Machine Learning Algorithms for Traffic Data Forecasting on The Norwegian Road Network

Stefano de Saraca

LUMSA – Tecniche Informatiche per la Gestione di Dati L-31

Abstract

Since the beginning of time humans have traveled across the globe, evolving and improving navigation skills and techniques by time.

With the advent of the digital era, we transitioned from using paper maps to online or offline maps systems which are capable of computing the best path from one point to another within seconds.

Such capabilities come from understanding how road networks are designed, digitally represented and collecting data from every source available to improve the overall comprehension of past, present and future conditions of the whole network.

This kind of data can be especially expensive and being collected and distributed by very few companies or organizations which consequently closing opportunities for open research by the online community.

In this context, the Norwegian Roads Administration (Statens Vegvesen) opens opportunities for anyone by openly supplying great amounts of public data from the NVDB (Nasjonal vegdatabank).

The goals of my research are to:

- Recreate the Norwegian road network using the graph theory
- Collect traffic data from all traffic registration points (TRP) across Norway
- Train and use machine learning models to forecast data from the registration points
- Use the future data predictions to compute at most five best routes from a node to another in the road network

Chapter 1 – The Data

To achieve all goals of the project I needed multiple kinds of quality and interoperable data which I gathered from three main sources:

- Trafikkdata API[1]
- Statens Vegvesen
- NVDB Vegkart[2]

Online Data Sources

Trafikkdata API

The Trafikkdata API is a GraphQL API which returns different kinds of data about road traffic, measured on one or multiple traffic registration points.

It also offers data about the registration points themselves and some additional data about roads.

The choice of GraphQL makes it very fast, reliable and easy to query through specific python clients like gql.

NVDB Vegkart

The NVDB Vegkart is a service provided by Statens Vegvesen which offers great quality and diverse data from all the Norwegian road network.

1.1 Country Part

The country is divided in 6 parts, each one representing a geographical section of the surface.

Attribute name	Data type
ID	integer
Name	text

Collected using the Trafikkdata API.

1.2 Counties

The country is divided in 15 counties (*fylker*), each one belonging to a country part and representing a section of the country as well.

Attribute name	Data type
ID	integer
Name	text
Country part	integer

Collected using the Trafikkdata API.

[1] Trafikkdata API official documentation: https://trafikkdata.atlas.vegvesen.no/#/om-api
API GraphQL interface: https://trafikkdata-api.atlas.vegvesen.no/

[2] NVDB Vegkart: https://vegkart.atlas.vegvesen.no/

1.3 Municipalities

Norway has 357 municipalities.

Attribute name	Data type
ID	integer
Name	text
Country part	integer
Geometry	geometry (polygon)

Collected using the Trafikkdata API and the NVDB Vegkart.

1.4 Road Categories

The Norwegian road network is organized in 5 road categories.[3]

Attribute name	Data type
ID	text
Name	text

Collected using the Trafikkdata API.

1.5 Traffic Registration Points

Traffic registration points[4] are specific locations where sensors are installed to record multiple data about traffic.

They record both motor vehicles and bikes data, but for this project only the first kind of data has been collected.

Also, some of them could not be active, so only the data from the active ones gets collected from the downloader component of the project.

Attribute name	Data type
ID	text
Name	text
Latitude	float
Longitude	float
Geometry	geometry (Point)
Road reference (short form)	text
Road category	integer
Road link sequence	float
Relative position	float
Country part ID	integer
Country part name	text
County ID	integer
Municipality ID	integer
Geographic number	integer
Traffic registration type	text

[3] Norwegian road categories: https://www.ssb.no/en/klass/klassifikasjoner/722
[4] Trafikkdata API data registration documentation: https://trafikkdata.atlas.vegvesen.no/#/om-trafikkdata

First data	datetime
First data with quality metrics	datetime

Collected using the Trafikkdata API.

1.6 Traffic Volume

Traffic volume represents the number of vehicles which passed through a traffic registration point within a specific timeframe.

In this project the reference timeframe is exactly 1 hour.

Attribute name	**Data type**
Row Index	integer
TRP ID	text
Volume	integer
Coverage	float
Is MICE	boolean
Zoned Datetime (ISO Format)	timestamptz

Collected using the Trafikkdata API.

1.7 Average Speed

Average speed data represents the average speed of vehicles which passed through a traffic registration point within a specific timeframe.

As for the traffic volume data the reference timeframe is 1 hour.

Since this kind of data can be sensitive due to the potential localization of specific cars along a trait of road, vehicles' speed has been aggregated by groups of 5 to ensure a good level of privacy.

Average speed data for this project has been limited to the Oslo area and within the years 2018 and 2020.

Attribute name	**Data type**
Row Index	integer
TRP ID	text
Average speed	float
Percentile 85	float
Coverage	float
Is MICE	boolean
Zoned Datetime (ISO Format)	timestamptz

Provided by Statens Vegvesen.

1.8 Road Network

The representation of the road network is represented by nodes and links in an undirected graph where:

- Each node is an intersection
- Each road is a link

Each node can have multiple links connecting it to the rest of the network.

1.8.1 Nodes

Nodes of the road network represent the intersections between two or more roads. They can have multiple shapes, one of them being a roundabout.

Attribute name	**Data type**
ID	integer
Node ID	text
Geometry	geometry (Point)
Road node ids	array[text]
Is roundabout	boolean
Number of incoming links	integer
Number of outgoing links	integer
Number of undirected links	integer
Legal turning movements	jsonb
Road system references	array[text]
Raw properties	jsonb

Provided by Statens Vegvesen.

1.8.2 Links

Links connect two nodes of the road graph. They don't have a weight at first (it will later be assigned during the graph creation by leveraging on other data.

Attribute name	**Data type**
ID	integer
Link ID	text
Type	text
Geometry	geometry (LineString)
Year applies to	integer
Candidate IDs	array[text]
Road system references	array[text]
Road category	text
Road placements	jsonb
Functional road class	integer
Function class	text
Start traffic node ID	text
End traffic node ID	text
Subsumed traffic node IDs	array[text]

Road link IDs	array[text]
Road node IDs	array[text]
Highest speed limit	integer
Lowest speed limit	integer
Max lanes	integer
Min lanes	integer
Has only public transport lanes	boolean
Length	float
Traffic direction wrt[5] metering direction	text
Is Norwegian scenic route	boolean
Is ferry route	boolean
Is ramp	boolean
Traffic volumes	jsonb
Urban ratio	float
Number of establishments	integer
Number of employees	integer
Number of inhabitants	integer
Has anomalies	boolean
Anomalies	jsonb
Raw properties	jsonb

Provided by Statens Vegvesen.

1.9 Toll stations

Toll stations are located on some roads, some of them get distributed as 3D points while others as 2D ones.

For the purposes of this project we'll store all of them as 2D points using geospatial functions to adequate the 3D ones.

Attribute name	**Data type**
ID	integer
Name	text
Geometry	geometry (Point)

Collected from the NVDB Vegkart.

1.10 Function Road Classes

Function road classes categorize roads based on their scope in the road network.

Attribute name	**Data type**
ID	integer
Name	text

Collected from the NVDB Vegkart.

[5] w.r.t. stands for "with respect to"

Chapter 2 – Overview on the Software Architecture

2.1 The Requirements

The software architecture has been a challenging task of the project which took months to improve involving multiple technologies and libraries to achieve a good level of stability.

The ultimate goal is to have the most efficient, clean, readable and well-structured code possible.

At the same time one other big non-negotiable was the software to be scalable to potentially process greater quantities of data.

A key role in training models on big data is to be able to spread the processing workload across multiple devices and being able to parallelize it on all the CPUs available.

Also, storing really large amounts of data needs a system which is capable to ensure integrity, consistency, being reliable and to enable the user to store spatial data and create and use custom functions.

Considering all of these requirements this is the tech stack of the project:

- **PostgreSQL**: a high-performance and widely used ORDBMS[6], which is a standard for many large-scale applications
- **PostGIS**: a popular and powerful PostgreSQL extension which offers great spatial data management and processing functions
- **Python**: the standard programming language for multiple applications, especially in machine learning and data science
- **Dask**: a powerful library which shares the same API the Pandas library, but doesn't store data in memory by executing operations through lazy evaluation dividing the whole dataset into smaller partitions.
  It also enables the user to spread computations across a cluster of devices and parallelize workloads
- **GQL**: a fast and reliable GraphQL python interface to query such APIs
- **AsyncPG and Psycopg3**: two intuitive and high-level libraries to query PostgreSQL databases asynchronously and synchronously
- **PyKrige**: a high-level and specialized Python library to execute spatial interpolation with Ordinary Kriging[7] on spatial data
- **NetworkX**: a powerful and user-friendly Python library specialized in graph structures and related algorithms.

2.2 Isolated Projects

The software organizes data and processing results in projects. Each one represents an isolated space where data can be stored, extracted for processing purposes and analyzed.

A project is backed by a custom database which holds all the data and supplies it when requested.

All projects have the same internal tables structure and don't share any data with other ones.

[6] ORDBMS: https://en.wikipedia.org/wiki/Object%E2%80%93relational_database
[7] Kriging: https://en.wikipedia.org/wiki/Kriging

A dedicated database keeps track of all projects and the user can choose to work on one project only at a time.

The projects creation is asynchronously computed to speed up the process, while the interrogations can be synchronous or asynchronous depending on the operations that are being executed.

2.2.1 Automatic Project Setup

When a project is created the system automatically sets it up with all the necessary data to execute even basic operations or future more complex ones.

This has to be done when connected to the Internet since this data is downloaded directly from the Trafikkdata API.

2.3 Modularity and Independence of the Databases

The software is built in independent modules, each one serving specific tasks.

All functionalities are accessible through the main module where the end user can actually operate on the data with the software.

2.3.1 Modules

The program main modules are:

- **DB Manager**: hosts a single class which offers methods to set up or update data on project databases and more
- **Downloader**: offers functions dedicated to downloading data from the Trafikkdata API
- **Brokers**: offers multiple types of brokers to use across the whole codebase to query the project database
- **Pipelines**: supplies the codebase with specific pipelines for ingestion, processing and machine learning prediction workflows for the data
- **Loaders**: offers a set of loaders to extract data from the database with a wide range of filters and additional transformation
- **ML**: a dedicated module only to hyperparameter tuning

2.3.2 Independence of the Databases

One of the key features of the software architecture of the project is the way each component of the program interacts with the database.

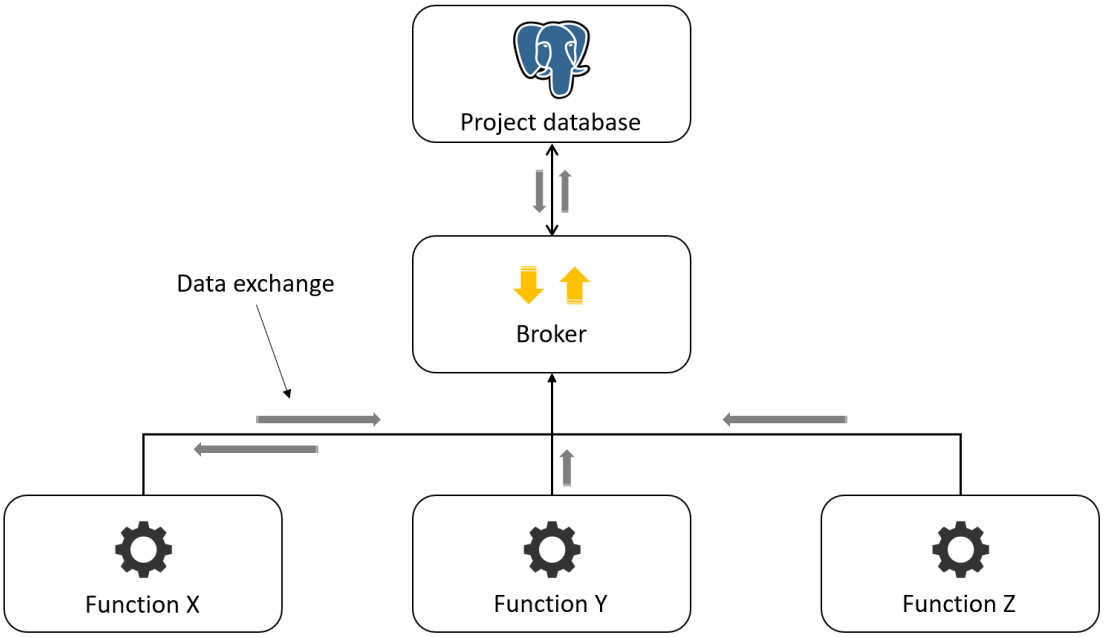
Each component requests data through brokers, which are the only components that can directly query the database.

Each broker serves one of two contexts, which are:

- Synchronous
- Asynchronous

They have the ability to supply different modules of the program with either convenience functions which directly return the data needed or let the module's functions execute custom SQL queries.

This way we ensure that, in case multiple functions need the same data, the returned content is always identical and consistent.



2.3.3 Asynchronous Batches Data Ingestion

Data ingestion is a delicate part of whole system, it involves communicating with third-party services and implementation of best practices to avoid undesired blockings from the data supplier, network errors and memory overflows.

That's why the project's system downloads data with a customizable number of asynchronous jobs where each one fills up batches iteratively until there's no more data to download.

By using batches we avoid needing to store all data at once and potential RAM overflows.

The downloader module also implements exponential backoff mechanisms to avoid flooding the API of requests in case specific problems may occur.

2.3.4 Data Preprocessing and Cleaning Through Modular Ingestion Pipelines

One crucial step between downloading the data and storing it is cleaning.

This is done by custom-designed asynchronous pipelines which canalize incoming data into a series of processes to re-structure it and address a series of potential problems.

The main transformations are:

- Checking if any data is actually available in the incoming JSON or CSV data
- Parsing the data efficiently and creating a dataframe structure

- Executing the MICE algorithm to impute missing values by generating synthetic data
- Ingesting the data into the project database

It's important to underline that the transformations are identical for both volume and average speed data, but executed in two different contexts.

Volume data is cleaned automatically immediately after the it's fetched from the Trafikkdata API.

Average speed data is cleaned automatically when the user wants to ingest it through a specific option in the software's menu.

2.3.4.1 Addressing Missing Values with MICE

Traffic registration points can sometimes not cover all hours of a time period or have missing values for certain timeframes.

That's why in the data cleaning pipelines the MICE[8] algorithm is implemented.

This algorithm takes an estimator and the dataset with missing values as its input and returns the dataset without missing values.

Multiple parameters can be set to improve its performances, like the direction of execution, the imputation strategy and so on.

2.3.4.2 Context Extraction to Improve MICE Performances

Because of the intrinsic functioning of the MICE algorithm, it's important, in case of especially small batches of data, to help the imputer estimator understand the "context" of the current batch values.

This means that if there aren't many values to support the imputation, the results could not closely represent what actual ones might have been.

In other words, we're attaching, if present, a block of past data to the one to ingest so that the imputer can better understand what missing values would have been relative to a greater time window.

The context window size is defined as:

$$\max\left(1, \left\lceil \frac{r}{2} \right\rceil\right)$$

r: number of rows in the current batch to feed to MICE

2.3.4.3 The choice of Zero Adjusted Gamma Regression for MICE

Volume and average speed data have one subtle, but very important problem, they can be zero when no cars pass by traffic registration points and these conditions can be rare.

An example of this would be night hours or holiday periods when people are out of the city.

[8] Multiple Imputation by Chained Equations
R package article: https://cran.r-project.org/web/packages/miceRanger/vignettes/miceAlgorithm.html
Python scikit-learn version: https://scikit-learn.org/stable/modules/generated/sklearn.impute.IterativeImputer.html

Regression models don't know this and imputations for cases like the ones mentioned above can be a real challenge.

The solution is zero-adjusted[9] (zero-inflated) models, which integrate the ability to predict rare zero values as well.

Zero inflated models in this project have been implemented with the Scikit-Lego[10] library. They implement a classificator to identify zeros in the data and instruct the imputer's estimator to predict them.

As estimator a gamma regressor[11] was chosen since traffic volume and average speed data can't be negative, so the lower bound has to be zero.

2.3.5 Data Extraction Through Custom Loaders

Multiple modules of the program need to extract data from storage to process it, but this is a task that can't be assigned individually to each module function.

Again, the risk is akin to the one mentioned in the description of the brokers module: the database and all specific functions of the software must remain separated.

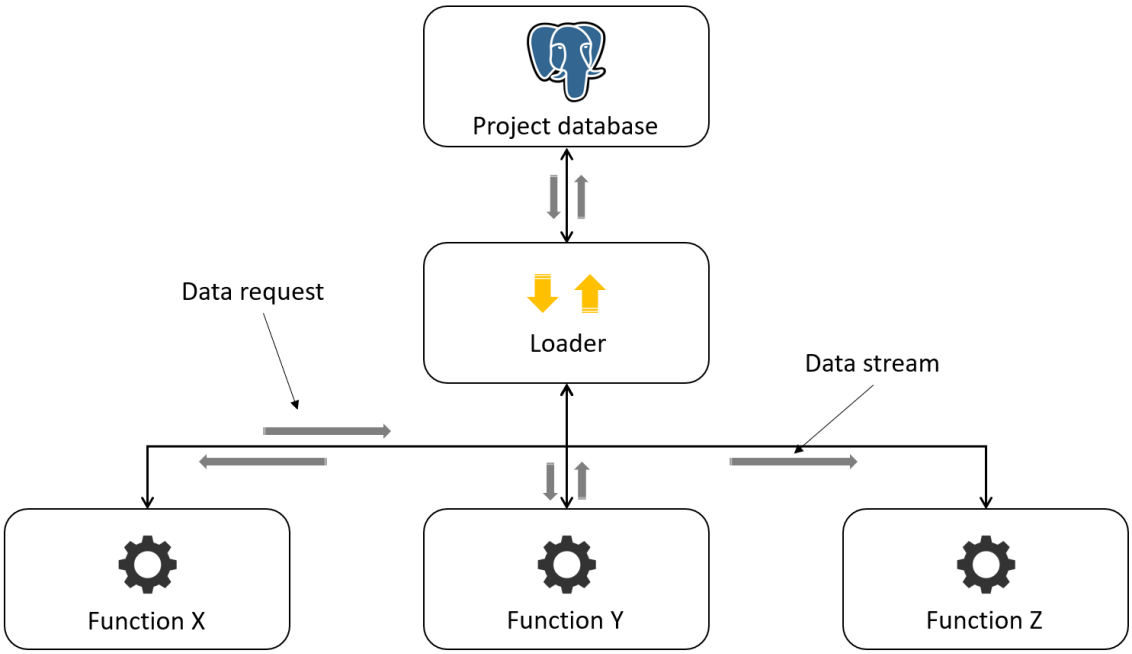
In this case, loaders have been created: specialized classes to pull out data from the database and allow the user or the functions to filter data by multiple parameters.

The extraction process is handled through record streams and processed in batches to avoid filling up the memory.

Each kind of data has its own loader, for example:

- Volume data loader
- Average speed data loader
- Road objects loader

[9] Zero-inflated models: https://en.wikipedia.org/wiki/Zero-inflated_model
[10] The Scikit-Lego python library: https://github.com/koaning/scikit-lego
The Zero Inflated Regressor class: https://koaning.github.io/scikit-lego/api/meta/?h=zero+inflated#sklego.meta.zero_inflated_regressor.ZeroInflatedRegressor
[11] Scikit-Learn's GammaRegressor: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.GammaRegressor.html



Graphical representation of loaders working mechanism

2.3.6 Machine Learning and Predicting Future Data

The project's core is the machine learning section, but counterintuitively as one might imagine, it's split between two different modules given the different nature of the two operations that characterize this part of the project.

The first part is the hyperparameter tuning one while the second one is the actual forecasting pipeline that returns the predicted data for the future.

2.3.6.1 Hyperparameter Tuning with GridSearchCV

To improve the predictive models' performances a hyperparameter tuning[12] functionality enables the user to test the same model with multiple parameter configurations and identify the best combinations to get the best out of each estimator.

This is done by using Dask-ML's GridSearchCV[13] and Dask Distributed.

Both libraries are part of the Dask ecosystem, the second one creates a distributed environment within a network where one or more schedulers distribute and coordinate one or more worker devices which actually execute the computations.

The first one is a distributed version of the popular Scikit-Learn GridSearchCV[14].

[12] Hyperparameter tuning (also called hyperparameter optimization): https://en.wikipedia.org/wiki/Hyperparameter_optimization
[13] Dask ML version of GridSearchCV: https://ml.dask.org/modules/generated/dask_ml.model_selection.GridSearchCV.html
[14] Scikit-Learn version of GridSearchCV: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

2.3.6.2 Time Series Splitting for Cross Validation

Being the dataset composed by multiple time series (one for each TRP) the cross validation must follow a precise order, where the model gets validated on only a part of the dataset at a time concatenated to the previously used ones.

For this purpose I used the TimeSeriesSplit[15] class from Scikit-Learn and set the cross validations number of groups to 10.

2.3.6.3 Future Data Prediction Pipeline

Future data forecasting is a critical pillar of the whole project. It's the actual section of the program which uses previously trained models to predict future data.

The forecasting process is relatively simple and can be divided in three steps:

- Creating an empty dataset to fill with future data
- Extracting past records to create lag features for the dataset to fill with future data
- Feeding the empty dataset to the model and predicting the data

Everything is carried on by a single pipeline which has which can be configured by the user to execute an optional virtual fine-tuning of the model on the TRP's data for improved predictive performances. This means that the model doesn't actually change, but when it gets used in that specific context it gets furtherly trained before actually predicting future data.

2.3.7 Road Network Representation and Analyses

The use of machine learning algorithms predictions to find the best path from two points is the ultimate goal of the project; to achieve it, the best data structure is certainly a graph.

For this project the chosen kind of graph is an undirected weighted graph, which doesn't account for the direction of the links between nodes.

The road network is built by loading nodes and links and putting them together using the NetworkX library.

After that a wide range of operations can be executed on it.

Operations available:

- Loading nodes (manually)
- Loading links (manually)
- Find up to *n* best routes from a generic point A to B
- Create a map of a route
- Create a map of a municipality with a traffic heatmap overlay to show the most trafficated areas (available for both volume and average speed data)
- Export a map
- Compute the betweenness centrality for the graph nodes
- Compute the degree centrality for the graph nodes
- Export the NetworkX graph to various file formats
- Save the graph's graphical representation as svg file

[15] Time Series Split: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html

Chapter 3 – Machine Learning Algorithms – Tuning, Training and Testing

As mentioned in the previous chapters the data forecasting with machine learning algorithms is one of the two core functionalities of the project.

Here will follow a detailed description of all the models and strategies used for to achieve this goal.

3.1 Models Used and Potential Expansions

For this project one of the main goals was the system to be as efficient as possible and some machine learning algorithms may require heavy computations and slow down everything.

But there are also plenty of amazing opportunities to try with models from other libraries.

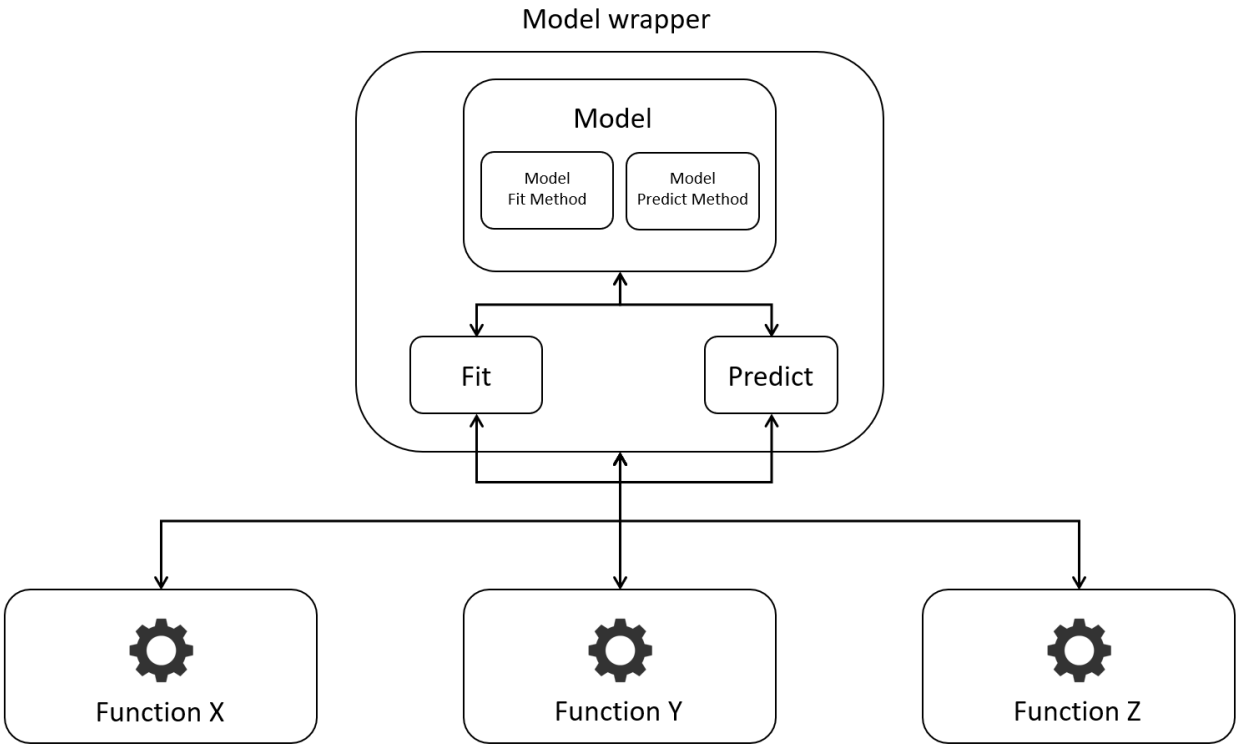
The chosen models for this project are:

- Decision Tree Regressor
- Hist Gradient Boosting Regressor

Both of them are obtained from the Scikit-Learn library and represent different approaches.

While the first one is a simple decision tree adapted for regression problems, the second one uses the histogram method combined with gradient boosting and despite this last characteristic it still has great performances.

To have a unified interface to use the models a model wrapper is implemented, which lets the user interact with only two methods to either train the model and predict the data.



3.2 A Different Approach on Model Training

As mentioned before in the first chapter, there could be multiple kind of road categories for each road and the traffic data we may want to predict can vary.

The first approach one would take is to train one or more models on all data, including both the road category and each target variable all together in the train and test sets, but that's not the choice made for this project.

The main idea behind the strategy adopted for this system is that we're interested in understanding the performances of multiple instances of different models, each one specialized for predictions on a certain road category and target variable[16].

The goal is to forecast future data on each single trait of a route by using a specific model based on its characteristics (road category and target variable) for each one.

3.3 Train-Test Split

The data used for all processes is split into 2 or 4 datasets depending on the operation to execute.

In both cases the data is split in predictors and target which correspond respectively to *X* and *Y*.

4-Sets Split

The classic 4 sets split divides the whole dataset into 2 parts for the training phase and the testing one.

This step is applied for the predictors set and the target one, resulting in 4 different sets.

The splitting point is the row which delimits the first 70% of the whole dataset, which is used for training and the remaining 30% is used for testing.

This kind of splitting is used for hyperparameter tuning, models training and in other scopes.

2-Sets Split

The 2 sets split aims instead for the model to leverage on all the data available avoiding any split for training and testing.

Thus, the resulting sets are just the whole predictors' data and the target one.

This kind of splitting is only used if the user wants to furtherly train the model only on a specific TRP[17] data.

[16] Target variable: the kind of data we want to forecast. For example: traffic volume or average speed

[17] TRP: traffic registration point

3.4 Data Preprocessing

The data preprocessing phase is crucial to feed the model good quality data for optimal predictions.

The transformations in this section are various, each one serves a specific purpose and they are implemented in a custom pipeline for each kind of data (traffic volume or average speed).

There are 6 steps to prepare the data before the actual usage with the machine learning algorithms:

- Z-Score (for outliers' removal)
- Lag features computation
- Categorical features encoding
- Geodesic coordinates transformation to UTM
- Features scaling
- Data sorting

3.4.1 Z-Score

The Z-Score computation for outliers' removal is optional, but it's activated by default since removing outliers is a standard procedure for data preprocessing in machine learning.

3.4.2 Lag Features Computation

Lag features help the model understand the past values of the target feature to predict.

The choice of which lag features to use also determines the maximum forecasting horizon which the model will be able to predict for.

Given this, the user can choose which lags to use and train the model to have a closer or farther forecasting horizon.

The default lags values used are: [24, 36, 48, 60, 72]

Each lag must strictly represent a number of hours since the measurements recorded by TRPs are hourly.

Each lag is computed grouping by TRP to ensure that each lagged value corresponds to the past data of the TRP it actually belongs to.

3.4.3 Categorical Features Encoding

Models learn categorical features better when they are encoded and there are multiple strategies to do that.

The chosen one for this project was Label Encoding[18] and the only feature that's going to be encoded is the ID of the Traffic Registration Point which the data belongs to.

[18] Label encoding: https://www.geeksforgeeks.org/machine-learning/ml-label-encoding-of-datasets-in-python/
Dask Label Encoder: https://ml.dask.org/modules/generated/dask_ml.preprocessing.LabelEncoder.html

3.4.4 WGS84 Coordinates Conversion to UTM

Geodesic coordinates, in WGS84 format[19], are very handy where it comes to localize an object on Earth's surface, but they have a major downside where it comes to machine learning and models needing to learn locations: they don't account for distances linearly because of the planet's curvature.

When two points lie closer to the poles, degrees, minutes and seconds don't represent the same distance as if the two points where closer to the equator.

So, models wouldn't understand the relationship between the difference in degrees, minutes and seconds as if it was described with other projections.

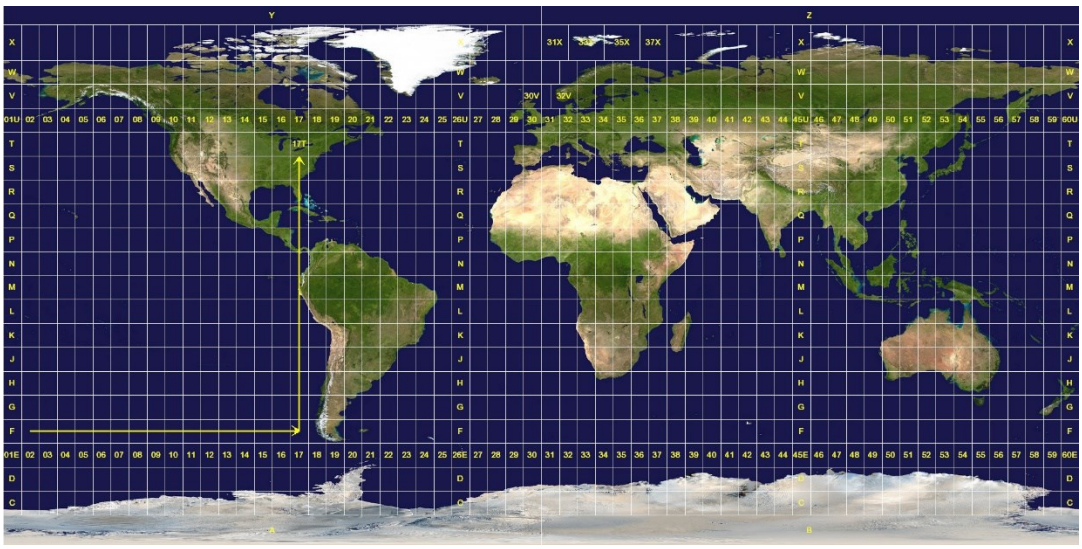
For this reason, UTM location-specific projections exist, each one is dedicated to a specific region on earth, minimizing distortion and representing actual distances through so called "eastings" and "northings"; some countries may even have multiple UTM zones.

As a matter of fact, the reference systems used in Norwegian systems may vary within specific UTM zones[20], defined by the international standard Euref89[21]: UTM 32, UTM 33, UTM 35 or standard UTM variants like UTM 34 and UTM 36 for covering large sea areas or for other applications.

Mainly these three reference systems are used for nationwide or more detailed location data:

- UTM 32 (municipalities in Innlandet, Oslo, Viken, Vestfold og Telemark, Agder, Rogaland, Vestland, Møre og Romsdal, Trøndelag.)
- UTM 33 (municipalities in Nordland and the Troms-part of Troms og Finnmark)
- UTM 35 (municipalities in the Finnmark-part of Troms og Finnmark.)

The nationwide UTM reference is UTM 33, but for this project the utm[22] Python library has been used to automatically determine the UTM zone of geodesic coordinates and get utm eastings and northings to actually use in the datasets to minimize distortions and feed the models with better quality data.



[19] World Geodetic System: https://en.wikipedia.org/wiki/World_Geodetic_System
EPSG:4326 – World Geodetic System 1984: https://epsg.io/4326
[20] Norwegian UTM projections: https://www.geonorge.no/en/references/references/coordiante-systems
Norway ESRI zones: https://epsg.io/?q=Norway
[21] European Terrestrial Reference System 1989:
https://en.wikipedia.org/wiki/European_Terrestrial_Reference_System_1989
[22] utm library for Python: https://pypi.org/project/utm/

3.4.5 Features Scaling

Scaling is a mandatory step in machine learning preprocessing where the features may have different orders of magnitude compared to each other.

For this project a MinMax[23] scaler has been used exclusively on target features, percentile_85 (where average speed is the target feature) and coverage.

3.5 Hyperparameter Tuning

Hyperparameter tuning leverages on parallel computing and is executed iteratively on a distributed cluster[24] using a grid of parameters trying all combinations.

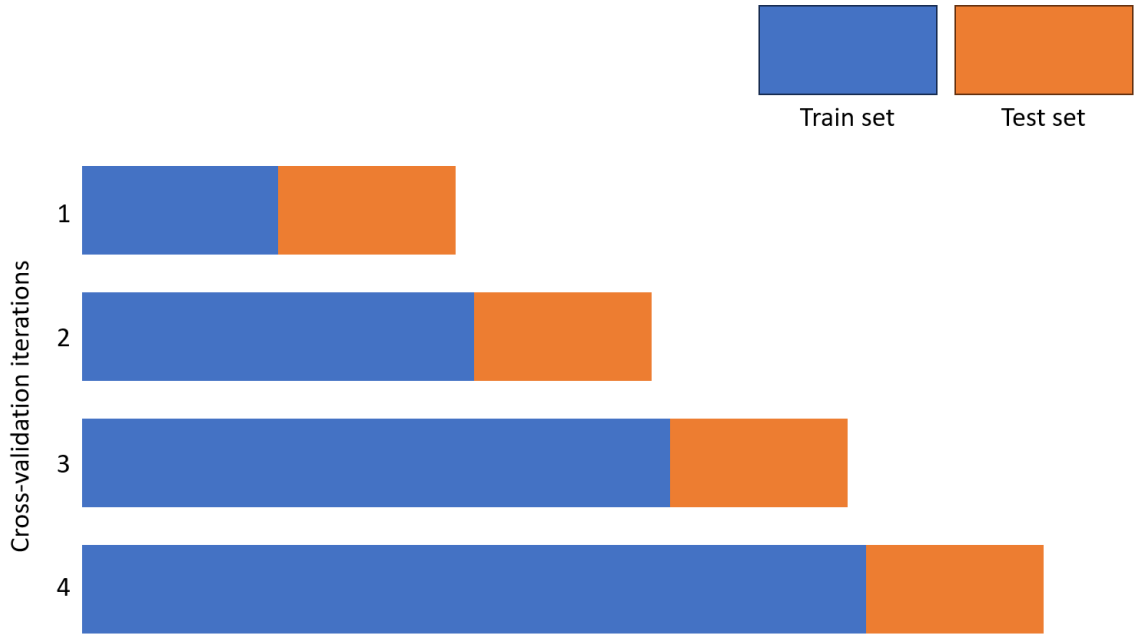
The algorithm used is known as GridSearchCV, it's a quite popular choice and the distributed version reduces drastically the execution time depending on the number of workers of the cluster.

The grid is customizable[25] by the end user and the results are automatically exported to a dedicated table inside the project's database.

The search re-fits the models to optimize the MAE loss function and uses a time series specific splitter[26] for the cross-validation process.

3.5.1 Time Series Cross-Validation

Time series cross-validation is designed to improve the model's performances by iteratively uncovering part of the whole time series it has to learn so that each time it learns from a new set of data and the past until that moment in time.



[23] MinMax scaler: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html
[24] Dask distributed creates a cluster of devices which interact with each other in a manager-worker architecture. Specifically, Dask uses a "scheduler" to manage and spread the workload to all workers.
There can also be multiple schedulers depending the size of the cluster itself.
[25] The user can insert a custom JSON file with grids for each target variable and model that will be used for the GridSearchCV
[26] TimeSeriesSplit: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.TimeSeriesSplit.html

3.6 Training

The model's training process it pretty simple and direct and can be described in 5 steps:

- An untrained model gets loaded from the database
- The training data gets loaded
- The model parameters to use get loaded from the database
- The model gets trained
- The trained model gets pickled[27] and exported to the database for further usages

The training is also distributed with Dask across the cluster workers to speed up the process.

3.7 Testing

The testing phase is executed twice, the first time in the grid search and second one when testing the models with a specific option in the system's menu.

The testing set is exactly the same one used in the grid search process and the scorings are calculated with a set of loss functions which are:

- MAE[28]
- MSE[29]
- RMSE[30]
- R^2 [31]

Computing forecasting prediction errors with different scorings helps us better determine the quality of the models and potentially understand what kind of improvements can be made.

[27] Data pickling: https://dagster.io/glossary/data-pickling
[28] MAE (Mean Absolute Error): https://en.wikipedia.org/wiki/Mean_absolute_error
[29] MSE (Mean Squared Error): https://en.wikipedia.org/wiki/Mean_squared_error
[30] RMSE (Root Mean Squared Error): https://www.sciencedirect.com/topics/engineering/root-mean-square-error
[31] R^2 (Coefficient of Determination): https://en.wikipedia.org/wiki/Coefficient_of_determination

Chapter 4 – Traffic Data Forecasting

Traffic data forecasting is the base on which the route-finding algorithm, which I'll describe in another chapter.

It's the section of the system which is responsible for predicting future data on TRPs by using the previously trained models and returning results to be then be used in other parts of the program.

Of course, given that there are thousands of TRPs in Norway it's not both efficient and reasonable to forecast traffic data for each one if not strictly necessary to find the traffic conditions on a route.

For this reason, the predictions are only executed on a single TRP at a time and the functionality is called "One-Point Forecast".

4.1 One Point Forecast

One-Point Forecast is the first core functionality of the system and it's structured to predict data for the future a given target feature, at a certain TRP, at a specific forecasting horizon.

The forecasting horizon needs to match with the smallest lag feature set by the user as also discussed in the previous chapters and paragraphs.

The user can enter all necessary data (like: TRP ID, target feature, lags, predictive model, etc.) before starting the prediction processes.

Everything is then executed in a pipeline structured in 5 main steps:

1. Loading past data to create lag features for the records yet to predict
2. Creating a skeleton dataframe where to enter future data predictions
3. Merging both datasets, preprocessing the result and removing past data
4. Forecasting future data
5. Returning the results with additional processing to make it readable

The results are stored and shown as a dataframe.

4.2 Decoding Cyclical Features

In the post-processing phase of prediction results cyclical features are decoded to make them readable again.

Decoding happens by using sin-encoded and cosine-encoded values together with the function:

$$k = \left(\left\lfloor \frac{\arctan 2\,(sin\_val, cos\_val)}{2\pi} \cdot period + start + \frac{1}{2} \right\rfloor \right) \bmod period$$

k = decoded value

sin_val = the sin-encoded value

cos_val = the cosine-encoded value

period = the periodicity of the cyclical feature

start = the start value of the cyclical feature

4.3 Unscaling Target Features

Unscaling target features is one crucial step to get usable data from the prediction results.

As the models get fed with scaled target features, they output scaled values, so by using the reversed scaler implemented at preprocessing phase unscaling takes place and true-sized predictions are uncovered.

Chapter 5 – Road Network Representation

5.1 Description of The Road Network

The Norwegian road network is very similar to many other ones in the world and Statens Vegvesen's data include great amounts of data to create a digital representation of it using the graph theory.

Specifically, the network is composed by lots of different data that can be separately extracted from the NVDB, some of it is purely about the construction in terms of infrastructure design and more, and other parts do actually offer important information where it comes to representing a road system and giving path-finding algorithms more ways to filter certain types of roads or network sections to avoid or favor.

5.2 Representation Through Graph Theory

Graph theory is certainly the first choice where it comes to represent a complex network like a road system.

The most important step of all is understanding which parts of a road system should be used as edges and which as nodes.

The obvious solution to this problem is to use roads as edges connecting nodes, which represent intersections.

Having set up the graph this way lets us also filter edges with specific characteristics, excluding them from specific operations like path finding or other analyses.

An example would be searching the shortest path between two nodes where the links don't include the ones where toll stations are placed.

The chosen library to realize and operate on graphs for this project is NetworkX.

5.3 Building the Graph

The type of graph chosen for the system is an undirected graph, where the edge weights are calculated at runtime of path-finding algorithms to optimize memory usage.

5.3.1 Nodes

Nodes represent intersections, they can be also roundabouts and have multiple incoming and outgoing edges. No transformations are executed in the data loading process except the extraction of latitude and longitude from the geometry of each node for future operations.

5.3.2 Edges

Edges represent roads or ferry routes connecting nodes, each edge is segmented into multiple sections and all of them are included in the unique edge's geometry.

Of course, each edge is uniquely identified with a starting node and an ending node, both referencing one within the nodes set.

No transformations are executed in the data loading process.

Chapter 6 - The Route-Finding Algorithm

This is the second core functionality of the whole system; it's an iterative algorithm which uses a research-range incremental approach to find the best path between two nodes starting from a base shortest path.

The algorithm takes as input:

- The graph
- A source node
- A destination node
- The target features to use to determine le traffic level
- Avoid toll stations filter
- Avoid ferry routes filter
- Avoid public transport-only lanes
- The radius in which to research TRPs for traffic detection (expressed in meters)
- The model to use for the predictions
- The maximum number of paths to return

6.1 Steps Sequence

These are the main steps of the route-finding algorithm:

1. Identifying the shortest path between the start and destination nodes
2. For each edge of the shortest path identify the TRPs which lie within the research radius imputed as parameter (as mentioned before)
   This process gets repeated until a minimum of 3 neighboring TRPs are identified. Each iteration increments the research radius by 500 meters
3. Calculate traffic predictions at forecasting horizon for each TRP
4. Define the path with all of its details (nodes, edges, predicted travel time, TRPs along the path, etc.)
5. Execute spatial interpolation on the predictions for each TRP to extract the target feature values for every edge's set of sections and determine the traffic class for each section
6. If the percentage of links that have a high-traffic class is higher than 50% try again by continuing the iterations until a better path is found.
   Meanwhile, at every iteration remove the top n edges which have the greatest weight to make sure that the paths found each time are different
7. Return the paths found sorted by predicted travel time

6.2 Edge Weighting

The weight of the graph edges is calculated as:

$$W = \left| \underbrace{\frac{L}{\frac{(V_{max}/100) \cdot 85}{3.6}} \cdot M_R \cdot 0.35}_{\text{travel time factor}} + \underbrace{0.10N_{min} + 0.10N_{max}}_{\text{lane factor}} + \underbrace{0.20 \cdot \frac{V_{max}}{100} \cdot 85 + 0.20V_{max}}_{\text{speed factor}} + \underbrace{0.05V_{min}}_{\text{lowest speed factor}} \right|$$

$L =$ edge length

$R =$ road category

$MR =$ road category multiplier

$Nmin =$ minimum number of lanes

$Nmax =$ maximum number of lanes

$Vmax =$ highest speed limit

$Vmin =$ lowest speed limit

Each one of these variables is available for all edges and contributes to the calculation of the edge weight.

Each factor indicates its contribution to the total weight of the edge, taking into account the theoretical travel time having the vehicle driving at 85% of the speed limit with a road category multiplier which accounts for the variability of the flow and possibility to go faster or slower than usual.

For example: a highway will have a higher capacity than a regular city street with fewer and smaller lanes with a lower speed limit, so it's more likely that one may travel even faster than usual (within the speed limit).

Also, higher flow roads like highways don't have traffic lights, so there's less variability in whether the vehicle will need to stop more or less times during a journey.

Road category multipliers are defined as:

- E = 0.5
- R = 0.7
- F = 1.3
- K = 1.5
- P = 2.0

Of course, the weight must always be positive, otherwise algorithms like A* or Dijkstra won't work correctly.

6.3 Finding the Shortest Path with A*

To find the shortest path between the start and destination nodes the A* has been implemented.

The advantage of using A* is the usage of a heuristic which eliminates the need to visit all nodes of the graph to find the shortest path.

The heuristic used for this project is the Minkowski distance[32].

The weighting function used is the one discussed in the previous paragraph that takes into account many different aspects of the road.

6.4 Spatial Interpolation with Ordinary Kriging

Models are trained to predict data also based on the location formatted as UTM eastings and northings, but these actually only represent the positions of TRPs and not a generic road that may have way different measurements.

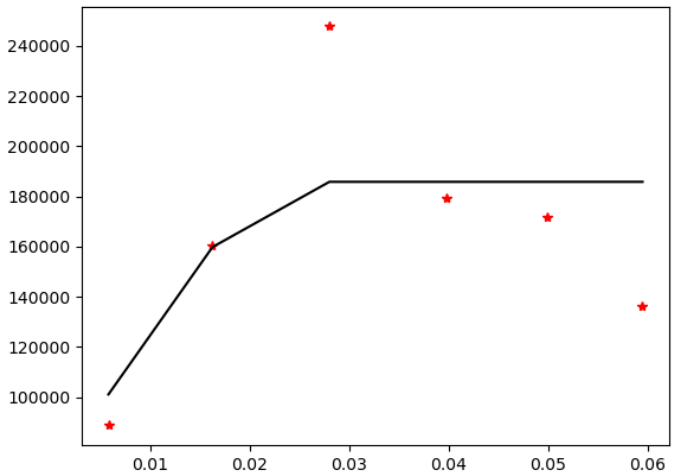
Also, one TRP doesn't explain all traffic of a specific area, especially considering the great difference in TRP presence between urban and rural areas.

Given that for each edge of the shortest path between two points we can leverage on all TRPs within a certain buffer, we can determine the state of traffic by interpolating data from each one to obtain a more realistic estimate of the real traffic situation.

There are multiple techniques for spatial interpolation[33], but for this specific project Ordinary Kriging has been used.

For this purpose the PyKrige[34] library has been the first choice given its ease of use and flexibility.

The variogram[35] model used for the kriging process is the "geographic" one since it accounts for the curvature of the Earth, improving the accuracy of the estimates.



Example of a variogram approximating how spatial correlation increases with distance

[32] Minkowski distance: https://en.wikipedia.org/wiki/Minkowski_distance
[33] Spatial interpolation: https://pro.arcgis.com/en/pro-app/3.4/tool-reference/spatial-analyst/understanding-interpolation-analysis.htm
[34] PyKrige library: https://geostat-framework.readthedocs.io/projects/pykrige/en/stable/
[35] Variogram: https://en.wikipedia.org/wiki/Variogram

6.5 Traffic Classes Assignment

Defining traffic class for each link section (lat-lon couple) is pretty easy, but requires aggregated traffic data for each edge.

First of all, weighted means on traffic data for each edge are computed to give a global view of the average traffic state of traffic in every road trait of the whole path.

The weighted means calculated identically for both volume data and average speed one as:

$$\text{weighted avg}_{\text{VOLUME}} = w_c \cdot \text{avg}_{\text{VOLUME}}^{\text{county}} + w_m \cdot \text{avg}_{\text{VOLUME}}^{\text{municipality}} + w_{rc} \cdot \text{avg}_{\text{VOLUME}}^{\text{road category}}$$
$$\text{weighted avg}_{\text{MEAN\_SPEED}} = w_c \cdot \text{avg}_{\text{MEAN\_SPEED}}^{\text{county}} + w_m \cdot \text{avg}_{\text{MEAN\_SPEED}}^{\text{municipality}} + w_{rc} \cdot \text{avg}_{\text{MEAN\_SPEED}}^{\text{road category}}$$

The average traffic state of an edge is determined by three factors calculated by grouping a target feature *x*, each one influencing more or less the final result depending on the weights that have been set:

- $\text{avg}_{\text{x}}^{\text{county}}$: given that a county may be bigger or smaller than others and/or include greater or smaller cities, this factor shouldn't greatly influence the final result
- $\text{avg}_{\text{x}}^{\text{municipality}}$: the municipality more deeply represents the traffic state of the edges which belong to that area given the more granular selection of roads to take into consideration for the averaging computations
- $\text{avg}_{\text{x}}^{\text{road category}}$: represent the traffic state on a road category, which can better account for the vehicles flow along the day given the great difference between certain categories.
  For example: the speed limit difference between highways and municipal roads is remarkable

A weight is assigned to each aggregated factor to have more flexibility on choosing the relevance of each one.

The standard weights are:

- $\text{avg}_{\text{x}}^{\text{county}}$: 0.25
- $\text{avg}_{\text{x}}^{\text{municipality}}$: 0.50
- $\text{avg}_{\text{x}}^{\text{road category}}$: 0.25

Based on the values recorded, the traffic class will indicate how much they differ from the average traffic condition of that road (percentage).

The assignment correspondence table is:

Condition (pct_diff)	**Traffic Class**
`< -25`	LOW
`-25 ≤ pct_diff < -15`	LOW_AVERAGE
`-15 ≤ pct_diff ≤ 15`	AVERAGE
`15 < pct_diff ≤ 25`	HIGH_AVERAGE
`25 < pct_diff ≤ 50`	HIGH
`> 50`	STOP_AND_GO

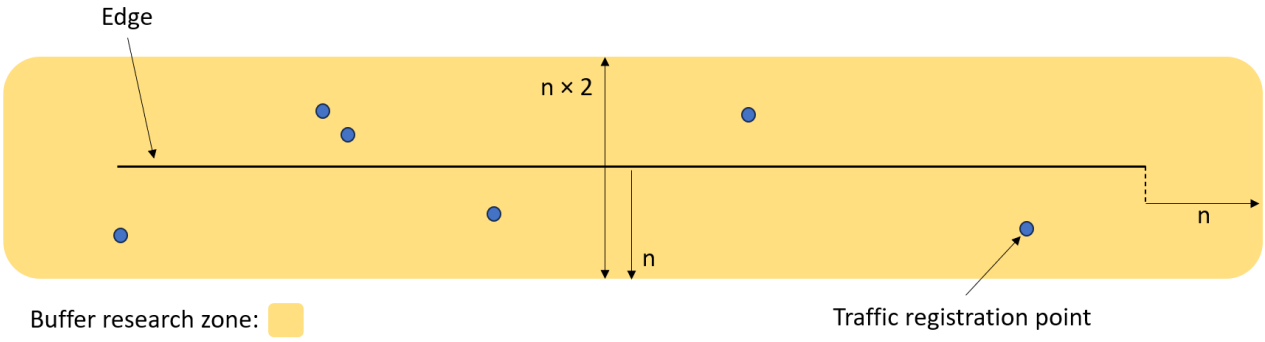
6.6 Buffer Zones and Neighbors Research

Buffer zones let us find specific objects within a certain radius from one or multiple points of reference.

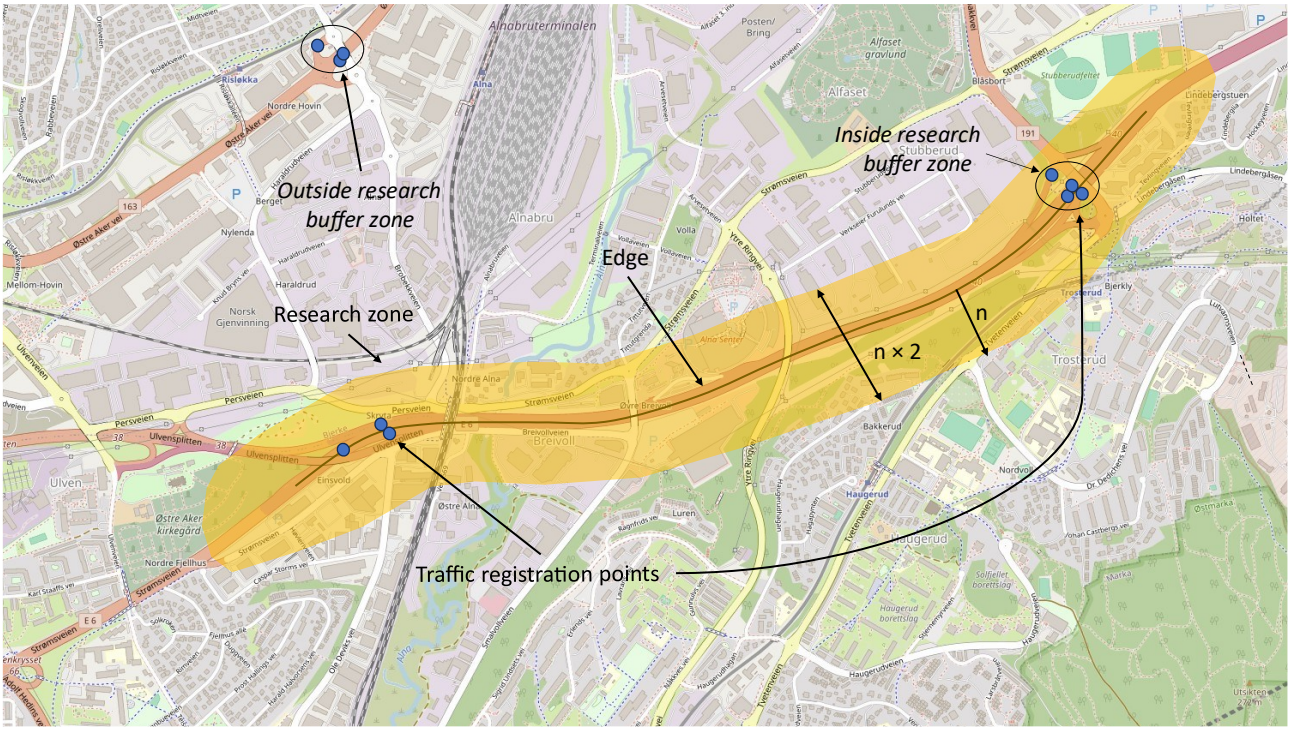
In the path-finding algorithm this comes particularly helpful when needing to find TRPs within a number of meters from each edge.

The buffer is calculated using a PostGIS specific function and in case of a line it is defined as a zone with a maximum distance from the line itself of n meters.

This means that the total buffer zone has a width of n × 2 since the buffer is computed for both sides of the line.



Buffer research zone example on a generic edge with neighboring TRPs



Buffer research zone example on a real road in Oslo with TRPs included in a buffer zone

Chapter 7 – Drawing Routes and Traffic Forecasts

In the previous chapter we've seen how the route-finding algorithm can iteratively research better paths and return them, but still couldn't visualized them on a map to be used for an actual navigation.

The functionalities described here let the user visualize not only a representation of the paths, but also a heatmap of the traffic conditions of a whole municipality by overlaying a 2D contour plot (also known as a heatmap) of a grid spatial interpolation of the traffic data predictions on an interactive map.

7.1 Drawing Routes

The project's system offers two convenience functions to plot routes, the first for a single one, while the second lets the user plot more than one path together to confront multiple journeys.

Drawing routes is pretty easy, first of all it's necessary to define the objects that will be included in the plot:

- Start node
- Destination node
- Path or paths
- TRPs used for the predictions
- Eventual toll stations

7.1.1 Plotting Nodes

Nodes, being simple intersections don't represent anything on the map besides their existence and the potential presence of a traffic light.

They aren't plotted since there's just no need given that the lines representing edges already form angles which let the viewer understand that there's an intersection.

The only nodes that are actually plotted are the start and destination ones which also have custom icons.

7.1.2 Plotting Edges

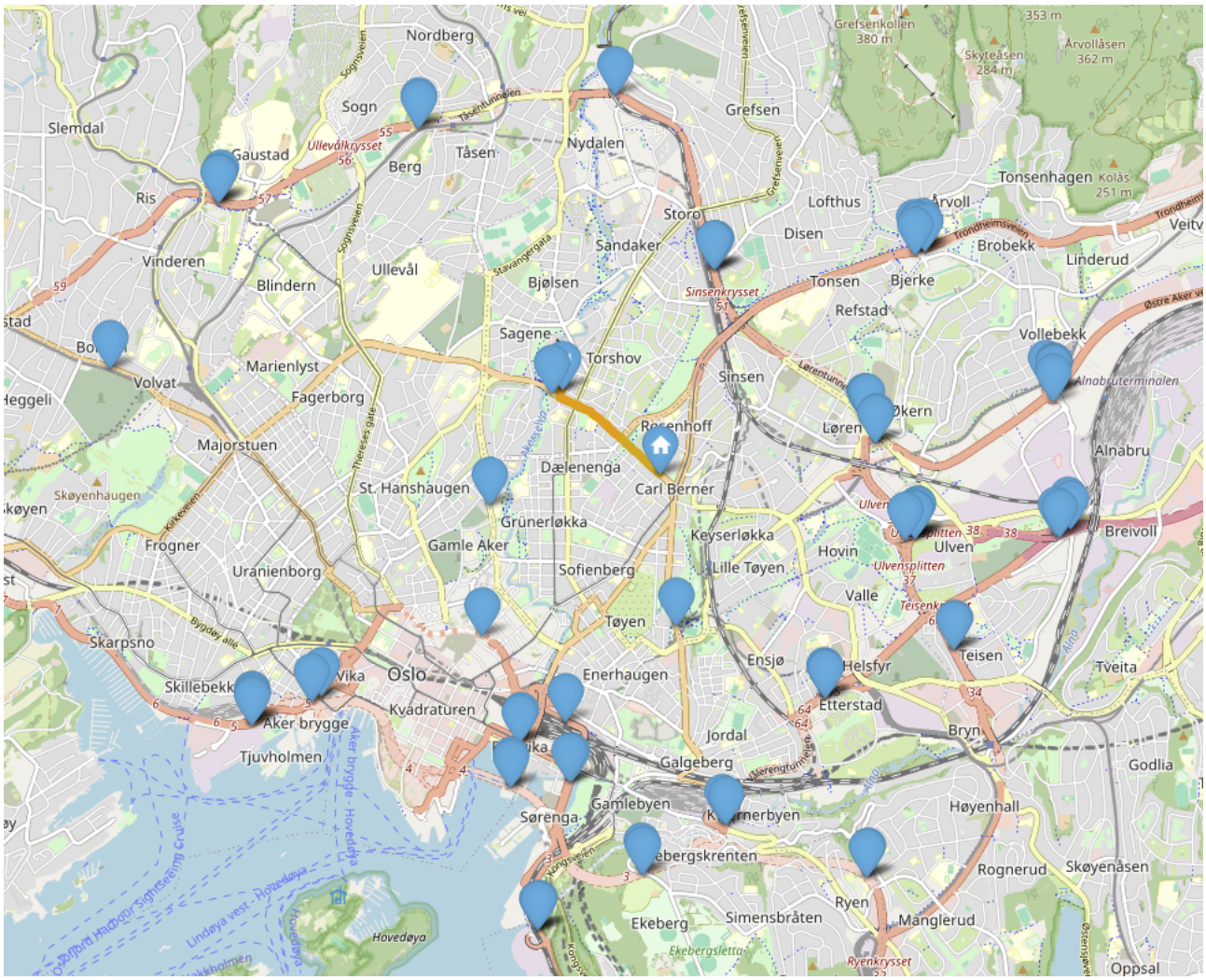
Plotting edges is fairly simple, they are just lines that connect two nodes, so taking each edge and the nodes it connects we can iteratively draw a unique a continuous line that stretches from the start to the destination node along the computed path.

Since each edge is composed of many sub-sections, each one of them is drawn individually and colored based on its traffic class.

7.1.3 Plotting Other Road Objects

Other road objects may need to be plotted on the map.

As for now, the only ones available (start and destination nodes, TRPs, toll stations), can be represented as points on the map; they are colored based on the object they represent and may have a custom icon to better indicate their function.



A plot of the path with coloring based on traffic conditions and TRPs marked as points on the map

7.2 Drawing Municipality Heatmaps

This feature is specifically thought to analyze the traffic conditions across a whole municipality.

The user can choose the municipality to take as target by using its ID and get a 2D contour plot (also known as heatmap) in return.

The idea is simple: if one needs to plan long distance routes across a city and doesn't want to rely on a single path obtained from the route-finding algorithm, a detailed representation of what it could encounter along the way is useful to elaborate a data-driven decision to avoid traffic congestions or bottlenecks.

Chapter 8 – Scalability and Future Developments

Scalability is a key-point in software architecture design and this project comes with great features, but also great improvements possibilities.

The main aspects to improve in the future of this project include of course being able to use it with more and more data overtime and process it within reasonable time.

This comes also with challenges and that's why more powerful technologies and frameworks should be taken into consideration when transformations and/or additions are made on a codebase.

Deploying directly on the cloud should be one thing to consider should the program transition to even greater data memory and processing speed requirements.

8.1 Memory Scalability

Memory scalability is among the greatest challenges when dealing with big data and machine learning, especially when using classic algorithms like Random Forest, Gradient Boosting and all derivates of these rather than neural networks.

This boils down just to how to split the learning phase in multiple steps to train the models on partitions of data iteratively.

Some distributed frameworks help with this task better than others given their highly active maintenance communities and the compatibility with other ones. This is why in the close future there will surely be a transition to PySpark and MLlib or similar technologies to improve the whole set of processes in the system by still maintaining lazy evaluation for data-intensive operations and potentially leave higher-level libraries like Pandas and Dask handle easier and smaller tasks.

This of course will make the system even more efficient and compatible with more models across a variety of machine learning libraries.

Improvements will surely include handling potential bottlenecks within data pipelines for even more efficiency in transformation processes and data-intensive tasks.

8.2 Computations scalability

Computations need to be fast and efficient and in case of big data handling: lazy.

This, as mentioned in the previous chapter on memory scalability, has been a major challenge when dealing with non-lazy libraries like Numpy, Pandas or Scikit-Learn.

Given the nature of most of the computations in the machine learning fields, a lot processing power is required and that's why being able to distribute the workload across multiple devices is crucial to improve processing times.

This is not only achievable with libraries like Dask, but also with frameworks like PySpark and others.

Potentially, if the data to handle becomes even more, a potential deployment of the system to the cloud would make everything easier when dealing with long-term finishing operations since they can be executed in remote data centers and not in home or small-company networks.

If convenient or necessary, highly complex and long transformations could be sped up by delegating them to high-performance languages like C++, Julia, etc.

8.3 Network Distributed Processing Scalability

As explained in the last part of the previous chapter, being able to distribute heavy workloads on a distributed network is crucial to improve processing time and speed and speed up the whole program execution.

As of now, the system uses Dask, which is a popular library for distributed parallel and lazy processing, but in the long run specifically-designed frameworks like PySpark scale better and are more actively maintained by their communities.

Apart from that, the expansion of the network where to distribute the jobs is just a matter of resources available, so the more the better.

8.4 Further Implementations of Asynchronous Processing

Asynchronous programming is a fascinating aspect of coding and can significantly increase performances when done correctly.

This project already implements asynchronous sections, mainly in database management and interaction.

Of course, to create asynchronous workflows, every component of them must be async and that's the main aspect to be aware of.

As a matter of fact, multiple implemented libraries aren't async at all, but can become by using some tricks, even though they aren't natively.

This also comes with adding complexity to the whole codebase, but when done right it speeds up things a lot.

Given all of this, the main sections where async programming will be implemented in the future will be the ones that aren't yet, for example:

- Loaders
- Preprocessing pipelines
- Road network (only specific processes)

8.5 Future Implementation of Parallel Predictions Processing with Dask

As discussed previously, Dask has a lot of potential, and one of the main functionalities is the capability to distribute the workload across many devices and parallelize it.

Parallelization with Dask[36] has only one, very technical and specific problem that involves pickling functions and methods.

The majority of the project's functionalities are based on object-oriented programming and this involves a large usage of bound methods (methods bound to the instance of a class).

This detail is particularly important since Dask cannot parallelize bound methods since they cannot be pickled, but only functions or static methods[37].

[36] Parallelization with Dask: https://examples.dask.org/applications/embarrassingly-parallel.html
[37] Static methods: https://realpython.com/ref/glossary/static-method/

This is why finding another way to parallelize those processes will surely be one thing to take into consideration to speed up the whole data prediction workflow.

Chapter 9 – Conclusions

Ultimately, this project has been a major challenge given all the different technologies to combine, different complex data types management, geographical projections, spatial data interpolation and more.

The need for this kind of systems is already satisfied by the greatest players on the market, but none of them has made their code available to the public as an open-source project.

Also, the data that's used by those players is incredibly expensive if one would want to use it through their APIs.

Given all of this, it's important to underline that this is just the start of something that can grow exponentially, hopefully with the help of a future community that can contribute to this initiative, possibly in the near future.

The result of the development of this project is a basic functioning program that enables a user to both analyze the Norwegian road network, investigate specific problems of it, predict future traffic data on multiple registration points and find the best path to go from a starting point to a destination one.

9.1 The Need for More Open-Source Data

Data is the fundamental part of data science and high quality one is even more necessary, now more than any time in the past history.

Of course, the ability to collect this kind of data in massive amounts is not easy and requires complex integrated systems (measurement points infrastructure, sensors, data pipelines, cloud storage, etc.) and to be economically sustainable some kind of business model is needed.

The only exception from this of course, is open-source, where the community (that may include companies as well) contributes to the sustainability of the project itself.

Other great successful open-source projects have achieved this, so this should be no less and give people even more freedom to choose which application they'd really like to use, especially if they care about privacy.

9.2 Data Privacy

Privacy in this field is especially important to avoid the possible localization of people and vehicles in space and time.

That's why traffic data must be treated with responsibility and adopting high standards of security, but at the same time making it accessible to those that need it for whichever (legal) purpose.

As for this project, there's no need to record people's movements to elaborate traffic forecasts, so the privacy aspect of it should be even easier to address.