ChronoWave
Programmer Documentation/Installation

Table of Contents

1. Installation

This section describes the requirements for installing and setting up ChronoWave on a local host. You should have the zip folder containing all the source code and documents and you should extract the zip for use.

1.1 Installing The Requirements

The main zip folder should contain requirements.txt file which can be installed via the command line as follows:

Windows: $pip install -r requirements.txt

Mac or Linux: $pip3 install -r requirements.txt

-Note: pip should come with any python installation

1.2 How to Run:

To launch the site on your local host server first you need to enable local hosting on your computer:

Windows

1. Go to Windows start button and click Control Panel
2. Click “Programs” link
3. Click the box labeled ”internet information services” and click “OK”
4. Reboot the computer

After you are able to use your computer’s local host, you can now simply navigate to the folder containing the main.py file that was installed with the zip folder and run the following:

Windows

$python main.py

MacOS/Linux:

$python3 main.py

The url should be displayed within the terminal and it should be http://127.0.0.1:5000

2. Programmer Documentation

2.1 Flask API (main.py)

Overview

The Flask API is mainly composed within our “main.py” file. This file contains all of our flask routes: (e.g. “/home”, “/” ,“/upload_data” …) and some functions for user file uploading such as allowed_file() and clean_working(). This is where the majority of the communication between the user and the database occurs. The user makes a request and the API uses a database manager module instance to access the required data.

Flask Route Contents

Each flask route is associated with one and only one html file. For example the file upload_data.html is accessed by the flask app through the route “@app.route(“/upload_data”, methods=[“POST”, “GET”].

Within each flask route is where we prepare any data that needs to be fetched from the database or stored in the database for that particular html file. We may use the db_manager instance to upload data that was retrieved from the user or to retrieve data from the database to display data to the frontend. We may also use the DataAnalyzer() from our metrics.py

Instances

This file imports our database manager and creates the database manager at the top of the file and assigns the instance to the variable name **“db_manager”**.

A flask instance is created with the variable name **“app”**. Upon instantiation we define the app's properties of “WORKING_DIR” to be set to the location of our working directory for temporary storage of files and data. Along with that we also define “ALLOWED_EXTENSIONS” which will be the allowed files for uploading data to the database.

A DataAnalyzer instance is created within the “/upload_forecast” route if the user has sent a POST request and a file with it. Specifically this class is used to calculate all of the metrics for a given forecast to store in the database.

2.2 Frontend Framework (templates)

Overview

The frontend framework consists of html files within the templates folder. There are html files such as base.html, navigation.html and footer.html which are foundations of each page. In other words each html page is contained within the base.html file. Other html files representing specific pages are then referenced within the block's of the base.html file to display the page. Having a consistent navigation bar, base and footer allowed for easy implementation of each page.

Universally Used Html's

(base.html) The base.html file uses jinja2 templates which are installed with flask to create blocks within the file that act as place holders for other html files. So each page is made up of the base.html and then the navigation.html is inserted into the block with "include navigation.html". The actual page changes depending on the selected flask routes by the user. When the user selects a html from the navigation bar (i.e. "upload_data.html") is inserted into the block content location within the base.html to display that html.

(navigation.html) The navigation.html contains all of the hyperlinks to specific html pages. This is displayed at the top of the base.html file by including it in a jinja2 template block. Navigation contains href links to all of the flask routes that should be accessible from the navigation bar. The url_for function is used to retrieve the flask app route and create a hyperlink to it.

(**footer.html)** This is mainly a visual html that is displayed at the bottom of the page.

Page Specific Html's

(admin.html) Admin html used for deleting sets and all metadata associated with that set. Is only accessible through typing in the url.

(download_data.html) A user can visit this page to download our time series set users can choose between .json, .csv, or .xlxs files for their data.

(help.html) Users can visit the help page to obtain any information about the functionality of ChrnoWave and to address any issues.

(Index.html) The home page for ChronoWave.

(performance_metrics.html) Users can visit this page to display any time series set's metadata and can select a specific forecast of that set to view the calculated metrics.

(upload_data) The html used for user uploading time series sets.

(upoad_forecast.html) Users can select a time series set and insert training data to create a forecast and store it in the ChronoWave database.

Jquery/AJAX

The following html files use ajax and jquery scripts to dynamically display time series sets within the database and allow the user to select a set to see metadata and potentially perform an operation on that set:

"download_data.html", "admin.html", "performance_metrics.html", "upload_data" and "upload_forecast".

Most of these scripts are used to dynamically display metadata about a selected time series set. Performance Metrics dynamically displays metadata about a set along with another selection of forecasts within a set. The forecast can be selected to display any metric calculations stored in that forecast.

2.3 Modules

Overview

The modules folder contains the following files: database.py, internal_data.py and metrics.py. Each has its own functionality and provides usage to the Flask API. The database_manager.py actually does all the communicating with the MongoDB database for ChronoWave.

Database Manager (database.py)

The Database Manager module defines the class: DatabaseManager. This class contains tons of methods for performing operations on the database. It requires the parameter arguments of a working directory and a PyMongo Client Url. There are numerous methods that can be used to communicate with the database in use, some important ones are: get_timeseries_set, store_timeseries_set and _ import_timeseries_set.

There are many more methods that are used and are important they can be visited in the database.py file for more information.

Internal Data (internal_data.py)

The internal data file contains the following classes: TimeseriesSet, Timeseries, DataSet, TimeSeries Descriptor, Forecast, ForecastingTask. These classes compose how the data will be structured in the mongodb instance. For example a Time Series Set may contain a list of time

series, and a time series may contain a list of datasets. So to simplify the process of sending data to the database these classes were made. A TimeseriesSet will contain a list of Time Series class objects. Many other classes contain other objects from other classes.

The main functionality of all of these classes are used within the database.py file. That file will use these classes to send data to the user or to retrieve data from the user and send it to the database.

Metrics (metrics.py)

This python file contains the DataAnalyzer class that can be used to create forecasts with the calculated metrics within them. This is where the calculations for the mae, smape and other metrics are performed by using the calculate_metrics() method.

The main flask route that uses this class is the upload_forecast route. The process is the user gives training data, then a DataAnalyzer class is created with this data, then finally the calculations are performed on that data and the results returned. The database manager instance then takes this data and stores it in the correct location for the forecast.

3. Acknowledgements

Contributors: Aleksandr Stevens, Isaac Perkins, Geoffrey Brendel, Cynthia Meneses