Time Series Analysis Support for Data Scientists
Software Design Specification

E. Petersen (ezekielp) - 2-8-2021 – v1.6

Table of Contents

1. SDS Revision History

This lists every modification to the document. Entries are ordered chronologically.

Date	Author	Description
1-22-2021	ezekielp	Added initial document to repo, will modify template for project
1-23-2021	ezekielp	Added first draft of preprocessing section to module section
1-30-2021	ezekielp	Added first draft of System Overview, updated 4.1, added 4.2
1-31-2021	ezekielp	Completed rough draft of sections 4 and 6, still need diagrams
1-31-2021	ezekielp	Added diagrams for section 5 and 4.1-4.3
1-31-2021	ezekielp	Added the rest of the diagrams and a draft of section 3
2-1-2021	ezekielp	Minor grammatical edits
2-6-2021	ezekielp	Updated function signatures, just waiting for ts2db/ tree methods
2-8-2021	ezekielp	Finalized the last function signatures

2. System Overview

The Transformation Tree discussed here is a tool for Data Scientists for evaluating and executing Time Series pipelines. Consisting of a library of functions for manipulating such n-ary trees, the software is built around nodes representing common steps used by Data Scientists when processing Time Series data. These steps are primarily broken down into preprocessing/IO, modeling, and visualization/evaluation modules, with each representing one or more secondary classes of nodes to further assist organization and type checking within the tree.

The software allows a Data Scientist to create a tree, add processing steps in the form of nodes, replicate entire trees, subtrees, or pipelines, and save and load pipelines or trees. It also allows for the replacement of one step with another of the same class and ultimately "execute" a tree or pipeline.

3. Software Architecture

The software architecture is built around three primary modules consisting of common logical steps taken by Data Scientists when analyzing Time Series. These modules are largely separated by their role in the analysis process and a brief description of their assigned functionality are as follows:

1. Preprocessing: Responsible for file I/O, cleaning up or manipulating raw TS data, and for splitting the data into train, validation, and test sets.
2. Modeling: Responsible for forecasting future measurements based on the training and validation sets.
3. Visualization and Evaluation: Responsible for plotting TS data and for evaluating the accuracy of the forecasted data by comparing to the test set.

The three components help define five primary node classes that make up the Transformation Tree, divided further based on roles for each function within the modules (shown in Figure 3.1). The classes are as follows:

1. prepNode: Holds information needed to clean up or otherwise manipulate raw TS data.
2. splitNode: Holds information needed to divide the TS data into sets.
3. modelNode: Holds information about which ML model will be used to forecast data.
4. visualizeNode: Holds information needed to plot TS data.
5. evalNode: Holds information about what method will be used to evaluate accuracy.

Together, the nodes define the contents of a Transformation Tree, which has methods necessary to do computations and I/O required for a pipeline analysis. These tree methods are the primary API for the Data Scientist. Prior to execution, the tree merely holds the arguments passed to it from the Data Scientist along with the names of the functions from the main modules that will be called.

The overall architecture was deemed appropriate mainly because it factors out information and allows for easy type checking on user input. The modules allow for the logic of TS analysis to be separated from the Transformation Tree logic and to be used as a library of functions. The node classes further allows for type checking so process steps are guaranteed to be in a legal place in the tree. Additionally, making the API consist only of tree methods (with few exceptions) limits the access that the Data Scientist has so type checking can be restricted to one place. Finally, with legitimate node operators further factored out into a table and node execution methods drawing on that table, it allows increased flexibility for adding future operations.

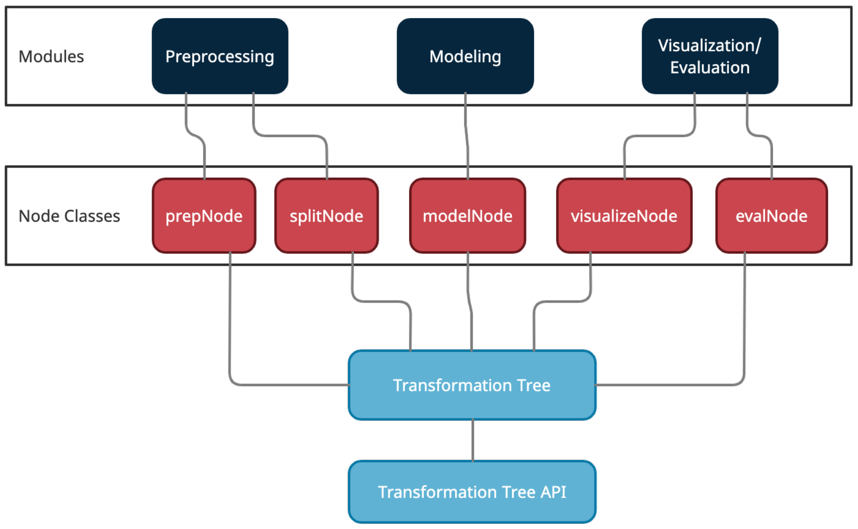


Figure 3.1 *Software architecture diagram. Describes the types of nodes representing the three primary modules and how the tree consists of these nodes. Methods on the tree are the functions accessible to the Data Scientist.*

4. Software Modules

4.1. Preprocessing

4.1.1. Role and primary function.

The preprocessing module is a library of functions that will be used as operators in the first layers of a Transformation Tree. Most functions will take raw TS data in the form of a DataFrame object (with perhaps other inputs) and return a modified version in the same format. The module allows for several preprocessing steps to be taken in a row once inserted into a tree.

This module also contains the data splitting functionality (into training, validation, and test sets) used prior to passing data to the modeling nodes. Only one data split step can be present in any one tree path.

The preprocessing module also contains the simple functions to read and write to a .csv file either to or from a DataFrame object. These functions will be instrumental in many of the functions making up the API.

4.1.2. Interface Specification

The following functions will be made available as operators to the prepNode or splitNode class. Individual function descriptions will be included as a comment in the appropriate file.

- denoise(ts)
- impute_missing_data(ts)
- impute_outliers(ts)
- longest_continuous_run(ts)
- clip(ts, starting_date, final_date)
- assign_time(ts, start, increment)
- difference(ts)
- scaling(ts)
- standardize(ts)
- logarithm(ts)
- cubic_roots(ts)

- ts2db(input_file_train, input_file_test=None)

4.1.3. A Static Model

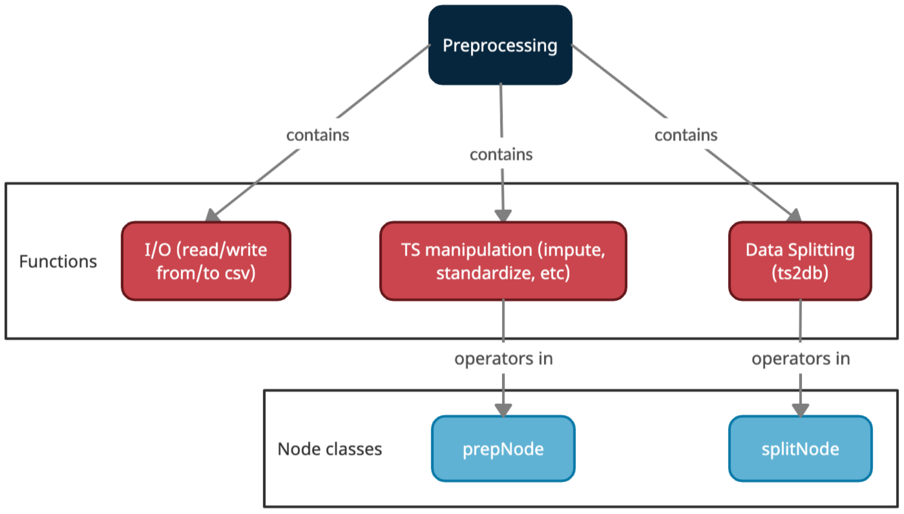


Figure 4.1 *Diagram of the preprocessing module's relationship to the tree*

4.1.4. Design rationale

The preprocessing module was designed in order to factor out the logic behind the execution of the pipeline. We want all of the standard functionality already used by Data Scientists available, but we wish to treat them as operators of the nodes of the Tree. The library design allows us to swap in and out the various functions and leave the Tree organization to its own module.

This was a fairly easy design decision to make with no other obvious alternatives as the functions included were required to be present in the final product and already separated into its own "module" in the project description. We could have placed all required functions, including those from the Modeling and Visualization/Evaluation into one large module, but it made more sense to split up according to function for later separation when designing the node structure within the tree.

4.1.5. Alternative designs

There were discussions of breaking up the module further into three parts with the intention of separating the IO and data splitting aspects of the preprocessing function set, however, since both of these pieces were fairly small and their functionality seemed to fall into the "preprocessing" archetype, it was deemed unnecessary. Originally, there was only going to be one node class per module, but due to the vastly different arguments of the splitting functions, we decided to factor those steps further.

4.2. Modeling

4.2.1. The module's role and primary function.

The Modeling module makes up the middle layers of the Transformation Tree and is the primary logic for the system. It takes in pre-split data from nodes of the preprocessing module and creates forecasts based on that data. This is later passed to the last nodes in the tree for comparison if the tree is being tested, or will output the forecasts as a final product if the “tree” is in production.

4.2.2. Interface Specification

The following functions will be made available as operators to the modelNode class. Individual function descriptions will be included as a comment in the appropriate file.

- mlp_model(input_dimension, output_dimension, layers)
- rf_model()

Note: The operator representing “mlp_model” is abbreviated to “mlp” and similarly “rf_model” is simplified to “rf”

4.2.3. A Static Model

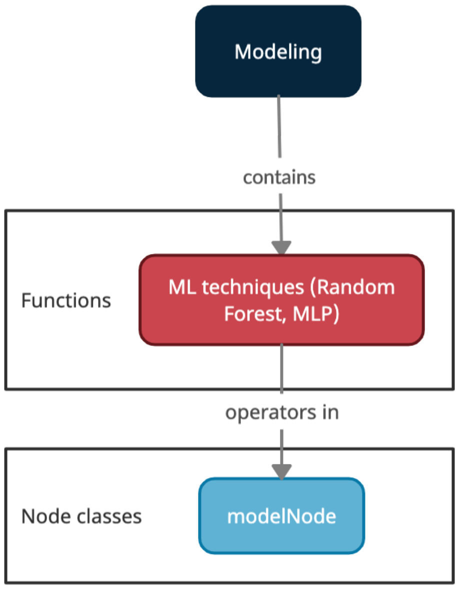


Figure 4.2 *Diagram of the modeling module's relationship to the tree*

4.2.4. Design rationale

Like the preprocessing module, the modeling module was designed in order to factor out more of the logic behind the execution of the pipeline. We want all of the standard functionality already used by Data Scientists available, especially the many models with which to do analysis, but we wish to treat them as operators of the nodes of the Tree. The library design allows us to swap in and out the various models and leave the Tree organization to its own module.

4.2.5 Alternative designs

There were discussions about potentially adding more ML models as operators to the modeling nodes, however, due to time constraints, we kept the limited number of required models. It is our hope that due to the similarity of input data for ML modeling techniques, future maintenance and improvements on the program would be a relatively straightforward process of adding to the "backend" library of functions.

4.3. Visualization/Evaluation

4.3.1. The module's role and primary function.

The Visualization and Evaluation module makes up the final layers of the Transformation Tree and is largely responsible for the output with which the Data Scientist will make decisions. It takes in forecast data and compares it with the test data set aside in the preprocessing stages to gauge accuracy of various pipelines. It is also responsible for plotting TS data in order to make preprocessing changes or modeling decisions.

4.3.2. Interface Specification

The following functions will be made available as operators to the evalNode or visualizeNode class. Individual function descriptions will be included as a comment in the appropriate file.

- plot(ts | ts_list)
- histogram(ts)
- box_plot(ts)
- summary(ts)
- shapiro_wilk(ts)
- d_agnostino(ts)
- anderson_darling(ts)
- qq_plot(ts)

- MSE(y_test, y_forecast)
- MAPE(y_test, y_forecast)
- sMAPE(y_test, y_forecast)
- RMSE(y_test, y_forecast)

4.3.3. A Static Model

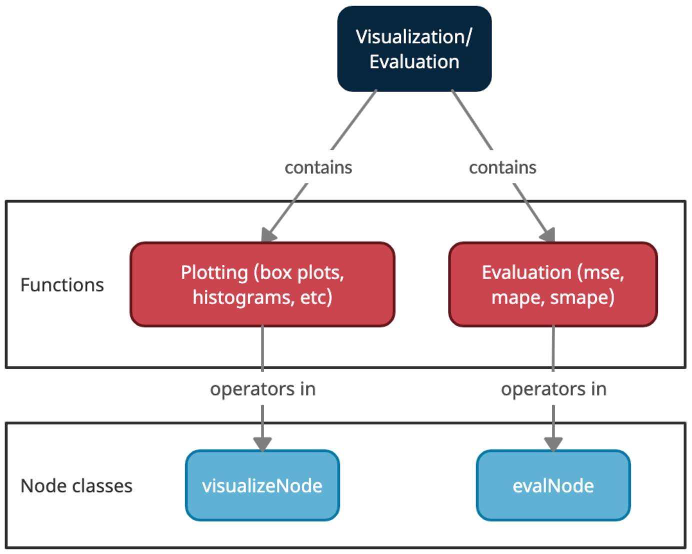


***Figure 4.3** Diagram of the visualization/evaluation module's relationship to the tree*

4.3.4. Design rationale

Like the previous modules, the visualization and evaluation module was designed in order to factor out more of the logic behind the execution of the pipeline. We want all of the standard functionality already used by Data Scientists available, but we wish to treat them as operators of the nodes of the Tree. The library design allows us to swap in and out the various plotting or evaluation steps and leave the Tree organization to its own module.

4.3.5 Alternative designs

There were discussions about potentially splitting the module into an evaluation module and a visualization module, but due to their limited functionality and similarity in helping to understand data, they were deemed appropriate to keep together. Since these were factored out, it should make adding more methods of data visualization possible in future updates to the code.

4.4. Transformation Tree

4.4.1. The module's role and primary function.

The Transformation Tree module is the largest module and is the primary interface for the user. It will handle adding nodes of various kinds to a tree to either evaluate pipelines or produce forecast data. The Tree is the main skeleton for the logic supplied by the other modules and is the main innovation for improving the efficiency of Data Scientists in this project.

4.4.2. Interface Specification

The following functions will be made available to the Data Scientist. Individual function descriptions will be included as a comment in the appropriate file.

Tree methods:
- add_prep_node(self, op_list, op, starting_date, final_date, increment)
- add_split_node(self, op_list, op)
- add_model_node(self, op_list, op)
- add_eval_node(self, op_list, op)
- add_visualize_node(self, op_list, op)
- execute_tree(self, infile)
- replicate_subtree(self, op_list)
- replicate_path(self, op_list)
- add_subtree(self, op_list, subtree)
- replace_operator(self, op_list, op)

Standalone functions:
- save_tree(tree, file_name)
- load_tree(file_name)

4.4.3. A Static Model

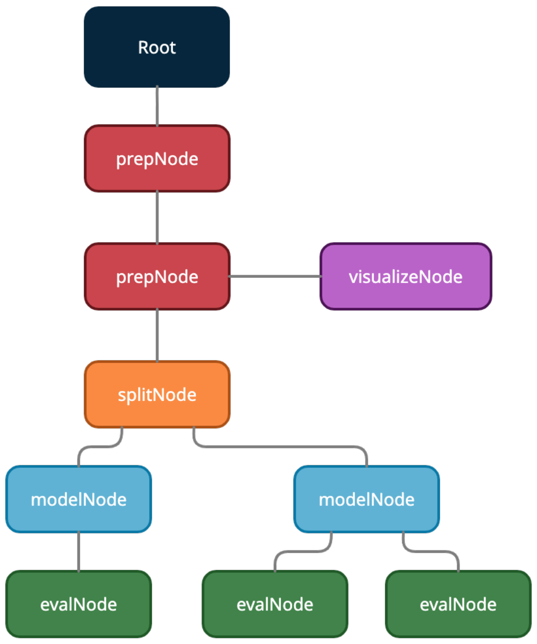


Figure 4.4 *Sample tree made up of the node classes showing the internal class relationships*

4.4.4. Design rationale

The Transformation Tree was designed with repetition in mind, as it allows Data Scientists to rapidly try many different subtly different pipelines to find the most accurate candidate. The n-ary tree structure allows for maximum flexibility and the various node classes allow for type checking and to factor out arguments of a particular step in a pipeline.

For example, by splitting up the visualization and evaluation functionality from the module of the same name into two node classes, we can factor out the functions that take a Time Series as an argument as well as the functions that take test and forecast data as arguments. It allows for functionality that is largely similar to reside in the same module, but to be simplified for when we wish to execute the tree, as each node has less information to store. It also permits us to check if a model node immediately precedes an evaluation node (as that is the only logical ordering for those two steps) and return errors or warnings if a user attempts to place an evaluation node elsewhere, perhaps immediately after a preprocessing node.

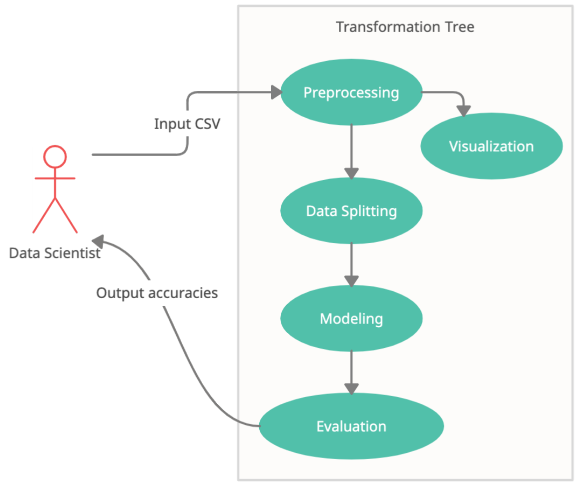
By placing the Transformation Tree in its own module, we also gain the improved readability and organization by having our entire API in one place for reference, as the tree and methods on the tree are the only direct interface for Data Scientists. When a user wants to add a model node for example, they can check the model module to see what operators are available and then check the Transformation Tree module to see how to call the tree methods with that modeling information.

4.4.5 Alternative designs

As implied above, we considered only having three node types, one for each module feeding the tree. However, even within one module there are many operators with arguments that vary widely, so we deemed it prudent to subdivide the module operators into smaller groups that each have a node class representation.

Additionally, we considered having a separate file for the API and keeping the tree separate. However, by establishing the majority of the API as the class methods for manipulating the tree, we gain the advantage having of the internal data of the tree (like the root node) be available to be manipulated without having to pass them to an external function.

5. Dynamic Models of Operational Scenarios (Use Cases)



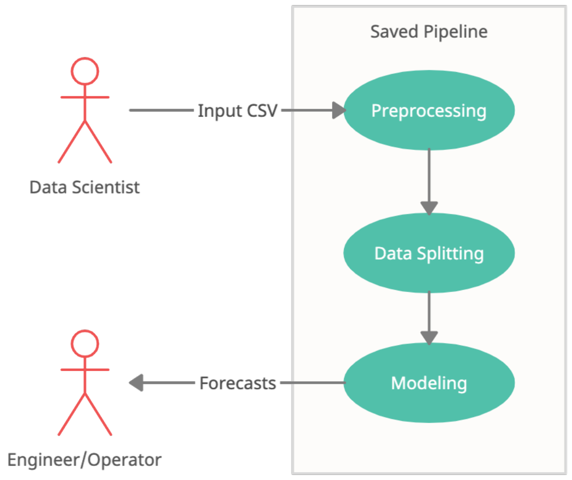


Figure 5.1 (left) *Pipeline testing use case using a Transformation Tree*
Figure 5.2 (right) *Production use case for using a working pipeline*

The above figures describe the two major use cases once the tree has been built. The testing use case (Figure 5.1) represents the full Transformation Tree during testing by the Data Scientist. It will output the accuracies and the associated pipelines for further evaluation about which pipeline to save. The production use case represents using a working pipeline to output forecasting data for use by the working professional stakeholder of the given situation. It largely ignores the now unnecessary evaluation and visualization steps and the primary output will be a .csv forecast file.

6. Acknowledgements

The UML diagrams in this document were created with the Creately web app.

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