



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

LABORATORY PART 1

270228 Advanced topics in data 2

GitHub repository:

<https://github.com/taed2-2526q1-gced-upc/TAED2-SmartHealth.AI/tree/main>

Matilde

Steffen

Renaux

October 27, 2025

Contents

1	Introduction	1
1.1	Goal of the project	1
1.2	Teammates' evaluation	1
2	Methodology	1
2.1	Milestone 1: Inception	1
2.1.1	Selection of problem and requirements engineering for ML	1
2.1.2	Dataset card	3
2.1.3	Model card	4
2.1.4	Project coordination and communication	6
2.1.5	Selection of the cloud provider	6
2.2	Milestone 2: Model Building – Reproducibility	7
2.2.1	Project structure	7
2.2.2	Code versioning	7
2.2.3	Data versioning	8
2.2.4	Experiment tracking	8
2.3	Milestone 3: Model Building – Quality Assurance	9
2.3.1	Energy efficiency awareness	9
2.3.2	Static code analysis	10
2.3.3	Model testing	11
2.3.4	Data testing	13
2.4	Milestone 4: Model deployment – API	14
2.4.1	ML-based component/system architecture	15
2.4.2	API design	17
2.4.3	API testing	18

3	Retrospective	20
4	Bibliography	23

1 Introduction

1.1 Goal of the project

The goal of this project is to develop a SmartHealth-AI tool that provides user guidance in relation to obesity. The project aims to build a model that takes health metrics as input and translates these into a personalized recommendation, helping to reduce the chance of obesity if detected.

1.2 Teammates' evaluation

☒ *"All team members agree that they had an equal contribution to this delivery and project: completing a fair share of the team's work with acceptable/high quality, keeping commitments, and completing assignments on time, helping teammates who are having difficulty when it is easy or important".*

2 Methodology

2.1 Milestone 1: Inception

2.1.1 Selection of problem and requirements engineering for ML

This specific project was chosen, because it is a topic of high relevance in contemporary society. According to World Obesity Federation "the total number of adults living with obesity will increase by more than 115 % between 2010 and 2030"[1]. Therefore, the need for preventive measures is urgent and this is where the proposed SmartHealth-AI hopefully can provide value as an accessible and user-friendly tool.

In order to succeed with the project, a set of requirements have been defined, which will be taken under consideration throughout the different phases of development. Below is a descriptive overview of them.

Non-functional requirements

The non functional requirements defines the model's capabilities in terms of performance, scalability, data restraints and reproducibility.

The first requirement concerns accuracy, the model must achieve a balanced accuracy of at least 0.85 and a sensitivity of at least 0.75 for each class.

A second requirement relates to generalization, when evaluated on an external dataset, the drop in accuracy should not exceed five percentage points compared to internal validation results. The model should not be overly influenced by regional, cultural, or demographic biases present in the training data. If the training data primarily originates from a specific geographic area or population group, the system must be validated against independent datasets representing diverse user profiles to ensure that its recommendations remain reliable and equitable across different contexts. Otherwise, the applicable geographical scope of the model must be clearly defined and documented.

Since the tool is intended for interactive use, inference time and generation of recommendation should remain below three minutes per user input.

In terms of scalability, the system should be able to handle up to 10 users simultaneously under normal operations. This should be adaptable to larger datasets without significant changes to the overall architecture. Scalability beyond this, with more than 10 users, can be achieved by a cloud provider service.

Finally, reproducibility is ensured by tracking iterations with fixed random seeds, version-controlling of datasets and defining dependencies in a shared environment file, so results can be replicated across different environments.

Functional requirements

The functional requirements of SmartHealth-AI is defined by the capabilities of the supervised machine learning system and the interaction with the users input. Therefore, the system should be able to process lifestyle and health information from the user and classify the individual into one of the obesity level categories.

The user should be able to enter the data manually through a simple interface. Then the

system uses the trained Random Forest classifier to predict the obesity level. The system must then provide both the classification output and personalized recommendations. The recommendation system will be designed to provide personalized health guidance based on a user’s obesity level. The core idea is to compare the user’s profile with individuals who share similar characteristics but fall within a healthy weight range. This comparison hopefully gives realistic and achievable lifestyle adjustments.

The recommendations should be generated and ready within 3 minutes, and a message should be sent to the user, so the experience is fast and practical for the user.

Future functional and non-functional requirements

At this time in the process we have not decided whether to make a website or APP where the system should be implemented. With a website our idea is that the system sends the recommendations directly on the website, so no personal details are needed. In that way we don’t have to think about GDPR regulations. The system would be a simple one-time interaction tool.

On the other hand with an APP would open the possibility for a more personalized assistant tool for long term guidance. However, this method introduces a lot of new aspects we have to consider and implement, like data privacy/security and ethical considerations.

2.1.2 Dataset card

The dataset card describes the ‘Estimation of Obesity Levels Based on Eating Habits and Physical Condition’ dataset introduced by Palechor & de la Hoz Manotas (2019) [2]. It consists of 2111 records and 17 attributes, including demographic, dietary, and lifestyle features, with a multi-class target variable (`NObeyesdad`) covering seven obesity categories ranging from insufficient weight to obesity type III.

- **Dataset details:** Collected via survey from individuals in Colombia, Peru, and Mexico (ages 14–61). Includes demographic, anthropometric, dietary, and lifestyle attributes.

- **Preprocessing:** Data cleaning, normalization, and oversampling with SMOTE to balance class distributions.
- **Intended use:** Educational and research purposes in obesity estimation and preventive health guidance.
- **Limitations:** 77% of the dataset is synthetically generated, limiting generalizability. Data is self-reported, and only represents three Latin American countries.
- **Ethical considerations:** Contains sensitive health-related data (weight, eating habits, activity). No personally identifiable information is included, but responsible use is required.

The full dataset card is available in the project repository:

<https://github.com/taed2-2526q1-gced-upc/TAED2-SmartHealth.AI/blob/main/docs/datasetcard.md>.

2.1.3 Model card

The model card has been made to provide a structured description of the SmartHealth-AI model, which is based on the Random Forest classifier. The model flow is showed in Figure 1 and shows how the lifestyle data is the input to the Random Forest model, which then categorizes the features and assigns each user to a category that makes the foundation for the individual recommendation.

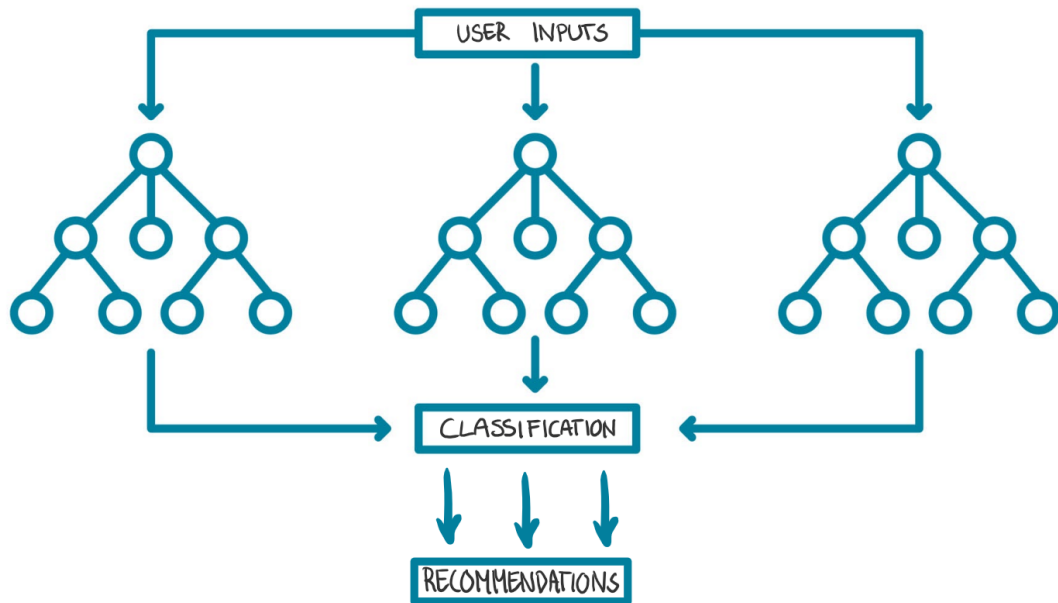


Figure 1: Fugure of the modelflow

The card includes the following main points:

- **Model details:** A supervised learning Random Forest model developed as part of the TAED2 course at UPC, intended solely for educational purposes.
- **Intended use:** To guide users toward healthier habits by providing preventive lifestyle recommendations when obesity risk is detected. The model is not designed for medical decision-making or professional diagnosis.
- **Limitations and risks:** The dataset contains synthetic components and may not fully represent real-world populations. As a result, predictions may carry bias and should not be interpreted as clinical advice or as a global tool.
- **Ethical considerations:** User privacy and transparency are emphasized, and the model is recommended for experimentation and coursework only.

The full model card is available in the project repository:

<https://github.com/taed2-2526q1-gced-upc/TAED2-SmartHealth.AI/blob/main/models/modelcard.md>.

2.1.4 Project coordination and communication

For the coordination of this project an iterative approach have been used. Due to the very clear course structure in terms of milestones for each laboratory session, we have chosen not to use a platform for the project coordination. We have instead worked together each session on the milestones attributed and then made 'homework' for each other for the next session, so we could ensure to follow the timeline. This have worked well, as we could all be a part of each milestone instead of dividing it up completely.

For project communication, we have mainly been together and been able to talk about the different issues that have occurred. However, for the communication outside of laboratory, we have made a discord group for the purpose.

GitHub has been the primary platform used for project. It is good for code management and version control, ensuring reproducibility. Furthermore, DagsHub has been connected to the GitHub repository for this project, as it allows for data management.

2.1.5 Selection of the cloud provider

The current project setup does not employ a dataset storage service. If a choice had been required during initialization, AWS S3 would have been selected due to its widespread use and seamless integration with machine learning workflows. However, upon further research, the UPC cloud services are identified as a more suitable solution for this project. The institutional availability of the UPC cloud for students, together with its free access and ease of use, makes it the preferred choice for future milestones. Even though the data presented in the project is relatively small in size, the use of a cloud provider still offers clear advantages, as it ensures that models and data are stored in a stable and reproducible environment that supports collaborative work.

2.2 Milestone 2: Model Building – Reproducibility

2.2.1 Project structure

In order to ensure a well-structured and reproducible workflow, the project was initialized using Cookiecutter, which provides an automatically generated setup with predefined folders and configuration files. This approach guarantees consistency across different environments and facilitates collaboration within the team.

When setting up the structure, a series of predefined parameters were selected to tailor the template to the specific needs of the project. The project was configured under the name `TAED2_SmartHealth.AI`, with the internal Python package named `taed2_smarthealth_ai`. It was set to run on Python 3.10, with environment management handled by `uv` and dependencies specified in a `pyproject.toml` file. No additional PyData packages or testing frameworks were included. For linting and formatting, the combined tools `flake8`, `black`, and `isort` were selected. This configuration enforces compliance with the PEP 8 style guide by detecting violations (`flake8`), automatically reformatting code to a consistent standard (`black`), and maintaining properly structured imports (`isort`). Together, these tools ensure not only compatibility with PEP 8 but also a uniform and reproducible code style across the project.

The project was released under the MIT license, while documentation support was disabled at initialization. Finally, the optional code scaffold was included to provide a basic starting point for development.

2.2.2 Code versioning

For the SmartHeath-AI project Git is used as the version control system. It's a great system for collaboration and development where Github Flow is served as primary workflow. Github flow is a effective branching strategy where the process is around a main branch and new feature branches can be made for development for models, functionalities etc. For our project different branches have been made. The first feature branch in our workflow was for data cleaning, so the data was fitted to future analysis and machine model training.

Thereafter, the data was suitable for model training. First off, a supervised decision tree model was created as a baseline model. Later on a random forest model was implemented and inserted in our main branch together with early results and findings.

This approach offers many advantages in terms of reproducibility. With Github flow it's easy to keep track of the development history of the project and how it will evolve.

2.2.3 Data versioning

To ensure reproducibility and traceability of the datasets used in the SmartHealth-AI project, we adopted Data Version Control (DVC). While Git provides versioning for source code, it is not designed to handle large or frequently changing data files. DVC complements Git by enabling lightweight tracking of data and model artifacts, while the actual files are stored in external storage (in our case, the remote repository provided by DagsHub, integrated with GitHub).

DVC structures the workflow into stages, corresponding to the main steps of the pipeline such as preprocessing, data splitting, and model training. Each stage specifies its inputs (e.g., raw data or scripts) and outputs (e.g., cleaned datasets, processed splits, trained models). These dependencies are declared in the `dvc.yaml` file, ensuring that if an input changes, only the affected stages are re-executed. The resulting lock file records the exact versions of inputs and outputs, guaranteeing reproducibility.

Both data and models are versioned in this way, with Git managing code and configuration, while DVC manages large files through external storage. This provides a clean separation between source control and data storage, while keeping them fully synchronized.

2.2.4 Experiment tracking

In order to track the experiments throughout the development of the model, MLflow has been implemented. It provides a way to log parameters, metrics, and artifacts in a structured manner, which makes it easier to compare results, reproduce experiments, and manage models over time. In the project, MLflow is expected to display the iterative de-

velopment process by ensuring that changes and improvements to the model are properly documented and organized

However, as the implementation of MLflow was initiated a bit later in the process of the model development, some of the changes that have already been made, are not documented. Specifically, the initial model was implemented as a decision tree classifier. However, it was quite early observed that a random forest model achieved significantly higher accuracy, and the model-type was therefore changed accordingly for the further development.

MLflow has now been implemented, and it is expected to provide structure and support for future iterations of the model. Through MLflow the model development can be automatically logged, making it possible to reproduce experiments and compare results in a systematic way. This is particularly important as the project aims to refine the model iteratively, with performance being evaluated against defined non-functional requirements such as an accuracy of at least 0.8 and robustness across diverse datasets.

2.3 Milestone 3: Model Building – Quality Assurance

This milestone focuses on validating that the SmartHealth-AI system meets the non-functional requirements defined in Section 2.1.1 through systematic testing and quality-assurance practices.

2.3.1 Energy efficiency awareness

We implemented the `train_obesity.py` training script with CodeCarbon’s emission tracker to continuously measure power and attribute energy and CO_2 emissions to each run. The choice of a Random Forest model already contributes to energy efficiency, as it avoids heavy accelerator use and long training epochs typical of deep learning methods. Key configuration choices were CPU-only training, high-frequency power polling (every 0.1 s) due to the short run duration, and file logging to the `emissions.csv` for downstream tracking.

From the output in the CodeCarbon log mentioned above, the average energy footprint of a single training run is:

- **Duration:** 1.08 s
- **Energy consumed:** 0.095 Wh (9.5×10^{-5} kWh)
- **Emissions:** 0.016 g CO₂eq
- **Avg. CPU power (observed):** ~ 318 W

Each run is extremely short and lightweight, reflecting the modest size of the dataset and the efficiency of the Random Forest model. The resulting environmental footprint is therefore negligible and has been documented in the model card.

2.3.2 Static code analysis

To support code quality and maintainability, our static analysis process employed both `Pylint` and `Ruff`. These complementary tools are particularly valuable in an MLOps workflow because they enforce consistent coding standards across data pipelines, model training scripts, and deployment components. By automatically detecting syntax errors, unused imports, naming inconsistencies, and style violations early in the development cycle, it reduce technical debt, prevent silent failures during automation, and make collaboration between team members more reliable.

For the `Pylint` analysis, selected folders were evaluated individually to obtain detailed quality scores. The `taed2-smarthealth-ai` folder initially achieved a score of 7/10, which includes modules for data preprocessing, model training, API logic, and Great Expectations integration.

Most of the issues were minor and easily correctable, such as missing docstrings, inconsistent variable naming, and deviations from import order conventions. After addressing these, the score improved to 8.53/10.

The `tests` folder initially achieved a score of 4/10, mainly due to missing docstrings, excessive line lengths, redundant variable definitions, and inconsistent naming conventions.

After applying automated formatting with `isort` and `black`, along with minor manual revisions, the score improved to 7.5/10. Approximately 2.5 points of this increase resulted directly from the automated corrections, which standardized import order, indentation, and overall code formatting.

Finally, the project’s root directory was analyzed using `Ruff`, a faster and more modern static analysis tool. The initial scan reported around 30 issues, most of which were automatically corrected through `Ruff`’s built-in fix functionality, cutting the number of remaining issues by roughly half. After running targeted validations across both folders, all checks completed successfully.

2.3.3 Model testing

Ensuring that the trained machine learning model performs correctly and consistently across different inputs is crucial for reliability. For this purpose, `Pytest` was used to automatically test the data preprocessing pipeline and the trained model. This approach confirms that the pipeline behaves as intended and that the model remains stable when exposed to new or unseen data.

The `test_model.py` file defines several `pytest` fixtures to load the trained pipeline, validation dataset, and label mappings. During the initial testing phase, three categories of tests were executed: one validating that the model’s accuracy met the required threshold of 0.85, and two assessing prediction behavior using distinct input profiles. These profiles shared certain attributes but differed in specific features, allowing verification that the model produced consistent and valid predictions under varying conditions. All three tests executed successfully, confirming that both the pipeline and model functioned as expected.

Beyond overall accuracy, it was also important to confirm balanced performance across all weight categories. This was evaluated using the `precision_recall_fscore_support` metric from `scikit-learn`, which computes recall for each class individually. Each category was required to achieve a recall score above 0.75 to ensure reliability and fairness.

Unit tests were implemented to verify that the data preprocessing functions correctly normalize and clean the input data. Four core functions were tested to ensure consistent text formatting, including trimming, lowercasing, and Unicode normalization, as well as proper handling of unmapped or missing values. All preprocessing components behaved as expected, producing standardized and error-free data suitable for model training and evaluation.

Following the preprocessing validations, parameterized tests were implemented to assess the model's predictive behavior across diverse user scenarios. Each test case simulated realistic input profiles with varying feature values such as age, gender, height, and weight. The model consistently produced coherent class labels, confirming its ability to generalize to new data.

Seven targeted tests further verified that the model assigned the correct weight category for each class using controlled, unseen input samples inspired by real data. All expected-label tests passed, demonstrating that the model's decision boundaries were correctly learned and applied.

A final stability test examined the model's robustness to small variations in input values. Minor adjustments in height and weight did not change the predicted class, indicating that the model provides stable and reliable outputs for nearly identical inputs.

Test summary: A total of 17 automated tests (plus one API test) were executed with `Pytest`, covering both data preprocessing and model validation components. All tests passed successfully:

- Data preprocessing tests — 4/4 passed
- Model accuracy and recall tests — 2/2 passed
- Model prediction and expected-label tests — 10/10 passed

- Test data validation — 1/1 passed

```
-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html  
===== 17 passed, 1 warning in 6.65s =====
```

Figure 2: Pytest output showing that all tests passed successfully (17/17)

Together, these tests provide a comprehensive quality assurance framework, ensuring that both the data pipeline and trained model are reliable, consistent, and aligned with the project’s performance goals.

2.3.4 Data testing

Beyond ensuring that the model works as required, it is also important that a machine learning model is trained and tested on correct and good data in order to obtain meaningful results. To achieve this, **Great Expectations** was integrated into the pipeline to verify that all input data complies with predefined quality standards. These validation checks prevent the introduction of incorrect or inconsistent inputs, which could otherwise lead to unreliable predictions or system errors.

The `gx_context_configuration.py` file defines the data validation rules established for each model attribute. Because the dataset contains different feature types, distinct validation criteria were specified for each. For numerical attributes, reasonable value ranges were defined to restrict future inputs within plausible limits. For example, while a user older than 120 years could theoretically cause a validation error, such a case is considered highly unlikely. Binary attributes were restricted to the values `[0, 1]`, and ordinal attributes were constrained to match their corresponding number of categories.

Great Expectations then validated the datasets against these defined constraints to ensure full consistency. In addition, **Pytest** was used to verify that the cleaned datasets adhered to all rules within the validation framework.

Together, the static code analysis, data validation, and automated model tests ensure that the system meets its non-functional requirements of maintainability, reliability, and

reproducibility. They also confirm that the model and data pipelines behave consistently across environments, providing a robust foundation for deployment in later milestones.

2.4 Milestone 4: Model deployment – API

In this milestone, we focused on deploying our trained Random Forest model so that it could be accessed as an interactive web service. The deployment was carried out on Virtech, a private server within the UPC internal network, following the Private Cloud Service for Students documentation.

We built the API using **FastAPI**, chosen for its simplicity, speed, and automatic generation of OpenAPI documentation. The trained model was serialized with Joblib. Complementary configuration files, such as the feature order and class labels, were defined in the project’s `params.yaml` and JSON descriptors to guarantee consistency between training and inference.

Environment-specific secrets (for instance the Google Gemini API key used to generate personalized advice) were stored securely in an `.env` file, loaded through `python-dotenv`. This prevents credentials from being hard-coded and simplifies portability between local development and server deployment.

The FastAPI application exposes several endpoints corresponding to the system’s logical components. A detailed description of each route and its specific purpose is provided in subsection 2.4.2 (API Design).

A lightweight frontend interface (`index.html`) was also created and served as a static file by FastAPI. This page lets users input their health and lifestyle parameters, triggers model inference, and dynamically displays results and advice through JavaScript `fetch()` calls to the API.

During development, the API was first tested locally on Windows using Uvicorn, enabling quick iterations and debugging. Once stable, the same application was deployed on Virtech so it could be accessed within the UPC network.

2.4.1 ML-based component/system architecture

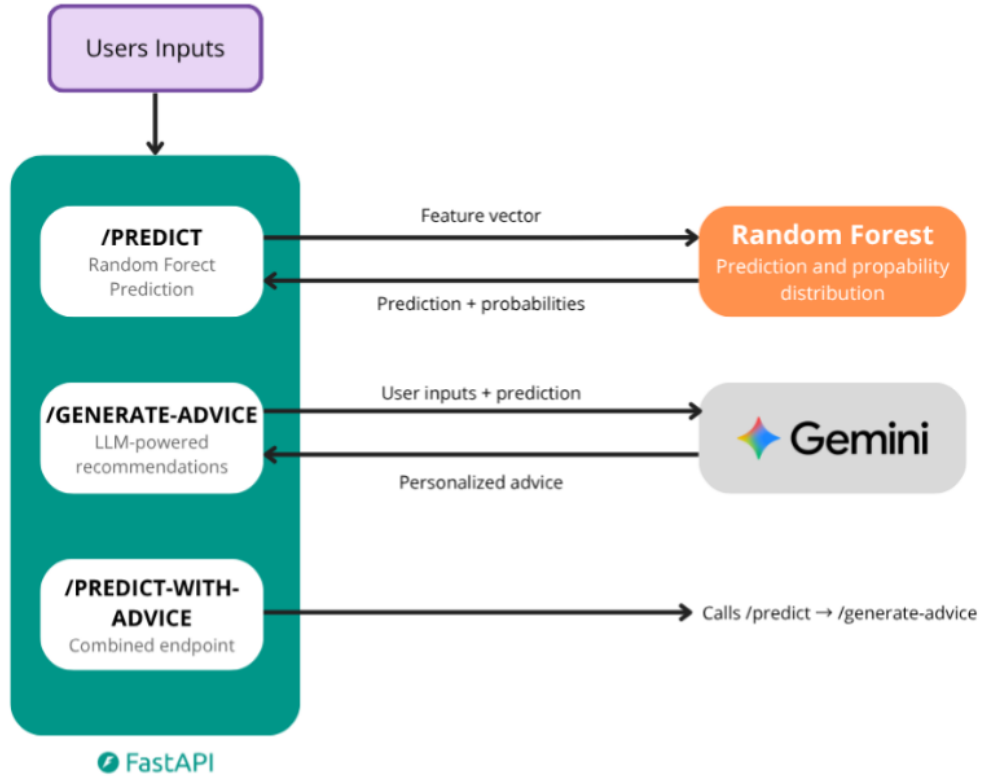


Figure 3: Logical architecture of the SmartHealth-AI system

From a **physical perspective**, the system is deployed on the **Virtech** private cloud infrastructure of UPC. Virtech was selected because it offers a secure and controlled environment within the university network, providing sufficient CPU resources for our application and allowing team members to access the service without exposing it publicly on the internet. The FastAPI application runs on a dedicated Virtech server instance using CPU resources for model inference. All essential resources, including the trained Random Forest model, configuration files (`params.yaml`, feature and class JSONs), and static frontend assets, are stored on the Virtech instance. Environment variables, such as the `GOOGLE_API_KEY`, are securely managed through an `.env` file, as mentioned earlier, that is loaded automatically at startup, ensuring that the Google Gemini integration is always active. If, for any reason, the external Google services become temporarily unavailable, the API continues to operate normally for predictions and returns a structured internal error for the advice generation endpoint, without interrupting the rest of the system.

From a logical perspective, the architecture is divided into two core components:

- **Prediction module:** receives a JSON payload containing the user’s health and lifestyle inputs. These are validated and converted into a numeric feature vector, then passed to the Random Forest classifier, which outputs the predicted obesity class, a confidence score, and a full probability distribution across categories.
- **Advice generation module:** takes both the original user inputs and the model’s prediction to create a structured prompt. This prompt is sent to Google Gemini through its API. The model response is parsed and formatted into a JSON object containing an array of recommendations and a disclaimer note.

The overall system design follows a clear **separation of concerns** pattern:

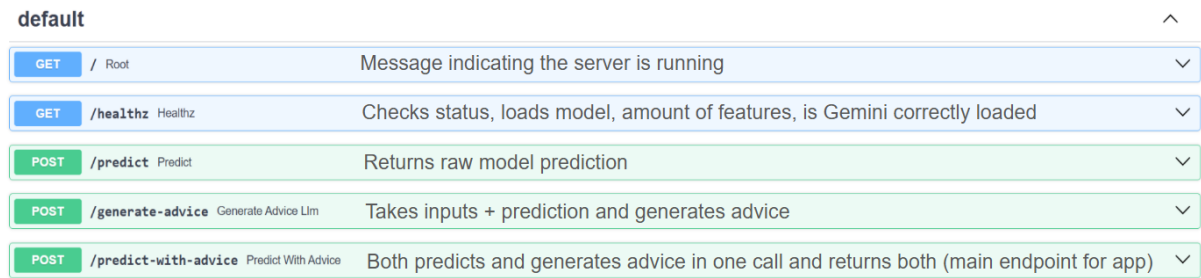
- the **web interface layer** (FastAPI + HTML/JS) manages user input and API routing,
- the **ML layer** handles feature processing and Random Forest inference,
- the **LLM layer** is responsible for natural-language advice generation.

This modular structure improves maintainability and robustness. Each part can operate independently, ensuring that predictions remain available even if the Gemini service is temporarily unreachable. The architecture is lightweight and easily portable, once stabilized it can be scaled on cloud environments if needed.

Finally, during the design phase, we also explored the possibility of integrating a local backup language model as a fallback mechanism to maintain continuous advice generation in case of connectivity loss with Google Gemini. This approach would involve hosting a smaller language model directly on the Virtech infrastructure. While the disk storage requirements for such a model would be feasible, a critical limitation emerged regarding RAM availability. The theoretical RAM space remaining on the server after accounting for existing services would be inadequate to run a non-fine-tuned model with sufficient capacity to generate coherent and clinically appropriate health recommendations. As a result, while this backup strategy was explored theoretically, it was not incorporated into the version that underwent testing or production deployment.

2.4.2 API design

For this part of the milestone, we designed and implemented five distinct API endpoints, each serving a specific purpose. Figure 4 shows the complete endpoint overview with their individual descriptions.



default ^		
GET	/ Root	Message indicating the server is running
GET	/healthz Healthz	Checks status, loads model, amount of features, is Gemini correctly loaded
POST	/predict Predict	Returns raw model prediction
POST	/generate-advice Generate Advice Llm	Takes inputs + prediction and generates advice
POST	/predict-with-advice Predict With Advice	Both predicts and generates advice in one call and returns both (main endpoint for app)

Figure 4: Description of the five endpoints

The two **GET** endpoints are straightforward and mainly serve diagnostic or routing purposes. The root route (`/`) simply returns a message confirming that the API is running, while the `/healthz` route checks the system status by verifying that the model is correctly loaded, the expected number of features is available, and that the Gemini API is properly configured.

The first **POST** endpoint, `/predict`, receives a JSON payload containing the user’s health and lifestyle data. It validates that all required features are present and numeric, then passes the feature vector to the Random Forest model. The response contains the predicted obesity class, the associated confidence score, and the probability distribution across all classes. If any features are missing, the API returns a 400 error with a detailed message listing the missing inputs.

The second **POST** endpoint, `/generate-advice`, takes both the user input and the model prediction, builds a structured prompt, and sends it to the Google Gemini API to generate personalized lifestyle recommendations. If the Gemini service is unavailable, the API returns a structured 503 error response while keeping the rest of the system operational.

Finally, the `/predict-with-advice` endpoint combines both steps into a single call, providing the model prediction and AI-generated advice in one response.

For the classroom demonstration, the application was designed to use this two-step architecture (`/predict` followed by `/generate-advice`) to make each stage of the workflow visible and easier to test independently. In a production setting, however, the `/predict-with-advice` endpoint would be used instead, as it combines both operations into a single call and provides a more efficient and user-friendly interaction for the app.

All endpoints communicate through **JSON** input and output formats, and FastAPI automatically validates that the incoming data matches the expected schema. It improves reusability, and make future scaling or replacement of the model or advice system easier. Example requests and responses can be tested directly via the automatically generated FastAPI documentation available at `/docs`.

2.4.3 API testing

During this milestone, we implemented automated testing to make sure that the SmartHealth-AI application was reliable and that all parts of the FastAPI server worked as expected. The tests were built using the `pytest` framework, following the structure and examples from the official FastAPI documentation [3]. The main purpose was to check that the API endpoints responded correctly, that the trained Random Forest model was properly loaded, and that valid user input produced a realistic prediction output.

Instead of testing the API manually each time, we used FastAPI's built-in `TestClient` to simulate requests directly in Python. This allowed us to check the behaviour of the API without having to start the server, which made the process faster and easier to reproduce. It also made it possible to automatically verify that later code changes do not break earlier functionality.

The goal of the testing setup was to make sure that all main endpoints returned the correct responses. The tests also checked that the error handling worked properly, for example if the user sends an incomplete JSON file or if the Gemini API key is missing. In addition, the tests confirmed that the model files (parameters, features, and classes) were successfully loaded at startup and that the prediction response followed the expected structure.

The test file imports the main FastAPI app from the file with the API code. Using the FastAPI TestClient, we wrote six tests:

- A root check verifying that the endpoint `‘/‘` returns a status code 200 and a startup message.
- A health check confirming that `‘/healthz‘` reports model status, feature count, and Gemini API configuration.
- A prediction test that sends a valid JSON input and checks that label, confidence, and probabilities are returned.
- An error test that ensures a 400 Bad Request is raised when features are missing from the input.
- An advice test that confirms `‘/generate-advice‘` returns a 503 response if no Gemini API key is configured.
- A combined test for `‘/predict-with-advice‘` using monkeypatching to mock the Gemini model response and verify that both prediction and advice are returned correctly.

When we ran the tests using pytest, all six passed successfully (6/6). This confirms that the FastAPI application correctly loads the trained model, handles missing or invalid inputs, and integrates smoothly with the advice generation module. The only warnings that appeared were DeprecationWarnings related to FastAPI’s `‘example‘` keyword, which do not affect the functionality or correctness of the API. The test output is shown in Figure 5, where all tests passed without errors.

Automated testing is important when ensuring that our application is stable and reproducible. With these tests in place, we can continue development without worrying that existing features break. It also helps meet our non-functional requirements related to reliability and reproducibility, since the same tests can be run in any environment to verify that the API behaves consistently.

Throughout Milestones 3 and 4, the team ensured that the functional and non-functional requirements defined in subsection 2.1.1 were fulfilled. Functional requirements, such

as providing personalized recommendations and real-time predictions through an accessible interface, were achieved via the FastAPI endpoints and the integrated advice generation system. Non-functional requirements related to reproducibility, energy efficiency, accuracy, and reliability were validated through DVC, MLflow, CodeCarbon, and automated testing with Pytest.

```
-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html  
===== 6 passed, 3 warnings in 5.93s =====
```

Figure 5: Pytest output showing that all API tests passed successfully (6/6).

3 Retrospective

Explain the main challenges, barriers, and opportunities you encountered during the project. Describe what you have learned, what you would do differently, etc. Explain what concepts you have used from other courses.

- **Renaux:** The main challenge for me was the limited amount of time available. At my home university, a similar course covers the same content over an entire semester, whereas here we had only about six weeks to complete the full project cycle. Another difficulty was working with teammates I did not know beforehand and who came from different academic backgrounds, it has been a long time since it happened to me. It required some adaptation to different working methods and ways of thinking.

I also found the integration with DagsHub challenging. I am still not completely confident with how it works, and this caused issues when deploying the API, some data or model were not properly synchronized.

Regarding opportunities, I appreciate this type of project because it always leaves room for improvement or new features. However, that flexibility is also a challenge, as it forces us to decide where to draw the line and focus our efforts within the given timeframe.

Throughout the project, I learned how an API works in practice and how to make a machine learning model accessible to users who are not technically oriented. I also discovered the advantages of using the uv environment manager, it was lightweight and reliable compared to previous experiences with Conda.

Overall, I would not change much in our approach. The project helped me consolidate knowledge from previous courses, such as model design and evaluation from my Machine Learning course and experiment tracking concepts from my Deep Learning course.

- **Steffen:** The project and first part of the course have been very exciting for me. It combined elements that were familiar with previous courses “Introduction to Machine Learning and Data mining”, with the machine learning part, where we trained and selected an appropriate model for our chosen dataset and implementing an API “Data and Data Science”.

One of the biggest learning experiences for me was working with GitHub and DagsHub. I had never used branches or version control before, and not with DagsHub integration at all. Learning how to collaborate through these platforms, and using tools for each milestone, was challenging at first but very valuable from a learning perspective.

In the beginning, I found it difficult to understand why Github/Dagshub was a good idea to use for a project like this, and why DVC, uv and mlflow needed to be implemented as well. Because of that, I spent a lot of time just figuring out how things worked and why they were useful. Once I understood the overall structure, however, the workflow became much clearer, and I could focus on developing the project itself instead of using time on understanding the project structure. I spend a lot of time using software engineering tools like Git, Pytest, Great Expectations, Pylint, Ruff. I believe those tools could be very useful for future studies and similar courses.

Another major learning point was the API implementation. In a previous course called “Data and Data Science, I had only worked with pre-built APIs, but here we had to design and implement one ourselves. This gave a much deeper understanding

of how APIs work and how to connect them to a machine learning model.

If I were to do the project again, I would spend more time in the beginning getting familiar with the tools and workflow structure, so I could focus earlier on the project. I now feel more confident working with Github, version control, and quality assurance techniques in machine learning projects.

- **Matilde:** The main challenges I experienced in this project were especially related to the overall setup and understanding how the different tools presented in the coursework worked together. I had never used GitHub for a project before, so that was a completely new experience for me. In the beginning, I really struggled, especially since we joined the course late due to a miscommunication with the university. It all moved quite fast, and even though this was taken into consideration, it still felt like I had missed several important steps that I had to figure out on my own.

However, once I got through that initial phase, I think I gained some very useful skills throughout the different milestones. I really liked how the course was structured and how it reflected the way work is done in the industry. At my home university, machine learning courses are usually more focused on the mathematical and theoretical aspects of different model types. Still, this background turned out to be very helpful here, as it gave me a good understanding of what an ML model does and how to build one.

If I were to take the course again, I would definitely spend more time in the beginning getting the setup right, as that would have made the whole experience smoother. In the end, everything came together and made sense, but it would have been nice to have that feeling earlier in the process. Overall, I think the workflows and practical skills I've learned in this course will be very helpful in my future career.

4 Bibliography

References

- [1] World Obesity Federation. *World Obesity Atlas 2025: Majority of countries unprepared for rising obesity level*. Accessed: 2025-09-25. Mar. 2025. URL: <https://www.worldobesity.org/news/world-obesity-atlas-2025-majority-of-countries-unprepared-for-rising-obesity-level>.
- [2] Fabio Mendoza Palechor and Alexis de la Hoz Manotas. “Dataset for estimation of obesity levels based on eating habits and physical condition in individuals from Colombia, Peru and Mexico”. In: *Data in Brief* 25 (2019), p. 104344. DOI: [10.1016/j.dib.2019.104344](https://doi.org/10.1016/j.dib.2019.104344). URL: <https://archive.ics.uci.edu/dataset/544/estimation+of+obesity+levels+based+on+eating+habits+and+physical+condition>.
- [3] Sebastián Ramírez. *Testing - FastAPI documentation*. Accessed: 2025-10-21. 2025. URL: <https://fastapi.tiangolo.com/tutorial/testing/#extended-fastapi-app-file>.