itself not enough to differentiate it from other visualization applications and libraries. In November 2024, Munoz et al developed a tool to interactively create decision trees through visualization of subsequent Linear Discriminant Analysis (LDA) diagrams [8]. As part of their development process, they created an interactive webtool, named Guided Decision Tree (GDT), for data handling, visualization, and guided tree creation. For data handling, their tool allows users to input and parse character delimited files, select features (X), select a target feature (y), and generate a subsequent model with Star Coordinate and LDA visualizations to provide insight to users. With this information generated, users have the ability to make modifications to the model by applying their expertise in the field on the models directly to see changes, including classification accuracy, compared against the DecisionTreeClassifier within the Scikit-Learn Python Library. The key distinction between the pieces of software described above and the one developed within the paper is the overall goal of each piece of software. The piece of software developed by Munoz et al specifically aims to enable experts in fields to directly modify one decision tree's nodes, assuming that experts have the knowledge of specific constraints that would impact the model's performance and testing these actions. As a direct contrast, the visualization software developed within this paper generates a forest of trees and allows researchers to quickly observe the performance of paths generated by the tree on testing data, model visualization and visualization of model validation on individual or groups of nodes. Hence, while on the surface, the description of each paper seems to be similar on surface value, the aims and inner works of the papers are quite different from each other.

3 DYNAMIC VISUALIZATION FOR ANALYZING RANDOM FOREST MODELS

3.1 Running the Code

3.1.1 Environment

As the system assumes that users will primarily be using the visualization software in a manner similar to the matplotlib.pyplot and dtreeviz libraries, the system containing all of the work is contained within a **JupyterLab Notebook** and all of the code was written using the **Python** programming language.

3.1.2 Code Repository

All of the code written for this project can be found on the following Github Repository: https://github.com/AdamWojtul28/Interactive-RF-Visualizer.

3.1.3 Required Steps to Run Code

1. The code in this Github was made using Anaconda Python, specifically JupyterLab. To use this code it is first necessary to download JupyterLab and Anaconda Python from the official website: https://www.anaconda.com/download
2. Install Git (which includes Git Bash) from this website: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
3. Create a folder in your system to host the project.
4. Open a CLI interface like Git Bash
5. Within the folder that was just created, type the following: git clone https://github.com/AdamWojtul28/Interactive-RF-Visualizer.git
6. Run Anaconda Navigator
7. Run JupyterLab
8. Complete the required library installations that are used to power this application, which can be completed in one of two ways.
    a) Run the cells in the notebook Decision_Tree_Visualizer.ipynb starting from the first cell.
    b) Open a terminal and run the following command **pip install pandas numpy scikit-learn dash plotly ipywidgets9)**
9. Run each line in the notebook either clicking on the Play Button in the software or with the Shift + Enter keyboard combination. Once the entire notebook is compiled and ran within a JupyterLab Notebook, the resultant visualization can either be hosted locally within the JupyterLab Notebook or by accessing http://localhost:{8062}, which automatically loads upon running the generate_dynamic_rf_classifier_visualizer(rfclf, X_train, X_test, y_train, y_test, port_number=8062) function. In order, the parameters include a RandomForestClassifier object, the training dataset input of type DataFrame, testing dataset input of type DataFrame, training dataset output of type DataFrame, and testing dataset output of type DataFrame.

3.2 Tree Structure Construction

The first step in the development of the application involved the construction of the tree structure that would subsequently be displayed with the Plotly functionality. Naturally, given the decision trees within the Scikit-Learn Library are all binary trees, recursion is naturally suited for use when constructing the tree. Therefore, it was pertinent to consider the structure of the tree nodes, including all of their necessary stored variables, and how this information would be updated for each node. With respect to the goal of this application, it became clear that several values would need to be stored at each node.

3.2.1 Value

The first and most obvious value to be stored in a tree node was the string display value that would show whenever a user hovers over a node in the Plotly figure. For non-leaf nodes, what

this value should be is evident, as in a majority of tree visualization applications, each node stores and displays the splitting condition. Therefore, as each node is entered, the RandomForestClassifier. estimators_.tree_.feature[node_value] and RandomForestClassifier. estimators_.tree_.threshold[node_value] are taken and are used to construct the value string. This value is calculated within each node and does not need to be recalculated.

3.2.2 Position

Given that the tree uses x and y coordinates to correctly display the tree, it is pertinent to both store and properly calculate these values upon tree construction. Naturally, the root is centered at the coordinate (0,0) but from there, determining the proper position of the nodes is not as easy as it seems. The reason for this is that every node must be placed in such a way as to prevent edges from overlapping with one another and to use the window's space as efficiently as possible so nodes are not too cluttered.

For small nodes, the impact of these two key details may be ignored and a simple formula may be used and in fact, this is how the initial positioning algorithm was designed. After the root was assigned, the subsequent nodes' positions are calculated making use of the fact that each node in the RandomForestClassifier. estimators_.tree_ stores the indices of its left and right child. Using these values, Algorithm 1 was used to assign values for the position of the left and right child of each node.

ALGORITHM 1: Position Assignment Initial Approach

```
left_pos = (pos[0] - 1 / (depth + 1), pos[1] - 1)
right_pos = (pos[0] + 1 / (depth + 1), pos[1] - 1)
```

For large trees, this algorithm proved to be very flawed, as trees with a depth of 10 resulted in visualizations that hardly resembled trees, with edges of nodes and nodes themselves overlapping with one another. Hence, while a simple algorithm like Algorithm 1 seemed intuitive enough at first, it could not be used in this application for good results and a new algorithm had to be proposed.

To simultaneously prevent edges from crossing one another and to assign more space to nodes with deeper subtrees, nodes were assigned by taking the midpoint between the minimum and maximum available x position, with defaults of -1 and 1 respectively and then updating the left and right children. To update the children, it was necessary to calculate their number of total children, updating the minimum to consider the proportion of their parent's total nodes that they stored. This calculation would only need to be completed once for the left child, as the x_minimum variable is updated to reflect the proportions at that point. In the y-direction, the calculation was simpler, as it simply involved the product of the depth of a node with the desired vertical gap passed into the function to start. While more advanced placement techniques are possible, with these techniques being used by other libraries, this was sufficient for the purposes of this application.

ALGORITHM 2: Position Assignment Final Approach

```
# Recursive function to position nodes
function position_nodes(node_id, depth, x_min, x_max):
    # Base Case: If the node_id is invalid (no node exists), return None
    if node_id == -1:
        return None

    # Calculate the x-coordinate as the midpoint of the total available horizontal range
    x_mid = (x_min + x_max) / 2

    # Calculate the y-coordinate as the depth multiplied by the established vertical gap
    y_pos = -depth * vert_gap

    # Assign position to the current node
    node.pos = (x_mid, y_pos)

    # Determine the number of leaf nodes in the left and right subtrees
    total_leaves = count_leaves(node_id)  # Total leaves in the subtree rooted at this node
    left_leaves = count_leaves(left_child_id)

    # Proportionally divide the horizontal space based on the number of leaves
    if left_child exists:
        # Calculate the width of the left child's horizontal range
        left_width = (left_leaves / total_leaves) * (x_max - x_min)

        # Recursively position the left child
        position_nodes(left_child_id, depth + 1, x_min, x_min + left_width)

        # Update x_min for the right child
        x_min += left_width

    if right_child exists:
        # The remaining space is for the right child
        right_width = (right_leaves / total_leaves) * (x_max - x_min)

        # Recursively position the right child
        position_nodes(right_child_id, depth + 1, x_min, x_max)

    # Return the positioned node
    return node
```

3.2.3 Color

To clearly highlight leaf nodes and make them plainly visible to users, since leaves are naturally associated with the color green, this application makes use of the natural mapping and colored all leaf nodes green and all non-leaf nodes blue.

3.2.4 Data

Since the primary focus of the software was to highlight which datapoints belong to which leaf nodes, it was pertinent to determine how to assign each of the values within a test dataset to a leaf node depending on the established ruleset of a tree. This is possible through the code in Algorithm 1 below, which once again relies on the stored information within the RandomForestClassifier. estimators_.tree_ generated through the Scikit Learn Library.

3.2.5 Confusion Matrix

To aid in the subsequent generation of the confusion matrix stored in each node, the indices containing the datapoints stored in the current node (stored in data above) are also used to extract the predicted and true values from the y_predictions and y_true arrays respectively, likewise with Algorithm 1.

ALGORITHM 1: Node Assignment

```
samples_in_node = np.where(tree.apply(X) == node_id)[0]
data = pd.DataFrame(X[samples_in_node], columns=feature_names)
data['Predicted'] = y_predictions[samples_in_node]
data['True'] = y_true[samples_in_node]
conf_matrix = confusion_matrix(data['True'], data['Predicted'], labels=np.unique(y_true))
```

3.2.6 Path

To efficiently determine the ruleset for each node, the path is recursively generated for each left and right child of the current node and passed into the constructor for the child when it is being made. Since the left child node contains all of the values that hold true for the current node's rule (which is always is less than or greater to) the left child's path is the current path plus the appended string " -> {condition} <= {threshold}" whereas the right child gets passed in the inverse condition " -> {condition} > {threshold}". By storing this value at every node, it is easy to determine each node's path, limiting the cognitive load that a user would otherwise need to use up to determine the path to a node.

3.2.7 Children

To recursively build the decision tree, as with other trees, the left and right children must be stored within each node, which are determined by calling create_bst_from_tree until a node id of -1 is reached within the RandomForestClassifier. estimators_.tree_.

3.2.8 Depth

While not stored explicitly in the code, since it is used to calculate the position of the x and y values, depth is calculated for each node starting from the initial first node which has a depth of 0.

3.3 Single Tree Figure Visualization

With the tree structure made, it is now possible to visualize the tree using the libraries using the create_figure_for_tree function. Within the function, the following process must take place. Firstly, in order to properly store the confusion matrices in the nodes, the predicted y_values must be calculated for a given estimator. Once this is completed, the decision tree is computed recursively, stored in the bst_root variable, and then the nodes are stored in a list that will be iterated over during the creation of the Plotly Graph Objects to be displayed in the Plotly Figure. One at a time, the relevant values of each graph object are assigned to a variable, including their position (x, y), text display value, color, and hover_text, which includes the text value of the node, its path, the number of samples it contains, and its confusion matrix. A scatter plot point is made and connected to its left and right nodes if it has any. Once the tree scatter plot is made, a title is added and all indicators that the figure was made with a scatter plot are removed, including the grid, zero lines, and tick labels, providing the appearance of a smooth tree structure.

To both enable the development and debugging of the decision tree visualizer, it was necessary to establish the general layout of the application. Therefore, one large HTTP division (div) was made for the application as a whole with sub-divs for the components of the application.

3.4 Dynamic Table and Figure Generation

With the tree structure made and displayed, now the responsive behavior based on the clicked nodes was implemented. This chronology was important because logically, it would not have been possible to create the envisioned visualizations without the nodes first appearing on the screen, as there would have been nothing to click; with this update, this new functionality could be added.

For the new functionality, it was important to track what the user had just clicked and in the case of a new node, it would update the state of the display, which includes a DataFrame.info() text, DataFrame.describe() table, the full DataFrame table, and a confusion matrix for the test dataset datapoints stored in a node. Initially, an @app.callback was implemented that would call a function every time a user clicked the graph and display solely the information in that node but in the final iteration of the graph, clicking on a node modifies the tables to consider the datapoints in this new node and all the nodes that were previously considered and displayed.

In the case that a user does not click on a node or select a group of nodes, no change occurs to the visualizations. However, in the case that a user clicks on a node or selects a group of nodes, its value(s) is added to a map and the corresponding nodes in the nodes list (from the previous section) are found. The indices of the elements contained in these nodes are stored, extracted into a DataFrame, and concatenated to another DataFrame which stores the nodes that were previously selected at some point. Once the final version of the DataFrame with all of the new updated values is finalized for the callback, a confusion matrix, DataFrame.describe() table, and DataFrame.info() text are generated below the tree visualization, and a message containing the list of all of the selected nodes thus far is displayed above the tree visualization to keep users informed about which nodes are being shown. In the event that users want to clear the list of nodes considered in the display, there is a large red Clear button which clearly signals to users that they may clear their visualizations by clicking the button. The callback and function

responsible for this process is also responsible for displaying each decision tree in the random forest model and therefore the code which generates this information will be displayed later.

3.5 Full Forest Visualization Layout

As discussed previously, all of the code and functionality described so far resulted in the generation of one tree visualization and the resultant visualizations caused by users clicking the tree figure. For the final step of the process, modifications were made to generate all of the decision trees in a Random Forest, with the intention of allowing the user to decide which individual tree to visualize. To accommodate this change, tabs were added to the layout of the application which were numbered in such a way that they corresponded to an estimator in the Random Forest. For example, clicking on the tab "Tree 1" would generate the first decision tree in the forest. A significant detail to consider is that changing the tab not only changes the decision tree being shown, but also the list of nodes above the tree and the visualizations below the tree, as each tree should be independent of the others in this respect, especially when considering that every tree has a different ruleset.

With all of the code now developed, it is now possible to describe the interactions supported within individual decision trees.

3.6 Save Data Button

The final piece of functionality added to the application includes a Save Data Button, which stores the df.info() table currently generated in a .txt file and which stores the df.describe() and DataFrame itself in a .csv file. By saving this information and the figures of the decision tree and confusion matrix, researchers can export this information to present important findings in an external setting, exemplifying yet another affordance of this system.

3.6 Plotly Interactive Behavior

Of the libraries applied to allow for the production and development of this application, Dash in particular is worth highlighting, as this is the library that enables the zoom-in and clickable functionality within the application and the library that powers the subsequent generation of figures below the graph. This is the same library that powers Plotly, which is a popular dynamic visualization library for various types of plots and figures like scatter plots and bar charts. Consequently, since the application developed here runs on the same base code as Plotly, it affords certain behavior that is similar to that in Plotly which can be accessed in the toolbar, shown in Figure 10. The following subsections will describe the behavior that is afforded by the inclusion of this toolbar. It is in large part due to the functionality of this toolbar that the application is dynamic in nature, akin to Plotly.

Figure 6: Buttons from left to right reveal the following text upon hover: 1) Download the plot as PNG, 2) Zoom, 3) Pan, 4) Box Select, 5) Lasso Select, 6) Zoom in, 7) Zoom out, 8) Autoscale, 9) Reset Axis, 10) Produced with Plotly.js (v2.20.0).

3.6.1 Download PNG

The first button that appears in the Plotly toolbar is the "Download PNG" button, with a purpose that is largely self-explanatory: given the current state of the generated tree visualization, the button will create a PNG representation that demonstrates exactly what is shown in part (b) of Figure 11. Consequently, whatever changes are made by the user to the tree visualization will be reflected in the downloaded image, allowing users to highlight and download important subsections of the tree, exporting them to an external png file.

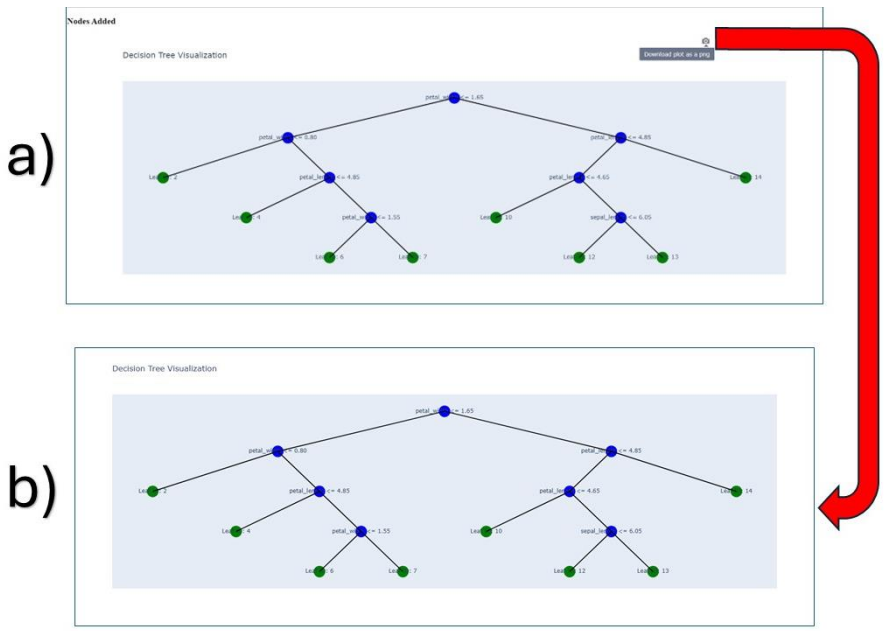


Figure 7: Demonstrates the resultant image generated (b) after clicking the "Download plot in a png" icon in the application, as shown in a).

3.6.2 Zoom

Unlike matplotlib.pytplot and dtreeviz which almost exclusively allow users to zoom into the tree visualization by a double click, clicking the "Zoom" button triggers a Zoom mode within the visualization. Within this Zoom-Mode, users are able to create bounded boxes within the current window; the edges of this window become the new bounds of the new window, providing a larger

image of the nodes in the window. The previous visualization libraries were substantially more limited in their zooming capabilities so researchers seeking to see a particular subset of nodes could easily be disappointed with the possibilities afforded to them before. Here, the window can include as few as one node and as many as all of the nodes, where a slightly more focused view of the entire tree is desired.

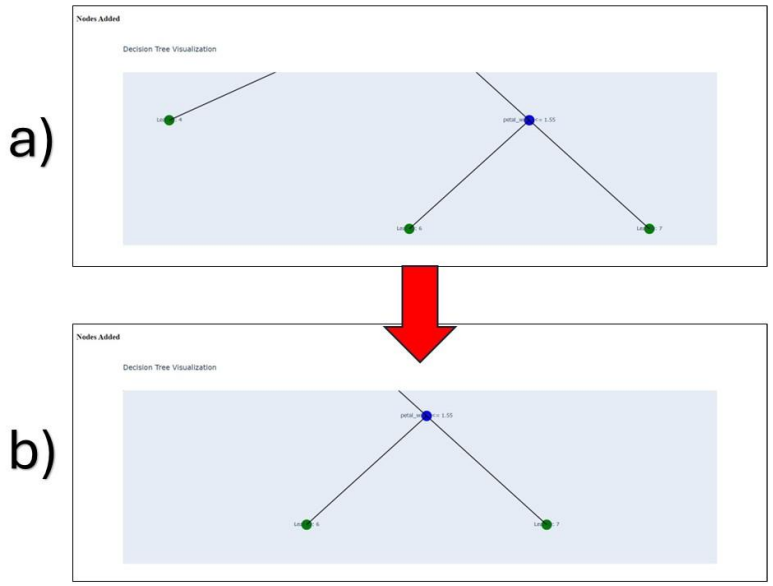


Figure 8: Demonstrates the result of applying the zoom operation with a bounded box approach.

3.6.3 Pan

For each window generated by the application software, if a user wants to move along either axis to move the center of their image, they may do so with the “Pan” button. By clicking this button, the standard Zoom mode, which allows users to make bounded boxes to zoom into, is disabled and instead navigation becomes the priority of the user. This features in itself separates itself from many existing tree visualization software applications, which only allow users to explore different parts of the tree by zooming in and out of the full-tree view, where all of the nodes are displayed. In combination with zoom, this can be a useful tool for users to move across a tree, allowing them to study different parts of a decision tree much more easily than in prior applications.

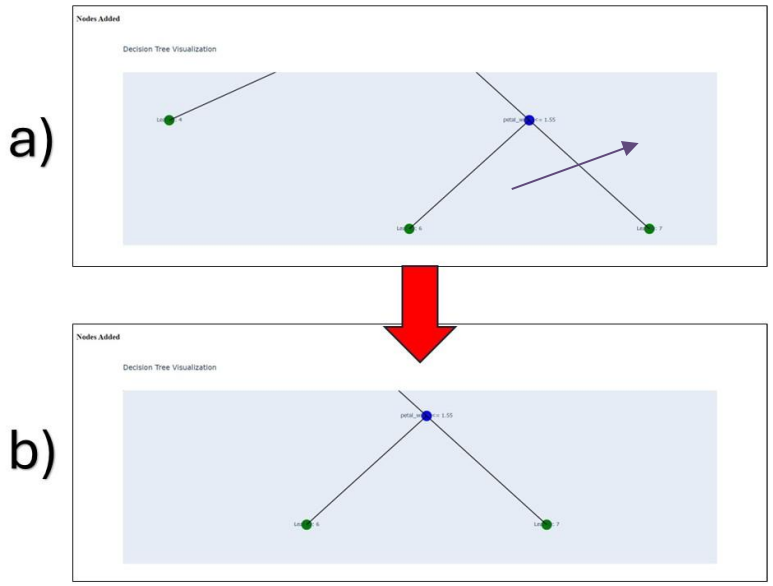


Figure 9: Showcases the resultant state of the application after using the Pan operation in the direction of the purple arrow in a).

3.6.4 Box Select

For instances where users would like to select several nodes and render their visualizations below the tree figure, they are provided this opportunity with the "Box Select" button. After a user selects this button, using the same mechanics as before for zooming in, they can form a bounded box and simultaneously select all of the nodes contained within the bounded box. Unlike the zoom option, once they make the initial box, they may adjust the area, and by extension nodes, selected. With this button, analyzing subsections of the tree become a very simple task, allowing researchers to easily study trends contained within those sections.

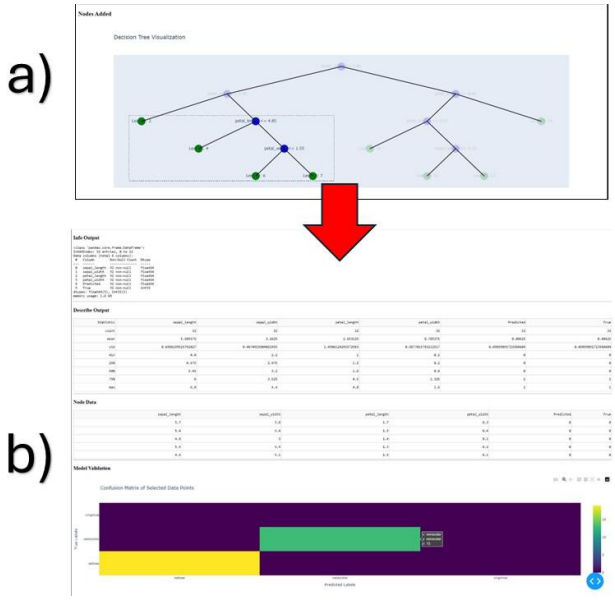


Figure 10: Part (b) shows the resultant tables and figures, including the df.info(), df.describe(), full df object, and confusion matrix for the objects selected with the Box Select tool in (a).

3.6.5 Lasso Select

For instances where users would like a greater degree of freedom to select groups of nodes without needing to click all nodes individually, they may use the “Lasso Select” button. After a user selects this button, they can make a lasso-like shape, which can be as rectangular or round as necessary, they can form a highlighted region to simultaneously select all of the nodes contained within the shaded area. Like before, this region may be adjusted to quickly modify the nodes selected within. With the Lasso Select button, researchers have the ability to select groups of nodes that may not necessarily be contained in a bounded box, providing even more convenience and flexibility to users than before.

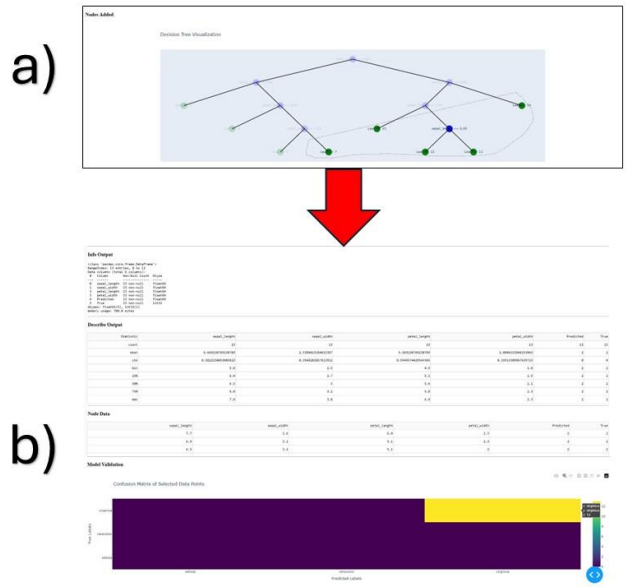


Figure 11: Part (b) shows the resultant tables and figures, including the df.info(), df.describe(), full df object, and confusion matrix for the objects selected with the Lasso Select tool in (a).

3.6.6 Zoom In

Just like how matplotlib.pytplot and dtreeviz allow users to perform a standard Zoom-in, this "Zoom in" button zooms into the center of the object, highlighting objects within the center of the screen and excluding objects that do not fall within this new window. For users who prefer button use to creating a bounded box for zooming into an image, this is an option that is available for them to make use of.

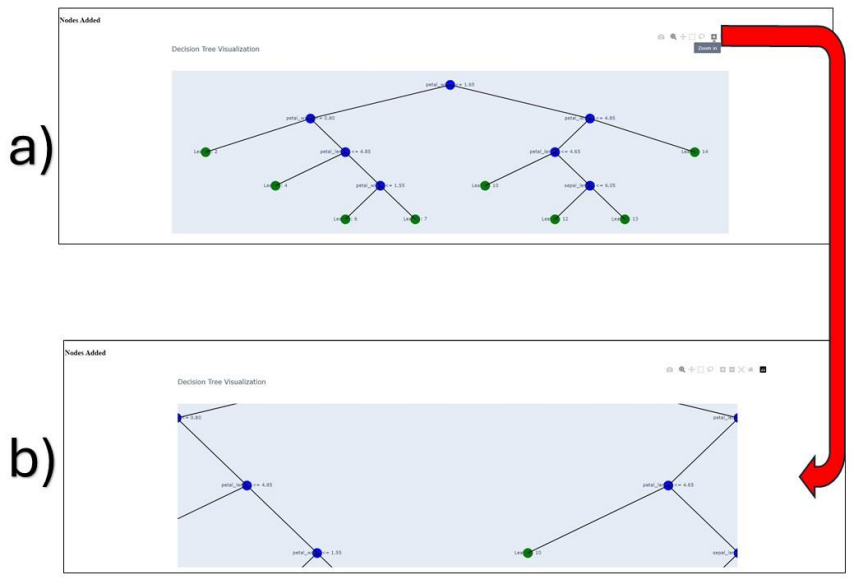


Figure 12: Part (b) reveals the impact of clicking on the "Zoom in" button on an instance of the application in part (a) of the figure.

3.6.7 Zoom Out

The “Zoom out” button naturally serves as the inverse of the “Zoom In” button previously described. If users want to undo the progress, they made by zooming into the figure, they may click this button and it will create a figure with a larger area of points covered, by extension, increasing the number of nodes displayed in the window.

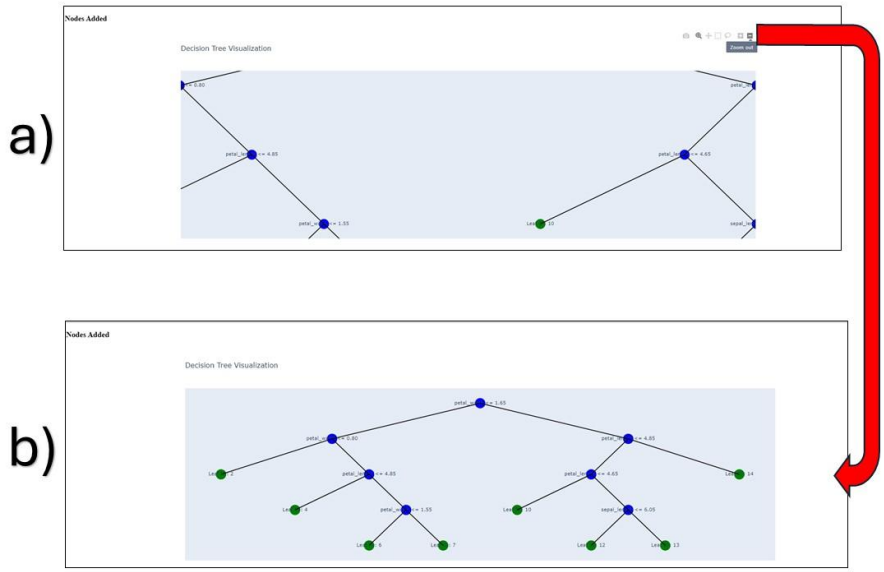


Figure 13: Part (b) reveals the impact of clicking on the “Zoom out” button on an instance of the application in part (a) of the figure.

3.6.8 Autoscale

This button reverts the progress made up until the selection of this button and reverts to the standard view, which includes a display of the entire tree, with all nodes in scope.

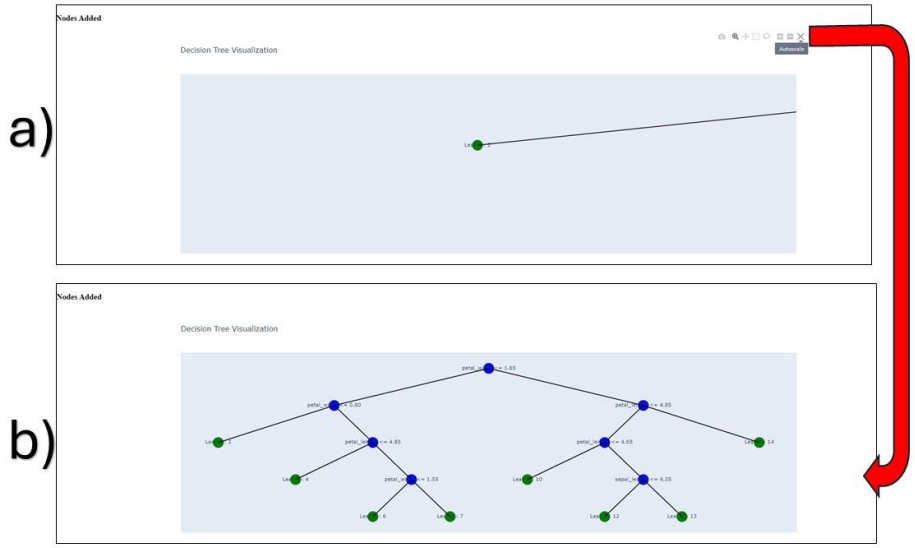


Figure 14: Part (b) illustrates the effect of clicking on the "Autoscale" button on an instance of the application in part (a) of the figure.

3.6.9 Reset Axes

For the purposes of this application, this button is identical to the Autoscale button, meaning that it also reverts the progress made up until the selection of this button and displays the entire tree.

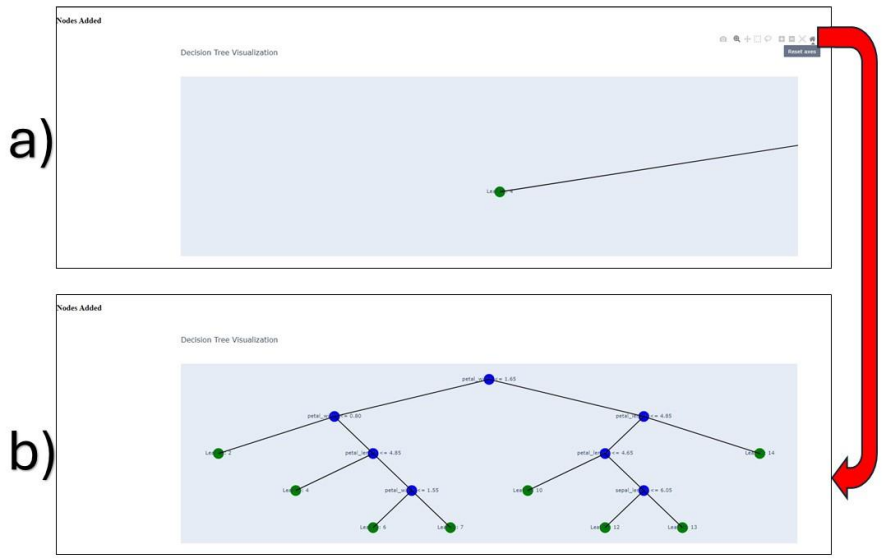


Figure 15: Part (b) illustrates the effect of clicking on the "Reset axes" button on an instance of the application in part (a) of the figure.

3.6.10 Logomark Button

Navigates to the homepage for the Plotly library, the library this application applied to achieve its desired effects.

3.8 Resultant Application Screenshots

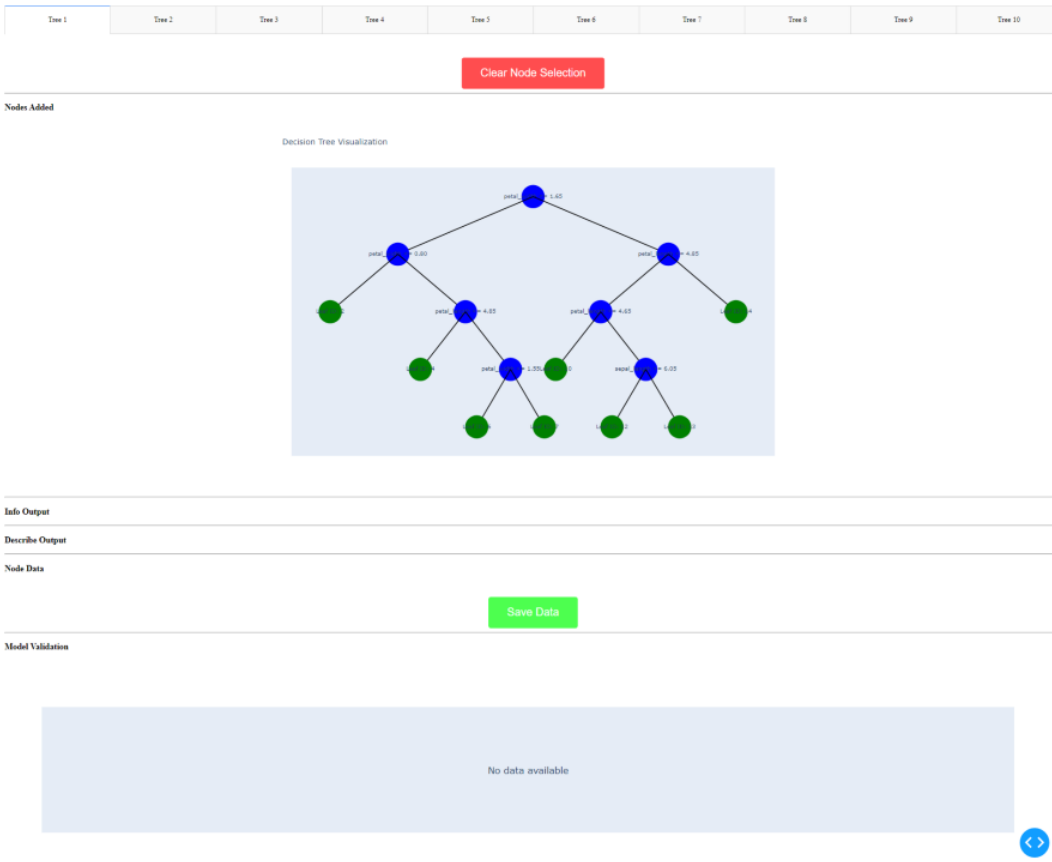


Figure 16: Reveals the state of the full application when no nodes are selected.

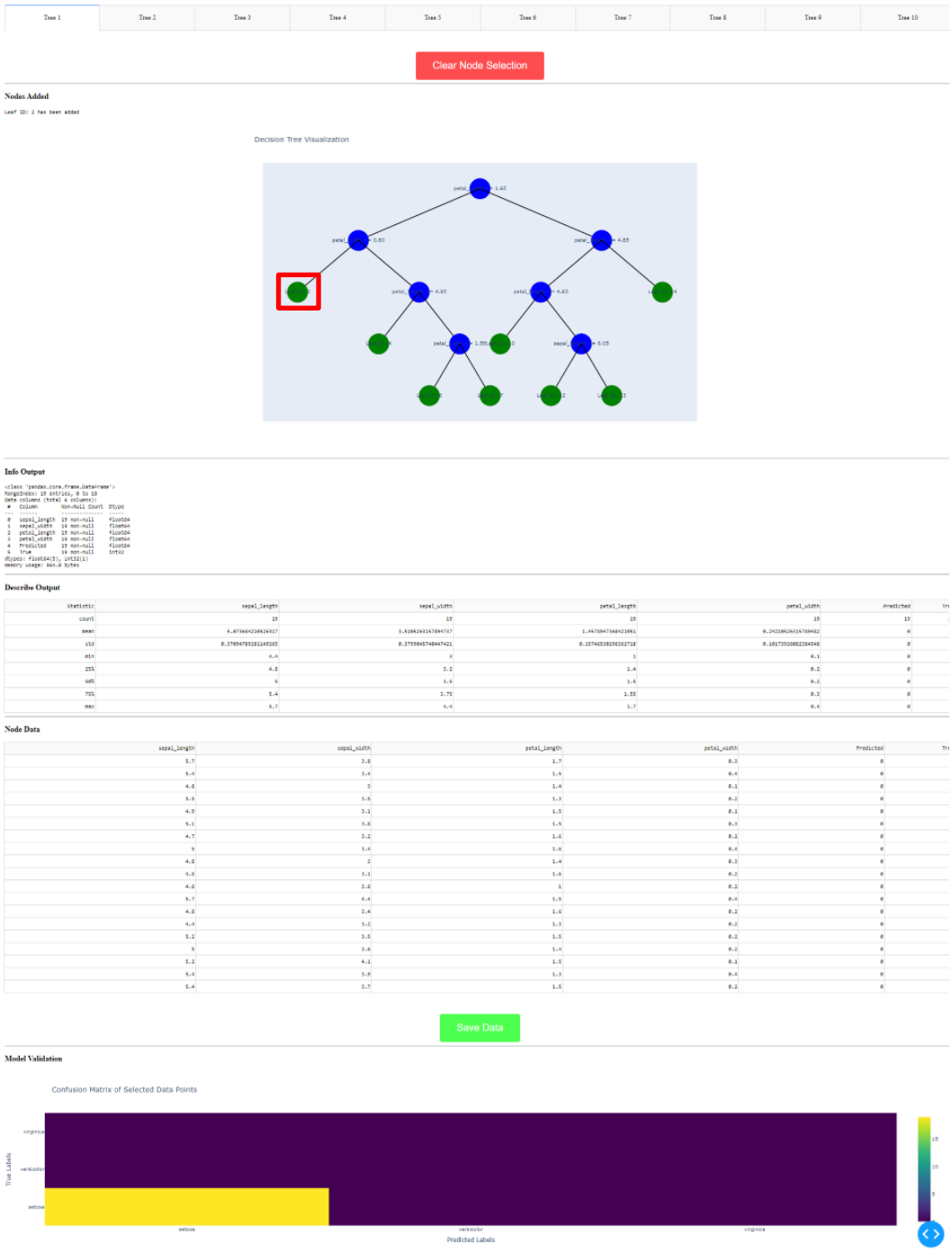


Figure 17: Reveals the state of the system when Leaf Node 2 from the first decision tree of the Random Forest is selected, the node in the visualization with the red box around it.

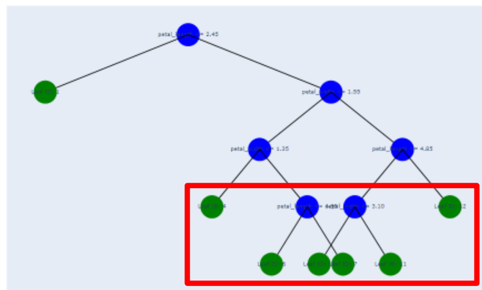
Tree 1	Tree 2	Tree 3	Tree 4	Tree 5	Tree 6	Tree 7	Tree 8	Tree 9	Tree 10

Clear Node Selection

Nodes Added

```
Leaf ID: 4 has been added petal_length <= 4.95 has been added Leaf ID: 6 has been added Leaf ID: 7 has been added sepal_width <= 3.10 has been added Leaf ID: 10 has been added Leaf ID: 11 has been added Leaf ID: 12 has been added
```

Decision Tree Visualization



Info Output

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 26 entries, 0 to 12
Data columns (total 6 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   sepal_length  26 non-null     float64
 1   sepal_width   26 non-null     float64
 2   petal_length  26 non-null     float64
 3   petal_width   26 non-null     float64
 4   Predicted     26 non-null     float64
 5   True          26 non-null     int32
dtypes: float64(5), int32(1)
memory usage: 1.3 KB
```

Describe Output

Statistic	sepal_length	sepal_width	petal_length	petal_width	Predicted	Tr
count	26	26	26	26	26	
mean	6.407692307692309	2.9192307692307696	4.984615384615385	1.7384615384615385	1.5	1
std	0.5635855115422224	0.33558001520689587	0.7682147135754732	0.39403240788231925	0.5099019513592785	0.5099019513592785
min	5.6	2.2	3.6	1.1	1	
25%	6.025	2.725	4.5	1.4249999999999998	1	
50%	6.4	2.9	4.85	1.7000000000000002	1.5	1
75%	6.7	3.1	5.5	2.075	2	
max	7.9	3.8	6.9	2.3	2	

Node Data

sepal_length	sepal_width	petal_length	petal_width	Predicted	Tr
6.1	2.8	4.7	1.2	1	
5.6	2.9	3.6	1.3	1	
5.8	2.7	3.9	1.2	1	
5.6	2.5	3.9	1.1	1	
5.7	2.8	4.5	1.3	1	
6	2.9	4.5	1.5	1	
6.8	2.8	4.8	1.4	1	
6.2	2.2	4.5	1.5	1	
6.7	3.1	4.4	1.4	1	
6.4	3.2	4.5	1.5	1	
6.7	3.1	4.7	1.5	1	
6.3	3.3	4.7	1.6	1	
6	3.4	4.5	1.6	1	
7.7	2.6	6.9	2.3	2	
6.9	3.1	5.1	2.3	2	
6.5	3.2	5.1	2	2	
6.5	3	5.8	2.2	2	
6.4	2.8	5.6	2.2	2	
6.1	3	4.9	1.8	2	
6.4	2.8	5.6	2.1	2	
7.9	3.8	6.4	2	2	
6.7	3	5.2	2.3	2	
6.7	2.5	5.8	1.8	2	
6.8	3.2	5.9	2.3	2	
6.3	2.5	5	1.9	2	
5.8	2.7	5.1	1.9	2	

Save Data

Model Validation

Figure 18: Reveals the state of the full system when the Box Select tool is applied on the nodes within the red bounded box are selected.

4 FUTURE WORKS

This library was designed with the intention of addressing random forest classification problems, which is made clear by its inclusion of the confusion matrix functionality. If the model were to be extended to random forest regression models, this last figure could be replaced with a scatter plot detailing the Mean Squared Error (MSE) of points, highlighting how it differs between certain points of the dataset. Alternative metrics may be considered, such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), or R-squared (R^2).

Additionally, creating an additional interface to directly compare several trees against one another, considering variables like trees diversity, depth, similarity, importance of features across could also be of value to consider an implement. While this would fall in line more with what software like SHAP offers, exploring how to incorporate metrics like entropy between trees could provide researchers with even better context regarding how trees relate to one another. Since the timeframe of this project was only one semester, these topics were only glossed over and the features present in the software currently were prioritized.

Finally, as mentioned during the position assignment section for the Node structure, certain tree visualization libraries use advanced placement algorithms to ensure no edges nor nodes overlap with one another. For this project, the former goal was successful, but the latter was not. Hence, in the future, implementing an even more advanced placement algorithm would result in an even cleaner visualization of the tree, especially for very deep trees.

5 CONCLUSIONS

Due to the largely explainable nature of random forest and decision tree classifiers, they can provide significant insight to researchers applying them. Current approaches relying on largely static visualizations have a difficult time adjusting to deep trees with a large number of nodes and provide minimally useful information in assessing the performance of testing data and it was these points of concerns that motivated the writing of this paper. By taking inspiration from existing libraries used for scatter plots and charts like Plotly and Dash, this paper introduced a new type of dynamic visualization that affords both studying individual decision trees and comparing different decision trees generated by Random Forest Models. To study individual and groups of nodes of a single decision tree, this library allows users to zoom into nodes of interest using a bounded box technique and allows users to study the datapoints stored in an individual node or a set of nodes through two different selection methods by subsequently generating tables and figures based on these nodes. As this functionality is offered for all of the trees making up a random forest, it is possible to compare and contrast trees simply by switching between the tabs of the interface. While there remain areas for future work, including adding functionality for random forest regression models and including more advanced features and metrics within display data, the development of new dynamic random forest visualization application has been successfully prototyped in the Python programming language.

ACKNOWLEDGMENTS

This work was completed with the guidance and supervision of Professor Albert Ritzhaupt, Professor Emmanuel Dorley, and Professor Rong Zhang, professors at the University of Florida. Without their words of advice and pointers, this work would look very different than it does right now. Their great motivation, kindness, and words of wisdom helped lead me in the right direction with the paper, leading to a project that I am quite proud of.

I would also like to thank Professor Aleksandra Karolak, my former supervisor at the Moffitt Cancer Center for agreeing to take a look at my application and for suggesting ideas for future exploration, as these featured very heavily in my Future Works section. My work with Professor Karolak at my time at the Moffitt Cancer Center greatly inspired the work that I completed for this Honors Thesis and so I would like to thank her again for supervising me while I was at Moffitt, as that experience continues to play an important role in my professional and academic development.

I would be remiss not to mention my family, especially my parents, Grzegorz and Izabella, and my brother Alexander. I thank all three of them for their love, support, and words of encouragement in my professional and academic endeavors, including this Honors Thesis. I would also like to give a special shoutout to Alexander for agreeing to proofread my paper and giving me pointers on what to fix and modify.

REFERENCES

[1] H. Cheng, P. S. Yu, and B. Liu. 2010. On the Importance of Comprehensible Classification Models for Protein Function Prediction. ACM Trans. Knowl. Discov. Data 4, 1 (Jan. 2010), 1–28. DOI: https://doi.org/10.1145/41425089

[2] S. Marsland. 2014. Machine Learning: An Algorithmic Perspective (2nd ed.). CRC Press, Boca Raton, FL, USA.

[3] M. N. Hoque, S. Shin, and N. Elmqvist. 2024. Harder, Better, Faster, Stronger: Interactive Visualization for Human-Centered AI Tools. arXiv preprint arXiv:2404.02147 (2024). DOI: https://doi.org/10.48550/arXiv.2404.02147

[4] B. Shneiderman.Human-centered artificial intelligence: Reliable, safe & trustworthy.*Int. J. Hum. Comput. Interact.*, 36(6):495–504, 2020. doi: 10.1080/10447318.2020.1741118

[5] B. Shneiderman.*Human-Centered AI*.Oxford University Press, Oxford, United Kingdom, 2022.

[6] S. Solanki. 2020. Treeinterpreter: Interpreting tree-based model's prediction of individual samples. (October 2020). Retrieved November 30, 2024, from https://coderzcolumn.com/tutorials/machine-learning/treeinterpreter-interpreting-tree-based-models-prediction-of-individual-sample#google_vignette

[7] A. G. P. Salama. 2018. Visualizing Decision Trees. Explained.ai. Retrieved November 30, 2024, from https://explained.ai/decision-tree-viz/

[8] M. A. Mohedano-Munoz, L. Raya, and A. Sanchez. 2024. Guided Decision Tree: A Tool to Interactively Create Decision Trees Through Visualization of Subsequent LDA Diagrams. Appl. Sci. 14, 22 (2024), 10497. DOI: https://doi.org/10.3390/app142210497