Final Report: Medicare Doctor Data

Zhuoyi Zhan, Anubha Nagar

I. Introduction:

In this project, we want to create a website that is connected with the doctor performance database in a straightforward and accessible way. The website can help users know what they can do with the databases and fulfill their needs. By simply typing in the input field and clicking buttons, users can search for doctors, update doctors' performance scores, and delete doctors' information in the database. We decided to use Python with packages streamlit to create our website because it can format the website for us. The data set we used is about medicare's doctor performance score that is provided by the Centers for Medicare & Medicaid Services. It includes patient satisfaction surveys and performance scores that are collected from different medical institutes. The dataset was acquired from kaggle. It was created by the Centers for Medicare & Medicaid Services (CMS). This dataset contains the information supplied to patients via that website, including patient satisfaction surveys and performance scores across over 100 metrics. With this dataset and website, we want to help patients assess and find doctors and hospitals.

II. Implementation:

The project has three main use cases: searching for doctors by name, location, or performance scores, updating doctors' performance scores, and deleting doctors' information. The project is implemented and divided in two parts: the front-end and back-end design. We wrote Python code to create the content and outlook of the website. The database is created and controlled by SQL database. We started with the construction of the database. To better organize the database, we need to normalize the data in at least the third normal form to reduce redundancy. The data source already decomposed into two tables: the information and doctor's individual score. So we loaded them into mySQL database. Then we need to inspect dependency among attributes. The primary key that identifies each doctor is *PAC_ID*. However, we found out that there are two other variables that can uniquely identify doctors, *NPI,* and *Professional enrollment ID*. To reduce redundancy, we remove the other two variables. The possible key dependency we found is between *zip code*, *city*, and *state.* After testing this dependency, whether *zip code* can determine *city* and *state*, we separate the three variables into a new table.

The final database design consists of four tables. There is a *doctor information* table, which contains doctor's information like their names, gender, graduated school, etc. There is an *individual performance score* table, which has measure titles, measure performance rate, etc. There is the *zip* table mentioned above. We also created an empty *performance archive* table for future use of storing the deleted doctors' information. *doctor information, individual performance score, zip* tables have one to one relationship. The primary key for *doctor information* is *PAC_ID*. The *doctor information* table references the *zip* table by foreign key *Zip_Code*. Here we attach the UML diagram of all the tables in the database.

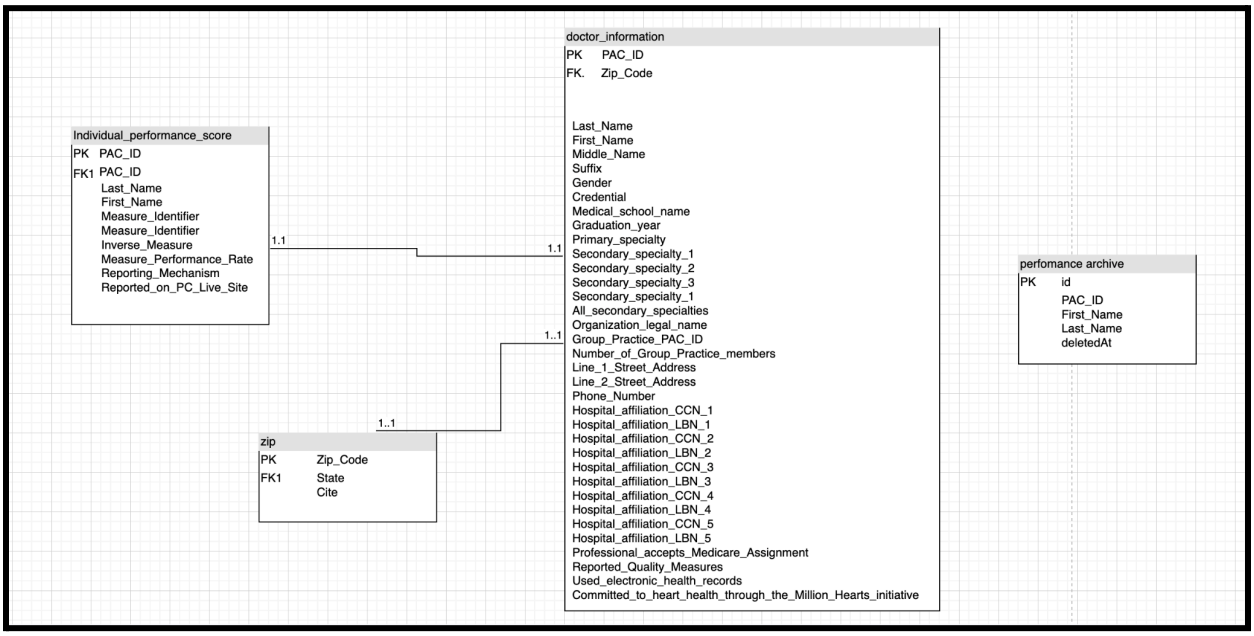

Figure 1: UML diagram for the medicare doctor database

III. Functionality Illustration:

There are many use cases that are executed in this project. The main idea is to be able to find doctors based on the patient's needs (be it location/performance). Ensure that patients are able to update doctor performance scores. Moreover, doctors who are not comfortable with their data being on the database can delete their data from the database by entering their PAC ID. In this section, we will explain each of these use cases in more detail.

III (a). Find a doctor:

Find a doctor feature is based on their first, middle and last name. The user enters their first name, middle name and last name on the website. The names typed need to be in capslock and there should be no error in spelling.

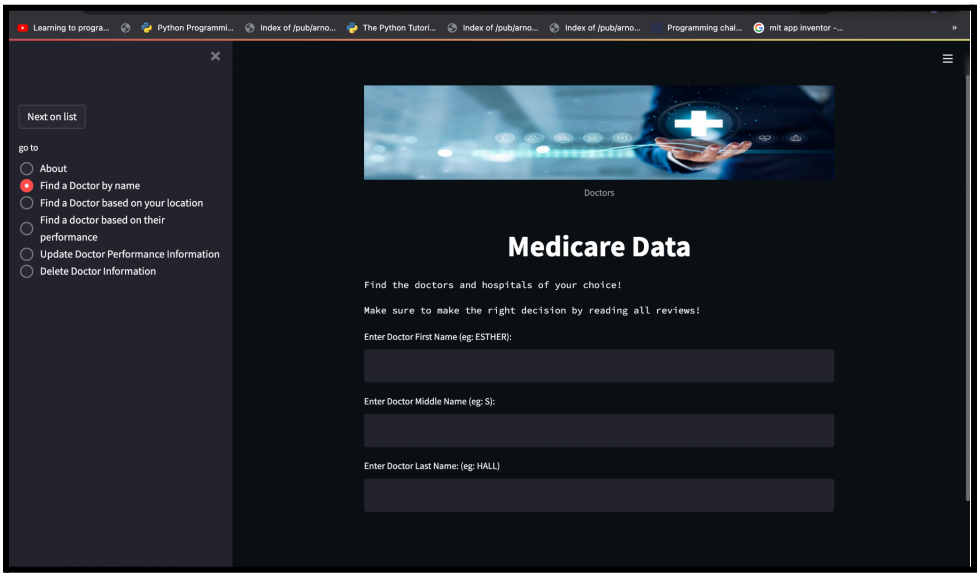

Figure 2: Find doctor page before input from user

Once the frontend has got the input from the user, then we import the package *mysql.connector* to connect to our database called Project2. Our connection was created using *mysql.connector.connect* function in python. This helped us successfully connect to the database. The database connection is called *mydb*.

Once the connection was successful we created a function called *run_query* that uses the database *mydb* and creates a cursor using *mydb.cursor() as cur.* After which, we write syntax for executing a query as *cur.execute(query).* After which, the *fetchall* function is used to extract the output. This function creation made execution of our code so much easier.

Now, once the person has entered the doctor name credentials, we check if the entered values are not blank and then proceed to running the query that selects the row from the database where the name is matching with the one entered. For this execution we use the *run_query* function that was generated. If it finds a successful match then it shows the PAC ID, First Name, Middle Name and Last Name. This will help the patient get the PAC ID of the doctor. Moreover, we can edit the code in a

way that we can print other data related to the doctor as well like where they did their degree from, how many people work under them and what their performance value is like(between 0-100).

III (b).Find a doctor based on location

In this page the user can choose the location, and based on this the doctor details will be printed. Here a patient can choose whether to get a doctor based on their city or state preference. Once the patient enters the city/state of their preference in capslock and there should be no error in spelling.

Once the person has entered their city (eg. NEW YORK), and clicks the submit button, the connection the database *mydb* is made using the method in III(a). Now we use the function *run_query* to run the query to use the database Project2. Now we check if the input is empty before moving on to the execution of our stored procedure. Now we use the function *cur.callproc()* to call the procedure *city_find/state_find,* if the person wants to find based on city or state respectively. In this stored procedure the joins the *megatable_p2* and the *zip* table. Then the stored procedure selects all rows where the city/state is based on what the user had entered. The table zip references Zip_Code in the megatable and hence can be joined based on that. Further, we have created a trigger which ….

Once the procedure is called, the doctor's details are printed for the user to see.

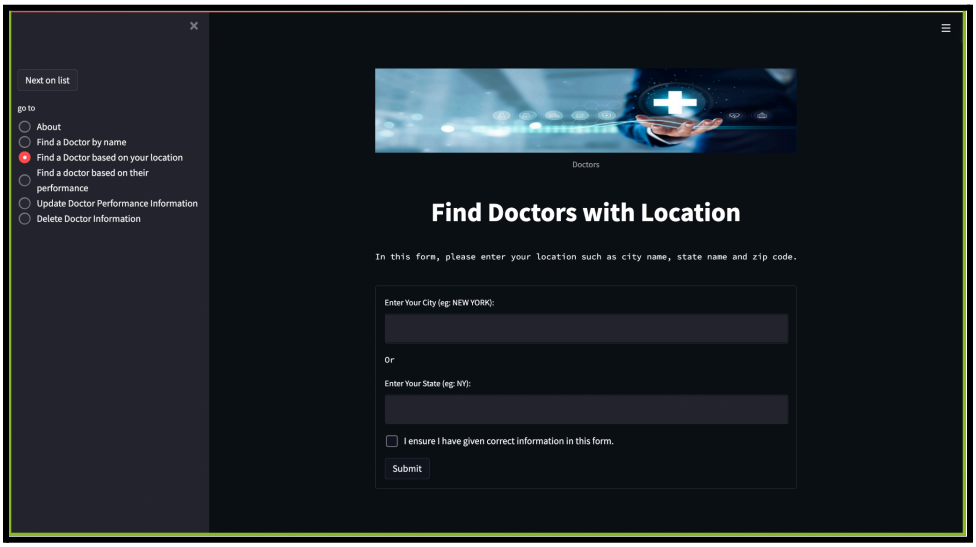

Figure 3: Find a doctor based on the location before input from the user

III (b).Find a doctor based on performance

In this page, a user is has a slider to choose the performance they require of a doctor based on their need. Once they have decided the value that they want, they can click the submit button. Once submitted we print the slider value to enure that the slider picked the correct value. The connection the the database *mydb* is made using the method in III(a). Now we use function *run_query* to run the query to use database Project2. Now we check if the input is empty before moving on to the execution of our stored procedure.

Now we use the function *cur.callproc()* to call the procedure *doc_perf*. The stored procedure joins *megatable_p2* and individual_performance_score by referencing PAC_ID. Once the procedure is called, it prints the doctors details with the same performance as mentioned by the user.

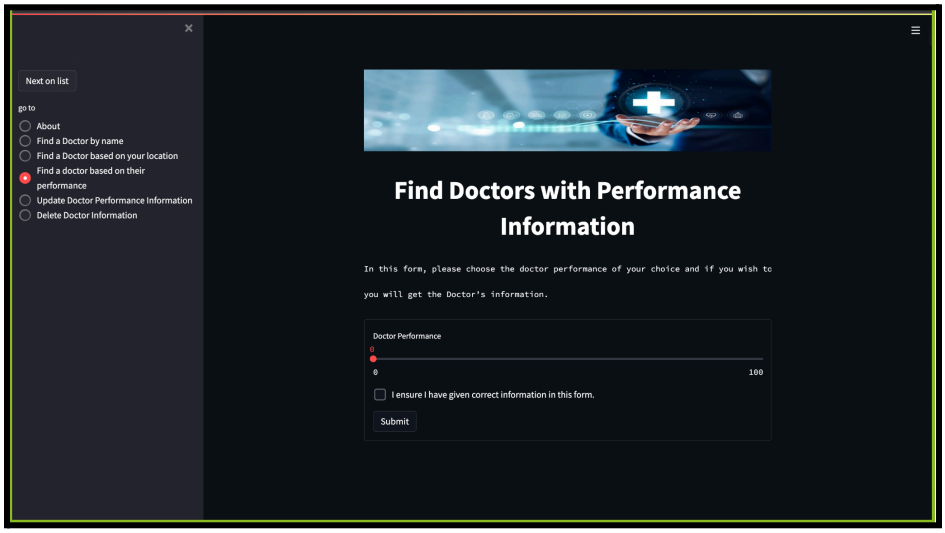

Figure 4: Find a doctor based on the performance page before input from the user

III (c).Update a doctor based on performance

In this page, the patient can update a doctors performance. They need to enter the PAC_ID of the doctor and choose the performance value on the slider. Once that is done, and the person clicks the submit button, the connection the the database *mydb* is made using the method in III(a). Now we use function *run_query* to run the query to use database Project2. Now we check if the input is empty before moving on to the execution of our stored procedure.

Now we use the function *cur.callproc()* to call the procedure *doc_update.* Then we select the row form the individual performance data where the PAC_ID is the same as the person whose performance was edited. This will help us ensure that the table was updated. The person can also see what their value was updated to.

There is also a trigger on update called *valuecheck* to check whether the performance score entered by the user is the same as the performance score in the database. This check is before the value is updated and will prompt the user to not enter the identical value.

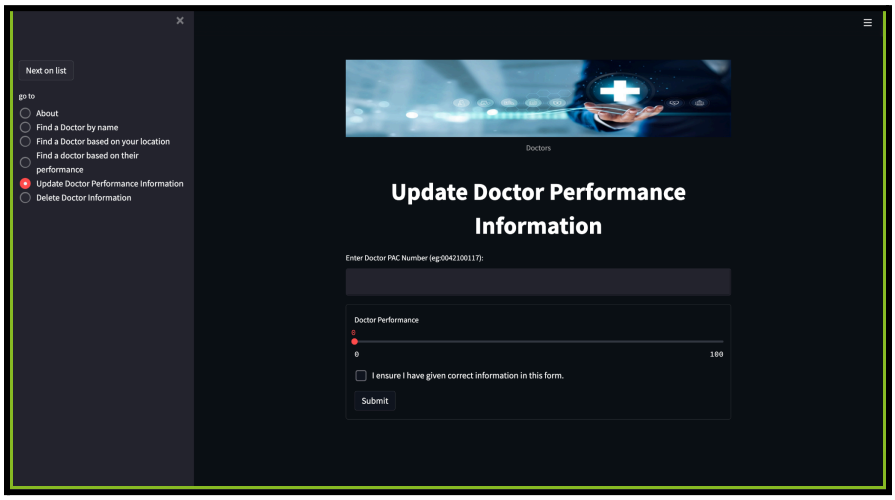

Figure 5: Update doctor performance

III (c).Delete a doctor based on PAC ID

In this page, the a person can delete doctor information.They need to enter the PAC_ID of the doctor and the row gets deleted. Once that PAC ID is entered, and the person clicks the submit button, the connection the database *mydb* is made using the method in III(a). Now we use function *run_query* to run the query to use database Project2. Now we check if the input is empty before moving on to the execution of our stored procedure.

Now we use the function *cur.callproc()* to call the procedure *doc_del* that deletes the row where PAC ID is the same as what was entered by the user. Furthermore, we created a new table that stores these deleted values so it can be recovered any time. The table is called performanceArchives which has PAC ID, First Name, Middle Name, Last Name and date and time when it was deleted. There is a view as well that is created that helps us call the trigger. We use the *run_query* function to run the trigger. Once that is done, there is a button to confirm what was deleted and we print the values deleted for the user to verify.

There is also a trigger on delete called *before_perf_delete* to store the deleted information about doctor in an archive table before the doctor is deleted from the database. Along with the trigger is a view showing the content of the archive table to help users to know which doctor was deleted.

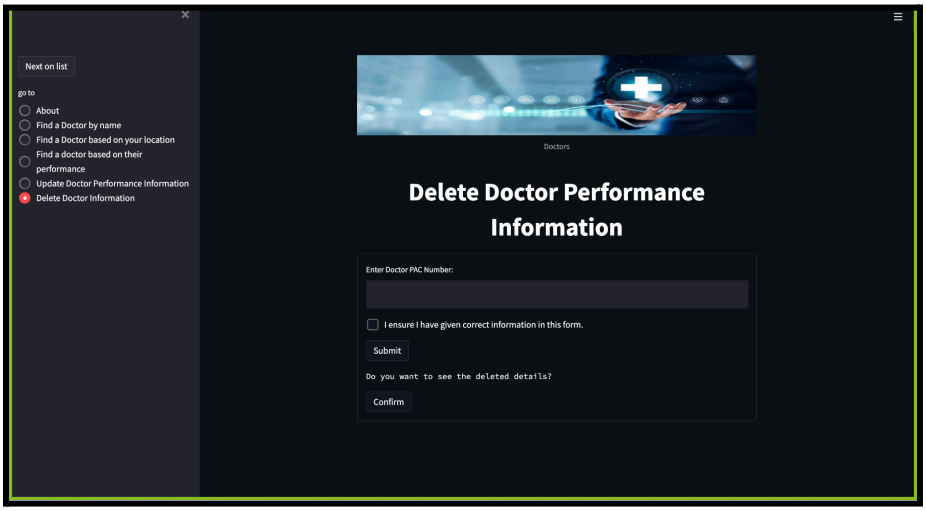

Figure 6: Delete doctor information

IV. Future Work

Our project can be extended into finding more features from the data, like giving the user a dropdown and they can choose all fields and based on that provide a doctor. Moreover an additional feature that takes the data from *performance_archives* and adds it to the megatable_p2 when the user deleted a row by mistake. Moreover we can make the website more aesthetically pleasing. These are some future works that we can think of.

V. Challenges:

We have encountered challenges on both the front-end and back-end part.. The first challenge is that we both have no experience with building websites with streamlit. After creating the structure of the website and connecting to the databases, we can only run views but not stored procedures. We overcame this with the help of teaching assistants. It turns out the syntax for stored procedure query is different from calling views. The difficulty we faced in the database is loading the data file properly. The values did not align with the headers. We learned a lot about the syntax of loading data from a file. We overcome this issue by specifying how the data should be terminated and enclosed.

VI. Work breakdown:

We participated in all the parts because we both wanted to gain experience in both front-end and back-end design, hence the work was very fairly divided in terms of the database and website development. In terms of division of labor for the application code, Anubha designed the navigation between web pages and discovered a cool widget, a slider. When creating the web pages, Zhuoyi and Anubha each took on three pages. And when perfecting the website, we worked together and made changes. As for the database code, Zhuoyi and Anubha all loaded the data and wrote code for testing dependency. Zhuoyi proposed the trigger to store deleted data in an archive table and the trigger to check for the same value. In addition, Zhuoyi and Anubha divided the work to create views and stored procedures. Both of us agree that our workloads are fair and equally split due to the fact we spent a lot of time communicating our thoughts and ideas.