**Readme**

1. **Introduction**

The route reviews collected from the crowd users through the application includes five factors namely, Road Condition, Women Safety, Water Logging, Traffic Congestion, and Frequent Gas Filling Stations. Hence, the review factor column in the dataset contains a five-tuple where each of the factors has been marked on a 1 to 5 scale. We have removed all user ids from the dataset for making it completely anonymous. The dataset contains the following columns.

* \_id : a unique identifier for the reviews.
* createdAt and updatedAt: timestamps when the initial entry was provided and when it is last updated.
* From and To: source and destination of the journey for which the reviews are provided.
* pathLat and pathLong: Latitude and Longitude of the source and destination.
* RF: the review ratings as a five-tuple for the review factors.

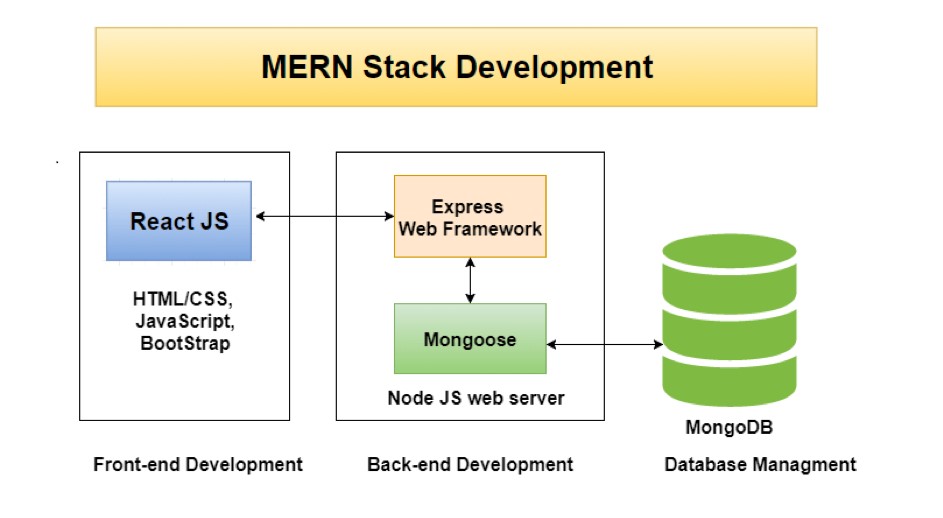
The subsequent sections discuss in detail the implementation strategy of the proposed crowdsourced road recommendation system along with suitable code snippets.

1. **Implementation details**

The application is extensive made with MERN stack. MERN stands for MongoDB, Express, React, Node, after the four key technologies that make up the stack.

* MongoDB - document database
* Express(.js) - Node.js web framework
* React(.js) - a client-side JavaScript framework
* Node(.js) - the premier JavaScript web server

The MERN architecture allows to easily construct a 3-tier architecture (frontend, backend, database) entirely using JavaScript and JSON. Fig-1 shows the MERN 3-tier architecture.



**Fig. 1** MERN architecture

**2.1 React-Bootstrap**

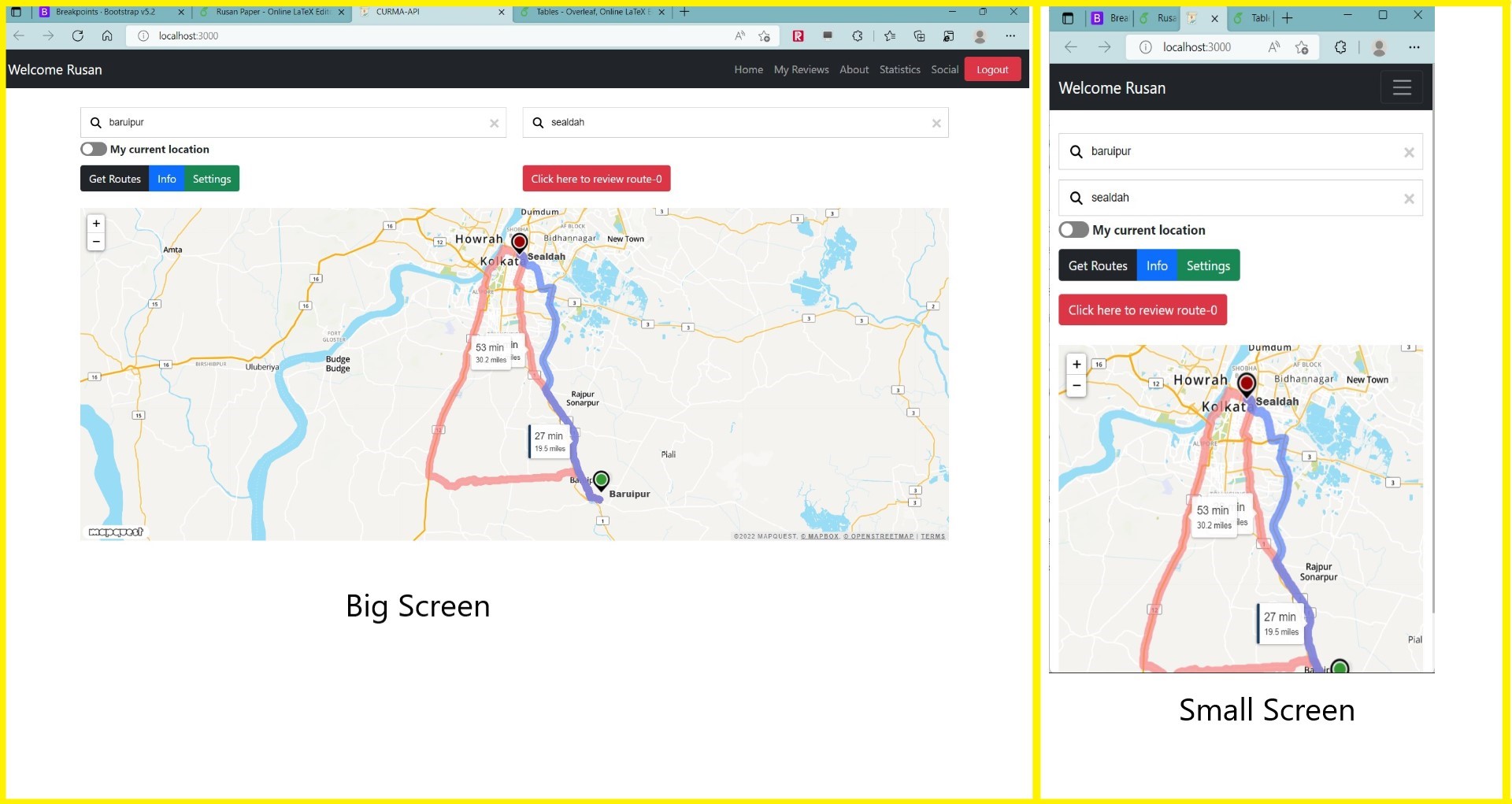
Our goal is to make our webapp responsive i.e. our app looks good in all types of screen sizes like mobile screen, laptop screens, etc. We use react-bootstrap for this purpose. Bootstrap works on breakpoints on the screen. According to these breakpoints, the bootstrap elements may be customized to look like. For example, on a small mobile screen, a Col can spread across the whole screen whereas on a large screen, it may be spread across on half of the screen. This can be done using *<*Col sm=”12” lg=”6”*>*content*<*/Col*>*. Here, 12 denotes the whole width of the screen and thus 6 denotes only half the screen. These breakpoints are regulated by media queries in the internal setup of bootstrap.

We used Container, Row, Column elements of bootstrap to properly align items in grid format.

Note: by default Row element of bootstrap has property of display:flex. In addition to this we used Modal element to make modals in our app and Table to print table. The navbar is made using Navbar element. Tab-1 shows the breakpoints for different screen sizes. Fig-2 shows the resulting rendering for different devices.

**Table 1** Bootstrap Breakpoints

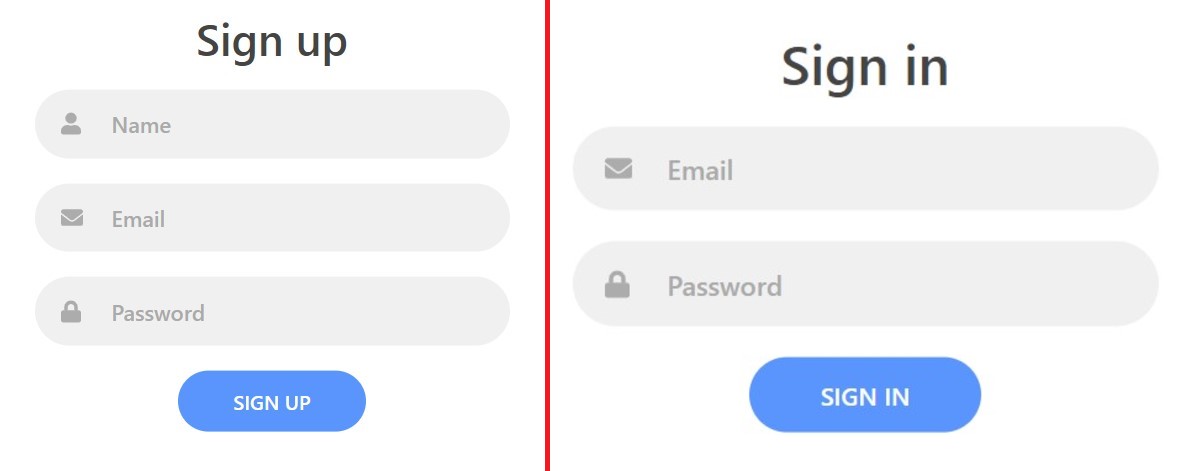
|  |  |  |
| --- | --- | --- |
| Breakpoint | Class infix | Dimensions |
| Extra small | None | *<*576px |
| Small | sm | ≥576px |
| Medium | md | ≥768px |
| Large | lg | ≥992px |
| Extra large | xl | ≥1200px |
| Extra extra large | xxl | ≥1400px |



**Fig. 2** Responsiveness of website

**2.2 Sign-up and Sign-in configuration**

For sign up there are several fields like name, email, password. When a user clicks on the sign up button, the fields are set as a property of an object and the object is sent to the server by post request through axios. Now as the server receives the user’s information via the object, it first validates whether the data is valid or not; for example whether the email is in correct format or the password has a minimum length or the values in the field are non-empty. If the validation fails which means user has submitted incorrect data, then the server sends a 401(unauthenticated) HTTP error code for incorrect email or password formats and 400 (Bad request) HTTP error code for empty or null values in some fields. Fig-3 shows sign up and sign in system in the app.



**Fig. 3** Signup and Signin feature

*let res* = *await axios.post*(′*/api/v*1*/auth/register*′*,obj*);

If the server validation is successful, it sends back a JWT (JSON Web Token) containing user info (not the password though), back to the front end. On the other hand, the server stores the user’s info into the Mongo DB database and the password is stored in encrypted format using Bcrpyt.js. The encryption is a good practice since even the organization themselves won’t know the passwords of all the users and safeguards user’s data privacy.

*let res* = *await axios.post*(′*/api/v*1*/auth/login*′*,obj*);

For sign in, similar approaches are used. The user’s details are sent in object format to the server by a post request through axios. Firstly, its checked whether the fields sent to the server is non-empty or not. Then it looks up in the database for the same record and checks whether the password matches or not. If password doesn’t match, 401(unauthenticated) HTTP error code is sent back as response.

**2.3 JSON Web Token(JWT)**

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs is signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA. In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

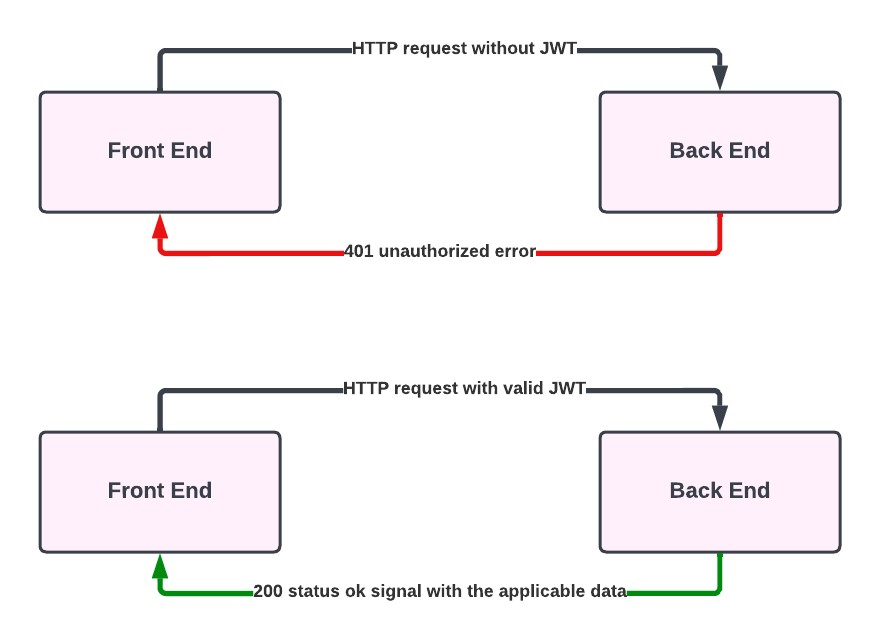
* Header
* Payload
* Signature

Therefore, a JWT typically looks like xxxxx.yyyyy.zzzzz. The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as, HMAC, SHA256, or RSA. To create the signature part, the server has to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that. The payload is the part where user details like name, email, etc are stored (not password).

All the routes of the server are protected which means, the user has to send this token along with the data to be a valid request to the server. It is sent in the Authorization header using the Bearer schema.

*Authorization* : *Bearer < token >*

Fig-4 shows the interaction of the frontend with the backend by JWT and without JWT.



**Fig. 4** HTTP request with and without JWT

**2.4 Bcrypt.js**

Bcrypt.js is a javascript package in node js that is used to hash password in our database instead of using just a plain text. It incorporates a salt to protect against rainbow table attacks, bcrypt is an adaptive function: over time, the iteration count can be increased to make it slower, so it remains resistant to brute-force search attacks even with increasing computation power. To hash a password in database:

const salt = bcrypt.genSaltSync(10);

//bigger the salt, greater the security but also slower in hashing and matching passwords

const hash = bcrypt.hashSync(”B4c0//”, salt);

To check a password the following code snippets are used.

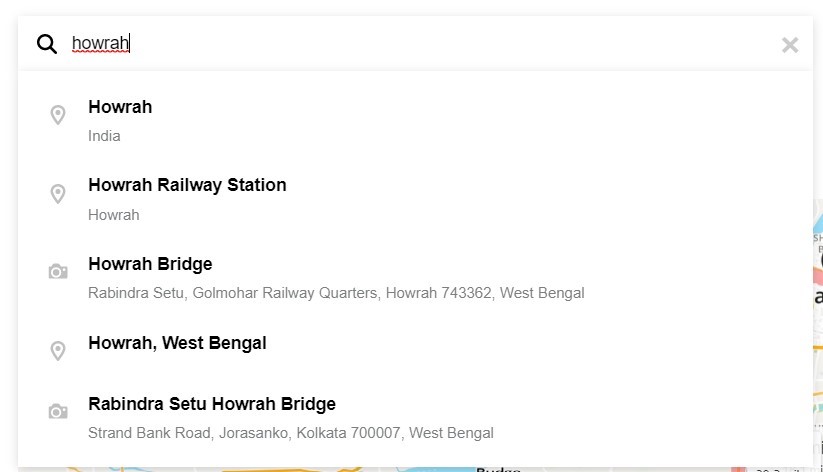
// Load hash from your password DB.

bcrypt.compareSync(”B4c0//”, hash); //true bcrypt.compareSync(”notbacon”, hash); // false

In the application, the passwords of the user are first encrypted using a salt. The more the length of the salt, the better the encryption.

**2.5 Home Page details**

Home page is the main hub of activity of the application. Here, a user can enter their source or destination and get multiples routes on the map with a single route highlighted based on their preference like road condition, women safety, etc. A user can also select a particular route and review that route. TomTom API has been utilized here to deal with the maps and routes.



**Fig. 5** Search autocomplete feature(TomTom)

Fig-5 shows an example of search prediction for a text entered by the user in the search box. For using TomTom search box autocomplete, we have to a create an object of the type SearchBox and pass in the constructor some search options like labels, idle time press, mininum number of character to start autocomplete, etc; along with the Services object from *@tomtom-international/web-sdk-services* package.

const ttSearchBox = new SearchBox(services, searchBoxOptions);

useEffect(()⇒searchRef.current.appendChild(ttSearchBox.getSearchBoxHTML())

,[]);

Here searchRef is a useRef react hook which refers to the search box which implements search autocomplete. So, we add our search box dynamically in our webpage by appendChild method of javascript. This is being done in a useEffect hook with empty dependency list, which means that only once as the contents of this page is loading this useEffect function will run.

We also have a current location button in the application. If it is pressed then the current location of the user automatically gets updated in the source field.

navigator.geolocation.getCurrentPosition(showPosition,showError);

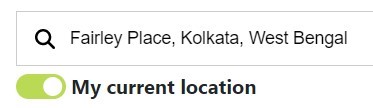
The above line is executed as the current location button is pressed. *showPosition* is a function that is used to perform the activities after we get the location of the user and *showError* is another function which is used to handle error in case we don’t get user’s location. In *showPosition* function, we call the reverse geocoding APIof TomTom using the user’s current latitude and longitude. The API replies back with the user’s address. Then this address is rendered in the source search box.

const res=await axios.get(‘https://api.tomtom.com/search/2/reverseGeocode/$*{*latitude*}*,

$*{*longitude*}*.json?key=$*{*process.env.REACT APP TOMTOM KEY*}*&language=en-GB&

entityType=MunicipalitySubdivision‘)

In Fig-6, an example of use of current location is shown.



**Fig. 6** Current location feature

Map rendering is performed using the Mapquest API. After the user selects his/her source and destination in the respective fields, he or she presses the get routes button to get the map and the routes between the respective location. Firstly, the key is set in the mapquest object by *L.mapquest.key*property. Then a *directions*object is made by calling L.mapquest.directions(). Then we use the *route()*function of it, where we supply the source (that is typed by user), destination (that is typed by user) and other details in the form of an object and pass another parameter createMap()function to create a map according to the property specified in the object passed.

L.mapquest.key = process.env.REACT APP MAPQUEST KEY

let directions = L.mapquest.directions();

directions.route(*{* start: document.getElementsByClassName(

’tt-search-box-input’)[0].value,

end:document.getElementsByClassName(

’tt-search-box-input’)[1].value, options: *{*

timeOverage:100, maxRoutes: 5, *}*

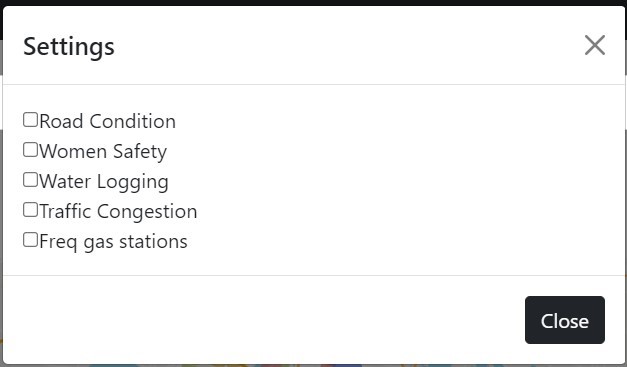
*}*,createMap);

In the createMap()function, first L.mapquest.map(’map’,obj)is called to create the map object where obj is an object where we pass important properties of the map like it’s center, layer, zoom, etc. Then we create a directionsLayerobject by calling the const customLayer=L.mapquest.directionsLayer(obj)where in *obj* we pass important properties of the start marker, end marker, route ribbon, alternate route ribbon, etc. Then we set the customLayer.on(’route selected’, function(eventResponse) *{…}*

In the event response function, the piece of code which needs to execute when the user selects a route is written. First, in the function, the *maneuvers* (containing array of latitudes and longitudes of the route selected by the user) are extracted from the *eventResponse*parameter passed in the function. Then the *maneuvers* are stored in the local storage for further access in different part of the website. We also store the source and the destination name in the local storage for the same reason. Then the *customLayer*is added to the map by customLayer.addTo(map);

**2.5.1 Route Selection**

After the map is rendered in the browser, by default a route (most likely the one having minimum distance) is shown as the main route and others as an alternate route. There are checkboxes available to the user for the review factors that will be taken into account and the corresponding main route will be calculated and displayed. Fig-7 shows a screenshot where a user can select his choice of review factors according to which route recommendations will be provided.



**Fig. 7** Checkboxes for route selection

So at first(while rendering the map itself), we send a HTTP POST request to the server along with the array of *maneuvers* (array of latitudes and longitudes) and the server returns an array containing average rating of each part of the part. This is possible because in the database we maintain a record for each part of the path, the user reviewed and that part’s average rating is returned. This is more clearly explained in the paper.

await axios.post(‘/api/v1/latlng‘,*{*maneuvers*}*,*{*headers: *{*

Authorization: ‘Bearer $*{*localStorage.getItem(”token”)*}*‘,}});

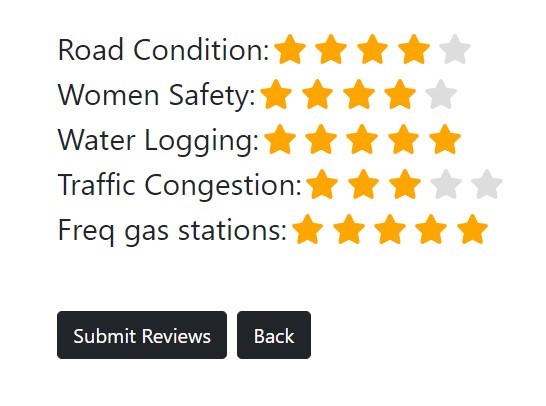
Here, we can notice that this token is actually the JWT token we got during login or sign-up. Now, for each review factor we store the sum of the respective ratings (of those review factors which the user wants) of all the parts in an array. Then from the array we see which route has the maximum rating. After knowing which route is to highlighted as the main route, that route is selected by the statement customLayer.selectRoute(reviewOptimization.optimizedRoute);

Here, *customLayer* is the layer of the map and selectRoute()is the function to change the main route. selectRoute(0) selects the default route (by Mapquest). *reviewOptimization*is a global object that keep records of results returned by the server after the HTTP POST request to ’/api/v1/latlng’and we are setting the *optimizedRoute*property to the route number which we calculated before.

**2.5.2 Review submission**

The user can select his or her route by clicking on it and can decide to review that particular by clicking on a button. Now, as the user selects a route, as stated earlier, the maneuvers (containing array of latitudes and longitudes) are saved in local storage by JSON.stringify(maneuvers)along with the source and destination. Then the user is directed to a page where they can give rating to all types of review factors.

After the user gives the rating of all the review factors and clicks on the submit button, its checked whether the user has given reviews to all the review factors or not. If not, then the user is being requested to review properly and submit it again. The *maneuvers* are separated into two arrays of latitudes and longitudes. The source and destination are also retrieved from the local storage. All these information about the related path and the review of the user are stored in the form of an object and sent via HTTP POST request to server. Then server goes on a series of validation as explained in the paper in details. Fig-8 shows the different review factors and the ratings a user has given.



**Fig. 8** Review submission by user

await axios.post(‘/api/v1/reviews‘,obj,*{* headers: *{*

Authorization: ‘Bearer$*{*localStorage.getItem(”token”)*}*‘,}});

await axios.post(‘/api/v1/train‘,obj,*{*headers: *{*

Authorization: ‘Bearer $*{*localStorage.getItem(”token”)*}*‘,}});

await axios.post(‘/api/v1/validate‘,obj,*{*headers: *{*

Authorization: ‘Bearer $*{*localStorage.getItem(”token”)*}*‘,}});

HTTP POST request to */api/v1/reviews*is used to submit the user’s rating, whereas */api/v1/train*is used to check whether review is fake or not against our pretrained machine learning model and */api/v1/validate*is to check whether the review is spam review or the review’s originality can be confirmed by user’s location.

**3. Security Features**

There are mainly four *npm* packages that is used to maintain security and illegal access of users in the application.

* *JWT*
* *Bcrypt.js*
* *Dotenv*
* *Cors*

There are certain values we cannot hardcode in our server end or front end like key of an application or password of database. For this we use *dotenv*package. We never hardcode our key directly but take the value from system variables. These variables can be setup in .env file in Node JS and *.env.local* file in React JS in the local environment and in deployment we can set it up in the domain website configuration. We hide the MONGO URI, JWT SECRET, JWT LIFETIME in the Node JS configuration. MONGO URI contains address along with password to connect with Mongo DB. JWT SECRET is a 256 bit generated text to sign the JSON web token. JWT LIFETIME is the validity of the JWT. Similarly, in react configuration we have REACT APP MAPQUEST KEY and REACT APP TOMTOM KEY. They are app keys of Mapquest and TomTom. We can access them anytime in the website by example:

process.env.MONGO URI, process.env.REACT APP MAPQUEST KEY

*Cors*package is used to modify the type of requests which can access the server. We can enable cross-origin requests as well as disable it. We can also use it to enable cross origin only for a particular domain. So, a lot of customization can be done with the cors package and it provides us security in the fields and domain we desire.

app.use(cors())

This allows all cross-origin reference. Any site can send HTTP request to the server.

const corsOptions = *{* origin: ’http://example.com’,

optionsSuccessStatus: 200*}*

app.use(cors(corsOptions))

This is an example where we send a *corsOptions* object in *cors* as parameter. Here, we have set the origin to a website. So, requests from only this website or domain will be served. Any request from any other domain would not be entertained by the server and will throw CORS error in the browser.

**4. Configuring the Regressors for route validation**

A separate controller is made in Node JS for machine learning. We can access this controller via a HTTP GET request with a request param in the URL:*/api/v1/train/:type*. In the controller firstly reviews are taken from the database by : const record=await Review.find(*{}*).

We specify the machine learning regression algorithm we want in the type parameter.

We have used 7 ML regressor algorithm namely Random Forest Regression, Simple Linear Regression, Decision Tree Regression, Polynomial Regression, Exponential Regression, Theil Sen Regression and Multivariate Linear Regression.

For example, for Random Forest regressor, we set in options object seed: 3, replacement:true, nEstimators: 25.

const *{*RandomForestRegression*}*= require(’ml-random-forest’); const regression = new RandomForestRegression(options); regression.train(trainingSet, predictions);

const result = regression.predict(trainingSet);

**5. Deployment**

We deployed our app in *heroku[[1]](#footnote-1)* as the deployment was free of charge at that time. There are a few configuration which needs to be done before we deploy our application to heroku. At first, we have to add a few lines of code in our app.js file from where our whole Node JS server runs.

if(process.env.NODE ENV === ’production’)*{* app.use(express.static(’./client/build’));

app.get(’\*’,(req,res)⇒ { res.sendFile(path.resolve( dirname,’client’,’build’,’index.html’)) *}*)

*}*

Our app is in deployment it will run a Postscript build and store the static pages of react in ’./client/build’. By the above lines of code snippet we are telling the server that when it is in production and it gets HTTP request other than the ones defined in the backend, it will render the index.html page that is generated by post build script.

In our *package.json* file we also have to make some changes like we have to set

”dev”:”concurrently \ ”npm run server \ ” \ ”npm run client \ ”” ”heroku-postbuild”: ”NPM CONFIG PRODUCTION=false

npm install −−prefix client && npm run build −−prefix client”

The first one is for starting the React JS and Node JS application simultaneously. We need to install concurrently package for executing this. We can always start the server along with React application in our local environment with *npm run dev*. The second one is for running the *heroku* postbuild script which tells the server to run *npm build* in the React application (when not in production).

The next step is to global install heroku CLI and git. Then we need to create our account in heroku. Then we should run the following commands:

$ heroku login

$ git init (If we are making the git repository for the first time) $ git add .

$ git commit -am ”make it better” $ heroku create -a curma-api

OR

$ heroku git:remote -a curma-api

$ git push heroku main

So, first we are logging into our *heroku* account. Then we initialize our git repository if it is not. Then we add all the changes made in the git. We commit our changes in the git. If we already created our remote site repository in our heroku account we can use *heroku git:remote* otherwise we need to create a new repository in *heroku* for deployment. Then we push our application to heroku server by git push heroku main (in some cases main has to be replaced by master). One last final thing to do is to set up our environment variables in the server. To do this we go in our app repository in heroku and click on the settings tab. There we can click on the Reveal Config Vars button and write our secret variables like mongodb connecting url and map keys.

1. https://www.heroku.com/ [↑](#footnote-ref-1)