D2.1b API for external contributors

Deliverable information	
WP	WP2
Document dissemination level	PU - public
Deliverable type	SW - software
Lead	UCM
Contributors	All partners
Document status	Version for Zenodo
Document version	V1.0
Date	28/09/2022

Document History

Version	Release date	Summary of changes	Partner
V0.1		First draft released	
V1.0		Revision	

Project information

Project start date: 1st of March 2021

Project Duration: 36 months

Project website: https://isee4xai.com/

iSee consortium

UCM	UNIVERSIDAD COMPLUTENSE DE MADRID	SPAIN
RGU	ROBERT GORDON UNIVERSITY	SCOTLAND
BT	BT FRANCE	FRANCE
UCC	UNIVERSITY COLLEGE CORK	IRELAND

Contact: hello@isee4xai.com

Project Coordinator:

Professor M Belen Díaz Agudo
Instituto de Tecnología del Conocimiento
Facultad de Informática,
Universidad Complutense de Madrid
C/ Profesor Jose Garcia Santesmases, 9
Ciudad Universitaria
28040 – Madrid, Spain

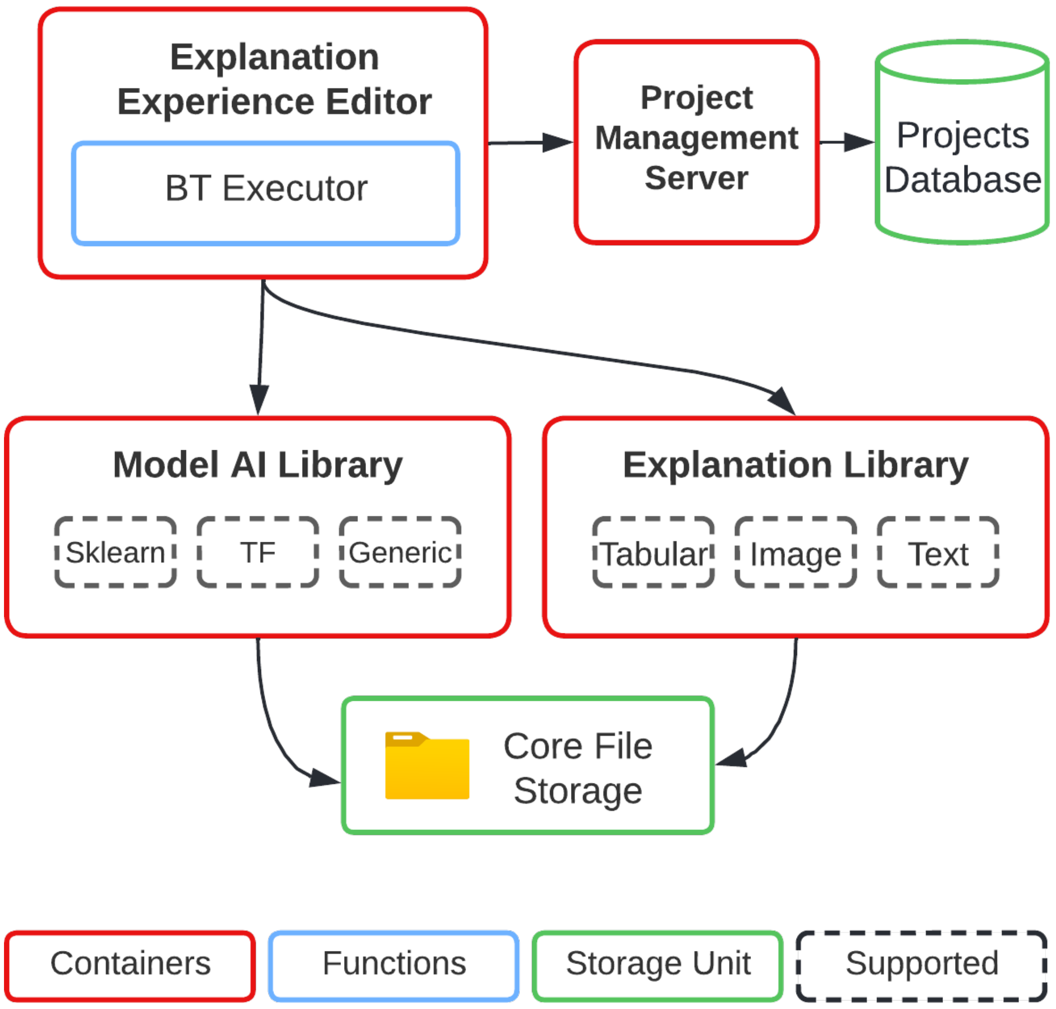
Summary

This document describes the software that provides the API for external contributions to the iSee platform. This software is available through several repositories that are described next. First section provides a global description of the architecture, whereas the following ones contain the user/developer documentation for each module.

Table of Contents

Global architecture

As illustrated in the following figure, there are four different containers in our architecture to where external developers can contribute.



The **Explanation Library** provides the XAI methods for the generation of explanations, while the **AI Model Library** contains the methods for the management of the models to be explained. Both of these containers utilize well-known libraries oriented to the development of machine-learning models, such as Scikit-learn, TensorFlow, Pytorch and so forth. The containers have direct access to a core file storage unit where the models and their additional configuration data are stored. Both libraries provide unified Rest Services APIs to support not only the functionality of the Explanation Experience Editor but also ad-hoc integration with external software. Both APIs have been developed in Python using the Flask Restful microframework.

The **Explanation Library** provides an unified layer to access and execute the explanation methods. These methods are implemented as an API that acts as a wrapper of third-party modules and libraries, following the dependency injection pattern.

To execute these explanation methods our platform also requires an additional module to manage the ML models to be executed and explained. For this task, we have developed the **AI Model Library**, that offers an additional API that allows uploading, updating, and managing the files associated with a ML model, as well as executing them. This module supports models exported from both Scikit-learn or TensorFlow. Thus, the Explanation Library can later access these files to generate explanations.

This API of the AI Model Library is composed by two different sub-APIs that support either Tensorflow or ScikitLearn. Both share common features that are abstracted in the "generic" API.

Generic API of the AI Model Library

The following document will explain the various functions needed to implement in order to communicate with the model library API for the iSee project.

Requirements

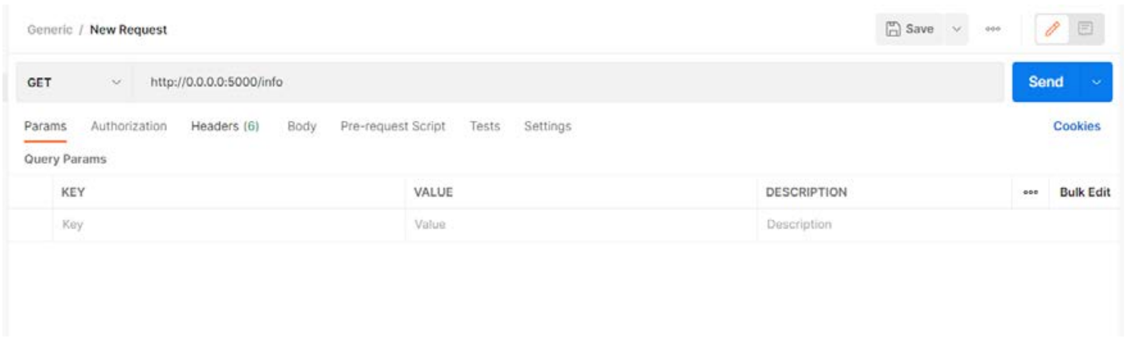
The client must implement the following capabilities:

Retrieving a model's parameters

In order to retrieve the parameters provided when uploading a model to the API the function `info` must be called in `GET` mode.

The function will return a json with the parameters.

Here we have an example using Postman:



Generic / New Request

GET http://0.0.0.0:5000/info

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

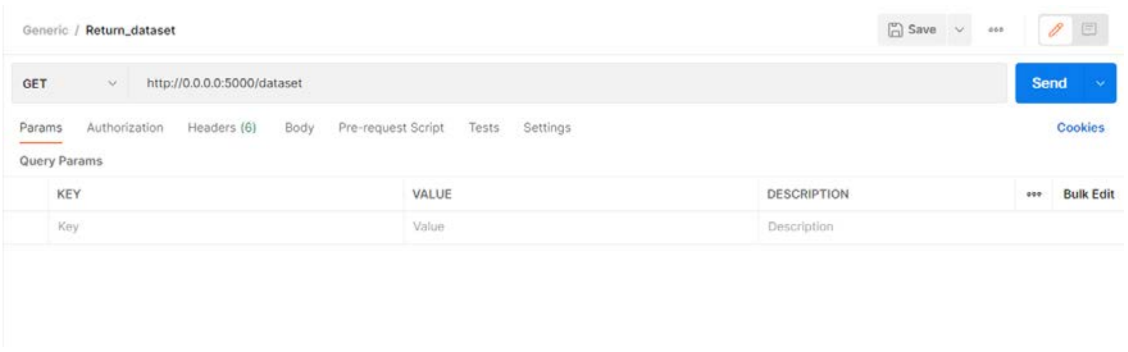
Retrieving a model's parameters

Retrieve a dataset (optional)

In order to obtain the dataset used to train a model from the API the function `dataset` must be called in `GET` mode.

The function will return the dataset associated with the model.

Here we have an example using Postman:



Generic / Return_dataset

GET http://0.0.0.0:5000/dataset

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Retrieving a model's dataset

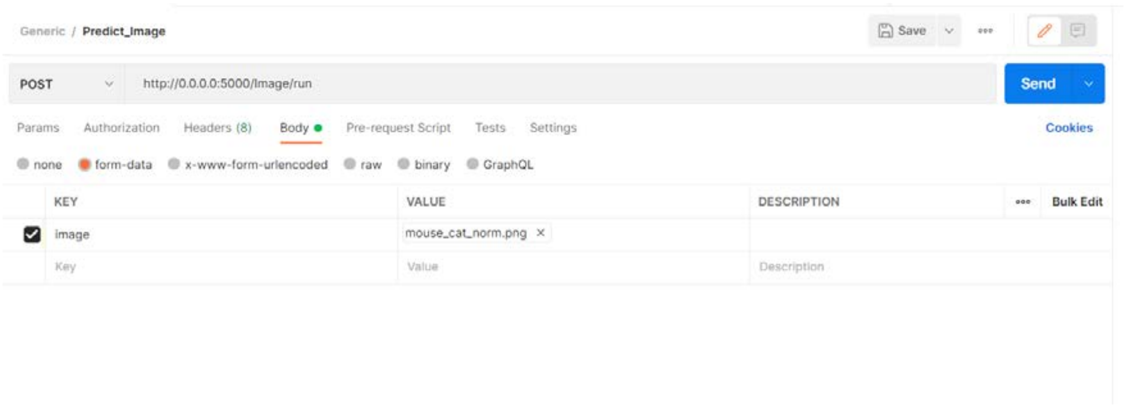
Predicting with an image

If the model uses images as input, in order to make a prediction based on an image using a model uploaded to the API the function `/Image/run` must be called in `POST` mode using the following parameters in the Body **form-data** section:

- `image`: this field must contain a file which corresponds to the image that we want to pass to the model.

The function will return a message with the model prediction.

Here we have an example using Postman:


Generic / Predict_Image

POST http://0.0.0.0:5000/Image/run

Params | Authorization | Headers (8) | Body ● | Pre-request Script | Tests | Settings | Cookies

none | ● form-data | x-www-form-urlencoded | raw | binary | GraphQL

	KEY	VALUE	DESCRIPTION
✓	image	mouse_cat_norm.png ×	
	Key	Value	Description

Predicting an image

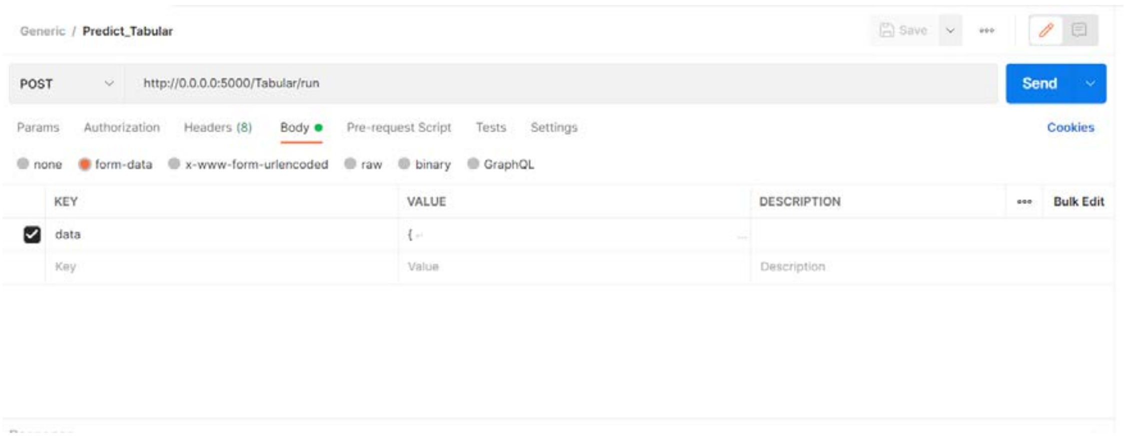
Predicting with a tabular set

If the model uses a dataset as input, in order to make a prediction based on a dataset using a model uploaded to the API the function `/Tabular/run` must be called in `POST` mode using the following parameters in the Body **form-data** section:

- `data`: this field must contain a text formatted as a json with a field named *"instance"* followed by an array containing the data to be passed to the model.

The function will return a message with the model prediction.

Here we have an example using Postman:


Generic / Predict_Tabular

POST http://0.0.0.0:5000/Tabular/run

Params | Authorization | Headers (8) | Body ● | Pre-request Script | Tests | Settings | Cookies

none | ● form-data | x-www-form-urlencoded | raw | binary | GraphQL

	KEY	VALUE	DESCRIPTION
✓	data	{	
	Key	Value	Description

Predicting with a dataset

Scikit-learn API of the Model Library

The following document will explain how to call the various functions implemented in the model library API for the iSee project, in particular we will describe the behaviour for the sklearn version.

Launching the API

The API can be executed in two ways depending if an storage path for the models is passed as a parameter or not:

Without an storage path

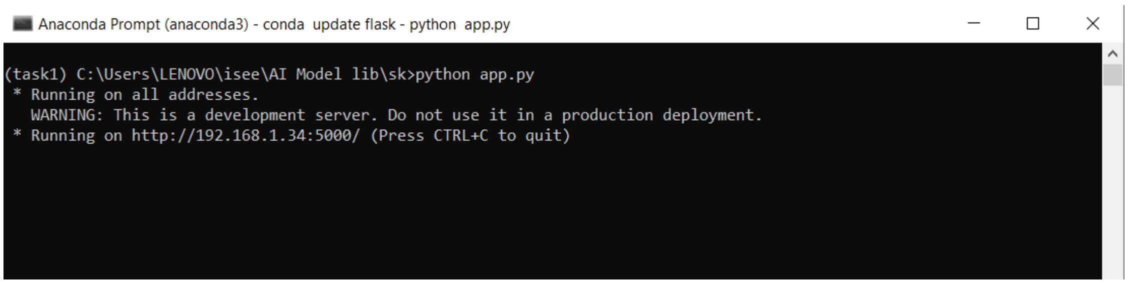
Requirements

- The directory containing the app.py file **must** also contain a **folder named Models**.
- The libraries in the requirements.txt must be installed.

Running the API

1. A console must be opened in the directory containing the app.py file.
2. Run the following command `python app.py`.
3. The API should be running in a localhost and request can now be send.

Here's an example:


Anaconda Prompt (anaconda3) - conda update flask - python app.py

```
(task1) C:\Users\LENOVO\isee\AI Model lib\sk>python app.py
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://192.168.1.34:5000/ (Press CTRL+C to quit)
```

Run no path

With an storage path

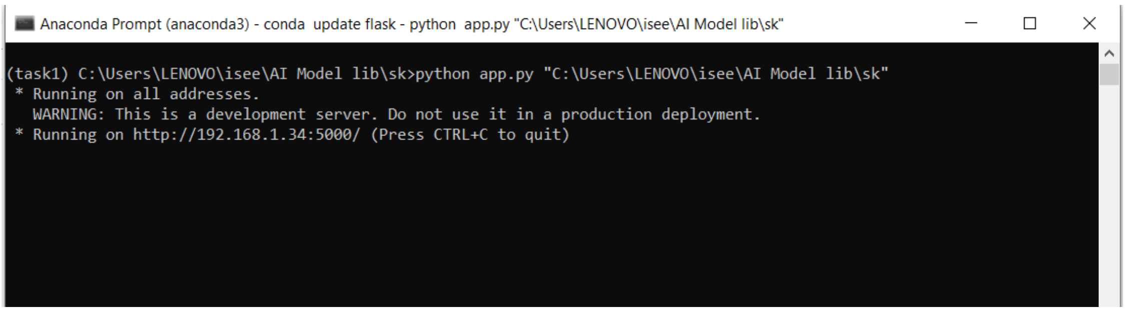
Requirements

- The directory passed as an argument must exists.
- The libraries in the requirements.txt must be installed.

Running the API

1. A console must be opened in the directory containing the app.py file.
2. Run the following command `python app.py "path"`. **REMINDER**: If the path contains spaces it must be written between quotation marks.
3. The API should be running in a localhost and request can now be send.

Here's an example:


Anaconda Prompt (anaconda3) - conda update flask - python app.py "C:\Users\LENOVO\isee\AI Model lib\sk"

```
(task1) C:\Users\LENOVO\isee\AI Model lib\sk>python app.py "C:\Users\LENOVO\isee\AI Model lib\sk"
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://192.168.1.34:5000/ (Press CTRL+C to quit)
```

Run with path

Basic functions

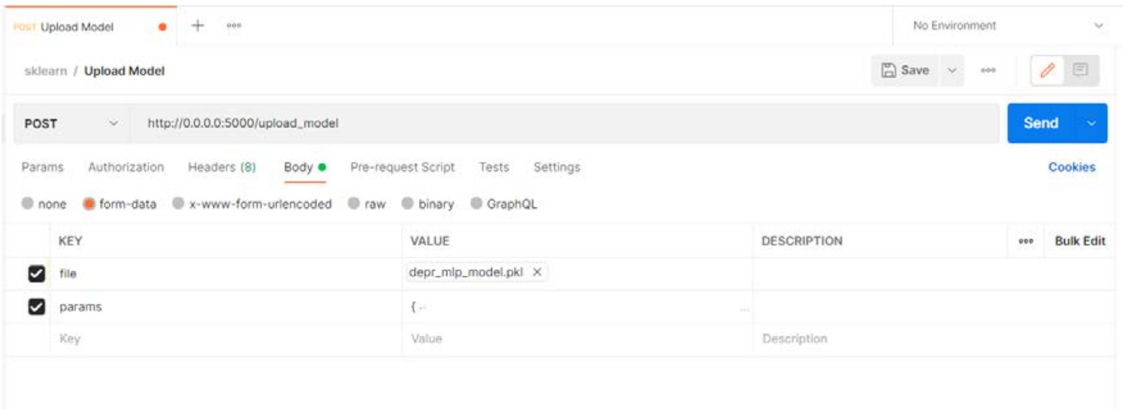
Uploading a model

In order to upload a model to the API the function `upload_model` must be called in `POST` mode using the following parameters in the Body **form-data** section:

- `file`: this field must contain a the model that we wish to upload.
- `params`: this field must contain a json formatted text with the various properties of the model.
- `id`(**optional**): this field contains the id which will be used to refer to the uploaded model in the rest of the functions. If it is left blank a random id will be assigned.

The function will return the **id** assigned to the model.

Here we have an example using Postman:


POST Upload Model

sklearn / Upload Model

POST http://0.0.0.0:5000/upload_model

Params · Authorization · Headers (8) · Body · Pre-request Script · Tests · Settings · Cookies

none · [✓ form-data] · x-www-form-urlencoded · raw · binary · GraphQL

	KEY	VALUE	DESCRIPTION
✓	file	depr_mlp_model.pkl	
✓	params	{	
	Key	Value	Description

Uploading a model

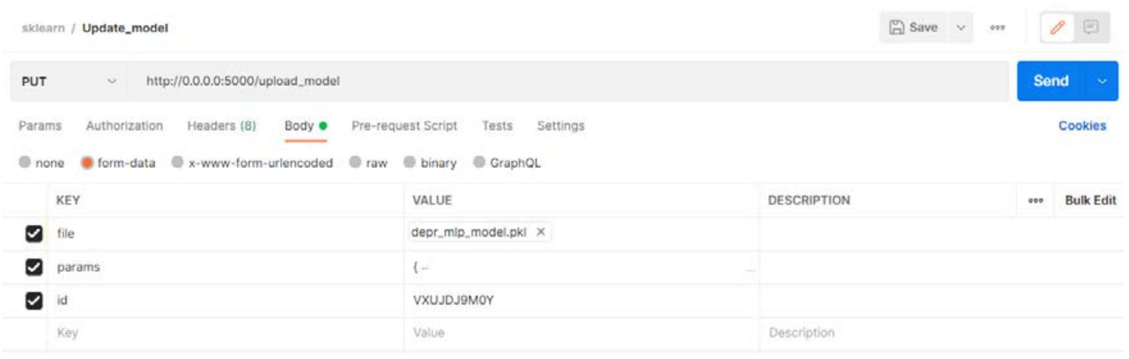
Updating a model

In order to update an existing model to the API the function `upload_model` must be called in `PUT` mode using the following parameters in the Body **form-data** section:

- `file`: this field must contain a the model that we wish to update.
- `params`: this field must contain a json formatted text with the various properties of the model.
- `id`: this field must contain the id of the model that we wish to update.

The function will return a message confirming the update of the model.

Here we have an example using Postman:


sklearn / Update_model

PUT http://0.0.0.0:5000/upload_model

Body: form-data

KEY	VALUE	DESCRIPTION
✓ file	depr_mlp_model.pkl	
✓ params	{…	
✓ id	VXUJDJ9M0Y	
Key	Value	Description

Updating a model

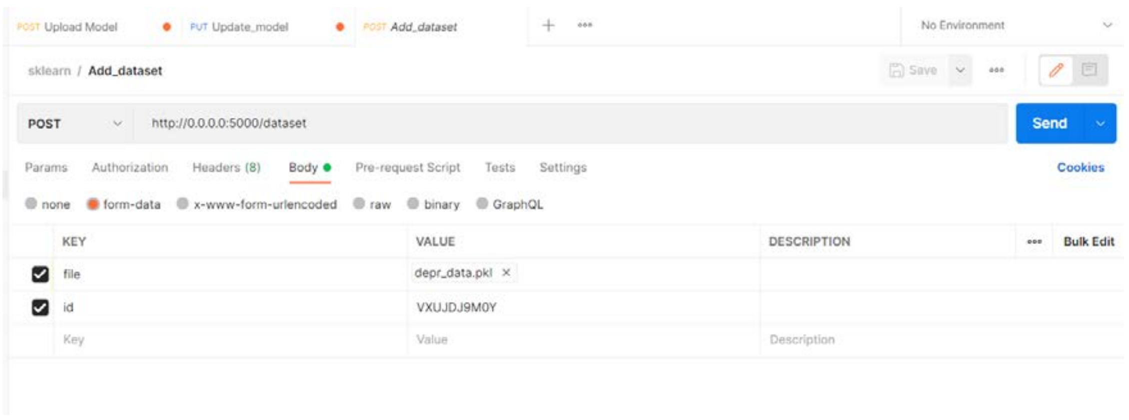
Uploading a dataset

In order to upload a model to the API the function `dataset` must be called in `POST` mode using the following parameters in the Body **form-data** section:

- `file`: this field must contain a *.pkl* file which corresponds to the dataset that we want to upload.
- `id`: this field must contain the id of the model whose dataset we are uploading.

The function will return a message confirming the upload of the dataset.

Here we have an example using Postman:


POST Upload Model | PUT Update_model | POST Add_dataset

sklearn / Add_dataset

POST http://0.0.0.0:5000/dataset

Body: form-data

KEY	VALUE	DESCRIPTION
✓ file	depr_data.pkl	
✓ id	VXUJDJ9M0Y	
Key	Value	Description

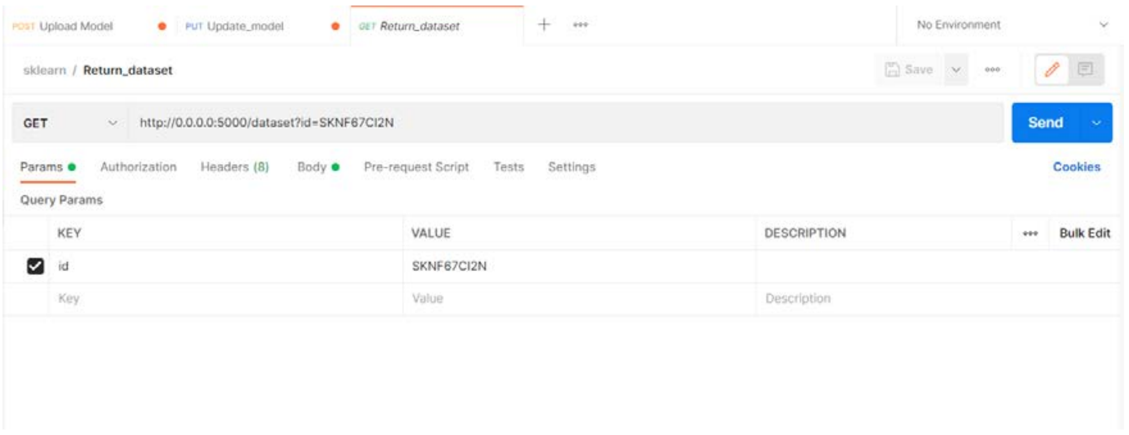
Uploading a dataset

Retrieve a dataset

In order to obtain the dataset used to train a model from the API the function `dataset` must be called in `GET` mode using the following parameters in the **URL parameter** section: - `id`: this field must contain the id of the model whose dataset we wish to retrieve.

The function will return the dataset associated with the id.

Here we have an example using Postman:



sklearn / Return_dataset

GET http://0.0.0.0:5000/dataset?id=SKNF67CI2N

Query Params

	KEY	VALUE	DESCRIPTION
✓	id	SKNF67CI2N	
	Key	Value	Description

Downloading a dataset

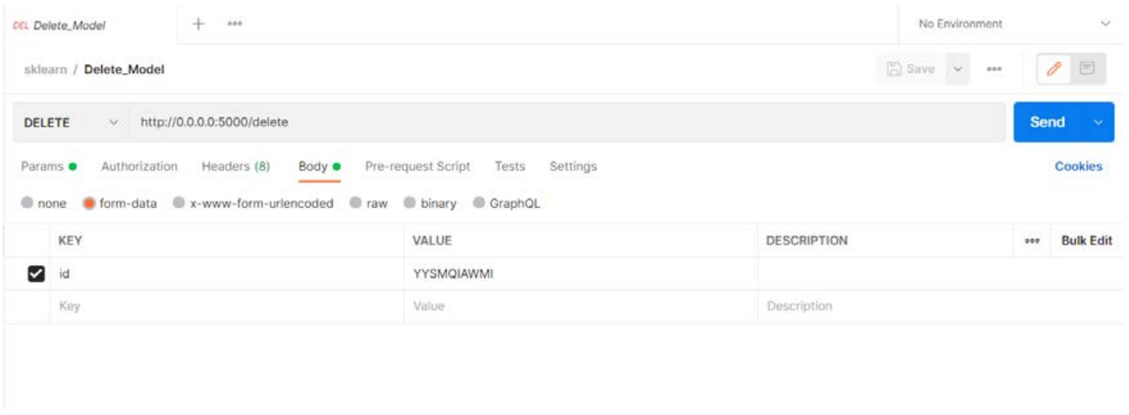
Deleting a model

In order to delete an uploaded model from the API the function `delete` must be called in `DELETE` mode using the following parameters in the Body **form-data** section:

- `id`: this field must contain the id of the model we wish to delete.

The function will return a message confirming the deletion.

Here we have an example using Postman:



sklearn / Delete_Model

DELETE http://0.0.0.0:5000/delete

Body: form-data

	KEY	VALUE	DESCRIPTION
✓	id	YYSMQIAWMI	
	Key	Value	Description

Deleting a model

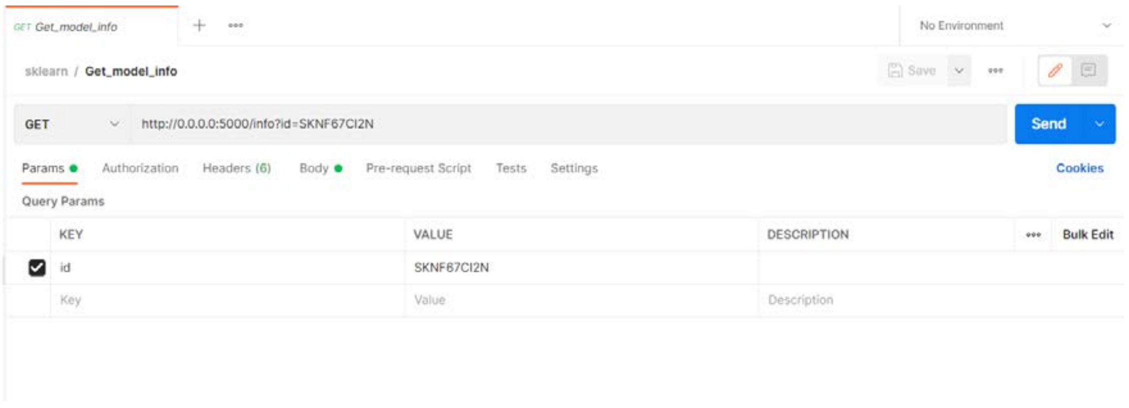
Retrieving a model’s parameters

In order to retrieve the parameters provided when uploading a model to the API the function `info` must be called in `GET` mode using the following parameters in the **URL parameter** section:

- `id`: this field must contain the id of the model whose parameters we wish to retrieve.

The function will return a json with the parameters.

Here we have an example using Postman:



GET Get_model_info

No Environment

sklearn / Get_model_info

GET http://0.0.0.0:5000/info?id=SKNF67CI2N

Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

	KEY	VALUE	DESCRIPTION
✓	id	SKNF67CI2N	
	Key	Value	Description

Retrieving a model's parameters

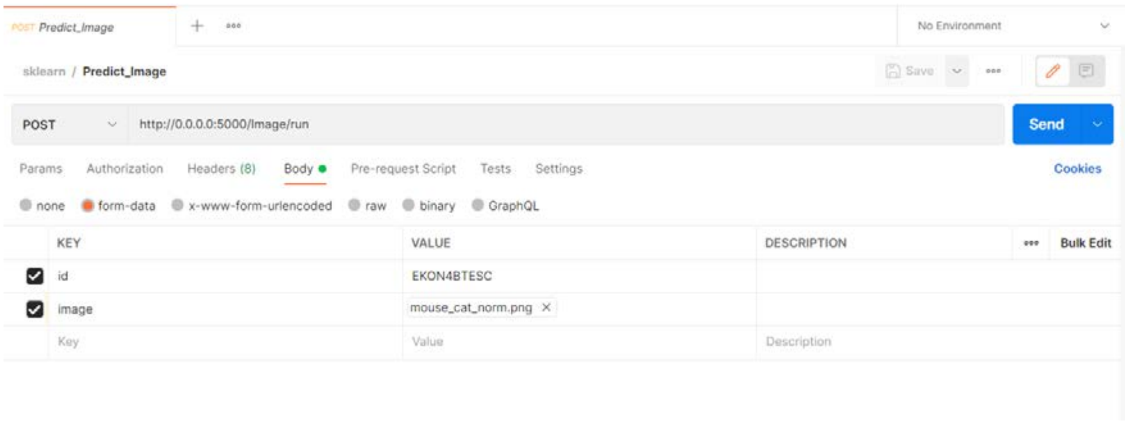
Predicting with an image

In order to make a prediction based on an image using a model uploaded to the API the function `/Image/run` must be called in `POST` mode using the following parameters in the Body **form-data** section:

- `image`: this field must contain a file which corresponds to the image that we want to pass to the model.
- `id`: this field must contain the id of the model that we want to use to make the prediction.

The function will return a message with the model prediction.

Here we have an example using Postman:



POST Predict_Image

No Environment

sklearn / Predict_Image

POST http://0.0.0.0:5000/Image/run

Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none ● form-data x-www-form-urlencoded raw binary GraphQL

	KEY	VALUE	DESCRIPTION
✓	id	EKON4BTESC	
✓	image	mouse_cat_norm.png	
	Key	Value	Description

Predicting an image

Predicting with a tabular set

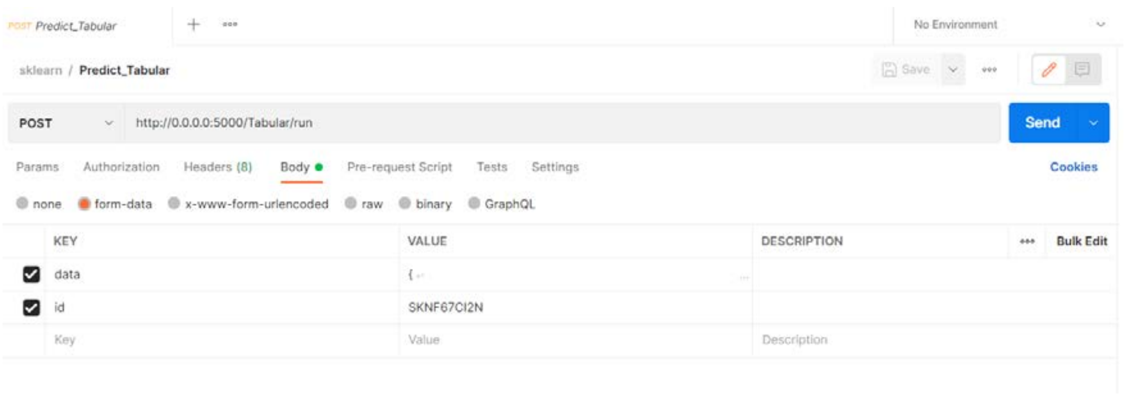
In order to make a prediction based on a dataset using a model uploaded to the API the function `/Tabular/run` must be called in `POST` mode using the following parameters in the Body **form-data** section:

- `data`: this field must contain a text formatted as a json with a field named *"instance"* followed by an array containing the data to be passed to the model.

- `id`: this field must contain the id of the model that we want to use to make the prediction.

The function will return a message with the model prediction.

Here we have an example using Postman:



POST Predict_Tabular

sklearn / Predict_Tabular

POST http://0.0.0.0:5000/Tabular/run

Body: form-data

KEY	VALUE	DESCRIPTION
✓ data	{	
✓ id	SKNF67CI2N	
Key	Value	Description

Predicting an image

TensorFlow API of the Model Library

The following document will explain how to call the various functions implemented in the model library API for the iSee project, in particular we will describe the behaviour for the TensorFlow version.

Launching the API

The API can be executed in two ways depending if an storage path for the models is passed as a parameter or not:

Without an storage path

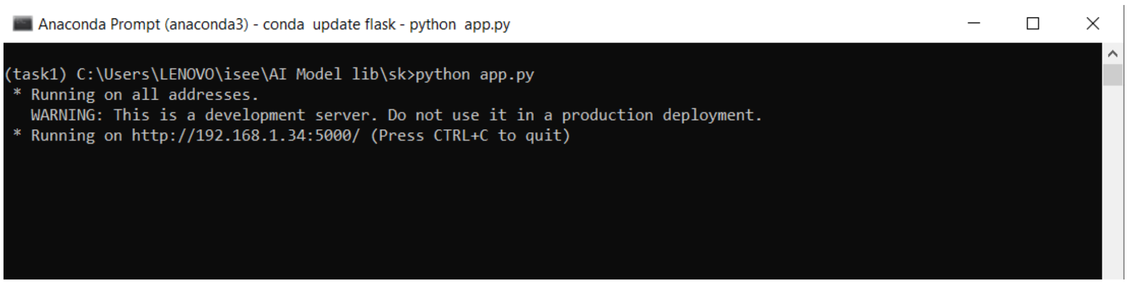
Requirements

- The directory containing the app.py file **must** also contain a **folder named Models**.
- The libraries in the requirements.txt must be installed.

Running the API

4. A console must be opened in the directory containing the app.py file.
5. Run the following command `python app.py`.
6. The API should be running in a localhost and request can now be send.

Here's an example:


Anaconda Prompt (anaconda3) - conda update flask - python app.py

```
(task1) C:\Users\LENOVO\isee\AI Model lib\sk>python app.py
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://192.168.1.34:5000/ (Press CTRL+C to quit)
```

Run no path

With an storage path

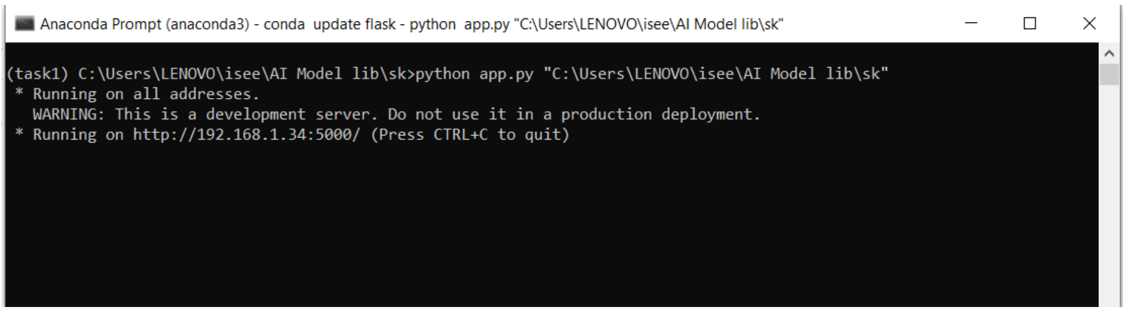
Requirements

- The directory passed as an argument must exists.
- The libraries in the requirements.txt must be installed.

Running the API

7. A console must be opened in the directory containing the app.py file.
8. Run the following command `python app.py "path"`. **REMINDER**: If the path contains spaces it must be written between quotation marks.
9. The API should be running in a localhost and request can now be send.

Here's an example:



```
Anaconda Prompt (anaconda3) - conda  update flask - python  app.py "C:\Users\LENOVO\isee\AI Model lib\sk"

(task1) C:\Users\LENOVO\isee\AI Model lib\sk>python app.py "C:\Users\LENOVO\isee\AI Model lib\sk"
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://192.168.1.34:5000/ (Press CTRL+C to quit)
```

Run with path

Basic functions

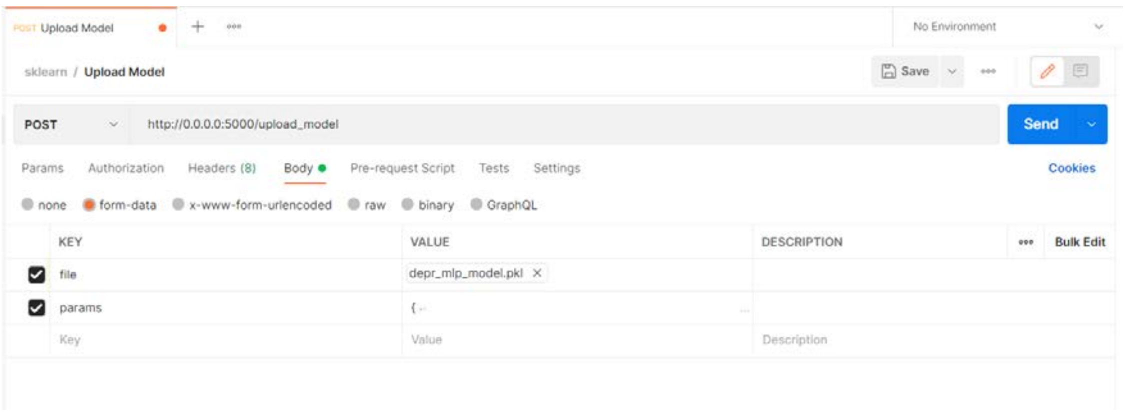
Uploading a model

In order to upload a model to the API the function `upload_model` must be called in `POST` mode using the following parameters in the Body **form-data** section:

- `file`: this field must contain a the model that we wish to upload.
- `params`: this field must contain a json formatted text with the various properties of the model.
- `id`(**optional**): this field contains the id which will be used to refer to the uploaded model in the rest of the functions. If it is left blank a random id will be assigned.

The function will return the **id** assigned to the model.

Here we have an example using Postman:



POST http://0.0.0.0:5000/upload_model

Body: form-data

	KEY	VALUE	DESCRIPTION
✓	file	depr_mlp_model.pkl	
✓	params	{	
	Key	Value	Description

Uploading a model

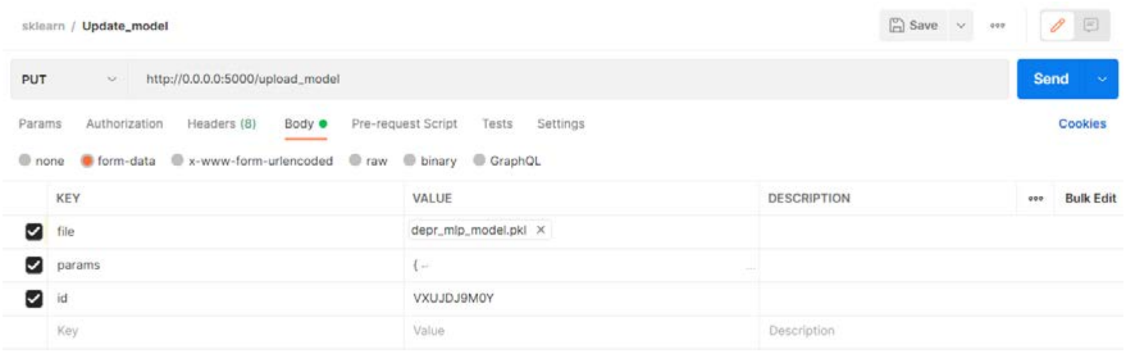
Updating a model

In order to update an existing model to the API the function `upload_model` must be called in `PUT` mode using the following parameters in the Body **form-data** section:

- `file`: this field must contain a the model that we wish to update.
- `params`: this field must contain a json formatted text with the various properties of the model.
- `id`: this field must contain the id of the model that we wish to update.

The function will return a message confirming the update of the model.

Here we have an example using Postman:



sklearn / Update_model

PUT http://0.0.0.0:5000/upload_model

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none ● form-data x-www-form-urlencoded raw binary GraphQL

	KEY	VALUE	DESCRIPTION
✓	file	depr_mlp_model.pkl	
✓	params	{...	
✓	id	VXUJDJ9M0Y	
	Key	Value	Description

Updating a model

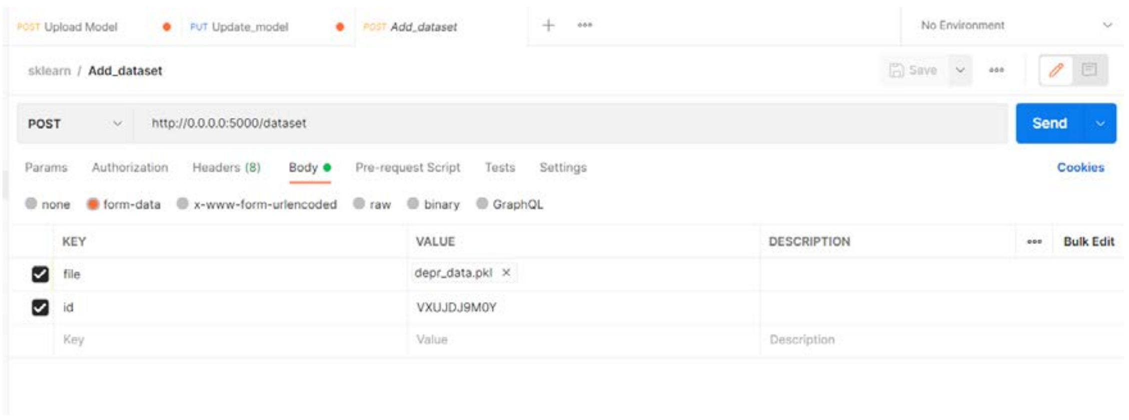
Uploading a dataset

In order to upload a model to the API the function `dataset` must be called in `POST` mode using the following parameters in the Body **form-data** section:

- `file`: this field must contain a *.pkl* file which corresponds to the dataset that we want to upload.
- `id`: this field must contain the id of the model whose dataset we are uploading.

The function will return a message confirming the upload of the dataset.

Here we have an example using Postman:



POST Upload Model | PUT Update_model | POST Add_dataset | No Environment

sklearn / Add_dataset

POST http://0.0.0.0:5000/dataset

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none ● form-data x-www-form-urlencoded raw binary GraphQL

	KEY	VALUE	DESCRIPTION
✓	file	depr_data.pkl	
✓	id	VXUJDJ9M0Y	
	Key	Value	Description

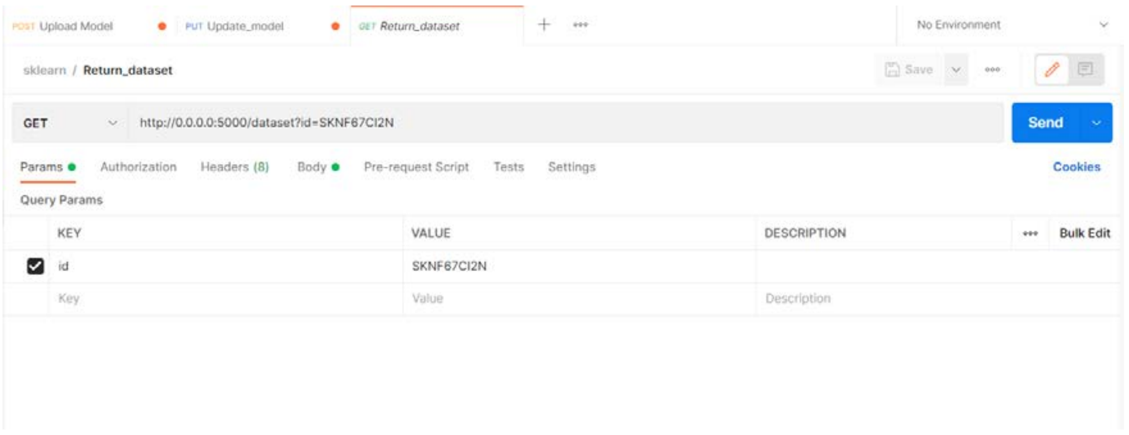
Uploading a dataset

Retrieve a dataset

In order to obtain the dataset used to train a model from the API the function `dataset` must be called in `GET` mode using the following parameters in the **URL parameter** section: - `id`: this field must contain the id of the model whose dataset we wish to retrieve.

The function will return the dataset associated with the id.

Here we have an example using Postman:



GET http://0.0.0.0:5000/dataset?id=SKNF67CI2N

	KEY	VALUE	DESCRIPTION
✓	id	SKNF67CI2N	
	Key	Value	Description

Uploading a dataset

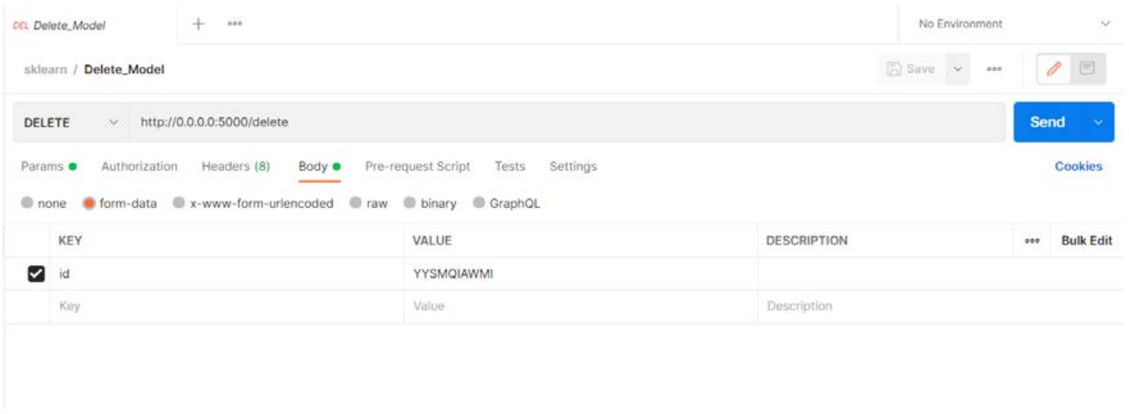
Deleting a model

In order to delete an uploaded model from the API the function `delete` must be called in `DELETE` mode using the following parameters in the Body **form-data** section:

- `id`: this field must contain the id of the model we wish to delete.

The function will return a message confirming the deletion.

Here we have an example using Postman:



DELETE http://0.0.0.0:5000/delete

	KEY	VALUE	DESCRIPTION
✓	id	YYSMQIAWMI	
	Key	Value	Description

Uploading a dataset

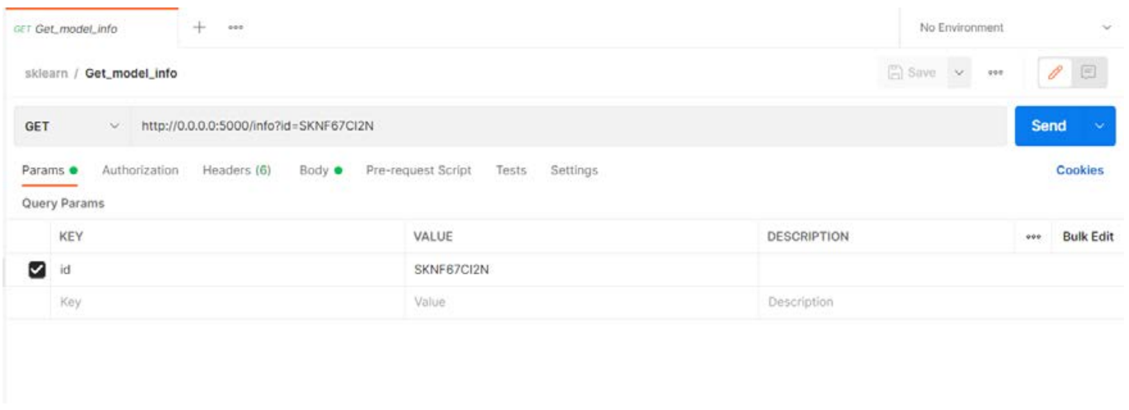
Retrieving a model's parameters

In order to retrieve the parameters provided when uploading a model to the API the function `info` must be called in `GET` mode using the following parameters in the **URL parameter** section:

- `id`: this field must contain the id of the model whose parameters we wish to retrieve.

The function will return a json with the parameters.

Here we have an example using Postman:



GET Get_model_info

sklearn / Get_model_info

GET http://0.0.0.0:5000/info?id=SKNF67CI2N

Query Params

	KEY	VALUE	DESCRIPTION
✓	id	SKNF67CI2N	
	Key	Value	Description

Uploading a dataset

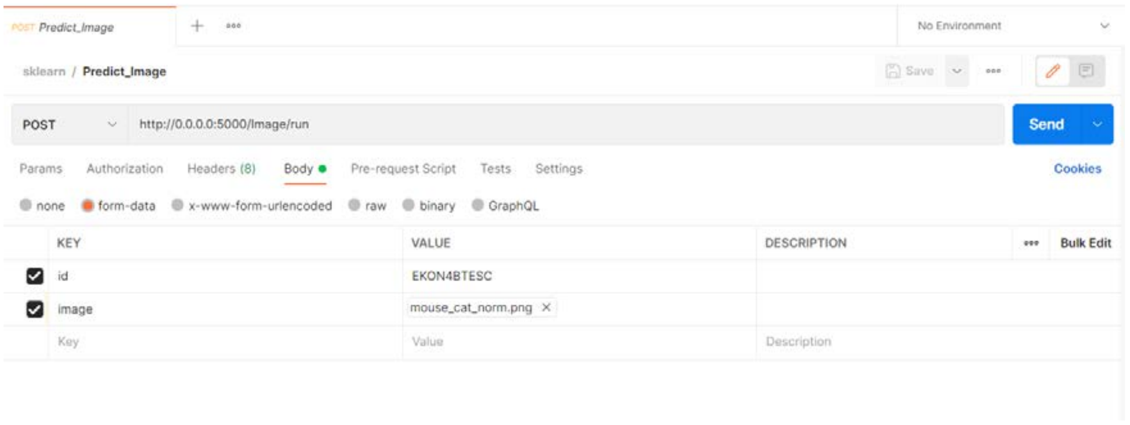
Predicting with an image

In order to make a prediction based on an image using a model uploaded to the API the function `/Image/run` must be called in `POST` mode using the following parameters in the Body **form-data** section:

- `image`: this field must contain a file which corresponds to the image that we want to pass to the model.
- `id`: this field must contain the id of the model that we want to use to make the prediction.

The function will return a message with the model prediction.

Here we have an example using Postman:



POST Predict_Image

sklearn / Predict_Image

POST http://0.0.0.0:5000/Image/run

Body: form-data

	KEY	VALUE	DESCRIPTION
✓	id	EKON4BTESC	
✓	image	mouse_cat_norm.png	
	Key	Value	Description

Uploading a dataset

Predicting with a tabular set

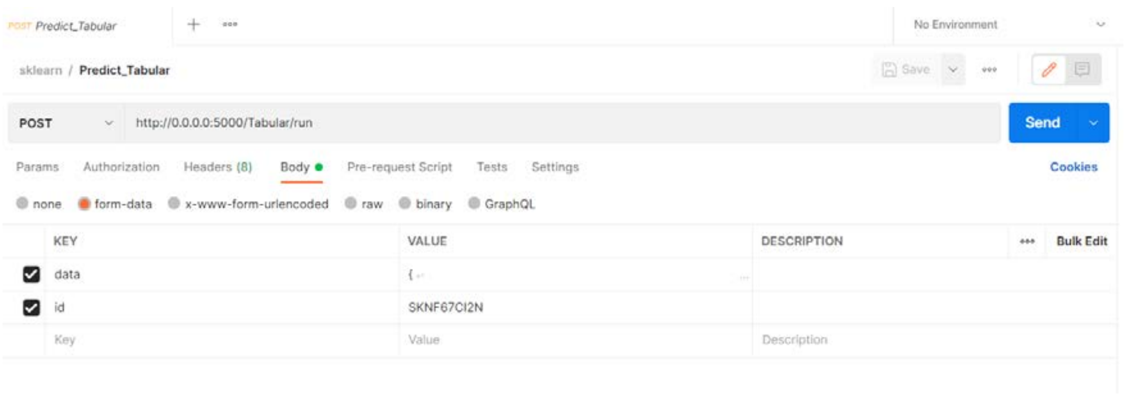
In order to make a prediction based on a dataset using a model uploaded to the API the function `/Tabular/run` must be called in `POST` mode using the following parameters in the Body **form-data** section:

- `data`: this field must contain a text formatted as a json with a field named *“instance”* followed by an array containing the data to be passed to the model.

- `id`: this field must contain the id of the model that we want to use to make the prediction.

The function will return a message with the model prediction.

Here we have an example using Postman:


POST Predict_Tabular

sklearn / Predict_Tabular

POST http://0.0.0.0:5000/Tabular/run

Body: form-data

	KEY	VALUE	DESCRIPTION
✓	data	{	
✓	id	SKNF67CI2N	
	Key	Value	Description

Uploading a dataset

Explainer Library

Using the API with Postman

This quick guide illustrates how to launch the Flask server and make requests to any of the explanation methods in the API using Postman.

Launching with Python

1) Clone the repository.

2) From the root folder, create a virtual environment for the installation of the required libraries with:

```
python -m venv .
```

3) Use pip to install the dependencies from the requirements file.

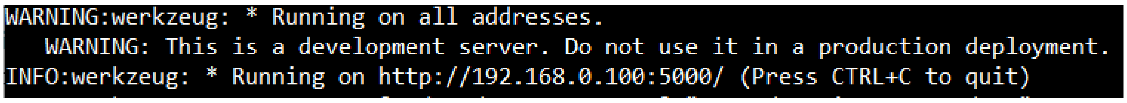
```
pip install -r requirements.txt
```

4) Once all the dependencies have been installed, execute the script to launch the server with:

```
python app.py
```

Making Requests

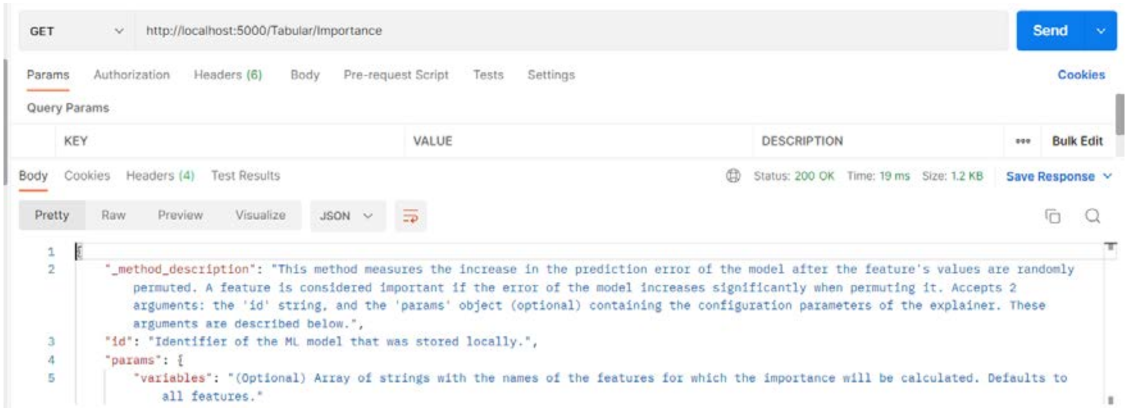
If the server was launched successfully, a similar message to the one in the image should appear, meaning that it is ready to receive requests to the specified address and port.



```
WARNING:werkzeug: * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
INFO:werkzeug: * Running on http://192.168.0.100:5000/ (Press CTRL+C to quit)
```

ServerLaunched

1) To make requests, open Postman and go to *My Workspace > File > New Tab*.
2) To get information about how to use a specific method, we can make a GET request. In the URL bar, specify the address and port of the server, followed by the name of the method, and send the request. The response is displayed in the bottom part of the console. For example, for Tabular/Importance:



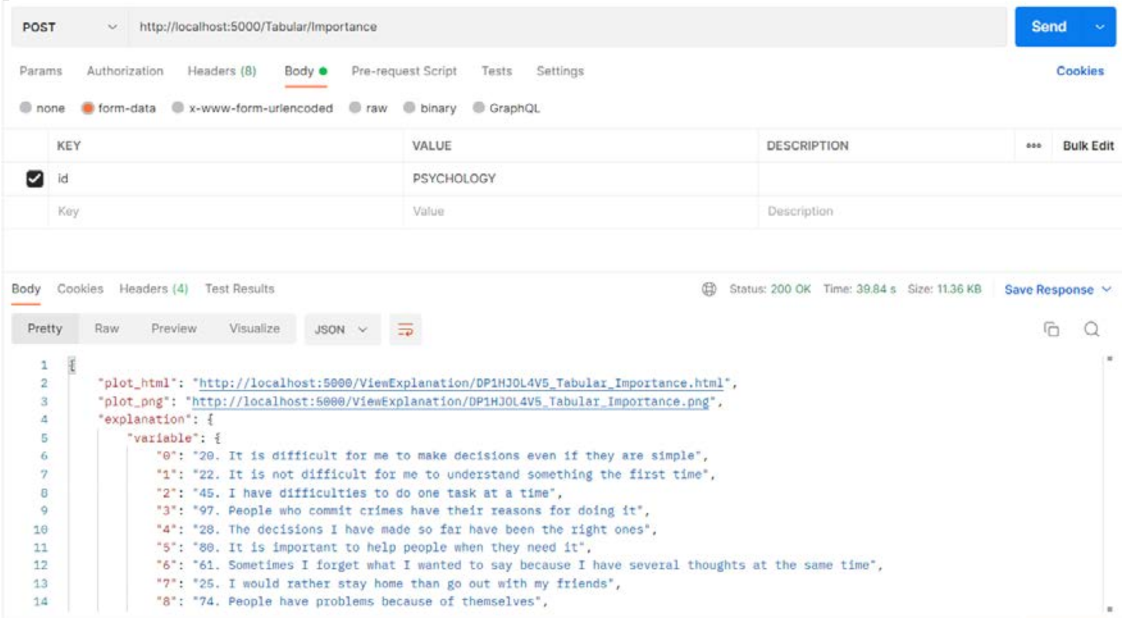
GET http://localhost:5000/Tabular/Importance

Status: 200 OK Time: 19 ms Size: 1.2 KB

```
{
    "_method_description": "This method measures the increase in the prediction error of the model after the feature's values are randomly
        permuted. A feature is considered important if the error of the model increases significantly when permuting it. Accepts 2
        arguments: the 'id' string, and the 'params' object (optional) containing the configuration parameters of the explainer. These
        arguments are described below.",
    "id": "Identifier of the ML model that was stored locally.",
    "params": {
        "variables": "(Optional) Array of strings with the names of the features for which the importance will be calculated. Defaults to
            all features."
```

Screenshot (119)

3) To execute the methods and get actual explanations, we have to make a POST request. To do so, change the request type to POST and go to *Body > form-data*. Here is where we specify the required parameters, such as the *id*, *url*, and the *params* object. These parameters are explained in greater detail below in the section *About the parameters*. In this example, I am using the psychology model available in the Models folder. The only parameter passed in this case was the *id*.



http://localhost:5000/Tabular/Importance

	KEY	VALUE	DESCRIPTION
✓	id	PSYCHOLOGY	
	Key	Value	Description

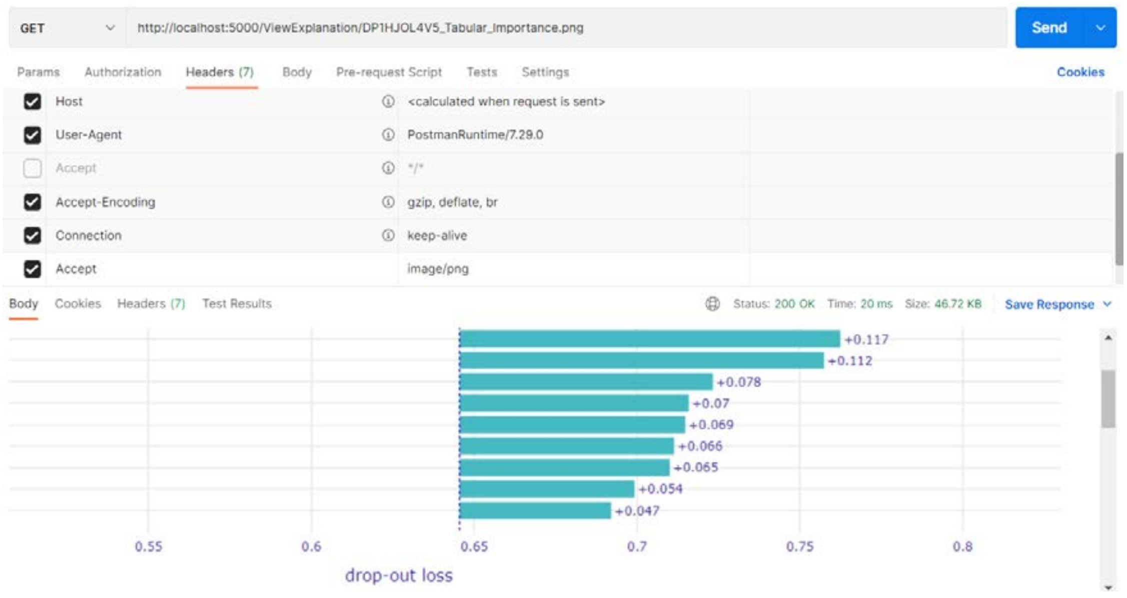
Status: 200 OK Time: 39.84 s Size: 11.36 KB

```
{
    "plot_html": "http://localhost:5000/ViewExplanation/DP1HJOL4V5_Tabular_Importance.html",
    "plot_png": "http://localhost:5000/ViewExplanation/DP1HJOL4V5_Tabular_Importance.png",
    "explanation": {
        "variable": {
            "0": "20. It is difficult for me to make decisions even if they are simple",
            "1": "22. It is not difficult for me to understand something the first time",
            "2": "45. I have difficulties to do one task at a time",
            "3": "97. People who commit crimes have their reasons for doing it",
            "4": "28. The decisions I have made so far have been the right ones",
            "5": "80. It is important to help people when they need it",
            "6": "61. Sometimes I forget what I wanted to say because I have several thoughts at the same time",
            "7": "25. I would rather stay home than go out with my friends",
            "8": "74. People have problems because of themselves",
```

Visualizing Explanations

The responses to the HTTP requests are given in JSON format. However, most of the methods return responses that also contain the URLs to plots or graphs of the explanations in HTML or PNG format. Before accessing the explanations, it is necessary to change the default JSON mime-type.

1) To visualize these explanations, click on the URL in the response. It will open a new request tab with the specified URL.
2) Go to *Headers* and disable the *Accept* attribute.
3) Add a new header with the same name, *Accept*, as a key and specify the value according to the type of file you are trying to access. For .png files, specify *image/png*. For .html files, specify *text/html*. Finally, send the request.


GET http://localhost:5000/ViewExplanation/DP1HJOL4V5_Tabular_Importance.png

✓	Host	<calculated when request is sent>
✓	User-Agent	PostmanRuntime/7.29.0
☐	Accept	*/*
✓	Accept-Encoding	gzip, deflate, br
✓	Connection	keep-alive
✓	Accept	image/png

Status: 200 OK Time: 20 ms Size: 46.72 KB

Screenshot (158)

About the Parameters

The required parameters may be different depending on the explainer, so it is recommended to see the documentation provided by the get method of the explainer being used.

- **id**: the *id* is a 10-character long string composed of letters and/or numbers. It is used to access the server space dedicated to the model to be explained. This space is a folder with the same name as the id located in the *Models* folder. This folder is created by the "Model AI Library" when a user uploads a model file (or an external URL), the training data (if required), and specific information about the model. Note that **if you want to use your own model**, you a folder with the followinf files to the "Models" folder:
  - *Model File*: The trained prediction model given as a compressed file. The extension must match the backend being used i.e. a .pkl file for Scikit-learn (use Joblib library), .pt for PyTorch, or .h5 for TensorFlow models. The name of the files must be the same as the id. For models with different backends, it is possible to upload a .pkl, but it is necessary that the prediction function of the model is called 'predict'.
  - *Data File*: Pandas DataFrame containing the training data given as a .pkl file (use Joblib library). The name of this file must be the id concatenates with the string "_data", i.e: PSYCHOLOGY_data.pkl. The target class must be the last column of the DataFrame. Currently, it is only needed for tabular data models.
  - *Model Info*: JSON file containing the characteristics of the model, also referred to as model attributes. Some attributes ar mandatory, such as the alias of the model, which is the common name that will be assigned to it. Also the backend and the task performed by the model are required in most cases. Other attributes are optional, such as the names of the features for tabular data, the categorical features, the

labels of the output classes, etc. Even though some of these attributes may be optional, they may considerably improve the quality of the explanation, mostly from a visualization point of view. Note that model attribues are *static*, they don't vary from execution to execution. Please refer to the model_info_attributes.txt file to see the currently defined attributes among all the explainers available.

Note: Regardless of the uploaded files, **all the methods require an id to be provided. If you want to test a method with your own model, upload a folder containing the previously described files to the Models folder, assigning an id of your choice**. See the example below for a model with id "PSYCHOLOGY".

- **url**: External URL of the prediction function passed as a string. This parameter provides an alternative when the model owners do not want to upload the model file and the explanation method is able to work with a prediction function instead of a model object. **The URL is ignored if a model file was uploaded to the server**. This related server must be able to handle a POST request receiving a multi-dimensional array of N data points as inputs (instances represented as arrays). It must return an array of N outputs (predictions for each instance). Refer to the *External URLs Examples folder* if you want to quickly create a service using Flask to provide this method. Please see the example in the section below.

- **instance**: This is a mandatory attribute for local methods, as it is the instance that will be explained. It is an array containing the feature values (which must be in the same order that the model expects). For images, it is a matrix representing the pixels. It is also possible for image explainers to pass a file instead of the matrix using the "image" argument.

- **params**: dictionary with the specific execution parameters passed to the explanation method. These parameters are optional and depend on the method being used. The value assigned to this parameters may signficantly change the outcome of an explanation. For example, the "target_class" of a counterfactual is an execution parameter. Refer to the documentation of each method to know the configuration parameters that can be provided.

Getting Explanations Using External URLs Models

In some cases, uploading a model file to the server is not desired by the user or simply not possible. Some explanation methods provide an alternative, as they only need access to the prediction function of the model. The prediction function can be easily wrapped as an HTTP POST method so that the explainers can access the prediction function by making requests to a server administered by the user. However, the implementation of the POST method must attain the expected format:

- **The POST method must receive a parameter named "inputs" and return an array with the predictions**. The format of the "inputs" parameter, as well as the output, must be as follows:

- For Tabular and Text models:
  - For Regression Models:
    - inputs: array of shape *(n, f)* where *n* is the number of instances and *f* is the number of features.
    - output: array of shape *(n,)* where *n* is the number of instances. Contains the predicted value for each instance.
  - For Classification Models:
    - inputs: array of shape *(n, f)* where *n* is the number of instances and *f* is the number of features. Contains the predicted probabilities of each class for each instance.
- For Image models:
  - inputs: Array of shape *(n, h, w)* for black and white images, and shape *(n, h, w, 3)* for RGB images, where *n* is the number of images, *h* the pixel height, and *w* the pixel width.
  - output: array of shape *(n, c)* where *n* is the number of instances and *c* is the number of classes. Contains the predicted probabilities of each class for each image.

Notice that if you are using a model from Tensorflow or Scikit-learn, the *predict* or *predict_proba* function of your model already matches this format. If you have models from different architectures, some additional wrapping code may be necessary to comply with this format.

For illustration purposes, we will implement the POST method with Flask using the psychology model. Example implementations of external URL prediction functions are available in the *External_URLs* folder.

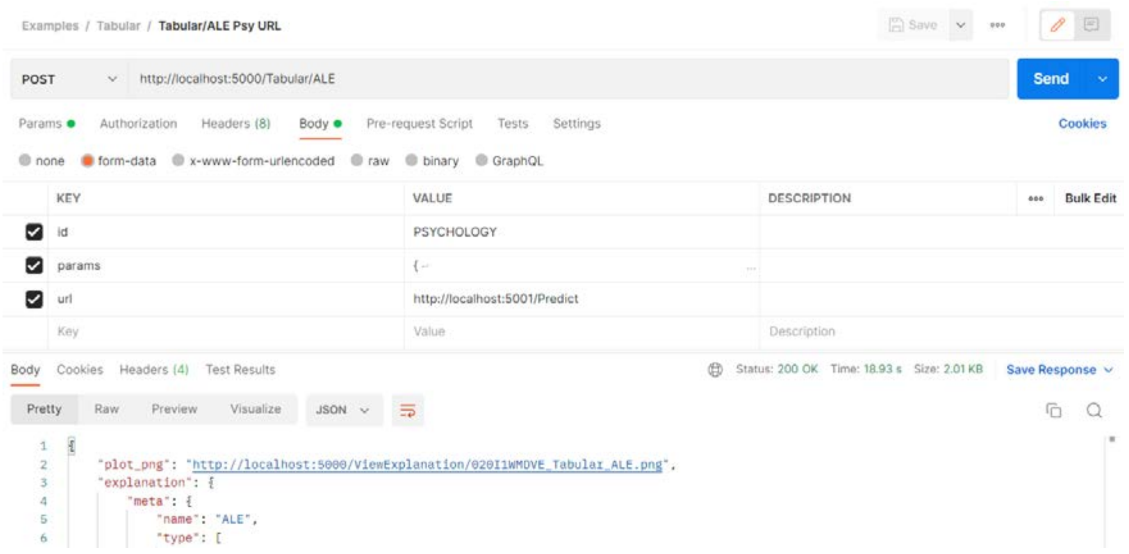
1) If you are testing locally, launch the explainer libraries server as described before.

2) For the server logic, load the previously trained model first. Then define the POST method and add the inputs parameter to the parser. Load the contents of the inputs parameter and pass them to the prediction function of your model. We use the predict_proba function, as the psychology model is a scikit-learn classifier. Finally, specify the path for the method by adding it to the API. **Note**: if you are testing locally, make sure to assign a different port from the explainer libraries server.

```
import sys
from flask import Flask
from flask_restful import Api,Resource,reqparse
import numpy as np
import json
import joblib

cli = sys.modules['flask.cli']
cli.show_server_banner = lambda *x: None
app = Flask(__name__)
api = Api(app)
```

```
#Load the model
model = joblib.load("PSYCHOLOGY.pkl")

class Predict(Resource):

    def post(self):
        #Add the 'inputs' argument
        parser = reqparse.RequestParser()
        parser.add_argument("inputs", required=True)
        args = parser.parse_args()

        #Get the inputs and pass them to the prediction function
        inputs = np.array(json.loads(args.get("inputs")))
        return model.predict_proba(inputs).tolist()

# Add the resource to the API
api.add_resource(Predict, '/Predict')

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5001)
```

3) Run the server and test the POST method by passing the *url* parameter to the explanation method. Remember that the url is ignored if a model file was uploaded to the server, so make sure no model file is present in the corresponding folder.



POST http://localhost:5000/Tabular/ALE

none [✓ form-data] x-www-form-urlencoded raw binary GraphQL

	KEY	VALUE	DESCRIPTION
✓	id	PSYCHOLOGY	
✓	params	{…	
✓	url	http://localhost:5001/Predict	
	Key	Value	Description

Status: 200 OK Time: 18.93 s Size: 2.01 KB

```
{
    "plot_png": "http://localhost:5000/ViewExplanation/020I1WMDVE_Tabular_ALE.png",
    "explanation": {
        "meta": {
            "name": "ALE",
            "type": [
```

Screenshot (166)

How to Collaborate to ExplainerLibraries

1) Fork the repo and clone it to our local machine.

2) Create our own branch.

3) Add the explainer file and make the necessary modifications.

4) Launch the application locally and test the new explainer.

5) Push the changes and create a pull request for review.

Adding New Explainers to the Catalogue

1) To add a new explainer, it is necessary to create a new Resource. First, go to the *resources/explainers* folder and select the folder corresponding to the data type of the explainer you want to add (If your explainer works with a different data type, please add the corresponding folder to the resources folder). For illustration purposes, we will walk through the steps of adding a "new" explainer (LIME tabular).

2) Inside the appropriate folder, ***create a new .py file*** with the name of your explainer. In our case, we create the lime.py file inside *resources/explainers/tabular/* .

3) Create a class for the explainer. This class needs to have ***two different methods: post and get***. You may also need to add an **__init__** method to access the paths of the models and uploads folders. In our example:

```
from flask_restful import Resource

class Lime(Resource):

        def __init__(self,model_folder,upload_folder):
        self.model_folder = model_folder
        self.upload_folder = upload_folder

    def post(self):
        return {}

    def get(self):
        return {}
```

5) In the **post method**, define the mandatory arguments that must be passed for the explainer to get an explanation. The method must receive at least an id to access the folder related to the model. After parsing the arguments, use the function *get_model_files*, passing the id to fetch the model, data, and info files. It is possible that some of these files do not exist, so make the appropriate checks before using them. Generally, the steps involve loading the Dataframe with the training data if it exists, then getting the necessary attributes from the info file, then getting the prediction function if possible, and finally getting the configuration parameters from the *params* object.

```
class Lime(Resource):

def post(self):
        parser = reqparse.RequestParser()
        parser.add_argument('id',required=True)
        parser.add_argument('instance',required=True)
        parser.add_argument('url')
        parser.add_argument('params')
        args = parser.parse_args()

        _id = args.get("id")
        url = args.get("url")
        instance = json.loads(args.get("instance"))
        params=args.get("params")
        params_json={}
        if(params !=None):
            params_json = json.loads(params)

        #Getting model info, data, and file from local repository
        model_file, model_info_file, data_file =
get_model_files(_id,self.model_folder)

        ## loading data
        if data_file!=None:
            dataframe = joblib.load(data_file) ##error handling?
        else:
            raise Exception("The training data file was not
provided.")

        ##getting params from info
        model_info=json.load(model_info_file)
        backend = model_info["backend"]  ##error handling?
        kwargsData = dict(mode="classification", feature_names=None,
categorical_features=None,categorical_names=None, class_names=None)
        if "model_task" in model_info:
            kwargsData["mode"] = model_info["model_task"]
        if "feature_names" in model_info:
            kwargsData["feature_names"] = model_info["feature_names"]
        if "categorical_features" in model_info:
            kwargsData["categorical_features"] =
model_info["categorical_features"]
        if "categorical_names" in model_info:
            kwargsData["categorical_names"] = {int(k):v for k,v in
model_info["categorical_names"].items()}
        if "output_names" in model_info:
            kwargsData["class_names"] = model_info["output_names"]

        ## getting predict function
        predic_func=None
        if model_file!=None:
            if backend=="TF1" or backend=="TF2":
                model=h5py.File(model_file, 'w')
                mlp = tf.keras.models.load_model(model)
                predic_func=mlp
            elif backend=="sklearn":
                mlp = joblib.load(model_file)
                predic_func=mlp.predict_proba
            elif backend=="PYT":
                mlp = torch.load(model_file)
                predic_func=mlp.predict
            else:
                mlp = joblib.load(model_file)
                predic_func=mlp.predict
        elif url!=None:
            def predict(X):
                return np.array(json.loads(requests.post(url,
data=dict(inputs=str(X.tolist()))).text))
            predic_func=predict
        else:
            raise Exception("Either a stored model or a valid URL for
the prediction function must be provided.")



        #getting params from request
        kwargsData2 = dict(labels=(1,), top_labels=None,
num_features=None)
        if "output_classes" in params_json: #labels
            kwargsData2["labels"] =
json.loads(params_json["output_classes"]) if
isinstance(params_json["output_classes"],str) else
params_json["output_classes"]
        if "top_classes" in params_json:
            kwargsData2["top_labels"] =
int(params_json["top_classes"])   #top labels
        if "num_features" in params_json:
            kwargsData2["num_features"] =
int(params_json["num_features"])

    ...
```

6) Add the actual code for the generation of the explanation to the post method. This depends entirely on the explanation method being used. Once the explanation has been created, convert it to a JSON format if necessary. If the explanation is returned as an html or png file, use the save_file_info function to get the upload folder path, the name that will be given to the file, and the url (getcall) that will be used to access the file. Save the file using this data and append the URL to the returned JSON. **Note:** the URL to access the file returned by save_file_info does not include the extension of the file, so it is necessary to append it at the end as it is shown in the example.

```
class Lime(Resource):

def post(self):

    ...

    explainer =
lime.lime_tabular.LimeTabularExplainer(dataframe.drop(dataframe.column
s[len(dataframe.columns)-1], axis=1, inplace=False).to_numpy(),
                                                                **{k: v
for k, v in kwargsData.items() if v is not None})
        explanation = explainer.explain_instance(np.array(instance,
dtype='f'), predic_func, **{k: v for k, v in kwargsData2.items() if v
is not None})

        ## Formatting json explanation
        ret = explanation.as_map()
        ret = {str(k):[(int(i),float(j)) for (i,j) in v] for k,v in
ret.items()}
        if kwargsData["class_names"]!=None:
            ret = {kwargsData["class_names"][int(k)]:v for k,v in
ret.items()}
        if kwargsData["feature_names"]!=None:
            ret = {k:[(kwargsData["feature_names"][i],j) for (i,j) in
v] for k,v in ret.items()}
        ret=json.loads(json.dumps(ret))

        ## saving to Uploads
        upload_folder, filename, getcall =
save_file_info(request.path)
        hti = Html2Image()
        hti.output_path= upload_folder
        hti.screenshot(html_str=explanation.as_html(),
save_as=filename+".png")
        explanation.save_to_file(upload_folder+filename+".html")


response={"plot_html":getcall+".html","plot_png":getcall+".png","expla
nation":ret}
        return response
```

7) For the get method, return a dictionary that serves as documentation for the explainer that is being implemented. In our implementations, we include a brief description of the explainer method and the parameters to the request, as well as the configuration parameters that should be passed in the *params* dictionary. If necessary, we also include an example of the *params* object. For example, for the Tabular/LIME implementation:

```
    def get(self):
                return {
        "_method_description": "LIME perturbs the input data samples
in order to train a simple model that approximates the prediction for
the given instance and similar ones. "
                           "The explanation contains the weight of
each attribute to the prediction value. This method accepts 4
arguments: "
                           "the 'id', the 'instance', the
'url'(optional),  and the 'params' dictionary (optiohnal) with the
configuration parameters of the method. "
                           "These arguments are described below.",
        "id": "Identifier of the ML model that was stored locally.",
        "instance": "Array representing a row with the feature values
of an instance not including the target class.",
        "url": "External URL of the prediction function. Ignored if a
model file was uploaded to the server. "
                "This url must be able to handle a POST request
receiving a (multi-dimensional) array of N data points as inputs
(instances represented as arrays). It must return a array of N outputs
(predictions for each instance).",
        "params": {
                "output_classes" : "(Optional) Array of ints
representing the classes to be explained.",
                "top_classes": "(Optional) Int representing the number
of classes with the highest prediction probability to be explained.
Overrides 'output_classes' if provided.",
                "num_features": "(Optional) Int representing the
maximum number of features to be included in the explanation."
                }
```

8) Lastly, add the class as a resource and specify its route in the *app.py* and in the *explainerslist.py* files. **Also update the model_info_attributes.txt** file if you are using a new model attribute that was not included before. In our example:

```
from resources.explainers.tabular.lime import Lime
api.add_resource(Lime, '/Tabular/LIME')
```

Software Repositories

Github Repositories

https://github.com/isee4xai/iSeeBackend/tree/dev/AI%20Model%20lib

https://github.com/isee4xai/ExplainerLibraries

Reproducible Capsule

https://explainers-dev.isee4xai.com

https://models-tf-dev.isee4xai.com

https://models-sk-dev.isee4xai.com

Source code repository branching conventions

Git and GitHub will be used to control the development of the different modules of the iSee platform. In order to ensure a high quality of the development and the management of the different code source repositories, the following rules will be applied:

- The **‘main’ branch** must only contain stable code and the different releases of the iSee modules. No direct pushes are allowed to this branch; all commits must be pushed through Pull Requests and will be merged into the ‘main’ branch after a code review.

- The **‘dev’ branch** is the main development branch that contains the code that will be deployed on the development test platform. An automated continuous integration / deployment (CI/CD) workflow will be created to a) build the virtual container and b) deploy it on the test platform when a new code is pushed to this branch.
  As for the ‘main’ branch, no direct pushes are allowed to this branch; all commits should be pushed through Pull Requests.

- **Other (temporary) branches**: these are the branches that can be created and deleted when needed, for bug fixes, experimental testing and new features. These branches must have a meaningful naming and be deleted when no longer used.

- iSee developers should work **on their own branches** and create a Pull Request when code is ready to be deployed.

Naming convention of the containers

All iSee modules are delivered as Docker containers (or similar technology) to promote the scalability and stability of the platform. The naming/tagging convention of these containers are as follow:

isee4xai/*<module>:<tag>*

where

- *<module>* is the module name aligned with code source repository name.
- *<tag>* can be either:
  - ‘dev’: the current code at development branch
  - ‘X.Y’: (eg: 1.2) the major and minor versions of official stable releases
  - ‘latest’: the latest official stable release.