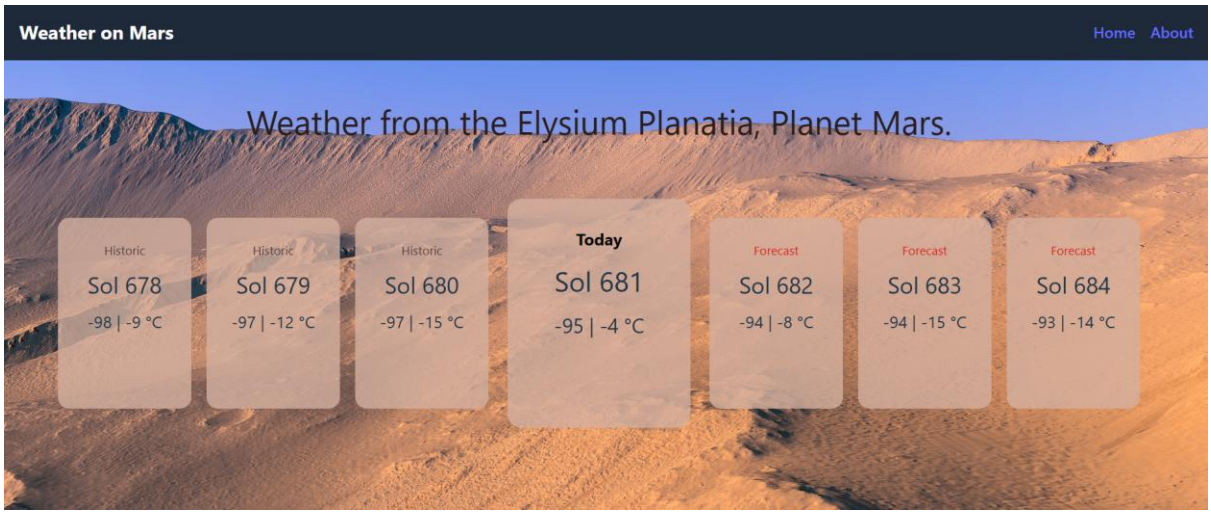
Mars Weather System – Report



In this project I underwent full-stack development of a responsive React.js app using Vite and Tailwind CSS to display weather data from Mars' Elysium Planatia retrieved from NASA's InSight API. In the backend I also included temperature forecasting functionality in Python, by first training a time series machine learning model using web-scraped historical data. This was a personal project done to develop and test my skills. Unfortunately, the Mars InSight lander stopped transmitting data in 2021, so the website does not provide completely live data. However, it is a proof of concept and functions as if the latest data retrieved is from the present. Development was done using Flask's development server and React's build output server.

If I were to develop further I would look into cloud service, but for now it serves as a working demo. I have written this report to communicate three key technical challenges that I faced.

Building a pipeline for historical data.

Whilst retrieving the weather data from the API was a straightforward and well-documented process, this only provided data from the past 7 sols (Martian days). To build an effective machine learning model, I would need the full archive of data collected.

It turned out that data was stored in a series of files, with instrument readings given for every few minutes rather than the clean summarized data given by the API. Thus, I needed to scrape and then reduce all this data. I used Python's BeautifulSoup to loop through subfolders and then files to retrieve minimum, maximum and mean temperatures. This also involved inspecting data formats and storage methods – e.g. evaluating derived rather than raw data, which was in a slightly nicer form. A small hurdle was that I was confused about which column could possibly be the temperatures until I realized that they were stored in Kelvin. See backend/scrape_temps.py. Data was collected from https://atmos.nmsu.edu/PDS/data/PDS4/InSight/twins_bundle/data_derived/.

I wanted to collect pressure data so tried a similar process; however, the web scraping was far slower – I calculated that it would take 20+ hours to complete. I'm not sure why this was so much slower than the temperature data, which was quick. I timed each step and diagnosed the expensive part as scraping data. I think that the website was responding slower to my heavy traffic. I decided to leave this for now as I the forecasting would not be incredibly accurate anyway – weather (although more stable on Mars, which lacks oceans) is notoriously

unpredictable and limited data from a single site would be unlikely to build a good predictor. I saw the predictor was more of a proof of concept.

See backend/forecasting_model/scrape_temps.py

Further Preprocessing and Model training

Another problem was the large amount of missing data, due to difficulties continuously transmitting data to Earth. I would need to preprocess this to fill in the missing values. I decided that since weather data changed gradually, linear imputation would be a good method (see the first steps of backend/train.py). Thus the data was all successfully pre-processed for fitting a 7-day sliding window time series model.

I lastly needed to decide on the model to use, an important design choice in any machine learning project. However, given that I had limited data, many imputations, and the data information said that the transmitted data could be inaccurate to up to 6degC, I did not think it was worth getting too bogged down in this step. I used a random forest regressor, which generally has high accuracy. With limited data (only 1200 sols), I used a test data proportion of 0.1 and achieved a mean squared error of less than 1degC (0.728).

See backend/forecasting_model/train.py

Full stack - frontend

Another challenge was serving predictions and visualizations in a user-friendly way. Having built the front end using React.js and styling with Tailwind CSS, I needed to set up a full-stack pipeline to allow seamless interaction between the front end, the backend, and the external API.

One challenge here was coordinating asynchronous calls in the frontend. For example, I had to ensure that weather data was loaded before triggering the forecast fetch, since the model relied on a rolling window of the most recent values. I managed state using React hooks (useEffect) and added loading screens to handle delays.

See src/Home.jsx.

Full stack - backend

To connect the frontend to the trained machine learning model, I needed to expose it through a clean, reliable API endpoint. I wrote a lightweight Flask application (backend/app.py) that could receive POST requests containing the most recent weather data, run it through the sliding window model, and return a 3-sol forecast in JSON format.

One of the issues was enabling cross-origin requests from the React frontend, which runs on a different port during development. Without CORS enabled, requests from the frontend were blocked by the browser. I resolved this by using Flask-CORS to explicitly allow cross-origin requests. I tested the endpoint using curl to simulate frontend requests and confirm that the returned JSON matched the expected structure.

See backend/app.py