

PV-Predictor

Florian Seif¹, Sukhdev Chaman²

¹florainalbert.seif@studenti.unitn.it

²sukhdev.chaman@studenti.unitn.it

<https://github.com/mission-jupiter/Solai>

Abstract— Our project focuses on the creation of a predictive system for the power-production of photovoltaic systems. Using weather conditions as input, the system is able to predict the power generated by a solar panel over the next 48 hours. The results obtained demonstrate good precision in the prediction of solar energy production, allowing better energy planning and management.

Keywords— Solar energy, Photovoltaic panel, Prediction

I. Introduction

Our project involves developing an application that provides accurate predictions for the power output of a solar panel in the next 48 hours following a request. The Application will utilize the location of the panel and weather forecasts as inputs for a linear regression model, enabling the generation of highly reliable predictions.

II. System Model

A. System architecture

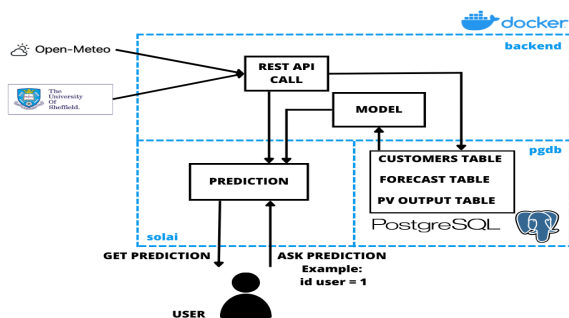


Fig. 1 System architecture

Our system architecture consists of three docker applications, designed to provide accurate predictions for the power output of a specific photovoltaic panel. All containers communicate in the same network and can therefore access the database.

- **pgdb**: A Postgres database to store weather forecasts, the PV outputs and customer information. Because the data has a clear predefined structure a SQL database fits the task. The database contains three tables *app.forecasts*, *app.pvlog* and *app.customers*, where *app.customer.id* is a foreign key in the other tables, because all forecasts and pvlogs are bound to a specific user, their pv system and their specific location.
- **backend**: An application to initialize the database. Calls the PV-API to receive the last 365 days of PV outputs of specific users and stores the data in the table *app.pvlog*. Populates the table *app.forecasts* with historical weather forecasts, that are stored in .csv files. The folder *weather_forecasts/* stores those forecasts and acts as a local data lake. Should be replaced and for later project stages should be replaced by S3 or other data lakes.
- **solai**: lightweight container that uses the unique *user_id* to call the pretrained machine learning model and calls the Weather Forecast API. Then makes a prediction with this data and displays the information to the user.

B. Technologies

In summary, our system relies on containerization of our application for efficient deployment. Docker containers provide lightweight, isolated environments that simplify application deployment, scalability, and portability while minimizing resource usage - and in comparison to other industry options are free to use. It allows us to run our application consistently across different environments and systems¹. The three containers previously mentioned use Python and Postgres and utilize publicly accessible APIs as data sources. We employ a Postgres database to store the collected data, which contains weather forecasts and real historical PV output. Postgres is a lightweight and broadly used SQL database, which due to the nature of our structured data is a perfect choice to store our data.

Because weather forecasts are always time sensitive we decided to store the results of our API calls as .csv files in the *weather_forecasts/* folder, which acts as our „data lake“.

By using this data, we construct a linear regression model that generates predictions based on current and future weather forecasts obtained from an integrated weather API.

To get our data, we use two publicly accessible APIs. The first is open-meteo², from which we call the data regarding the weather forecast that serve first to train our model and after to predict the output with the statistical model. The open-meteo API gives the weather forecasts (see Tab.1 for the variables called from API) for every hour of the coming days, so this data will be stored in a table where each row is the hour of the forecasts and the

columns are the all weather variables, the time of the API call and the hour of the forecasts; We decided to store different calls made at different times to train the model and understand the error of the forecasts. The second API by the University of Sheffield provides data on the outputs of different photovoltaic systems³.

All the data is stored in PostgreSQL, as we made the decision to utilize an SQL database instead of a NoSQL database. This choice was driven by the fact that our data can be organized and stored within a well-defined structure, specifically in tables. Given the homogeneous nature of our data, exploiting the ACID properties of an SQL database became highly advantageous. By doing so, we gain the benefits of ensuring data integrity, transactional consistency, and overall reliability throughout the system⁴.

To build and train the model, we combine data on past weather forecasts and the power outputs of a given photovoltaic panel.

The variables of the forecast	Description
temperature_2m	temperature at 2 m
relativehumidity_2m	relative humidity at 2 m
surface_pressure	surface pressure
windspeed_10m	wind speed at 10m
windspeed_80m	wind speed at 80m
windspeed_120m	wind speed at 120m
windspeed_180m	wind speed at 180m
winddirection_10m	wind direction at 10m
winddirection_80m	wind direction at 80m
winddirection_120m	wind direction at 120m
winddirection_180m	wind direction at 180m
direct_normal_irradiance	Direct solar irradiance as average of the preceding hour on the horizontal plane and the normal plane (perpendicular to the sun)
direct_normal_irradiance_instant	instant direct solar irradiation

Tab.1: variables called from open-meteo API

¹ V. Sharma, H. K. Saxena and A. K. Singh, "Docker for Multi-containers Web Application," 2020 2nd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA), Bangalore, India, 2020, pp. 589-592, doi: 10.1109/ICIMIA48430.2020.9074925.

² Open-meteo API.

Available: <https://open-meteo.com/>

³ Sheffield solar API.

Available: <https://www.solar.sheffield.ac.uk/pylive/>

⁴ de Oliveira, V.F.; Pessoa, M.A.d.O.; Junqueira, F.; Miyagi, P.E. SQL and NoSQL Databases in the Context of Industry 4.0. *Machines* 2022, 10, 20. <https://doi.org/10.3390/machines10010020>. Academic Editor: Xiang Li

III. Implementation

The implementation of our logical model consisted in the creation of one codebase that resulted in three docker containers.

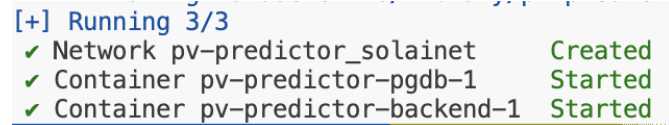
These services are used to manage different tasks of data collecting, initialize the database, training the model and getting the predictions. The codebase can be found in the repository: <https://github.com/mission-jupiter/Solai>

We decided to use three services in our docker application, each of those represents three different containers in our application. Those 3 services are: *pgdb*, *backend* and *solai*. All three services are connected to the same network, allowing communication between services, but because they use the same codebase the implementation details are not bound to specific containers and can be adjusted easily.

The codebase is structured in the following logic that explains quite well how the implementation is managed:

```
/
weather_forecasts/
  1/
    2023-01-01 01:01:00.csv
  Dockerfile-backend
  Dockerfile-prediction
  docker-compose.yml
  trained-model.pickle
  app.py
  run_prediction.py
    backend/
      db-utils.py
      ml-utils.py
      api-utils.py
      init-db.py
      init-db.sql
```

With the command *docker compose up -d --build pgdb backend* we start up our database and populate it with data.



```
[+] Running 3/3
✓ Network pv-predictor_solainet      Created
✓ Container pv-predictor-pgdb-1      Started
✓ Container pv-predictor-backend-1   Started
```

Fig 2: docker compose up -d --build pgdb backend

The initial database structure is defined in *init-db.sql*, which is needed by *pgdb*. The main functionality of the *backend* container is now to load all the *.csv* files stored in *weather_forecasts/* into the database. We need to store the forecast results in a data lake like manner, because due the locally run nature of the database the information is lost when we shut down the container.

The database connection is managed in *db-utils.py*, which offers all the functionality you connect to the database, execute sql queries, read data from the database to pandas DataFrames or to write a pandas DataFrame to a table. The *api-utils* offers classes to connect to the two APIs introduced before. When we call weather forecasts they automatically are stored as *.csv* files, for later use, pv data is just stored in the database.

The *ml-utils* contain the functionality to train, evaluate and use a Linear Regression model with *scikit-learn*. We decided to keep as an independent variable only the variable concerning "direct_normal_irradiance_instant"(DNI), since our task of the project is to focus on the architecture of the application and not on the part of machine learning. We explicitly decided on the DNI variable because analyzing the training data with a correlation matrix (Fig. 4), the solar irradiance variable is the most influential variable on power output.

As described above we build and run the database and the backend first. The database gets initialized and populated and the regression model gets trained with the available data. The database keeps running locally.

Afterwards we can build and run the prediction. With `<user_id>` we refer to the unique identifier of a specific user, that the user can use to access his pv forecast. Later on this could be implemented as an API with authentication to ensure data privacy.

Docker compose build solai

Docker compose run -rm solai `<user_id>`

```
~/Documents/PV-Predictor on main i3 !2 docker compose run --rm solai
[+] Building 0.0s (0/0)
[+] Creating 2/0
✓ Container pv-predictor-pgdb-1 Running
✓ Container pv-predictor-backend-1 Created
[+] Running 1/1
✓ Container pv-predictor-backend-1 Started
[+] Building 0.0s (0/0)
```

time	forecast	pred
2023-06-26 19:00:00	206.1	14.424315
2023-06-26 20:00:00	0.0	-0.759648
2023-06-26 21:00:00	0.0	-0.759648
2023-06-26 22:00:00	0.0	-0.759648
2023-06-26 23:00:00	0.0	-0.759648
2023-06-27 00:00:00	0.0	-0.759648
2023-06-27 01:00:00	0.0	-0.759648
2023-06-27 02:00:00	0.0	-0.759648
2023-06-27 03:00:00	0.0	-0.759648
2023-06-27 04:00:00	0.0	-0.759648
2023-06-27 05:00:00	0.0	-0.759648
2023-06-27 06:00:00	55.9	3.358661
2023-06-27 07:00:00	351.5	25.136340
2023-06-27 08:00:00	580.5	42.007411
2023-06-27 09:00:00	679.2	49.278917
2023-06-27 10:00:00	697.5	50.627129
2023-06-27 11:00:00	726.6	52.771007
2023-06-27 12:00:00	792.4	57.618678
2023-06-27 13:00:00	784.6	57.044030
2023-06-27 14:00:00	702.0	50.958657
2023-06-27 15:00:00	642.0	46.538289
2023-06-27 16:00:00	565.3	40.887585
2023-06-27 17:00:00	570.8	41.292785
2023-06-27 18:00:00	439.8	31.641649
2023-06-27 19:00:00	260.6	18.439483

Fig 3: Solai Output

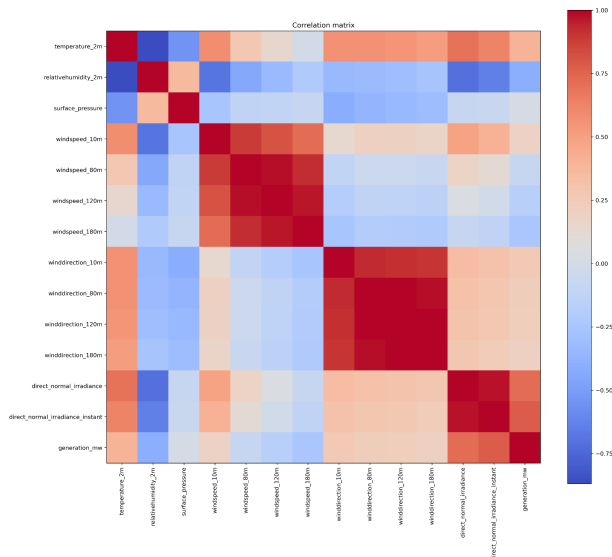


Fig. 4: the correlation matrix of our variables

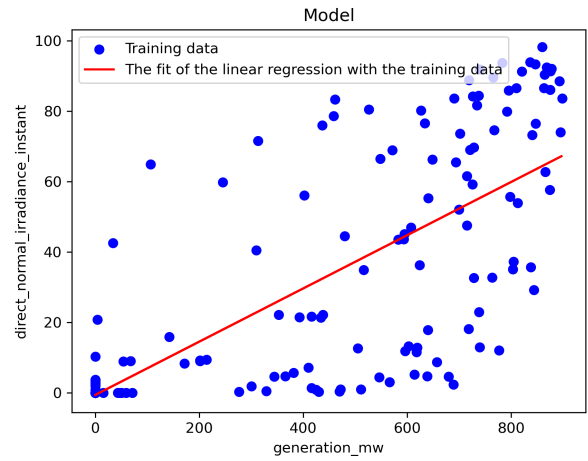


Fig. 5: The fit of the linear regression model with the training data

IV. Results

As a result, we have managed to create a working application that operates smoothly, with all the services collaborating harmoniously with each other. The database is efficient and can be used locally, knowing that running the database locally is not the optimal choice. In the future, for further updates, we will be able to host the database remotely, allowing for greater scalability and accessibility.

On github, you can find our application ready to run.

v. Conclusions

The most important task of the project was application architecture, but the model can be improved.

As mentioned before, the statistical model has been trained on an independent variable to keep the model less complicated since the project asked to focus more on the application architectures. With further and more detailed statistical analysis more variables can be introduced to reduce endogeneity or the whole model architecture could be changed to a neural network or maybe more applicable a LSTM. Because forecasts get less accurate the further they are away, this historical component could fit a LSTM very well to reduce the error between the weather forecast and real weather, which due to a first analysis, averaged over our data resulted in about a 10% difference.

Other than that the database should be hosted remotely and not locally. Once because of data loss and to make it widely accessible.

Another point to be improved is to make use of industry wide solutions for data storage like S3 or other data lakes, that can store large unstructured data amounts for our weather forecasts.

REFERENCES

- [1] V. Sharma, H. K. Saxena and A. K. Singh, "Docker for Multi-containers Web Application," *2020 2nd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*, Bangalore, India, 2020, pp. 589-592, doi: 10.1109/ICIMIA48430.2020.9074925.
- [2] Open-meteo API.
Available: <https://open-meteo.com/>
- [3] Sheffield solar API.
Available: <https://www.solar.sheffield.ac.uk/pvlive/>
- [4] de Oliveira, V.F.; Pessoa, M.A.d.O.; Junqueira, F.; Miyagi, P.E. SQL and NoSQL Databases in the Context of Industry 4.0. *Machines* **2022**, *10*, 20. <https://doi.org/10.3390/machines10010020>. Academic Editor: Xiang Li