COS30049 – Computing Technology Innovation Project

Assignment 3: Full-Stack Web Development for AI Application

Report on Air quality and Health

Workshop: 1-10 (Wednesday, 16:30)
Tutor: Hao Zhang

Group 29 – Safety Windy

Phong Tran - 104334842
Mai An Nguyen - 103824070
Khanh Toan Nguyen - 104180605

Table of Contents

1 System Architecture 3

1.1 Front-end 3

1.2 Back-end 3

1.3 AI Model 3

2 Front-end Implementation 4

2.1 Key Features 4

2.2 Explanation of Front-end and Back-end Connection 10

3 Back-end Implementation 12

3.1 FastAPI Server Setup 12

3.2 API Endpoint Documentation 14

3.2.1 @app.get("/") 14

3.2.2 @router.get("/pollutants") 14

3.2.3 @router.post("/predict") 15

4 AI Model Integration 17

4.1 Overview 17

4.2 Integration 17

5 Conclusion 19

6 Bibliography 20

1 System Architecture

Overall, the system architecture consists of the front-end, back-end, and the AI model. The front-end would submit input for model prediction and retrieve data to visualize. The back-end would receive the request from the front-end and process the request by sending data to the AI model and retrieving response prediction from the AI model, then returning the response to the front-end. The AI model would input the requested value and return the predicted burden mean to the back-end. This diagram shows how these components communicate with each other.

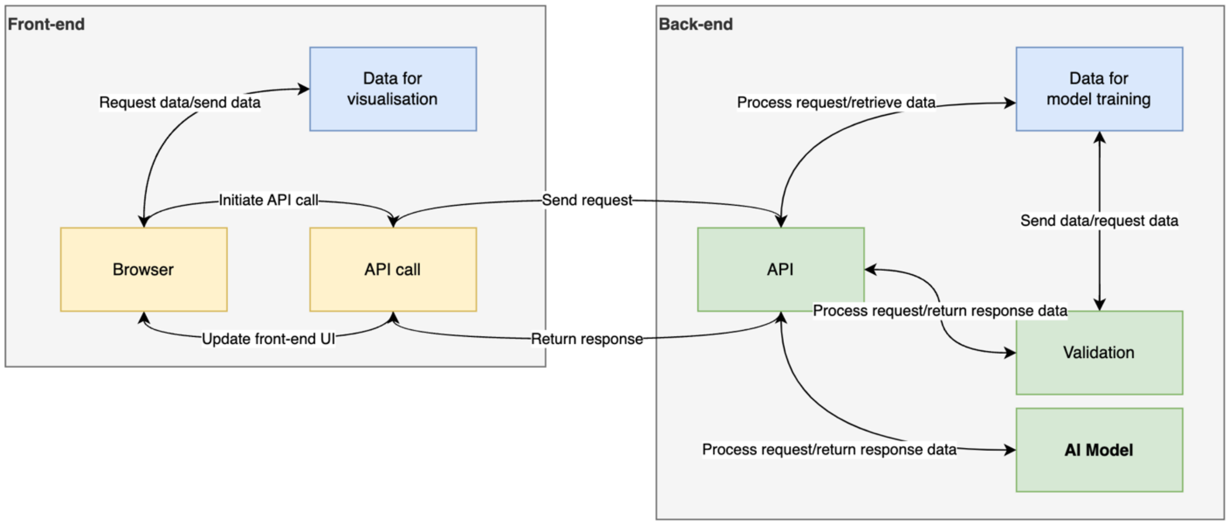


Figure 1. System architecture

1.1 Front-end

The website's front-end consists of interactive visualisations for users to discover the data regarding air quality and health, as well as a user input form with validation for input data prediction. This user form would be used to submit data for model prediction. The website would retrieve the training and the predicted data for data visualisations and real-time data, as well as display prediction output.

Further information regarding the technicality of the front-end will be described at 2 Front-end Implementation.

1.2 Back-end

The back-end is built following the FastAPI framework for building APIs with Python on web applications. Within the FastAPI server, our team has defined routes using 2 HTTP methods, GET and POST, to retrieve data input, validate, and return data prediction from the AI model.

The technicality of the back-end would be discussed further at 3 Back-end Implementation.

1.3 AI Model

The AI model is stored in a class within a Python file. The class would contain functions for data preprocessing, model training, and model prediction. As POST is triggered, AI model functions are initiated with the requested input and return the result.

More information on the technicality of the design will be discussed at 4 AI Model Integration.

2 Front-end Implementation

For Front-end implementation, React has been assessed for the full-stack application.

React is one of the best JS UI libraries for creating and maintaining views. Users can utilize reusable components in React. Moreover, React has a large and active community with extensive ecosystem. There is a significant difference in the number of people using React compared to other frameworks like Angular and Vue, where React takes the majority.

2.1 Key Features

Our website has a Visualisation page, containing the visualisations and the user input form.

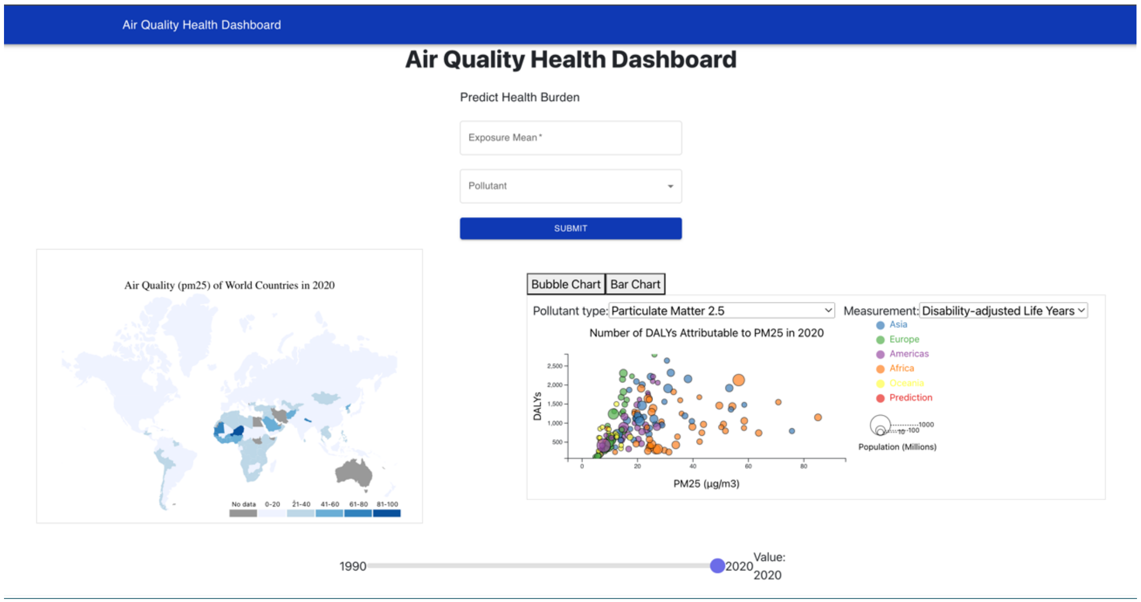

Air Quality Health Dashboard
Air Quality Health Dashboard
Predict Health Burden
Exposure Mean *
Pollutant
SUBMIT
Air Quality (pm25) of World Countries in 2020
No data
0-20
21-40
41-60
61-80
81-100
Bubble Chart
Bar Chart
Pollutant type: Particulate Matter 2.5
Measurement: Disability-adjusted Life Years
Number of DALYs Attributable to PM25 in 2020
DALYs
PM25 (µg/m3)
Asia
Europe
Americas
Africa
Oceania
Prediction
Population (Millions)
1990
2020
Value: 2020

Figure 2. The Website's Front-end

Regarding the main features, when entering the visualisation page, visualisations consist of 3 main visualisations, which are Choropleth, Bar chart and Bubble chart. All the charts have focus on delivering comprehensive information with rich insight and usage within the topic of air quality and health. The visualisation charts inherited the **Tufte design guidelines** (Tufte 2001) to produce clarity, Integrity, and efficiency in presenting data and **Munzner's visualisation design guideline** (Munzner 2014) for selecting the right chart corresponding to our dataset and purpose. Furthermore, we enhance user experiences by **positioning charts horizontally**, in which user can be able to gain more efficiency in retrieving information. Additionally, **color choices** for the visualizations are carefully considered and selected to ensure accessibility for a wide range of users, including those with color blindness. All of the visualizations employ **IBM's color palette**, which is specifically designed to maximize accessibility and inclusivity (IBM 2024). The palette provides a range of contrasting colors that remain easily distinguishable, even for viewers with color vision deficiencies. This approach makes the charts both distinguishable, aesthetic and visually appealing.

Choropleth illustrates pollutant particular matter 2.5 micrometers exposure mean for each country, meanwhile disability-adjusted life year (DALYs) by of each illness due to each of air pollution type by country through a **bar chart** and the data of exposure value, life lost, and population of each country via a **bubble chart**. Additionally, the **user input form** is integrated into the page for prediction. Prediction results will be displayed as text on the screen and as plot points on the bubble chart, implementing **real-time updates** on data visualisation with an interactive slider and tooltips.

The visualisations are programmed using the d3.js library, created by Bostock (2011). We have chosen this due to its high customizability and flexibility, ability to bind to DOM elements, and high compatibility with other libraries. It also provides extensive resources for data processing, scaling, enabling responsive interactions and event handling. Compared to chart.js, which is another JS library for visualisation (Chart.js | Open source HTML5 Charts for your website 2024), d3.js gives programmers more control as visualisations in chart.js are predefined. Moreover, d3.js could handle larger datasets and contain more built-in options for interactivity as well.

These are the different types of visualisations implemented on the website:

- **Choropleth:**
    - This is a map that is shaded according to a range of values presented in the legend. The saturation with the value of color in each country represents the range of the value of exposure to pollutant (PM2.5) in each country is in. This would help users compare the exposure to a type of air contaminant, particularly PM2.5, among countries. The countries could be clicked for country selection to be displayed for further details on the Bar Chart.
    - The reason for selecting only the PM2.5 metric is that it encompasses other pollutants such as NO_2, ozone, and hazardous air pollutants (HAPs), incorporating whichever pollutant levels are lower.
    - This choice allows the choropleth map to present a more comprehensive view, offering an overview of air pollution levels. It also optimizes the strength of the Bubble Chart, which can then focus on providing more detailed information about individual pollutants, allowing viewers to analyze specific elements in depth if they wish to explore further.

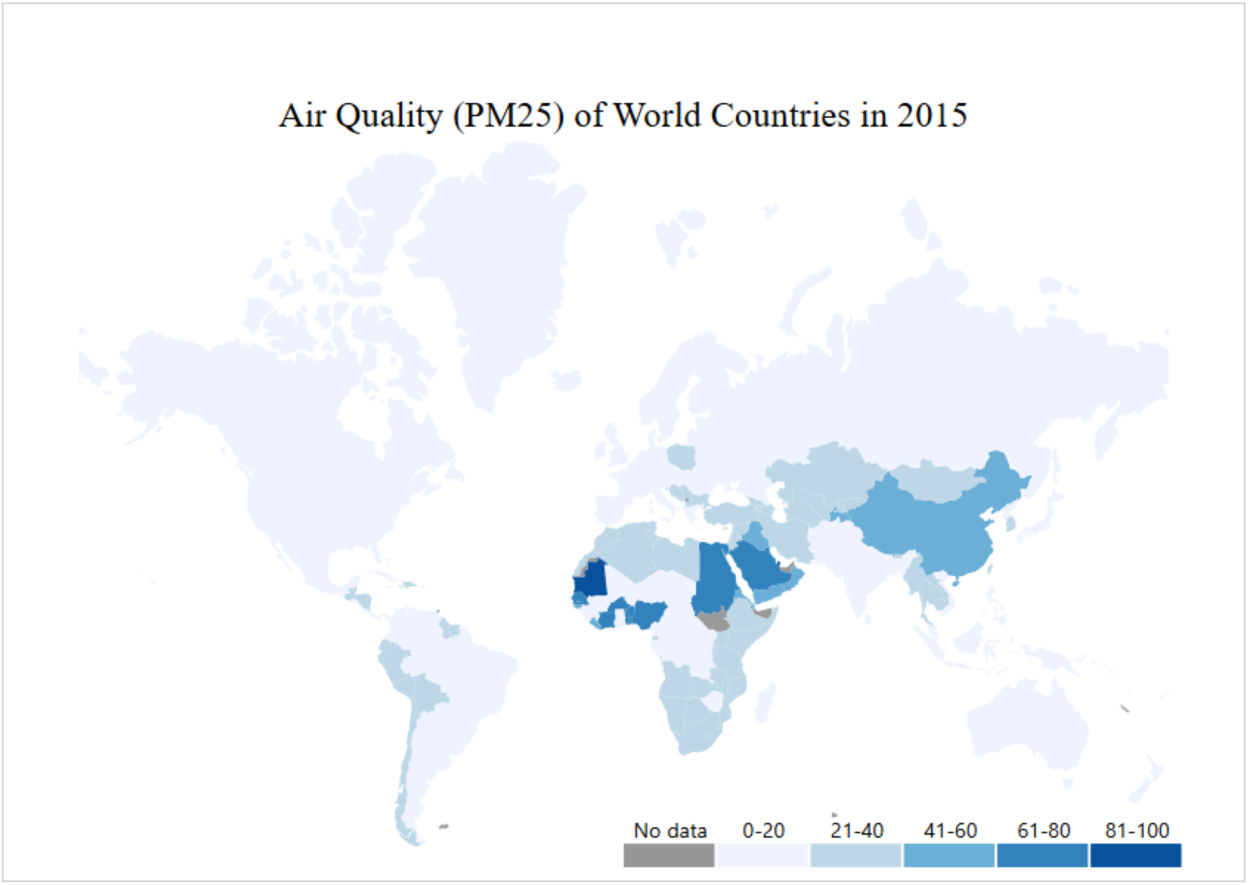


Figure 3. Choropleth

- Bubble Chart:
    - This type of visualisation uses circles plot to show relationships of numeric variables, between DALYs burden of the country corresponding to each pollutant type through an axis option.
    - In this situation, we have used exposure of the chosen type of air contaminant and life loss (DALYs) due to that contaminant as x and y positions respectively. The horizontal and vertical positions indicate the values for an individual circle, as well as the circle's radius which is determined by the population of each country.
    - Together with the circle's colouring, indicating the continent the country belongs to, users could observe relationships between exposure to pollutants, life loss, population size, and geographical location of the country's data by year.
    - Values of user input and prediction of the form are also updated on this chart.

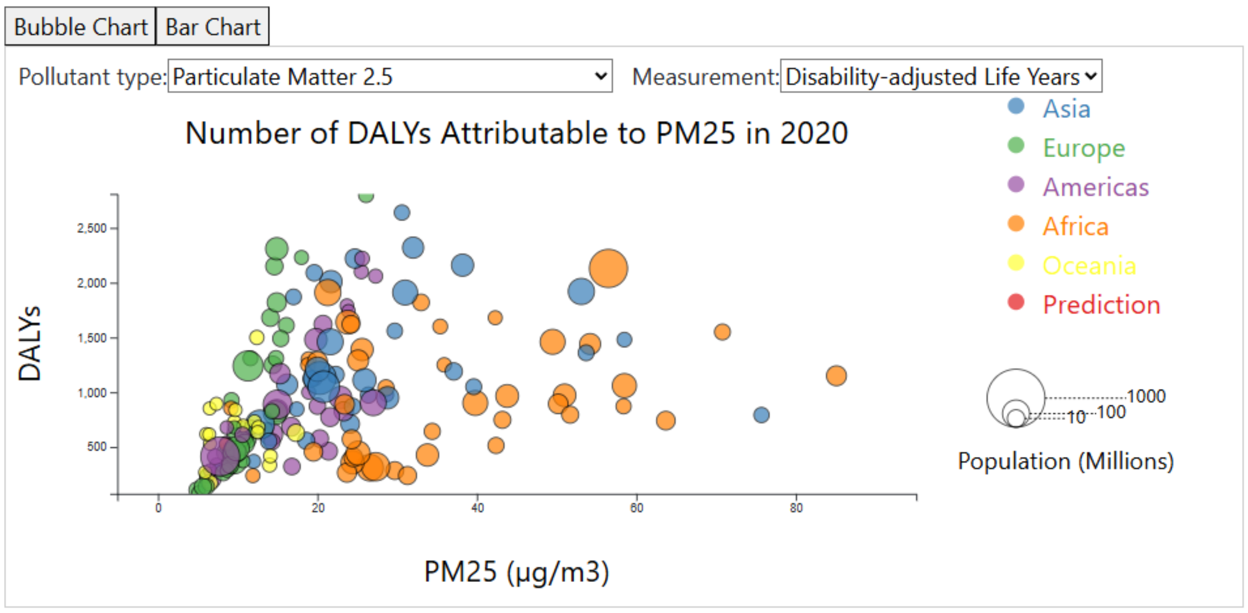


Figure 4. Bubble Chart

- Bar Chart:
    - As known as the column chart, this plots numeric values for levels of a categorical feature as bars.
    - The x-axis represents the illness due to the chosen air pollutant, while the y-axis determines the rate of life loss, playing as the scale for each bar's height. The bar of each illness is indicated by the values at the x-axis as well as the colour, noted in the legend. This provides users with further information regarding life loss by selected pollutants and countries of air pollution-related illnesses, which is in our dataset CSV file, making the user understand the health impact of the pollutant in that country.
    - We have also implemented an advanced feature, which allow the chart to only display the chosen pollutant by the country selected by the geographical location of the country on choropleth
    - Users can also choose the contaminant type on the dropdown list for a broader view of the consequences of contaminants burdening on health.

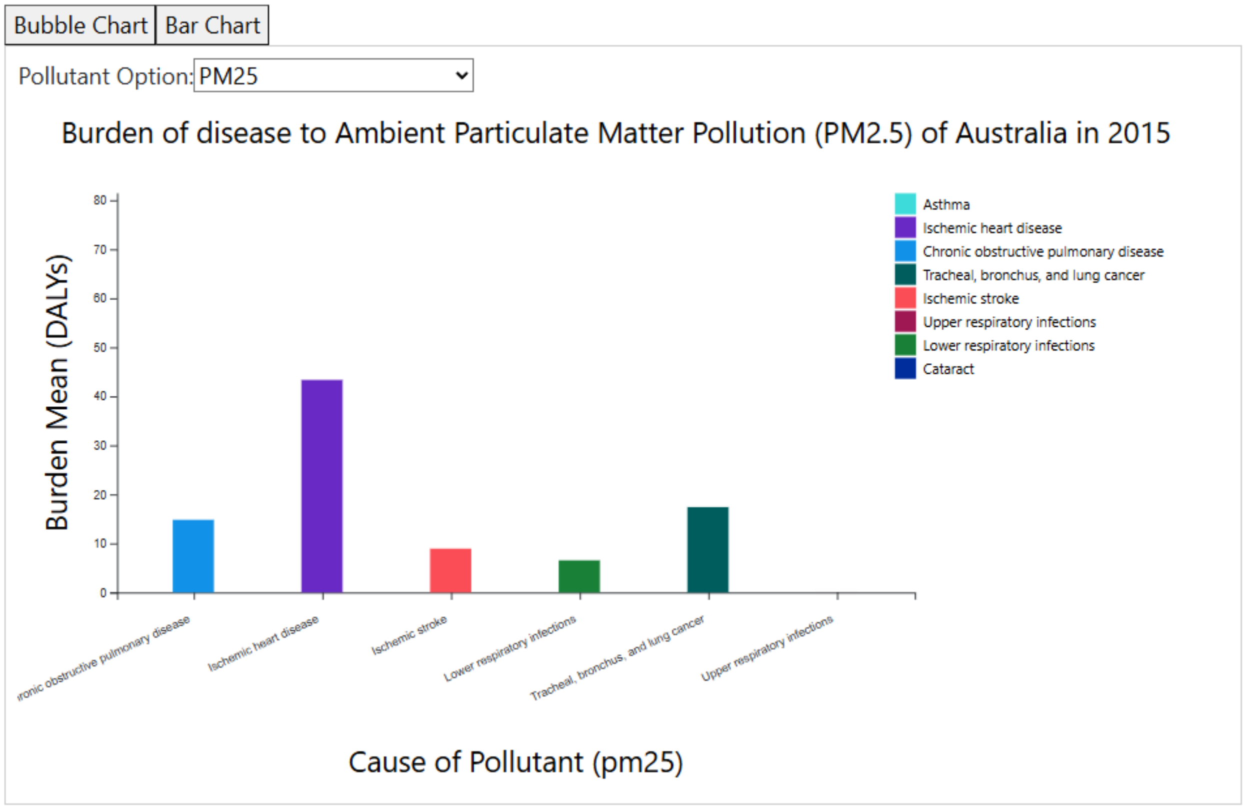


Figure 5. Bar Chart

As a visualisation component is hovered, its data will be displayed within a tooltip div. For example, when hovering over a country in the choropleth or a data circle in the bubble chart, the detailed exposure value, the type of pollutant, the name of the country, and more information are shown.

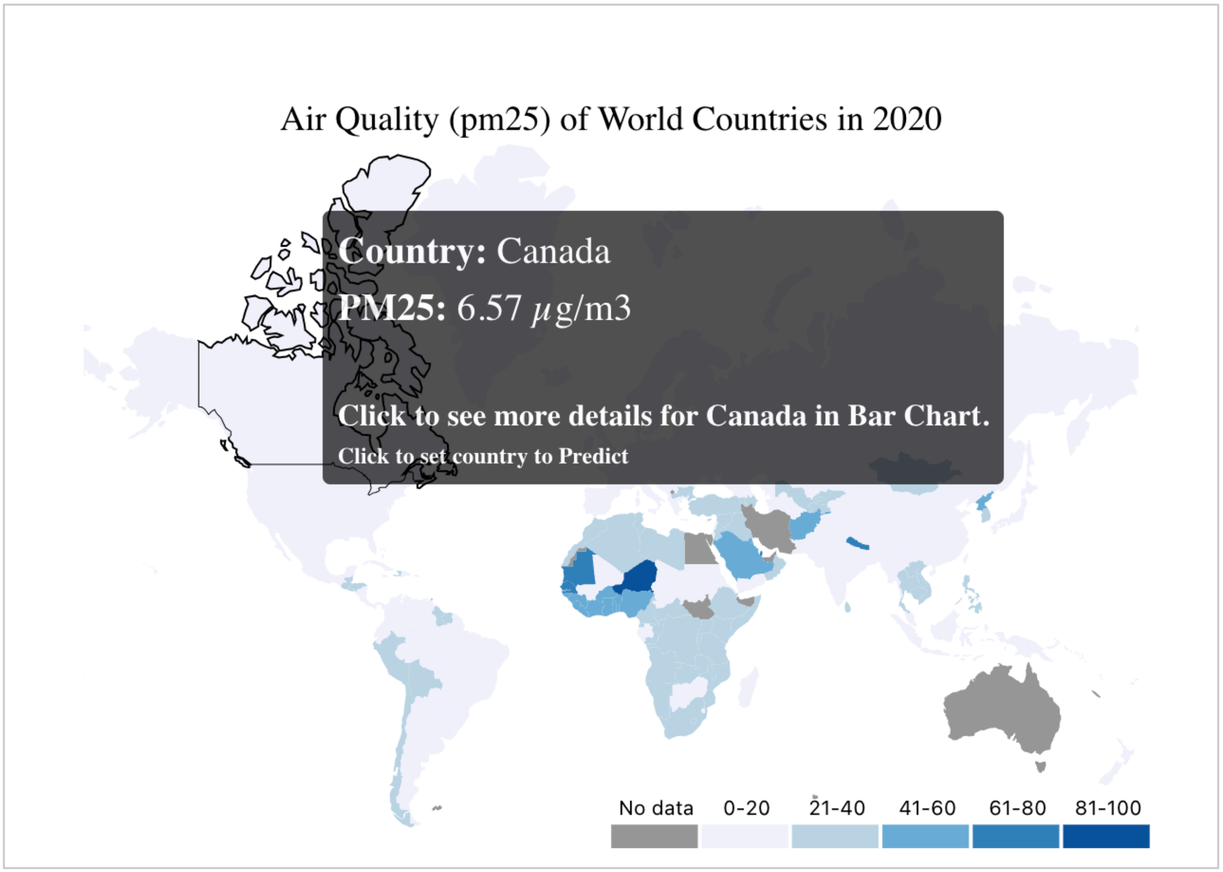

Air Quality (pm25) of World Countries in 2020
Country: Canada
PM25: 6.57 μg/m3
Click to see more details for Canada in Bar Chart.
Click to set country to Predict
No data
0-20
21-40
41-60
61-80
81-100

Figure 6. Canada is hovered on the Choropleth

There is also a slider to determine the year of the data. This helps to determine the input data for the Choropleth and the Bar Chart, as they visualise the data not only by pollutant and values but also by year as well. As the user slid to choose the year for display, the data visualised on these graphs changes with the transition.

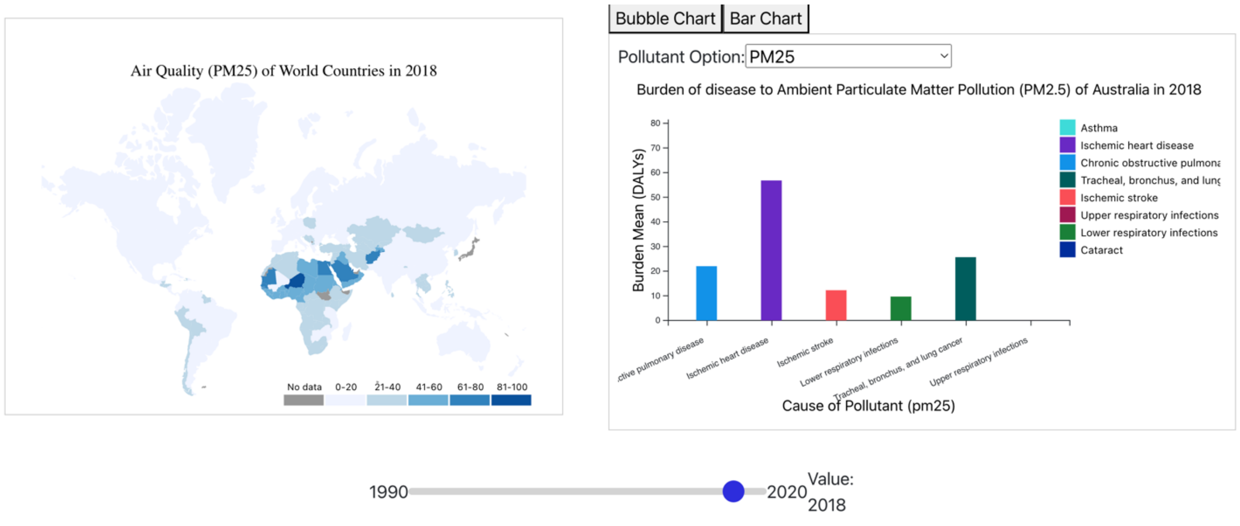
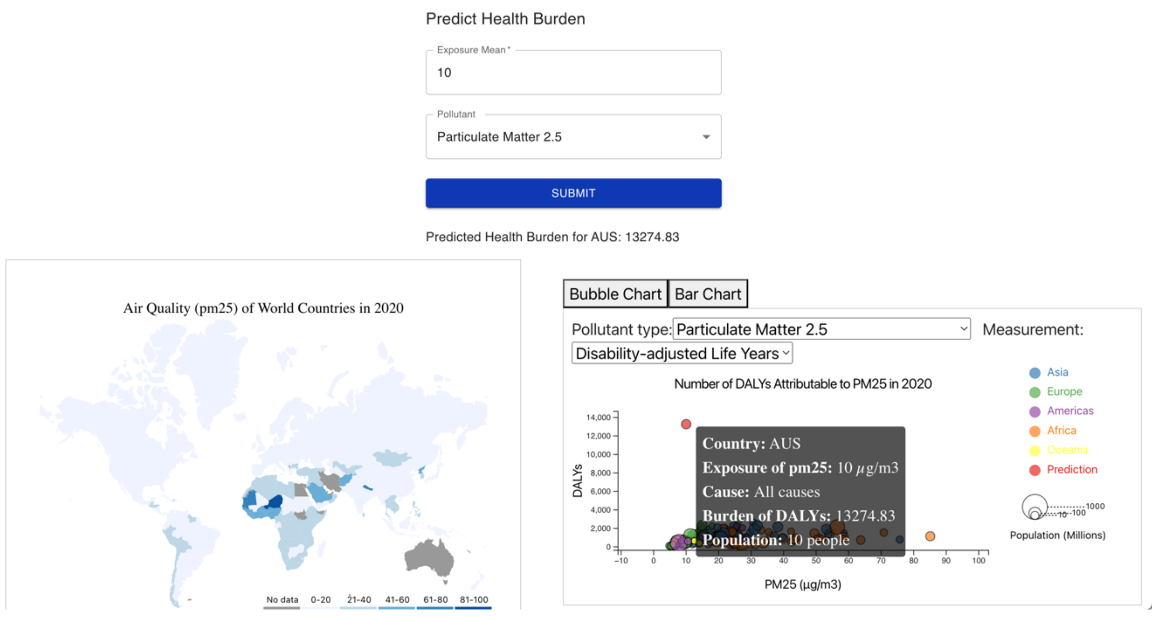

Air Quality (PM25) of World Countries in 2018
No data
0-20
21-40
41-60
61-80
81-100
Bubble Chart
Bar Chart
Pollutant Option: PM25
Burden of disease to Ambient Particulate Matter Pollution (PM2.5) of Australia in 2018
Burden Mean (DALYs)
80
70
60
50
40
30
20
10
0
Asthma
Ischemic heart disease
Chronic obstructive pulmona
Tracheal, bronchus, and lung
Ischemic stroke
Upper respiratory infections
Lower respiratory infections
Cataract
ctive pulmonary disease
Ischemic heart disease
Ischemic stroke
Lower respiratory infections
Tracheal, bronchus, and lung cancer
Upper respiratory infections
Cause of Pollutant (pm25)
1990
2020
Value:
2018

Figure 7. Data visualised changes as the Year changes

Regarding the user input form, users would submit their selection of country and pollutant for display, with the input data for prediction. The country and the pollutant input would undergo validation to look for matching values to see if they are valid inputs, while the input exposure value would go through a different check type to see if it is a numerical value. If any error occurs, it will be displayed on the screen. These validations are done in the front-end as well as the back-end with the POST endpoint before model prediction. This would be described further in 3 Back-end Implementation and 4 AI Model Integration. After the front-end retrieves the prediction from the back-end, the result would be displayed in text and appended to the bubble chart visualisation as mentioned, as the input data with the corresponding prediction is added to the dataset. To see the prediction point, which is colored in red, reclick the Bubble Chart option button or click the Bar Chart option, the reopen the Bubble Chart.


Predict Health Burden

Exposure Mean *: 10

Pollutant: Particulate Matter 2.5

Predicted Health Burden for AUS: 13274.83

Pollutant type: Particulate Matter 2.5

Measurement: Disability-adjusted Life Years

Country: AUS
Exposure of pm25: 10 μg/m3
Cause: All causes
Burden of DALYs: 13274.83
Population: 10 people

Figure 8. User Input Form and Real-time Update

To increase interactivity, our team has utilised hover and transition effects, as well as components such as a slider, dropdown list, and user input form to provide as much information as possible and let users opt for data based on their preferences. We have also considered other aspects of our web design and visualisation display to improve user experiences. For example, instead of showing all visualisations and user input at once and letting the user scroll through the page, we have placed a choropleth on the left side and other charts with user input form on the right side of the screen. The reason for our decision is because according to the research (Rottach et al. 1996) human eyes perform better horizontally when follows horizontal predictable movement than vertical, diagonal, and other directions, this allows smoother comparison between charts, thus retrieving information more efficient.

2.2 Explanation of Front-end and Back-end Connection

Our team has set up the React component to render a form with input fields to gather user data and display feedback, including validation errors and prediction results. The component is returned at useEffect() within constant FormInputPrediction.

Within the component, divisions, a form with labels and inputs, and a button are utilised for creating containers, form handling, and submitting the form. After submitting the input via the button "Predict," If errors are not null or the length is more than 0, the corresponding error will be displayed in a div. If not, a div will show the prediction output. For values to be shown at a suitable position in the component as well as to enable data submission and prediction retrieval, our team has stated variables such as iso3, exposureMean, and pollutant to store user inputs. Besides, our team also defined prediction to hold the prediction result from the server, errors to store general error messages, and formErrors to keep track of only input validation errors.

To describe further, when the user submits the form, handleSubmit() is triggered. There, the inputs would undergo validateInputs first, which checks the format of the input. This is achieved by using if else cases to append errors into an error object, where the error of each input field corresponds with the attribute of the error object. If there is an error in any fields, the errors would display on the screen and the handleSubmit end.

Otherwise, if there are no errors, setErrors is reset for later error handling and setPrediction is initiated. Then, the client would send a POST request with input data, reshaped into suitable JSON format to the back end by fetching the endpoint /predict ('http://localhost:8002/predict').

```
const response = await fetch('http://localhost:8002/predict', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ iso3, exposure_mean: exposureMean, pollutant }),
      });
```

To explain, fetch() is a function in JavaScript that initiates an HTTP request. The HTTP method has been set up here as the POST method, which will send the input data to the specified endpoint. The headers attribute indicates metadata about the request, which in this situation, notifies the back-end that the data being sent is in JSON format. The body defines the actual data being sent to the back-end, which has been converted into a JSON string containing country code (iso3), exposure_mean (exposure to pollutant), and pollutant (the chosen pollutant) for prediction. By using await, the code would wait for the response of the request before proceeding.

When the API sends the response, this is stored in the const response. Another constant named 'data' is used to convert the response from an HTTP request into a JavaScript object. If the response is fine, the prediction will be set to the returned prediction. Else, the error would be set. Thus, as an error is caught, the error would be printed out on the console for debugging and on the screen for displaying.

Here is a diagram to show their interactions.

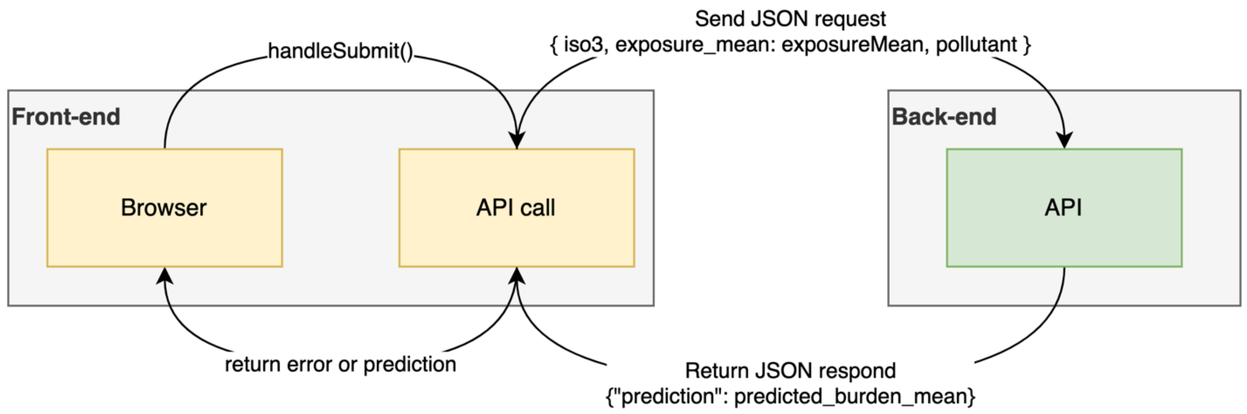

Send JSON request
{ iso3, exposure_mean: exposureMean, pollutant }
handleSubmit()
Front-end
Browser
API call
Back-end
API
return error or prediction
Return JSON respond
{"prediction": predicted_burden_mean}

Figure 9. Interactions between the Front-end and the Back-end API for sending and retrieving data

3 Back-end Implementation

For back-end implementation, FastAPI has been chosen for API set-up.

FastAPI is a web framework for building API with Python (Ramírez n.d.). It supports asynchronous programming using async, allowing it to handle many requests at once, which is why FastAPI is one of the fastest Python frameworks available for building APIs. Compared to other frameworks, particularly Flask (Pallets n.d.), the Flask framework presents a steeper learning curve, particularly for individuals who may be new to Python or have limited time and expertise to explore its features comprehensively. The documentation available for FastAPI is also more extensive than that of Flask, with more resources offering in-depth coverage of its full functionality. In terms of performance, FastAPI is generally faster than Flask as well, especially for projects requiring high performance and scalability.

3.1 FastAPI Server Setup

Our team has provided instructions on FastAPI installation and other app configurations within the README.md.

After FastAPI installation, python files are set up according to the separation of concerns. Our team has used app.py, regression.py, regression_model.py, and request.py files.

Here is how the backend directory is set up:

```
Root/backend/
├── app/
│   ├── app.py
│   ├── api/
│   │   ├── endpoints/
│   │   │   └── regression.py
│   │   ├── models/
│   │   │   └── regression_model.py
│   │   ├── schemas/
│   │   │   └── request.py
├── data/
│   └── air_quality_health_2.csv
├── requirements.txt
└── README.md
```

Here are their functionalities:

- app.py: initiate and set up the FastAPI, include all middleware functions and exception handlers, as well as import the routers
- regression.py: initiate router and set up all endpoints of GET and POST
- regression_model.py: contains all functions for input validation, model training and predicting, which are called by async function at POST.
- request.py: contain classes define format of ValidationInput and PredictionOutput for function input and JSON response respectively

Our team have used 2 request types, which are GET and POST, to enable retrieving input data and returning the prediction to the front-end, as well as retrieving datasets, processing requests to and receiving returns from AI model training and predicting. For other methods, our team does not need to update existing data and delete a specified resource, therefore, PUT and DELETE are not implemented.

To use FastAPI routers, FastAPI() needs to be called. Then, methods of GET and POST are utilised. Under each method, an asynchronous function is called corresponding to each endpoint to run the functionalities, such as returning the state of the website, initiating the regression model, and so on.

We have implemented FastAPI by creating a variable 'app' to store the FastAPI object. Then, we called the methods via the object. To briefly describe, our app.py currently contains these middleware functions and exception handlers, as well as an endpoint to verify website connection:

- @app.middleware("http"): measure and log the response time of specific API endpoints, helps with performance monitoring and debugging
- app.add_middleware: adds Cross-Origin Resource Sharing to restrict web page making requests to different domains, define which origins are permitted to access the web resources. Our team has allowed requests from any domain to access the API, as well as all HTTP methods and headers in requests. The credentials are also included in requests to allow authorization, which we may develop in the future.
- @app.get("/"): return welcome message to verify users are in the website.
- @app.exception_handler(HTTPException): catches HTTPException, which is usually raised when there is an issue with an HTTP request.
- @app.exception_handler(RequestValidationError): catches RequestValidationError, which is raised when request data does not meet the expected validation criteria defined.
- @app.exception_handler(Exception): catches any other errors within the web application.

Within the app.py, the routers are called, which we stored in regression.py. In regression.py, we have created a router object to store the endpoints. They are:

- @router.get("/pollutants"): retrieve pollutants for user input form
- @router.post("/predict", response_model=PredictionOutput): post input data for prediction (PredictionOutput is a class included in request.py)

Within the regression_model.py, our team set up these functions, which are called for different purposes in API endpoints for validation and prediction:

- def read_countries_from_csv: reads countries in dataset (which is used for prediction). This is called by def validate_data to verify if input country matches with a specific country in the dataset for further processing.
- def validate_data: validates input data in the format defined by ValidationInput class by verifying the connection to the dataset, whether if all inputs are written, if input country matches with a country in the dataset, if that country has data for specified pollutant using def read_countries_from_csv. This is called by @router.post("/predict", response_model=PredictionOutput)
- def predict_burden: runs the AI model. Within this function, data preprocessing, model training, model evaluation, and model prediction would be done. This is called by @router.post("/predict", response_model=PredictionOutput) after def validate_data has been run.
- def load_and_filter_data: is called by def predict_burden to filter data for data preprocessing.
- def remove_outliers: is called by def predict_burden to remove outliers for data preprocessing.
- def scale_features_targets: is called by def predict_burden to normalize data for data preprocessing.
- def split_data: is called by def predict_burden to split data based on train test approach for model training and evaluation.

- def train_linear_regression: is called by def predict_burden to train linear regression model based on processed training data for model training.
- def train_polynomial_regression: is called by def predict_burden to train polynomial based on processed training data for model training.
- def evaluate_model: is called by def predict_burden to calculate R square (R2) and Mean Square Error (MSE) from predicted training data and test data for model evaluation.
- def select_best_model: is called by def predict_burden to evaluate model based on the returned R2 and MSE of both models and choose the best model for prediction.
- def make_prediction: gets the input of user input form and produce prediction output. This involves normalize the input, predict using the chosen model, and denormalise the output. This is called by def predict_burden

Within request.py, we have defined these classes to handle different issues:

- ValidationInput(BaseModel): define data types and input structure of ISO3 code of countries, pollutant, and exposure value of selected pollutant for validation purpose.
- PredictionOutput(BaseModel): define prediction output response.

Documentation on the API endpoints in the regression.py would be provided and explained further in 3.2 API Endpoint Documentation.

3.2 API Endpoint Documentation

Our team has used 3 endpoints, which are @app.get("/"), @router.get("/pollutants"), and @router.post("/predict") as mentioned in 3.1 FastAPI Server Setup.

3.2.1 @app.get("/")

This GET request would be triggered as the website is opened. It returns a welcome message as the JSON response, which is shown below.

```
{
      "message": "Welcome to the Air Quality Health API"
}
```

3.2.2 @router.get("/pollutants")

This GET request retrieves the pollutants from the dataset CSV file to be displayed on the front-end for pollutant selection. If the file is not found or not accessible, an error message will be used for JSON response. Otherwise, the pollutants are returned.

If the dataset file is not found, the 404 Not Found code will be returned. The JSON response for this would be like what is shown below.

```
{
     "status_code": 404,
     "detail": "Data file not found."
}
```

If there is an error parsing or reading the dataset file, the 500 Internal Server Error code will be returned. The JSON response for this would be like what is shown below, where specific error messages would be taken from exception handler functions.

- Parsing error:

```
{
      "status_code": 500,
        "detail": "CSV Parsing Error: <specific error message>"
}
```

- Reading error:

```
{
    "status_code": 500,
    "detail": "Error reading data file: <specific error message>"
}
```

Otherwise, the JSON response as pollutants could be fetched is like what shown below.

```
{
        "pollutants": [{"key": key, "display": pollutant_mapping.get(key, key)} for key in
pollutants]
}
```

If no errors are found, the pollutants are collected into a 'pollutants' array, which contains unique values of pollutants. The 'key' indicates the value in the pollutant attribute, while pollutant_mapping.get(key,key) determines the short description or the full name of the pollutant. An example of what may be responded to is:

```
{
"pollutants": [
                {
                          "key": "no2",
                          "display": "Nitrogen Oxide (NO2)"
                },
                {
                          "key": "pm25",
                          "display": "Particulate Matter 2.5 (PM)"
                },
                {
                          "key": "hap",
                          "display": "Harzadous Air Pollutants (HAP)"
                },
                {
                          "key": "ozone",
                          "display": "Ozone"
                }
        ]
}
```

3.2.3 @router.post("/predict")

This POST request gets the input from the submitted form, which includes country code (ISO3), pollutant, and exposure value to the pollutant, to go through validations and model prediction.

As country, pollutant, and exposure value corresponding to the user's options for prediction are sent to POST, the asynchronous function of the endpoint would process validation for data preprocessing. To do so, the ValidationInput class would check the format of the input first. Then, the input goes through the validate_data function, which checks if the CSV file can be retrieved for model training, training data contains the selected country and pollutants. Within this function, if any error is encountered, the error would be raised and a JSON response regarding the error displays instead of further processing the code in the function validate_data, as well as stopping the POST method.

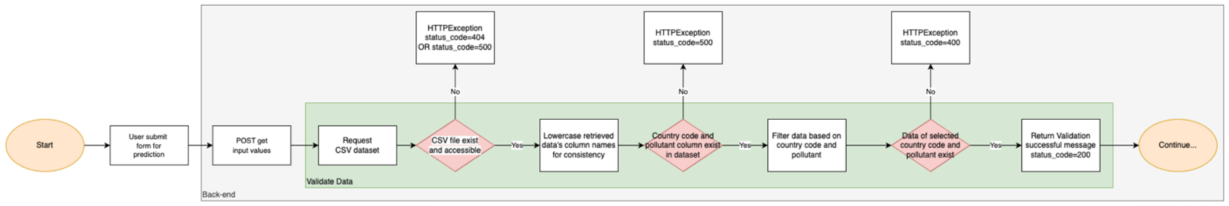

Start
User submit form for prediction
POST get input values
Request CSV dataset
CSV file exist and accessible
No
HTTPException status_code=404 OR status_code=500
Yes
Lowercase retrieved data's column names for consistency
Country code and pollutant column exist in dataset
No
HTTPException status_code=500
Yes
Filter data based on country code and pollutant
Data of selected country code and pollutant exist
No
HTTPException status_code=400
Yes
Return Validation successful message status_code=200
Continue...
Validate Data
Back-end

Figure 10. Flowchart of data and input validation

If the CSV dataset does not exist, status code 404 is set. Additionally, if there is an error parsing or reading the data file, status code 500 is set. The JSON response is similar to those in the GET request. More details of the JSON response could be found in 3.2.2 @router.get("/pollutants").

If the required attribute for prediction is missing in the dataset, error 500 will display. The JSON response would be:

```
{
        "status_code": 500,
        "detail": ["Missing required column in CSV: {col.upper()}."]
}
```

Where col is the column missing.

If data is not found for the chosen pollutant and country, error 400 will be displayed. The JSON response would be like what is shown below if no data is found for the specified country, where iso3 is the country code, and pollutant is the chosen pollutant in the input.

```
{
        "status_code": 400,
        "details": [
                          "No data found for the specified ISO3 country code: {iso3}.",
                          "No data found for the specified pollutant: {pollutant} in
country:{read_countries_from_csv(iso3, CSV_FILE_PATH)}."
]
}
```

Another situation may happen, is when data for the country is found, but no data for the specified pollutant in the country is retrieved. The JSON response would be:

```
{
        "status_code": 400,
        "details": ["No data found for the specified pollutant: {pollutant} in country:
{read_countries_from_csv(iso3, CSV_FILE_PATH)}."]
}

```

If all validation passes, the response would be like here.

```
{
        "Validation successful",
        "status_code": 200
}
```

Then, the asynchronous function predict(input_data) would continue. The prediction result would be stored in the result variable from the output of the function predict_burden(input_data), which processes the AI model prediction. This will be further explained in 4.2 Integration.

Finally, as the result is returned, the asynchronous function is complete. The JSON response of the POST request would be as below, where the result is the prediction output in float data type, defined by the PredictionOutput class in request.py.

```
{"prediction": result}
```

4 AI Model Integration

4.1 Overview

For assignment 3, our team have integrated the regression model from assignment 2. This model's objective is to predict the impact of air quality on health. To do so, our team has investigated data from each country and each year, involving the exposure value of each pollutant and burden mean, or rate of life lost (DALY) and used regression models to find the pattern.

For each country, linear regression and polynomial regression are considered for the prediction of DALYs based on the exposure mean. Linear regression suits situations where these variables have an approximately linear relationship, otherwise, polynomial regression would be more robust. Nonetheless, polynomial regression is more prone to overfitting. By using data from each country to run the models and evaluating both model performances according to scores like R2 or MSE, a linear or polynomial regression model would be chosen for the prediction.

To implement the AI models, the linear_model module from Scikit-learn library (scikit-learn developers 2024), PolynomialFeatures for data transform, MinMaxScaler, drop, merge, or filter functions from pandas library for preprocessing, which is contributed by the pandas development team (2024), are utilised. They would be imported into the full-stack software as a Python file.

4.2 Integration

To integrate the AI model into the full-stacked web application, the back-end APIs would connect with the model via the POST method, as the back-end component does input validation and uses the function predict_burden to post the request to the AI model. Within this endpoint, the back-end could also get the prediction results returned from the AI model as well.

To do so, firstly, the AI model file is transformed from a Jupiter notebook file to a Python function, named predict_burden in the regression_model.py, and is modified for the web application. This regression model would have code lines to preprocess data, train the model, and predict the output, which are built based on functional cells in the notebook. As models for each country and pollutant are different, the model needs to be rerun as new input is submitted. Therefore, we chose not to save the model as a PKL file.

When the user selects the country and the pollutant, as well as enter the exposure mean corresponding to their options for prediction, FastAPI's POST endpoint would retrieve the data input as described and validation of criteria for data preprocessing would be done. These are mentioned in the previous part 3.2.3 @router.post("/predict").

Then, as pollutant and country were selected and verified, the function predict_burden run at the POST endpoint of the back-end API, where the submitted input would undergo data preprocessing, model training of both linear regression and polynomial regression, model evaluation, and model prediction based on the evaluation, with help from Scikit-learn libraries.

To preprocess the data, the dataset would be filtered out based on the chosen country and pollutant for training using def load_and_filter_data. Then, the data filtered would be cleaned. To do so, quartile values of exposure to pollutants and burden life loss due to that pollutant in the training dataset are calculated to remove values greater than the upper bound and less than the lower bound of these two attributes using def remove_outliers. Following that, normalisation would be done by utilising MinMaxScaler and fit_transform function from the Scikit-learn library within def scale_features_targets.

Finally, the datasets would go through a linear regression model (def train_linear_regression) and a polynomial regression model (def train_polynomial_regression). We use the train test split method to evaluate the model to choose the best model for corresponding to the country and pollutant as we store the training and testing data by calling def split_data. The models would fit into the training data and complete prediction, then go through model evaluation based on the testing data by running def evaluate_model.

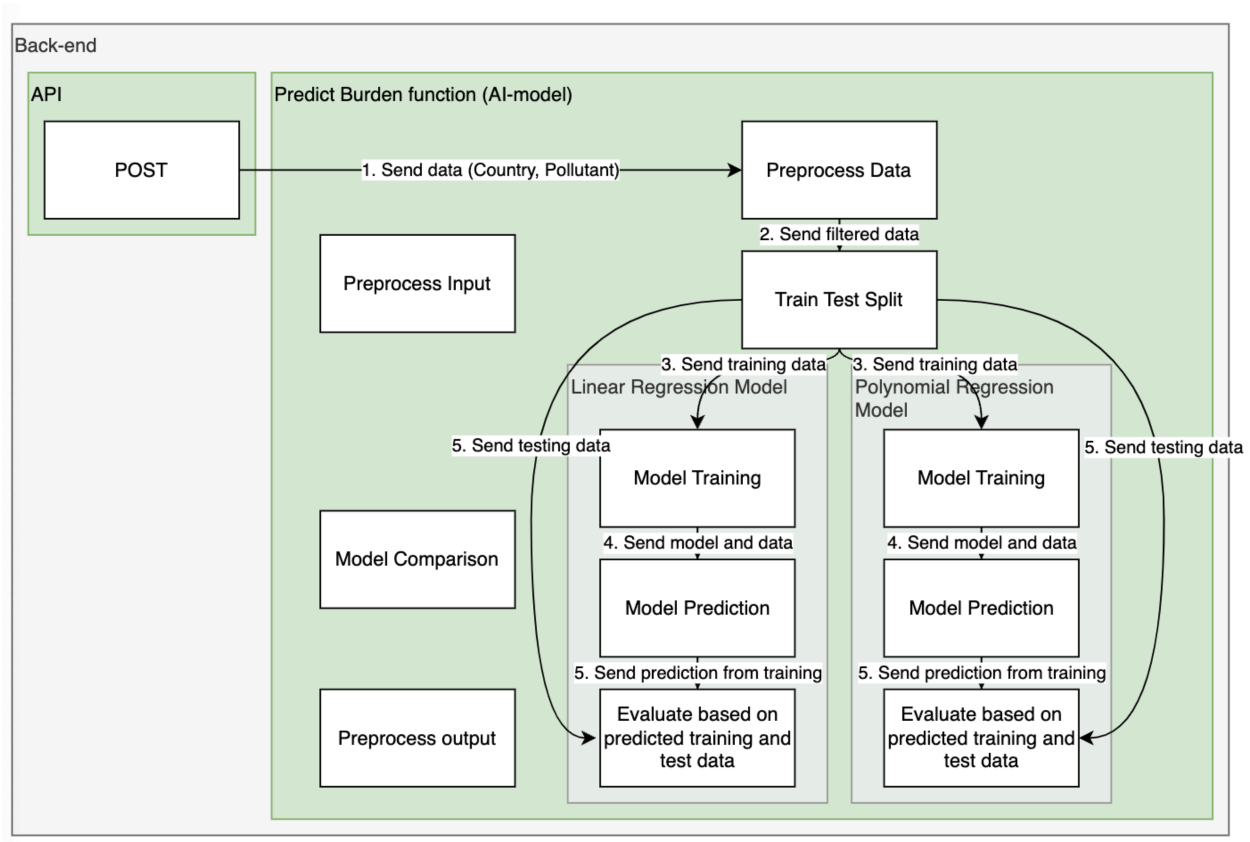


Figure 11. How AI model is trained

Then, the R2 and MSE scores of both models are produced with the help of def select_best_model, which would be used for comparison to choose the most suitable model for prediction.

The submitted input of exposure value enters the make_prediction function, where it would undergo normalisation before entering the chosen model via the MinMaxScale() used previously for normalising the dataset. This is to ensure that they are within the same range and represented using the same scale. The normalised value then would be predicted by both models. Then, the output would be denormalised to get the real accurate result. The value then is returned to the POST endpoint.

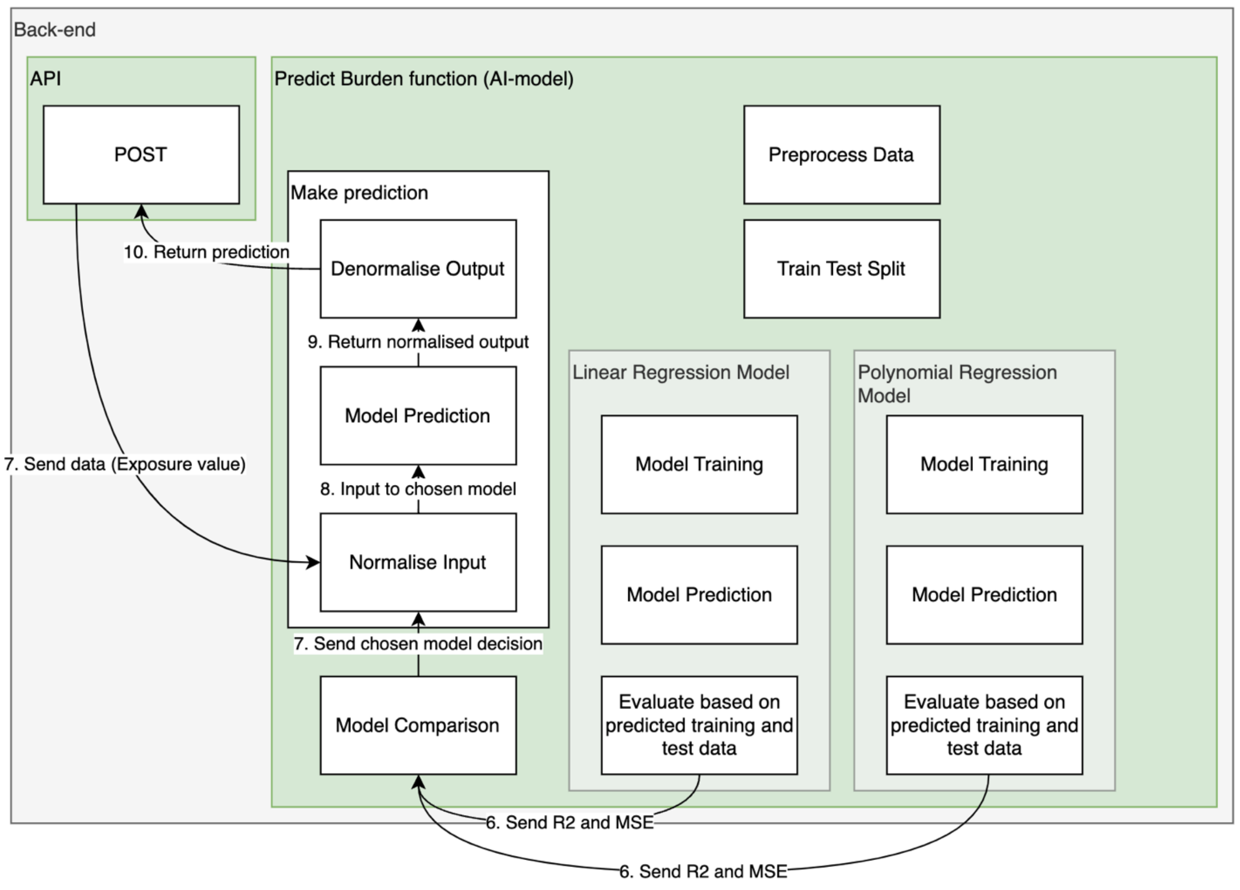


Figure 12. How data is predicted

5 Conclusion

Our project's objective is to provide detailed and interconnected statistics on air quality and health by looking into data on air pollutants such as nitrogen dioxide, ambient particulate matter, household pollutants of solid fuels, and ozone, as well as the life loss of illnesses they cause. This application has been able to deliver information regarding exposure to air pollution and life lost due to air pollution-related illnesses according to data from each country and types of pollutants, which satisfies our team's main objective and addresses the real-world issue. Through interacting with the visualizations, users could explore data on exposure to pollutants in each country within a time scale via the choropleth, the life lost by illness due to each type of pollutant via the bar chart, and the relationship between air quality and health via the bubble chart. Moreover, using the input form, they could study the predicted life lost in a country due to an air pollutant knowing the exposure to that contaminant. With these, users could better understand the situation of air pollution and its health impact throughout time specifically in individual countries.

For future improvements and enhancements, our team hope to do better on the validation of the user input, as well as implement test cases to handle errors. These will make our website more robust and complete, as we better understand the weakness of our functionalities to deploy updates. Besides, we want to improve the performance of data visualisations and AI models to be efficient, as there are some delays due to the huge volume of training data. Moreover, we hope to integrate more visualisations and AI models for users to have a variety of information to discover. This aims to improve the storytelling aspect of the application.

6 Bibliography

Bostock, M 2011, *D3 by Observable | The JavaScript library for bespoke data visualization*, D3, viewed 16 October 2024, <https://d3js.org/>.

Chart.js | Open source HTML5 Charts for your website, 2024, Chart.js, viewed 16 October 2024, <https://www.chartjs.org/>.

IBM 2024, *IBM Design Language – Color*, Ibm.com, viewed 01 November 2024, <https://www.ibm.com/design/language/color/#specifications>.

Munzner, T 2014, *Visualization analysis & design*, A K Peters Visualization Series, Crc Press, New York.

Pallets, n.d., Welcome to Flask — Flask Documentation (3.0.x), viewed 01 November 2024, <https://flask.palletsprojects.com/en/stable/>.

Ramírez, S n.d., FastAPI, tiangolo, viewed 16 October 2024, <https://fastapi.tiangolo.com>.

Rottach, K G, Zivotofsky, A Z, Das, V E, Averbuch-Heller, L, Discenna, A O, Poonyathalang, A, & Leigh, R J 1996, *Comparison of Horizontal, Vertical and Diagonal Smooth Pursuit Eye Movements in Normal Human Subjects*, Vision Research (Oxford), vol. 36, no. 14, pp. 2189–2195, <https://doi.org/10.1016/0042-6989(95)00302-9>.

scikit-learn developers 2024, scikit-learn: Machine Learning in Python, scikit-learn, viewed 26 September 2024, <https://scikit-learn.org/stable/index.html>.

The pandas development team 2024, pandas documentation, pandas, viewed 26 September 2024, <https://pandas.pydata.org/docs/index.html>.

Tufte, E R 2001, *The visual display of quantitative information*, Graphics Press, Cheshire Connecticut.