<Team TUE2>

Software Design Specification

Team: Tue2 -2-10-2021

Table of Contents

1. SDS Revision History

[Date]	[Author]	[Description]
Feb 9th, 2021	Theodore Yun	- Initial meeting with creation of SDS document.
Feb 10th, 2021	Everyone	- Finished and formatted the final draft

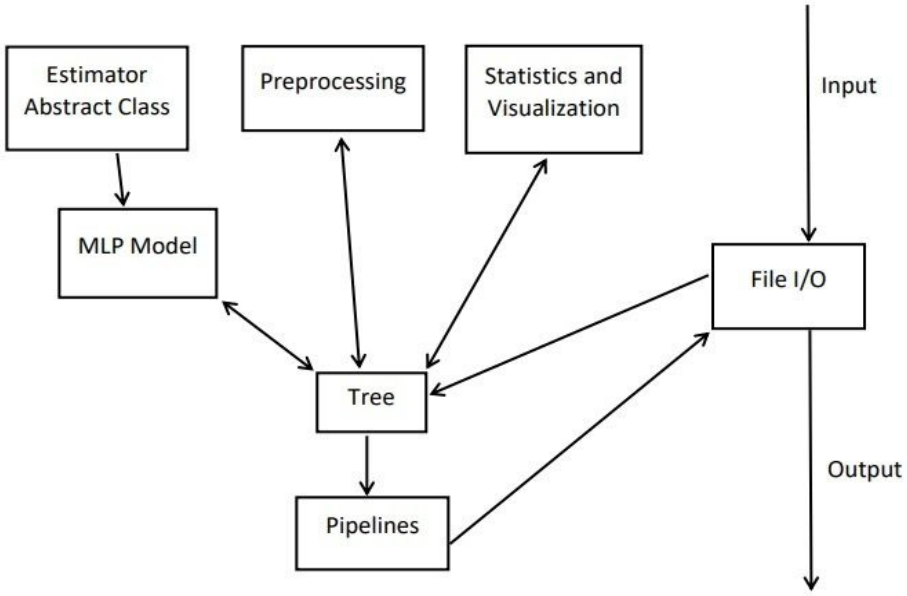
2. System Overview

This Library is a comprehensive suite of tools for a data scientist to use. The following functionalities are provided through individual modules:

1. Multiple Pre-processing functions to mold data found in preprocessing.py
2. Multiple Statistics and visualization functions to be used on data found in statistics_and_visualization.py
3. CSV reading capabilities found in fileIO.py
4. A mlp model found in mlp_model.py
5. A Tree Class, with functions passed into nodes found in tree.py
6. A Pipeline Class, to be made from a tree that can execute the functions in the tree across data found in pipelines.py

The Library is designed so that all of the functionalities previously mentioned can be accessed by importing lib.py into a Python script.

3. Software Architecture



1. Rationale

1. All interaction between modules in the system is through the tree and pipeline modules.
2. Having a central module simplifies interaction between other components. Fewer lines of coupling between modules means it will be easier to check for error, as any problems will be isolated to that direct interaction between components.
3. Input and output is isolated to a single module so file opening and closing can remain as simple as possible -- no need to work with files anywhere else.

4. **Software Modules**
    1. **Modules for data**
        1. **Estimator.py**

            1. Abstract Base Class that is used to inherit functions. Add methods to Override functions.

        2. **fileIO.py**

            1. **Functions:**
                1. **csv_has_timestamps:** Checks if the first column of a csv file appears to contain timestamps. Refactored to include __csv_process_header functionality.

                2. **csv_to_dataframe:** Reads csv file, checks if the file contains a header, and passes a file pointer at the start of the data to pandas.read_csv. This prevents any headers making it into the series values.

                3. **read_from_file:** Wrapper for csv_to_dataframe.

                4. **read_from_file_no_check:** Wrapper for pandas.read_csv(). Does not perform any checks of the data file.

                5. **write_to_file:** Write the predictions from the mlp model to the output file in the form of a csv with two columns (time, value).

        3. **mlp_model.py**

            1. **MLPModel:** Basic multilayer perceptron model for time series. Inherits Estimator abstract class.

                1. **fit:** Train the MLP Regressor model to estimate values like those in y_train. Training is based on date values in x_train.

                2. **score:** Scores the model's prediction accuracy.

                3. **forecast:** Produces a forecast for the time series's current state.

                4. **split_model_data:** Non-destructively splits a given data into sets readable for this estimator.

2. **Functions:**

    1. **train_new_mlp_model:** Wrapper function for initializing and training a new mlp model.

    2. **write_mlp_predictions_to_file:** Wrapper function for passing test data, estimator, and file name to fileIO.py.

4. **pipelines.py**
    1. **Pipeline: Pipeline class for executing Trees**

        1. **save_pipeline:** Saves the current pipeline to the saved states history.

        2. **make_pipeline:** Creates a pipeline from a selected tree path.

        3. **add_to_pipeline:** Adds function(s) to pipeline, if it can connect.

        4. **functions_connect:** Checks that the preceding functions output matching the proceeding functions input.

        5. **run_pipeline:** Executes the functions stored in the pipeline sequentially (from 0 -> pipeline size).
5. **preprocessing.py**

    1. **Functions:**

        1. **denoise:** To denoise a time series we perform a rolling mean calculation. Window size should change depending on what the user wants to do with the data.

        2. **impute_missing_data:** This function fills in missing data in the timeseries. There are multiple ways to fill in blank data (see docstring).

        3. **impute_outliers:** Outliers are disparate data that we can treat as missing data. Use the same procedure as for missing data (sklearn implements outlier detection).

        4. **longest_continuous_run:** Isolates the most extended portion of the time series without missing data.

5. **clip:** This function clips the time series to the specified period's data.

6. **assign_time:** In many cases, we do not have the times associated with a sequence of readings. Start and increment represent delta, respectively.

7. **difference:** Produces a time series whose magnitudes are the differences between consecutive elements in the original time series.

8. **scaling:** Produces a time series whose magnitudes are scaled so that the resulting magnitudes range in the interval [0, 1].

9. **standardize:** Produces a time series whose mean is 0 and variance is 1.

10. **logarithm:** Produces a time series whose elements are the logarithm of the original elements.

11. **cubic_root:** Produces a time series whose elements are the original elements' cubic root.

12. **split_data:** Splits a time series into training, validation, and testing according to the given percentages. training_set, validation_set, and test_set are unique subsets of the main set (ts) which do not have any overlapping elements.

13. **desgin_matrix:** The input index defines what part of the time series' history is designated to be the forecasting model's input. The forecasting task determines the output index, which indicates how many predictions are required and how distanced they are from each other (not necessarily a constant distance)

14. **design_matrix_2:** returns the design matrix composed of a range of data taken from a set of points of time.

15. **ts2db:** Combines reading a file, splitting the data, converting to database, and producing the training databases.

2. A Static and Dynamic Model

1. statistics_and_visualization.py

1. **Functions:**

    1. **myplot:** Plots one of more time series.

    2. **myhistogram:** Compute and draw the histogram of the given time series. Plot the histogram vertically and side to side with a plot of the time series.

    3. **box_plot: P**roduces a Box and Whiskers plot of the time series. This function also prints the 5 number summary of the data.

    4. **normality_test:** Performs a hypothesis test about normality on the time series data distribution. Besides the result of the statistical test, you may want to include a quantile plot of the data. Scipy contains the Shapiro-Wilkinson and other normality tests; matplotlib implements a qqplot function.

    5. **mse:** Computes the MSE error of two time series.

    6. **mape:** Computes the MAPE error of two time series.

    7. **smape:** Computes the SMAPE error of the two time series

2. tree.py

1. **Node: A node class**

2. **Tree: N-ary tree**

    1. **search:** Searches the tree for target, returns Node if found, otherwise None.

    2. **insert:** Creates a new node with func to be inserted as a child of parent.

    3. **delete:** Deletes the given Node, children are removed.

    4. **replace:** Replaces the function of the given Node with func.

    5. **get_args:** Gets function arguments the previous node does not fulfill.

6. **match:** Checks if the child node can be attached to the parent node matches child output to parent input.

7. **serialize:** Formats the tree into a string, so we can save the tree easily.

8. **deserialize:** Returns the serialized string back to a tree.

9. **save_tree:** Saves the current tree state as a pre-ordered list.

10. **restore_tree:** Saves the current tree state as a pre-ordered list.

11. **traverse:** Returns a preorder traversal of the Tree as a list.

3. Rationale

1. **mlp_model.py**
    1. From a high-level perspective, mlp_model.py is made to work as smoothly as possible with the main transformation tree module. Design for mlp_model.py is based solely off the functionality of the MLPRegressor class as defined in the scikit-learn library. The class wraps an MLPRegressor as an attribute. Thus, the purpose of the class methods are to prepare incoming pandas DataFrame objects to be used by the MLPRegressor object. Along with the MLPModel class itself, included as well are a couple non-class functions which wrap functionality like initialization, training, forecasting, and writing to file so the class can be used in the transformation tree without the tree needing to accommodate passing classes between nodes.

2. **pipelines.py**
    1. This module is designed to execute the paths or "pipelines" on a set of data over a given Tree. We implemented this functionality with a python class, allowing us to create a Pipeline object to streamline the process.

3. **tree.py**
    1. We designed this module to implement a N-ary Tree. This Tree is a collection of linked Nodes, each with their own functions The advantage of a Tree structure in executing mlp and preprocessing functions is that a user can have a single Tree that executes many different pipelines. The Tree structure also allows intuitive branch tracking for users (the difference between applying function A or function B to a data set D can be easily viewed by creating a Tree with a branching from a parent node to nodes A and B).

4. Alternative designs

1. Our alternative design would be modules that are being structured in a straight line rather than being centralized in the tree/pipeline. Or perhaps to create data structures such as cyclic graphs, in which the user can decide when to end looping or sample from looping. This alternative data structure may be useful in some specific use cases, but as a tool for developing intuition of mlp processes and on the merits of organization the tree structure seems superior.

5. Operational Scenarios (Use Cases)

Use Case: building a transformation tree and executing a machine learning pipeline.

Using this Library, Data Scientists will be able to model transformation trees to assist them in time series analysis. Time series data will be contained in pandas DataFrames, and the preprocessing functions implemented provide a variety of options to analyze and shape the data (such as normalization, scaling, denoising etc…).

A Transformation tree will consist of nodes containing a function. Preprocessing -> forecasting (mlp model) -> modeling (graphically or writing output to file).

Users will construct these trees using the tree methods listed in the tree class. Then by calling methods of the pipeline class, paths of the tree can be executed.

A detailed example of this process can be found in the READ.md and test_tree_and_pipelines.py file.

Use Case: Adding a custom estimator to be used by a transformation tree

Included in the library is a default multilayer perceptron model which can be trained with time series data to predict what a value may be at a given date/time. If the user wants to add a new estimator, they will have to use the base abstract Estimator class which is also included in the library. Creating a new custom estimator involves basic knowledge of:

- Python class
- Inheritance
- Object oriented programming

To complete this task, users must do the following:

1. Import the main library by typing “from lib import *” at the top of the file.
2. Create a new class which inherits attributes from the Estimator class
    1. A class can be defined by typing “class New_Estimator(Estimator)”
3. Estimator abstract class does not have any attributes so super().init() does not need to be added to the initialization method of the new class.

4. The following methods need to be included in the new class:
    1. New_Estimator.fit() - trains the new estimator based on the user's definition
    2. New_Estimator.forecast() - asks the estimator forecast a specific time or set of times
    3. New_Estimator.split_model_data() - splits data for the model to train or forecast based on (specific to user implementation)
5. The following attributes may or may not be necessary depending on the specific implementation of the custom estimator:
    1. input_dimension - integer representing number of columns of data to consider when training and forecasting
    2. output_dimension - integer representing number of columns of data to learn to predict
6. Including the aforementioned methods will override the abstract methods in the Estimator base class
7. Specific functionality is up to the user, but the three methods will be enough to work with the rest of the library.

6. References

See requirements.txt for full list of imported libraries

1. Pandas documentation: https://pandas.pydata.org/pandas-docs/stable/index.html
2. Numpy documentation: https://numpy.org/doc/1.20/reference/index.html
3. Docker documentation: https://docs.docker.com/
4. Python unittest documentation: https://docs.python.org/3/library/unittest.html
5. Matplotlib documentation: https://matplotlib.org/api/index.html
6. SkLearn documentation: https://sklearn.org/documentation.html
7. SciPy documentation: https://docs.scipy.org/doc/scipy/reference/

7. Acknowledgements

The document was written based on the SDS format that is given from class.

All modules' description from README.md from bitbucket.

Software Engineering: Principles and Practice by Hans van Vliet. 2007. Guidance for project organization and agile practices.

Juan Flores and Joseph McLaughlin for office hour support.