Project 1
Programmer Documentation

T. Christenson (tc), Yifeng Cui (yc), Brian Gunnarson (bg), Jacob Rammer (jr), Sam Peters (sp)

-- 2-10-2021 --

-- v1.0 --

Table of Contents

1. Introduction 1
2. Preprocessing 2
3. Modeling and Forecasting 15
4. Statistics and Visualization 17
5. Tree 23
6. Operator Keys 30

----

1. Introduction

This document includes all the component specific documentation. When discussing each module we will dive into the software dependencies, how the source code files relate to each other, the purpose and inner workings of each source code file, and the purpose, arguments, effects, and returned values of each function, class and method.

Each component was written by a different group member so we thought it would be best to split the documentation for this section person-by-person. The top of each section lists who the author(s) is (are) for referencing purposes.

----

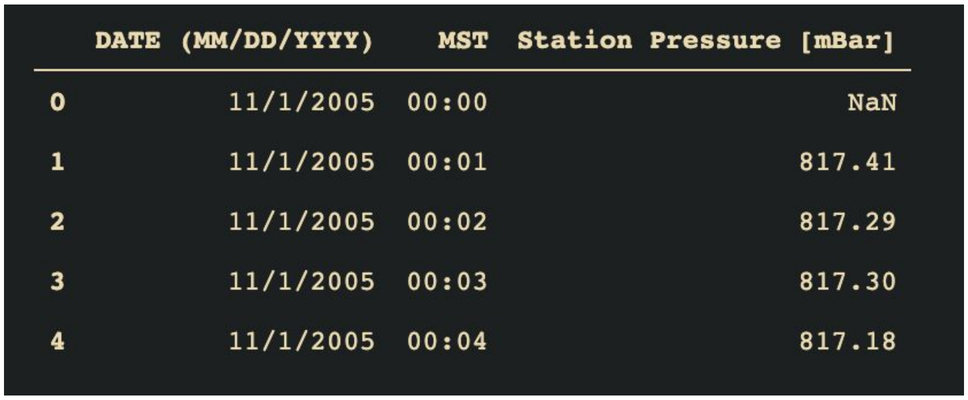
2. Preprocessing

By Jacob Rammer and Yifeng Cui

Preprocessing.py: Preprocessing.py contains all of the preprocessing methods from the project hand out. The preprocessing methods are methods to manipulate time series and time series data or return information about a time series. Since preprocessing manipulates a time series, we opted to create a TimeSeries class for easy data management and ease of use. Methods included in preprocessing are as follows: read_from_file, write_to_file, assign_time, clip, denoise, impute_missing, difference, impute_outliers, longest_continuous_run, scaling, standardize, logarithm, cubic_root, split_data, design_matrix, and ts2db.

TimeSeries Class Variables:

- Self.data: this is the dataframe object. Life is the whole program execution



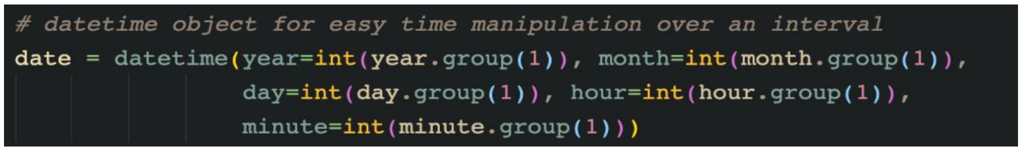
```
   DATE (MM/DD/YYYY)    MST  Station Pressure [mBar]
0          11/1/2005  00:00                      NaN
1          11/1/2005  00:01                   817.41
2          11/1/2005  00:02                   817.29
3          11/1/2005  00:03                   817.30
4          11/1/2005  00:04                   817.18
```

Visual representation of a dataframe

- **read_from_file(self, file_name: str) -> Self**
    - File_name: file_name is a string representing the input file location. EX: /Documents/file.csv
    - This method returns a Timeseries object.
    - This method reads in a csv file and converts that file into a Pandas dataframe. In order for this method to work correctly, the input file must have a header on the first line stating what each of the values represent. These headers are also needed to correctly label the columns in the dataframe so you can easily identify what each

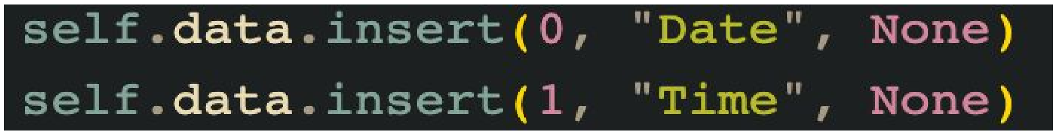
    column represents (see image above). The way a dataframe is structured is each line is a row in the dataframe. We have set up our dataframes in this format: date, time, data.
    - Completed: 01/16/21 (Jacob Rammer)
    - Time to complete: < 1 hour
  - Variables
    - This method does not hold variables. Instead, this method updates self.data to a dataframe.
  - Line-by-Line
    - Try to open the file given the path specified.
    - If that file is found, use pandas.read_csv to convert that file to a dataframe object to self.data.
    - If a file is not found, throw an exception so the program doesn't crash.
- **Write_to_file(self, file_name: str) -> None**
  - File_name: is a string representing the output file location. EX: /Documents/file.csv
  - This method takes a Pandas dataframe and writes it to a CSV file. This method can create new files if the specified output file does not exist.
  - Completed: 01/16/21 (Jacob Rammer)
  - Time to complete: < 1 hour
  - Variables
    - This method does not hold variables. Instead, this method takes self.data and writes the contents to a CSV file
  - Line-by-Line
    - Take self.data and use pandas.to_csv to write the data to a CSV file
- **Assign_time(self, start: str, increment: int) -> None**
  - Start: start is a string representing a date and time of day. The format of start must be in this form to work correctly since I am using regex to extract the date and time: mm/dd/yyyy hh:mm (space between yyyy and hh). For example: 01/01/2020 12:30. The time format used here is 24 hours, so use 13:00 for 1PM.

- Increment: increment is an integer representing how many hours to increment the time. As previously stated, increment is an int so it must be a whole number, such as 1.
- This method assigns a date and time column to a dataframe if one is missing so we can correctly plot it later. The way this method works is by extracting the start string into the correct group such as month, day, year, hour, minute using regex. Once the string has been extracted and all groups have the correct data, I then create a datetime object for easy time manipulation.

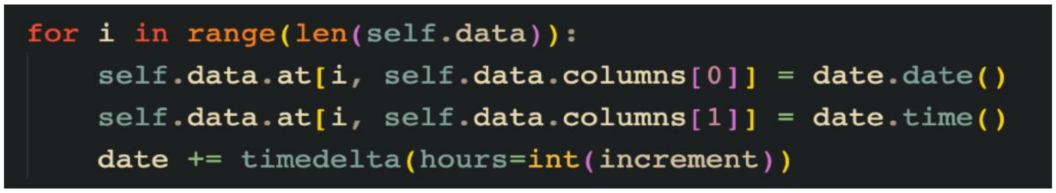
  ```
  # datetime object for easy time manipulation over an interval
  date = datetime(year=int(year.group(1)), month=int(month.group(1)),
                  day=int(day.group(1)), hour=int(hour.group(1)),
                  minute=int(minute.group(1)))
  ```

  Once that datetime object is created, we then insert a “date” and “time” column that was missing in the dataframe.

  ```
  self.data.insert(0, "Date", None)
  self.data.insert(1, "Time", None)
  ```

  Once the columns headers are inserted, we iterate over the whole dataframe adding the date in the first column and time in the second column and shifting the data values to the last column of the dataframe. Datetime has a method called timedelta which we utilized to increment the time by the specified hours and this method will also automatically roll the date to the next day if needed. This method modifies the dataframe.
- Completed: 01/24/2021 (Jacob Rammer)
- Time to complete: 3-5 hours

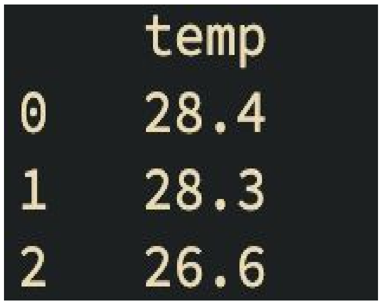
  

  ```
  for i in range(len(self.data)):
      self.data.at[i, self.data.columns[0]] = date.date()
      self.data.at[i, self.data.columns[1]] = date.time()
      date += timedelta(hours=int(increment))
  ```

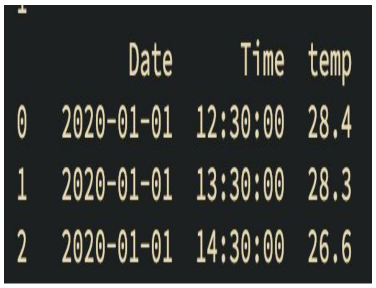
- Variables
  - Month_reg: is the regex pattern to match the month. This is a local variable so variable life is while the method is running.

        - Day_reg: is the regex pattern to match the day. This is a local variable so variable life is while the method is running.
        - Year_reg: is the regex pattern to match the year. This is a local variable so variable life is while the method is running.
        - Hour_reg: is the regex pattern to match the hour. This is a local variable so variable life is while the method is running.
        - Minute_reg: is the regex pattern to match the minute. This is a local variable so variable life is while the method is running.
        - Month: is the extracted segment from start that is matched from the month_reg. This is a local variable so variable life is while the method is running.
        - Day: is the extracted segment from start that is matched from day_reg. Is a string. This is a local variable so variable life is while the method is running.
        - Year: is the extracted segment from start that is matched from year_reg. Is a string. This is a local variable so variable life is while the method is running.
        - Hour: is the extracted segment from start that is matched from hour_reg. Is a string. This is a local variable so variable life is while the method is running.
        - Minute: is the extracted segment from start that is matched from minute_reg. Is a string. This is a local variable so variable life is while the method is running.
        - Date: is a datetime object that is created from the month, day, year, hour, minute variables. This is a local variable so variable life is while the method is running.
    - Line-by-Line
        - Lines that contain *_reg are regex pattern declarations. These will be used later to search the input string, start, for the specified pattern.
        - Lines that contain re.search search the input string and match the corresponding value to the variable. So, for example, month = re.search searches start for the month pattern: *mm*/dd/yyyy hh:mm.

        - Next, we create a datetime object called date so that we can later utilize datetime methods to manipulate the time since this assign_time increments the time by a specified increment. Since datetime expects ints, we have to cast our variables from ints as the start variable is a string.
        - Next, we create a new column in self.data in the first column called "Date".
        - Next, we create a new column in self.data in the second column (from the left) called "Time".
        - Then we loop through the whole dataframe assigning dates and times the appropriate column. We setup our dataframe so that date was always in the far left (index 0), time in the middle (index 1) and data in the far left (index 2)



```
   temp
0  28.4
1  28.3
2  26.6
```



```
         Date      Time  temp
0  2020-01-01  12:30:00  28.4
1  2020-01-01  13:30:00  28.3
2  2020-01-01  14:30:00  26.6
```

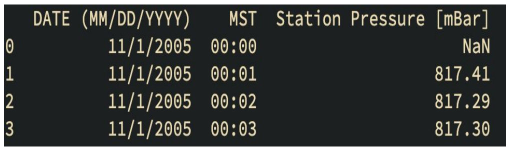
Using 1 hour time increments

- **Clip(self, starting_date: str, final_date: str) -> TimeSeries**
    - Starting_date: starting date is a string in the form of mm/dd/yyyy (01/01/2020)
    - Final_date: ending date is a string in the form of mm/dd/yyyy (01/01/2020)
    - This method searches the original time series and extracts data between a specified date. After some research, rather than implementing this method ourselves, we discovered there was already a library written to accomplish what we wanted called PyJanitor. This method is as simple as calling pyjanitor.filter_date(header_name, starting_date, final_date) on the original dataframe. Header_name is the name of the date column in the original dataframe. Once the specified dates are extracted, this

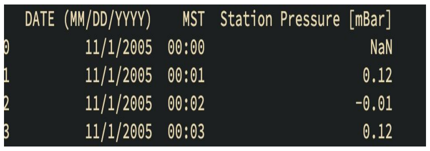
method returns a new TimeSeries object. This method does not modify the original time series.
    - Completed: 01/24/2021 (Jacob Rammer)
    - Time to complete: 2-3 hours
    - Variables
        - First_date: Holds the name of the column header. Is a string and a local variable.
        - Clipped: will be a new DataFrame that holds the extracted portion of self.data. This variable is returned as a new TimeSeries object, so lifetime is the while the program is running.
    - Line-by-Line
        - We first grab the name of the date column so we can copy it to the new DataFrame that is produced by this method.
        - Then we use pyjanitor.filter_date to search self.data for the specified date range providing the name of the date column so it can be copied over.
        - We then return a new TimeSeries object with the extracted dates from the line above.
- **Denoise(self) -> TimeSeries**
    - This method has no input arguments.
    - This method simply works to denoise a time series by first calling impute_missing() then calling impute_outliers(). This method modifies the original time series and also returns a new TimeSeries object.
    - Completed: 01/27/2021 (Jacob Rammer)
    - Time to complete: < 1 hour
    - Variables
        - There are no variables for denoise
    - Line-by-Line
        - First, we call the impute missing method to fill in missing values.
        - Then, we impute outliers and remove them, making the time series cleaner.

        - Both these lines modify self.data and we also return a new TimeSeries with the denoised data.

- **Impute_missing(self) -> None**
    - This method has no input arguments.
    - This method fills in missing values such as NaNs (not a number) using a built-in Pandas method called fillna. The way fillna works is by iterating through the whole dataframe looking for NaNs and backfilling the data. This method modifies the original time series.
    - Completed: 01/24/2021 (Jacob Rammer)
    - Time to complete: < 1 hour
    - Variables
        - Temp: temp is a copy of self.data that is first searched for missing values then are filled. This variable is then copied back to self.data. This is a local variable so variable life is while the method is running.
    - Line-by-line
        - First, we create a new variable to hold the self.data dataframe so the existing data is untouched. We then use pandas.fillna() that grabs the previous known value from the trend in self.data, filling in any missing values.
        - Then assign temp to self.data to update self.data with the new values.
- **Difference(self) -> TimeSeries**
    - This method has no input arguments.
    - This method calculates the difference between the data columns in a time series by utilizing the Pandas.shift() function. This method calculates the difference from the bottom up. This method does not modify the original time series but returns a new TimeSeries object with the difference calculated (picture 2).



```
  DATE (MM/DD/YYYY)    MST  Station Pressure [mBar]
0         11/1/2005  00:00                      NaN
1         11/1/2005  00:01                   817.41
2         11/1/2005  00:02                   817.29
3         11/1/2005  00:03                   817.30
```



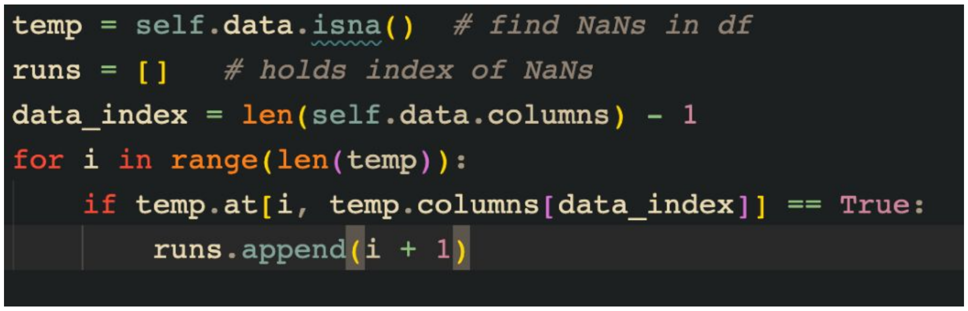
```
  DATE (MM/DD/YYYY)    MST  Station Pressure [mBar]
0         11/1/2005  00:00                      NaN
1         11/1/2005  00:01                     0.12
2         11/1/2005  00:02                    -0.01
3         11/1/2005  00:03                     0.12
```

    - Completed: 01/26/2021 (Jacob Rammer)

    - Time to complete: 5-6 hours
    - Variables
        - Data_index: holds the data index of the dataframe (int). This variable is then used to index the dataframe to the data column so we can find the difference between rows. This is a local variable so variable life is while the method is running.
        - Temp: is a copy of self.data. This is used so that we can return a new TimeSeries object without modifying the original self.data variable. This is returned as a new object so lifetime is program execution.
    - Line-by-Line
        - First, we grab the length of self.data (how many columns). Since dataframes use 0-based indexing, we must subtract one from length.
        - Next, we make a copy of self.data so it's not modified.
        - Then we look at all rows in temp and perform the operation of subtracting data values from the bottom up. This is accomplished by using the pandas.shift(-1) method.
        - We then return a new TimeSeries with the temp variable DataFrame containing the values of the difference between rows.
- **Impute_outliers(self) -> None**
    - This method has no input arguments.
    - This method imputes outliers, both high and low. Low quantile is defined by the bottom 0.1 and the high quantile is defined by the top 0.1. This method searches the dataframe and drops outliers within the specified low and high values. This method modifies the original dataframe and does not return anything.
    - Completed: 01/27/2021 (Jacob Rammer)
    - Time to complete: 5 hours
    - Variables

        - Temp: is a copy of self.data. Used so we don't touch self.data initially. Local variable so lifetime is while the method is running.
        - Data_index: is used so we can find the index of the data column to manipulate. Local variable so lifetime is while the method is running.
        - Q_low: is used to grab the low quantile of the TimeSeries object. Local variable so lifetime is while the method is running.
        - Q_high: is used to grab the high quantile of the TimeSeries object. Local variable so lifetime is while the method is running.
    - Line-by-Line
        - First, we make a copy of self.data so we can later reference the values in the dataframe.
        - Next, we once again find the index of the data column so we can index the dataframe in the data column.
        - We then compute the lowest values in the dataframe by using pandas.quantile().
        - Then we also compute the highest values in the dataframe by using pandas.quantile().
        - Finally, we search the whole dataframe looking for values that are above and below our q_low and q_high values and drop them from the self.data.
- **Longest_continuous_run(self) -> TimeSeries**
    - This method has no input arguments.
    - This method searches through the whole dataframe looking for the longest continuous run. A continuous run can be defined as the largest number of rows in a dataframe that do not contain NaNs (not a number) values in the data column. This is accomplished by using a method supplied by Pandas called isna() which first changes the data column to Boolean values (NaNs are True). We then iterate over the whole dataframe looking for True values and appending them to an index list. After the first iteration, we then look at the index array to see which run is the longest. We then

return the longest run as a new TimeSeries object. This method does not modify the original data.



```
temp = self.data.isna()  # find NaNs in df
runs = []   # holds index of NaNs
data_index = len(self.data.columns) - 1
for i in range(len(temp)):
    if temp.at[i, temp.columns[data_index]] == True:
        runs.append(i + 1)
```

- Completed: 01/30/2021 (Jacob Rammer)
- Time to complete: 5-7 hours
- Variables
    - Temp: temp is a copy of the self.data dataframe so we can call Pandas.isna() without modifying self.data. This variable holds all data of self.data except the data column are turned into booleans to represent if they are NaNs or not. Local variable.
    - Runs: is a list that holds index values of NaNs in the dataframe so we can later reference them to find the longest run. Local variable.
    - Data_index: is an int that once again holds the index of the data column for dataframe indexing. Local variable.
    - Ret: will be the DataFrame that is returned after the longest run is computed. So, this variable lifetime is while the program is executing. Local variable.
    - First: holds the index of the first index of the self.data variable that is a valid number. Local variable.
    - Last: holds the index of the last index of the self.data variable that is a valid number. Local variable.
    - Temp: is a new variable in the for-loop scope that holds the rows between the index first and last. So, this variable is a DataFrame so we can check the length of the DataFrame to determine the longest run.
- Line-by-line

        - First, we create a temp variable that holds all objects in self.data, except values in the data columns are turned into booleans.
        - Then create a runs variable that holds the index of NaNs
        - Once again find the index of the data column so we can later index those values.
        - Then we iterate over the temp variable looking for True values. If there is a NaN in the original dataframe, it is turned to "True" in temp. We then append the current index so we can look at the length later.
        - After this loop is finished, append the last index of the dataframe to runs, this accounts for the longest run being at the end of the file.
        - We then create a blank dataframe holding zero items, so the length is currently 0.
        - Then we create a for loop to iterate over index values stored in runs.
        - First holds the first index in runs, last holds the second index. So, if our runs = [1, 4, 8], on the first loop, first = 1, last = 4, second loop, first = 4, last = 8 and so on and it's clear here that the longest run is between index 4 and 8.
        - We then create another temp variable in the loop that holds all data between first and last that has been extracted from self.data.
        - We can then check the length of the dataframe using python.len(), and if the length of the temp DataFrame is longer than ret, we assign ret to the longer dataframe.
        - Then we return a new TimeSeries object with the longest run.

- **scaling(self):**
    - Completed: 01/25/21 (Yifeng Cui)
    - This method has no input arguments

    - This method uses a copy of the current Dataframe. Each number element in CopyData is read through the Loop and these values are scaled by the formula.
    - Returns: TimeSeries with scaling data.
    - Variables:
        - new_def: A copy of dataframe
- **standardize(self):**
    - Completed: 01/25/21 (Yifeng Cui)
    - This method has no input arguments.
    - This function uses a copy of the current Dataframe. Each number element in CopyData is read through the Loop and these values are standardized by the formula.
    - Returns: TimeSeries with standard data.
    - Variables:
        - new_def: A copy of dataframe
- **logarithm(self):**
    - Completed: 01/25/21 (Yifeng Cui)
    - This method has no input arguments.
    - This method uses a copy of the current Dataframe. Each number element in CopyData is read through the Loop and these values are going to be computed using NumPy(np.log10()).
    - Returns: TimeSeries with logarithm data.
    - Variables:
        - new_def: A copy of dataframe
- **cubic_root(self):**
    - Completed: 01/25/21 (Yifeng Cui)
    - This method has no input arguments.
    - This method uses a copy of the current Dataframe. Each number element in CopyData is read through the Loop and these values are unrooted.
    - Returns: TimeSeries with cubic root.
    - Variables:
        - new_def: A copy of dataframe
- **split_data(self, perc_training, perc_valid, perc_test):**
    - Completed: 01/27/21 (Yifeng Cui)

    - perc_training: A giving value from demo.py for splitting a time series into training
    - perc_valid: A giving value from demo.py for splitting a time series into validation
    - perc_test: A giving value from demo.py for splitting a time series into testing
    - This method uses .tolist() to list the data from the panda dataframe, and make two new lists to save the time and number from array, finally splits the number value into training, validation, and testing according to the given percentages.
    - Returns: N/A
    - Variables:
        - Array: a list of dataframe
        - timeList : the time value from array
        - varlist: the number value from array
- **design_matrix(self, input_index, output_index):**
    - Completed: 01/29/21 (Yifeng Cui)
    - input_index: The part of the time series' history is designated to be the forecasting model's input.
    - Output_index: is supplied by the user (data scientist)
    - Returns: The matrixes of training and testing
    - Variables:
        - x_train: input training dataset determined by input index value
        - y_train: output training dataset determined by output index value
        - x_start: the start of input part
        - x_end: the range from 0 to input index.
        - y_start: the start of the output part, which is x_end.
        - y_end: the range from input index to output index.1
        - x_test: input testing dataset determined by input index value
        - y_test: output testing dataset determined by output index value
- **ts2db(self, input_file_name, perc_training, perc_valid, perc_test, input_index, output_index):**
    - Completed: 02/02/21 (Yifeng Cui)
    - input_file_name: the csv file needs to read

- perc_traning: A specified percentage value for splitting a time series into training
- perc_valid: A specified percentage value for splitting a time series into training
- perc_test: A specified percentage value for splitting a time series into training
- input_index: The part of the time series' history is designated to be the forecasting model's input.
- Output_index: The part of the time series' history is designated to be the forecasting model's output.
- Returns: x_train, y_train, x_test, y_test
- Variables:
    - trainingMatrix: the training matrix by calling design matrix with given value
    - testMatrix: the training matrix by calling design matrix with given value
    - x_train: an input training dataset determined by trainingMatrix, it works for fit() function in modelingAndForecasting.py
    - y_train: an output training dataset determined by trainingMatrix, it works for fit() function in modelingAndForecasting.py
    - x_test: an input testing dataset, a list of testing determined by testMatrix
    - y_test: an output testing dataset determined by testMatrix

---

3. Modeling and Forecasting

By Tyler Christenson

modelingAndForecasting.py: modelingAndForecasting.py contains both available machine learning regression models as well as their 'fit' and 'predict' methods. To better fit the project specification, we wrapped each of these functions into parent functions that took in parameters that were better suited for integration with the other modules in this project. Function names included mlp_model, rf_model, fit, and predict, where one of the two models were passed into the fit and predict functions to then call their respective fit and predict methods from scikit-learn.

Modeling functions:

- **mlp_model(input_dimension, output_dimension [, layers]) -> sklearn.neural_network.MLPRegressor object**
    - input_dimension: describes a parameter of the Multilayer Perceptron (mlp) neural network. The input dimension of the model is the number of elements in the input training set. This value is passed as an integer and corresponds to the number of neurons in the input layer, as well as the hidden layers, of the model. This is defaulted to 100.
    - output_dimension: describes another parameter of the Multilayer Perceptron neural network. The output dimension represents the number of neurons in the output neuron layer as it is the number of elements expected in the output training set. Defaulted to 1.
    - layers: describes the number of neuron layers in the model. Defaulted to 10.
    - Variables
        - network: a list that describes the dimensions (size of input_dimension) of the hidden layers within the MLPRegressor model. Passes this list into the MLPRegressor object and overrides the default hidden_layer_sizes list.
    - This function returns a MLPRegressor object from the scikit-learn library.
    - Completed: 1/29/21 (Tyler Christenson)
- **rf_model() -> sklearn.ensemble.RandomForestRegressor object**
    - Takes no inputs (uses all sklearn defaults for RandomForestRegressor)
    - Returns a RandomForestRegressor object
    - Completed: 1/30/21 (Tyler Christenson)

Modeling and Forecasting methods:

- **fit(model, x_train, y_train) -> Trained regression model object**
  - model: takes one of mlp_model or rf_model as input.
  - x_train: input training dataset. Taken from TimeSeries.ts2db() in preprocessing.py
  - y_train: output training dataset. Taken from TimeSeries.ts2db() in preprocessing.py
  - Variables
    - y_train_array: utilizes numpy.ravel on the passed y_train argument to assure that it is 1-dimensional.
  - Returns a trained regression model object (one of mlp_model or rf_model)
  - Completed: 2/1/21 (Tyler Christenson)
- **predict(model, X) -> List of predicted data**
  - model: takes one of mlp_model or rf_model as input.
  - X: input test dataset. Taken from TimeSeries.ts2db() in preprocessing.py
  - Variables
    - Predictions: stores the list of predicted data
  - Returns the predictions list
  - Completed: 2/3/21 (Tyler Christenson)

--

4. Statistics and Visualization

By Brian Gunnarson

Visualization.py: Visualization.py contains all of the statistics and visualization methods from the project handout. The visualization methods are the methods to create plots, histograms, and box-plots of the time series data. The statistical methods are the methods used to conduct a normality test on time series data and to calculate errors in forecasting models. Since we opted to create a TimeSeries object in the preprocessing section, we chose to utilize the data already stored in this object when visualizing and running statistical tests. The methods included in statistics and visualization are as follows: plot, histogram, box_plot, normality_test, mse, mape, and smape.

Dependencies

Visualization.py has 4 dependencies which are:

- matplotlib: to import pyplot for visualization purposes
- preprocessing: to import the TimeSeries object we have storing data
- typing: to import List for typecasting and documentation purposes
- scipy.stats: to import shapiro for our Shapiro-Wilkinson normality test and to import probplot to display our data in a quantile plot

Relation to Other Source Code Files

The statistics and visualization module for this program is related to 5 other source code files:

- preprocessing.py: This is used for the TimeSeries object. It is crucial for visualization.py because it's where we get the data that we need to create plots and run normality tests. Additionally, we get the y_test variable used in all 3 error checking functions (mse, mape, and smape) from the ts2db method in the TimeSeries object.
- modelingAndForecasting.py: This module is used to obtain the y_forecast variable used in all 3 error checking functions (mse, mape, and smape) from the predict function.
- tree.py: This module calls the functions in visualization.py when building branches of the transformation tree.

- operatorkeys.py: This module is used when building the transformation tree in tree.py and therefore used when calling functions in visualization.py. This module stores the type of input and output values of each function in visualization.py.
- ts_analysis_support.py: This module imports visualization.py as a way to put all of our source code files together in one place

Purpose and Inner Workings of visualization.py

visualization.py contains 8 separate functions: 3 for visualization purposes, 3 for statistical purposes, and 1 for both.

Visualization Functions:

- plot():
    - Input: A single TimeSeries object from preprocessing.py
        - Note: to make integration between this module and other modules simpler, we decided to only accept one time series object at a time for this function
    - Output: None
    - Side Effects: A plot of the time series data is displayed
- histogram():
    - Input: A single TimeSeries object from preprocessing.py
    - Output: None
    - Side Effects: A histogram and a plot of the time series data are displayed side by side
- box_plot():
    - Input: A single TimeSeries object from preprocessing.py
    - Output: None

    - Side Effects: A box and whiskers plot of the time series data is displayed. Additionally, a 5-number summary is printed.

<u>Statistical Functions</u>

- mse():
    - Input: the time series data being tested which is represented as a list (y_test) and our forecasting model's data which is also represented as a list (y_forecast)
    - Output: the mean squared error as a float
- mape():
    - Input: the time series data being tested which is represented as a list (y_test) and our forecasting model's data which is also represented as a list (y_forecast)
    - Output: the mean absolute percentage error as a float
- smape():
    - Input: the time series data being tested which is represented as a list (y_test) and our forecasting model's data which is also represented as a list (y_forecast)
    - Output: the symmetric mean absolute percentage error as a float

<u>Both Statistical and Visualization</u>

- normality_test():
    - Input: A single TimeSeries object from preprocessing.py
    - Output: None
    - Side Effects:
        - Prints whether the data is normal or not normal
        - Produces a quantile plot of the data in the time series

Purpose, Scope, and Lifetime of Variables

There are no global variables in this module so it’s easier to break variables down function by function:

- histogram():
    - fig: a matplotlib Figure object produced by using plt.subplots(). Used as a throwaway variable. It’s alive throughout this entire function.
    - axes: an array of Axes objects from matplotlib produced by using plt.subplots(). Used to format the histogram and plot created by this function. It’s alive throughout this entire function.
- normality_test():
    - results: an array of column data to pass into scipy.stats’ shapiro method in order to obtain the p-value we need for the normality test. It’s alive throughout the entire function.
    - stats: the test statistic provided by using scipy.stats’ shapiro method. Used as a throwaway variable. It’s alive after we populate the results variable and until the end of the function.
    - p: the p-value used for a hypothesis test about the data. It’s provided by using scipy.stats’ shapiro method and it’s alive after we populate the results variable and until the end of the function.
- mse():
    - result: a variable to store our calculations so we have something to return at the end of the function. It’s alive throughout this entire function.
    - n: the amount of data in our forecasting model provided by using python’s built in len() method. Used for determining the range of the for loop in the following line. It’s alive after the result is initialized and until the end of the function.

    - diff: used as a temporary storage variable for calculations. This variable is only alive inside the body of the for loop containing it.
- mape():
    - result: a variable to store our calculations so we have something to return at the end of the function. It’s alive throughout this entire function.
    - n: the amount of data in our forecasting model provided by using python’s built in len() method. Used for determining the range of the for loop in the following line. It’s alive after the result is initialized and until the end of the function.
    - diff: used as a temporary storage variable for calculations. This variable is only alive inside the body of the for loop containing it.
    - inner: used as a temporary storage variable for calculations. This variable is only alive inside the body of the for loop containing it.
- smape():
    - result: a variable to store our calculations so we have something to return at the end of the function. It’s alive throughout this entire function.
    - n: the amount of data in our forecasting model provided by using python’s built in len() method. Used for determining the range of the for loop in the following line. It’s alive after the result is initialized and until the end of the function.
    - numerator: used as a temporary storage variable for calculations. This variable is only alive inside the body of the for loop containing it.
    - test_data: used as a temporary storage variable for calculations. This variable is only alive inside the body of the for loop containing it.

    - forecast_data: used as a temporary storage variable for calculations. This variable is only alive inside the body of the for loop containing it.
    - denominator: used as a temporary storage variable for calculations. This variable is only alive inside the body of the for loop containing it.

----

5. Tree

By Sam Peters

tree.py: Tree.py contains all the classes and methods used to implement executable trees and pipelines of operator functions.

Dependencies

Tree.py has 4 dependencies which are:

- pickle: Saving and loading python objects to and from binary files
- copy: Used to create deep copies of nested data structures
- queue: Queue data structure used during tree execution
- preprocessing: Used to create TimeSeries objects at the root of each tree.

Relation to Other Source Code Files

The tree is designed to contain and execute operator functions from the preprocessing, modelingAndForecasting, and visualization, but is built in a flexible way where functions from other libraries can be added instead if the data scientist modifies the input_keys and output_keys dictionaries to specify what input/output the other functions take. Section 6 specifically covers the use and format of input_keys and output_keys dictionaries in more detail.

Classes

TransformationTree

Class used to hold and execute branches of operators.

Methods:

__init__(self, operator, args, parent=None, tag="", save_result=False) -> TransformationTree

- Initializes a Node object containing operator, tag, args, parent, children, and save_result attributes
- Args:
    - operator (function): function to be executed when self.apply_operator method is called
    - args (list): list of positional arguments to be included when calling self.operator function
    - parent (Node, optional): Parent of the initialized Node in the tree. Defaults to None.
    - tag (str, optional): Tag used to identify and find the initialized Node in the tree. Defaults to "".
    - save_result (bool, optional): Dictates whether the result of calling self.apply_method should be saved during tree execution. Defaults to False.
- Returns:
    - TransformationTree: Created TransformationTree object

_execute(self, path=None)-> None

- Generic tree execution method called by self.execute_tree and self.execute_path. Modifies self.results.
- Args:
    - path (list, optional): List of nodes in a path to execute. If None whole tree will be executed. Defaults to None.

execute_tree(self) -> None

- Execute the entire tree by calling self._execute() without a path argument. Modifies self.results execute_path(self) -> None

- Executes a single path in the tree by calling self._execute(path=end_node). Modifies self.results.

get_nodes_by_operator(self, tag) -> [Node]

- Finds all nodes in the tree containing the input operator function (operator).
- Args:
    - operator (function): function to find
- Returns:
    - [Node]: List of Nodes containing the same operator function as the input operator function.

get_nodes_by_tag(self, tag) -> [Node]

- Finds all nodes in the tree with a tag equal to the input tag. Calls self._get_nodes()
- Args:
    - tag (str): tag to find
- Returns:
    - [Node]: List of Nodes containing the same tag string as the input tag string

_get_nodes(self, value, mode: str) -> [Node]

- Generic tree search method that finds Nodes in the tree with a given value. Used by self.get_nodes_by_tag and self.get_nodes_by_operator.
- Args:
    - value (str or function): the value the function checks for
- Returns:
    - [Node]: List of Nodes containing the given value

add_operator(self, operator, args, parent_node, tag="", save_result=False) -> Node

- Adds a Node containing a new operator to the tree.
- Args:
    - operator (function): Operator function to be called during tree execution
    - args ([Any]): Positional arguments using during operator function call

    - parent_node (Node): Parent of the new Node, must be an existing Node object in the tree.
    - tag (str, optional): String used by users to identify specific important Nodes later on. Defaults to "".
    - save_result (bool, optional): Specifies whether the result of executing the operator function should be saved to self.results,
    - for additional processing after the tree is finished executing. Defaults to False.
- Raises:
    - CompatibilityError: Signifies that two successive operator functions are incompatible with each other.
- Returns:
    - Node: Newly created Node in the tree.

replace_operator(self, operator,args, node, tag="", save_result=False) -> Node

- Updates the attributes of an existing node with new operator, args, tag, and save_result values. Calls self.add_operator()
- Args:
    - operator (function): New operator function
    - args ([Any]): New list of positional args to be passed to the operator function during execution
    - node (Node): Node to be updated
    - tag (str, optional): New tag value. Defaults to "".
    - save_result (bool, optional): New save_result value. Defaults to False.
- Returns:
    - Node: Newly created Node in the tree.

replicate_path(self, start_node: Node, end_node: Node) -> Node

- Add a copy of a path of nodes to the tree. Path goes from start_node to end_node. New path is added as a child of start_node.parent
- Args:
    - start_node (Node): First node in the copied path
    - end_node (Node): Last node in the copied path
- Returns:
    - Node: first node in replica tree path

_copy_node(self, node) -> Node

- Creates an exact copy of a given Node, except for the Node's children and parent attributes, which it sets to empty. Used internally.
- Args:
    - node (Node): Node to be copied.
- Returns:
    - Node: Returns reference to Node copy

_modify_tags(self, subtree_root, modifier) -> None

- Modifies all the nodes with tags in a given subtree by appending the modifier string to the their current tags
- Args:
    - subtree_root (Node): Node at the root of the subtree
    - modifier (str): String to be appended to modified tags

_check_compatibility(self, new_node, parent_node) -> Bool

- Checks if the required input for the new node's operator matches the expected data types that will be passed to the new operator by the previous nodes in the tree branch.
- Args:
    - new_node (Node): node to be added to the tree branch
    - parent_node (Node): potential parent of new_node, already in the tree branch
- Returns:
    - Bool: True if the required input of the new_node matches the expected data types generated by the branch. False otherwise.

get_path_str(self, end_node) -> str

- Returns string representing path from the root node of the tree to a given node (end_node)
- Args:
    - end_node (Node): The last node in a path of the tree
- Returns:
    - str: String representation of a path of nodes

export_pipeline(self, end_node) -> Pipeline

- Creates and returns a Pipeline object containing the path from the root of the tree to a given node (end_node)
- Args:
    - end_node (Node): The last node in the path that will be added to the pipeline
- Returns:
    - Pipeline: a Pipeline object containing a given path

Pipeline

Static pipeline of operator functions that process data. Can be executed but not edited.

Methods:

__init__(self, tree: TransformationTree, pipeline_end_node: Node) -> Pipeline

- Initializes a Pipeline object containing tree, end_node, and results attributes
- Args:
    - tree (TransformationTree): Transformation tree containing the branch we want to make into a pipeline
    - pipeline_end_node (Node): Last node of the branch we want to make into a Pipeline.
- Returns:
    - Pipeline: a Pipeline object containing a given path

run_path(self) -> None

- Executes a Pipeline path and modifies self.results.
- Calls:
    - TransformationTree.execute_path: calls this method to run a path of the tree

<u>Node</u>

Node class used in tandem with TransformationTree class. Holds and executes operator functions.

Methods:

__init__(self, operator, args, parent=None, tag="", save_result=False) -> Node

- Initializes a Node object containing operator, tag, args, parent, children, and save_result attributes
- Args:
  - operator (function): function to be executed when self.apply_operator method is called
  - args (list): list of positional arguments to be included when calling self.operator function
  - parent (Node, optional): Parent of the initialized Node in the tree. Defaults to None.
  - tag (str, optional): Tag used to identify and find the initialized Node in the tree. Defaults to "".
  - save_result (bool, optional): Dictates whether the result of calling self.apply_method should be saved during tree execution. Defaults to False.
- Returns:
  - Node: the newly created Node object

apply_operator(self, dynamic_data: list) -> Any

- Calls self.operator, passing in the values contained in dynamic_data and self.args as positional arguments
- Args:
  - dynamic_data (list): List of values to be passed in as positional arguments during execution of operator function
- Returns:
  - Any: The result of calling self.operator

__str__(self) -> str

- Returns string representing Node object

- Returns:
    - str: String containing operator name, or the operator name and tag string

__repr__(self) -> str
- Returns string representation of Node object
- Returns:
    - str: Takes the form of "Node(operator_name)" or "Node(operator_name):tag" depending on if the node's tag == ""

<u>Other Functions</u>

save(object, filename) -> bool
- Saves a TimeSeries or Pipeline object to a file.
- Returns:
    - bool: Boolean value representing success (True) or failure (False) of the operation

load(filename) -> Any
- Loads a TimeSeries or Pipeline object to a file.
- Returns:
    - Any: loaded object, type depends on type of object loaded. False if operation failed.

--

6. Operator Keys

By Brian Gunnarson

operatorkeys.py: operatorkeys.py contains two dictionaries that are crucial for the functionality of tree.py. One dictionary is related to the type of input a function or method takes and the other is related to the type of output a function or method produces. Note that these inputs and outputs use strings to describe what is available for us to use at a certain point in building the tree branch. For instance, we can’t go onto modelingAndForecasting.py functions since they require x_train and y_train without first using the ts2db method in preprocessing.py, since that is where these variables are produced.

Dependencies

operatorkeys.py has 3 dependencies which are:

- preprocessing: to import all of the methods associated with a TimeSeries object
- modelingAndForecasting: to import all of the functions defined in this module
- visualization: to import all of the functions defined in this module

Relation to Other Source Code Files

The operator keys module for this program is related to 5 other source code files:

- preprocessing.py: This is used for describing all of the inputs and outputs of the methods in the TimeSeries object.
- modelingAndForecasting.py: This is used for describing all of the inputs and outputs of the functions defined in this module.
- visualization.py: This is used for describing all of the inputs and outputs of the functions defined in this module.
- tree.py: This module uses the dictionaries defined in operatorkeys.py in order to help build branches without any errors.
- ts_analysis_support.py: Imports operatorkeys.py and all other source files as a way to keep them all in one place for ease of use.

Purpose and Inner Workings of operatorkeys.py

Operatorkeys.py contains two dictionaries that help tree.py build branches of the tree without any errors. When tree.py builds a branch, it adds one node at a time. This node contains an operator (a method or function defined in one of our source files) so when we add this operator to the branch we need to make sure if the output from it's parent node can be used as input for the current node.

The first dictionary defined in operatorkeys.py, operator_input_keys, uses every method and function defined throughout preprocessing.py, modelingAndForecasting.py, and visualization.py as keys. The value associated with all of these keys is the type of input that these methods or functions take in. Similarly, the second dictionary, operator_output_keys, also uses every method and function defined throughout preprocessing.py, modelingAndForecasting.py, and visualization.py as keys. Their associated values are the outputs that these methods or functions produce.

Purpose, Scope, and Lifetime of Variables

The only variables defined in operatorkeys.py are operator_input_keys and operator_output_keys. These are both global variables and are imported into tree.py for further use.