Electric Vehicles usage in Südtirol

Giulio Mattolin
Università degli Studi di Trento
1^{st} Year Master of Science in Data Science
Trento, Italy
giulio.mattolin@studenti.unitn.it

Francesca Padovani
Università degli Studi di Trento
1^{st} Year Master of Science in Data Science
Trento, Italy
francesca.padovani@studenti.unitn.it

ABSTRACT

This report discusses the implementation and the structure of a big data system built to predict the usage and the status, for a given day and a specific time slot, of e-charging stations and plugs in the Italian region of Trentino Alto Adige. The predictive model has been constructed integrating different types of data, such as weather, season, geographic altitude of locations, that will be described in detail below. The service, through a simple interface, is capable of providing to a user, who could be either a private driver or a maintenance technician, with the possibility to predict the status of a station up to the next 14 days.
Keywords: energy, electric vehicles, plugs, stations

1 INTRODUCTION

This project has the ultimate goal of providing precise and real-time predictions of the status of occupation of e-charging stations from Trentino Alto Adige region. Our objective was thus to train a model with data about the past with respect to the usability of the stations and their plugs, benefit from it in order to help users in predicting the state of a plug in a specific day and hour and guide them in the choice of the station where to charge their electric cars.

In order to do so, we took advantage of an API offered by Open Data Hub Südtirol which returns historical data of the status of many stations in the North of Italy and of each single plug present in the charging points. We decided to make our analysis excluding stations positioned outside of the region of Trentino Alto Adige since they were out of our interest. In addition, we included data regarding weather, altitudes of the geographic area where the plugs were installed and also the seasonality of the year concerning tourism. For this, we assigned a value ranging from 1 to 4 to different periods of the year according to the subdivision provided by the *Alta Badia Dolomites* [1].
During the data exploration phase we inspected the data and in parallel consulted the documentation. In the Open Data Hub Südtirol API call we specified the field **pactive = True** to ensure that the stations were actively sending data to the Open Data Hub in the period of interest. Here below we list some critical observations and then explain which choices have been made after consulting the assistance service of the Open Data Hub. The information about the stations' state are stored in the **pmetadata.state** field and can assume various values such as TEMPORARYUNAVAILABLE, FAULT, UNKNOWN, OCCUPIED, ACTIVE, AVAILABLE. Furthermore, the **mvalue** field refers to the state of the plugs located in the station and can take on two different values which define the availability or not of the plug. We noticed that sometimes stations do not result functional but the related plugs are still available to the clients. In this case it was difficult to give a proper sense to this and thus we decided to contact Open Data Hub to receive some clarifications. The response we received was aligned with our assumptions; indeed, sometimes it is decided not to allow clients to charge their car at that station and in this case only the status of the station is updated and not the one of each single plug, leading to inconsistencies. This interpretation has to be taken into account by users when they receive the prediction from our model.

2 SYSTEM MODEL

2.1 System architecture

The entire pipeline is composed of different stages, as it can be noticed in Figure 1. During the collection phase, we retrieved the data of stations regarding the year 2019, considered more reliable and consistent than 2020 which was affected by the COVID19 pandemic. By calling the API, we iteratively got the data of every day of the year, preprocessed them in order to drop some unimportant information and transformed them from json format into a tabular format. After that, we stored each chunk of data regarding a day into a bucket in Google Cloud Storage as csv files. At the end of the process the bucket contained 359 csv files, not 365 because the API gave problems for some days, with a dimension spanning between 21 MB and 36 MB and containing the data of every station updated

© 2021 Copyright held by the owner/author(s). Published in Proceedings of the 24th International Conference on Extending Database Technology (EDBT), March 23-26, 2021, ISBN 978-3-89318-084-4 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

every 5 minutes. We then applied the same procedure for the collection and preprocessing of data about the weather and the altitudes.

In the next stage of the pipeline, we transferred the data to the data warehouse Big Query, in order to merge our files and perform further adjustments to have a complete view of them. At the beginning we had three different tables which contained data respectively of the stations, the weather and the altitude. Later, we used SQL language to query our tables to perform some tweaks on the stored data, such as changing the data types and extracting the hour from the timestamps, and ultimately to join the tables in a final and exhaustive dataset with nearly 30 million records. (image structure table)

After having collected and stored the data of interest in a warehouse, we made the decision of implementing our predictive model using a random forest as a learning method to predict the state of a station and its plugs. In particular, the random forest was trained with 100 trees on 75% of the entire dataset, while the remaining 25% was used as a test set to acknowledge the model's performance. Then, after the training, the model was saved in order to be exploited by our application.

For the final stage of our pipeline we decided to create a web application, in order to provide to the user an interface to interact with and see the prediction results. In support of our web application, we also implemented an instance of Redis, an open source in-memory data structure store, in which we stored the specifics of each station in order to easily retrieve them for the prediction and make them available also to the user. The implemented web application provides three functionalities: the first one allows to have an overview of all the stations available and their specifics; the second one permit to access the information of a specific station and its plugs; the third one allows to make a prediction on the status and availability of a particular station and its plugs on a given day and time. Finally, to optimize the deployment we took advantage of Docker with which it is possible to easily manage and run the final solution.

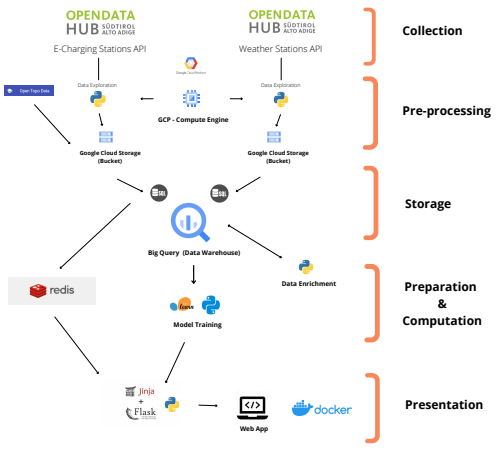

Figure 1: Pipeline

2.2 Technologies

Our project has mainly relied upon the services provided by Google Cloud Platform, which is a cloud computing platform that provides a series of cloud services including computing, data storage, data analytics and machine learning.

In the first stage, to collect the data from the API, we counted on a virtual machine created thanks to the Google Compute Engine to run the related Python scripts. This was done because the data to retrieve were many and thus the process took a lot of time to finish. For this, instead of running our scripts locally, we decided to take advantage of a VM and automate the operations. After that, to store the numerous csv files containing the data, we used Google Cloud Storage, which is an object storage that enables the storing of files of any format in containers called buckets. We implemented this step before the storage of the data in the warehouse in order to add a layer of redundancy to improve the fault tolerance of our system.

From Google Cloud Storage the data was then transferred to Google Big Query, which is a data warehouse that enables to integrate, manage and query data in a scalable way. The platform utilizes a columnar storage paradigm that allows for much faster data scanning as well as a tree architecture model that makes querying and aggregating results significantly easier and efficiently. Its use became very handy because it enabled us to import data directly from Google Storage in csv format and adjust our data, performing queries on them and building our final dataset.

Having the dataset on which to train the predictive model, we connected to Big Query and took advantage of the Python library scikit-learn to build a random forest with 100 trees. The final solution was then achieved by realizing a web application through the use of the Python microframework Flask. Moreover, since the predictive model requires the specifics of the stations and the user should be able to acknowledge what stations can be predicted and their specifics, we supported the web application using Redis. Indeed, through Redis it was possible to store the required data and efficiently access them in a key-value structure, in which the key was a station ID and the value was the information related to it and its plugs.

Eventually, Github offered us the capability for version control and hosting, while Docker supported us in the optimization of the deployment phase. As main programming language for all our scripts we used Python because of the full range of packages and libraries it offers to manage the API calls, the interactions with Google Cloud Platform, the training of the machine learning model and implementation of the web application.

3 IMPLEMENTATION

Here you can find the link to the *git repository* of our project which also includes the Docker image we used [2]. In the Readme file you will be instructed on how to activate it and set it up.

The above-mentioned repository contains a folder named **retrievals** which contains three scripts we developed to perform the API calls and pre-process the data received in json format. In the first script **e_stations_retrieval.py** we set the starting date at which we wanted to begin the collection of our historical data, which was the 1st of January 2019, and the end date at which to stop, the 31st of December 2019. With a for loop we iterated through each hour of a day and constructed our call in this constrained way, setting the p.active field equal to true. Through this we considered only the stations which were actively sending data to the Open Data Südtirol. The for loop let us iterate through the 24 hours of the day, retrieve the related data, preprocess it and append it to a pandas Dataframe. During the preprocess stage, data was converted from a json-structure to a tabular structure thanks to the function json_normalize of pandas and subsequently the unnecessary columns were dropped. At the end of each day we created a .csv file with the resulting data and we sent it to the bucket of Google Cloud Storage thanks to the library google-cloud. In the **e_weather_retrieval.py** script we applied the same procedure to retrieve historical data about the weather of two stations in Sudtirol province hour by hour. We decided to retrieve only the data about temperature and level of precipitation of two stations, located respectively on the eastern part of Bozen and on its western side, in order to split in two parts the region and afterwards assign to each e-charging station the data of the closest weather station. At the beginning we wanted to have a subdivision by four (North-East, North-West, South-East, South West), but then we discovered that the weather stations of Trentino were not active in 2019. As before, we used a for loop in which we retrieved the data, preprocessed it, in order to have a tabular structure and remove the unnecessary columns, and created a csv file for each day that was included into a bucket in Google Cloud Storage. In the **altitudes.py** we collected data about the altitude of the stations. To do this, we queried, thanks to the library pandas_gbq, the table on BigQuery in which we stored the data regarding stations and we got the geographic coordinates of each station saved in a Dataframe. Then we iterated through each location and by calling the *Open Topo Data API* [3] on every pair of latitude-longitude we obtained the relative altitude. Finally, we updated the Dataframe including the column of the retrieved altitudes and uploaded it as a table into Big Query. As a later passage, we joined this partial dataset containing the altitudes with the final table.

Moreover, in the repository you can find the code through which we have initialized and trained our model, a random forest classifier, in the file **random_forest.py**. We started by retrieving data from the final table in BigQuery and preprocessing them in order to work with the model. In this stage we changed the type of the columns related to the date into an integer and we converted the categorical variables into dummy variables. After that we split the data into train and test sets, with 75% of the total data on the first one and the remaining 25% on the second one, and we fitted our model with 100 trees on the training set by taking advantage of the library scikit-learn. Finally, we assessed the model accuracy on the test set and stored our model in a .joblib file.

In the python file named **redis_keys.py** we wrote some code which allowed us to create a Redis snapshot with data regarding the stations. Initially, we iterated through all the stations of our final table in BigQuery and constructed a dictionary in which we stored as keys the stations' IDs and as values their respective plugs. As a second step we configured the key-value pairs of a local Redis instance from the previously constructed dictionary using a for loop and dumping as values strings in a json-like format. Afterwards, from the command line, we used the redis command SAVE to take a snapshot of our full dataset at that specific point in time and produce a RDB file that we then used to initialize the redis instance that supports our web application.

In the folder called **docker_app** we developed our final solution and the definitive layer with which the users will interact in an active way. The service is dockerized using a Docker image. We have carefully built the docker-compose.yml file, the Dockerfile and the requirements.txt. In the docker-compose.yml we specified the two complementary services, namely our web application and the redis instance, with the directory containing the RDB file used to initialize the Redis instance; in the Dockerfile we included all the commands to assemble the image such as the starting Python image, the environment variables of Flask, the exposed port and a line to unzip the .joblib file; in the requirements.txt we listed all the required Python dipendencies. In the file **webapp.py** we implemented our Web Application. At this stage we have integrated the usage of Python for the backend side and HTML for the frontend, which can be found in the **template** folder, taking advantage of Flask. At first, we connected to the Redis instance and loaded the trained random forest model together with a .csv file containing the schema that it requires to run. We implemented three main functionalities which give the possibility to:

(1) have a look at the entire list of e-charging stations located in the region Trentino Alto Adige
(2) select a single station's ID and inspect its specifics
(3) receive a prediction for the availability of a station of interest and the plugs of its pertinence

When a user firstly accesses the web application a homepage is displayed thanks to the function home which makes use of the related html file and shows the links to the abovemetioned functionalities. The first one, implemented in the api_all function, returns a table with all the e-charging stations and their specifics. It does this by retrieving the data from the Redis instance with a for loop, fitting them in a pandas Dataframe which is then passed to the totable.html template that presents the results. The second one, implemented in the api_id function, performs both a GET and a POST request. The first one hands back a small form which enables the user to insert the id of the station of its

interest, whereas the second one returns a table with the details of the selected station and its plugs following the same procedure adopted by the previous function. The third one, implemented in the api_prediction function, performs once again two types of requests. The GET request permits to display the form in which to insert the id of the station and the date of the prediction, divided by year, month, day and hour, using the date of the current day as default when the page is opened. In the POST request we start by getting the data provided by the user and then retrieving from Redis the information relative to the station of interest. Furthermore, with the given date, we gain the weather forecast data, by calling the *Weather API* [4], and incorporate the adequate seasonality, using a Likert scale ranging from 1 (off-season) to 4 (peak season). Subsequently, the obtained data are preprocessed in order to be reshaped in the same format in which the model has been trained and therefore it expects as input. Through a for loop iteration we select the plugs related to the given station and perform a prediction for each of them. Eventually, we fit them in a Dataframe which then is passed to the totable.html template, which shows the results to the user.

4 RESULTS

Our ultimate result is a web application that aims at assisting the clients of the service in the choice of the station in which to charge their electric cars batteries in the Trentino Alto Adige region. The service can be run through its docker image which has a final size of 5 GB and builds the service from scratch in ... s. To make the download of the package on GitHub more lighweight we decided to zip the .joblib file related to the model, since it weighed 3.5 GB, and then unzip it during the building process.

The outputs of the three functionalities in the web app are tables that return the useful information according to the user's requests. The table that shows the entire list of stations resambles the schema of our dataset stored in BigQuery and contains information returned by the API which sometimes could result in being incomplete, e.g. the column named Category contains some empty values. Eventually, the main result of our service consists in a table including the predictions of all plugs which refers to a station, with both the predicted plug state and station state together with the specific date and time. During the development of the web app we mostly focused on the functionalities compared to the aesthetics of our interface.

5 CONCLUSIONS

Our system is limited with respect to the fact that we have based our model on data which refer only to one year, i.e. 2019, and are not receiving updates with newly and recently collected data. In order to account for this situation and the case in which historical data were not available, we should have collected data in streaming and continuously incorporated them in our database. This could have been done by integrating a Pub/Sub service, which can be set as a messagging-oriented middleware in the retrieval between the API and the Google Cloud Storage bucket and in the data transfer between the bucket and Big Query. Moreover, our predictive model has a very high accuracy, but this is due to the already stated reason that the historical data of 2019 on e-charging stations' status, which are at the basis of our learning algorithm, were very stable and sometimes were not changing at all. In addition, as we have already extensively discussed in the introduction section and as you can clearly detect from the figure above, it could happen that for some stations you will observe a TEMPORARY UNAVAILABLE state but at the same time its plugs will be available. This is counter intuitive and the clients should be instructed, as we already pointed out, not to take into consideration that station for charging their private car. Indeed, as the Open Data Hub Sudtirol underlined, in these cases only the station state should be taken into consideration.

REFERENCES

[1] Alta Badia Dolomites. Periodi di stagione in alta badia. https://www.altabadia.org/it/hotel-dolomiti/periodi-di-stagione/periodi-stagione-alta-badia.html. Last accessed on 13/07/2021.

[2] Francesca Padovani, Giulio Mattolin. Bdt-project. https://github.com/GiulioMat/BDT-Project. Last accessed on 13/07/2021.

[3] Open Topo Data. Open topo data. https://www.opentopodata.org/datasets/etopo1/. Last accessed on 13/07/2021.

[4] Weather API. Weather api. https://www.weatherapi.com/. Last accessed on 13/07/2021.