Detecting and Predicting Clusters of Evolving Stellar Systems

Final Report

Client/Advisor: Goce Trajcevski

Team Members:

Adam Corpstein

Joel Holm

Willis Knox

Becker Mathie

Philip Payne

Ethan Vander Wiel

Team Email:

sdmay21-30@iastate.edu

Team Website:

https://sdmay21-30.sd.ece.iastate.edu

04/25/2021

Table of Contents

Project Introduction

Acknowledgement

The team would like to thank the Electrical and Computer Engineering department at Iowa State University for the resources and education we have received throughout our time spent at the university. We would also like to thank Professor Goce Trajcevski for his efforts in and outside our weekly meetings. His feedback and advice has guided our team greatly.

Problem and Project Statement

As astrophysicists continue to record data for stellar systems, new possibilities for research emerge. Large databases exist full of stellar data. This includes attributes such as red shift, mass ratio, hydrogen ratio, etc. Astrophysicists would like the ability to organize this data into clusters so that any two systems with similar attributes are in the same cluster and dissimilar systems are in different clusters. Astrophysicists are generating their own data for future states of stellar systems. This would add the dimension of time to the data (since the lifetime of a stellar system could be millions of years, simulating future states is necessary). By organizing clusters by time, further analysis can be taken as this stellar data mature.

The driving force for this project is to organize stellar data to aid astrophysicists in their research. To solve the problem of organizing stellar data, we've applied pre-existing clustering algorithms to the datasets. The datasets have had their data interpolated. Interpolating data allows any two stellar systems with different time scales to be compared. From the webpage astrophysicists can select their desired database, attributes, weight values, clustering algorithm, and other parameters. This data is sent to the web server to generate clusters. After generating the clustered data, users can visualize using 2D and 3D graphs. These graphs represent the data over a single or range of timesteps.

Intended Users and Uses

The end users for this project are astrophysicists who are researching and/or working with stellar data. These astrophysicists are used to working with stellar data provided by Sloan Digital Sky Survey or the Gaia Archive. They may be familiar with the query language SQL as it's used in these websites.

Assumptions and Limitations

We have come up with the following assumptions and limitations to refine the scope of the problem statement.

Assumptions

The user can connect to the internet

We are assuming that our intended users have access to the internet. Without an internet connection, a user does not have the capability to access the application, and therefore they will be unable to use it.

The user has a basic understanding of clustering and weighting functions

We are assuming that users have enough knowledge about the provided clustering techniques to understand how it affects the output data. Related to this, the user will need to provide their own weights in regards to each attribute. This also affects the output data, so users will need a solid grasp of how to write their own weighting functions.

Limitations

The application should not overuse the client's hardware

All algorithmic work, such as data normalization and star clustering, will need to be completed on the server and not on the user's device. This is to ensure the application stays responsive and easy to use.

Types of clustering data is limited to what is in the pre-existing databases

The different types of data the users will be able to select for clustering is limited to what information is stored in the databases the application will be accessing. Since we are connecting to well established, pre-existing databases, the users must restrict their clustering queries to types of data that these databases currently contain.

Requirements

Functional Requirements

- A web based application with a user interface for analyzing stellar evolutions.
- Users are able to configure which parameters are of interest for clustering
- Users are able to apply specific weights to their selected parameters that determine the level of importance of each parameter when clustering
- Users have the ability to specify specific intervals of interest for clusters
    - This interval could either be an explicit time range or a more general event based range

- Given the user specified information, a remote server will access stellar data from a database and then compute how different stellar systems will cluster during their evolution
- The possibility for additional databases to be added for clustering functionality. These databases need not be related to stellar data. Any data that evolves with respect to time may be utilized.
- Databases added will be interpolated based on timesteps. This will create a uniform time scale for every data set.

Non-Functional Requirements

- Cluster requests are placed into a queue system to allow the user to make multiple requests at once.
- The server is equipped to handle less than 100 simultaneous users.
- Server should be available 24/7 with weekly server resets

Environmental Requirements

- The application requires internet connection to be functional. This connection needs to be constant and stable to allow users to access the application at any time
- The remote server needs to have the capability to connect to multiple databases

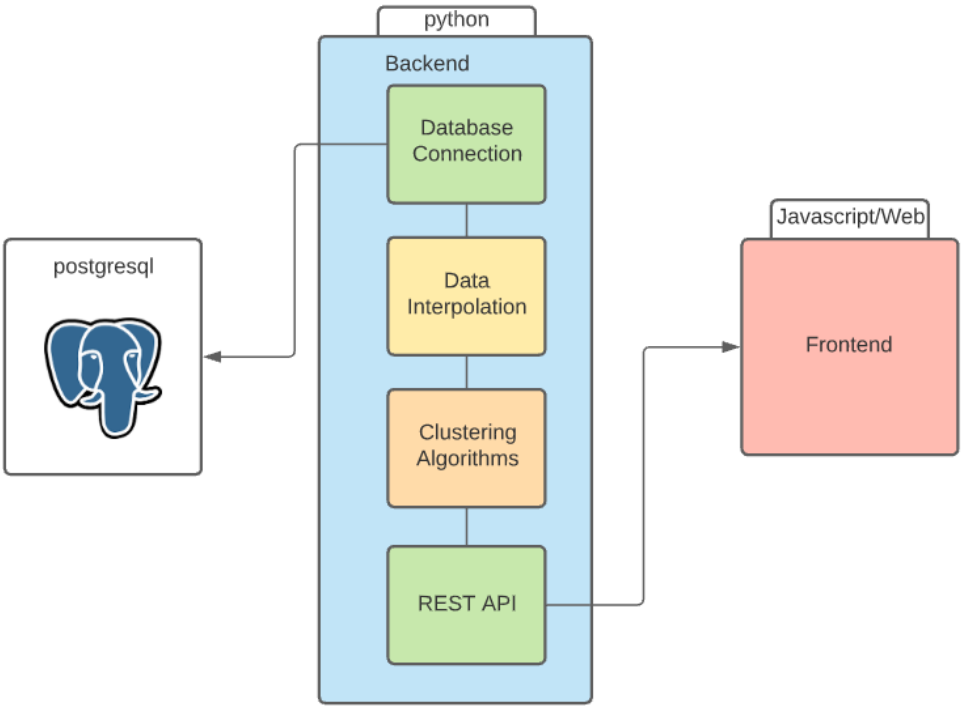
Economic Requirements

- Laptops or personal computers are needed to interact with the user interface of the application
- Iowa State University has provided our team with a remote server to host our application. All costs associated with the server are paid for by Iowa State

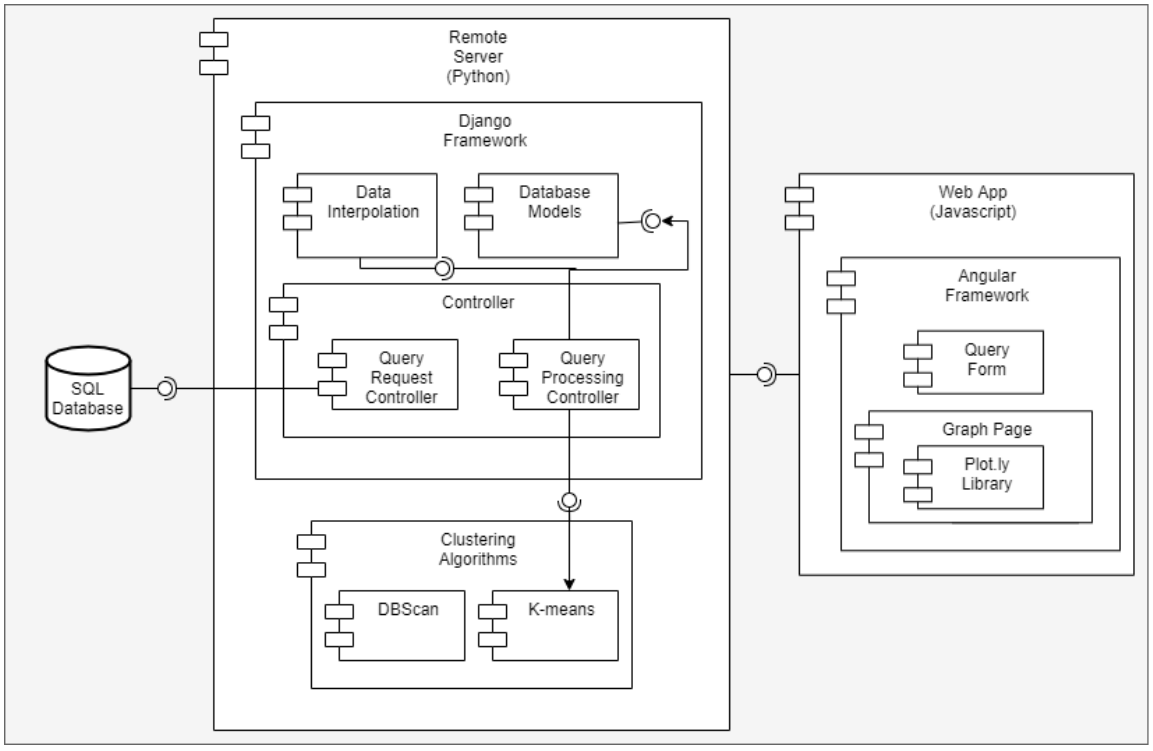
Revised Project Design

To start, our system handles events on the frontend. Here, a user chooses different algorithms, weights of data, and output styles they want. The frontend compiles this information and sends it to the backend via a REST request. The backend processes the received information, queries for data using an SQL connection, and runs the simulation. After the simulation is completed, the backend will send the information back to the frontend as a REST response and the frontend will display it. This process is quick and efficient, but powerful enough to handle almost any desired simulation from the user.

Architecture Diagram



Component Diagram



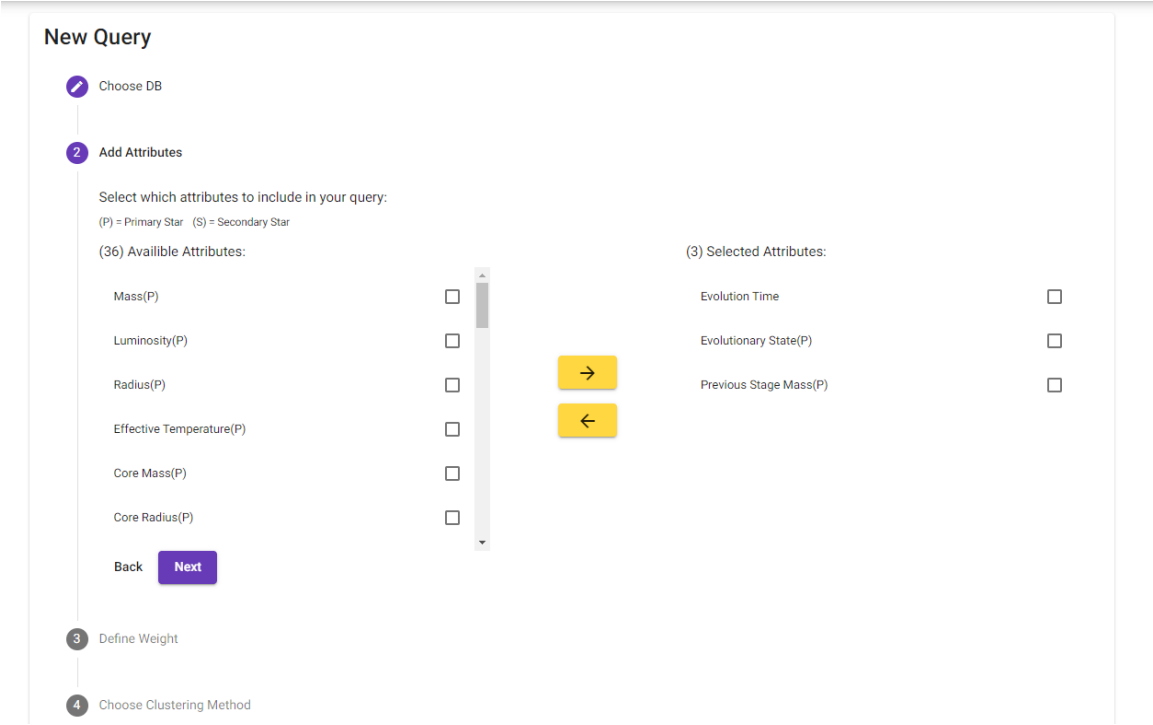
Implementation Details

Frontend Design

The front end was designed and mostly implemented without communication to the backend. Using angular, we were able to implement the initial design components and get a good prototype quickly which left us time to implement the more complicated data rendering and backend communicating parts of the frontend.

The most difficult part of the frontend was the data rendering. The data coming from our backend is complex and needs to be organized properly for effective visualization. This also required some research into the best graphing techniques to render clustering data.

Communication with the backend was not difficult as all modern javascript frameworks come ready to communicate with REST APIs.



Backend Design

Our backend needed to accomplish three main goals. First was to connect to our database and get relevant data for clustering. Next was to parse the data, apply needed normalization functions, and run our clustering algorithms. Lastly, we needed the backend to connect with our frontend through a REST API and send back the necessary results of our clustering.

Connecting the backend to our database was quite easy. Python can quickly connect to local PostgreSQL databases using environment variables for a username and password. We can run simple queries to get the requested star data and pass that to our clustering algorithms.

For clustering, we mostly used pre-existing libraries that performed the clusters for us. All the backend needed to do was prep the data to be processed by the clustering algorithms in a meaningful way.

Finally, we send the resulting data back to the frontend as an HTTP response that can easily be parsed.

Deployment

For deployment, we use two docker images, one for the UI and one for the API. These images are based on our testing environment so we can be confident that our application will perform the same in both production and testing environments. We use docker compose to deploy those images with all of our secrets - which includes passwords, configurations and more - to our production server.

Engineering Standards Used

In the implementation, there were two databases integrated into our backend. These databases followed standards presented in [1]. This paper describes a tool used by astrophysics. The tool is called MESAstar and it simulates stellar data. Given an initial set of attributes, it can predict how those attributes will change over long periods of time. If humans had recorded stellar data for the past millions of years, then we would use that data. Since our records are in their infancy, compared to the lifecycle of stars, we need to simulate data. Astrophysicists simulate data within the standards presented in this paper.

Testing Process and Testing Results

As for testing, our first step was to get CI/CD and basic unit tests working towards the beginning of our implementation. Ensuring that updates to the system don’t collapse our backend is a requirement of any software development team. Unit tests helped to maintain a consistent goal of the system despite any changes.

Frontend Testing

The front end is tested using the Cypress framework. This is a widely used framework that provides functionality for end-to-end tests. End-to-end tests run through a use case scenario. For example, one path that we test is a successful user query ending on the cluster visualization page. Since the tests are written in Cypress, in the future they can be used as regression tests. These will validate that previous requirements haven’t been impacted by new changes. We’ve also written robustness tests. These tests go over edge cases where user input is invalid. We test that the code responds in a healthy way so the application keeps running normally and the user is prompted to what they did wrong.

Backend Testing

For backend testing we wanted to focus on a few key parts of our application. The first is testing the REST API itself. We wanted to make sure requests to the backend were handled properly, including meaningful status codes, edge cases for poor input, and successful attempts to request information. Not only do we need to ensure users can only give valid information to the backend (through frontend testing), we need to be careful that if invalid information is requested that our backend handles it correctly.

The next thing we wanted to test is correct use of our algorithms. Since we are importing a lot of the key algorithms in our project, we are assuming the implementations are correct. However, we can still test how our backend gives and receives data through the algorithms.

Testing Results

One technical challenge we ran into with testing was verifying the results of our application. Since the information we produce is complicated in nature, it is challenging to make meaning out of our results. Above that, being able to verify that our final data is correct is a process that will need to be peer reviewed by scientists using the application.

With more time to implement our project, we would have added more interaction with the scientific community to ensure our data is meaningful.

Our goal with frontend testing was making sure use cases were covered and data is properly received and sent to the backend. The goal of the backend testing was ensuring that data is handled and used in a satisfactory way.

Related Literature

Clustering Algorithms

K-means

- One of the clustering algorithms we're using is K-means. This is a well known process and has many descriptions/tutorials on the internet. The main reference point for our implementation of K-means is derived from the article *K-means: A Complete Introduction* from the website towardsdatascience.com [2].

DBScan

- A second clustering algorithm is DBScan. This algorithm is also well known and documented on the website towardsdatascience.com under an article titled *DBSCAN Clustering — Explained* [3].

Both of these algorithms are already implemented in a common Python library: scikit-learn [4]. Our team decided to leverage this free to use library for this project instead of trying to directly rewrite these clustering algorithms ourselves.

Research Papers

The following research papers introduce similar problems that are solved via clustering algorithms. Concepts and implementation details are pertinent to our clustering of stellar data through time.

Computing Longest Duration Flocks in Trajectory Data

This paper presents the concept of monitoring flocks and meetings during a period of time. The concepts of flocks and meetings are expanded to include the specification varying or fixed; this is all a variation of our notion of clusters. Algorithms are provided for each cluster type and efficiency is discussed [5].

Discovery of Convoys in Trajectory Databases

This paper discusses three algorithms for finding clusters during a period of time. Each tracted node moves along a “trajectory”, computing which trajectories are closest to each other is the problem. The paper presents a density based clustering algorithm and adds optimizations to increase efficiency [6].

Closing Material

References

[1] Bill Paxton, Matteo Cantiello, Phil Arras, Lars Bildsten, Edward F. Brown, Aaron Dotter, Christopher Mankovich, M. H. Montgomery, Dennis Stello, F. X. Timmes, and Richard Townsend. “Modules For Experiments In Stellar Astrophysics (MESA): Planets, Oscillations, Rotation, And Massive Stars”. *The American Astronomical Society (The Astrophysical Journal Supplement Series).* 208 (3), August 2013.

[2] Jeffares, Alan. K-means: A Complete Introduction. https://towardsdatascience.com/k-means-a-complete-introduction-1702af9cd8c, 2019.

[3] Yildirim, Soner. DBSCAN Clustering — Explained. https://towardsdatascience.com/dbscan-clustering-explained-97556a2ad556, 2020.

[4] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gael Varoquaux. “API design for machine learning software: experiences from the scikit-learn project”. ECML PKDD Workshop: Languages for Data Mining and Machine Learning. 108-122. 2013.

[5] Gudmundsson, Joachim, and Marc van Kreveld. (2006, November). Computing Longest Duration Flocks in Trajectory Data. Proceedings of the 14th Annual ACM International Symposium on Advances in Geographic Information Systems, 35-42.

[6] Jeung, Hoyoung, and Man Lung Yiu, Xiaofang Zhou, Christian S. Jensen, Heng Tao Shen. Discovery of Convoys in Trajectory Databases. https://doi.org/10.14778/1453856.1453971, 2010

Appendix A - Design Document

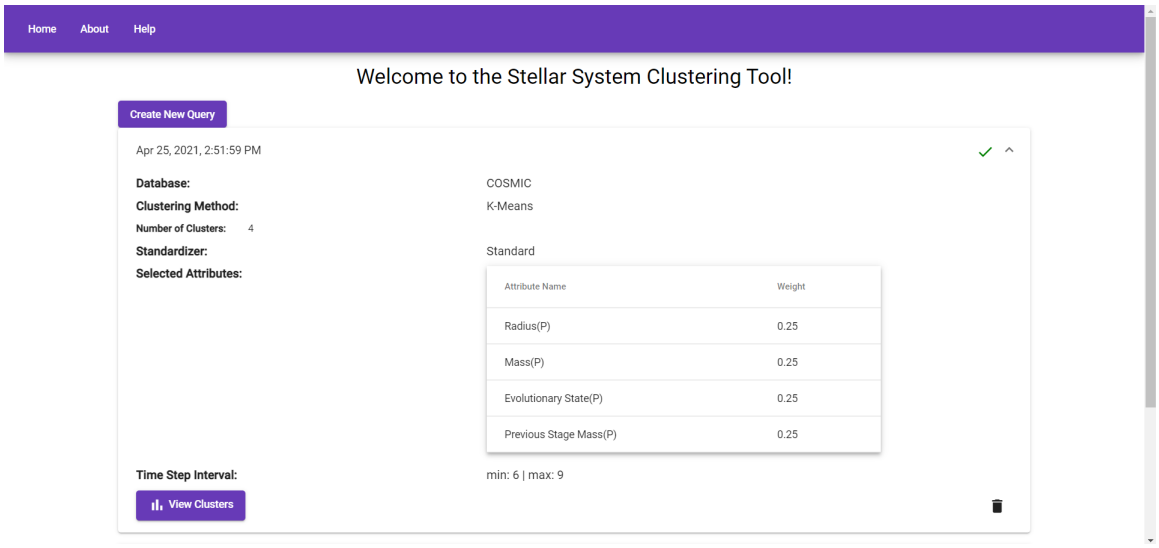
For more information about our plan for this project, we have provided a link to our design document from the previous semester. Details about our project timeline and technology considerations are explained in this document.

https://sdmay21-30.sd.ece.iastate.edu/docs/design_doc_final.pdf

Appendix B - User Manual

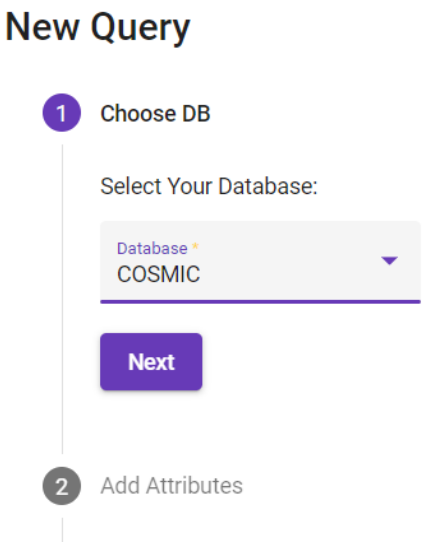
Home Page

The home page is the entry point for the application. It allows the user to Create a New Query, visit the About Page, or the Help Page. The About page contains background information about our product, including a clustering explanation, database explanation, and developer/project information. The Help Page contains this User Manual. Below the Create New Query button is a list of saved user queries. After submitting a query, it is automatically saved to this list.

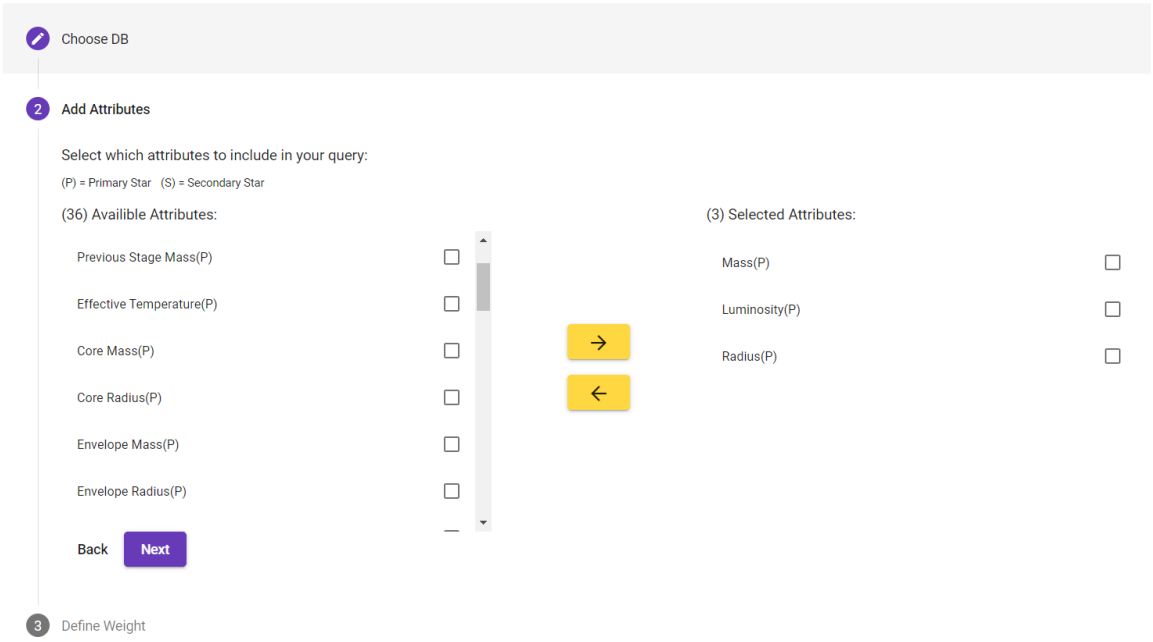


Query Page

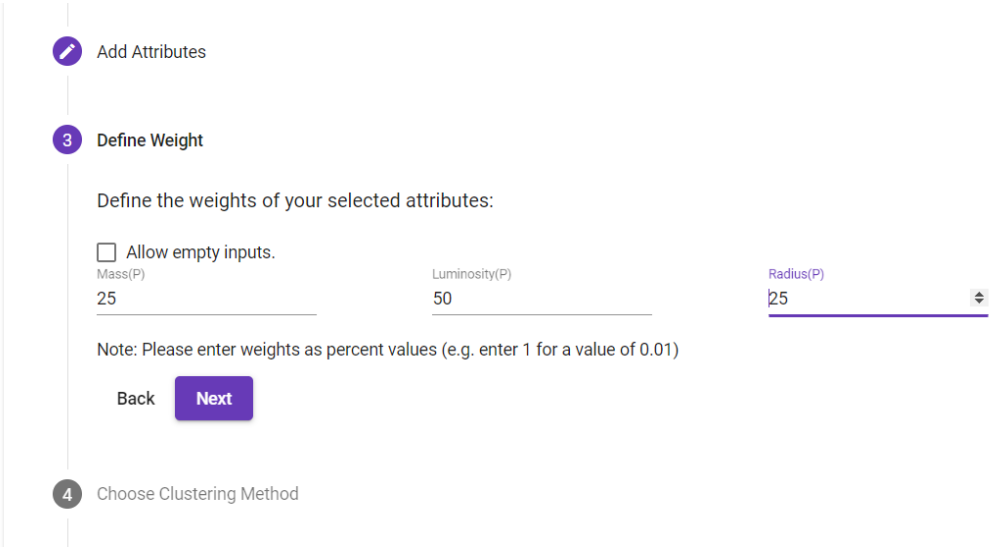
The first step when creating a query is selecting the database. Two databases are implemented for this version, COSMIC and POSYDON.



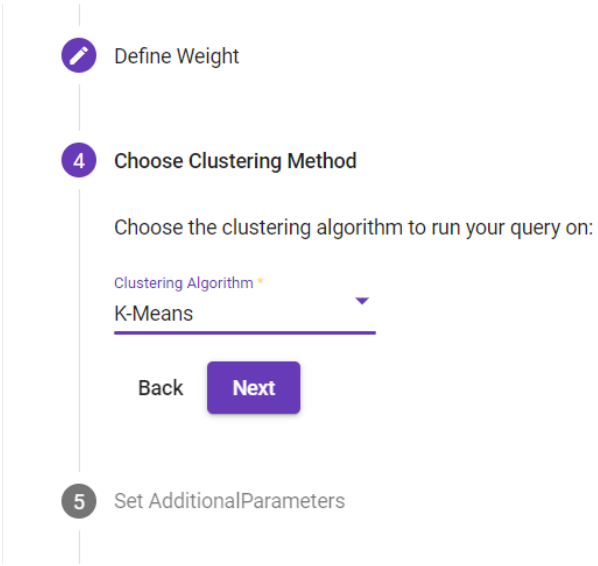
The second step is to select the attributes to cluster with. At least one attribute is required.



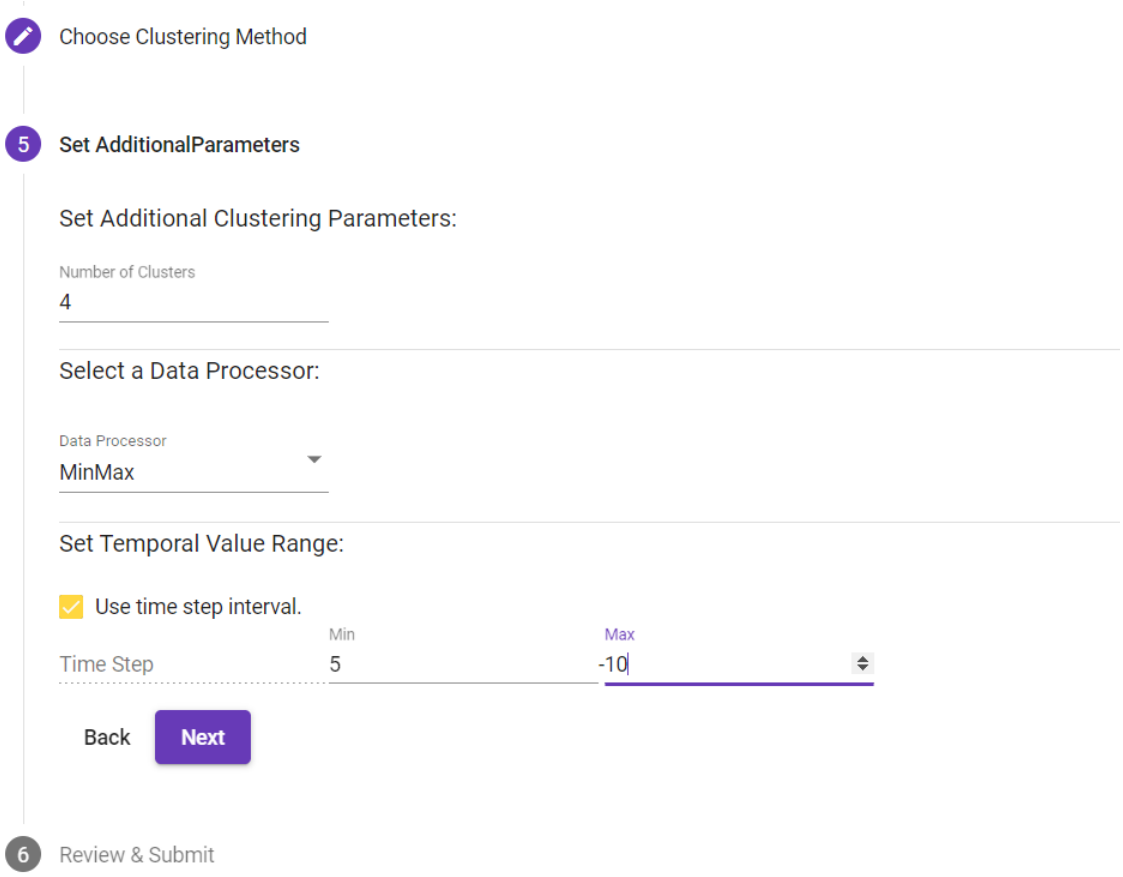
Third, attributes can be weighted. The larger the weight of an attribute, the more clustering will consider it. The sum of weights must add to 100. Selecting “Allow empty inputs” will evenly weight all attributes.



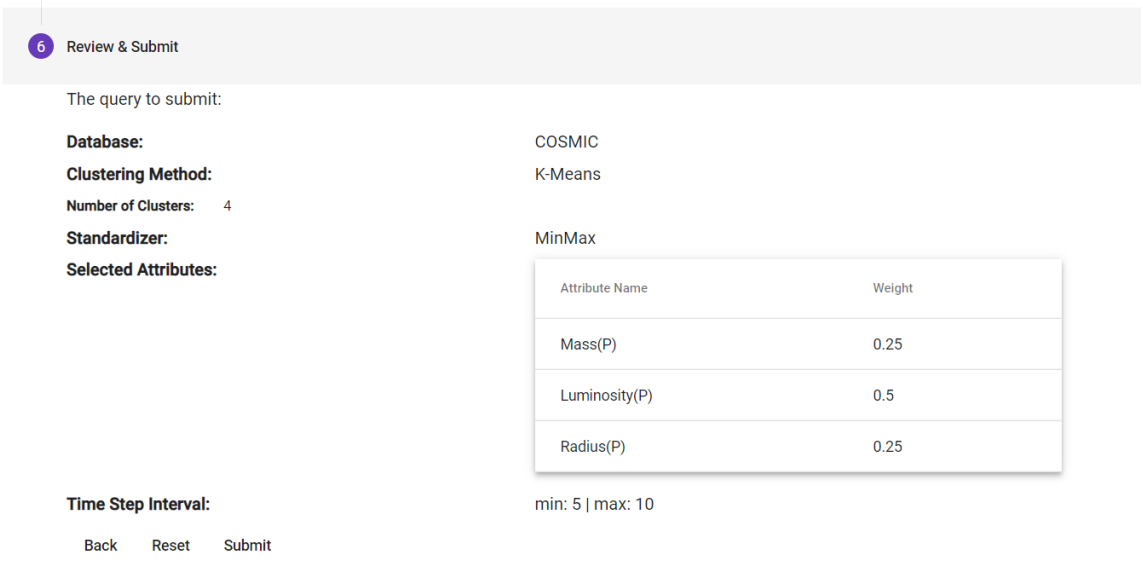
Fourth, the user can select which clustering algorithm to use. The two algorithms, K-Means and DBScan, each have their own pros and cons. More detail is located in the About Page.



Fifth, based on the clustering algorithm selected, users may need to provide additional parameters. For example, for K-Means users need to specify the number of clusters.

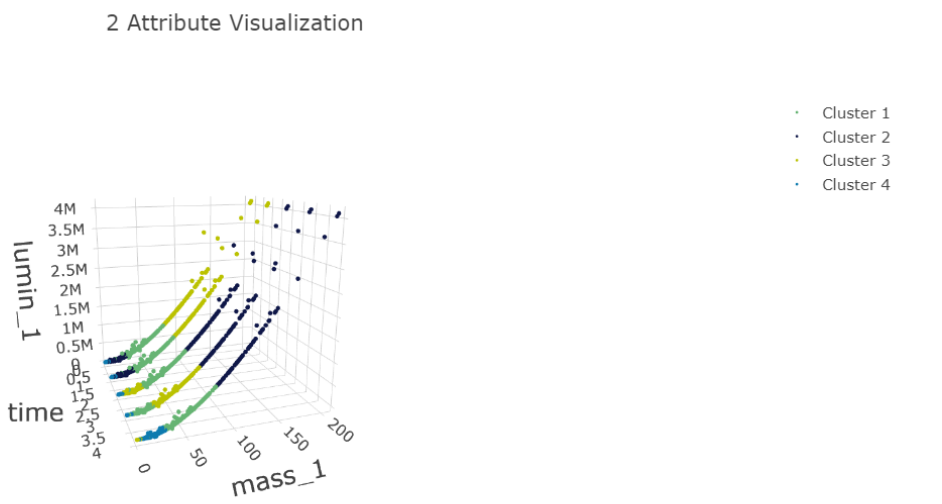


Lastly, users can review all input into the form. When satisfied, the query can be submitted to the backend.

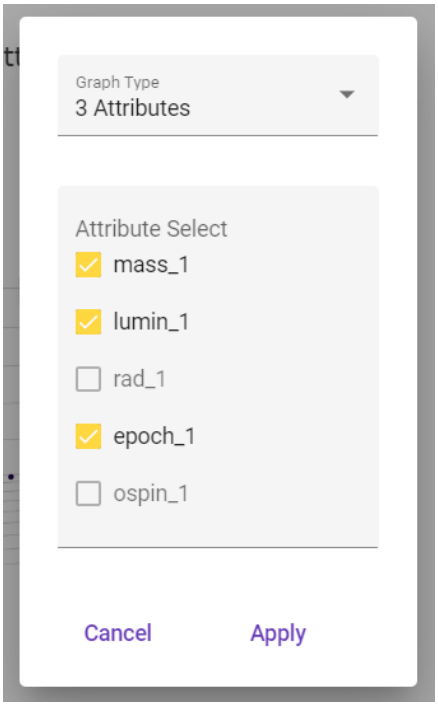


Graph Page

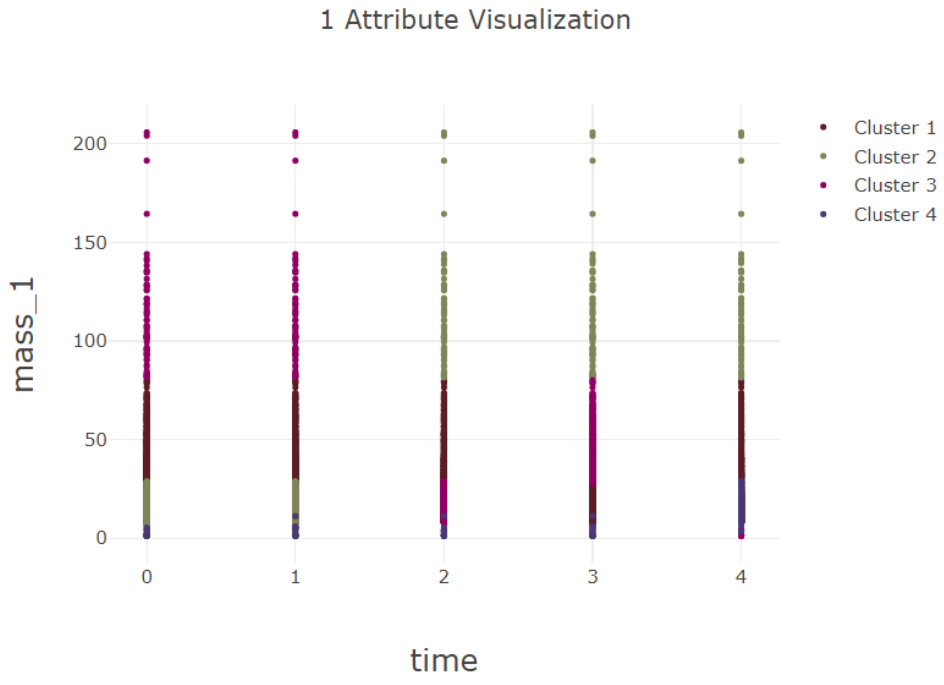
After a query submission, users can visualize their data with 2D and 3D graphs. The default graph on query submission is the 2-attribute display over time.



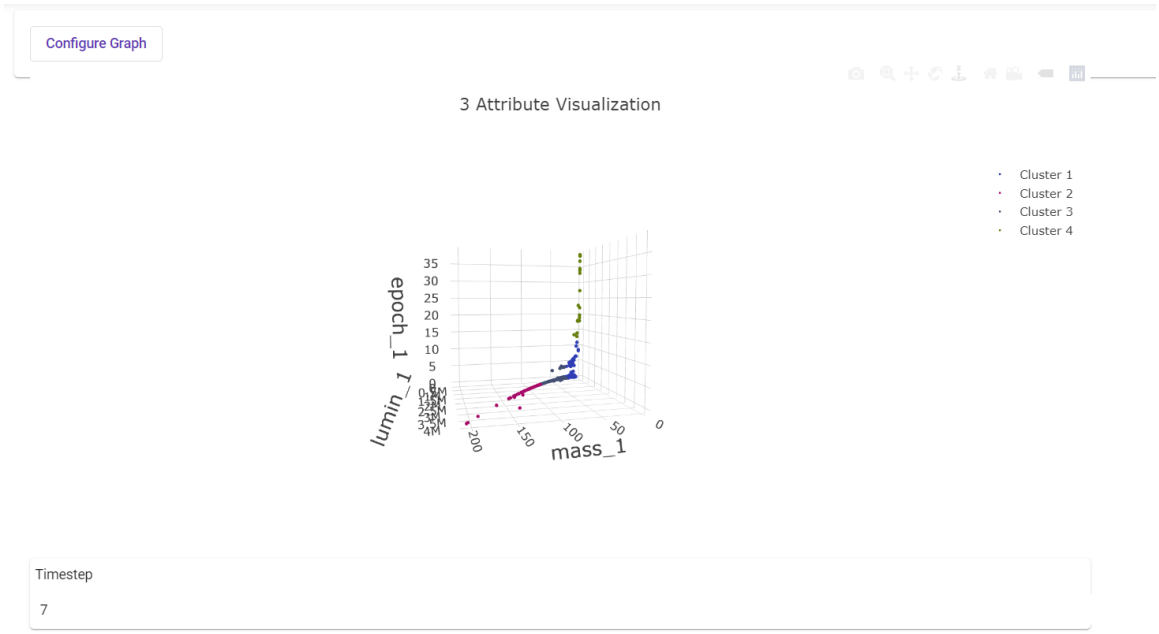
The user can select different combinations of attributes to display using the configure button. The type of graph can also change. The following are possible graphs: 1 attribute over time on a 2D graph, 2 attributes over time on a 3D graph, or 3 attributes at single time steps on a 3D graph.



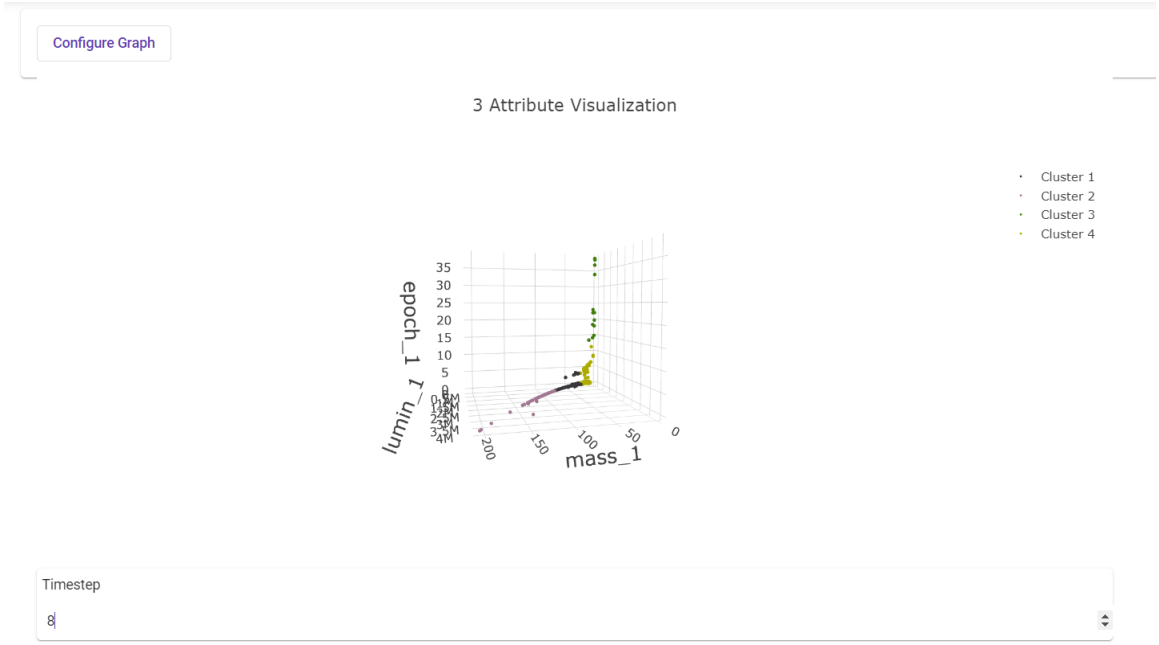
1 attribute over time on a 2D graph:



3 attributes at single time steps on a 3D graph:



Users can move the time step up and down to view how clusters change over time (Notice that Timestep has changed from 7 to 8).



Appendix C - Deployment Manual

Creating Tables

To create all the necessary tables, run python manage.py migrate in the API folder to automatically create tables. If you are confused about what tables you may or may not need, refer to models.py in the API project. Those models should describe all the columns you need.

Importing Data

We imported a COSMIC database given to us into postgresql from a csv format, but we had to break up our database into multiple files due to a limitation of postresql. We then added a table that includes this information about each attribute in the stellar data. If you already did a database migration like described above, this attributes table should already be created.

Table Name: **attributes**

database_name **(TextField)**	display_name **(TextField)**	enabled **(Boolean)**

We got the attribute list from this resource - [Describing the output of COSMIC/BSE: Columns names/Values/Units — cosmic 3.3.0 documentation (cosmic-popsynth.github.io)]

We created an example csv file you can import particularly for COSMIC databases in API/data in our project.

Once you completed that your attributes table should look like:

	database_name [PK] text	display_name text	enabled boolean
1	b_0_1	Neutron Star Magn...	true
2	b_0_2	Neutron Star Magn...	true
3	bin_num	Unique binary Index	true
4	bin_state	State of the Binary	true
5	deltam_1	Mass Transfer Rat...	true
6	deltam_2	Mass Transfer Rat...	true
7	ecc	Eccentricity	true
8	epoch_1	Epoch(P)	true
9	epoch_2	Epoch(S)	true
10	kstar_1	Evolutionary State(...	true

Deploying Application

For deployment, we leverage docker-compose. To deploy, make sure you have the following environment variables set.

Environment Variable	Description
DB_URL	The url of the database
DB_USER	Username for database
DB_PASSWORD	Password for database
ENV	Schema for database

Once you have these environment variables set, simply run this command from the command line from the API directory.

docker-compose -f docker-compose.yml up -d

Docker compose should bring up both the API and UI and can be accessed from the host machine. Port 8000 is used for the API and the UI uses the HTTP standard port 80.