Tackling Road Incidents Through the Power of Big Data and Machine Learning

CMPT 733 - Big Data Lab II

April 11, 2025

Rajan Grewal
Ibrahim Ali
Arshdeep Chhokar
Alex Sichitiu

Motivation and Background

With the increasing availability of historical traffic event data and the maturity of big data tools like Apache Spark, there's a timely opportunity to build scalable, data-driven systems for forecasting incident durations.
Road incidents such as collisions, stalled vehicles, or construction zones often cause major disruptions in traffic flow, leading to lost time, increased emissions, and reduced public safety. Knowing how long these disruptions can be crucial in decision making.

Who can use our Model

- Traffic Management Centers: Can use our model to make more informed decisions about rerouting and resource allocation.
- Navigation Platforms: Can provide users with more accurate ETAs by incorporating incident duration predictions.
- Government: Can use this insight to better understand the dynamics of road disruptions and optimize infrastructure planning
- Drivers: Can use the insights to make a decision, whether to wait it out or choose another route.
- Emergency Services: Can allocate resources effectively, plan faster response routes, or assess high-risk zones for accidents and hazards.
- Navigation App Developers: Can integrate real-time hazard/closure data into maps and routing features.

Related Work

Previous research and systems in the field of Intelligent Transportation Systems (ITS) have explored incident detection and traffic flow prediction. However, fewer efforts have been made in predicting the *duration* of individual incidents. Some studies have used linear regression or simple decision trees on limited datasets. In contrast, this project leverages large-scale event data with more sophisticated models such as Random Forests, Gradient Boosted Trees, and Neural Networks. It also integrates a full ETL and prediction pipeline using PySpark and TensorFlow, making it scalable and production-ready.

Related Work - Examples

Title: Incident duration prediction using a bi-level machine learning framework with outlier removal and intra–extra joint optimisation
Description: A study proposed a novel bi-level framework for predicting incident durations by combining baseline machine learning models.
URL: https://www.sciencedirect.com/science/article/pii/S0968090X22001589

Title: Predicting the duration of motorway incidents using machine learning
Description: A comparative study evaluated various machine learning algorithms to estimate the duration of motorway incidents on Ireland's M50 motorway.
URL: https://etrr.springeropen.com/articles/10.1186/s12544-024-00632-6

Problem Statement

Question 1: How long will a road incident last, based on the information available at the time it is reported?

This question is important for enabling faster and more informed decisions in traffic management, improving resource deployment, and minimizing the impact on drivers. However, accurately predicting incident duration is challenging due to the following reasons:

- Missing or Incomplete Data – Key fields may be missing or sparsely populated.
- Delayed Updates – Real-time information may lag or be updated inconsistently, leading to inaccurate initial data.
- Imbalanced Duration Distribution – Most incidents are short, making it hard for models to learn from longer-duration events.
- Feature Correlation – Some features may be correlated or redundant, which can affect model performance.
- Overfitting Risk – With many features and few long-duration incidents, models can easily overfit.
- Weather Conditions – Rain, snow, or fog may impact both the duration and the accuracy of reporting.
- Feature Correlation – Some features may be correlated or redundant, which can affect model performance.
- Nonlinear Relationships – Many features may have complex or nonlinear effects on incident duration.

Question 2: Where are most of the road incidents located and which should we prioritize?

This question especially concerns the services: emergency and construction. Leveraging these insights, the responders can provision their resources accordingly. There are some caveats involved with this, which include:

- Severity Misclassification – the severities rely on accurately reported data. If data is misreported, the clusters will not be accurate.
- Stale Data – The analysis involves historical and real-time data, and no distinguishment can currently be made, which can lead to mismatches.
- No Temporal Context – As it stands, there is no timestamp associated with the points, so users will not know when the incident actually occurred. A hotspot on the map may not be a real-time hotspot.
- Outliers – DBSCAN filters outliers, but these aren’t always to be ignored. There could be an important event here.
- Visual Overlap – Incidents with similar coordinates stack visually. While QuickSight shows counts on hover, dense urban areas may still be hard to interpret at a glance.

Data Science Pipeline

ML Prediction Pipeline

The data is pulled from *etl-ml.py* and transformed for machine learning analyses through Gradient Boosted Tree, Random Forest, and Neural Network machine learning models. These models are all stored locally.

1. Road event data is accessible via the DriveBC Open511 API. Active and archived events can be accessed via the same endpoint which returns at most 500 events at a time and accepts an offset parameter. Acquiring the data is relatively slow due to three key server-side issues. Hence, we could not accelerate the process using better hardware or parallelization.
    a. The API limits the request rate. Consequently, the API invocation script must pause for a few seconds when the server responds with error code 429.
    b. The events dataset includes “bad” records which if included in the API request, will cause the server to throw error code 500. The records must be skipped by the invocation script. To do this, the script would repeatedly halve the number of requested events when error 500 was returned in order to eventually find the faulty record (i.e. when the requested number of records was 1) and skip it.
    c. The server would take a long time to return results when large offset values were used (e.g. 500 records per 15 sec). We found no solution to this.

    After about 15 hours of cumulative runtime, we extracted approximately 580,000 events stored in 16 GB of data.
2. ETL: We ingest historical road incident data (JSON), remove duplicates, filter missing values, convert timestamps, and extract lat/lon coordinates from GeoJSON. Cleaned data is saved as Parquet.
3. Feature Engineering: We calculate the incident duration and extract features like latitude, longitude, event_type, severity, status, num_roads, and num_areas.
4. Transformation: Categorical features are indexed. All features are assembled and scaled using VectorAssembler and StandardScaler.
5. Model Training & Evaluation: Three models were tested: Random Forest (selected), Gradient Boosted Trees, and a Neural Network. Random Forest performed best and was persisted for deployment.
6. Prediction: New event data is processed using the same pipeline, and predictions are saved to Parquet for downstream use

Real-time ML Pipeline

This pipeline has two data sources. The first is pseudo-real-time data from Drive BC’s road incident endpoint, *https://api.open511.gov.bc.ca/events,* and the second is manually extracted historical data via the API. Each time new data is streamed in, clustering is performed with the new data and all of the historical data, allowing the new data to be seamlessly integrated into clusters with historical data.

1. Pseudo-real-time data stream: The first data source is requested via the DriveBC Open511 API through the *drive-events-function.py* Lambda in AWS. For the sake of cost

effectiveness, we do not have the data queried often, but rather have all of the logic of a real-time data stream set up.
2. AWS Kinesis: This is the data stream handler and utilizes *drive-events-function.py* as its producer. The data is consumed by our consumer function, *etl-events.py*
3. ETL: *etl-events.py* file, running as a script in AWS EMR, receives the data in a pseudo-real-time manner. We leverage spark streaming for this logic. The file is parsed through a series of conversions in order to be extracted from the data stream, and cleaned in a similar manner as *etl-ml.py*. Historical data ETL is done by *etl-historical.py* with similar cleaning logic.
4. Intermediate S3: The real-time cleaned data and historical cleaned data are both stored in *etl-tb/output*, as they have the same format.
5. Clustering: *cluster.py* is also stored as an EMR script and will automatically trigger through the *run-clustering.py* Lambda function in AWS upon any modification in *etl-tb/output*. There are two analyses here. Both are done with DBSCAN. The first is clustering based solely on latitude and longitude, and the second is based on latitude, longitude, and severity.
6. Final S3: The clustered data is stored in *clustered-data/coord-clusters* and *clustered-data/severity-clusters*.
7. Athena: The data is queried by Athena and a table is created for each clustering analysis.
8. QuickSight: The data from the Athena tables is pulled into QuickSight for visualization purposes. Hovering over a point will display the latitude, longitude, count of points (only applies to directly overlapping points), and severity level (for the severity analysis only)

Methodology

Exploratory Data Analysis

Raw JSON event data was extracted, transformed and loaded to Parquet using a simpler Spark ETL script than the one used for ML modeling. Initial analysis on the cleaned data was done using Spark DataFrame operations. However, since our operations were simple enough, we found that storing our data in AWS S3, importing it into an AWS Athena database and connecting that database to AWS QuickSight for visualization was a better solution. This was done to both explore the data for insights that could help ML modeling and to create a pipeline for simple road event analytics using cloud computing.

GBT/RF/NN

To tackle the problem of predicting road incident durations, we designed a scalable machine learning pipeline using a combination of big data tools and predictive modeling techniques. Apache Spark (PySpark) was the backbone of our pipeline, chosen for its ability to efficiently process large-scale data in a distributed environment. We used Spark MLlib for both feature engineering and model training, taking advantage of built-in tools like StringIndexer for categorical encoding, VectorAssembler for feature consolidation, and StandardScaler for normalization.

For modeling, we evaluated three approaches: Random Forest, Gradient Boosted Trees (GBT), and a Neural Network (MLP). Random Forest was ultimately selected as the best-performing model due to its lower RMSE and strong generalization capabilities. Gradient Boosted Trees served as a competitive baseline, while the neural network, built using TensorFlow and Keras, allowed us to experiment with deep learning on structured data. Although the MLP did not outperform tree-based models, it provided valuable insight into alternative approaches.

We computed several features from the cleaned dataset, including numerical indicators such as the number of affected roads and areas, as well as geographic coordinates. Categorical fields like event type, severity, and status were encoded for model compatibility. The target variable, duration, was calculated as the time difference between the incident's creation and resolution timestamps.

Each model was evaluated using common regression metrics: RMSE, MAE, MSE, and R^2 score. Hyperparameter tuning for the Random Forest model was done via cross-validation using Spark's CrossValidator and ParamGridBuilder. To ensure consistency and reproducibility, the complete machine learning pipeline—including preprocessing steps and the trained model—was saved using PipelineModel. This enabled us to apply the exact same transformations and predictions to new incoming data during deployment.

Clustering

For the clustering analysis, we decided to choose DBSCAN as the clustering algorithm. The reason we decided to do clustering is that it is a great way to visualize data, especially geographical data. Geographical visualizations are generally quite easy to interpret for the general public and many popular dashboards include a map visualization. Since this data is all British Columbia based, we can easily overlay data points over a map of British Columbia.

DBSCAN was chosen for several reasons as well. Unlike K-Means, for example, we do not need to predefine the number of clusters. Predefinition of clusters would be difficult with such a large dataset, and would only be made more difficult with a real-time data stream. DBSCAN excels at finding clusters of irregular shapes, which is exactly what we have in this case. It is also density-based, which is great for geospatial data, as much of the data is present in specific, dense areas. It's effective at filtering outliers, that is, the incidents which are far away from the rest of the incidents, allowing us to easily identify areas of lower priority. Using the Haversine distance, which is built into DBSCAN, it is easy to interpret geospatial data.

As far as application goes, there were two analyses done for clustering. Both are done in PySpark, with the actual clustering algorithm in a Pandas UDF, as PySpark does not have DBSCAN as a built-in function. We chose a minimum sample size of 10, and eps of 50. These larger, more relaxed constraints are optimal for our scenario as smaller parameters would yield an overwhelming number of clusters, making data difficult to interpret and less meaningful. The first clustering analysis was performed solely off of latitude and longitude. The clusters were calculated leveraging the haversine distance formula. The second clustering analysis was done

with latitude, longitude, and severity levels. Unknown severities are filtered out, as they can result in misleading data. The categorical severity data is mapped to integers, and DBSCAN is employed with euclidean distance, as we are now not only dealing with geospatial data, but also severity levels, rendering haversine a poorer fit.

Evaluation

Exploratory Data Analysis

We succeeded in creating a dashboard for simple historical event analytics. One notable observation was that the number of events would spike in the winter months. Upon analyzing event subtypes, we conclude that this is because the most common events are winter related (e.g. “PARTLY_ICY”, “SNOW_PACKED”). However, none of the observations we made were particularly helpful for ML modeling.

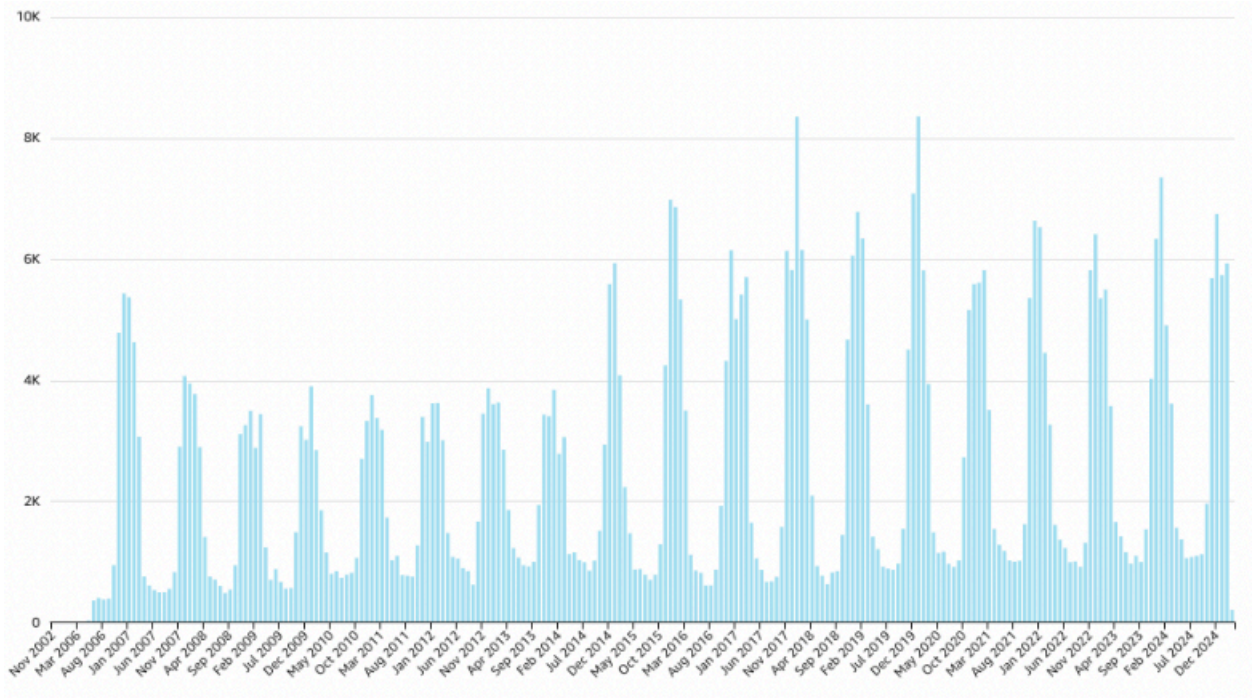

Figure 1: Number of events vs time grouped by month and year.

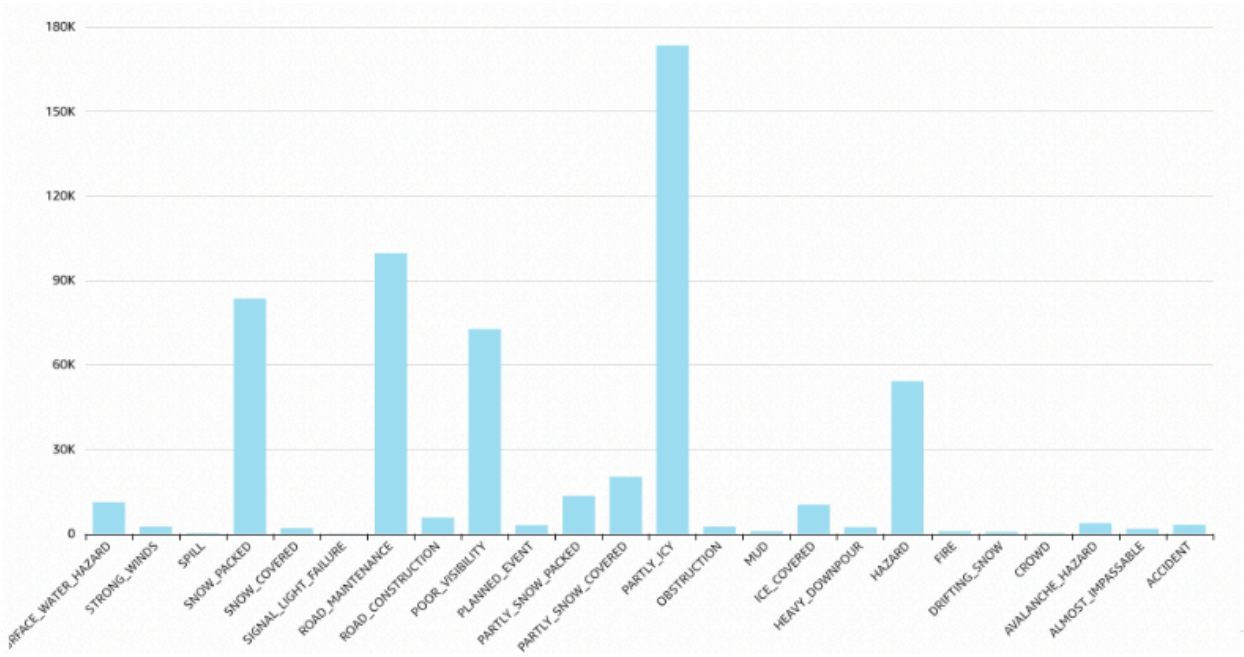

Figure 2: Number of events vs event subtype.

GBT/RF/NN

Our solution performs well and remains consistent across different data volumes. When trained on 25% of the dataset, Random Forest outperformed other models with an RMSE of 7.75 and an MAE of 1.35, slightly beating the GBT model. These metrics showed that the model was already learning meaningful patterns in the data.

When scaled up to the full dataset, Random Forest remained the top-performing model, with an RMSE of 9.55 and an MAE of 1.73—still slightly ahead of GBT. Surprisingly, error for both models *increased* when training on a larger dataset. This is likely due to the partial dataset not being a representative sample of the full dataset. The partial dataset was simply the first ~120,000 events from the API, which tended to be those earlier in time. It is likely that this subset was easier to model (i.e less complex) using the given approaches than the full dataset. Despite this error increase, the relative performance held steady, indicating that the model generalizes well.

The results make sense considering the features used—event type, severity, location, and number of roads/areas impacted—which are all intuitively related to how long an incident might last. The fact that performance remained stable with a larger dataset, combined with a reusable pipeline, makes the solution both reliable and scalable.

A.1 Performance on 25% of the Dataset
Random Forest Regressor

- RMSE: 7.75
- MAE: 1.35
- MSE: 60.09

- R^2 Score: 0.0213

Gradient Boosted Trees (GBT)
- RMSE: 7.75
- MAE: 1.34
- MSE: 60.13
- R^2 Score: 0.0207

A.2 Performance on Full Dataset
Random Forest Regressor
- RMSE: 9.55
- MAE: 1.73
- MSE: 91.24
- R^2 Score: 0.0164

Gradient Boosted Trees (GBT)
- RMSE: 9.58
- MAE: 1.74
- MSE: 91.78
- R^2 Score: 0.0106

Clustering

For the coordinate-based clustering, we are only clustering based off of latitude and longitude. This makes sense because many real-world events are spatially correlated. Generally, if many incidents occur near one another, they are usually caused by the same underlying factor, for example, an earthquake, or construction zone. We can gauge out which areas have a higher density of issues, and deploy services based off of these clusters. It would also be easy to assign a fixed number of services to certain areas. (Higher numbers to larger clusters, and smaller ones to smaller clusters).

For the severity-based clustering, we threw in an extra measure to the above analysis, the severity. The reason for this is priority. There may be a high-density of incidents in a given area, but the question we really want to answer here is: are these events something we should be worried about? It would not be optimal to deploy emergency services to something that does not require it. This third dimension assists in unfolding our priorities. Using this, services can be deployed in a much more optimal manner.

By leveraging both of these analyses, it greatly aids government services in terms of prioritization, volume, and location of deployment.

Data Product

GBT/RF/NN

Our data product is a prediction pipeline that estimates how long a new road incident will last. It takes a JSON file of incidents, processes the data using a saved Spark ML pipeline, and outputs predicted durations. The pipeline handles all steps—cleaning, feature extraction, encoding, and prediction—automatically. It uses the trained Random Forest model to generate accurate, consistent results and saves the predictions in Parquet format. This product makes it easy for users to forecast incident durations with just a single script and three inputs: the data file, the model path, and an output location.

Clustering

The pipeline for this takes in both real-time data, and historical data, and clusters off of that. The real time data, as previously mentioned, is only called when we decide to invoke it, to save on costs. The historical data is stored in our S3 and does not change. Both of these datasets undergo their cleaning processes and get stored in *etl-tb/output*. Clustering is done off of this data. The two analyses, coordinate and severity clustering, have their results stored in the S3 buckets *clustered-data/coord-clusters* and *clustered-data/severity-clusters*, respectively. The data is then queried from those buckets using Athena, and brought over to QuickSight. Visualizations are outlined below.

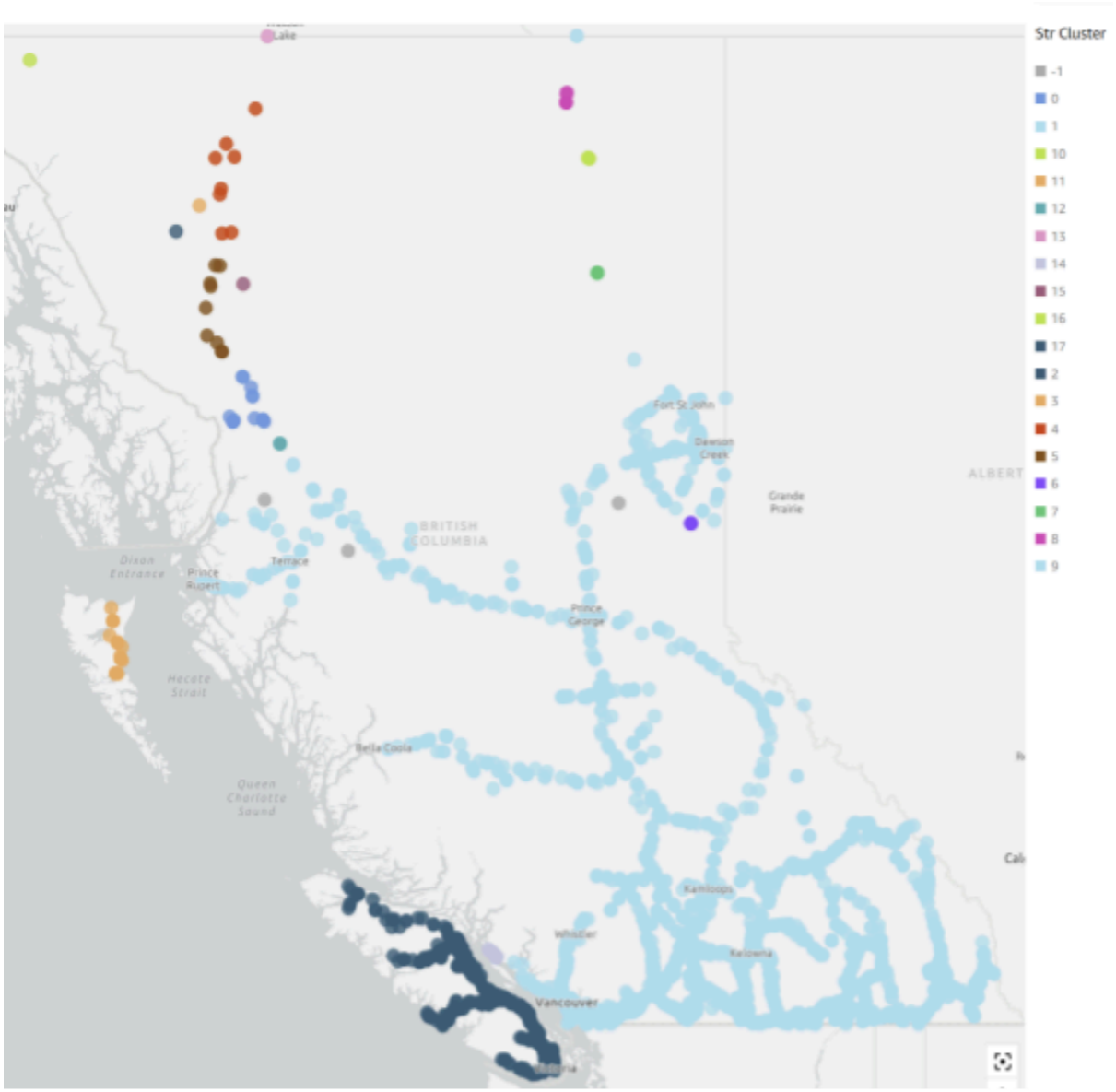

Figure 1: QuickSight coordinate cluster map. Highway patterns are picked up fairly well by DBSCAN.

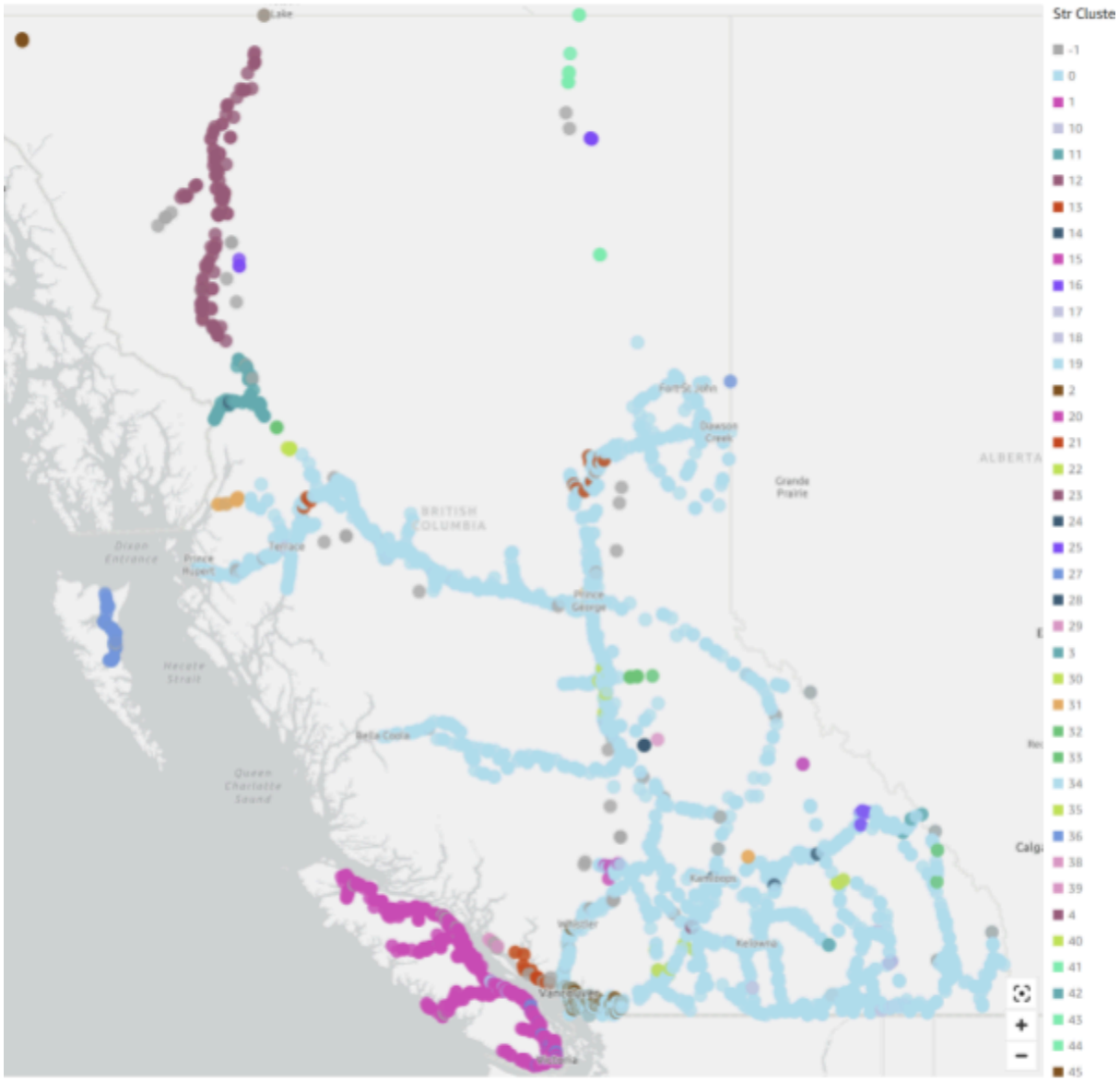

Figure 2: QuickSight severity cluster map *(i)*. We can see an increased number of clusters from the above figure due to the added feature.

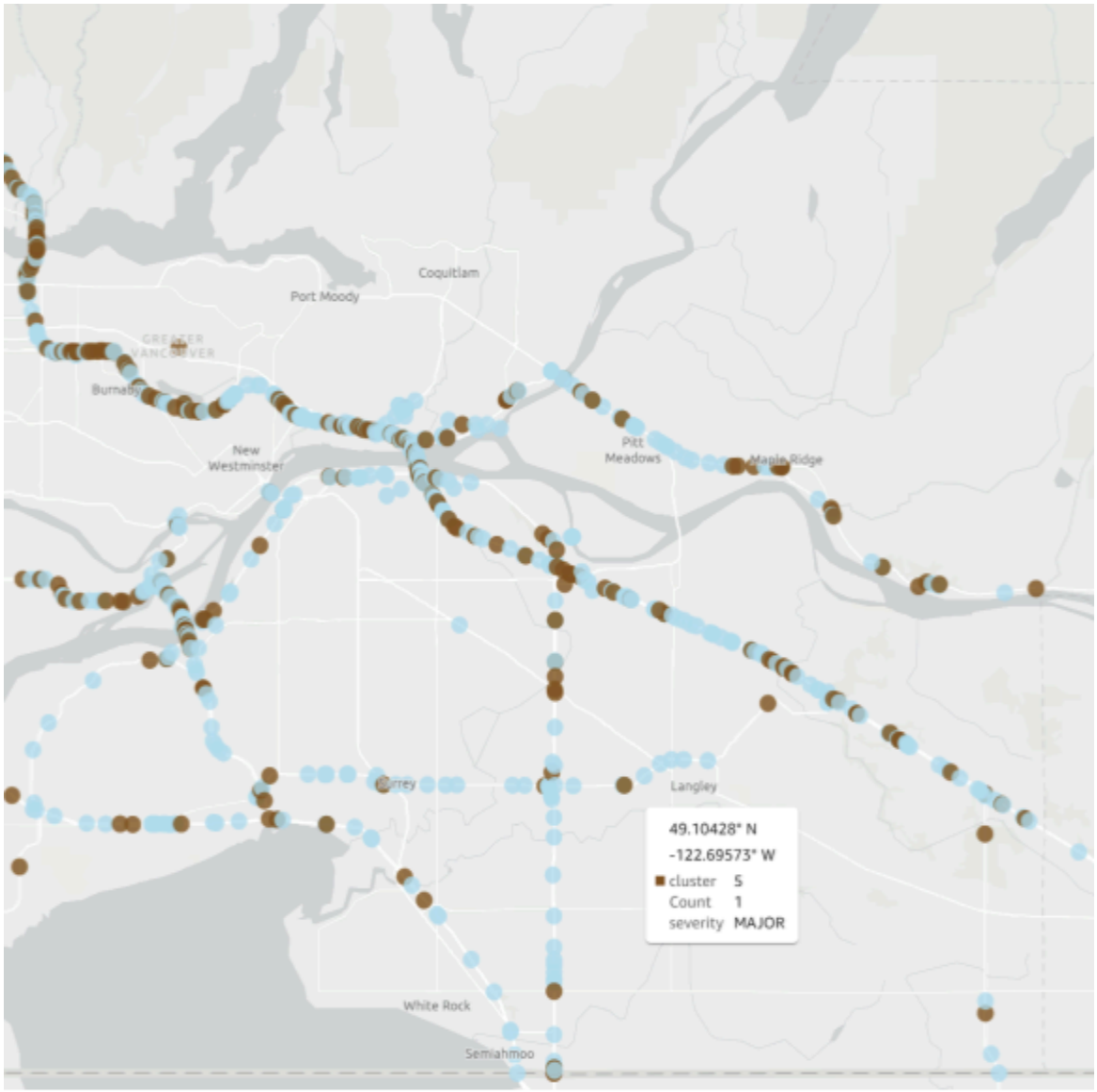

Figure 3: QuickSight severity cluster map *(ii)*. This map is zoomed into the lower mainland and displays two clusters despite a tight area, one for minor severities, and one for major.

Lessons Learned

Many valuable lessons were learnt during the development of this data and machine learning analysis pipeline. We used Amazon Web Services as our infrastructure, which proved to have a lot of setup with many components.

For the project, our initial instinct was to use Kafka as the messaging layer between our drive events lambda function and our EMR cluster for data processing. However, we encountered numerous issues related to Kafka's networking setup which involved configuring producers and consumers to operate across subnets and security groups in the cloud. We made multiple attempts to set up brokers, port accesses, and NAT gateways for internet access which all posed additional roadblocks in our progress.

This led us to seek alternatives, and we finally decided to use Kinesis which provided a much simpler setup with fully managed scalability and direct integration with other AWS services such as Lambda and S3. After implementing this change, we had our data pipeline up and running in a short amount of time which not only streamed data to our EMR job reliably, but also required very minimal configuration. This allowed us to focus solely on the data streaming logic, significantly improving our development speed.

Harvesting event data was more difficult than anticipated despite it coming from an API. In addition to the aforementioned error and latency issues, sometimes the API would return an empty event list despite there being more available data; an empty list was the termination condition for the script leading to early stops. This proved problematic as we had to log our progress in the dataset to restart an erroneously terminated script. Furthermore, we could not assume that a script left to run unsupervised (e.g. overnight) would complete data harvesting. As such, harvesting had to be done by running the API invocation script multiple times and combining the collected data. If anything, this emphasizes how you cannot assume any seemingly simple task is actually simple.

Debugging EMR scripts would take an enormous amount of time, since one must wait for the cluster to fully finish spinning up. An easier method we realized was to test locally. This also came with challenges, as the correct dependencies must be downloaded and passed via the command line. The development of *etl-events.py* was a very lengthy process. This file employs spark streaming, reading from Kinesis, and writing to S3 buckets, which turned out to require a whopping 11 jar file dependencies, which were downloaded from Maven.

Data streaming was something new to us, and turned out to have some interesting quirks. The connection to AWS Kinesis was tricky to wire, as there were many small differences amongst documentations we came across when it came to how to set up the stream reading logic. Eventually, the correct combination of instructions was found. There was peculiar behaviour

involving the dropDuplicates() function in PySpark, which would drop non-duplicate data from the stream, so we moved it further down the pipeline, after the data stream was completed.

AWS in general has differences when it comes to running files locally, and it did require us to run many tests within AWS itself. This caused our AWS bill to significantly increase, especially with the integration of real-time data streaming. We decided to shut down the streaming process for the sake of saving money. All of the streaming logic is there, so there was no need to skyrocket our bill just for the sake of data flowing in real time.

We initially had the clustering algorithm running in Spark, with pandas UDFs. It turned out that this was not a good idea. After some painful debugging, we realized that Spark was splitting the data into partitions, each of which only has knowledge about its own data, and not other partitions' data. This caused the clusters to make no sense at all. We converted the analysis to be done on Pandas, all in one process. This, unfortunately did slow down the algorithm significantly, but solved the issue.

Summary

This project was designed to predict how long road incidents will last and where the areas of high priority incidents are using both historical and real-time data. It will help services such as traffic management centers, navigation apps, and emergency services all make smarter and more optimized decisions. The project leverages a full ETL and ML pipeline, built on Apache Spark and deployed on AWS. This setup allows us to extract, process, analyze, and visualize the data.

The ML component begins with *etl-ml.py* which cleans the data. Three models, namely, Random Forest, Gradient Boosted Trees, and a Neural network are employed on this data. The Random Forest model delivered the best overall performance in terms of RMSE and MAE, and was selected for deployment.

The clustering component begins through the *drive-events-function* in AWS Lambda, which pulls the data and lets *etl-events.py* clean it out for storage in S3, where it is merged with the historically cleaned data in *etl-tb/output*. DBSCAN runs on this data and helps produce the QuickSight visualizations outlined in the diagrams.

Ultimately, the project delivered two robust data products: a road closure duration prediction pipeline and a clustering-based spatial analysis pipeline. Together, they offer practical, scalable solutions for analyzing, understanding, and managing road incidents more effectively in real time and over the long term.