Traffic Flow Prediction System

Benjamin Taylor (104454504)
Ruchika De Zoysa (104316170)
Ashley De Vries (103988905)

COS30018 – Intelligent Systems
Option A - (Topic 2) - Extension option 3

27/10/2024

Introduction

This project focuses on developing a traffic flow prediction system, building upon an existing framework provided. The system leverages traffic data from various SCATS (Sydney Coordinated Adaptive Traffic System) intersections across the Melbourne region to predict traffic flow at any given time. It incorporates a variety of data processing techniques and machine learning models, including modifications to GRU, LSTM and SAE's while introducing a new, project-specific model called CNN.

Additionally, the project integrates Dijkstra's algorithm, traditionally used for finding the shortest path, to determine the optimal travel route between intersections. A fully functional graphical user interface (GUI) allows users to input their starting and destination locations, providing them with the best route based on real-time traffic flow predictions.

Overall System Architecture

Overall Architecture

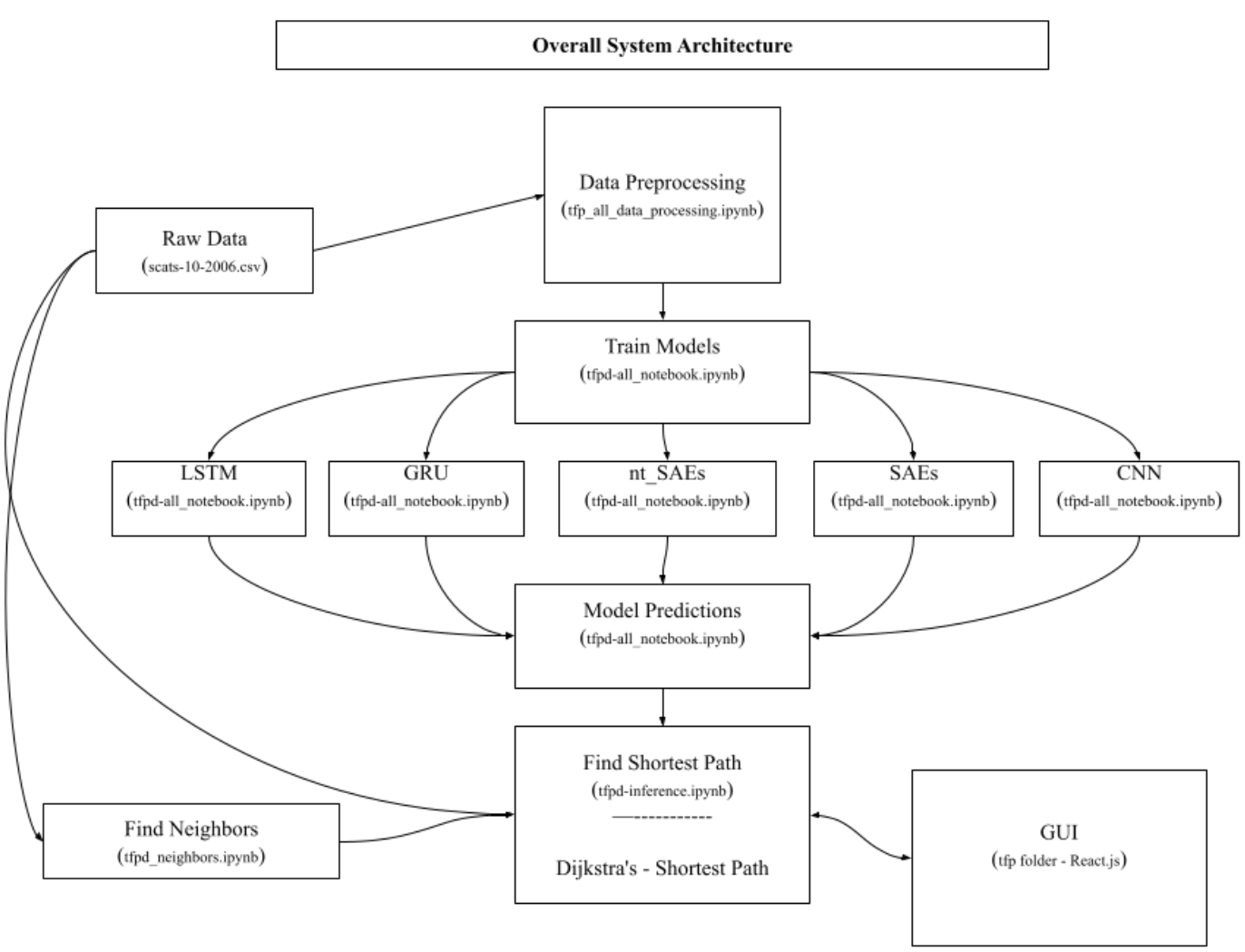

Figure 1. Overall System Architecture. Created Using Google Draw

Summary of System Architecture:

1. The Raw Data is loaded into Preprocessing.
2. Data Preprocessing Splits the data into training and testing sets for each scat.
3. All prediction models are defined using the training and testing sets.
4. The models (LSTM, GRU, SAES, nt_SAEs and CNN) are trained and save their predictions.
5. Shortest path from SCAT_a to SCAT_b is found using Dijkstra's algorithm using the model predictions and scat neighbors
6. GUI allows users to input their starting and destination SCAT and displays the optimal route for the user

Data Preprocessing

All data processing takes part in tfp_all_data_processing.ipynb. Firstly, the program loads in the raw dataset (scats-10-2006.csv) that contains the SCAT number, Location, Latitude, Longitude, and the amount of traffic through that intersection every 15 minutes, saved as V00 for the initial time all the way to V95. Moreover, all unused columns are removed, such as the way the data was collected (NB_TYPE_SURVEY) or the internal storage stat.
Following, the time e.g. V00 or V01 is converted to a useable time variable, the total amount of unique SCATs (40 in this case) is found and the data known as 'Vflow' which is the amount of traffic per SCAT at a given time is split into a training (70%) and a testing (30%) set and saved in the intersection/train or intersection/test folder respectively. The data is split into training and testing sets to allow the future models to learn from historical data, thanks to the training set and test based on unseen data, the testing set. The 70-30 split allows for a strict balance between training and testing, avoiding any potential over or underfitting of the future models. Finally, a lag which specifies that amount of previous data points to predict the next value in the future models is initialized. In this case a lag value of 12 is selected for each SCAT site.

Prediction Models

Three prediction models were selected and used to predict the flow of traffic for a given SCAT site. Two of which were iterations from the provided GIT (nt_saes, GRU and LSTM), while a newly introduced model called SAEs and CNN was also implemented. The goal of the predictions models is to predict the VFlow for the next 15 minutes per given SCAT using the previous six values.

Firstly, the test and training datasets are initialized, randomly shuffling the training data set and assigning a MinMaxScaler to ensure all the data sits between 0 and 1. Afterwards, a train_models function is defined which is the framework for all of the models. It takes in the selected model (defined later), the training and testing data sets with the lags, the SCAT number and config for the parameter for training. From there, the model is compiled using the Mean Squared Error (MSE) as the loss function, RMSprop optimizer to update the models weights based on the gradients during back propagation and finally Mean Absolute Percentage Error (MAPE) to measure the accuracy of the model. Onwards, the history of the model is stated with the training and testing sets, the batch size, the number of epochs (number of full pases over the training set), the validation split (the amount of data set aside to monitor the models performance during training, this is constantly set at 5%) and the verbose logs which is equal to 0. Finally, a models folder is created if it doesn't already exist that saves the model output and the loss per SCAT.

After the models function is created, a do_train function is established that orchestrates the training process for all of the models. Using the model name as an input, the lag and config which consists of the epochs and batch size is defined. From there a function loops through the training and testing folders to find the desired SCAT number, extracting the testing and training file for that given SCAT. Then the model architecture is defined and can be trained using the selected files. The model architectures are listed below by the number of neurons per layer:

1. LSTM ([12, 64, 64, 1])
2. GRU ([12, 64, 64, 1])
3. nt_SAE's ([12, 400, 400, 400, 1])
4. SAEs ([12, 6, 3, 6, 12, 1])
5. CNN (lag, 1)

LSTM

Once the models are defined. They can officially be built. Long Short-Term Memory (LSTM) was selected and altered from the GIT as a neural network architecture, aimed at handling sequential data, which is perfect for the Vflow. In this case, the LSTM is split into 5 layers as seen in figure 2. The input layer, which receives the time series data and a feature per set, the traffic flow. The the first layer known as the first layer neurons processes and stores the input sequence and learns its dependencies (patterns overtime), from there the patterns are outputted to the second layer, and in this case using a return sequences allows the output of layer one to be an entire sequence of predictions rather than just a single output. The second layer takes the input sequence from the first layer and further stores and processes the sequence to find more patterns and summarizes the sequence into a single prediction. Onwards, layer two outputs the hidden state which is the summary of the entire sequence for a prediction. Layer two doesn't use a return sequence like layer one, meaning the output for layer two will be the final output of the sequence rather than a prediction for each step of the sequence. Once the prediction is found using layer two, a dropout layer is used as a regularization technique to prevent any overfitting by "dropping" random neurons during training. This forces the model to learn more robust patterns and prevents any over relying on specific neurons or memorization. Finally, the last layer, known as the Dense layer is a fully connected layer that produces the final output of the model. This layer takes the output from the LSTM layers and connects each neuron to every neuron in the previous layer and an activation function is used, in this case a sigmoid function that outputs values between 0 and 1.

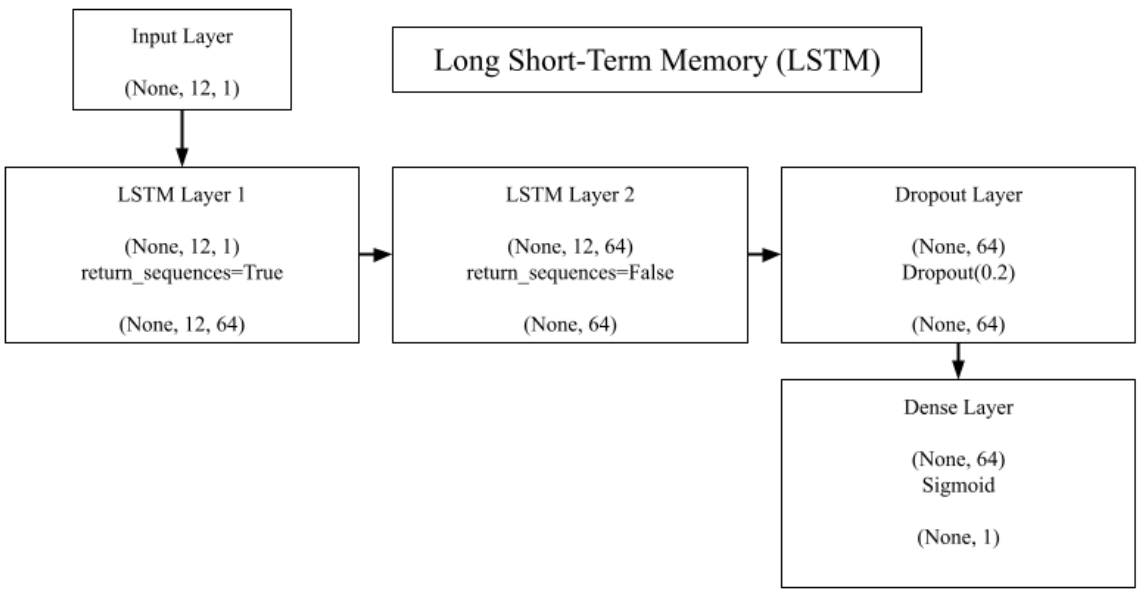

Figure 2. LSTM Diagram. Created Using Google Draw.

GRU

Similarly to LSTM, a Gated Recurrent Unit (GRU) model is created. GRU is also a neural network but with a much more simplified structure as it has fewer parameters/gates thus being computationally less intensive. Firstly, the GRU input layer receives the sequential traffic flow data. Secondly, the first GRU layer processes the input and learns its patterns/dependencies overtime. Although similar to LSTM, GRU combines gates such as the forget and input gates into a single gate called an 'update' gate allowing the flow of information with fewer parameters. While LSTM would keep the gates separate. Moreover, the first GRU layer also allows for return sequences as the output/input of the second layer. The second layer takes the sequential inputs, and merges them into a single value for prediction (not allowing return sequences), this is the same as layer two for LSTM. Afterwards, the dropout layer is implemented to prevent overfitting and allow more regularization through the random dropping of neurons (the same as the dropout layer for LSTM). Lastly, the dense layer processes the output from layer two and applies a sigmoid activation function, once again the same as the Dense layer for LSTM.

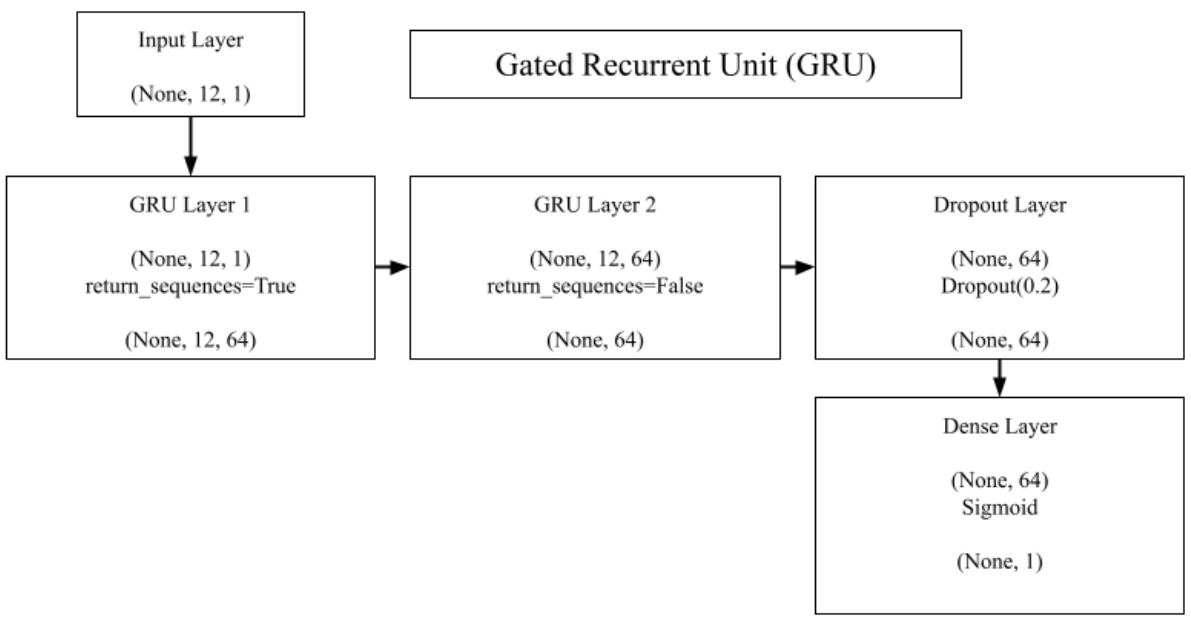

Figure 3. GRU Diagram. Created Using Google Draw.

NT_SAES/SAES

A Stacked Autoencoder (SAE) was created, in this case a nonlinear transformation, that excels in dimensional reduction and feature learning by compressing and learning essential features of the data. The SAE architecture which is a "black box" as what is being done in the hidden layers is internal workings is divided into five key parts as seen in figure 4.

1. First Autoencoder layer (SAE1): This is the input layer, it takes the raw data and passes it through a hidden layer. This hidden layer transforms the input using various neurons and followed by an activation function in this case a sigmoid one. SAE1 learns the initial patterns of the input, compresses it into a lower-dimensional form and passes it to the second autoencoder layer.
2. Second Autoencoder layer (SAE2): The second autoencoder takes the output of SAE1 and further compresses the data focusing on higher level features. SAE2's hidden layer extracts more abstract patterns from the data and refines the learned features, allowing the system to model more complex patterns.
3. Third Autoencoder layer (SAE3): Similarly to the previous layers, SAE3, takes the output from SAE2, compresses the already-learned features further. Allowing for even more in depth representations of the data. Ultimately, increasing the models accuracy.
4. Final Stacked layer: The fourth layer combines the results from the three individual previous autoencoders and uses three hidden layers (corresponding to each autoencoder) and applies non-linear transformations (sigmoid activations) at each level. Afterwards, a dropout layer is used similarly to what is done in LSTM and GRU, to generalize the results and avoid overfitting.
5. Dense Layer: Finally, the dense layer which acts as the output layer produces the final predictions (similarly to what is done in LSTM and GRU) through a sigmoid activation function, providing output values between 0 and 1.

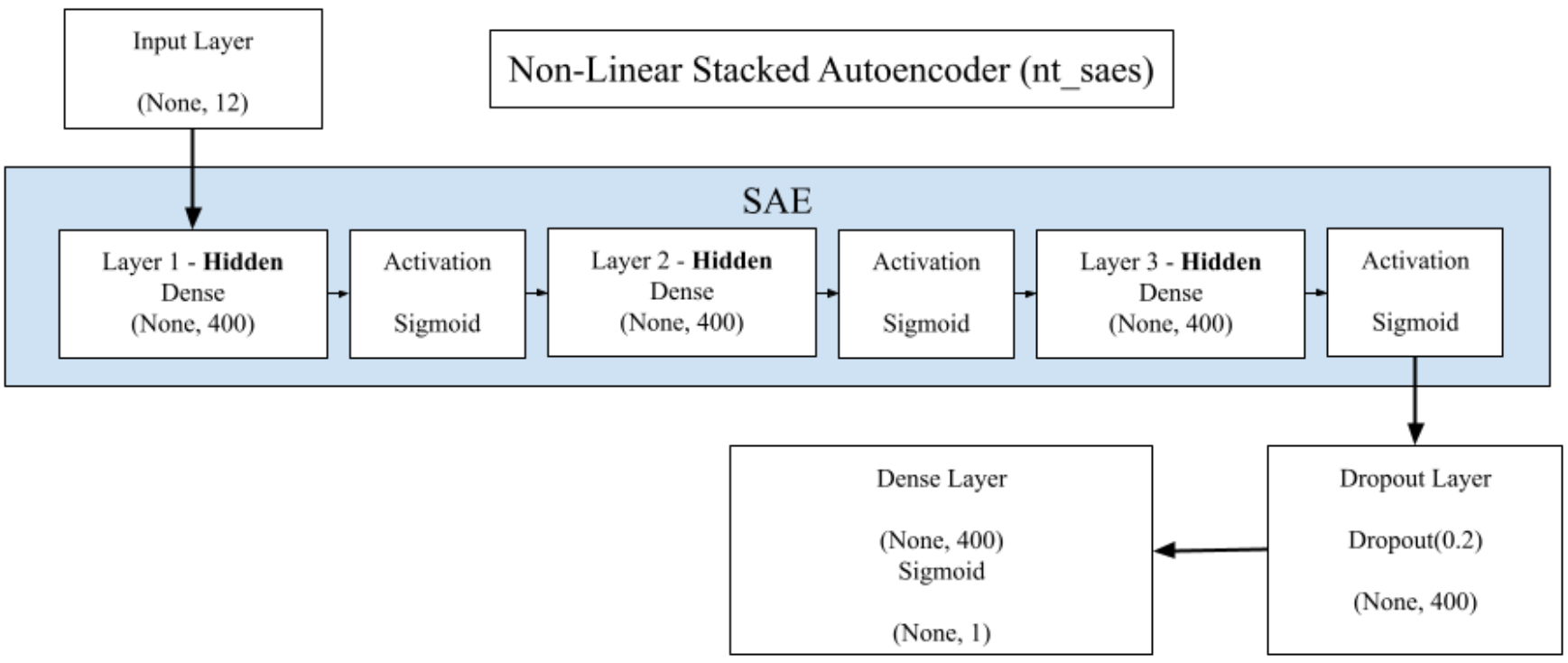

Figure 4. Nt_saes Diagram. Created Using Google Draw.

CNN

The CNN model is utilized to predict traffic flow by learning from past patterns in traffic data. CNNs are traditionally used for image processing but can also effectively handle time-series data, especially when identifying local dependencies and patterns across sequential data. In this project, CNN is applied to process traffic flow values (VFlow) for a specific intersection, leveraging a series of convolutional layers to extract relevant features from historical traffic data.

The CNN model structure is configured to take a set number of past traffic flow values (lag) as input and predict the traffic flow for the next time step. The model's layers include:

- Convolutional Layer: This layer applies filters across the input time series data to capture local patterns, allowing the model to identify recurring trends in traffic flow.
- Pooling Layer: Reduces the dimensionality of the data, focusing on the most prominent features while reducing computational load.
- Flatten and Dense Layers: These layers process and combine the extracted features, resulting in a single output that predicts the future traffic flow.

The CNN model is trained using Mean Squared Error (MSE) as the loss function and is optimized to minimize prediction errors by adjusting its weights through backpropagation. By incorporating CNN into the prediction models, the system aims to capture both local and sequential patterns within the traffic flow data, providing accurate short-term predictions for a given intersection.

Routing

The routing system aims to build and find the optimal path between two neighbors. This is conducted in 4 steps.

1. finding Estimated Vehicle Speeds using VFlow: Using the VFlow (traffic flow rate) for an intersection, the average speed of vehicles traveling through that intersection can be computed. This is done using a relationship between the VFlow, a predefined inflection point (the critical flow rate), and the assumed road capacity. When traffic flow is high, vehicle speeds decrease, and when traffic flow is low, vehicle speeds increase. However, vehicle speeds are capped at the road's speed limit of 60 km/h.
2. Calculating Distance Between Intersections: The geographical distance between intersections is computed using the longitude and latitude data for each SCATS site. This distance is used to estimate travel times when combined with vehicle speed data.
3. Average Lag: Given the current time and day of the week, the model identifies previous traffic flow patterns (lags) for similar days and times. These lag values are used as input to predict the current traffic flow and estimate vehicle speeds at different times of the day, ensuring that the predictions account for variations based on the time and day of the week.
4. Dijkstra's Algorithm to Map Routes: Once the lags and vehicle speeds are estimated, Dijkstra's algorithm is applied to find the optimal route between intersections. The algorithm uses an external document called 'neighbors', which provides the connections between SCATS sites (e.g., how each SCATS is connected to its neighboring intersections). Dijkstra's algorithm then calculates the travel time between one SCATS and its neighboring SCATS using the vehicle flow rate, average speed, and time of day as weights. The algorithm will compute the total path cost using the weights, and direct to/through the SCAT with the lowest path cost (as seen in figure 5). If the route crosses multiple SCATS intersections, the algorithm sums up the travel times for each segment until it reaches the destination SCATS. The result is the shortest path between the starting and destination SCATS, along with the estimated total travel time.

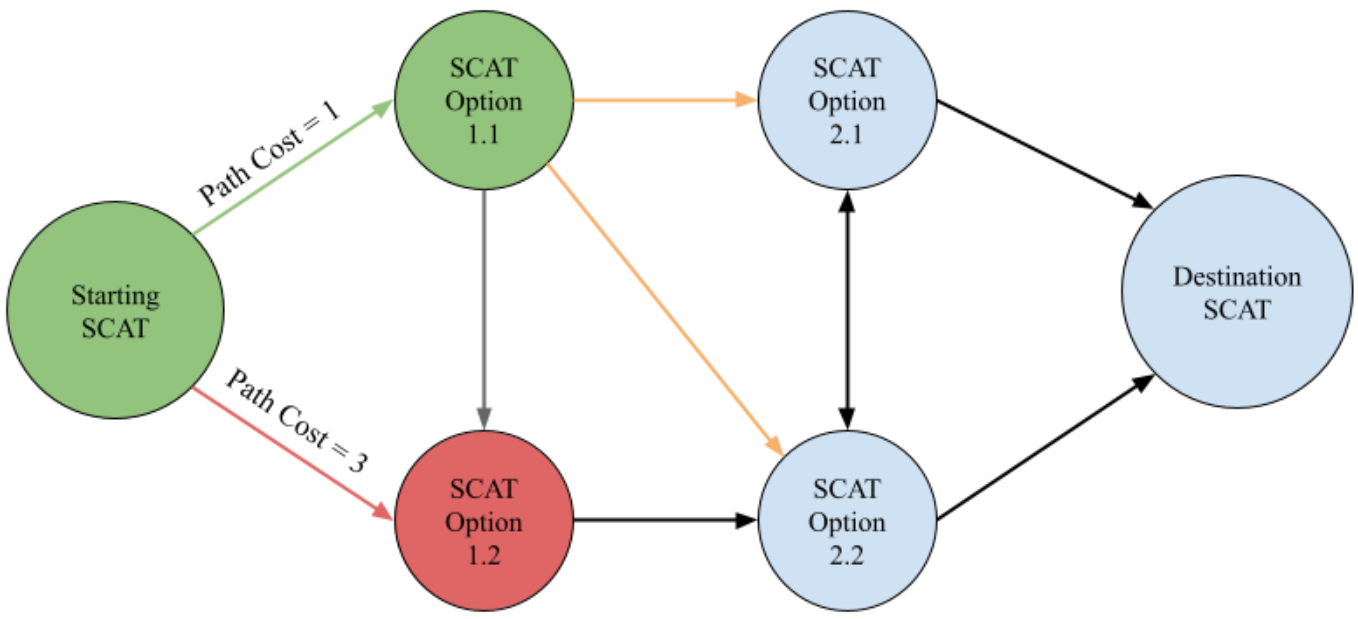

Figure 5. Dijkstra's algorithm. Created Using Google Draw.

As illustrated above in Figure 5. Diskstra’s algorithm maps the route from starting to destination SCAT. Firstly, a SCAT is inputted as the starting SCAT. From there, there are two routes to choose from, either to SCAT 1.1 or SCAT 1.2. However, using the weights defined above, the path cost to SCAT 1.1 is less than the cost to SCAT 1.2. Therefore, the route will go this way. From SCAT 1.1, it can either go to SCAT 2.1 or SCAT 2.2, the reason it can’t go back to SCAT 1.2 is because that node has already been visited. The algorithm will continue to compute the path costs and selecting options until the Destination SCAT is reached.

GUI

The GUI implemented uses React.js and leaflet. Firstly all variables are defined and the intersections are mapped onto an image of Melbourne using each intersection's longitude and latitude. From there, the GUI allows for the user to select a starting SCAT and a destination SCAT and makes a call back to the Routing Program. The routing program then inputs the starting scat along with all other weights such as vehicle flow rate, average speed, and time of day and computes the route using Diskstra's algorithm (as mentioned above) to get from the selected starting stat to the final destination SCAT. Once the route has been found, it is sent back to the React.js program and the route is mapped, connecting each intersection traversed. And the GUI also has a button that sends a request to the backend python program to display alternate routes along with their route and path cost.If alternate paths are identical to the optimal path they will not be displayed. Illustrated below in Figure 6, is the route from SCAT 4321 to SCAT 3812.

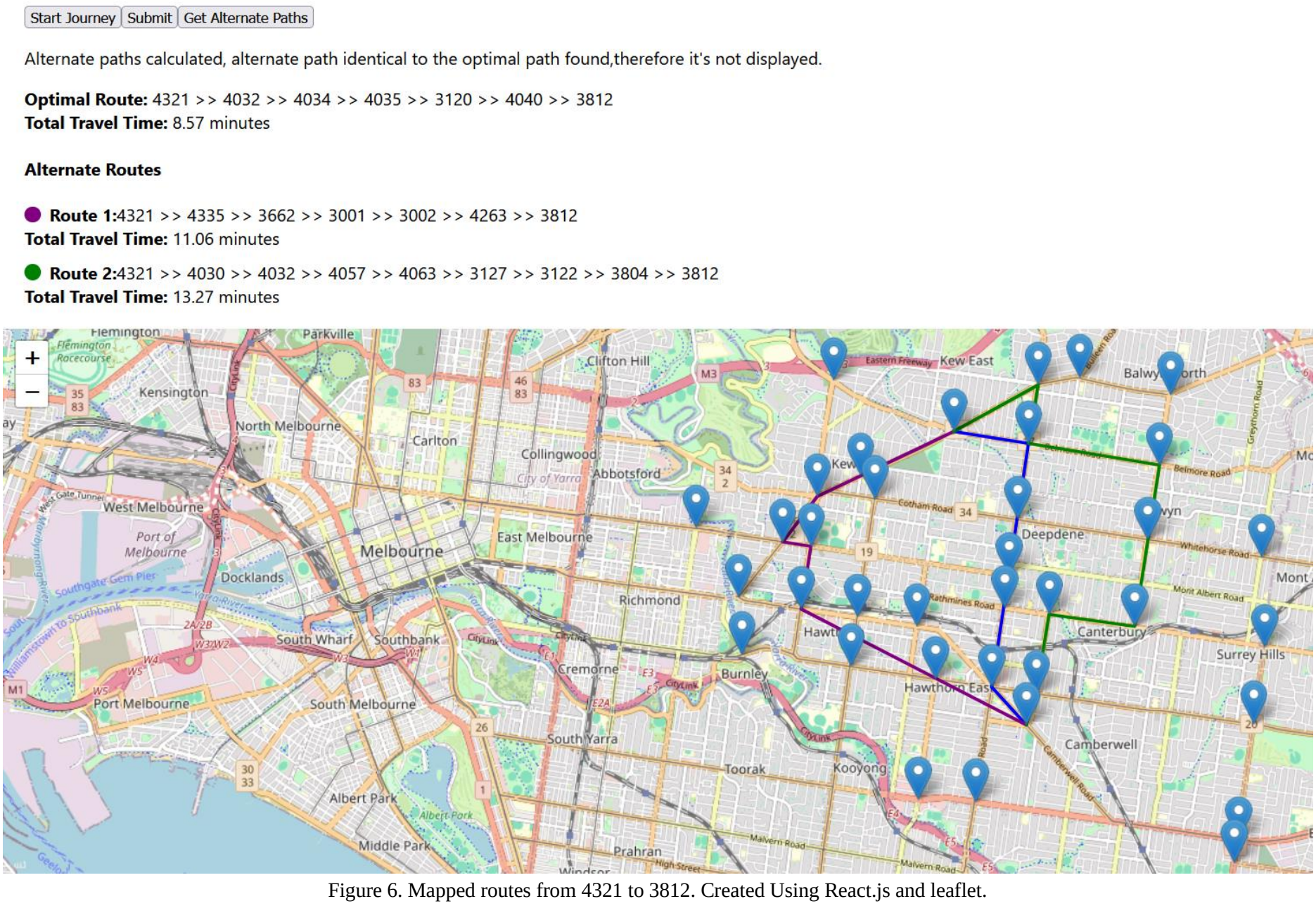

Figure 6. Mapped routes from 4321 to 3812. Created Using React.js and leaflet.

Implemented interaction protocols

Data Flow Between Components

There is a continuous and seamless communication between all layers of the system via internal function calls and/or intermediate representation like dataframes and model predictions. As illustrated in the overall system architecture, each section builds off one another. For example, in order to train the models, the data must be processed in preprocessing. Or, to find the shortest path, the system must make an internal function call to all models outputs to use the models along with an internal file call towards the neighbors csv file to compute the fastest route.

Front-end, Back-end Communication

The GUI (front-end) and the backend (models, routing etc.) connect seamlessly via RESTful API calls. When the user inputs a starting and destination SCAT number within the front-end, the GUI sends a POST request to the end point within the backend. This request contains the starting SCAT (selected by the user), the destination SCAT (selected by the user) and the timestamp. The fastAPI backend processes the post request and uses the routing algorithm to compute the optimal path, in turn returning the results back to the frontend, facilitated via HTTP requests and JSON data to be displayed back to the user.

Caching for Performance

To improve the performance of all models, specifically the fastest route one, the models are continuously cached. This ultimately avoids the models and scalers having to reload on every request. For example, if the user wants to go from SCAT_a to SCAT_b. The route will be found and then stored in the cache. Therefore, if the users requests this route again, it will be saved in the cache and won't require the models to reload. Also, in the cache each of the intersections for the route will be saved, so for an alternative route, if it traverses an intersection that is already saved in the cache, it also won't be required to load that model/SCAT again. In summary, the cache helps speed up repeated requests by reducing the overhead of reloading the machine learning models each time, allowing a more comfortable user experience.

Implemented search/optimization techniques

Prediction Models Optimization

All of the traffic flow predictions models (LSTM, GRU, NT_SAE, SAE and CNN) are optimized via standard machine learning techniques:

MinMax Scaling

Data is preprocessed using MinMaxScaler to normalize the traffic flow (Vflow) values between 0 and 1. This ensures that all features contribute equally to the model's learning process. Ultimately preventing features with larger numeric ranges from dominating those with smaller ranges, which is particularly important in models like LSTM, GRU, and NT_SAE, which are sensitive to feature magnitudes.

Early Stopping and Dropout

Dropout layers are used in both the neural network models (LSTM and GRU) to prevent any overfitting of the data by randomly dropping units during training (this is explained in both dropout layers). While early stopping is also used to halt the training process if/when no improvement is seen in the validation performance after several epochs. This helps speed up the training process and helps save computing resources as the model is already optimized to its potential.

Batch Size and Epochs

The training process is optimized using a carefully selected batch size of 256 and 500 epochs, allowing the models to learn efficiently without overfitting or underfitting. Having a larger batch size results in faster training since more data could be processed in parallel, however this would require more computing resources (VRAM in particular, for GPU based training). The batch size of 256 was chosen as it was the highest our training GPU could handle with 24GB of VRAM. Onwards, more epochs would theoretically allow the model to continue improving as the data is seen more times (this is only if the optimal solution isn't already found and early stopping as mentioned above isn't ignited), the models may also find more complex patterns and overall increase the models performance. However, more epochs would also mean more computing resources and for an extended period of time. If there were fewer epochs, the model would train faster, requiring less resources and reduces the risk of overfitting, but also risks poor model performance as a result of underfitting as the model hasn't learnt enough from the data. Hence, a common epoch count of 500 was utilized.

Routing Optimization

The optimal path between SCATS is found using Dijkstra's algorithm, which minimizes the total travel time between intersections based on real-time traffic flow predictions. Several optimization techniques were used:

Graph-based Search

Dijkstra's algorithm was applied to the graph of SCATS intersections, where each node represents an intersection and each edge represents the travel time between them. Where the edge costs are computed by weights such as vehicle flow rate, average speed, and time of day (as addressed in detail in Routing.)

Heuristic Cost Calculation

The travel time between two intersections is found using a combination of real-time traffic flow predictions, speed limits, and geographical distances, found using the provided latitude and longitude and calculated via Haversine formula. This algorithm evaluates all routes and selects the path with the lowest path cost in terms of travel time.

Caching Predictions

As briefly addressed in caching. The predicted traffic flows and their corresponding models are cached, improving the response time of the GUI routing system by reducing the need for redundant model predictions.

Other Calculations

Average Lag Calculation

To account for the time and day of the week used, previous traffic data (known as lags) is average to predict the current traffic conditions. This Optimization technique ensures that the selected prediction model alters its output given the historical traffic patterns, improving accuracy regardless of the time of day, for example during rush hours. This lag calculation is explored in more detail within Overall System Architecture.

Dynamic Vehicle Speed Calculations

The program dynamically finds the vehicle speed via the VFlow. This enables the model to accurately estimate the travel time between intersections, to then be used in Dijkstra's algorithm. The speed is capped at 60 km/h but can vary depending on the current traffic flow. This is further addressed within the Routing section of Overall System Architecture.

Examples/Scenarios of system

Longest Possible Route - Shortest Possible Route

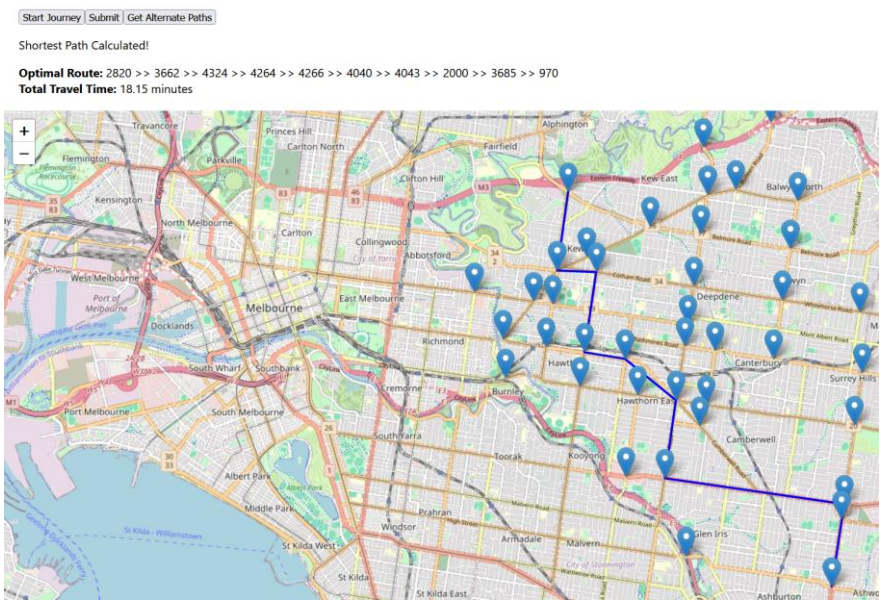

Figure 7. Longest Possible route from 2820 to 970. Created Using React.js.

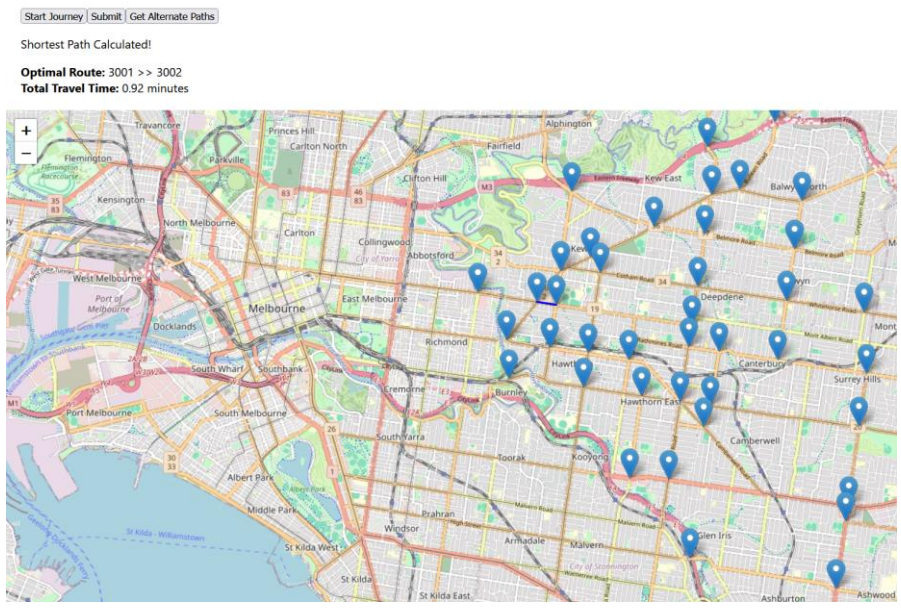

Figure 8. Shortest Possible route from 3001 to 3002. Created Using React.js.

The shortest and longest possible routes illustrate:

1. How the system deals with going through multiple intersections - with multiple possible routes
2. How the system deals with going through no intersections (start to destination) - with only a single route

Peak Traffic Hours

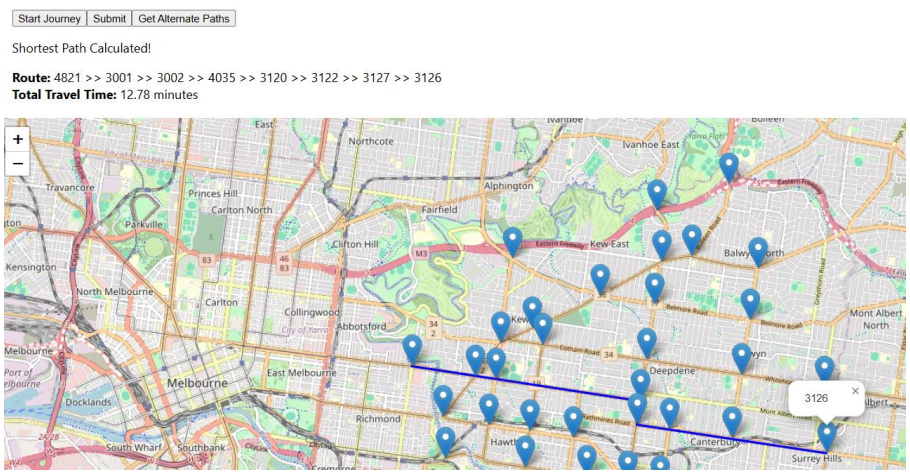

Figure 9. Not Peak Hours route from 4821 to 3126. Created Using React.js.

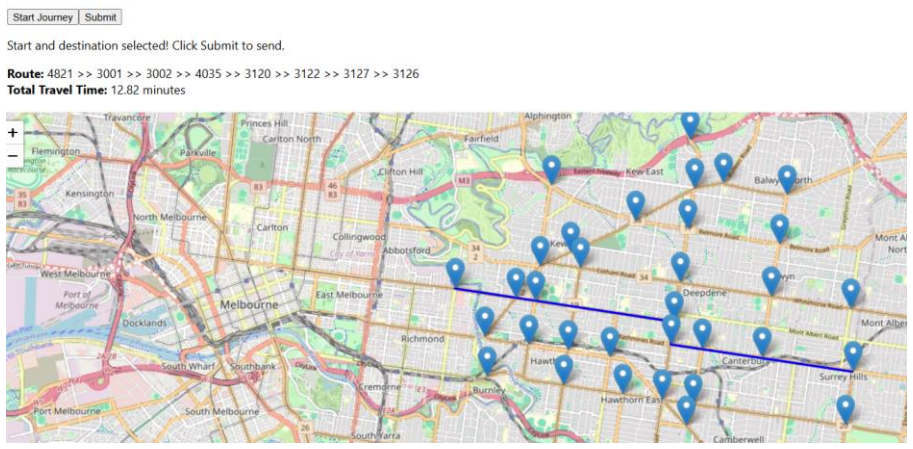

Figure 10. Peak Hours route from 4821 to 3126. Created Using React.js.

Testing the route on and off peak hours illustrates how the path costs/the time it takes to go from one intersection to another provided the time changes. During peak hours, on Saturday at 5:50pm, it takes almost 13 minutes to go from intersection 4821 to intersection 3126. While not during peak hours it also basically takes 13 minutes but is ever so slightly faster (0.04 seconds faster). This may mean that this selected route is not directly impacted during peak hours. The system takes the current operating system time as the time to start the journey. The system was tested through the week to simulate the traffic hours.

Changing Model Variant

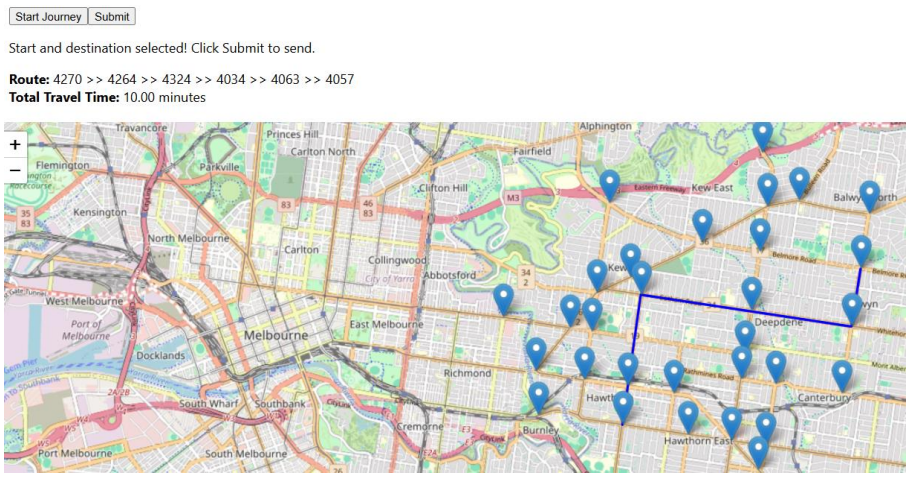

Figure 11. SAE model from SCAT 4270 to 4057. Created Using React.js.

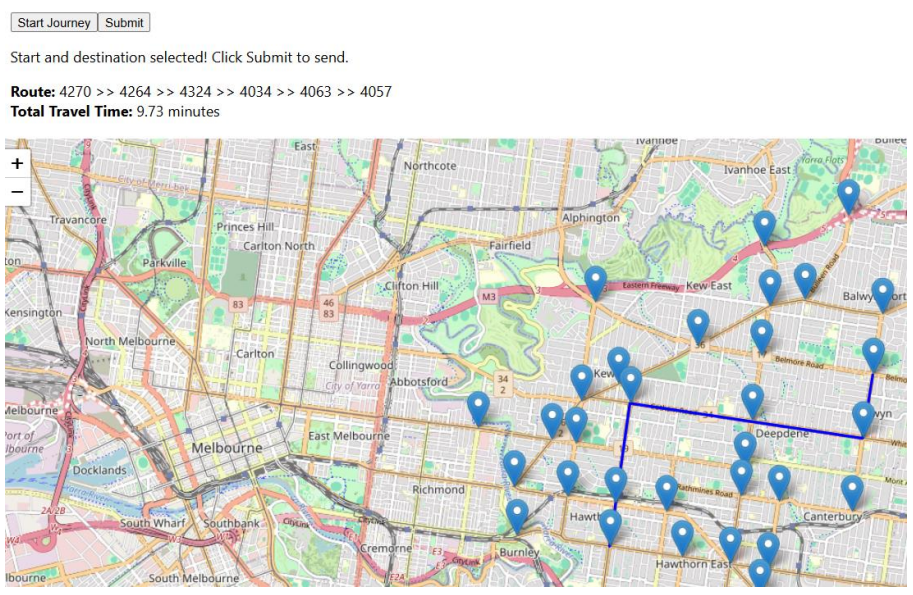

Figure 12. LSTM model from SCAT 4270 to 4057. Created Using React.js.

By changing the model variant to predict the route from the current used Model (SAE) to an alternative model. In this case LSTM. Displays that the route will remain the same (as expected) however the total travel time will slightly change. A difference of only 0.23 seconds is miniscule and doesn't emphasize which

model's total travel time is more accurate. With that said, this highlights if the model variant is changed, the predictions for time it takes to get from one SCAT to another SCAT doesn't change enough to be impactful for the user and either SAE or LSTM could be used to find the route and the path costs.

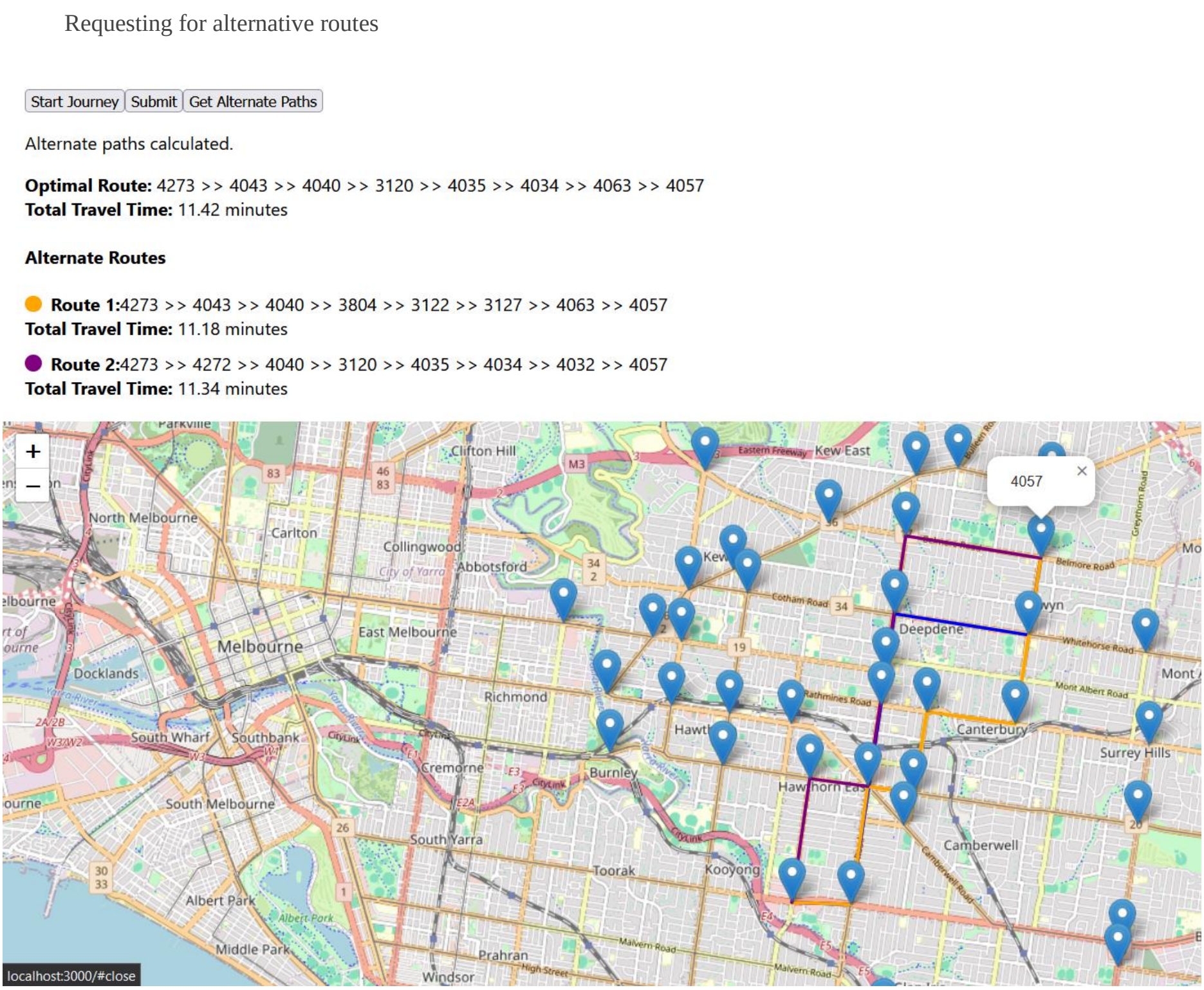

Figure 13.Alternative routes from SCAT 4273 to 4057. Created Using React.js

By requesting for alternative routes, the system will use A* algorithm to find the next best routes using the machine learning algorithm. This feature aims to mimic the functionality google maps offer by displaying alternative routes to users.

Critical Analysis of Implementation

Data Preprocessing Robustness

Strengths: Comprehensive Preprocessing process, ensuring each SCAT has usable and clean data. Training and testing data sets are appropriately split 70 - 30 and appropriate lag values and MinMaxScaling are implemented for reliable model predictions. All Scat locations and neighbors found but with some manual implementation required.

Limitations: Lots of the data have inconsistent data patterns, are missing values (Long and Lat for SCAT 4266 AUBURN_RD N of BURWOOD_RD) or simply incorrect values (all scats Long and Lats were not directly on the actual locations). Although preprocessing did a job addressing these gaps, further outlier detection or anomaly handling could've been implemented to reduce the manual changes that had to be made and enhance the models accuracy and reliability.

Model Performance and Scalability

Strengths: The LSTM, GRU, CNN, and NT_SAE models exhibit strong capabilities in managing sequential traffic data, demonstrating high accuracy in predicting flow rates by leveraging previous traffic patterns. This is evidenced by consistently high Explained Variance Scores (EVS) across various SCATS sites, generally above 0.9, which indicates that these models capture a significant proportion of variance in the traffic data. Each model effectively balances accuracy and computational resource efficiency, with configurations such as batch sizes, epochs, and dropout layers tailored to prevent overfitting. This

optimized setup has helped achieve Mean Squared Error (MSE) values that remain relatively stable across sites, with low computational overhead, making these models suitable for environments where processing power is a consideration.

Limitations: Despite their performance, these models have limitations that could be addressed to improve accuracy and consistency. Increasing model complexity, such as by adding more layers or nodes and using larger batch sizes or more training epochs, could enhance predictive capabilities, allowing for more nuanced learning and potentially reducing Mean Absolute Percentage Error (MAPE) across sites. Currently, MAPE values range between 15% and 40%, revealing variability that suggests the models' accuracy could be more consistent, particularly for certain SCATS numbers with unique traffic patterns. This fluctuation indicates that some models may not generalize as effectively across all SCATS sites. However, such model enhancements are currently limited by computing resources, constraining their potential for deeper learning and more finely tuned predictions across diverse traffic scenarios.

Real-time Prediction Limitations

Strengths: The integration of MinMax scaling and early stopping prevents overfitting and optimizes model performance. Cached model predictions improve response times, making real-time traffic predictions more feasible for the GUI.

Limitations: Traffic patterns can be highly unpredictable, and while the models account for historical patterns, they may not respond well to abrupt changes (e.g., accidents, unexpected road closures). Further work could integrate live traffic feeds or adaptive learning methods to address real-time unpredictability. Moreover, as the data is from 2006, more up to date data would allow for more accurate real-time results.

Routing Algorithm Efficiency

Strengths: Dijkstra's algorithm is a proven solution for shortest pathfinding and is well-suited for the relatively fixed structure of SCATS intersections, providing consistent results based on time-of-day data. A* algorithm is well known for solving navigation problems since it is capable of taking heuristics into account and coming up with the shortest path with great efficiency. A* has been implemented in order to find the alternate paths since Dijkstra's is able to provide only one optimal path while A* is capable of displaying multiple paths using the unvisited nodes.

Limitations: Dijkstra's algorithm could be slow with many nodes if the number of intersections increases significantly, particularly during high-frequency queries. A* have been used to calculate alternative routes since Dijkstra's is only able to output the optimal route. However, when using A* Sometimes the alternate paths are identical to the optimal path. In these situations, the identical path is removed, and sometimes this leads to having no alternate paths.

GUI and User Experience

Strengths: The React.js-based GUI offers a clear and interactive way for users to input routes and view travel times. Real-time feedback (e.g., showing route paths and total travel time) makes the interface intuitive and helpful for users. The front end being developed as a web app helps isolate the front end from the backend, minimizing the room for error and having a web-based component allows the system to be scalable than a local application, The React based front end can be scaled to use various APIs to integrate or draw data from external sources.

Limitations: The React.js is relatively slow as a result of the machines computing power and heavy backend. Although this has been sped up using caching, the response time of the GUI, may push users away from using it. React.js requires routing to connect to the backend, the routing and servers add an additional layer of complexity to the system. A routing machine or a paid API such as google maps API could have been used to ensure the routes always follow along a road.

Summary/Conclusion

In conclusion, the chosen system along with its adapted architecture is able to accurately identify the route and the time it takes to go from one user selected intersection to another user selected intersection. The program is able to accurately handle routes that have to cross multiple intersections or none, as well as computing the time it takes provided the time of day and day of the week it is. The models (LSTM, GRU, and NT_SAE) selected are appropriate in predicting the traffic flows (VFlow), given the historic data through the use of testing and training data sets, scalers and lag values. Moreover, all models appropriately balance precision with computing efficiency via epochs and batch sizes. The GUI user experience is comfortable to use, however is slow, but has been attempted to counter through caching. To extend the project further, more up-to-date data could've been implemented, more SCATs could've been used or an even more friendly user GUI could be constructed. But with that said, the system created does accurately provide the user with a Traffic Flow Prediction System from a selected scat to another, backed by machine learning models.

How to run the program

The following instructions are for the windows platform. Please refer to the demo video.

How to run the GUI and the main program:

1. Unzip the file
2. Read the readme file
3. Install the requirements (pip install -r requirements.txt **and** npm install leaflet react react-dom react-leaflet react-scripts)
4. All data should be processed, and models should be trained
5. Once in TRAFFIC-FLOW-PREDICTION folder: open a terminal/cmd
6. Run the backend - tfpd.py (python tfpd.py) - this is a web server; you don't have to touch once running
7. Enter tfpapp folder in cmd/terminal (cd tfpapp)
8. Install requirements for the React app using (npm install) in the app folder.
9. Run the Reactapp from tfpapp folder (npm start) - Will redirect you to your browser (http://localhost:3000)
10. Click Start Journey - Then Choose Starting and Destination SCAT - Then press submit (Can take up to 30 seconds the first time around)
11. To get alternative routes, click Get Alternative routes (Can take up to 45 seconds to 1 minute the first time around)

How to view the model training, testing and performance evaluations:

1. Unzip the file
2. Open the notebook tfpd-train_eval_notebook.ipynb
3. Optional- Run all the cells (Warning: this is resource intensive, and training will take more than 4 hours)
4. Run main: this will test the pre trained models and output a graph for comparison