TRAFFIC FLOW PREDICTION IN ROAD NETWORK

Full Methodology Guide

NOVEMBER 7, 2023

EFSTRATIOS KARKANIS
P19064

In this document, I analyze you the methodology I followed for traffic flow forecasting with the use of Strict Path Queries (SPQs). The provided code performs several data preprocessing and transformation tasks on a dataset of San Francisco Yellow Cabs' GPS records (link to dataset: http://cabspotting.org/). The goal of the code is to prepare the data for further analysis, including trajectory splitting and map matching using the Valhalla Meili API.

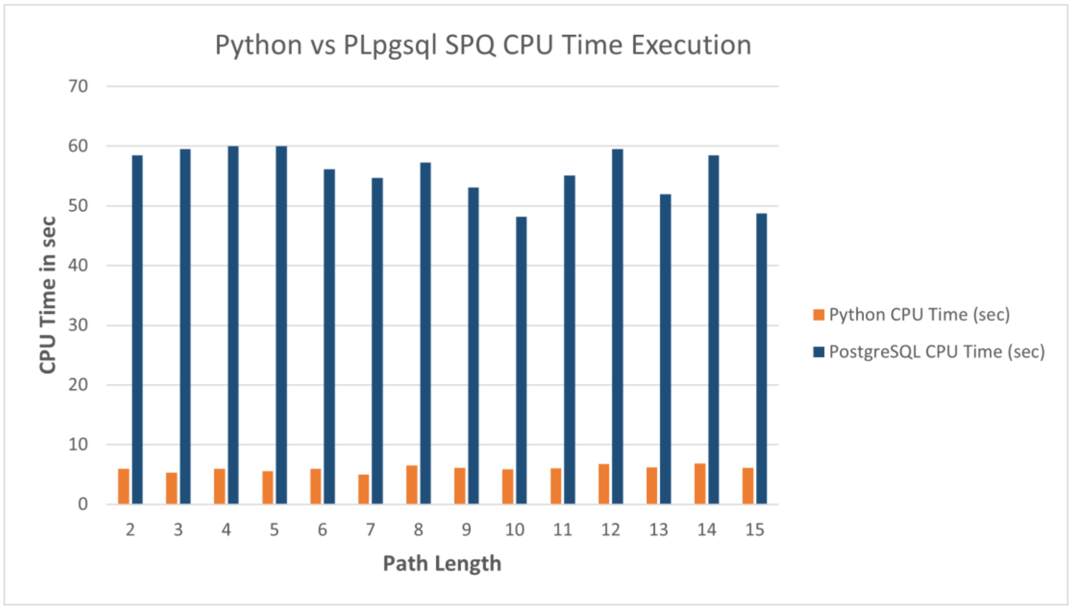
First, the code initializes the necessary variables and sets the path to the directory where the data is stored. It also creates an empty dataframe named all_data, which will store all the data from the individual files. The data used for this research comes from a project called the cabspotting project. Specifically, this dataset contains about 11,000,000 GPS data of various taxis (Yellow Cab Vehicle) in the San Francisco, California area. The whole data sampling occurred in May 2008. It is worth noting that the dataset consists of several files. Each of them includes the trajectory of one taxi.

Next, the code iterates over each file in the specified directory and reads the data into a temporary dataframe. It assigns a unique Taxi ID to each file and then concatenates the temporary dataframe with the all_data dataframe to aggregate all the data in a single dataframe. Afterwards, the code converts the "Date Time" column from Unix timestamps to pandas datetime format. It then selects a specific week of data by filtering the dataframe based on the date range. The data in the all_data dataframe is sorted based on the Taxi ID and timestamp information to ensure the trajectories are organized sequentially. The "Occupied" column, which denotes taxi occupancy, is dropped as it is not relevant to the research.

The code proceeds to split the trajectories based on the time intervals and file IDs. It introduces a new column called "Traj ID" (Traj ID is the Trajectory ID) and assigns unique IDs to each trajectory within a Taxi ID. Trajectories with consecutive GPS points and within a specified time interval are assigned the same Traj ID, while trajectories with longer time intervals or different Taxi IDs receive different Traj IDs. Trajectories containing only one OSM Way ID are removed from the dataset as they do not provide sufficient information for analysis. The resulting dataset is then printed, displaying the minimum and maximum dates of the selected week.

The code then prepares the data for map matching using the Valhalla Meili API. The latitude and longitude coordinates from the all_data dataframe are passed as input to the API for each trajectory. A new dataframe named visited_segments is created to store information about each trajectory's visited segments. It includes columns such as "Taxi ID," "Traj ID," "OSM Way ID," "Start Time," and "End Time." The code iterates over the trajectories, sends requests to the Valhalla Meili API, and receives responses containing the exact path followed by each trajectory in the form of OSM Way IDs. The response data is processed, and the visited_segments dataframe is populated with the relevant information. Trajectories containing only one OSM Way ID are again removed from the visited_segments dataframe.

The provided code, also, performs several tasks to analyze and visualize the traffic flow patterns in a dataset of San Francisco Yellow Cabs' GPS records. The code focuses on constructing a time series dataset by filling it with the count of trajectories that pass through specific paths at specific time intervals. The analysis is performed using the SPQ (Strict Path Query) function. This function has been implemented in Python and plpgSQL languages. In the graph below, we compare the two identical implementations:



In the x-axis of the graph there is information about the batches of paths. We analyse batches of paths containing from 2 to 15 edges. Each batch contains twenty paths of equal number of edges. So, for each batch, we call the SPQ function in python and plpgSQL twenty times. The y-axis represents the CPU execution time. This is the total time that the SPQ needs to run for each batch and for each of the two programming language implementations. We conclude that the Python implementation of SPQ function is much faster in terms of CPU execution time.

In order to construct the time series dataset, the following steps are implemented:

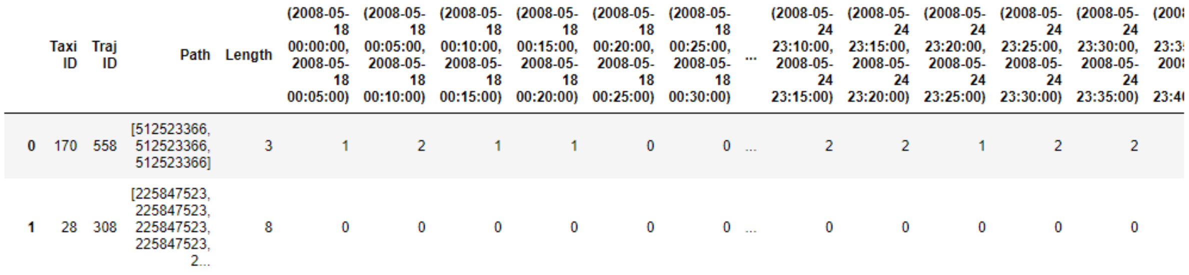
First, the code creates the time intervals for the time series dataset. It identifies the minimum and maximum timestamps in the dataset and calculates the total duration in seconds. The time intervals are defined as half-hour intervals within the selected week of data.

The code proceeds to generate random unique paths of different lengths based on the visited_ segments dataset. The paths consist of only consecutive OSM Way IDs. The paths, along with their corresponding length, Taxi ID, and Traj ID, are stored in a dataframe named "paths".

The python SPQ function is then implemented to return the count of trajectories that strictly follow a given path within a specified time interval. The function checks if the trajectory's OSM Way ID sequence matches the input path and returns the count of matching trajectories.

The time series dataframe is created, starting with an empty dataframe named "time_series". The relevant columns, including Taxi ID, Traj ID, Path, and Length, are filled based on the paths dataframe. The SPQ function is called for each time interval and path combination, and the count of matching trajectories is recorded in the corresponding cell of the time series dataframe.

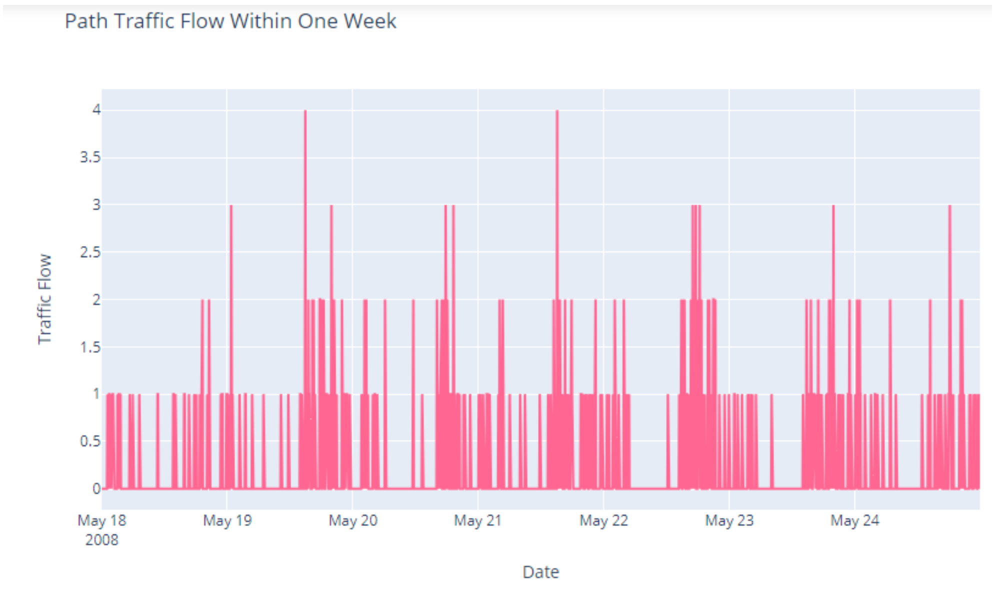
The resulting time series dataframe is saved to a text file for further analysis and visualization. It provides a comprehensive view of the traffic flow patterns for different paths over the selected week.



	Taxi ID	Traj ID	Path	Length	(2008-05-18 00:00:00, 2008-05-18 00:05:00)	(2008-05-18 00:05:00, 2008-05-18 00:10:00)	(2008-05-18 00:10:00, 2008-05-18 00:15:00)	(2008-05-18 00:15:00, 2008-05-18 00:20:00)	(2008-05-18 00:20:00, 2008-05-18 00:25:00)	(2008-05-18 00:25:00, 2008-05-18 00:30:00)	...	(2008-05-24 23:10:00, 2008-05-24 23:15:00)	(2008-05-24 23:15:00, 2008-05-24 23:20:00)	(2008-05-24 23:20:00, 2008-05-24 23:25:00)	(2008-05-24 23:25:00, 2008-05-24 23:30:00)	(2008-05-24 23:30:00, 2008-05-24 23:35:00)	(200 23:3 200 23:4
0	170	558	[512523366, 512523366, 512523366]	3	1	2	1	1	0	0	...	2	2	1	2	2	
1	28	308	[225847523, 225847523, 225847523, 225847523, 2...	8	0	0	0	0	0	0	...	0	0	0	0	0	

This is a part of the time series dataset.

After that, we perform several visualizations on the time series data. We plot the distribution of path lengths in the dataset and showcases some time series plots of the traffic flow for specific paths.



This is an example of traffic flow in a random path of the dataset within one week time interval.

After that, we continue with the second part of our work. To be more specific, the following are done:

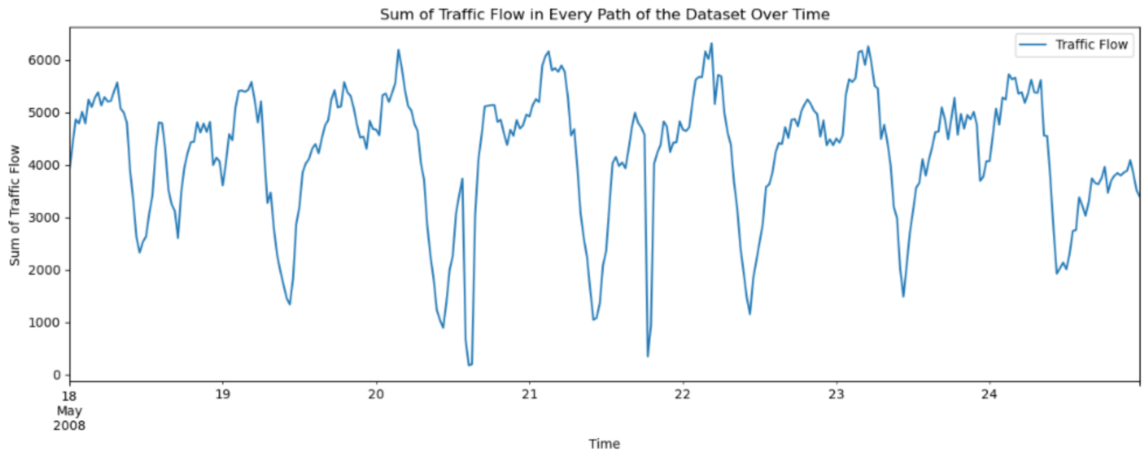
The code defines a list of column names and generates the columns for the time series dataset. The columns represent time intervals, starting from May 18, 2008, at 00:00:00 and increasing in 30-minute increments until May 24, 2008. The last timestamp is removed from the list. Next, the code assigns the new column names to the "time_series" dataframe using the generated columns.

The code then proceeds to transform the time series dataset from a wide format to a long format. It uses the "melt" function to reshape the dataframe, making the time intervals a single column named "Time Column" and the corresponding traffic flow values in a column named "Traffic Flow." The "Time Column" values are converted to pandas datetime format, and the rows are sorted by "Path" and "Time Column."

After that, we gather some weather data. These weather data were acquired using this link: https://www.visualcrossing.com/weather/weather-data-services. The code reads the weather data from a CSV file and drops unnecessary columns that either have NaN values or are not associated with traffic flow. The "conditions" column, which contains categorical values, is one-hot encoded using the "get_dummies" function. The code also updates one of the categorical values to merge similar conditions. The weather data is then filtered to include only the rows within the interval [2008-05-18, 2008-05-24].

Next, the time series dataset and the weather data are connected based on their indexes (the indexes of both dataframes are the datetime timestamps) using the "merge" function. **Important note**: the weather data contain samples at an hourly basis, while the traffic flow data we produced earlier are of a 30-minute time intervals. The "weather_data" dataset is resampled to 30-minute intervals and forward-filled to fill any missing values.

The code proceeds to visualize the sum of traffic flow in every path of the dataset over time. It groups the traffic flow values by the timestamp and calculates the sum for each time interval. The results are plotted to provide an overview of traffic flow patterns. This step is necessary, because by this way it is clear that the traffic flow in general follows a specific path.



In the plot above, the x-axis represents the time information of one week interval. The y-axis represents the sum traffic flow in every path of the dataset at each time step. We conclude that there is a seasonality pattern here: Traffic Flow burst early in the morning and late in the night of each day, while in the middle hours of the days, traffic flow drops significantly. This is a very valuable information.

In order to help the forecast procedure, code creates additional features based on the time information. It extracts the hour, day of the week, day, and minute from the "Time Column" and applies circular encoding to represent the cyclic nature of these values. The code also introduces features for the three-hour interval to describe in which interval of the day each timestamp falls into.

Then, the preprocessing and visualization part ends. The next part is the forecasting one. In this part, we try different model and compare them. Before denoting and training these models, the code applies a train-test split to the data. The train dataset includes all the data for each path until May 23, 2008, inclusive, while the test dataset contains the most recent data. Finally, the code sorts the time series data by "Path" and "Time Column" to ensure proper organization.

Accurate predictions of traffic flow can significantly improve transportation efficiency, optimize traffic control systems, and alleviate congestion. For this reason, we explore the application of different models for time series forecasting of traffic flow, aiming to develop accurate prediction models that capture the temporal patterns in traffic data.

- Models Used:

The code utilizes four distinct models: XGBoost, LSTM (Long Short-Term Memory), Random Forest, and Encoder-Decoder. Each model has unique characteristics that make it well-suited for capturing temporal patterns in time series data.

1. XGBoost Model:

The XGBoost model is a powerful gradient boosting algorithm widely used for regression and classification tasks. In this code, we employ the XGBoost model to forecast traffic flow in a time series. Initially, the model is trained with default hyperparameters, and its performance is evaluated using RMSE and MAE score. To further optimize its performance, the code implements several strategies:

One key optimization in the code is finding the optimal lookback timesteps using sliding window approach. The code iterates through different lookback sizes and trains the XGBoost model for each size. The root mean squared error (RMSE) is calculated for each model, allowing us to identify the optimal lookback size that yields the lowest RMSE. Additionally, the code employs GridSearchCV, a method for hyperparameter tuning, to search for the best combination of hyperparameters for the XGBoost model. By performing a grid search over a predefined parameter grid, the model's performance is evaluated using cross-validation.

The XGBoost model is trained using the training set, consisting of X_train and y_train datasets. By fitting the model with the training data and evaluating it on the validation set, the XGBoost model learns to capture the temporal dynamics of traffic flow.

2. LSTM Model:

The LSTM model is a type of recurrent neural network (RNN) architecture specifically designed to handle sequential data. In the context of time series forecasting, LSTM models excel at capturing dependencies and temporal patterns. In this code, an LSTM model is employed to forecast traffic flow, leveraging its ability to model sequential information.

To enhance the LSTM model's performance, the code incorporates several optimizations. Firstly, the data as transformed using a sliding window approach. We feed the model with previous timesteps and then, the model tries to forecast the last value (current timestep) in the window. The optimal length of the window has been denoted before training the XGBoost model. The training and testing data are preprocessed by scaling and reshaping them to meet the requirements of the LSTM model. Subsequently, the model architecture consists of multiple LSTM layers with dropout regularization to prevent overfitting and improve generalization.

The LSTM model is trained using the provided training set, which includes X_train and y_train datasets. The model is trained to learn the underlying patterns and dependencies in the time series data. By fitting the model to the training data and evaluating it on the validation set, the LSTM model gradually improves its ability to forecast traffic flow accurately. LSTM model is evaluated using RMSE and MAE scores.

3. Random Forest Model:

Random Forest is an ensemble learning method that combines multiple decision trees to make predictions. In this code, the Random Forest model is utilized for time series forecasting of traffic flow. The model is trained using the training set, and hyperparameters are tuned using GridSearchCV to find the optimal combination.

The Random Forest model is trained using a training set consisting of input features (X_train) and corresponding target values (y_train). These input and target sets are the same used while training the XGBoost model. The training process involves building an ensemble of decision trees, where each tree is constructed using a random subset of the training data. The randomness introduced in the tree construction helps to reduce overfitting and improve generalization.

During training, each decision tree in the Random Forest is grown using a technique called bagging (bootstrap aggregating). Bagging involves randomly sampling the training data with replacement, creating multiple bootstrap samples. These bootstrap samples are used to train individual decision trees in the ensemble. Additionally, at each split in the tree, a random subset of features is considered, further introducing randomness and reducing correlation among the trees.

Once the Random Forest model is trained, it is evaluated on a separate validation or test set to assess its performance. The evaluation is typically done by making predictions on the validation or test set using the trained model. The predicted values are then compared to the true target values from the validation or test set. The RMSE and MAE scores are used to represent a numerical evaluation of the model.

4. Encoder-Decoder Model:

The Encoder-Decoder model is a sequence-to-sequence model architecture commonly used for tasks such as machine translation and time series forecasting. In this research, an Encoder-Decoder model is employed to forecast traffic flow. The model consists of an encoder network to encode the input sequence and a decoder network to generate the output sequence.

The Encoder-Decoder is trained using a combination of input sequences (trainX) and corresponding target sequences (trainY). These sets are the same as in the ones used to train the rest of the models above. We use a sliding window technique (as described above). The training process involves training two separate recurrent neural networks (RNNs): the encoder and the decoder.

- ❖ Encoder: The encoder RNN processes the input sequence (trainX) and generates a fixed-length vector called the context vector or the latent representation. The encoder's purpose is to capture the relevant information from the input sequence and summarize it into a compressed representation.

- ❖ Decoder: The decoder RNN takes the context vector as input and generates the output sequence (trainY) one step ahead at a time. It uses the context vector and the previously generated outputs to predict the next element in the sequence.

During training, the model iteratively processes batches of input sequences and their corresponding target sequences, updating the weights based on the prediction errors. The process continues for a predefined number of epochs or until a convergence criterion is met.

After training, the Encoder-Decoder model is evaluated on a separate validation or test set to assess its performance. The evaluation process involves feeding the input sequences from the validation or test set into the trained model and comparing the predicted sequences with the true sequences.

The evaluation results, including the actual values and predicted values from every model, are stored in the "total_predictions" DataFrame. This DataFrame allows for easy comparison and analysis of the model's performance. The results are further analyzed and visualized.

We conclude that the XGBoost model has the greatest performance, compared to LSTM, Random Forest and Encoder-Decoder Models. It is important to denote that this research can be mainly applied to short term traffic flow prediction. Due to the fact that traffic flow follows a nonlinear pattern, it is difficult to be accurate with long term prediction.

Needless to say that these models above are the most common used one in traffic flow forecasting. XGBoost and Random Forest algorithms are tree based, while LSTMs and Encoder-Decoder models are RNN neural networks. However, there are a lot of other important implementations that use Convolutional Neural Networks or Graph Neural Networks or combination of the above trying to forecast future traffic flow (short term or long term).