Explanation module for **Explanations Food Recsys**

Documentation: Matteo Mannavola, Roberta Sallustio

The **foodWebApp** folder of **Explanations Food Recsys** contains a versatile explanation module which can take as input one or two recipe recommendations, accompanied by various user characteristics, and can provide different kinds of justifications for such recommendations.

The explanation module is made up of 3 Python scripts: **web_expl.py**, **expl_types.py** and **cmd_expl.py**.

Changes: it was previously made up of only web_expl.py, which also included the contents of expl_types.py. The responsibilities are now separated and web_expl.py contains only the core server functionalities, while the methods that handle the creation of the justification have been moved into expl_types.py. Another new script is cmd_expl.py, a command line client.

web_expl.py

The **web_expl.py** script is a Flask server (currently running on localhost, port 5003) which handles HTTP GET requests that provide a series of parameters (one or two recipes, explanation type and style, user characteristics) and returns a justification in json format.

When beginning to handle each GET request, the server loads data from different json files:

- **Nutrient.json** (risks, benefits and daily reference intake of different nutrients)
- **Restrictions.json** (explanations for restrictions like vegetarian, lactose-free…)
- **RichIn.json** (lists of ingredients that are rich in specific nutrients; explanations for the required nutrients in different age groups)
- **Sustainability.json** (lists of ingredients with high and low carbon footprint and water footprint sustainability)
- **Seasonality.json** (lists of seasonal ingredients for each season)
- **Dopamine.json** (list of dopamine-inducing ingredients; dopamine explanation)

The information contained in these files is used in different kinds of justifications.

Changes: previously, only Nutrient.json and Restrictions.json were available.

Then, a dataset with information about recipes from GialloZafferano is loaded (**dataset_en.csv**). One or two recipes are also extracted from the GET request arguments.

Changes: previously, dataset_en_v2.csv was used, but it contained wrong numeric values.

The user characteristics are then extracted from the GET request arguments and stored in a dictionary:

- **Age** (numerical age or U20/U30/U40/U50/U60/060)
- **Mood** (bad/good/neutral)
- **Stressed** (yes/no)

- **Depressed** (yes/no)
- **BMI** (over/under/normal)
- **Health_style** (1/2/3/4/5)
- **Health_condition** (1/2/3/4/5)
- **Activity** (low/high/normal)
- **Sleep** (low/good)
- **Cooking_exp** (1/2/3/4/5)
- **User_time** ([1,200] mins; 0=no contraints)
- **User_cost** (1/2/3/4/5; 5=not important)
- **Goal** (lose/gain/no)
- **User_restriction** (vegetarian/lactose-free/gluten-free/low-nickel/light)
- **User_ingredients**
- **Sex** (m/f)
- **Season** (winter/spring/summer/autumn)

User_restriction and User_ingredients can accept more than one value, each separated by a comma (“,”).

Changes: Season was not available previously. User_restriction previously accepted only one value. Multiple values for User_ingredients were previously separated by “-“, but now it is “,” (in both restrictions and ingredients) to accommodate for restrictions like lactose-free, gluten-free etc.

Then, the explanation **type** and **style** are also extracted from the GET arguments.

The type indicates a specific aspect of the recipe which the justification will focus on, while the style can be single (individual explanations for the two recipes), comparative (comparison between the two recipes) or both.

Changes: previously, the style could not be selected. It was not a request argument, so it was only possible to obtain all explanations (single explanations for both recipes and comparative). Furthermore, it was previously necessary for every request to contain two recipes; now a request with a single recipe can be handled.

After checking the validity of the given type and style values, the requested justifications are obtained by calling the **get_str_exp** function from **expl_types.py**. The justifications are then returned in json format.

Changes: previously, the value of the type argument was not checked to be in the correct bounds.

expl_types.py

The **expl_types.py** script is imported into web_expl.py. Its most important function is **get_str_exp**, which is called by web_expl.py and orchestrates the creation of the justification by selecting the given type. The other functions contain implementations for the creation of the specific explanation types.

Changes: previously, get_str_exp was divided in two methods (get_str_exp_one and get_str_exp_two) with the same structure, but different function calls. Furthermore, get_str_exp has been made more robust to missing/wrong values in the GET request. For instance, the absence of specific user preferences (e.g. user_time) is now interpreted as a neutrality, and a justification is still given.

Most explanation types do not have a single dedicated function, but two variants: single and comparative.

Changes: previously, some explanation functions also used supporting methods called “sentence builders”. They have been now integrated in the explanation functions themselves to reduce redundancy.

The explanation functions are the following:

- **Popularity (type=0)**
  - **popularity_one**
    Inputs: recipe
    The popularity_one explanation function shows the average rating of the given recipe and the number of ratings.
    Changes: previously, this function only provided a static explanation by assuming that the given recipe was very popular without giving any additional information.

  - **popularity_two**
    Inputs: recipeA, recipeB
    The popularity_two explanation function shows a comparison of recipeA’s rating count with respect to recipeB’s. Their respective average ratings are also shown.
    Changes: previously, this function only compared the recipes’ rating counts without showing them.

- **Food Goals (type=1)**
  - **foodGoals_one**
    Inputs: recipe, user
    The foodGoals_one explanation function takes into account the user’s sex and goal to check if the given recipe may be a good fit for the user. Depending on the user’s sex, a different value of daily calorie intake is used (2500 Kcal for men, 2000 Kcal for women); a single meal is assumed to contain 40% of such daily intake. This quantity is compared with the recipe’s calories taking the user’s goal into account.
    Changes: previously, this function didn’t check the recipe’s calories, but only said that the given recipe was a good choice for the user regardless of their goal. The user’s sex was also not taken into account. The daily calorie intake was assumed to be 1900 Kcal.

  - **foodGoals_two**
    Inputs: recipeA, recipeB, user

The foodGoals_two explanation function works similarly to foodGoals_one, but compares the recipes' calories in order to pick the recipe that may be a better fit for the user.
Changes: previously, the user's sex was also not taken into account. The daily calorie intake was assumed to be 1900 Kcal.

- **Food Preferences (type=2)**
    - **foodPreferences_one**
      Inputs: userRestrictions, listRestrictions, restrictions, recipe
      The foodPreferences_one explanation function checks if a given recipe follows the user's restrictions. If they are all respected by the recipe, the explanation provides information for all restrictions (loaded from Restrictions.json); otherwise, the restrictions that are not followed by the recipe are listed. Also, if no user restriction is specified (-1) or if the specified restrictions are not within the list of restrictions handled by the system (-2), respective error codes are returned.
      Changes: previously, this function only handled a single user restriction and did not check if the recipe actually respected such restriction, but rather said that the recipe respected it regardless.

    - **foodPreferences_two**
      Inputs: userRestrictions, listRestrictions, restrictions, recipeA, recipeB
      The foodPreferences_two explanation function calls foodPreferences_one on two recipes. If foodPreferences_one returns an error code (-1 or -2), foodPreferences_two returns the same one.
      Changes: previously, this function only handled a single user restriction and did not check if the recipes actually respected such restriction, but rather said that the recipes respected it regardless.

- **Food Features (type=3)**
    - **foodFeatures**
      Inputs: recipeA, recipeB, nutrients
      If recipeB is None, the foodFeatures explanation function compares the amounts of various nutrients (in grams) in recipeA with 40% of the respective daily reference intake (assumed to be a single meal intake). Otherwise, the amounts of such nutrients are compared between the two recipes. Two lists are also returned: greatList and smallList. They respectively contain the nutrients that exceed and subceed the average meal intake's (or recipeB's) respective quantity.
      Changes: previously, foodFeatures was divided in foodFeatures_one and foodFeatures_two. They have now been joined to reduce redundancy. Furthermore, foodFeatures_one used to compare the amount of nutrients in a recipe to the whole daily reference intake, which resulted in most recipes not being able to exceed such quantity. Other small changes were made to further reduce redundancies by parametrizing instructions.

- **User Skills (type=4)**
    - **convertRecipeDifficulty**
      Inputs: recipe_difficulty
      The convertRecipeDifficulty function is used in userAge_one and userAge_two to convert an Italian string for a recipe difficulty (Molto facile, Facile, Media, Difficile, Molto Difficile)

or a number in a 1-5 scale into the corresponding English translation (very simple, simple, quite simple, difficult, very difficult) and number in a 1-5 scale.
Changes: this is a new function. Previously, this conversion was only done in userSkills_two.

- **userSkills_one**
  Inputs: user_skills, recipe
  The userSkills_one explanation function shows a comparison between the user's skills and the recipe's difficulty.
  Changes: previously, this function didn't actually compare the user's skills and the recipe's difficulty, but rather assumed that the recipe's difficulty was the same as the user's skills.

- **userSkills_two**
  Inputs: user_skills, recipeA, recipeB
  The userSkills_two explanation function shows a comparison between two recipes' difficulties and then compares the easier recipe's difficulty to the user's skills.
  Changes: previously, this function didn't actually compare the user's skills and the easier recipe's difficulty, but rather assumed that the easier recipe's difficulty was the same as the user's skills.

- **Food Features Health Risk/Benefit (type=5 risk; type=6 benefit)**
  - **compareCalories**
    Inputs: recipeA_name, recipeB_name, recipeA_cal, recipeB_cal
    The compareCalories function shows a comparison of the quantity of calories between two recipes.
    Changes: previously, this comparison was done multiple times in different functions. It is now done in this function to reduce redundancy.

  - **rankNutrientsOffsets**
    Inputs: recipeA, recipeB, nutrients
    The rankNutrientsOffsets function returns two lists: greatList contains recipeA's top 2 nutrients that surpass 40% of the reference daily intake (or recipeB if it is not None); smallList contains recipeA's 2 nutrients with the lowest quantities.
    Changes: this is a new function.

  - **checkClosestNutrientsToRI**
    Inputs: recipe, nutrients
    The checkClosestNutrientsToRI function returns a list containing the top 2 nutrients whose values in the given recipes are closest to 40% of their respective reference daily intake.
    Changes: this is a new function.

  - **foodFeatureHealthRiskBenefit**
    Inputs: recipeA, recipeB, nutrients, type
    The foodFeatureHealthRiskBenefit explanation function calls the foodFeatures function and appends its output to the explanation.
    Then, if the "type" parameter is "risks", the rankNutrientsOffsets function is called. If it returns a non-empty greatList, a random risk associated to a high assumption of each of those nutrients (loaded from Nutrient.json) is added to the explanation; otherwise, smallList is used instead, and a random risk associated to a low assumption of each of those nutrients (loaded from Nutrient.json) is added.

Otherwise, if the "type" parameter is "benefits", the checkClosestNutrientsToRI function is called. A random benefit associated to a healthy assumption of each of the nutrients returned by it (loaded from Nutrient.json) is added to the explanation.
Finally, if recipeB is not None, compareCalories is also called.
Changes: previously, this function was split in four: foodFeatureHealthRisk_one, foodFeatureHealthRisk_two, foodFeatureHealthBenefit_one and foodFeatureHealthBenefit_two. They have now been joined in a single function to reduce redundancy. Furthermore, those functions used to show a random risk (or benefit) for all nutrients, making the explanation quite long, and didn't account for risks related to a low assumption of nutrients.

- **User Features Health Risk (type=7)**
    - **userFeatureHealthRisk**
      Inputs: user, recipeA, recipeB, nutrients
      The userFeatureHealthRisk explanation function calls the rankNutrientsOffsets function and then connects recipeA with some of the user's characteristics (BMI, mood, depressed/stressed) to provide a fitting explanation about risks. If recipeB is not None, compareCalories is also called.
      Changes: previously, this function was split in userFeatureHealthRisk_one and userFeatureHealthRisk_two. They have now been joined to reduce redundancy. Furthermore, more checks on the recipe are done while making the explanation, in order to make sure that it fits more accurately.

- **User Features Health Benefits (type=8)**
    - **userFeatureHealthBenefits**
      Inputs: user, recipeA, recipeB, nutrients
      The userFeatureHealthBenefits explanation function calls the checkClosestNutrientsToRI function and then connects recipeA with some of the user's characteristics (BMI, mood, depressed/stressed) to provide a fitting explanation about benefits. If recipeB is not None, compareCalories is also called.
      Changes: previously, this function was split in userFeatureHealthBenefits_one and userFeatureHealthBenefits_two. They have now been joined to reduce redundancy. Furthermore, more checks on the recipe are done while making the explanation, in order to make sure that it fits more accurately.

- **User Time (type=9)**
    - **userTime_one**
      Inputs: user_time, recipe_values
      The userTime_one explanation function compares the user's preferred preparation time with the recipe's preparation time.
      Changes: previously, user_time and the recipe's preparation time were not actually compared, but just shown on their own.

    - **userTime_two**
      Inputs: user_time, recipeA_values, recipeB_values
      The userTime_two explanation function compares the preparation times of two recipes and shows the user's preferred preparation time.
      Changes: redudancy has been reduced.

- **User Costs (type=10)**
    - **convertCost**
      Inputs: cost
      The convertCost function is used in userCosts_one and userCosts_two to convert an Italian string for a recipe cost (Molto basso, Basso, Medio, Elevato) or a number in a 1-4 scale into the corresponding English translation (very low, low, medium, high) and number in a 1-4 scale.
      Changes: this is a new function. Previously, this conversion was done in userCosts_one and userCosts_two and was handled differently.

    - **userCosts_one**
      Inputs: user_cost, recipe_values
      The userCosts_one explanation function compares the user's preferred cost with the recipe's cost.
      Changes: previously, this function didn't actually compare the user's preferred cost and the recipe's cost, but rather assumed that the recipe's cost was the same as the user's preference.

    - **userCosts_two**
      Inputs: user_cost, recipeA_values, recipeB_values
      The userCosts_two explanation function compares the costs of two recipes and shows the user's preferred cost.
      Changes: redudancy has been reduced.

- **User Lifestyle (type=11)**
    - **rsa_score**
      Inputs: recipe_values
      The rsa_score function computes a score that establishes how healthy a given recipe is according to the FSA guidelines.

    - **getScores**
      Inputs: score
      The getScores function converts a FSA score into a string (very healthy, healthy, average healthy, unhealthy, very unhealthy) and a number in a 1-5 scale.

    - **userLifestyle_one**
      Inputs: user_health_lifestyle, user_health_condition, recipe_values
      The userLifestyle_one explanation function calls the rsa_score function to compute the given recipe's RSA score, then converts it with getScores and takes it into account while comparing the user's desired lifestyle with his current condition.
      Changes: redudancy has been reduced.

    - **userLifestyle_two**
      Inputs: user_health_condition, recipeA_values, recipeB_values
      The userLifestyle_two explanation function calls the rsa_score function to compute the given recipes' RSA scores, then converts them with getScores and compares them in relation to the user's current condition.
      Changes: redudancy has been reduced.

- **User Ingredients (type=12)**
    - **checkRecipeIngredientsInList**
      Inputs: ingredients, recipe_values
      The checkRecipeIngredientsInList function looks for specific ingredients that satisfy certain properties (e.g. user's favourite, rich in specific nutrients, sustainable…), listed in the "ingredients" parameter, in a given recipe. Such overlapping ingredients are returned.
      Changes: this is a new function. Previously, this was done redundantly in userIngredients_one and userIngredients_two. It has now been made more generic in order to be also used by different functions other than userIngredients_one and userIngredients_two.

    - **listFavIngredientsInRecipe**
      Inputs: favIngredientsInRecipe, recipe_values
      The listFavIngredientsInRecipe function is used in userIngredients_one and userIngredients_two to show the subset of the user's favourite ingredients that are contained in a given recipe.
      Changes: this is a new function. Previously, this was done redundantly in userIngredients_one and userIngredients_two.

    - **userIngredients_one**
      Inputs: user_ingredients, recipe_values
      The userIngredients_one explanation function calls the checkRecipeIngredientsInList function to check if a given recipe contains any of the user's favourite ingredients. If the returned list is not empty, the listFavIngredientsInRecipe function is called to provide an explanation. The list of favourite ingredients contained in the recipe is also returned.
      Changes: redundancy has been reduced.

    - **userIngredients_two**
      Inputs: user_ingredients, recipeA_values, recipeB_values
      The userIngredients_two explanation function calls the checkRecipeIngredientsInList function to check if the given recipes contain any of the user's favourite ingredients. If at least one of the returned lists is not empty, the listFavIngredientsInRecipe function is called on the two recipes to provide a comparative explanation. The lists of favourite ingredients contained in the recipes are also returned.
      Changes: redundancy has been reduced.

- **User Age (type=13)**
    - **convertAge**
      Inputs: user_age
      The convertAge function checks if the given user age is numeric instead of a string representing a range (U20, U30, U40, U50, U60, O60). If so, it is converted into one of such ranges in order to be compatible with the userAge_one and userAge_two functions.
      Changes: this is a new function. Previously, only the ranges listed before were accepted by the system.

    - **userAge_one**
      Inputs: user_age, recipe_values, richIn
      The userAge_one explanation function calls the convertAge function. Then, the checkRecipeIngredientsInList function is called in order to evaluate if ingredients that are

rich in specific nutrients (depending on the user's age group) are present in a given recipe. The provided explanation shows which of such nutrients are present in the recipe and which are lacking. Furthermore, a motivation of the importance of the nutrients needed in the specific age group is also provided.
Changes: previously, the lists of ingredients that are rich in specific nutrients and the explanations of the importance of such nutrients were defined redundantly in the code. They have now been moved into the RichIn.json file. Furthermore, this function used to show only the nutrients that are present in the given recipe, but not the ones that are lacking. Other small changes were made to further reduce redundancies.

  - **userAge_two**
    Inputs: user_age, recipeA_values, recipeB_values, richIn
    The userAge_two explanation function calls the convertAge function. Then, it calls userAge_one on the two recipes and concatenates their explanations.
    Changes: redundancy was greatly reduced by reusing userAge_one.

- **Sustainable Ingredients (type=14)**
  - **ingredientsSustainability_one**
    Inputs: recipe, sustainability
    The ingredientsSustainability_one explanation function looks for the ingredients with high and low sustainability (carbon footprint and water footprint, both stored in Sustainability.json) that are present in a given recipe by calling the checkRecipeIngredientsInList function. Such ingredients are then listed in the explanation. The respective lists are also returned.
    Changes: this is a new function.

  - **ingredientsSustainability_two**
    Inputs: recipeA, recipeB, sustainability
    The ingredientsSustainability_two explanation function calls the ingredientsSustainability_one function on two recipes and then compares the quantities of ingredients with high and low sustainability between the recipes. The lists of ingredients with high and low sustainability for both recipes are also returned.
    Changes: this is a new function.

- **Seasonal Ingredients (type=15)**
  - **ingredientsSeasonality_one**
    Inputs: recipe, season, seasonality
    The ingredientsSeasonality_one explanation function looks for the ingredients that are in season (stored in Seasonality.json) that are present in a given recipe by calling the checkRecipeIngredientsInList function. Ingredients are considered in season compared to the given "season" parameter, which can be given by the user; otherwise, it is automatically extracted from the current date. Such ingredients are then listed in the explanation. The list of seasonal ingredients is also returned.
    Changes: this is a new function.

  - **ingredientsSeasonality_two**
    Inputs: recipeA, recipeB, season, seasonality

The ingredientsSeasonality_two explanation function calls the ingredientsSeasonality_one function on two recipes and then compares the quantities of seasonal ingredients between the recipes. The list of seasonal ingredients for both recipes are also returned.
Changes: this is a new function.

- **Dopamine Ingredients (type=16)**
  - **ingredientsDopamine_one**
    Inputs: recipe, dopamine
    The ingredientsDopamine_one explanation function looks for the dopamine-inducing ingredients (stored in Dopamine.json) that are present in a given recipe by calling the checkRecipeIngredientsInList function. Such ingredients are then listed in the explanation. The list of dopamine-inducing ingredients is also returned.
    Changes: this is a new function.

  - **ingredientsDopamine_two**
    Inputs: recipeA, recipeB, dopamine
    The ingredientsDopamine_two explanation function calls the ingredientsDopamine_one function on two recipes and then compares the quantities of dopamine-inducing ingredients between the recipes. The list of dopamine-inducing ingredients for both recipes are also returned.
    Changes: this is a new function.

- **Description (type=17)**
  The description explanation type does not have any dedicated function. It is handled directly in get_str_exp (and smartExplanation) by appending the recipes' descriptions to the explanation.

- **Smart Explanation (type=18)**
  - **smartExplanation**
    Inputs: user, recipeA, recipeB, listRestrictions, nutrients, restrictions, richIn, sustainability, seasonality, dopamine
    The smartExplanation explanation function provides a personalized explanation for one or two recipes based on various user characteristics.
    As shown in the Smart Explanation flowchart, this function checks different user parameters in a specific order, starting with restrictions, goal and other psychophysical information. Then, if the user does not have a specific goal and/or need to change their lifestyle, the function looks for preferences like favourite ingredients, preferred recipe difficulty and so on. If no preferences are found and no other type of explanation holds, the fallback is a basic description of the recipe(s) and their popularity.
    Changes: this is a new function.

cmd_expl.py

The **cmd_expl.py** script is a client that can be invoked from command line in order to send a HTTP GET request to the server (assuming it is already running).

Using the argparse module, various arguments are added, each one corresponding to a HTTP GET request argument expected by the server. The most important parameters were set to “required”: type, style and imgurl1 (recipeA). The restrictions and the user ingredients arguments can accept more than one value.

Argument	Expected values	Required	Multiple values
--type	[0,18] (see list below)	Yes	-
--style	-1 for single and comparative; 0 for single explanations; 1 for comparative explanation	Yes	-
--imgurl1	URL of recipeA’s image from GialloZafferano	Yes	-
--imgurl2	URL of recipeB’s image from GialloZafferano	-	-
--user_age	numerical age or U20/U30/U40/U50/U60/O60	-	-
--mood	bad/good/neutral	-	-
--stress	yes/no	-	-
--depression	yes/no	-	-
--bmi	over/under/normal	-	-
--health_style	1/2/3/4/5	-	-
--health_condition	1/2/3/4/5	-	-
--activity	low/high/normal	-	-
--sleep	low/good	-	-
--difficulty	1/2/3/4/5	-	-
--user_time	[1,200] mins; 0=no contraints	-	-
--user_cost	1/2/3/4/5; 5=not important	-	-
--goal	lose/gain/no	-	-
--restr	vegetarian/lactose-free/gluten-free/low-nickel/light	-	Yes
--user_ingredients	zero, one or more favourite ingredients	-	Yes
--sex	m/f	-	-
--season	winter/spring/summer/autumn	-	-

Explanation types:

0. Popularity
1. Food Goals
2. Food Preferences
3. Food Features
4. User Skills
5. Food Feature Health Risk
6. Food Feature Health Benefits
7. User Feature Health Risk
8. User Feature Health Benefits
9. User Time
10. User Costs

11. User Lifestyle
12. User Ingredients
13. User Age
14. Ingredients Sustainability
15. Ingredients Seasonality
16. Ingredients Dopamine
17. Description
18. Smart Explanation

Command line request example:

```
python cmd_expl.py --type 2 --style -1 --mood neutral --stress no --
depression no --bmi over --activity low --goal lose --sleep low --restr
gluten-free vegetarian --imgurl1
https%3A%2F%2Fwww.giallozafferano.it%2Fimages%2Fricette%2F201%2F20113%2Ff
oto_hd%2Fhd650x433_wm.jpg --imgurl2
https%3A%2F%2Fwww.giallozafferano.it%2Fimages%2Fricette%2F176%2F17635%2Ff
oto_hd%2Fhd650x433_wm.jpg --difficulty 1 --user_time 0 --user_cost 5 --
health_style 5 --health_condition 5 --user_ingredients oil carrot --
user_age U40 --season winter --sex m
```

When the script is invoked from command line, all specified arguments are parsed and used to create a string that is then sent to the server as a HTTP GET request. The response received by the server is then printed on command line and also saved in a json file (explanation.json).

Resulting request sent to the server:

```
http://127.0.0.1:5003/exp/?type=2&style=-
1&mood=neutral&stress=no&depression=no&bmi=over&activity=low&goal=lose&sl
eep=low&restr=gluten-
free,vegetarian&imgurl1=https%3A%2F%2Fwww.giallozafferano.it%2Fimages%2Fr
icette%2F201%2F20113%2Ffoto_hd%2Fhd650x433_wm.jpg&imgurl2=https%3A%2F%2Fw
ww.giallozafferano.it%2Fimages%2Fricette%2F176%2F17635%2Ffoto_hd%2Fhd650x
433_wm.jpg&difficulty=1&user_time=0&user_cost=5&health_style=5&health_con
dition=5&user_ingredients=oil,carrot&user_age=U40&season=winter&sex=m
```

Changes: this is a new script.