



Table of Contents

[**Objective 1**](#_heading=h.gjdgxs)

[**Project Scope 2**](#_heading=h.30j0zll)

[**Technical Challenges 3**](#_heading=h.1fob9te)

[**Technical Architecture 4**](#_heading=h.3znysh7)

[**About Data 4**](#_heading=h.2et92p0)

[**Assumptions 5**](#_heading=h.tyjcwt)

[**Design 5**](#_heading=h.3dy6vkm)

[**Data Flow 6**](#_heading=h.1t3h5sf)

[**Frontend (Application): 6**](#_heading=h.egs0wv86jqtf)

[**Backend (Machine Learning final\_preds.py): 10**](#_heading=h.99agcky50b59)

[**Issues 16**](#_heading=h.4d34og8)

[**Results 17**](#_heading=h.2s8eyo1)

[**Appendix 18**](#_heading=h.17dp8vu)

[**Sources 18**](#_heading=h.3rdcrjn)

# **Objective**

The objective of this project is to build an App named “Urgent” which would produce the list of nearest and lowest wait time urgent care facilities so that users can feel a sense of ease of transportation during an emergency. It would help customers know which hospital to go to incase of an emergency as the app would tell where there is less wait time based on location and distance. The app would take the user’s zip code as an input and automatically capture their exact location to list out the urgent care facilities near to them. ERtrack.net allows us access to the hospitals data using an API link. It has details about the facilities like the hospital name, address, latitude, and longitude as well as the real time wait time. The database of ERtrack.net gets updated every 15 mins.

The ERTack.net data serves as our main source of data. Only the hospitals present in its database is made available as an output for our App. A paid google maps API, will allow Urgent to take real time traffic data points and distance between the user and the Hospital.

The business value that the sponsor can receive from this project is multi-fold-:

Firstly, by developing such an app, the sponsor can create a competitive advantage in the market by providing a convenient and efficient solution to their customers.

Secondly, the app can improve the operational efficiency by reducing the workload on medical staff. Customers can use the app to check the wait times at different urgent care facilities and choose the one that best suits their needs. This can reduce the number of phone calls and inquiries received by the  staff.

Thirdly, the app can provide valuable insights into customer preferences and behavior. The sponsor can use this data to optimize their operations, improve their services, and tailor their marketing campaigns to better target their customers.

# **Project Scope**

The application would likely require integration with various data sources such as hospital wait time data, mapping data, and real-time traffic data to provide accurate recommendations. In addition, we need to take into consideration user testing, quality assurance, and ongoing maintenance and support of the application. The development team may need to work closely with stakeholders such as urgent care facilities, hospitals, and third-party data providers to ensure the number of daily patient arrivals and the reasons for their visits for a more detailed look out of the wait times in each facility. They may also have to include marketing and promotion of the application to potential users, as well as ongoing monitoring and analysis of user behavior and feedback.

We had tried to integrate the insurance providers' data to allow customers to check if the hospital or urgent care facility is covered in their insurance plan, however the data was unprocurable due to access limitations. We had also tried to incorporate the services provided by each facility into the app but the data for services provided by each of the facilities included in the ERTrack.net source was unavailable and so was outside the scope of this project.

# **Technical Challenges**

For a few sets of hospitals, the ERtrack.net API throws an Internal Server Error while trying to fetch the wait times or there was no data available for example, some facilities had -1 as wait times other than their closing time. For these hospitals, we have imputed the missing wait times using the KNNImputer.

The current wait time was available on the ERTrack.net website, however we wanted to display the next wait time when the patient reaches the facility. For this we had to make predictions based on the historical wait time data present on the website. We explored multiple techniques like LSTMS, ARIMA model, Moving Average model and finally went forward with the moving average model due to its quick turnaround time and accuracy.

We had to come up with a prediction algorithm that is fast since we cannot make the patients wait in such an emergency. Moving average was the quickest of all and hence we went forward with the same.

In the app, we fetch the patient’s current location and look for facilities not only in the same state as the user but also in their nearby states to address the situations where the patients live in borderline regions.

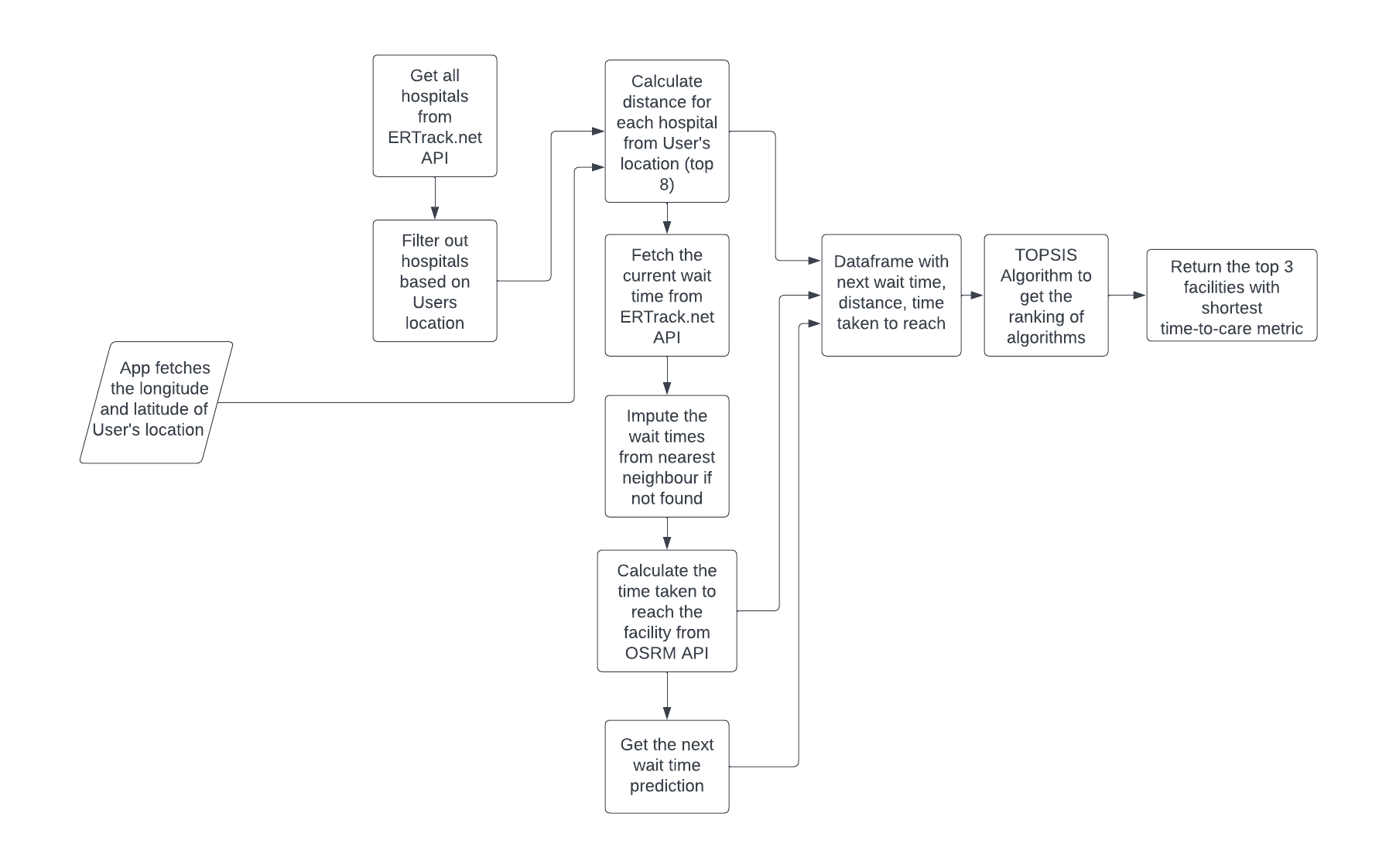
We had to check for patterns in the wait time prediction to track the closing time and other unavailability of the facilities before making a prediction on them.

We have given more weightage to the distance as compared to the next wait time and traffic time as we don’t want the patient to travel too far in an emergency. To achieve this, we have used the TOPSIS algorithm to rank the top 3 facilities based on the user’s location.

The app has been built and it works when run on the PyCharm software. The APK file however needs some debugging since it crashes when installed on the phone. This might also be because the machine learning logic needs a lot of time to fetch the data from the website for each hospital. Due to limited skills, we were unable to debug the APK file further. However, this can be taken up as a phase 2 process along with further enhancements and optimization of the app.

For Optimizing the machine learning logic we tried the asynchronous calls to the data sources using the asyncio library however, the performance did not improve drastically so we did not incorporate it into the final code. The code were we tried to implement the asyncio is present in the Code\_tried.ipynb file.

# **Technical Architecture**



## **About Data**

Our primary source of data is **ERTrack.net API**. It aims to record live updating ER and Urgent Care wait times from publicly available sources. We are using two endpoints to access the data, <https://ertrack.net/api/hospitals/> to fetch all the hospitals and [https://ertrack.net/api/hospital/{hospital\_id}/history/](https://ertrack.net/api/hospital/%7bhospital_id%7d/history/) to get the wait time of a single facility. Both the endpoints return the data in the form of JSON objects which are then transformed into pandas’ data frame for further processing. For each hospital, we get the hospital id, hospital name, address, latitude, longitude, county, state and fips code from the first endpoint and observation times and wait time from the second endpoint. There are 3267 hospital data present in ERTrack.net.

We are using **Open Source Routing Machine’s API (OSRM)** to calculate the time taken to reach a facility from the users current location. We are using [http://router.project-osrm.org/route/v1/car/{lon\_1},{lat\_1};{lon\_2},{lat\_2}?overview=false](http://router.project-osrm.org/route/v1/car/%7blon_1%7d,%7blat_1%7d;%7blon_2%7d,%7blat_2%7d?overview=false) endpoint to fetch the time taken between two locations by passing the latitude and longitude of both the locations as parameters to the APIs endpoint.

Apart from OSRM, we are also using **Geopy** as another source of data in our application. Geopy is a Python client for several popular geocoding web services. It makes it easy for Python developers to locate the coordinates of addresses, cities, countries, and landmarks. The mobile app fetches the users current location in the form of latitude and longitude using the GPS of the mobile device. These coordinates are then passed to the machine learning logic where we are using geopy’s **Nominatim** function to get the exact address details of the user.

We are also using **Geopy’s** **geodesic** function to measure the distance between two points using the geodesic distance or great-circle distance calculation method. The geodesic distance is the shortest distance on the surface of an ellipsoidal model of the earth. We are using this function in our code to calculate the distance between the user's location and all facilities in the nearby region. This is done to filter out and obtain the top 8 facilities based on shortest distance to the user.

## **Assumptions**

1. The App will only display the hospital/facilities present in the ERTrack.net database
2. For certain hospitals, where the wait time is unavailable (because of error in the ERTrack.net API), the wait time is just an approximation since they are imputed using the KNN Imputer.
3. Up-to-date data will always be available in ERTrack.net as it gets refreshed every 15 mins.
4. The latest data gets updated in the ERTrack.net database with incorrect date and timestamp. However, the latest record for the wait time is the wait time for the current date and time. Based on all this historical data, we have tried to predict the next wait time which is nothing but the wait time that would be there when the user reached the facility.
5. Considering all facilities in the ERTRack.net dataset irrespective of type ( urgent care, ER or medical labs).

## **Design**

* **Geopy API:** Geopy is a Python client for several popular geocoding web services. Geopy makes it easy for Python developers to locate the coordinates of addresses, cities, countries, and landmarks across the globe using third-party geocoders and other data sources. Geopy is tested against CPython (versions 3.7, 3.8, 3.9, 3.10, 3.11) and PyPy3. geopy 1.x line also supported CPython 2.7, 3.4 and PyPy2. We have used Nominatim to extract the address details from the user location coordinates and Geodesic function to measure the distance between two locations.[3]
* **KNN Imputer:** We have used KNNImputer by scikit-learn k-Nearest Neighbors Algorithm for imputing missing wait times in case of an error while fetching the wait times from ERTrack.net.[6]
* **OSRM API:** It is a C++ implementation of a high-performance routing engine for shortest paths in road networks. It is a flexible import of OpenStreetMap data and supports car, bicycle, walk modes. We have used it to estimate the real time traffic and calculate the time taken to reach each facility from the user's location.[2]
* **TOPSIS Algorithm :** The TOPSIS algorithm is a multi-criteria decision-making method used to rank alternatives based on their distance to ideal and nonideal solutions. It stands for Technique for Order of Preference by Similarity to Ideal Solution. We will be using the TOPSIS algorithm to rank the results. We need to give more weightage to the travel time as a patient in emergency would not like to travel far even if the facility has a lower wait time. We are using the TOPSIS algorithm for determining possible solutions based on the shortest distance from positive ideal solution and furthest from negative ideal solution. It is considered to give similar results to human behavior[1].
* **Kivy:** Kivy is a free and open source Python framework for developing mobile apps and other multitouch application software with a natural user interface. With Kivy, you can use Python to create applications suitable for Android, iOS, Linux, Windows, and many other web applications. The library uses Python and Cython as a base.[4]
* **Buildozer:** Buildozer is a tool that aims to package mobile applications easily. It automates the entire build process, downloading the prerequisites like python-for-android, Android SDK, NDK, etc. Buildozer manages a file named buildozer.spec in your application directory, describing your application requirements and settings such as title, icon, included modules etc. It will use the specification file to create a package for Android, iOS, and more.[5]

## **Data Flow**

### Frontend (Application):

We have built this application using the Kivy library in Python. We have built the APK file for both Android as well as iOS devices along with the release files for deployment on Google Play Store. Since releasing the app on Google Play Store was a paid service and could only be done after thorough testing, we haven’t implemented the same.

To create the application we use PyCharm software and there were multiple files created for different elements of the application. Each of the files are described below:

**1. main.py :**

This is a Python script using the KivyMD framework to create a mobile application. It defines a MainApp class that extends the MDApp class, and it contains the following methods:

on\_start(): This method is called when the app starts and it sets the theme of the app to red.

navigation\_draw(args): This method is a placeholder for a navigation draw, which is a widget that allows users to access different parts of the app by opening a menu or drawer from the side of the screen.

change\_screen(screen\_name, direction='forward', mode=""): This method changes the screen that is currently displayed in the app. It takes a screen\_name parameter that specifies the name of the screen to display, and optional parameters direction and mode that specify the direction and mode of the screen transition.

The script also defines three classes that extend the Screen class: HomeScreen, PredScreen, and MainScreen. These classes are used to define the content of each screen in the app.

In addition, the script imports several modules and classes from other files:

specialbuttons: This module defines custom button widgets that are used in the app.

ScreenManager: This class is used to manage the different screens in the app.

Label: This class is used to create text labels in the app.

HomeMapView: This class is defined in the homemapview module and is used to display a map in the app.

HomeGpsHelper: This class is defined in the homegpshelper module and is used to get the user's current GPS location.

final\_call: This function is defined in the final\_preds module and is used to get a list of nearby locations based on the user's GPS location.

**2. main.kv :**

This is a Kivy language file (.kv) that defines the layout of the main screen of the app. It contains a Screen widget that contains a MDToolbar widget and a NavigationLayout widget, which in turn contains a ScreenManager widget.

The MDToolbar widget is a Material Design toolbar that displays the app title and has an elevation of 8 (which gives it a shadow effect). Its background color is set to the primary color of the app's theme, which is red.

The NavigationLayout widget is used to create a layout that allows users to navigate between different screens in the app. It contains a ScreenManager widget that manages the different screens of the app.

The ScreenManager widget has two screens defined: HomeScreen and PredScreen. These screens are defined in separate .kv files (homescreen.kv and predscreen.kv), which are included using the #:include directive. The HomeScreen screen has the name home\_screen and the PredScreen screen has the name pred\_screen.

The size\_hint and pos\_hint properties are used to set the size and position of the ScreenManager widget within the NavigationLayout widget. The size\_hint is set to (1, 1), which means it will take up the entire available space. The pos\_hint is set to {"top": 1, "left": 1}, which means it will be positioned at the top-left corner of the NavigationLayout widget.

The anchor\_title property of the MDToolbar widget is set to "center", which centers the title text within the toolbar. The size\_hint of the MDToolbar widget is set to (1, .1), which means it will take up 10% of the available height.

**3. homescreen.kv :**

This is a Kivy language file that defines the layout of the HomeScreen of the mobile app.

The HomeScreen consists of a FloatLayout, which contains a MDToolbar, a HomeMapView, and a GridLayout with a LabelButton.

The MDToolbar displays the title "Set Your Location on the Map" and has a primary light background color.

The HomeMapView is a custom widget that displays a map and allows the user to set their location on the map.

The GridLayout contains a LabelButton with the text "OK". The LabelButton has a background color that depends on its state (normal or pressed), and its on\_release event changes the screen to the PredScreen by calling the change\_screen() method of the MainApp class.

**4. predscreen.kv :**

It defines the UI screens for a mobile app.

The HomeScreen contains a MDToolbar with a title, a HomeMapView, which I assume is a custom widget that shows a map view of the user's location, and a GridLayout containing a LabelButton. The LabelButton has a release event that triggers the change\_screen() method of the app, which changes the current screen to the PredScreen.

The PredScreen contains a MDToolbar with a title, a GridLayout containing a LabelButton and a MDLabel. The LabelButton has a release event that displays the contents of the flat\_list attribute of the app in the MDLabel. The GridLayout also contains another LabelButton that triggers the change\_screen() method and goes back to the HomeScreen.

Overall, the app is designed to allow the user to set their location on a map and display nearby healthcare facilities on the PredScreen.

**5. gpsblinker.py :**

This is a custom class GpsBlinker that inherits from MapMarker in Kivy Garden's mapview module. The GpsBlinker class adds the ability to blink the marker on a map view using the blink() method.

The blink() method uses the Animation class from Kivy to change the outer\_opacity and blink\_size properties of the marker. The animation starts with outer\_opacity=1 and blink\_size=25 (which is the default size of the marker), and it changes them to outer\_opacity=0 and blink\_size=50. The animation is bound to the reset() method, which is called when the animation completes.

The reset() method is responsible for resetting the outer\_opacity and blink\_size properties to their default values and then calling the blink() method again to repeat the animation. This creates a continuous blinking effect for the marker.

**6. gpsblinker.kv :**

This file defines a widget that can be used to create a blinking marker on a map. This is the Kivy language file for the GpsBlinker widget. It defines the appearance of the widget with two circles, an inner and an outer one. The outer circle is the blinking circle, which changes its size and opacity to create the blinking effect. The inner circle has a fixed size and is always red.

The default\_blink\_size is set to 25, which is the default size of the inner circle. blink\_size and outer\_opacity are properties that control the size and opacity of the outer circle, respectively.

The canvas.before block contains the drawing instructions for the widget. The outer circle is drawn with the color defined by the app.theme\_cls.primary\_color property, which is a tuple of RGBA values. The opacity of the color is controlled by the outer\_opacity property. The size of the circle is defined by blink\_size and its position is calculated based on the size of the widget. The inner circle is drawn with a fixed size and color.

**7. homegpshelper.py :**

This code defines a HomeGpsHelper class with methods to handle GPS-related functionality in a Kivy app. Here's a summary of what each method does:

run: This method is called to start GPS tracking. It first gets a reference to the GpsBlinker object and starts blinking. On Android, it requests location permissions and, if granted, starts GPS tracking with a minTime of 1 second and minDistance of 1 meter.

update\_blinker\_position: This method is called when a new GPS position is received. It updates the position of the GpsBlinker object and centers the map on the new position if it hasn't been centered yet.

on\_auth\_status: This method is called when the GPS status changes. If GPS is enabled, it does nothing. Otherwise, it opens a popup dialog to ask the user to enable GPS access.

open\_gps\_access\_popup: This method opens a popup dialog to ask the user to enable GPS access. It schedules the dialog to be opened after a delay of 2 seconds.

run\_dialog: This method creates and opens the GPS access popup dialog.

Note that this code relies on external modules like plyer and android.permissions which are used in the buildozer.spec file during the apk development.

**8. homemapview.kv :**

This code defines the view for the map on the home screen, which includes a GpsBlinker object that will be used to display the user's location on the map.

The HomeMapView class is defined and inherits from kivy\_garden.mapview.MapView. It has properties for latitude, longitude, and zoom level, and sets the initial latitude and longitude to the location of the Chicago Bean.

The double\_tap\_zoom property is set to True, which allows the user to double-tap on the map to zoom in. The on\_zoom event is defined to prevent the user from zooming out beyond a certain level by setting the zoom level to 12 if it is less than 12.

The GpsBlinker object is added to the map using the id of the blinker. The lat and lon properties of the GpsBlinker are set to the latitude and longitude of the HomeMapView.

**9. homemapview.py :**

This code defines a new class HomeMapView that inherits from the MapView class from the kivy\_garden.mapview module. The HomeMapView class is currently empty, meaning it does not have any additional properties or methods beyond those provided by the parent MapView class. The details for the map view screen is specified in the .kv file with the same name.

This class is intended to be used as the map view for the app, as evidenced by its name and its use in the kv file (home\_screen.kv).

**10. specialbuttons.py :**

ImageButton and LabelButton are two custom Kivy widgets that inherit from ButtonBehavior, which adds button-like functionality such as click detection and release behavior, to a Image and a Label, respectively.

ImageButton is an image-based button, which can be created by using the Image widget, setting its source property to the image file path, and adding ButtonBehavior to it.

LabelButton is a label-based button, which can be created by using the Label widget and adding ButtonBehavior to it.

Both widgets can be used like any other Kivy widget and can have properties and events customized to suit the application's needs.

**11. icon.png :** It is the icon/logo for our Urgent App.

There is a buildozer.spec file which is used to create the APK file for the application. The ‘Installation Steps for APK file.ipynb’ has all the steps that we used to install buildozer library and run it to create the APK and release files.

### Backend (Machine Learning final\_preds.py):

Once the user enters their zip code, the location services (GPS) will capture their current latitude and longitudes, this will be displayed in a map so that the user can change the pointer on the map.

The latitude and longitude will be passed to the main function.

***Function: main()***

***Input:*** Users latitude, Users longitude

***Output:*** top 3 hospitals with the time-to-care metric to be given out as predictions in the App

It takes two arguments, user\_lat and user\_lng, which represent the latitude and longitude of a user's location.

The function then retrieves data for all hospitals from a given API, removes hospitals where the longitude and latitude are None, removes unwanted characters from the address column, and adds a new column indicating whether each hospital is an emergency room (ER) or not.

The function then uses the get\_wt, cal\_time\_taken, get\_next\_wait\_time, and get\_topsis\_algo functions to generate a list of the top three hospitals with the shortest predicted wait times for the user.

Finally, the function converts the predicted wait times and total wait times for each of the top three hospitals into a human-readable format (hours:minutes:seconds) and returns a sorted list of the top three hospitals and their total wait times, sorted by shortest to longest total wait time.

All the hospital data will be fetched from the ERTrack.net endpoint and stored in a dataframe df\_all\_hospitals. We have removed the rows were the latitude and longitude are null and removed unwanted characters from the address column. Then we added a new column Is\_ER wherever the hospital name had an ER present in it.

Next will call the function get\_wt() and get the top 8 hospitals in a dataframe as output.

Once we get all the wait times (predicted and imputed) for all the top 8 hospitals, then we will call the function cal\_time\_taken() to calculate the actual time taken to reach facility.

Next we will call the get\_next\_wait\_time() to get the predictions for next wait time when the user arrives at the location. If the wait time is 0, then we have replaced it by one this is done as an adjustment for the TOPSIS algorithm to correctly rank the facilities.

Now since we have the predicted wait time, distance and the time taken, we will pass this as a dataframe to the TOPSIS algorithm function get\_topsis\_algo() to get the top 3 best predictions to be displayed in the app.

We still need to calculate the time-to-care metric. The final list of the urgent care facilities will be based on a time-to-care metric. This time-to-care metric is the sum of the next wait time(predicted) at the facility and the time taken to reach a facility from the user’s location. We will calculate this for the hospital ids given as output from the TOPSIS function and this will be given out as output of the main() to the App in the form of a list.

***Function: get\_wt()***

***Input:*** Users latitude, Users longitude and main dataframe

***Output:*** Dataframe of Top 8 hospitals with their hospital id, name, address, current wait time, type id, distance in miles, is it an ER flag, other values present (boolean).

It does this by first determining the state of the user based on their coordinates, and then filtering the hospital dataframe to only include hospitals in neighboring states and the user's own state.

Next, it calculates the distance between each hospital and the user, and appends this distance, along with other hospital information (e.g. name, address, etc.) to a list. This list is then converted to a dataframe and sorted by distance.

The function then calls another function, **get\_wait\_times**, to obtain wait times for each hospital. If a hospital has a NaN wait time, the function calls another function, **Knn\_imputer**, to impute a value for that hospital's wait time based on the wait times of other hospitals.

Finally, the function returns the top 8 hospitals with their wait times (imputed or not).

***Function:* remove\_unwanted\_chars()**

***Input:*** column

***Output:*** column

The **remove\_unwanted\_chars** function takes a column as input and removes any "#" characters from the column values using regular expressions

***Function:* parse*()***

***Input:*** string value

***Output:*** string value

The **parse** function takes a string as input and tries to parse it as a float, a date, or returns the original string if neither of these operations is possible. It has three nested **try-except** blocks to handle these cases. Here's a breakdown of how it works:

1. The first **try** block attempts to convert the input string to a float using the built-in **float** function. If this succeeds, the parsed float value is returned.
2. If the first **try** block fails (i.e., the input string is not a valid float), the second **try** block attempts to parse the input string as a date using the **parse** function from the **dateutil** module. This function attempts to automatically parse a wide range of date and time string formats. If the **parse** function succeeds, the parsed datetime object is returned.
3. If both the first and second **try** blocks fail, the third **except** block catches the exception and returns the original input string.

***Function:* get\_past\_week\_wait\_times()**

***Input:*** dataframe

***Output:*** past wait time

This function takes in a dictionary **data** containing observation times and wait times for a hospital and returns the wait time from one week ago.

First, it extracts the observation times and wait times from **data**. It then finds the most recent observation time and adds 30 minutes to it to get a future time. It then calculates the observation time 1 week before the future time.

The function then searches for the index of the observation time just greater than the calculated past time. If no observation time is greater than the past time, the function returns **None**. Otherwise, it returns the corresponding wait time.

Overall, this function is useful for comparing the current wait time at a hospital to the wait time from one week ago, allowing for analysis of trends over time.

***Function:* get\_wait\_times*()***

***Input :*** hospital\_id, is\_ER

***Output:*** wait time and boolean variable indicating the presence of other values (other than -1)

This function **get\_wait\_times** takes two input parameters - **hosp\_id** and **is\_ER** - and returns the wait time and a boolean flag indicating whether there are other values for wait time in the past history of the hospital.

Here's how the function works:

1. It initializes a dictionary **raw\_history** to store the raw data obtained from the API, and sets **other\_val** to False, which will be used to track if there are other wait times besides -1 in the history.
2. It sends an HTTP GET request to the API endpoint to obtain the past history of wait times for the given hospital ID. If the API call fails due to an HTTP error, the function sets **wait\_time** to -2 and sets the current time as the observation time.
3. If the API call succeeds, the function converts the raw data to a pandas DataFrame and extracts the latest wait time and observation time from the DataFrame.
4. If the latest wait time is -1, the function checks if there are other values for wait time in the past history. If there are other values, it sets **other\_val** to True.
5. If the latest wait time is not -2 and **other\_val** is True, the function checks the past week's wait times to determine if the hospital is closing soon. If the latest wait time is -1 and there are no other values for wait time in the past history, the function sets **wait\_time** to NaN.
6. If **is\_ER** is True and **wait\_time** is -1 or -2, the function sets **wait\_time** to 0 (since ERs are open 24/7).
7. The function returns **wait\_time** and **other\_val**.

***Function:* KNN\_Imputer*()***

***Input:*** top 8 hospitals dataframe

***Output:*** Imputed wait time dataframe

This function **Knn\_imputer** takes a DataFrame **df** as input and imputes missing values in the **wait\_time** column using K-Nearest Neighbor (KNN) imputation.

The first step is to extract the **wait\_time** and **distance** columns from **df** and drop any rows with null values. These columns are used to fit the KNN imputer.

Next, the KNN imputer is instantiated with **n\_neighbors=4**. The **fit** method is called on the data.

Then, the function identifies missing data using the **isna** method, and creates a new DataFrame called **missing\_data** that only contains the **wait\_time** and **distance** columns of the missing data.

The **transform** method is called on the KNN imputer using **missing\_data** as input. This returns the imputed values for **wait\_time** in a 2D array.

Finally, the missing values in the **df** DataFrame are replaced with the imputed values using **loc** and the **isna** method. The updated DataFrame is then returned.

***Function:* cal\_time\_taken*()***

***Input:*** latitude and longitude of the user and dataframe

***Output:*** dataframe

The cal\_time\_taken function takes three arguments: usr\_lat, usr\_lng, and df. It computes the travel time from the user's location to each hospital in the given DataFrame df, using the OSRM API. It returns the DataFrame with an additional column time\_taken that contains the travel time to each hospital in seconds.

Here is a breakdown of what the function does:

1. Create an empty list time\_taken to store the travel times.
2. Loop through each row in the DataFrame df.
3. Get the latitude and longitude of the hospital from the current row.
4. Make a request to the OSRM API to get the travel time from the user's location to the hospital.
5. Extract the travel time from the API response and append it to the time\_taken list.
6. Add the time\_taken list as a new column to the DataFrame df.
7. Return the updated DataFrame.

***Function:* get\_wait\_times\_df*()***

***Input :*** hospital\_id

***O*utput:** dataframe

This function retrieves the wait time history data for a hospital with a given ID from the API endpoint and returns it as a Pandas DataFrame. If the API request fails or returns an HTTP error code, the function returns -2. The function uses the **requests** library to make the API request and the **dateutil** library to parse the date strings in the API response. The function also uses the **urljoin** function from the **urllib.parse** library to construct the full URL for the API request.

***Function:* get\_next\_wait\_time*()***

***Input :*** dataframe

***O*utput:** dataframe

The **get\_next\_wait\_time(df)** function seems to be using another function **get\_wait\_times\_df(hosp\_id)** to retrieve the historical wait times for a given hospital, and then uses rolling average to predict the next wait time. I see that it handles cases where the historical wait times are not available or the prediction is not possible by assigning values -2 or -5 respectively. Overall, the function seems to be well-implemented.

***Function:* rank\_according\_to()**

***Input:*** dataframe and candidates

***Output:*** sorted candidates in descending order

This function **rank\_according\_to** takes in two arguments, **data** and **candidates**.

* **data** is an array of numeric values to rank.
* **candidates** is an array of the same length as **data**, containing the candidates that correspond to each value in **data**.

The function returns the candidates, sorted in descending order based on their rank according to the values in **data**.

The function first calculates the ranks of each value in **data**. It uses the **argsort** function to sort the data and returns the indices that would sort the array. Then, it initializes an array **almost\_ranks** with the same shape as **data**, and fills it with the sorted indices. Finally, it adds one to the ranks and subtracts one (to shift the range from 0-based to 1-based), and uses this to index into the **candidates** array, sorting it in descending order based on rank.

***Function:* get\_topsis\_algo()**

***Input:*** dataframe

***Output:*** top 3 hospital ids

The TOPSIS algorithm is a multi-criteria decision-making method used to rank alternatives based on their distance to ideal and nonideal solutions. It stands for Technique for Order of Preference by Similarity to Ideal Solution.

The inputs required are a dataset of alternatives with their performance on several criteria, and the weights of each criterion. The output is a ranking of the alternatives based on their overall closeness to the ideal solution.

The algorithm involves several steps:

1. Normalize the data: divide each value in the dataset by its column sum to make them all comparable on the same scale.
2. Calculate the weighted normalized data: multiply each value in the dataset by its corresponding weight.
3. Identify the best and worst values for each criterion based on whether it represents a benefit (maximization) or a cost (minimization).
4. Calculate the separation measures for each alternative, which represent how far it is from the ideal and nonideal solutions.
5. Calculate the similarities to the ideal and nonideal solutions for each alternative.
6. Rank the alternatives based on their similarity to the ideal solution and their separation from the anti-ideal solution.

***Function:* seconder()**

***Input:*** x mins

***Output:*** seconds

The function **seconder** converts a time duration in minutes to seconds by multiplying it by 60 (the number of seconds in a minute). It uses the **timedelta** class from the **datetime** module to represent a duration in minutes. The **total\_seconds()** method of the **timedelta** class returns the total number of seconds in the duration.

## **Issues**

**Closed:**

* Discrepancies in the dataset (ERTrack.net)
  + Missing values for some facilities
  + Server Error from the ERTrack.net for few facilities
* Need faster prediction algorithm to get quick turnaround in the app
* Current wait time was available, so we needed to predict the next wait time when the patient arrived at the facility.
* We had to look for facilities in nearby states as well in case patients lived in the borderline region.
* We had to consider the closing and unavailability times when predicting the next wait time of each facility.
* We didn’t want the patients to travel a long distance in case of emergency so we gave more weightage to the distance parameter when ranking the facilities in the TOPSIS algorithm.
* We were to use the Google Maps API to calculate the distance between two locations but since it was a paid service we switched to Geopy.

**Open:**

* The machine learning algorithm is taking some time when fetching data from the ERTrack.net server. This is causing the app to crash when run on the mobile device. The application needs some optimization in order to be able to run on a mobile device. We had tried to incorporate the asyncio library to asynchronously perform the fetching of wait times for each facility; however, this did not seem to improve the performance of the code. We need to look for another alternative to fix this issue.
* APK file crashing in mobile device. This needs to be debugged and since we lack the skill set required for the same we could not go further with this.

# **Results**

We have formulated a machine learning logic to fetch the nearest and lowest wait time facilities based on the users location and the three key factors; hospital wait time, distance, and real-time traffic while predicting the top 3 facilities to the user.

We have subsumed this logic into a mobile application made using the Kivy library in Python code which acts as a frontend to the user. This Kivy application is further converted to an APK file using the Buildozer library which can be run/installed on a mobile device.

The project has been a great learning experience. We have learnt:

* About the Incorta Platform and got certified
* How to develop a mobile application using Kivy and implement it using Buildozer
* How to deal with changing scopes
* You cannot go through with all the ideas
* Sometimes, the simplest method turns out to be the best method
* Always keep a buffer time period to accommodate any changes in the project timeline

# **Appendix**

In the solution proposed, KNN algorithm was suggested for finding the top 3 hospitals given the waiting times and distance.

Suggested Change: Upon our research, we found there is no need for KNN algorithm as we get the real time wait time and distance from the ERTrack.net and Geopy, respectively, which can be used directly to list out the top 3 facilities.

# **Sources**

1. D. H. Yuda, S. Riyanarto "Determine The Best Option for Nearest Medical Services Using Google Maps API, Haversine and TOPSIS Algorithm" https://ieeexplore.ieee.org/document/8350709
2. OSRM API <http://project-osrm.org/docs/v5.7.0/api/#general-options>
3. Geopy API <https://geopy.readthedocs.io/en/stable/>
4. Kivy https://kivy.org/
5. Buildozer <https://buildozer.readthedocs.io/en/latest/quickstart.html#init-and-build-for-android>
6. KNN Imputer <https://scikit-learn.org/stable/modules/generated/sklearn.impute.KNNImputer.html>