[BDT-2021]
Traffic Project Bolzano report

Aurora Tumminello
Università degli Studi di Trento
aurora.tumminello@studenti.unitn.it

Leonardo Venturoso
Università degli Studi di Trento
leonardo.venturoso@studenti.unitn.it

1 INTRODUCTION

The purpose of our project was to design and implement a big data system able to predict in near-time the traffic associated with specific and crucial points of interest monitored by Bluetooth sensors in the Bolzano area. To achieve it, we considered the data available from Open Data Hub Südtirol, which takes into account also Merano and Trento's areas.

Initially, we focused on the exploration of Open Data Hub documentation, by using its existing Swagger API and Analytics Application. The service provides several datasets, including carpooling, bikesharing, roadweather, traffic and bluetooth stations. In our case, bluetooth stations was the ideal data type to accomplish our goal, since it provides station name and position, timestamp and number of vehicles that crossed it. Actually, there are some additional data attached, but they are redundant and referring to the conventions used (e.g. GPS coordinates, timestamp format and type of object, i.e. BluetoothStations).

The service also offers traffic data, which were not used in our project given the fact that they focus on highways traffic and the majority of them is missing. Therefore, we decided to use Bluetooth Stations data, also because the available data monitor the situation near the freeway tollbooths. We also evaluated other sources of data, such as Google Maps, Bing and Waze but most of them do not cover Bolzano area or need a fee for accessing them.

It would have been interesting to explore the type of vehicles that composes the traffic, dividing them into light and heavy, as the application of Open Hub shows, also in view of further analysis of city traffic patterns. Unfortunately, by exploring them, it is noticeable that data are incoherent (i.e. the sum of all of them overcomes the sum of all vehicles detected) and most of them missing for a specific category.

2 SYSTEM MODEL

In this section we're going to introduce the conceptual, logical and physical models, used to address this project in early stages. Next, we're going to present the different steps of our data pipeline, its functionality and its connections.

2.1 Conceptual, Logical and Physical Models

Starting with the three models above mentioned, here follows a generic presentation of them, to get an overview of the problem we're going to tackle:

- **Conceptual**: given a place and a timestamp, the problem is to predict the traffic in Bolzano (i.e. the number of vehicles detected through Bluetooth sensors stations) in near real-time. Stations are predefined places where sensors are arranged and their number increase gradually over time.
- **Logical**: from the conceptual model, two main entities seem to emerge. The first one is BluetoothStation, with details about every station (i.e. name, longitude, latitude), whereas the second one is Measurement and contains the number of vehicles in a given timestamp and in a specific station. Each measurement has a unique station, while each station can have multiple measurements over time.
- **Physical**: the API returns by default JSON files, which are then converted to a tabular form, in order to be saved in a relational database. The choice to adopt a relational structure is based on the fact that since we do not hold many features for each record (a maximum number of three columns only), thus keeping JSON files would be quite expensive for the total amount of data we will gather. For each entity described in the logical model, there will be a table.

2.2 Data Pipeline

Based on the previous three models, we designed a data pipeline divided into the following steps, as it can be seen it Figure 1:

2.2.1 *Data Collection.* Since historical BluetoothStation data are also available, we decided to organize data collection into two different phases, in order to scale up the data load at the ingestion phase. The first one is focused on collecting historical data (i.e. batch processing) from the first observation caught by Bluetooth sensors (i.e. January 2013) till this April, relying on Python requests that convert the JSON into a CSV form. While the second phase is about stream data (i.e. stream processing), namely data that go from the last data downloaded till two hours from the time of the request, due to the actual unavailability of data in proper real-time. This process is automated by using Github Actions, a tool for creating a virtual machine that executes a scheduled script every 30 minutes.

2.2.2 *Data Ingestion.* For both phases, MySQL is used, since a relational database is best suited for this type of data. We also evaluated other alternatives, to try different technologies, such as:

- *DynamoDB*, which belongs to Amazon Web Services. The idea behind this choice was to use a Lambda function connected with a Cloudwatch event in order to insert new data on a DynamoDB table at a scheduled time. Given its complexity and free tier limits, it was dismissed.
- *Redis*, which unfortunately accepts just one primary key and each measurement is identified by the couple station-timestamp.
- *Cassandra*, initially studied to augment query speed for the application, it was discarded due to its complexity and to the cost of data migration from SQL to CQL, on which this NoSQL database is based. Moreover, since our database is meant to be remote on Amazon Web Services EC2, no GUI is available to work with Cassandra, whereas MySQL Workbench offers a more intuitive and user-friendly interface.
- *MongoDB*, used at the end of the pipeline for saving model predictions. It was initially discarded for saving measurement data since document based, without a relational structure, and its storage limitation per collection (i.e. 512MB). A further depth on the critics and limits can be found in the section 4.

We finally decided to choose MySQL for its simplicity, familiarity with SQL and well-adaptation to our domain, despite there were some efficiency issues. In addition, MySQL is generally more used than previous NoSQL alternatives and therefore more documentation and support are available. MySQL database instance called `bluetoothstations` was created on Amazon RDS and connected to MySQL Workbench, therefore it is reachable with a remote connection, making data accessibility more flexible.

2.2.3 Preparation and Computation. Since the aim of the presented project is mainly to predict the traffic, the central part of the pipeline is dedicated to this objective. In particular, given the huge size of data gathered, Spark was used through Databricks community edition to train the model on historical data and to export model structure and weights in order to reuse them for predicting the traffic in the future. The model is composed by different Keras layers for neural networks, in particular the Sequential, the LSTM (Long Short Term Memory) and the Dense layers. The middle one, LSTM, is typically used for sequential data prediction, which applies perfectly to our case study.

Initially, we used simpler models, such as Logistic Regression and Decision Trees, but they provided low values compared to the actual data. Therefore, we focused on this simple neural network, on which we encountered different problems:

- As the model involves sequential data only, the timestamp feature was discarded, since the sequentiality of time is implicitly present in the model itself. Moreover, stations are encoded as numbers starting from 1, because the model does not accept categorical data. Initially, we used a LabelEncoder, but it was necessary to save inside an external file the encoding of each station, otherwise, at each initialization of the encoder, each place could receive a different label, making the training part useless by losing information related to the specific station.
- Data needs to be preprocessed such that in every row we consider data from different timings $(t, t-1, t-2, ...t-n)$. The algorithm proposed by Jason Brownlee [1] did not consider the sequentiality of data for each single station, whose data was mixed up with other places' traffic. We modified this algorithm by grouping data for station and by creating a n-sequential dataframe for each station. In the end, the latest prediction for each station was returned, given that LSTM returns prediction also for existing count values too.
- Converting all 70 subsets of the original dataframe (i.e. one for each station) into numpy arrays for the model is highly expensive, since Spark does not allow to take dataframe values directly and converting the $collect()$ result to numpy array is costly too. Since we noticed that Spark DataFrame are not optimized for these operations, we considered to predict future traffic in local, limiting Spark just to the initial training phase.
- Brownlee's algorithm originally scales all columns of data. Since we created a dataframe for each station, in order to avoid that the station label was scaled to 0 for all of them, we limited the scaling part just to the counter of vehicles.

Once the model structure and weights were properly saved, the workflow was transposed on a local python script, whose execution is automated through Github Actions, scheduled every two hours, because of Github minutes limit. The script itself iterates to get predictions for the following hours, considering latest data (i.e. last two hours available). Predictions are then saved inside MongoDB to be accessed from the application.

The decision of moving from MongoDB to MySQL is driven by our curiosity in experiencing different technologies, in particular a NoSQL one, to deal with data. In addition, through MongoDB Compass, some pipelines processes for querying and visualizing data were simplified.

2.2.4 Presentation. To provide a more complete and visually effective end result, we decided to create a web application. This allows to have an impactive and informative interface, that covers our Data Science path from data gathering to data presentation. Nonetheless, it is more pleasing to the eye and it communicates in a more effective way useful insights about the traffic in the Bolzano area that numerical values alone do not provide. Our application is hosted on Shiny server, an R package for building interactive web applications. In particular, *flexdashboard* is used to structure the application in four different pages with a Bootstrap behaviour, to make the code more readable and the website more flexible. We chose Shiny to host a web application without any cost and highly monitored, written in RMarkdown. Many examples of similar projects are available at the following link. The first page aims to accomplish the initial task of this project, showing the real-time traffic prediction, whose data are provided by the machine learning model described above by querying a MongoDB collection. The second page is entirely dedicated to stations, their position on the map, mean traffic per hour and their proportion for each moment of the day (i.e. Morning, Afternoon, Evening, Night). The third page is focused on the historical trend o traffic in the overall stations, where it is possible to select the date range and visualize the most busy dates and times. The fourth page shows some insights about the traffic, focusing on weekday, hour, month and season of the year, depicting less traffic during weekends, evening and nights and more traffic during hot seasons (i.e. Spring and Summer). Additional details about the application data are available inside the last page "About".

In order to adopt an easy-to-transfer solution we also decided to create a Docker image of our MySQL Database and python scripts. The creation of a database image was carried out by using an EC2 instance though a bastion host. More details are present in the READ.me file inside the shared repository.

3 IMPLEMENTATION

3.1 Data Collection

Regarding the folder *"py"*, this contains the python scripts required for data acquisition. Specifically, as previously reported, this phase was divided into two: one for the batch collection of historical data and one for near real time data.

First, we created a script `BluetoothStation.py` to manage the two entities that emerged from the logical model, i.e. one object dealing with the stations and one with the measurements. Next, the main idea here was to create a script called `download_history.py` with functions (`get_stations_details()` and `get_data_of_day()`) that could make a HTTP request to the site API using requests and obtain a list of the available Bluetooth stations using the BluetoothStation object, to insert inside the Station table, and a list of Measurement objects for a specific date range. As previously stated, this script was used to collect in batch historical data.

The next step was to create the suitable script for stream data collection, i. e. `download_real_time.py` which through the function `get_missing_data()` obtain data in a specific range that goes from the last data downloaded till the time of the request, despite the data publisher makes latest information available till 2 hours in the past from now.

3.2 Github Actions for automation

A specific repository has been created for this project on GitHub to periodically store multiple versions of the files and allow access to multiple users. Moreover, Github offers a service for executing automatically some scripts, based on `.yml` files inside the workflow directory. In details, it creates an ubuntu virtual machine with the latest python version installed, then specified requirements are installed and in the end the provided script is executed.

At the start of the project, we initially created a virtual environment to separate required dependencies by different scripts and files. Since some libraries require more than 100MB limit per file imposed by Github, we abandoned the idea of activating the virtual environment during a scheduled job, therefore about one minute per each execution is dedicated to the installation of the requirements.

There are other alternatives to the programmed execution offered by Actions, such as Amazon CloudWatch Events, Databricks jobs, Task scheduler on local computer etc. Most of them require a payment or aren't feasible (e.g. require the computer to be always on). A reasonable solution, despite has some limitations (i.e. 3,000 minutes per month) seems to be Github Actions, where we programmed real-time data to be downloaded every 30 minutes and traffic to be predicted every two hours. These timings were chosen as a compromise between minutes limitation and the necessity of gathering real time data.

Every single execution log can be accessed in the Actions section of the repository. Eventual fails of executions are notified via email to all contributors. As can be seen inside yml files, there are some environmental variables taken from Github secrets, which consent us to access sensitive credentials about database connection. These secrets are hidden to everybody, except for those who hold the configuration file.

3.3 Data Ingestion

The second macro-phase was to manage the amount of data separately. A script called `Database_Manager` has been created for this purpose. Two classes have set up in order to ingest data properly: `MySQLStationManager` `MySQLStationManagerAWS`.The first created in a local instance a database called `bluetoothstations` containing two different tables (one for the stations information and one for vehicle counter for each timestamp and station), while the second one created a remote instance of AWS RDS database connected with MySQLWorkbench.

To accomplish the last part, a standard connection (TCP/IP) was created on MySQLWorkbench, using as credentials the IP, user, password and the port on the server host. In order to guarantee remote access, the inbound rules of the instance have also been set up properly on the AWS RDS console. In this way, data can be managed both locally and remotely, depending on which class is called.

For our purpose, both historical data and stream data were entered remotely through the function `get_data_in range()`, which inserts the downloaded data in the proper table by calling inside the class `MySQLManagerAWS`. However, regarding near real time data ingestion, in order to optimise the input time, data were first transformed into a temporary CSV file, then inserted by means of a query and finally the content of the temporary file was emptied (but still existing because of the lack of writing permissions of Github) for future use.

3.4 Model

The following subsections describe in details how the model works, distinguishing the process into the training phase (held with Spark) and the prediction one (a continuous loop executed on Github Actions).

3.4.1 *Training.*

(1) Connection with MySQL DB by creating the JDBC connector;
(2) Opening the connection to the measurement and station tables inside the DB;
(3) Create temporary view, necessary to pass from scala to python;
(4) Creating Spark DataFrames in python, joining measurement with station codes for labelling the categorical feature to numerical;
(5) Train and test splitting based on the last 2 days, executing the following algorithms for both of them:
    - Create a subset of the original dataframe for each station;
    - Create a dataset with sequential data, returning also the necessary scaler to inverse_transform predictions;
    - Converting everything to NumPy array and split X and y;
(6) Creation of the model and fit on the training data;
(7) Defining a function to create a dataframe with actual predictions for all stations
(8) Export the model and its weights inside the DataBricks filestore, downloaded through URL.

3.4.2 *Prediction.*

(1) Initially take the latest data available and go backwards of two hours (note that latest data is NOT the actual DateTime, but nearly two or three hours in the past due to unavailability of data by publishers);
(2) Prepare one dataframe for each station;
(3) For each sub-dataframe, scale counter and express rows as a sequence of data at time $t, t-1, \dots t-5$. Note that we took 5 to not underfit the model to the latest data and to not overfit by considering too many past values of traffic, since it varies widely from an hour to another;
(4) Split features and past data (X) from the outcome (y). Past data may refer both to actual data and to their concatenation with predictions in further iterations;
(5) Import the model saved from previous iterations of the script and the initial training on DataBricks;
(6) Predict the outcome, make the inverse transformation of the scaler, map the label to the corresponding station and consider just the last prediction for each station, since the previous refers to an already predicted temporal sequence;
(7) Update the model files and insert the prediction database inside MongoDB, where each prediction for each station at a given timestamp is saved as a single document;
(8) Repeat all steps from 2 to 7, joining predictions to the original dataset in order to get predictions till 10 hours in the future with respect to the latest DateTime available in the database. This allows us to execute the script every two hours and still have predictions.

3.5 Application

RStudio offers a variety of connections with the most disparate databases, including MongoDB and MySQL. For the connection with MySQL, MariaDB connector was used, because it speeds up the entire application. Credentials are stored in a configuration file, which is not published on GitHub. The application naturally divides input (user input, as theme, date, station) and output (plots, tables) parts of every single page, providing an intrinsically interactive space for users. To enhance this feature, additional interactive libraries were used, such as Plotly, MapBox, leaflet and ggplot (then converted with ggplotly). A lack of our application is related to the street connections between different stations, therefore in the main traffic map where streets are not drawn. This is due to the fact that we do not hold a GeoJSON with coordinates of start and end of every single street related to every Bluetooth Station, therefore we limited our work to bubbles in correspondence of geographical coordinates of Stations.

Notice that shiny itself is slow whenever the application is started. To reduce the timing of loading of the contents inside the app, most of data is cached, such that the majority of them is preloaded in few seconds. Unfortunately, still, it is necessary to query the database in order to get real time and up to date information, which initially took about 30 seconds to execute.

For this reasons, we boosted the query process by introducing:

- An Index based on stations, such that data retrieval of the specific station is more efficient;
- Summary tables, which compute some statistics on the original data, such as summing or averaging observations based on stations, date, hour, weekday etc. Based on this reasoning, we created `time_group` (grouped by month, hour and weekday) and `station_traffic_pe` (grouped by station and hour).
- Triggers, activated whenever a new observation is inserted inside the db, that updates the summary tables by summing up or averaging the old values with new ones. Since the db is never modified and no data is erased, triggers are activated only during insertion.

The corresponding code is available in the *sql* directory in the Github repository.

4 CONCLUSION AND FURTHER DIRECTIONS

We tried to reach the goal of our project by creating a web application able to predict and display traffic in Bolzano's area. However, we are aware of some limitations of our work and that some improvements are possible. Specifically:

- More data could have been gathered from different data sources. We explored and analyzed different alternatives, such as Bing, Google Maps, Waze and minor traffic applications, but most of them require a payment, or specific tokens or do not provide information about the zone of interest;

- Script automation could have been scheduled more frequently, but due to the free tier offered by Github Actions and limitations of other services, we preferred to stay within the available resources range;

- MySQL may not be the best choice, considering existing alternatives, but it is certainly one of the most used and for our purposed it fitted quite well, considering additional structures added (summary tables, indexes, triggers, etc). Still, we evaluated different NoSQL alternatives, such as Cassandra and MongoDB, but we stayed along MySQL due to the remote connection availability, to the absence of storage limit (which instead is included inside Amazon RDS) and its simplicity;

- Regarding the model, as stated before, LSTM based models require past data to prepare temporally. Since it is intrinsically univariate (i.e. one single feature, which is the outcome), the data pre-processing algorithm was built *ad hoc*, considering not only count but also the station. This operation of subsetting according to the station and creating a t-n sequential dataframe is quite expensive, especially if we consider that Spark has no direct way of taking numpy array values. Moreover, since the model relies on past data, it cannot predict more data in the future if there are no updates in the data published by Open Hub (in fact, since we make a loop of predictions to get in the current timestamp, values tend to decrease from an iteration to another). Further improvements should consider different models that still consider the place, the time and eventually seasoning and weekdays;

- In the end, the latest critic to our work is related to the app. It could have been realized with different frameworks, for instance python based such as python dash, mainly because of shiny slowness and difficulty of reading the code. However, despite its limitation, we used it for spacing across different programming languages, libraries and frameworks, opting for a completely free technology.

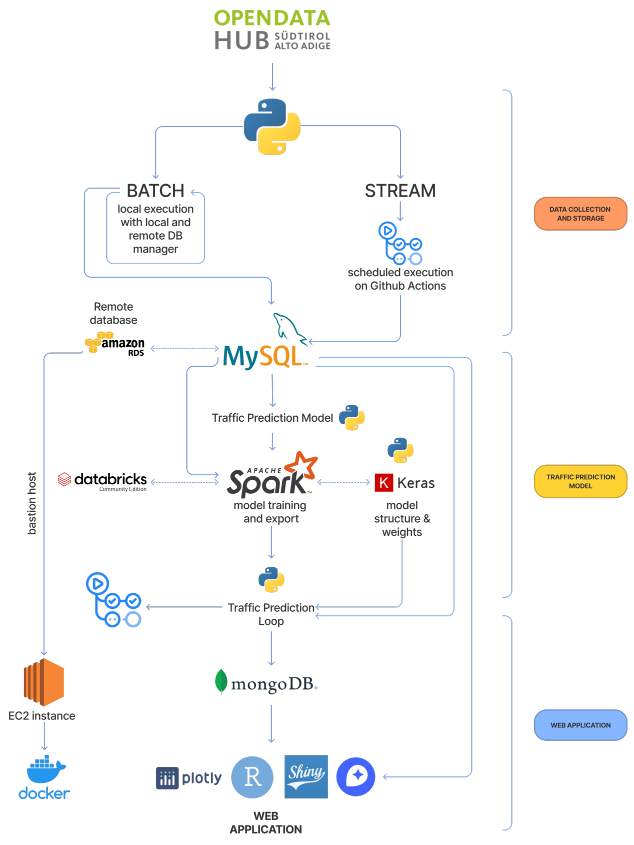
OPENDATA HUB SÜDTIROL ALTO ADIGE
BATCH
local execution with local and remote DB manager
STREAM
scheduled execution on Github Actions
Remote database
amazon RDS
MySQL
DATA COLLECTION AND STORAGE
bastion host
Traffic Prediction Model
databricks Community Edition
Spark
model training and export
Keras
model structure & weights
TRAFFIC PREDICTION MODEL
Traffic Prediction Loop
EC2 instance
mongoDB.
docker
plotly
R
Shiny
WEB APPLICATION
WEB APPLICATION

Figure 1: Traffic Data Pipeline

REFERENCES

[1] Jason Brownlee. 2017. Multivariate Time Series Forecasting with LSTMs in Keras. https://machinelearningmastery.com/multivariate-time-series-forecasting-lstms-keras/.

[2] Aurora Maria Tumminello Leonardo Venturoso. 2021. Traffic Bolzano Github Repository. https://github.com/AuroraArctic/Traffic-Project.

[3] Aurora Maria Tumminello Leonardo Venturoso. 2021. Traffic Bolzano Model Prediction (Training Phase). https://community.cloud.databricks.com/?o=1442008419935082#notebook/1440926709870068/command/1440926709870089.

[4] Aron Bengtsson Linus Aronsson. 2019. Machine learning applied to traffic forecasting. https://odr.chalmers.se/bitstream/20.500.12380/300031/1/CSE%2019-05%20CPL%20Aronsson%20Bengtsson.pdf. (2019).

[5] The ODH Team Revision. 2021. Open Data Hub Sudtirol Alto Adige. https://opendatahub.readthedocs.io/en/latest/guidelines/authentication.html#authentication-hub.

[6] Ihor Shylo. 2018. Big Data Dashboard with R Shiny. https://medium.com/@ihor.shylo/big-data-dashboard-with-r-shiny-bfa3090239ec.

[7] weide zhou. 2020. How to setup Github Actions to run my python script on schedule. https://github.community/t/how-to-setup-github-actions-to-run-my-python-script-on-schedule/18335/2.