Time Series Analysis Support for Data Scientists
Software Design Specification
by Keyboard Warriors

Table of Contents

1. SDS Revision History

CJ = Cameron Jordal
JK = JT Kashuba
NK = Noah Kruss
NT = Nick Titzler
RV = River Veek

Date	Author	Description
1/18/21	NK + JK	Initial creation of document using template provided by Prof A. Hornof
1/19/21	NK + JK	Worked on Software Architecture and Software Module sections
2/6/21	NK	Worked on Software Module section for Tree class
2/9/21	NK	Worked on Software Architecture section
2/9/21	CJ + NT + RV	Worked on Section 3 subsections (Software Architecture)
2/9/21	JK	Edited Section 3 (Software Architecture)
2/9/21	CJ + NT + RV	Worked on Section 4 subsections (Software Modules)
2/9/21	NK	Worked on architecture description for Section 3
2/10/21	NT	Edited 4.3 design rationale
2/10/21	JK	Finalized Section 5 Dynamic Models of Operational Use
2/10/21	JK	Proofread and Edited Entire Document for Verbiage/Typos
2/10/21	RV	Fixed spelling/grammar mistakes
2/10/21	NT	Fixed spelling/grammar mistakes
2/10/21	CJ	Fixed spelling/grammar mistakes

2. System Overview

This system is a code library that provides functionality for processing time series data—specifically cleaning it up, displaying different forms of output, and producing/applying forecasting models, all in the form of a data processing tree. The Time Series Tree (henceforth referred to as a TS_Tree) stores references of the functions the data will be processed through after calling an execute_tree or execute_path command. Each path in the tree can also be thought of as a “pipeline”, where a sequence of functions from the root to a specified leaf node are carried out on a chosen time series.

3. Software Architecture

Set of Components:

1. Time Series (TS) Tree
    a. System for storing, organizing, and executing data function calls to be executed on time series data that is passed through the tree.
2. Preprocessing Module
    a. Grants the user a library of functions that can manipulate data sets in various ways.
3. Visualization Module
    a. Allows users to plot time series data with standard plots, histograms, and boxplots. It also allows users to find the mean standard error (MSE), the mean absolute percentage error (MAPE), and the symmetric mean absolute percentage error (SMAPE). Lastly, the user can conduct normality tests on the data.
4. Forecasting Module
    a. Allows users to create a multi-layer perceptron regression model to forecast future values in a time series.
5. File IO Module
    a. Gives users the ability to read in .csv files with any number of headers, and write out .csv files.

Design rationale and explanation of tree architecture:

The system was designed with the goal of the end product being flexible for users and programmers alike. Users are able to use the functionality of a TS_Tree object with data from different time series, while still making it simple to add additional functions to the system if they want to update a path within the tree (or many branches within the tree) in order to produce different outcomes on their given time series data sets. To accomplish this goal, our TS_Tree does not store any time series information. Nodes in the TS_Tree store references to functions that are only applied to the data once an execute_path or execute_tree command is invoked.

The reasoning behind these design choices was to enable a single TS_Tree to be used for any number of different time series databases, as well as enabling users to change/update the sequence(s) of functions within their TS_Tree without forcing them to create new TS_Trees from scratch, thereby improving the system's user-friendliness. Our goal of simplicity was further developed through a design decision to have a uniform type of node that can be added to the TS_Tree. The user specifies the function they want a node to store (along with the specific input parameters of the chosen function); from a high-level view, nodes are handled by the tree in the same way.

With how the system is designed, the functions that are referenced by nodes are invoked once an execute command is called. The execute_path command passes a reference to a file (which contains the time series data that a TS_Tree will be processing) along a user specified path, where each node returns an updated version of the data depending on which function was referenced in that node. This return value is then automatically passed as an argument to the next nodes function, with no effort from the user. In making this design choice, our system was left with a much smaller overhead in terms of memory storage.

Note: A node's input_parameters have default values of None, so users will need to assign values to the specific arguments they use (which depends on the function). More detail on each function's specific input_parameters at the bottom of the README.md in the section "Some helpful examples at a glance." Validity checks confirm that the user is passing the correct input_parameters for the specific function referenced by a given node.

Another design choice was made to increase modularity. If nodes were married to a specific function and the tree was assembled with these function-nodes, the modularity would be much less forgiving for future developers. Due to the way the system is structured, it is quite simple for developers to increase functionality of the application. All that needs to be done is to write a new function in its corresponding module (ex: preprocessing) and users will be able to call add_node with the newly implemented function.

Additionally, the output of an execute_tree command is a dictionary object that has the outputs of each path within the tree. Each path returns a (most likely altered) time series - and the time series that is returned is placed in the dictionary with the key:value pair represented as the following:

key:value = leaf node:result of that path

The goal for storing output in this method is to provide a concise storage system for the user to easily compare outputs from different paths. In doing so, users are able to see how differing sequences of function calls produce different permutations of time series data in order to have a well-rounded analysis of the time series data.

4. Software Modules

4.1 - Tree Module

4.1.1 - Time Series (TS) Tree

Role and Primary Function

This module is the primary one the user will access through importing the project library. The TS_Tree provides the user the interface system for storing data, organizing data, making operation calls, and creating/executing pipelines. The users will create a TS_Tree, add/replace nodes, and copy/add subtrees to assemble the TS_Tree, and then perform an execute command to pass time series data through the tree and perform the given operations on the data for preprocessing, visualization, forecasting, and/or file I/O.

Interface to Other Modules

TS Trees are created by the data scientist (user) and consist of Operation Nodes. The user has a variety of functions for editing and executing TS Trees which include the following.

TS Tree methods:

- __init__(self)
- print_tree(self)
- add_node(self, operation, parent_index, data_start = None, data_end = None, increment = None, perc_training = None, perc_valid = None, perc_test = None, inpute_filename = None, output_filename = None, m_i = None, t_i = None, m_0 = None, t_0 = None, layers = None)
- replace_node(self, operation, parent_index, data_start = None, data_end = None, increment = None, perc_training = None, perc_valid = None, perc_test = None, inpute_filename = None, output_filename = None, m_i = None, t_i = None, m_0 = None, t_0 = None, layers = None)
- execute_pipeline(self, node_index)
- execute_tree(self, node_index)

Functions that operate on TS Tree:

- copy_subtree(tree, node_index)
- add_subtree(tree, node_index, sub_tree)
- copy_path(tree, node_index)
- save_tree(tree, save_file_name)
- load_tree(tree, save_file_name)

A Static Model

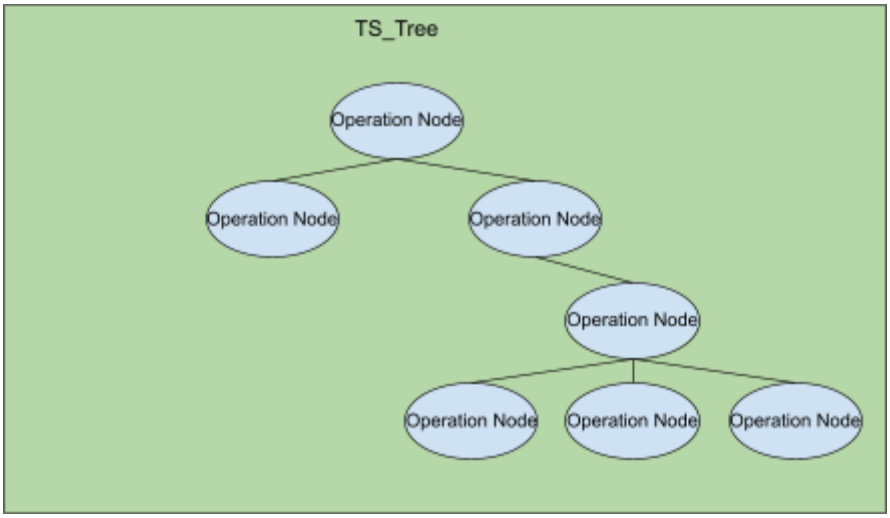

Figure 1

Design Rationale

We designed our TS_Tree class to be composed of a single type of node object. Our reasoning for this was that we wanted to keep our tree class to be as generic as possible in order to support easy integration and adaptation to the project requirements of the data scientist or user.

4.1.2 - Operation Nodes

Role and Primary Function

A hidden object for storing tree structure information (parent, child_list), reference to data function calls, and the connected inputs for said function call within a list.

Interface to Other Modules

Operation nodes are created, edited, and accessed solely through TS Tree operations. This is done by the TS Tree passing a string representation of what operation function the node is supposed to hold, along with values for the function inputs to go along with the operation. The node then takes the function string and uses it as the key to a global operation dictionary. The value in the dictionary that relates to the key will give the node the reference of the operation function to store.

Stores reference to function calls to operations within:

- Preprocessing module
- Forecasting module
- Visualization module

Stores inputs for function calls within the node (input parameters / function_inputs):

- data_start (float)
- data_end (float)
- increment (float)
- perc_training (float)
- perc_valid (float)
- perc_test (float)
- input_filename (str)
- output_filename (str)
- m_i (float)
- t_i (float)
- m_0 (float)
- t_0 (float)
- layers (list)

A Static Model

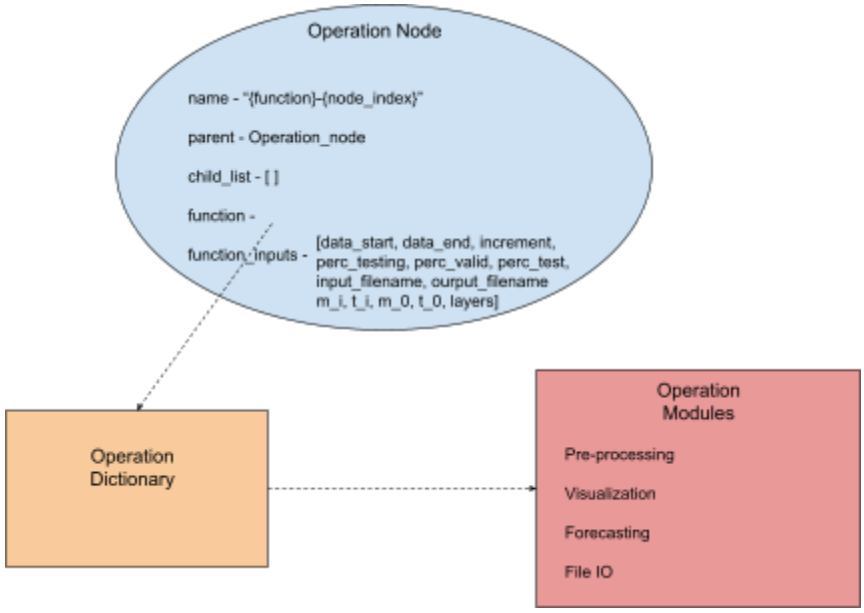

Figure 2

Design Rationale

We designed the Operation Nodes to store a reference to the function call that should be performed when executing a pipeline that includes the node. By storing a reference to the function call, as opposed to actually making a call to the function, the node is able to handle different data being passed through the tree. Additionally, this helps decrease the processing requirements of our system because function calls only get executed when execute_tree is called.

As for storing a list of all of the possible inputs a function could take, our reasoning was to make our Operation Node class be generic. In turn, we had our nodes be able to handle any function that exists within Preprocessing, Visualization, and Forecasting modules. Another benefit of designing the nodes in this method is the ease of editing the inputs that an operation takes, along with making our system able to be easily edited to handle new operations a data scientist might request in the future. For example, to handle a new operation the only changes that would need to be made to our code would be a new input within our global operation dictionary to store the new operation function reference and the index of the inputs the operation will take.

4.2 - Preprocessing Module

Role and Primary Function

The preprocessing module consists of 15 different functions, each of which grants the user a multitude of options for manipulating sets of data. When utilizing the functionality of the main tree module, the user can accept data from a file, create a pipeline of various preprocessing functions, manipulate the data, and then write the new data to a file.

Interface to Other Modules

The following functions can be added to the TS tree in the form of Operation Nodes in a particular pipeline; each function can also be called individually by importing the preprocessing library of functions.

- denoise(ts)
- impute_missing_data(ts)
- impute_outliers(ts)
- longest_continuous_run(ts)
- clip(ts, data_start, data_end)
- assign_time(ts, data_start, increment)
- difference(ts)
- scaling(ts)
- standardize(ts)
- logarithm(ts)
- cubic_root(ts)
- split_data(ts, perc_training, perc_valid, perc_test)
- design_matrix(ts, data_start, data_end)
- ts2db(input_filename, perc_training, perc_val, perc_test, data_start, data_end, output_filename)
- db2ts(db)

References to these functions are made in the operational nodes module; actual calls are made in the execute_path() function.

Design Rationale

Early on in the design process, the decision was made to make all preprocessing functions return a time series in the form of a Pandas DataFrame or database. This decision made the act of adding pipelines to the tree a much more streamlined process, with the output from one function becoming the input for the next.

4.3 - Visualization Module

Role and Primary Function

The role and primary function of the visualization module is to plot time series data. The time series data can be plotted in the form of a standard plot, boxplot, or histogram. Additionally, in this module there are functions to perform normalization tests, calculate mse, smape, and mape.

Interface to Other Modules

The visualization model interfaces with the Tree module. Although the forecasting model can be called at anypoint, it is only executed with the execute_tree function.

- plot(ts, input_filename)
- histogram(ts, input_filename)
- box_plot(ts, input_filename)
- normality_test(ts)
- mse(y_test, input_filename)
- mape(y_test, input_filename)
- smape(y_test. Input_filename)

Design Rationale

The design is made to be as simple as possible, using the built in functions that come standard with the pandas and numpy libraries. The plotting functions utilize the built in functionality in pandas dataframes, combined with simple uses of the math library for mse, mape, and smape. Initially, we did not include functionality that would allow for the plots to be saved, this functionality was added afterward.

Additionally, the mse, mape and smape functions take a database and a file name so that the tree would not need to store excess information. This allows our tree to run with greater efficiency.

4.4 - Forecasting Module

Role and Primary Function

The role of this module is to provide the user with a means by which to generate models of the time series from which future values can be predicted. This is currently done with a multi-layer perceptron model which accepts a database of time series data, an object that is divided into two matrices: one containing current states and the other containing corresponding future states. The forecast is a predicted set of future states.

Interface Specification

Only ways that the mlp models can be called is through model and forecast. The model function creates a model based on a set of training data. The size and quantity of hidden layers can also be defined. The window size also allows the user to adjust the interpolation of numbers from a provided scale to a uniform one.

- mlp_model(train, layers, window_size)
- mlp_forecast(model_data, input_filename)

Static and Dynamic Model

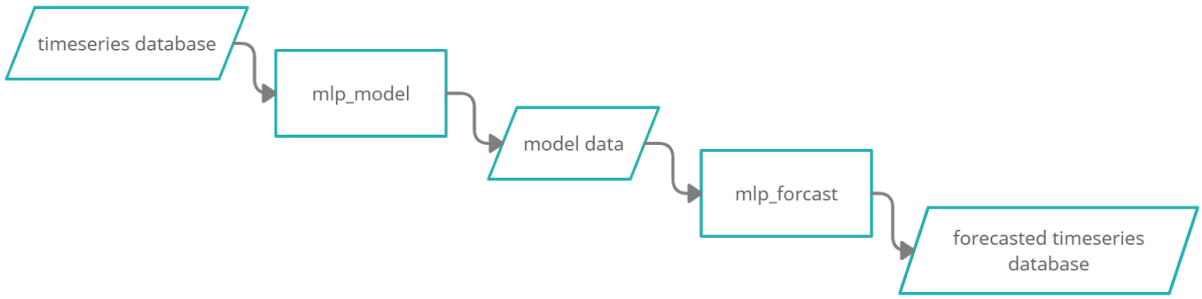

Figure 3

Design Rationale

The modeling was designed in this way to provide a simple interface from which modeling and forecasting can be done. It is assumed that a user does not want to interact with all the components and would rather just specify a few parameters. Thus, the creation of a model of a certain size, trained on a given set of data, and interpolated from a given window size are all combined into a single function with default values of model size and interpolation. This alleviates unnecessary operations from the user. Forecasting is just as easy. The mlp_forecast function takes the model data (with a mlp model as well

as a window to de-interpolate the data) and a data file containing current states to forecast from.

Alternative Designs

For future improvements, the model could allow for many more parameters or adjust the kinds of interpolation used (linear vs logarithmic). Parameters for the learning rate of training the model, solver types, activation functions, and many more could also be added. The forecasting function could also take database entries that have not yet been written to files as well.

4.5 - File IO Module

Role and Primary Function

The primary role and function of the file I/O module is to read in datafiles, and write out data files in a .csv form. The module has the ability to read in datafiles with any number of header columns, so long as they are in a .csv format.

Interface to Other Modules

This module is always first in the tree. It represents the first node that is added. It creates a time series object that interacts with all subsequent functions that are called.

- read_from_file(input_filename)
- write_to_file(ts, output_filename)

Design Rationale

The design choices for file io were meant to make the module as simple as possible, while providing the necessary functionality. Initially the team wrote our own code to read in datafiles, but after experiencing trouble with managing the variety of possible inputs, we fell back on using the pandas built in functionality.

The read_from_file is written to accept both pickle files, and csv files. Instead of trying to determine from the string what the file type is, we used a try, except block. It first tries to open the file as if it was a csv, if this errors, it will attempt to open it as a pickle. If this also fails, then it will inform the user of this, then close the program.

5. Dynamic Models of Operational Scenarios (Use Cases)

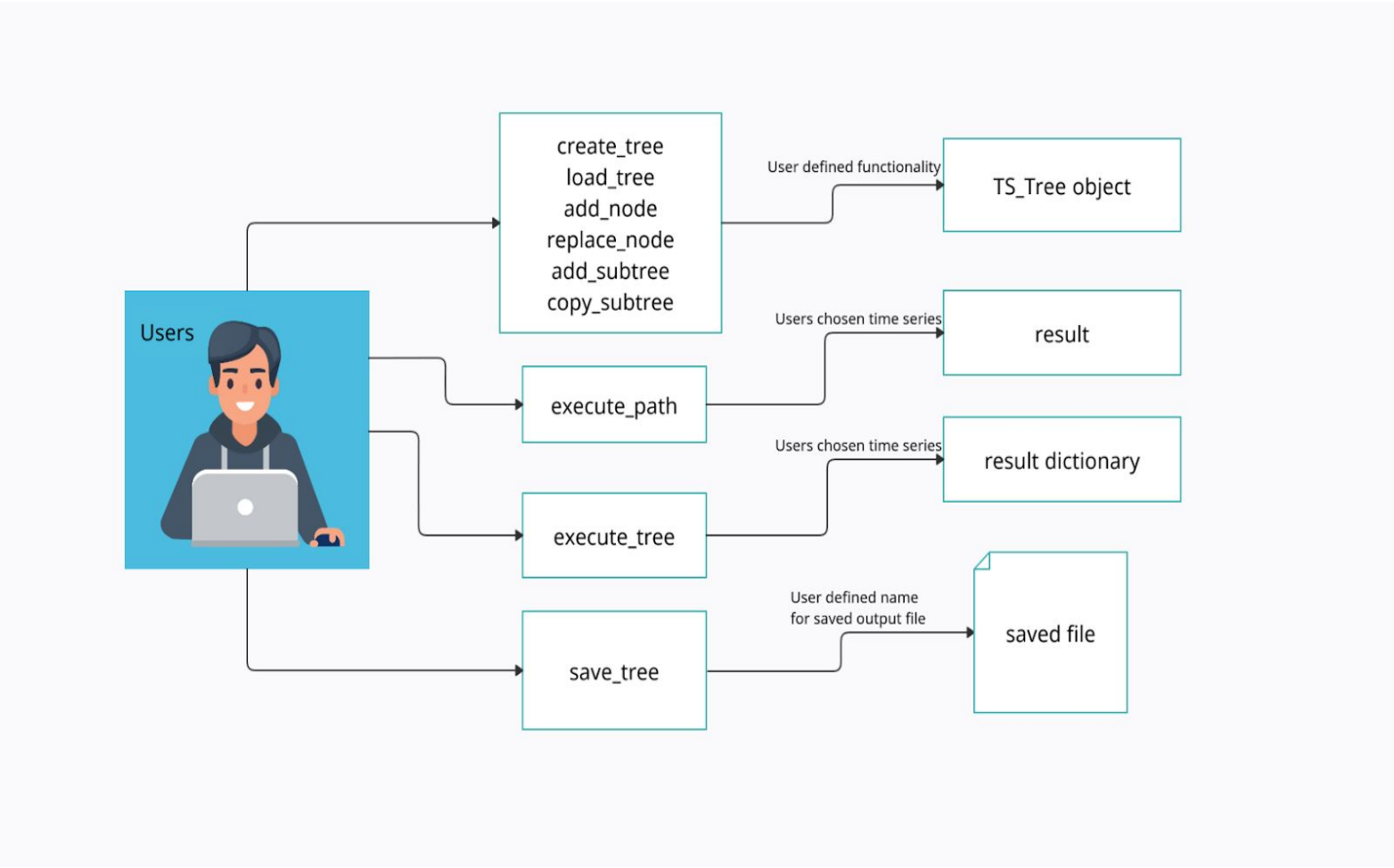

Figure 4

Figure 4 provides a dynamic model of every Tree Operation Use Case for this system that a user can invoke when constructing trees. From top to bottom:

1) create_tree/add_node (etc.) - Here the user is able to define the functionality of the nodes in their tree. After invoking any of these Tree Operations, the user will have an updated TS_Tree object that reflects their functional goal.
2) execute_path - Here the user is able to execute a path in a given TS_Tree from the root to a specified leaf node on a time series of their choice. (For detailed info on the result of execute_path, see SRS Section 2.6 - Use Case 5 “Post-Condition”)
3) execute_tree - Similar to #2 above, execute_tree will invoke execute_path for every leaf node in a TS_Tree. The difference here is that the result will be a dictionary. (For detailed info

on the structure of this dictionary, see SRS Section 2.6 - Use Case 6 “Post-Condition”)
4) save_tree - Here the user is able to save the structure of a tree in a .txt file.

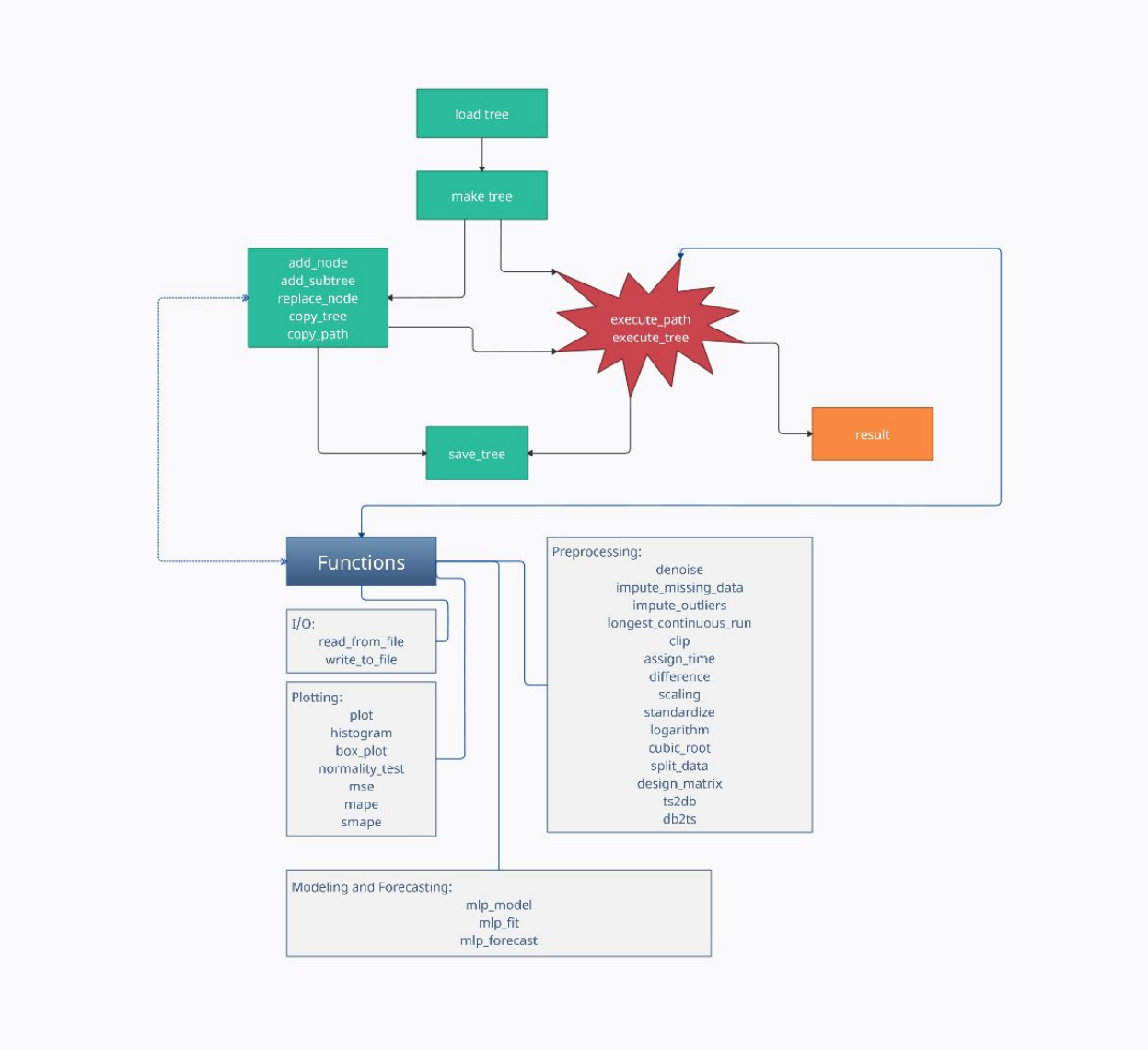

Figure 5

Figure 5 demonstrates a static model of the system architecture. *Functions* is connected to two boxes to illustrate that function calls are *referenced* in nodes created by the user (dotted line), but aren’t actually invoked / called until execution (solid line). The user may also save a tree at any point in the construction process, either before or after execution.

6. References

Faulk, Stuart. (2011-2017). CIS 422 Document Template. Downloaded from https://uocis.assembla.com/spaces/cis-f17-template/wiki in 2018. It appears as if some of the material in this document was written by Michal Young.

IEEE Std 1016-2009. (2009). IEEE Standard for Information Technology—Systems Design—Software Design Descriptions.
https://ieeexplore.ieee.org/document/5167255

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Commun. ACM, 15(12), 1053-1058.

7. Acknowledgements

The SDS template was provided by Juan Flores, who updated it from Anthony Hornof for CIS 422.