README 1

Recipe Finder and Glucose Peak Calculator  
  
Final Project

# Table of contents

* Table of contents
* 1. Initial Idea and motivation
  + 1.1 Motivation
  + 1.2 Concept
* 2. The project and its structure
  + 2.1. Summary
  + 2.2 Image generation and development preparation
    - 2.2.1 ./.scripts
    - 2.2.2 ./dock
    - 2.2.3 ./.vscode
  + 2.3 Application Code
    - 2.3.1 src
      * API Interfaces
      * Commons
      * Db
      * Glucose Prediction
      * Models
    - 2.3.2 templates:
    - 2.3.3 app.py
  + 2.4 Unit Tests
* 3. Building and running as a docker image
  + 3.1 Building the image:
  + 3.2 Running the image:
  + 3.3 Download the latest published image from DockerHub:
* 4. App usage
  + NOTE THAT: Required API keys are not included in the repository
* 6. Developing guidelines
  + 6.1. Opening project in VS Code
  + 6.2. Debugging with flask
  + 6.3. Notes on development
  + 6.4 Handling secret variables
* 7. Conclusion
  + 7.1 Degree of application of the knowledge acquired:
  + 7.2 Relationship with other subject in the master's degree:
  + 7.3 Future improvements

# 1. Initial Idea and motivation

## 1.1 Motivation

For a diabetic, it is sometimes difficult predicting how much insulin should he administrate when eating a certain meal.

The main sources of glucose in a meal are either:

* **Sugars**: Short chains, or straight up molecules, of glucose that are directly absorbed by the organism, and have a strong yet short-lasting impact on the blood-glucose. On a graph, it translates in a short high peak in measurements
* **Carbohydrates**: Longer glucid chains that take longer to break before being absorbed, and have a lower intensity, longer acting impact on the blood-glucose graph, translating in a longer, lower curve.

It is not always obvious how to estimate how a meal will affect the patient; there can be some ingredients that are not intuitively related to a high carbohydrates' concentration.

This is why we think creating an application that could help to predict how a patient's glucose will behave after consuming a known meal would be a great idea!

## 1.2 Concept

The proposed application is based on searching by ingredient (the ones the user has in its fridge) and generating recipes based on said ingredients.

Then, by accessing his sensor's API, the last 24h of readings can be accessed, and from those and the nutritional values of the selected recipe, a forecast of his glucose can be predicted.

# 2. The project and its structure

This solution can be deployed as a single container, which will launch the solution in this repository.

Diagram of the solution:

The main app (*SERVER*) runs when the container is started.

The server is in charge of:

* Serving a website on localhost:5000 (WEB CLIENT part of the diagram above) and handling the user's interactions.
* Making requests to external APIs (*Spoonacular* for recipes and nutritional information, and *LibreView* for the user's glucose readings)
* Taking the glucose data, the user's input for intended insulin administration, and the carbohydrates in the selected recipe, and estimating the next hour's glucose.

Below is a summary of the solution's directory structure and an explanation of each one of the modules in that solution.

## 2.1. Summary

root/  
│  
│ #------------ IMAGE GENERATION AND DEVELPOMENT PREAPARATION ------------  
│  
├──.scripts/ # Scripts for helping task automations  
│ ├── env-setup.bash  
│ └── env-setup\_deploy.bash  
│  
├──dock/ # instructions for building the image  
│ └── Dockerfile  
│  
├── requirements.txt # Python dependencies  
│  
├──.vscode/ # VS Code specific settings  
│ ├── launch.json # Debug configurations  
│ ├── settings.json # Editor settings  
│ ├── extensions.json # Recommended VSCode extensions  
│ └── tasks.json # Build tasks  
│  
│ #-------------------------- APPLICATION CODE --------------------------  
│  
├──src/ # Source code regarding different implementations, and  
│ │ # contains the libraries app.python will call,regarding   
│ │ # other API calls, models for information, helpers and   
│ │ # handlers, etc.   
│ │   
│ ├── API\_interfaces/ # Classes in charge of handling external   
│ # API communications  
│ ├── commons/ # Helpers for different tasks  
│ ├── db/ # [UNUSED] -> Database  
│ ├── glucosePrediction/ # Classes for glucose prediction  
│ └── models/ # Models for structuring information  
│   
├──templates/ # Directory holding the frontend part of the project,  
│ │ # namely the code and resources for the web  
│ │   
│ ├── css/ # Styles  
│ ├── js/ # Frontent Javascript Code  
│ ├── html/ # HTML pages  
│ ├── resources/ # Resources for the site (such as logos, fonts, etc)  
│ └── index.html # Main directory, the one the app will launch  
│  
├── app.py # Main Flask application  
│  
│  
├── .env # Public environment variables (not sensitive)  
│  
├── solution.code-workspace # Project file for VS Code  
│  
│ #-------------------------- UNIT TESTING --------------------------  
│  
└──tests/ # UnitTesting files (mainly for helping with a more TDD)

## 2.2 Image generation and development preparation

This section of the project contains scripts and utils for helping running the project

### 2.2.1 ./.scripts

Scripts in charge of preparing the system for running the code, aimed to Linux:

* env-setup.bash Is a script in charge of:
  + Installing the needed Linux apt packages, such as python, python-venv and python-is-python3
  + Creating a virtual environment in /.venv ​
  + Installing all the pip dependencies found in requirements.txt
  + Downloading the model for glucose prediction to /reggressionGlucoseSimple.joblib ​, which is too big to be uploaded to GitHub
* env-setup\_deploy.bash does the same as env-setup.bash but with minor tweaks for running inside the container.

### 2.2.2 ./dock

Contains the *Dockerimage*, in charge of building an image from the current develop branch of this project

### 2.2.3 ./.vscode

Contains files for helping with development in VS Code:

* launch.json Contains the debug configurations for running the project in debug
* tasks.json Contains tasks that can be run from VS Code that help getting the project ready for development (essentially runs env-setup.bash)
* extensions.json Contains the recommended extensions for helping in development, which will be recommended to the user when opening the project in VS Code
* settings.json Contains the editor settings (paths, behaviors...)

## 2.3 Application Code

This group of directories and files contain the code of the application itself. It can be divided into two main sections:

* src which contains the backend code
* templates which contains the frontend code

Aside from those two folders. The file that runs the whole application is app.py, which serves a Flask API.

### 2.3.1 src

#### API Interfaces

This folder includes programs in charge of communicating with other external APIs:

* [**Spoonacular**](https://spoonacular.com/food-api/) is an API dedicated to nutritional information. For this project, it accomplishes two main functions:
  + Generating recipes from a list of ingredients
  + Providing details about the preparation of the recipes, as well as about the nutritional values of said recipes.
* [**LibreView**](https://www.libreview.com) is a service that helps Diabetes Patients and users of the [FreeStyle LibreLink](https://www.freestyle.abbott) sensors manage their blood glucose data.

It also includes an Interface class (iAPI\_interface) that helps to define the base function an API interface program must have.

#### Commons

Commons is a collection of helpers that aim to centralizing some common tasks:

* DateTiemHelper.py implements some functions for converting from *date-time* objects to timestamp strings and vice versa, as well as performing some calculations or modifications over time-related variables
* FileManagement.py helps to manage file-related tasks, such as creating files, deleting, checking if a path is valid for python...
* LoggerInitializer.py contains a class that initializes a logging.logger object with the desired parameters, accounting for format or colored logging.
* PlotlyGraphHelper.py creates graphs in HTML from data
* Serializable.py is perhaps the most important of the commons, works as an abstract class or interface for defining the methods a serializable class must implement (such as converting from dictionary to an instance of said class) as well as common methods (such as serializing, deserializing or converting an object to a dictionary)

#### Db

Although it is CURRENTLY UNUSED, it consists of two programs:

* iDatabaseCompliant: Working as an interface for defining objects that account for being stored in a database table, such as creating a schema for creating a table for said object, inserting an object in a table, or fetching an object in a table.
* DataBase.py is a set of methods for initializing the database for the project.

#### Glucose Prediction

Contains the methods for initializing the predictor from the object stored in reggressionGlucoseSimple.joblib (a file that is not included in the current repository and will be downloaded by running a task in VS Code, F1 > Tasks: Run Build Task > Environment Setup), and for predicting from data with that model

#### Models

Contains models for objects frequently used in this application:

* LibreView: Objects related to glucose readings and the LibreView API; help connecting to the API and converting received data into vectors or other types more useful for this application
* Spoonacular: Objects related to the Spoonacular API, such as recipes or nutritional information
* ClientRequestModels.py: Objects that model the client inputs to the project's API

### 2.3.2 templates:

Contains the files regarding the web interface

### 2.3.3 app.py

It is the main program of the application. It serves a Flask API and serves endpoints for interacting with the rest of the modules in the application.

## 2.4 Unit Tests

The /test ​ contains multiple files for debugging the application using the python UnitTest module

# 3. Building and running as a docker image

## 3.1 Building the image:

The image will be built (at the moment, at least) from the develop branch of this repository.

The build instructions are stored in /dock/Dockerfile ​:

# 1. Use ubuntu as base  
FROM ubuntu:22.04  
  
# 2. Install git  
RUN apt-get update && apt-get install -y \  
 git \  
 && apt-get clean  
  
# 3. Clone the app repo  
WORKDIR /app  
RUN git clone -b develop https://github.com/mihaiBront/CloudComputing\_FinalProject.git .  
  
# 4. Run script to create environment for the app  
RUN bash -c "source .scripts/env-setup\_deploy.bash"  
  
# 5. Expose the port  
EXPOSE 5000  
  
# 6. Command for starting the app with container  
CMD ["/bin/bash", "-c", "source .venv/bin/activate && flask run --host=0.0.0.0 --port=5000"]

1. Takes the latest instance of the Ubuntu Docker Image
2. Installs GIT for cloning the repository
3. Creates a directory for the application(app) in the root, sets it as the working directory, and clones the latest commit from the develop branch to that folder.
4. Runs the script in charge of getting all dependencies ready (in this case, since it is working from inside the repository, .scripts/env-setup\_deploy.bash)
5. Exposing the port where the application will be served
6. Setting the startup command

The creation of a variant of the env setup script is due to the fact that the scripts that are run from the *Dockerfile* are already being run as root, and are not compatible with sudo

For creating the image, move to the dock directory:

cd dock/

Then, run the following command:

docker build -t <image-name> .

## 3.2 Running the image:

Run the following script:

docker run -p 5000:5000 --name <container-name> <image-name>

Or alternatively, if you want to run detached from the container:

docker run -p 5000:5000 -d --name <container-name> <image-name>

## 3.3 Download the latest published image from DockerHub:

link: [docker.io/glucose-prediction-cc](https://hub.docker.com/r/mihaibront/glucose-prediction-cc)

# 4. App usage

Accessing localhost:5000 will reveal this page:

Where:

1. **Ingredients input**: Insert the ingredients you have for generating recipes
2. **Recipes List**: The app will suggest three recipes containing, at least, those inputted ingredients. Selecting one will turn it green, and its details will be shown in the lower-right side of the screen
3. **Glucose Graph**: Graph containing the predictions taking into account *last 24h of glucose*, the *insulin input* and the *carbs* of the recipe
4. **Insulin Input**: The doses of insulin the user intends to administrate himself
5. **Recipe Details**: Details about the selected recipe

## NOTE THAT: Required API keys are not included in the repository

Navigate to localhost:5000 and insert the required API keys:

* *Spoonacular* (you can register)
* *LibreView* API key and account token (you can get them by accessing the service through their web and spying on requests with the inspect tool)

# 6. Developing guidelines

| Concept | Requirement |
| --- | --- |
| Python | version>=3.12 |
| IDE | Visual Studio Code |

## 6.1. Opening project in VS Code

1. Press F1 to open command palette
2. Have Python 3.12.X installed
3. Search for "*Tasks: Run Build Task*" and click on it
4. Run the "*Environment Setup Task*", which will install all necessary dependencies and get the virtual environment ready
5. Press F1 again and select interpreter as

## 6.2. Debugging with flask

1. Go to the debug pane (left bar in Visual Studio Code)
2. Find "*RUN AND DEBUG*" on the top of that pane (you will see a Play button and a *drop-down* selector)
3. In that *drop-down* selector, chose Python Debugger: Flask ({Project Name})
4. Run it. It will launch flask in *debug mode* with *hot reload*.

## 6.3. Notes on development

* All necessary pip installs must be added to the requirements.txt, since will be used for the container setup as well
* Changes in settings for this project must be done at "workspace" level, so they are updated to all users

## 6.4 Handling secret variables

Secret variables like API keys, database credentials, and other sensitive information should be handled as follows:

* Create a .env.local file in the root directory of the project
* The .env.local file is already added to the .gitignore file, so these secret variables don't get tracked.

***Warning!***  
  
All sensitive information such as "API" keys, passwords or any login information which use is intended for testing or development should be stored inside this env.local file.

***Pending...***  
  
Definition of where and how to store login information. For the purposes of this project, we will simply add them to the env file and handle them from there

# 7. Conclusion

## 7.1 Degree of application of the knowledge acquired:

The main concept from this subject (*SJK005 - Cloud Computing*) that we applied to this project is the creation of container images and the deployment of containers with Docker/Podman.

Since this specific application relates to **big-data** and **machine learning**, we also applied some concepts from the *Cloud Computing, Big Data, Al and ML0*

Unfortunately, the project didn't need the implementation of various containers nor any *container orchestration* it didn't reach the complexity.

## 7.2 Relationship with other subject in the master's degree:

We used knowledge acquired in the *SJK006 - Big Data* and *SJK004 - Machine Learning* subjects for creating the predictor model. The predictor model has been trained from a dataset taken from *Kaggle*, [brist1d - blood glucose dataset](https://www.kaggle.com/competitions/brist1d)

The model training required:

* Extensive data engineering on the dataset in order to reduce the dimensionality of the dataset (for instance, combining the exercise columns with the calories' consumption ones, or resolving NaN values).
* Training the model with a simple random forest regression model.

## 7.3 Future improvements

To fit the delivery date for this project, some compromises have been made:

* The readings we present as the "last 24h readings" are not actually the last 24h's, but an average of last week's glucose readings, modified to look like they are in real time. This is due to the lack of documentation for the ***LibreView*** API. The method for getting last week's report has been found by spying on the communications from their web client using the browser's developer tools.  
  As an improvement for the future, the API should be studied in detail to be able to get an accurate petition of the last 24h
* The model that we trained for this application is very basic, and it only predicts one point in time: one hour in the future from the petition (the prediction curve you can see in the website's graph is actually an interpolation of the last 2h of readings and that one-hour-in-the-future point).  
  As an improvement, the selected model should be one that fits the case better, supporting time-series predictions more accurately.  
  Also, the dataset only provided the glucose measurement 1h in the future. As another improvement, the dataset should be modified, so it provides an output vector of, let's say, 2h, with measurements every 10min as validation for the model.