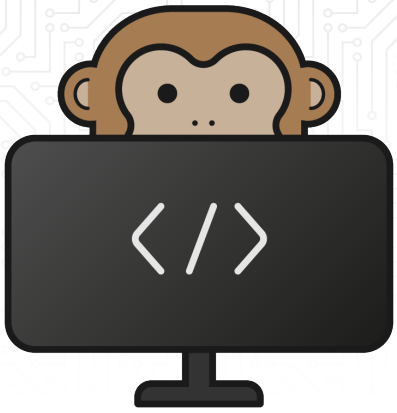

ProApes

proapes11@gmail.com

Developer Manual

Version	1.0.0-1.10
Approval date	2020-05-15
Project Manager	Federico Carboni
Editors	Federico Carboni Francesco Bari Alessandro Discalzi
Verifiers	Valentina Signor Fiammetta Cannavò
State	Approved
Distribution List	*ProApes* *Prof. Tullio Vardanega* *Prof. Riccardo Cardin*
Usage	External

Summary

This document is intended to present the system's technologies and architecture to developers interested in extending and maintaining *Predire in Grafana* software.

History

Version	Date	Changes	Author	Role
v1.0.0-1.10	2020-05-15	*Approval of the document for RA*	Federico Carboni	*Project Manager*
v0.2.0-1.10	2020-05-13	*Review and corrections of coherence and cohesion (*Verifier*: Valentina Signor)*	Federico Carboni	*Programmer*
v0.1.3-1.10	2020-05-11	*Drawing up of section §6 and appendix §A (*Verifier*: Fiammetta Cannavò)*	Alessandro Discalzi	*Programmer*
v0.1.2-1.10	2020-05-10	*Drawing up of sections §4 and §5* Verifier*: Fiammetta Cannavò)*	Federico Carboni	*Programmer*
v0.1.1-1.9	2020-05-09	*Drawing up of §3 section (*Verifier*: Fiammetta Cannavò)*	Federico Carboni	*Programmer*
v0.1.0-1.9	2020-05-08	*Review and corrections of coherence and cohesion (*Verifier*: Valentina Signor)*	Alessandro Discalzi	*Programmer*
v0.0.3-1.9	2020-05-07	*Drawing up of §2 section (*Verifier*: Fiammetta Cannavò)*	Francesco Bari	*Programmer*
v0.0.2-1.9	2020-05-06	*Drawing up of §1 section (*Verifier*: Valentina Signor)*	Federico Carboni	*Programmer*
v0.0.1-1.9	2020-05-05	*Creation of LaTeX document*	Giacomo Piran	*Project Manager*

Index

List of figures

1 Introduction

1.1 Aim of the document

The aim of the *Developer Manual* is to present the architecture of *Predire in Grafana*$_{\mathbf{G}}$ product and the source code organization, and to provide useful information for the maintenance and extension of the project. This document also aims to illustrate the installation and local development procedures, cite the frameworks and third parties libraries involved and to present the project structure at progressive levels of detail, thanks to the use of package, class and sequence diagrams.

1.2 Aim of the product

The terms of contract$_{\mathbf{G}}$ **C4** - *Predire in Grafana* arise from the need to constantly monitor resources, applications and information, after applying a DevOps$_{\mathbf{G}}$ approach to the software lifecycle. *ProApes* group propose is to develop a *Grafana*$_{\mathbf{G}}$ *plug-in*$_{\mathbf{G}}$ for *Zucchetti S.p.A.* to support the *Grafana* monitoring tool that applies *SVM*$_{\mathbf{G}}$ techniques and *Linear, Exponential or Logarithmic Regression*$_{\mathbf{G}}$ on the flow of data received to recevice alarms or signals between operators of the Cloud service and the software production line.

1.3 Prerequisites for a good comprehension

For a better comprehension of the document content, especially for the diagrams presented in the following sections, the reader should have at least a general smattering of *UML2.0*$_{\mathbf{G}}$ language, of Machine Learning algorithms like *Linear*, *Exponential* and *Logarithmic Regression* and *Support Vector Machine*, of *Grafana* platform and *React*$_{\mathbf{G}}$ framework.

1.4 Glossary

In this document, there could be terms with ambiguous or contradictory meanings. To avoid incomprehension, at the end of the document, in appendix §A, you can find a glossary with these terms and their explanation. In the following sections, to promote clarity without being redundant, glossary words will be marked with a subscripted "G" at their first occurrence in every section.

1.5 Versions notes

ProApes group guarantees the proper operation of the *Predire in Grafana* product only if the exact versions of the libraries, frameworks and *Grafana* software that are listed in the rest of the document are used. Although subsequent versions of the tools used are likely to maintain backward compatibility, this cannot be guaranteed. The group assumes no responsibility in case the product is whole or in part affected by problems attributable to the use of third-party software with different versions other than those specified.

1.6 References

1.6.1 Normative references

- **Terms of contract C4**:
  https://www.math.unipd.it/~tullio/IS-1/2019/Progetto/C4.pdf

1.6.2 Informative references

- **Model-View Patterns - Didactic material of the Software Engineering course**:
  https://www.math.unipd.it/~rcardin/sweb/2020/L02.pdf
  - MV patterns structure, slides 6 - 14;
  - MVVM, slides 29 - 50.
- **SOLID Principles - Didactic material of the Software Engineering course**:
  https://www.math.unipd.it/~rcardin/sweb/2020/L04.pdf
  - Single Responsibility Principle, slides 5 - 10;
  - Open-Close Principle, slides 11 - 17;
  - Liskov Substitution Principle, slides 18 - 23;
  - Interface Segregation Principle, slides 26 - 30;
  - Dependency Inversion Principle, slides 32 - 36.
- **Class diagrams - Didactic material of the Software Engineering course**:
  https://www.math.unipd.it/~tullio/IS-1/2019/Dispense/E01b.pdf
  - Properties and operations, slides 11 - 34;
  - Features, slides 35 - 38.
- **Package diagrams - Didactic material of the Software Engineering course**:
  https://www.math.unipd.it/~tullio/IS-1/2019/Dispense/E01c.pdf
  - Packages and dependencies, slides 12 - 15.
- **Sequence diagrams - Didactic material of the Software Engineering course**:
  https://www.math.unipd.it/~tullio/IS-1/2019/Dispense/E02a.pdf
  - Participants and signals, slides 8 - 16;
  - Modeling, slides 18 -21.
- **Behavioral Design Patterns - Didactic material of the Software Engineering course**:
  https://www.math.unipd.it/~tullio/IS-1/2019/Dispense/E08.pdf
  - Strategy Pattern, slides 32 - 40.
- ***Grafana* documentation**:
  - Indications for plug-in development
    https://grafana.com/docs/grafana/latest/plugins/developing/development/;
  - *Grafana* plug-in code styleguide
    https://grafana.com/docs/grafana/latest/plugins/developing/code-styleguide/;
- **Third part library for *RL* algorithm implementation**:
  https://github.com/Tom-Alexander/regression-js
- **Third part library for *SVM* algorithm implementation**:
  https://github.com/karpathy/svmjs
- ***InfluxDB*$_G$ *APIs*$_G$**:
  https://docs.influxdata.com/influxdb/v1.7/tools/api/

1.6.3 Legal references

- ***MIT*$_{\mathbf{G}}$ license**:
  https://opensource.org/licenses/MIT

2 Technologies and third-party libraries

Technologies and third-party libraries used for the *Predire in Grafana* project are briefly described below.

2.1 Technologies

2.1.1 *Grafana*

Grafana is a multi-platform *open source*$_{\mathbf{G}}$ analytics and interactive visualization software. It provides charts, graphs and alerts for web interface when connected to supported data sources and allows to be expanded through plug-ins. Grafana is a popular data monitoring software and is often used in combination with *time series databases*$_{\mathbf{G}}$ such as *Prometheus*, *Graphite* and *InfluxDB*. In the *Predire in Grafana* project this software plays the role of host for the plug-in, which will extend its features with those provided by the plug-in itself.

- **Version**: 6.7.x;
- **Download link**: https://grafana.com/grafana/download

2.1.2 *InfluxDB*

InfluxDB is an *open-source time series database* developed by *InfluxData*. It ensures high performance and is optimized for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics, sensors data, and real-time analytics. It is used in the *Predire in Grafana* project as the main *datasource* for reading and writing the monitored data.

- **Version**: 1.7.x;
- **Download link**: https://portal.influxdata.com/downloads/

2.1.3 *Telegraf*

Telegraf is a plugin-driven *server agent*$_{\mathbf{G}}$ for collecting and sending metrics and events from databases, systems and IoT sensors. It's connected to *InfluxDB* for the collection of hardware metrics (CPU, RAM, Disk I/O...) to be used as external datasource for *Grafana* during the development of the plug-in.

- **Version**: 1.13.x;
- **Download link**: https://portal.influxdata.com/downloads/

2.1.4 *React*

React is a *JavaScript* library for building user interfaces. It can be used as a base in the development of single-page or mobile applications. However, *React* is only concerned with rendering data to the DOM$_{\mathbf{G}}$, and so creating *React* applications usually requires the use of additional libraries for *state management* and *routing*. As part of the *Predire in Grafana* project, *React* is used to create the user interface of both the internal module (prediction) and the external module (training).

- **Version**: 16.13.x;
- **NPM *Repository*$_{G}$ link**: https://www.npmjs.com/package/react

2.1.5 *TypeScript*

TypeScript$_{\mathbf{G}}$ is an *open-source* programming language. It's a strict syntactical superset of *JavaScript*$_{\mathbf{G}}$ based on *ECMAScript 6*. The language extends the *JavaScript* syntax so that any program written in *JavaScript* is also able to work with *TypeScript* without any modifications. It's designed for the development of large applications and is intended to be compiled in *JavaScript* to be interpreted by any *web browser* or *application*. *TypeScript* was chosen as the primary language for coding the *Predire in Grafana* plug-in.

- **Version**: 3.7.5 or higher;

2.1.6 *JSON*

JSON is the text serialization format required for exporting predictor files from the external module to the internal one, in order to monitor data. It is an extremely popular format for *client-server*$_{\mathbf{G}}$ applications data exchange. It is based on objects, i.e. pairs of key/value, and supports booleans, strings, numbers and list types. It's simple, readable and it doesn't require any particular compile processing to be modified.

2.1.7 *CSV*

The *comma-separated values (CSV)*$_{\mathbf{G}}$ is a file format based on text files and used for importing and exporting (spreadsheets' or databases') data tables. There is no formal standard that defines it, but only some more or less consolidated practices. It's required to provide training data to the external module to produce the predictor file.

2.1.8 *HTML5*

HTML5 is a *markup* language for structuring and presenting the content of web pages. It is used in *Predire in Grafana*, with *React*, to define the structural elements of the plug-in.

2.1.9 *CSS3*

CSS3 is a style sheet language used for describing the presentation of a document written in *HTML5*, *XHTML* and *XML* documents, such as websites and related web pages. The rules for composing *CSS3* are contained in a set of directives. The use of *CSS3* allows the separation of the *HTML* pages' content from their layout and allows a clearer and easier programming way, both for the authors of the pages and for the users. It guarantees the reuse of code and its more easy maintenance. Being the plug-in, *Predire in Grafana*, composed of *HTML* elements, *CSS3* is used for their stylization.

2.1.10 *NodeJS*

Node.js is a *cross-platform*, *event-driven*, open-source *JavaScript* runtime for executing *JavaScript* code. Many of its basic modules are written in *JavaScript*. It's been an essential part of the coding phase of the *Predire il Grafana* plug-in due to its primary functionality.

- **Version**: 13.7.x;
- **Download link**: `https://nodejs.org/it/download/`

2.1.11 *Yarn*

Yarn is a package manager for *JavaScript* language. It was released by *Facebook* with the aim of solving some problems and defects compared to *NPM*. *Yarn* is more performing, in terms of speed, and implements a cache system to install previously downloaded *packages* even without internet connection. It's been used by the group to perform code building operations.

- **Version**: 1.22.x;
- **Download link**: https://classic.yarnpkg.com/en/docs/install

2.1.12 *SonarCloud*

SonarCloud is a tool for the *Continuous Code Quality*$_{\mathbf{G}}$, based on *SonarQube*. *SonarQube* is an *open-source* platform, widely used to periodically inspect the quality of source code, vulnerabilities and code smells. It supports more than twenty different languages. It can be linked to a public *repository*$_{\mathbf{G}}$ for maximizing the *throughput*$_{\mathbf{G}}$ and improving product releases.

- **SonarCloud link**: https://sonarcloud.io/

2.2 Third-party libraries

2.2.1 *MobX*

MobX is a library created for *React* that allows the management of the *state* of the components in a simple and scalable way. It allows the implementation of the Observer design pattern which is not natively supported in *React*.

- **Version**: 6.2.2;
- **NPM *Repository* link**: https://www.npmjs.com/package/mobx-react

2.2.2 *ESLint*

ESLint is a static code analysis tool for identifying problematic patterns found in *JavaScript* code. It is used to identify code errors without having to build it, and also determines the quality of the written code. In particular, what is reported are syntax errors, imports that are never used within the project, declared but unused variables and much more. It also allows configuration by the developer with customized analysis rules.

- **Version**: 7.x.x;
- **NPM *Repository* link**: https://www.npmjs.com/package/eslint

2.2.3 *Jest*

Open-source testing framework developed by *Facebook* and used to test code written in *JavaScript*. Initially used for testing *React* code, now it's increasingly becoming the reference for tests for *Babel*, *Angular*, *NodeJS* and *TypeScript* projects.

- **Version**: 25.1.0 or higher;
- **NPM *Repository* link**: https://www.npmjs.com/package/jest

2.2.4 *Regression-js*

Regression-js is the *JavaScript* library used to implement the *Linear Regression*, *Exponential Regression* and *Logarithmic Regression* algorithms in the training module. It allows the training and produces the predictor when given a *CSV* file containing the data for the training.

- **GitHub *Repository* link**: https://github.com/Tom-Alexander/regression-js

2.2.5 *Svmjs*

Svmjs is the *JavaScript* library, written by Andrej Karpathy, used to implement the *Support Vector Machine* algorithm in the training module. It allows the training feature and produces the predictor when given a *CSV* file containing the data for the training.

- **GitHub *Repository* link**: `https://github.com/karpathy/svmjs`

3 Setup

Predire in Grafana product consists of two separate modules. The first one is external and implements the training functions; the second one is internal to the *Grafana* system and provides the monitoring functions. Both work on browsers, so the operating systems that will support the plug-ins will not be listed. $UNIX_{\mathbf{G}}$ environment will be used for demonstrations as the *OS* used for development.

3.1 System minimum requirements

The requirements necessary for the correct use of the *Predire in Grafana* product are listed below.

3.1.1 Prerequisites

- *Grafana* v6.7.x;
- *NodeJS* v13.7.x;
- *Yarn* v1.22.x.

3.1.2 Hardware Requirements

- 2GB RAM;
- CPU dual-core.

For more information you can read the documentation provided by *Grafana*:

https://grafana.com/docs/grafana/latest/installation/requirements/

3.1.3 Browser

- *Google Chrome* v58 or higher;
- *Microsoft Edge* v14 or higher;
- *Mozilla Firefox* v54 or higher;
- *Apple Safari* v10 or higher.

3.2 Installation

This section shows all the steps necessary to install the external training module and the Prediction module, inside the *Grafana* system.

3.2.1 Training Module installation

To install and start the module follow these steps:

- download and extract the Training Module folder (.zip format) from the following repo:

  https://github.com/Kero2375/proapes-predire-in-grafana;

- move the content into any *folder* you want;
- open (*Windows*) command prompt or the terminal (*Linux/Mac*);
- use the following command

```
cd /path
```

to access the folder where the content was extracted;

- you will need to install all dependencies first using the command

```
yarn install
```

followed by the command

```
yarn start
```

and open in the browser the path visibile on the terminal (i.e. `http://localhost:3000`).

3.2.2 Prediction Module installation

To install and start the module follow these steps:

- download the folder containing Prediction Module files from the *GitHub repository*

https://github.com/Kero2375/proapes-predire-in-grafana

If you are using the command prompt or terminal, download files from $Git_{\mathbf{G}}$ using the command

```
git clone https://github.com/Kero2375/proapes-predire-in-grafana.git
```

- extract files into `../grafana/plugins` located inside *Grafana* root;
- open the terminal, go to the plug-in folder and launch the command

```
yarn install
```

Yarn will download all the dependencies necessary to build the plug-in correctly.

To make the plug-in ready for use after the installation process, run the command in the terminal within the folder where the plug-in has been installed:

```
yarn dev
```

or

```
yarn watch
```

Then you'll need to start *Grafana* server. The launcher is located into `/bin` within *Grafana* root. You can also start the server in terminal with the following command

```
sudo systemctl start grafana-server
```

Now open the *browser* and connect to the address of the server where *Grafana* is installed followed by any port number used.

Note: if *Grafana* server is installed on the same system where you want to access and the used port it's the default one, you can use the following link:

```
localhost:3000
```

The plug-in will be available in the sidebar of the *Grafana* welcome page as soon as it is enabled on the plug-in search page.

3.3 Workspace

The installation procedures of the other tools necessary for the development of the plug-in are briefly explained below.

3.3.1 *Grafana*

Grafana is the platform on which the *Predire in Grafana* plug-in works. This is an essential technology for development but not created by the group, therefore the instructions for its setup are sent back to the official documentation:

https://grafana.com/docs/grafana/latest/installation/

3.3.2 *InfluxDB*

InfluxDB is the data source used for the development of the plug-in which interfaces with *Grafana*. This is also a fundamental technology for product development but it is not created by the group so the instructions for its setup are sent back to the official documentation:

https://docs.influxdata.com/influxdb/v1.7/introduction/installation/

You may need to edit the *InfluxDB* configuration file located in the directory

`/etc/influxdb/influxdb.conf`

3.3.3 *NodeJS*

NodeJS is the event-oriented *JavaScript* runtime required to compile the code. This is an essential support technology for the development of *Grafana* plug-ins which are written in *JavaScript* or *TypeScript* languages. The instructions for its setup are sent to the official documentation:

https://nodejs.org/it/download/

3.3.4 *React*

React is a fundamental requirement since it is used by the application that the plug-in will extend. No need to make any explicit calls or use *React* graphic libraries but only use the variable

`props`

to connect the front-end to the *JavaScript/TypeScript* controller.

3.3.5 Needed libraries

Regression-js and *Svmjs* are the libraries needed to implement the algorithms of *Linear Regression*, *Exponential Regression*, *Logarithmic Regression* and *Support Vector Machine* into the training module. They have been provided by the proposing company and can be found at the following *GitHub repositories* with the instructions for their import into the project:

- ***Regression-js***: https://github.com/Tom-Alexander/regression-js.git
- ***Svmjs***: https://github.com/karpathy/svmjs.git

4 Test

This chapter is intended to guide developers in how to control how the code and its syntax work. The tools used in the development of the *Predire in Grafana* plug-in to carry out static and dynamic analysis tests are shown below. Since the product is divided into two modules, it is useful to specify that all indications on the use of the following *testing* tools will be the same both for the external training module and for the internal Prediction module.

4.1 *Jest*

Jest testing framework it's used to perform unit tests. The framework configuration is located in the project directory into the file

```
jest.config.js
```

To run the tests, simply run the command

```
yarn test
```

Yarn will automatically perform all the tests specified in the configuration file.

4.2 *ESLint*

Static code analysis tests (i.e. checks on the quality of the written code) are carried out by the *ESLint* tool. To test the code, you need to run the command

```
eslint
```

followed by the path that leads to the directory where configuration file is contained. For example

```
eslint ../grafana/plugins/react-app/config_file.js
```

In case of errors, the system will mark the type and location. Otherwise, nothing will be notified to the user.

4.3 *SonarCloud*

To perform the *code coverage*$_{\mathbf{G}}$ activity it is used *SonarCloud* tool. *SonarCloud* is a *SonarQube*-based *cloud* platform for continuous code quality control. It performs automatic code reviews by doing static analysis and detection of bugs, code smells and security vulnerabilities. There are no commands to use: once configured on the repository it will perform its activity every time the tests are run.

4.4 *GitHub Actions*

The *Continuous Integration* service that has been used is *GitHub Actions*, provided by *GitHub*$_{\mathbf{G}}$. This allows the creation of customized workflows, or automatic processes created from your needs. This aims to automate the software development lifecycle thanks to a wide range of tools and services.

5 Product architecture

The general architecture of the *Predire in Grafana* project is modeled in two distinct modules, both adhering to the architectural *Model-View-ViewModel (MVVM)* design pattern.
The first module, external to the $Grafana_{\mathbf{G}}$ system and called **Training Module**, will train on a dataset imported by the user, producing a *JSON* file containing the definition of the predictor.
The second module, internal to *Grafana* and called **Prediction Module**, will associate the predictor produced by the external module to the data flow monitored in *Grafana*. Thanks to its latest updates, it will be also possible to carry out training, natively provided in the Training Module, also within the Prediction Module itself.
The main reasons for choosing the *MVVM* architectural pattern are the following:

- both modules are written using the *React* framework and it was considered the most integrable pattern for this framework;
- strong reuse of components (model and view) in other contexts, without the need to modify them;
- strong decoupling between *business logic* and *presentation logic*.

5.1 Training Module Architecture

5.1.1 Architectural pattern: MVVM

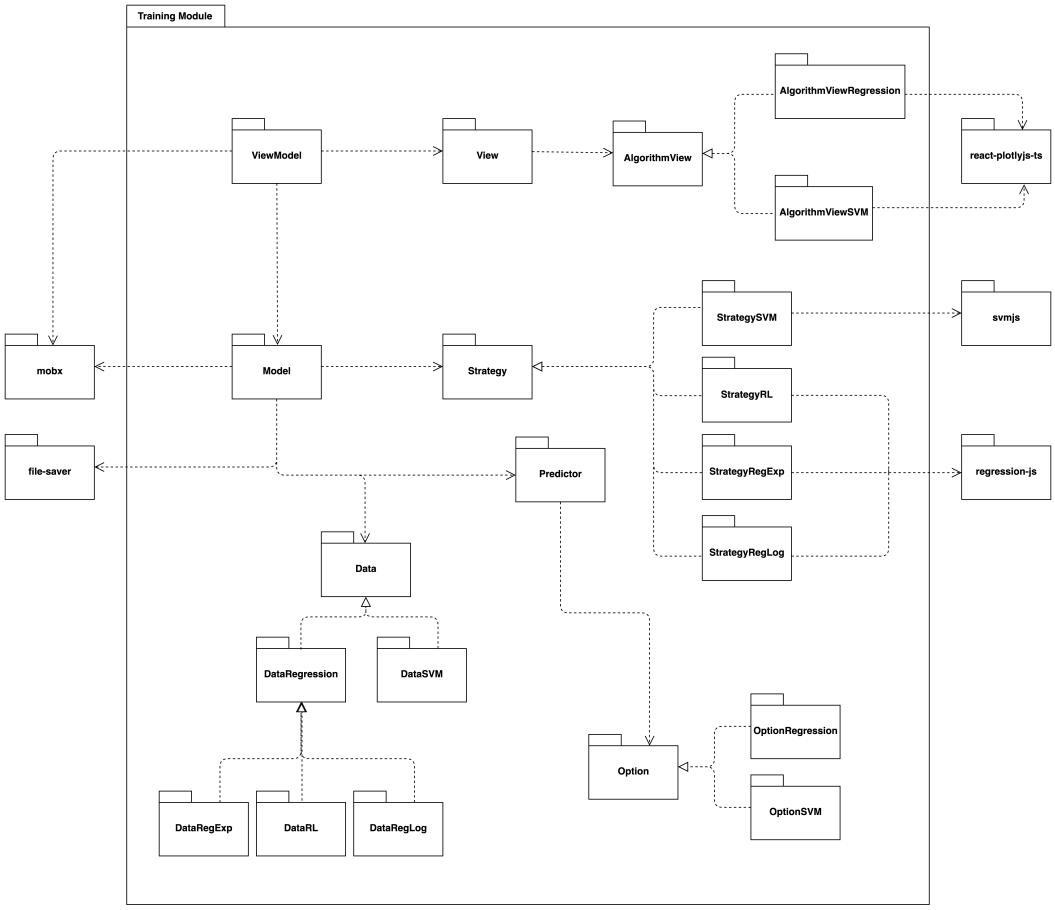

Figura 1: Training Module package diagram

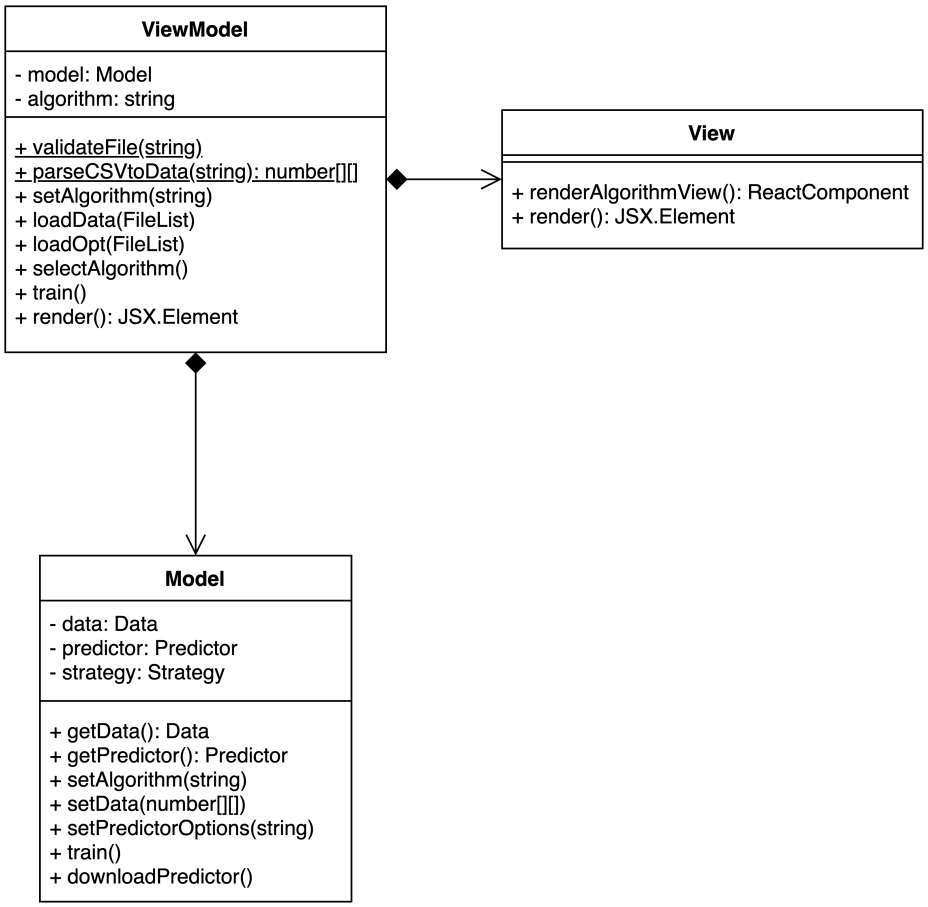

Figura 2: Implementation of *MVVM* pattern into the *Training Module*

As previously specified, the *MVVM* architectural pattern was used for this module. The passage of data from the *Model* to the *View* takes place through the modification of a variable `props` injected by the *Controller* (*ModelView*). The *Controller* passes through these `props` the right methods to call when the user interacts with the view. This use of `props` favors the separation between *presentation logic* and *business logic*. In addition, the *ViewModel* has also a reference to the *Model*, through a `Model` data field, for communicating with the model by calling its public methods.

5.1.2 *MobX*

We used the *MobX* library to update the data relating to the *Model* and to notify the *Controller* of the change of this data. It allowed the implementation of the *Observer* mechanism, not natively supported in *React*. More in detail, it has been used to mark the Model data fields as *Observable* and the *ViewModel* as *Observer*. In this way, whenever the *Model*'s state is updated by some methods, its modification is instantly received by the *ViewModel*, which can update the View.

5.1.3 Detailed architecture: Strategy

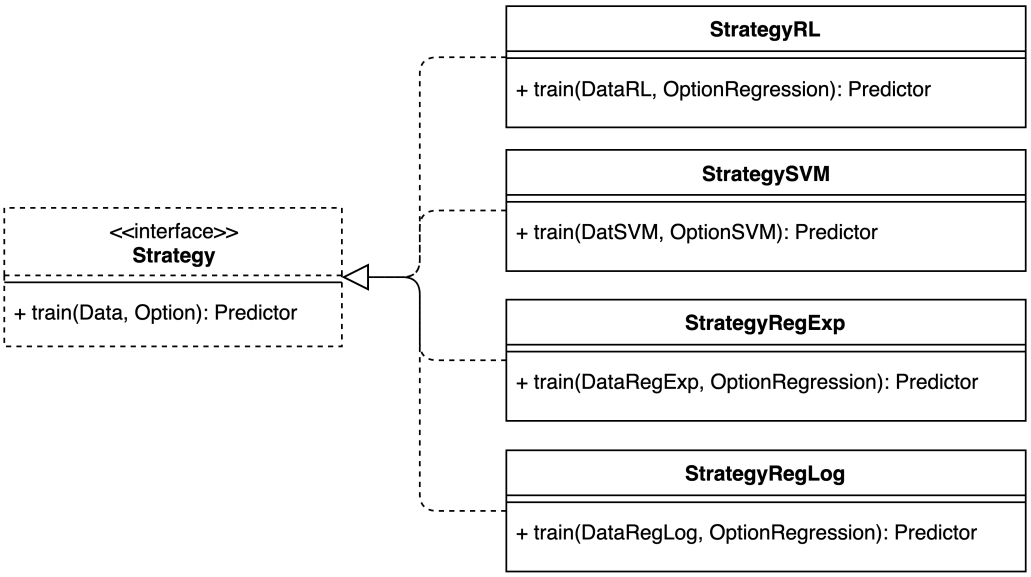

Figura 3: Strategy pattern for choosing the algorithm

The Strategy pattern consists of a fundamental `train()` method for performing the training. It is implemented in the most appropriate way according to the needs of each *Machine Learning* algorithm. As expected, the Strategy is instantiated only once the user has chosen the algorithm, dynamically and through an associative array. This allows to associate the string selected by the user with the most suitable concrete strategy for the choice.

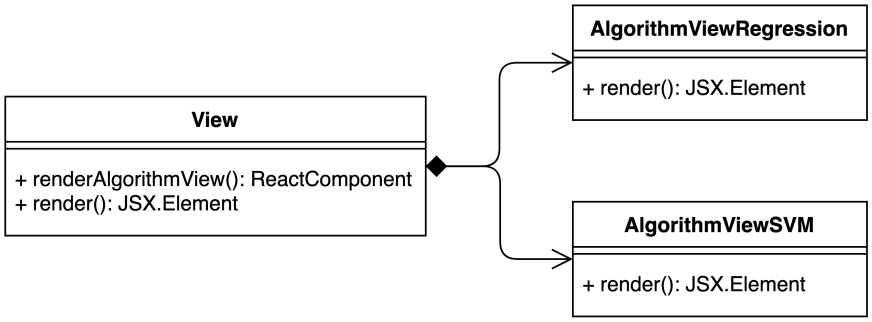

Figura 4: Views according to the chosen algorithm.

In this case, thanks to the Strategy pattern it is possible to choose which specialized view to render. This happens thanks to an associative array which allows the association of the string selected by the user with the most suitable concrete strategy for the choice. More in detail, it happens thanks to `renderAlgorithmView()`: it initially does not display anything, but once it is selected, the algorithm renders the view in a specific way. The `render()` method, on the other hand, renders all other *HTML* elements of the view that do not depend on the choice of the algorithm.

5.1.4 `train()`

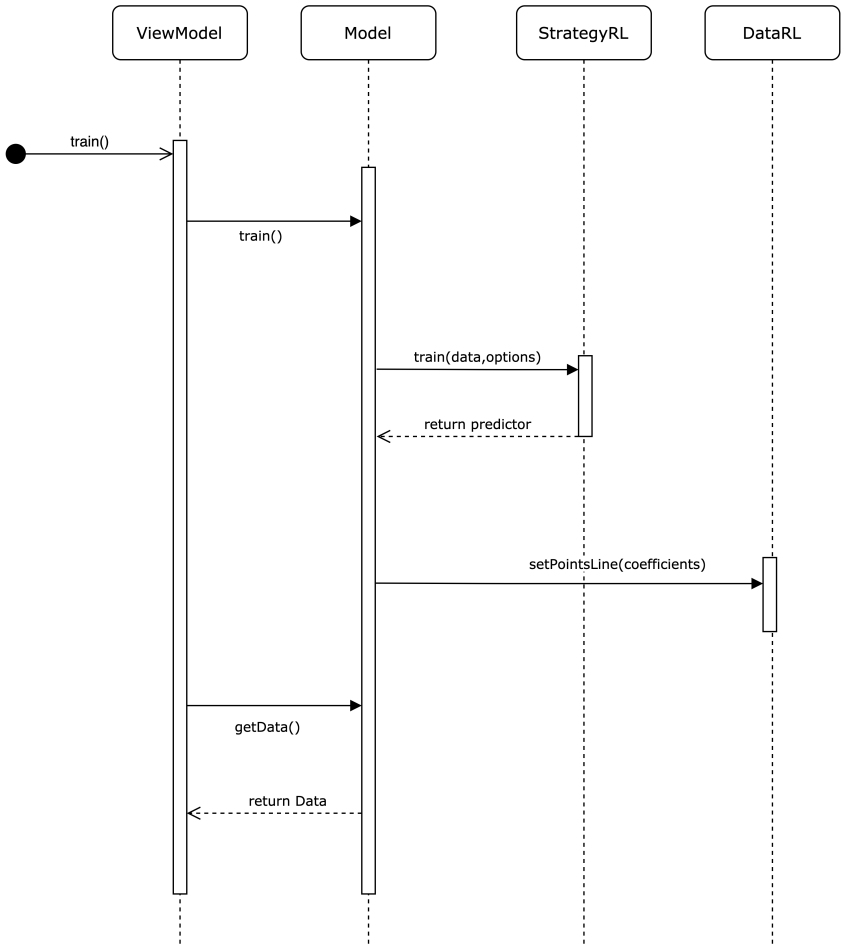

Figura 5: Sequence diagram for the `train()` function.

The `train()` function is one of the two key functions of the external module and is used for training through the *Machine Learning* algorithm chosen by the user. It is invoked in the *ViewModel* at the click of the user in the UI, and from here it is called in the *Model*. Based on the algorithm previously chosen in the UI, the relative Strategy will be used, and the data (stored in the *Data* object) and the configuration options (stored in the *Options* object) will be passed. The predictor represented by the coefficients of the algorithm prediction equation will then be returned. At this point, the `setPointsLine()` method will save the coefficients in the *Data* object that can subsequently be retrieved via the `getData()` method.

In the sequence diagram, for simplicity, only the *Linear Regression* algorithm was taken as an example. Of course, the others algorithms provided by the plug-in will also adhere to the same operating scheme.

5.1.5 `loadData()`

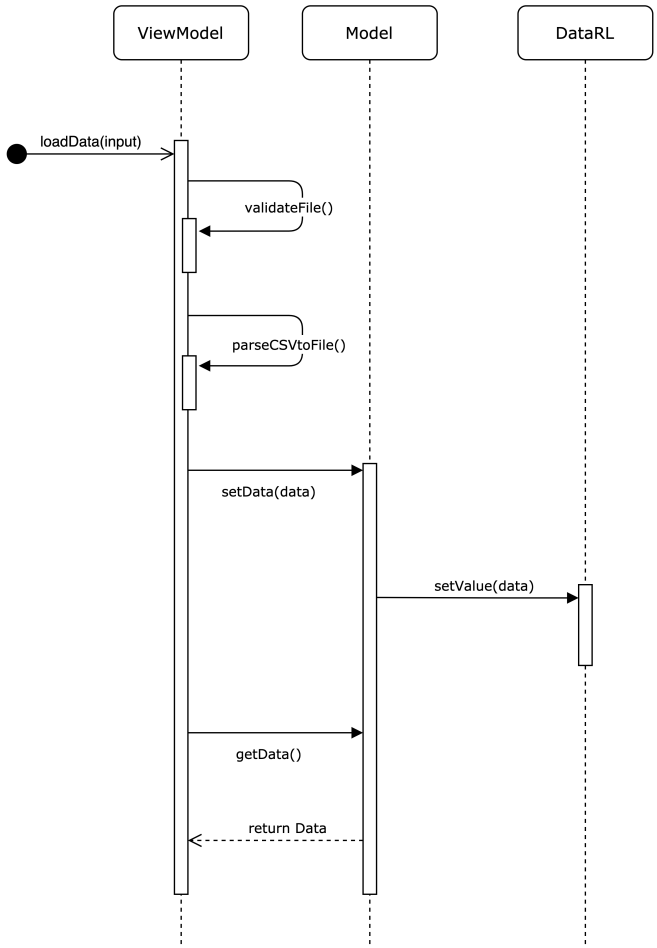

Figura 6: Sequence diagram for `loadData()` function

The second noteworthy function of the external module is the `loadData()` function. Used to load training data into the module, it is invoked once the *CSV* file in the UI is inserted. After an initial check on the correctness of the format of the inserted file (`validateFile()`), and a subsequent parsing (`parseCSVtoFile()`) to transform the content into data usable by the plug-in, the `setData()` method will be invoked for saving the *Data* object of the *Machine Learning* algorithm chosen by the user. Saved data can then be retrieved via the `getData()` method.

In the sequence diagram, for simplicity, only the *Linear Regression* algorithm was taken as an example. Of course, the others algorithms provided by the plug-in will also adhere to the same operating scheme.

5.1.6 Detailed architecture: Classes

The classes are detailed below with the related diagram

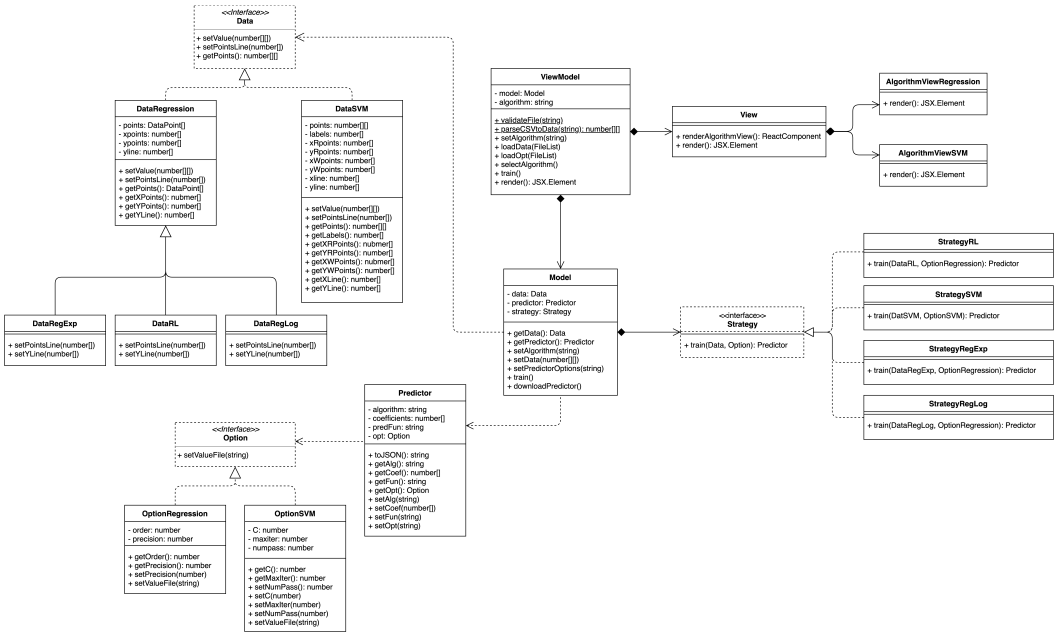

Figura 7: Class diagram of the Training Module

5.2 Prediction Module architecture

5.2.1 Architectural pattern: MVVM

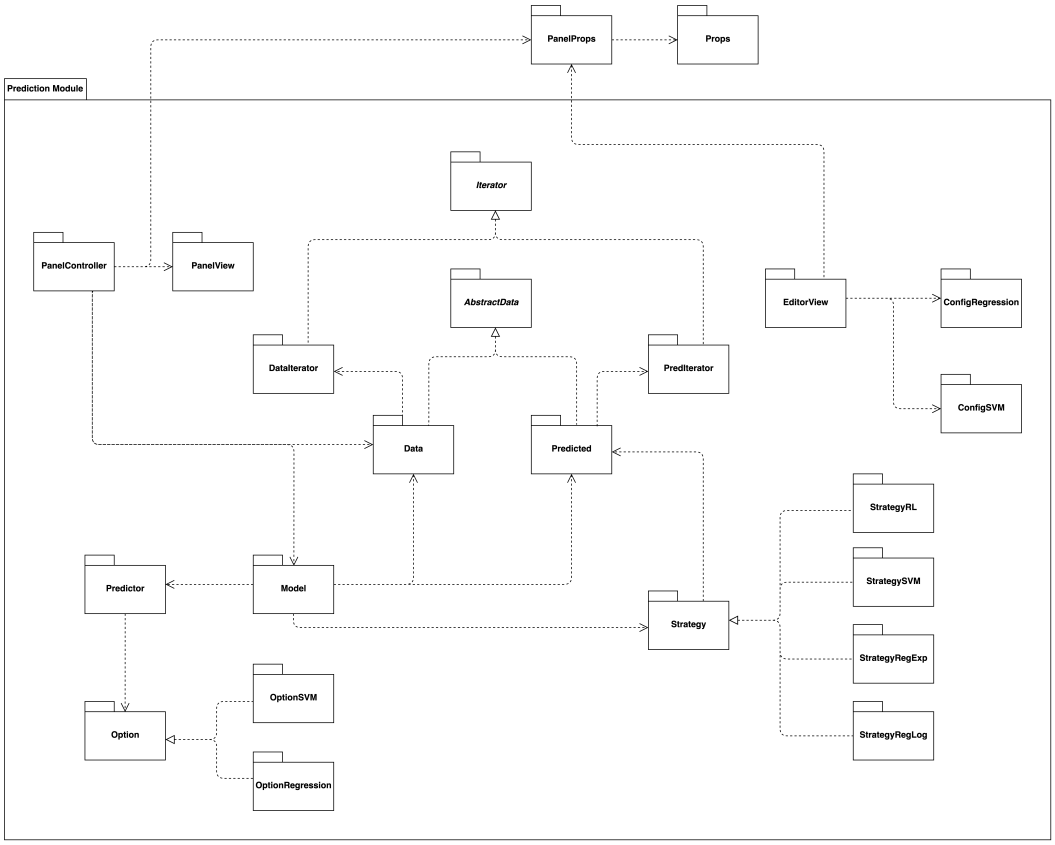

Figura 8: Package diagram for the Prediction Module

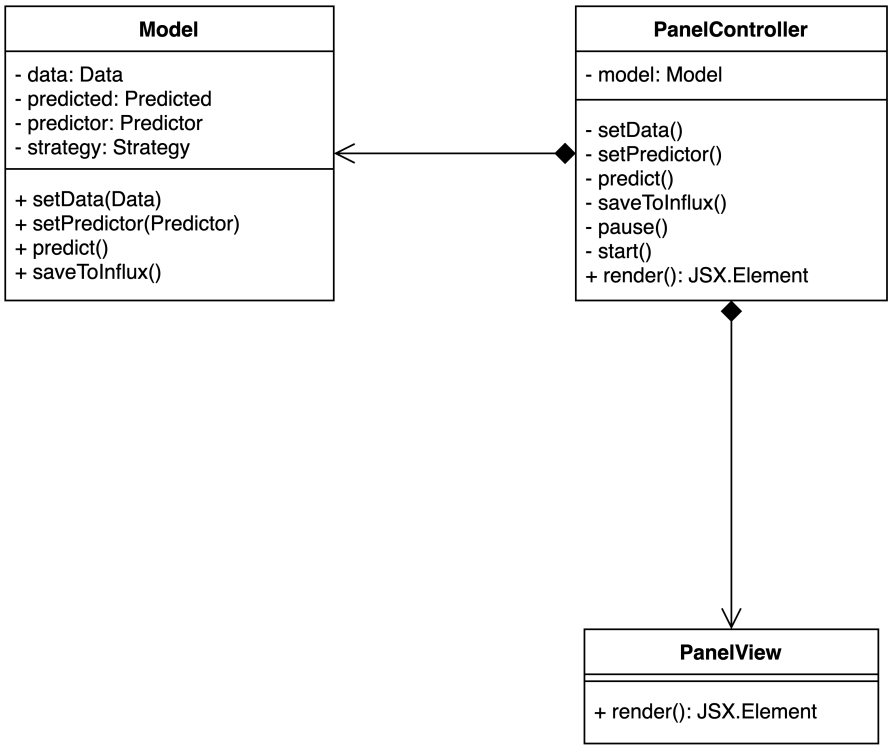

Figura 9: *MVVM* pattern implementation in the Prediction Module

Similarly to the Training Module, the *MVVM* pattern was used for the Prediction Module for the *Panel* architecture. It is composed of the *Model* model, *PanelController* controller and the *PanelView* view. The *PanelController* communicates with the *Editor* part through the `props` variable which is automatically instantiated by *Grafana* and used to store the results of the queries made on the datasource$_{\mathbf{G}}$. As for the *EditorView* instead, it was decided to not use an architectural pattern cause it has only the task of modifying and updating the `props` variable. It will be the task of the main panel to take charge of controls, get the queries (`setData()`), make the prediction (`predict()`), and store the data again in the database (`saveToInflux()`). It will be the *Panel* that will be continuously updated at each time interval, specified by the user through a specific selector already implemented by the *Grafana* platform.

5.2.2 Detailed architecture: Strategy

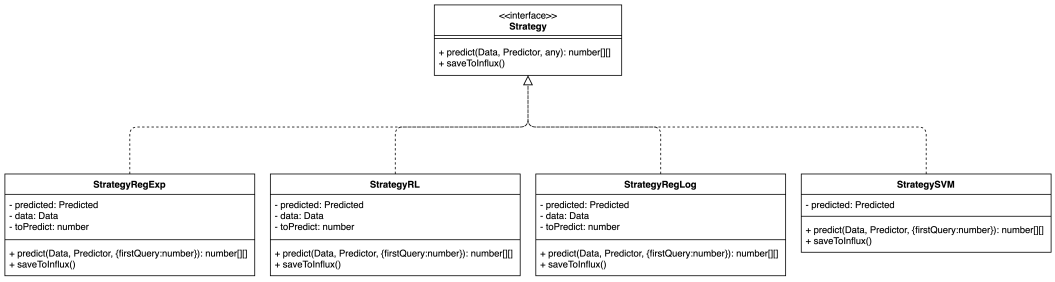

Figura 10: Strategy for the reconstruction of the prediction function

The Strategy design pattern was also used for the Prediction Module. The aim is to differentiate algorithms of *Linear Regression*$_{\mathbf{G}}$, *Exponential Regression*$_{\mathbf{G}}$, *Logarithmic Regression*$_{\mathbf{G}}$ and *Support Vector Machine*$_{\mathbf{G}}$ in order to recreate the prediction function

calculated from the coefficients passed by the *JSON* file. Specifically, this choice is made at runtime using an associative array. It consists of only one `predict()` method that accomplishes this. Each specific strategy implements the `saveToInflux()` method to save the values of the prediction made within the setted datasource.

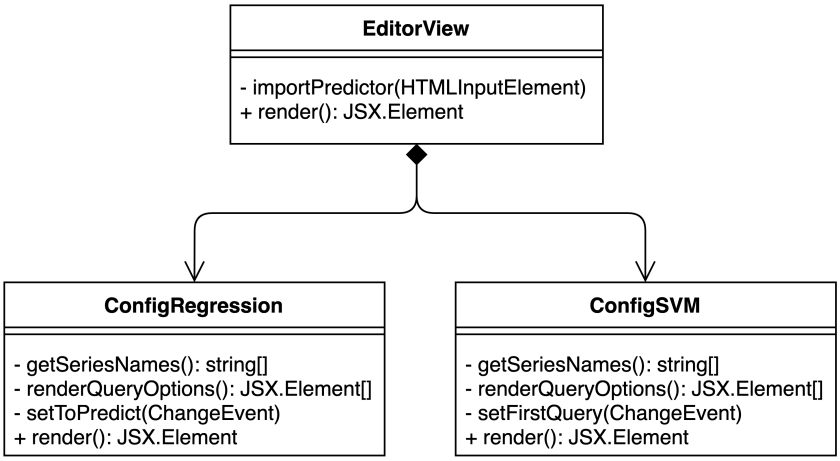

Figura 11: Views based on imported predictor

Thanks to the Strategy pattern it is possible to choose the specialized view *EditorView* to render. *ConfigRegression* and *ConfigSVM* are instantiated here only if the predictor was first imported using the `importPredictor()` method, otherwise an empty component is rendered with the `render()` method. They have the task to display the function in a string format and they can modify the `opt` of the predictor, which contains the methods `toPredict()` or `firstQuery()`. *ConfigRegression* and *ConfigSVM* will render these configurations and, upon their change, report the changes to the `props`. *ConfigRegression* is used for all types of regression (*Linear, Exponential and Logarithmic*) as they are interchangeable regarding their visualization.

5.2.3 `updatePrediction()`

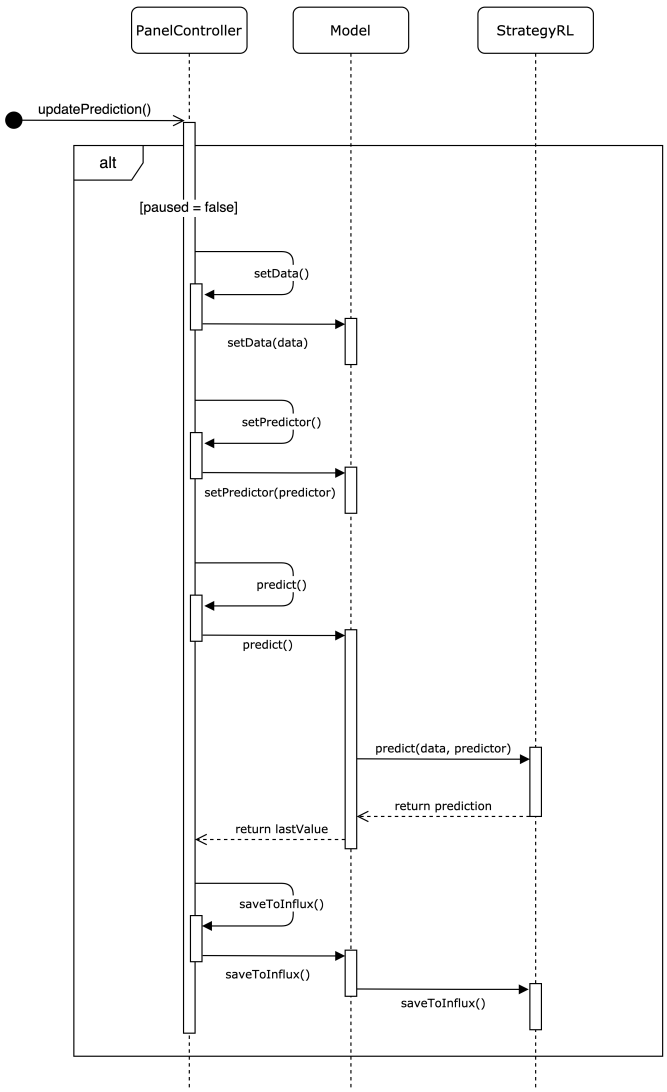

Figura 12: Sequence diagram for `updatePrediction()` function

The sequence diagram describes the `updatePrediction()` function. It is used to update the realtime prediction during active monitoring (`paused=false`). Once the data to be monitored from the datasource (`setData()`) and the predictor previously calculated in the training module (`setPredictor ()`) have been taken, the `predict()` method will be invoked and will calculate the predicted values to be subsequently displayed in the *Grafana* graph. This method will input the data to be monitored and the predictor. Predictor will also contain information about the *Machine Learning* algorithm used previously in the training phase. The output, resulting from the execution of `predict()`, will be the last predicted value calculated in the time interval chosen by the user in the *Grafana* dashboard. The values will be constantly saved in the database (`saveToInflux()`) and then be shown in the visualization graph.

In the sequence diagram, for simplicity, only the *Linear Regression* algorithm was taken as an example. Of course, the others algorithms provided by the plug-in will also adhere to the same operating scheme.

5.2.4 Detailed architecture: Classes

Detailed classes with related diagram are listed below.

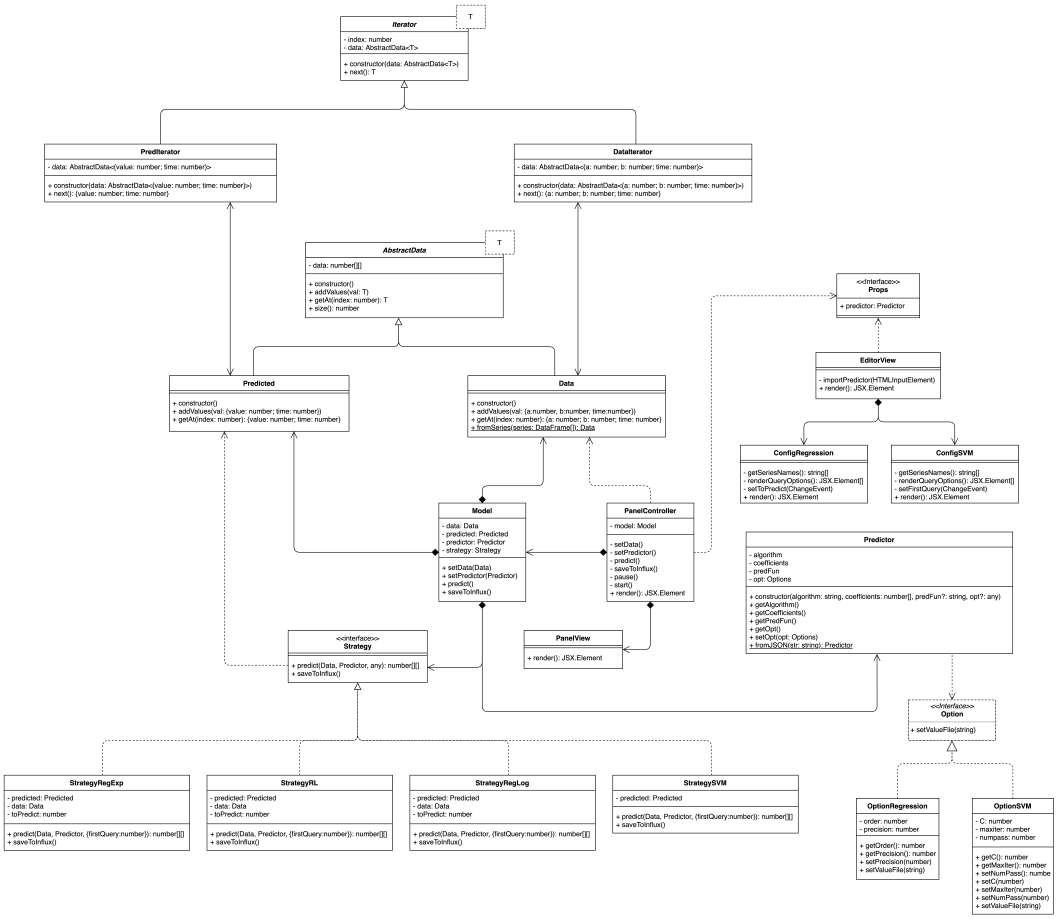

Figura 13: Class diagram of the Prediction Module

Note: within the Prediction Module the possibility of performing the training directly within the *Grafana* software was implemented. The architecture that allows this operation is similar to that presented in the section of the Training Module §5.1, therefore no further documentation is provided. However, the same data structures (`Predictor` and `Options`) are used to store information relating to the predictor and the configuration options of the *ML* algorithm. The rest can be understood as a completely detached module from which the *porting* was made.

6 Extending *Predire in Grafana*

This section explains how to extend the *Predire in Grafana* product. The additions in this case are intended as related to the *Machine Learning* algorithms that the product natively supports.

6.1 Training Module

To add a new type of *Machine Learning* algorithm in the Training Module you will need to:

- inside the `strategies.ts` file, add a new Strategy (*NewConcreteStrategy*) related to the new algorithm. It will be necessary to update the associative array containing the list of of the algorithms identifiers with the id related to the new algorithm;

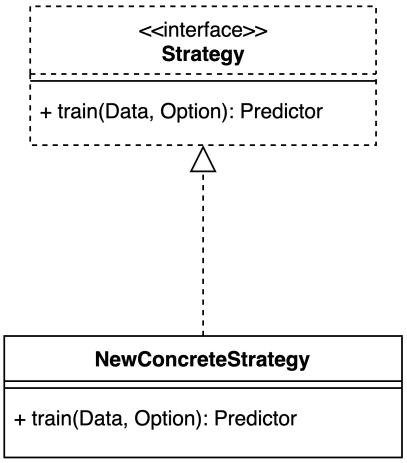

Figura 14: Adding a new strategy *NewConcreteStrategy*

- add a new View (*NewView*) for the specialized view, updating the `renderAlgorithmView()` method to render the new view just added;

Figura 15: Adding a new specialized view *NewView*

- add a new class of Options related to the new algorithm (*NewOptionAlgorithm*). These configuration options will be specific for each new training algorithm and will be specific of any *Machine Learning* library used. To generalize the addition, the figure indicates it as a 1..*N* list of new options;

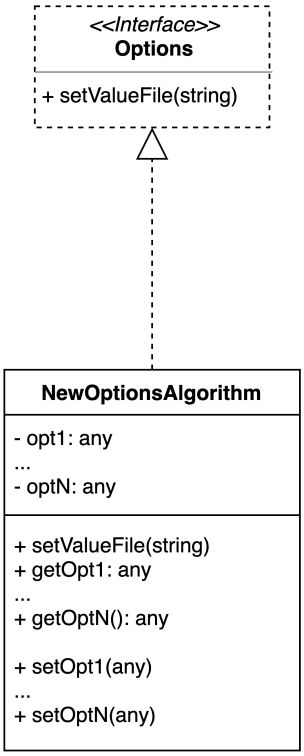

Figura 16: Adding a new class of Options *NewOptionAlgorithm*

- add a new Data class referred to the algorithm data to be represented in the graph (*NewDataAlgorithm*). The type of data will depend on the implemented algorithm and on what information will need to be represented. The methods `setValue(number[][]`, `setPointsLine(number[])` and `getPoints()` will be implemented to obtain the information of the data, to set them and to rapresent them as points in the graph.

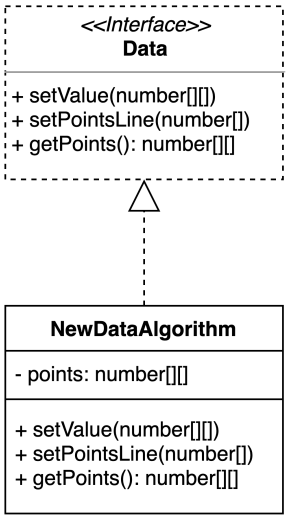

Figura 17: Adding new Data class *NewDataAlgorithm*

6.2 Prediction Module

To add a new type of *Machine Learning* algorithm inside the Prediction Module you will need to:

- add a new Strategy (*NewConcreteStrategy*) related to the new algorithm. It will be necessary to update the associative array containing the list of the algorithms identifiers with the id related to the new one;

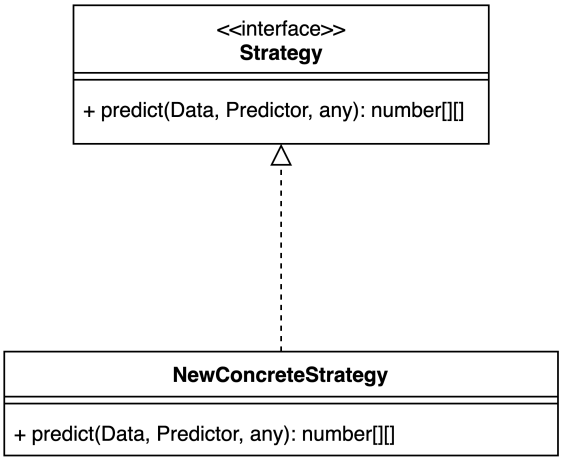

Figura 18: Adding new algotithm *NewConcreteStrategy*

- add the configuration file for the new *React* component (*NewConfig*). This component will allow you to edit the algorithm configuration (such as the *query* to predict) and will depend on the algorithm that you want to implement.

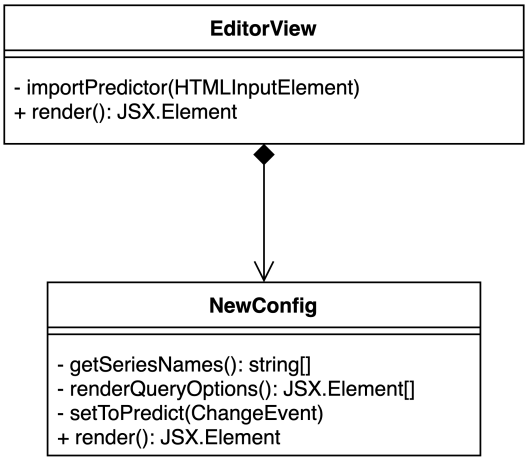

Figura 19: Adding new component configuration *NewConfig*

- add a new class of Options related to the new algorithm (*NewOptionAlgorithm*). These configuration options will be specific to each new algorithm. To generalize the addition, the figure indicates it as a $1..N$ list of new options;

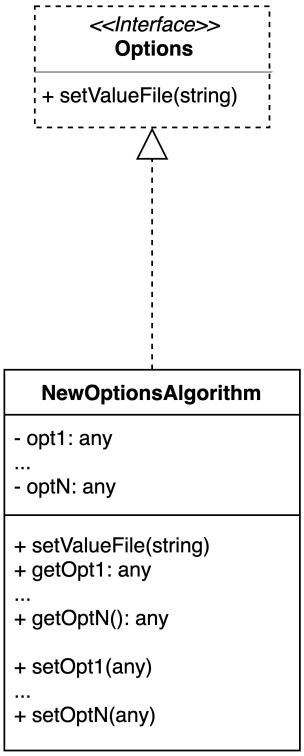

Figura 20: Adding new Options class *NewOptionAlgorithm*

Note: within the Prediction Module the possibility of performing the training directly within the *Grafana* software was implemented. The architecture that allows this operation is similar to that presented in the section of the Training Module §5.1, therefore no further documentation is provided. However, the same data structures (`Predictor` and `Options`) are used to store information relating to the predictor and the configuration options of the *ML* algorithm. The rest can be understood as a completely detached module from which the *porting* was made. If you wish to extend the algorithms available for training inside the Prediction Module, you can refer to what is described in §6.1 as completely analogous.

A Glossary

A

API: Acronym for *Application Programming Interface*. Indicates a set of procedures (generally grouped by specific tools) suitable for carrying out a given task. Often this term represents the software libraries of a programming language.

C

Client-Server: Network architecture in which a computer *client* or terminal is connected to a *server* using a specific service.

Continuous Code Quality: Aspect of the *Continuous Integration*$_{\mathbf{G}}$ which includes the automatic test running and the static-dynamic analysis of the code at every build. It is a way to ensuring the quality of the code.

Code Coverage: In testing activty, it is the measure of how many lines/blocks/arcs of code are executed while the automatic tests are in progress.

Continuous Integration: Practice in which the development activity takes place through a versioning system; it consists in frequently aligning the working environments towards the shared environment. Acronym used: CI.

CSV: *Comma-separated values* is a text file-based format used for importing and exporting a data tables.

D

Datasource: It is a name assigned to the connection set on a database by a server.

DevOps: Software development method that aims at communication, collaboration and integration between developers and *information technology(IT)* operators. DevOps will is to respond to the interdependence between software development and *IT operations*, aiming help an organization in developing software products and services faster and more efficiently. Lot of importance is given to quality.

DOM: *Document Object Model*, it is a form of representation of structured documents as an object-oriented model.

E

EsLint: Static code analysis tool to identify problematic patterns found in *JavaScript*$_{\mathbf{G}}$ code.

- https://eslint.org/.

G

Git: Distributed version control system to track changes to source code during software development. It's designed to coordinate work between programmers, but can be used to track changes in any set of files. Its goals are speed, data integrity and support for distributed and nonlinear workflows.

GitHub: Hosting service for software projects. Implementation of the distributed version control tool *Git*$_{\mathbf{G}}$.

Grafana: System used by Zucchetti S.p.A. for monitoring applications. This functionality becomes particularly relevant within a *DevOps*$_{\mathbf{G}}$ scenario with three main tasks:

- system health check and subsequent verification that performance falls within precise parameters;
- identification of weak points to be corrected by the development team;
- supply of elements to define a scale of priorities in improvements and new implementations.

The monitoring scope is quite broad; if extreme situations are detected, alerts (e-mails) are sent to managers.

- Plug-in development guide
  https://grafana.com/docs/grafana/latest/plugins/developing/development/;
- Plug-ins code styleguide *Grafana*
  https://grafana.com/docs/grafana/latest/plugins/developing/code-styleguide/;

I

InfluxDB: is an *open-source time series database (TSDB)* developed by InfluxData. It is written in *Go* and optimized for time series high-availability storage related to data monitoring operations.

- https://docs.influxdata.com/

J

JavaScript: object and event oriented scripting language, used in both client and server side web programming.

Jest: *open-source* testing framework developed by *Facebook* and used to testing *JavaScript* code.

- https://github.com/facebook/jest

M

MIT License: license for the open-source code, born at the *Massachusetts Institute of Technology (MIT)* in late 1980s. It restricts the reuse of the code protected by this license.

N

NodeJS: runtime for executing *JavaScript*$_{G}$ code. It is *open-source*$_{G}$, cross-platform and event-oriented.

- https://nodejs.org/

O

Open-Source: term used in IT fields to refer to a software in which the actors make the source code public. All of that is regulated through the application of user licenses. Open-source software allows worldwide programmers to coordinate and work on the same project.

R

React: *JavaScript*$_{G}$ library developed by *Facebook*. It allows the creation of Web Interfaces.

- https://github.com/facebook/react

Exponential Regression: variant of the *Linear Regression* $_{G}$ in which the data provided, not allowing the representation of a straight line, are transformed with mathematical operations to be arranged to represent an exponential type function. The recommended external library for training via *Exponential Regression* is:

- https://github.com/Tom-Alexander/regression-js

Linear Regression: acronym *RL*, is a technique for predicting numerical values using the "least squares" method. The *Linear Regression*, as the name implies, imagines that the law underlying the observed data can be expressed with a straight line. Each observed point is placed in a system to determine the straight line coefficients. Since it would be an overestimated system as soon as there are more than two points (so impossible to be resolved), the sum of the square of all the differences between the found values and the estimated values is considered. Minimizing this sum, we find the best straight line to approximate data. The recommended external library for training via *RL* is:

- https://github.com/Tom-Alexander/regression-js

Logaritmic Regression: variant of the *Linear Regression*$_{G}$ in which the data provided, not allowing the representation of a straight line, are transformed with mathematical operations to be arranged due to represent a logarithmic type function. The recommended external library for training via *Logaritmic Regression* is:

- https://github.com/Tom-Alexander/regression-js

Repository: also abbreviated as "Repo", it's an environment of an information system in which files, documents and metadata, relating to a project activity, are stored and managed.

S

SonarCloud: Cloud code quality analysis service, open-source$_{\mathbf{G}}$ and free, for repositories. Developed by *SonarQube* developers, it offers the possibility to import repositories from code hosting services such as *GitHub*$_{\mathbf{G}}$.

- https://sonarcloud.io/

Support Vector Machine: algorithms used to classify, in order to solve the "curse of dimensionality" that arises when dealing with large amounts of data. In this context, a thinning of data is observed as dimensions are added through the consideration of several variables.
The *SVM* looks for the hyperplane that best divides observed data into two classes. In this way it well resist also to the increase of dimensions and the thinning of points in the space. The recommended external library for training via *SVM* is:

- https://github.com/karpathy/svmjs

T

Telegraf: a server agent specialized in receiving and sending information and events from/to databases, systems, and *IoT* sensors.

- https://github.com/influxdata/telegraf

Time Series Database: often used as an acronym (*TSDB*), it is an optimized software system for storing and making usable historical series of objects, i.e. objects that associate time values with numerical values that took place in a specific instant of time.

Throughput: actual capacity of a telecommunication channel.

TypeScript: free and *open-source*$_{\mathbf{G}}$ programming language developed by *Microsoft*. This is a typed superset of *JavaScript*$_{\mathbf{G}}$.

U

UML: *United Modeling Language*. Modeling and specification language based on the object-oriented paradigm, featured by visual, semi-graphic and semi-formal notation. It provides lot of diagrams, formal textual elements and free text elements, which make up a unique language, in order to simplify object-oriented design and programming.

UNIX: a portable operating system for computers initially developed by a research group of the *AT&T and Bell Laboratories*, in which Ken Thompson and Dennis Ritchie also firstly appeared. Historically it has been the most used OS on mainframe systems since the seventies.

Y

Yarn: *NodeJS* package manager, compatible with *npm* package registry.

- https://github.com/yarnpkg/yarn