Developer Manual

TeamAFK - Project Predire in Grafana

gruppoafk15@gmail.com

Informations about the document

Version	1.0.0
Approval	Olivier Utshudi
Drafting	Simone Federico Bergamin Fouad Farid
Check	Simone Meneghin
Use	External
Addressed to	Prof. Vardanega Tullio Prof. Cardin Riccardo TeamAFK

Description

Developer manual made by *TeamAFK* for the project *Predire in Grafana*.

Changelog

Version	Date	Description	Name	Role
1.0.0	2020-07-15	Document approved for RA	Olivier Utshudi	*Manager*
0.1.2	2020-07-09	Updated §7. Document checked.	Fouad Farid Simone Meneghin	*Programmer Verifier*
0.1.1	2020-07-04	Removed §3.5. Updated §9. Document checked.	Simone Federico Bergamin Simone Meneghin	*Programmer Verifier*
0.1.0	2020-06-10	Document approved for RQ	Olivier Utshudi	*Manager*
0.0.6	2020-06-04	Written and checked section §7 - §A	Davide Zilio Fouad Farid	*Programmer Verifier*
0.0.5	2020-06-04	Written and checked section §6	Simone Federio Bergamin Fouad Farid	*Programmer Verifier*
0.0.4	2020-06-04	Written and checked section §5	Olivier Utshudi Simone Meneghin	*Programmer Verifier*
0.0.3	2020-06-04	Written and checked section §4	Simone Federico Bergamin Simone Meneghin	*Programmer Verifier*
0.0.2	2020-06-03	Written and checked sections §2- §3	Olivier Utshudi Simone Meneghin	*Programmer Verifier*
0.0.1	2020-06-03	Written and checked section §1	Simone Federico Bergamin Fouad Farid	*Programmer Verifier*

Contents

1 Introduction . . . **6**
- 1.1 Document's purpose . . . 6
- 1.2 Predire in Grafana's purpose . . . 6
- 1.3 Glossary . . . 6
- 1.4 References . . . 6
  - 1.4.1 Technologies . . . 6
  - 1.4.2 Legal . . . 6
  - 1.4.3 Informative . . . 7

2 System requirements . . . **8**
- 2.1 Minimum hardware requirements . . . 8
- 2.2 Compatible operating systems . . . 8
- 2.3 Compatible browsers . . . 8

3 Installation . . . **9**
- 3.1 Instruments installation . . . 9
  - 3.1.1 Node.js installation . . . 9
  - 3.1.2 Git installation . . . 9
- 3.2 Grafana installation . . . 9
  - 3.2.1 WEB Grafana execution . . . 9
- 3.3 Plugin and Tool installation . . . 10
  - 3.3.1 GitHub repository clone . . . 10
- 3.4 Dependencies . . . 10
  - 3.4.1 Developer dependency . . . 11

4 System enviroment configuration . . . **13**
- 4.1 IntelliJ IDEA configuration . . . 13
  - 4.1.1 Path system configuration . . . 13
- 4.2 Project import . . . 14
- 4.3 ESLint configuration . . . 15
  - 4.3.1 Automatic configuration . . . 15
  - 4.3.2 Manual configuration . . . 16
- 4.4 Grafana plugin panel enviroment configuration . . . 17
  - 4.4.1 package.json content . . . 17
- 4.5 Training tool enviroment configuration . . . 18
  - 4.5.1 package.json content . . . 18

5 Test . . . **20**
- 5.1 Edit tests . . . 20
- 5.2 Code Coverage . . . 20

6 Technologies . . . **21**
- 6.1 ESLint . . . 21
- 6.2 InfluxDB . . . 21
- 6.3 Telegraf . . . 21

6.4 NodeJS . . . 21
6.5 NPM . . . 21
6.6 React . . . 21
6.7 JSX . . . 21
6.8 TypeScript . . . 21
6.9 JSON . . . 21
6.10 CSV . . . 22
6.11 Travis CI . . . 22
6.12 Linear Regression . . . 22
6.13 Support Vector Machine . . . 22

7 Product architecture . . . 23
7.1 Training tool . . . 23
7.1.1 Architectural design . . . 23
7.1.2 Detailed design . . . 24
7.1.2.1 Model . . . 24
7.1.2.2 View . . . 25
7.1.2.3 ViewModel . . . 28
7.2 Prediction plug-in . . . 33
7.2.1 Architectural design . . . 33
7.2.1.1 Grafana's role . . . 34
7.2.2 Detailed design . . . 34
7.2.2.1 Model . . . 34
7.2.2.2 View . . . 35
7.2.2.3 Controller . . . 36

8 Product extensibility . . . 40
8.1 Integration of new prediction algorithm . . . 40

9 File structure . . . 41
9.1 JSON file structure . . . 41
9.1.1 Linear Regression . . . 41
9.1.2 Support Vector Machine . . . 41
9.2 CSV file structure . . . 42

A Glossary . . . 44

List of Figures

3.2.1 WEB Grafana page at `http://localhost:3000` . . . 10
4.1.1 IntelliJ IDEA first execution . . . 13
4.1.2 Node.js and NPM settings . . . 14
4.2.1 Project path on opening . . . 15
4.3.1 Automatic ESLint configuration . . . 16
4.3.2 Manual ESLint configuration . . . 17
7.1.1 Training tool package diagram . . . 23
7.1.2 Training tool Model class diagram . . . 25
7.1.3 Training tool Model class diagram . . . 28
7.1.4 Training tool Model class diagram . . . 30
7.1.5 TrainSVM sequence diagram . . . 32
7.2.1 Prediction plug-in package diagram . . . 33
7.2.2 Prediction plug-in Model class diagram . . . 35
7.2.3 Prediction plug-in View class diagram . . . 36
7.2.4 Prediction plug-in Controller class diagram . . . 38
7.2.5 `viewGraph()` sequence diagram . . . 39
9.1.1 Example of Linear Regression JSON file . . . 41
9.1.2 Example of Support Vector Machine JSON file . . . 42
9.2.1 Example of Linear Regression CSV file . . . 43
9.2.2 Example of Support Vector Machine CSV file . . . 43

List of Tables

3.4.1 Table of Training Tool dependency 10
3.4.2 Table of Prediction Plugin dependency 11
3.4.3 Table of Prediction Tool developer dependency 11
3.4.4 Table of Prediction Plugin developer dependency 12

1 Introduction

1.1 Document's purpose

The developer's manual allows each developer reader to absorb *Predire in Grafana*$_G$'s product key information to maintain and extend the product itself. This document describes the product in its totality, giving the developer an exhaustive explanation required for his tasks.

1.2 Predire in Grafana's purpose

Predire in Grafana is a plugin made for Grafana which is an open source$_G$ platform commonly used to analyse data series. The plugin allows users to predict datas on a stream data. *Predire in Grafana* plugin uses a JSON$_G$ file which contains a trained algorithm definition to get predictions. Users can use an external training tool, which use Machine Learning$_G$, to get these JSON' files. At the moment only Support Vector Machine and Linear Regression algorithm are implemented. In more detail input datas, like cpu's usage and cpu's temperature, are constantly monitored to get predictions on the aspect you want to examine. Predictions are shown through Grafana GUI$_G$ and continue to be updated after being calculated from datas coming from a database. Thanks to this operators can monitor each process and intervene at the root of the problem whenever necessary.

1.3 Glossary

At the end of the document in the appendix is available a glossary where explanations for new or ambiguous terms can be found. These are marked with a subscript G.

1.4 References

1.4.1 Technologies

These links are a reference to the documentation of these specific technologies:

- Node.js: `https://nodejs.org/en/docs/`;
- Git: `https://git-scm.com/doc`;
- Grafana: `https://grafana.com/docs/grafana/latest/`;
- Grafana plugin: `https://grafana.com/docs/grafana/latest/developers/plugins/`;
- React: `https://reactjs.org/docs/getting-started.html`;
- ESLint: `https://www.jetbrains.com/help/idea/eslint.html`.

1.4.2 Legal

- Apache license: `https://www.apache.org/licenses/LICENSE-2.0`.

1.4.3 Informative

- https://en.wikipedia.org/wiki/Linear_regression;
- https://en.wikipedia.org/wiki/Support_vector_machine.

2 System requirements

Here the requirements for use of the product are listed.

2.1 Minimum hardware requirements

Here the requirements for use of the product are listed.

- 2GB of RAM;
- 5GB of space on a drive;
- Dual core processor.

2.2 Compatible operating systems

The software was developed and tested on the following operating systems:

- Windows 10;
- MacOS 10.15;
- Ubuntu 18, 20.

2.3 Compatible browsers

Predire in Grafana can be accessed through the following browsers:

- Google Chrome: version 58 or newer;
- Mozilla Firefox: version 54 or newer;
- Apple Safari: version 10 or newer;
- Microsoft Edge: version 14 or newer;
- Opera version: 55 or newer.

3 Installation

3.1 Instruments installation

3.1.1 Node.js installation

The installation of the runtime JavaScript Node.js can be done by visiting Node.js page at https://nodejs.org/en/download/. In this site the developer can download the most suitable version of Node.js for his operating system. For Linux user is also possible to use the package manager provided by the OS$_G$ and exclusively for Ubuntu/Debian developers can run in terminal this code:

```
apt-get install nodejs
```

3.1.2 Git installation

For the installation of the version control system, the developer needs to reach Git site's at https://git-scm.com/downloads. Inside the 'Download' section are available the links to download the compatible version with his operating system. Also Linux base system can install Git from their package manager or running from Ubuntu/Debian terminal this line:

```
apt-get install git
```

3.2 Grafana installation

Developers can install Grafana by reaching its download page at https://grafana.com/grafana/download. For a correct using of the plugin you must select Grafana 6.7.3 version.There they can download compatible version for MacOs, Windows and Linux base system. Furthermore, the most common Unix base systems can install Grafana running terminal lines showed in the same page.

3.2.1 WEB Grafana execution

Depending from which OS the developer is working on,the execution of WEB Grafana can be done by:

- **Windows**: opening "bin" folder in Grafana installation folder (the path should be like `C:/Program Files/grafana-6.7.3/bin/`), and double-clicking on "grafana-server.exe";
- **Linux and MacOS**: running on terminal the following line (may `sudo` option is needed):
  ```
  systemclt start grafana-server
  ```

After that, the developer needs to reach the address http://localhost:3000/ through a browser. For the first access, the developer needs to fill the fields username with "admin" and password with "admin", and once he/she is in, he/she will need to register himself/herself into the system for next accesses.

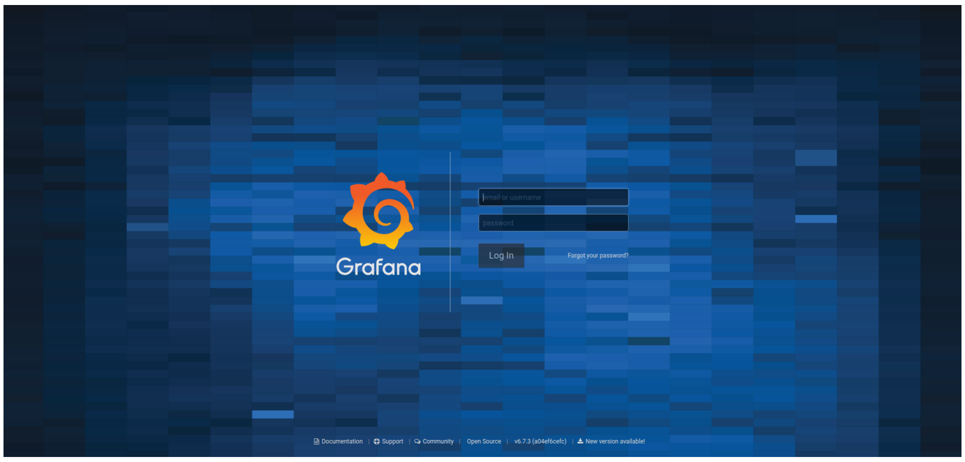

Grafana
email or username
password
Log In
Forgot your password?
Documentation | Support | Community | Open Source | v6.7.3 (a04ef6cefc) | New version available!

Figure 3.2.1: WEB Grafana page at http://localhost:3000

3.3 Plugin and Tool installation

The following sections will guide developers to install correctly both Training Tool and Prediction Plugin.

3.3.1 GitHub repository clone

The developer to clone the GitHub repository needs to open the terminal and choose a folder inside the filesystem with command:

```
cd /path/to/folder
```

After that, in the same location, the developer has to run this line:

```
git clone https://github.com/teamafkSWE/PredireInGrafana-SW.git
```

This line creates the folder that contains the source code of Training Tool and Prediction Plugin.

3.4 Dependencies

Dependencies are a list of packages needed to develop and run the project. For this reason, other developers will need to install all them to run correctly the tool and plugin. This operation can be done by moving to tool or plugin's folder and there run this line:

```
npm install
```

Developers have to run this command once inside the tool's folder and one more time inside the plugin's folder, because the two products have different list of dependencies. The following tables contain all the dependencies adopted to make both tool and plugin.

Table 3.4.1: Table of Training Tool dependency

Packacge	Version
@testing-library/react	9.5.0
@testing-library/user-event	7.2.1

Table 3.4.1: (continua)

Package	Version
bootstrap	4.4.1
chart.js	2.9.3
is-promise	2.2.2
react	16.13.1
react-bootstrap	1.0.1
react-chartjs-2	2.9.0
react-csv-reader	3.5.0
react-dom	16.13.1
react-script	3.4.1

Table 3.4.2: Table of Prediction Plugin dependency

Package	Version
@influxdata/influxdb-client	1.3.0
axios	0.19.2
react-files	2.4.8

3.4.1 Developer dependency

Developer will have to install dependencies to run correctly the project. All them are reported down below.

Table 3.4.3: Table of Prediction Tool developer dependency

Package	Version
@testing-library/jest-dom	4.2.4
csv-parser	2.3.2
eslint-plugin-react-hooks	4.0.4

Table 3.4.4: Table of Prediction Plugin developer dependency

Package	Version
@grafana/data	next
@grafana/toolkit	6.7.2
@grafana/ui	next
webpack	4.43.0

4 System enviroment configuration

4.1 IntelliJ IDEA configuration

A correct IntelliJ IDEA configuration needs to configure the path system$_G$ and after that to import an existing project.

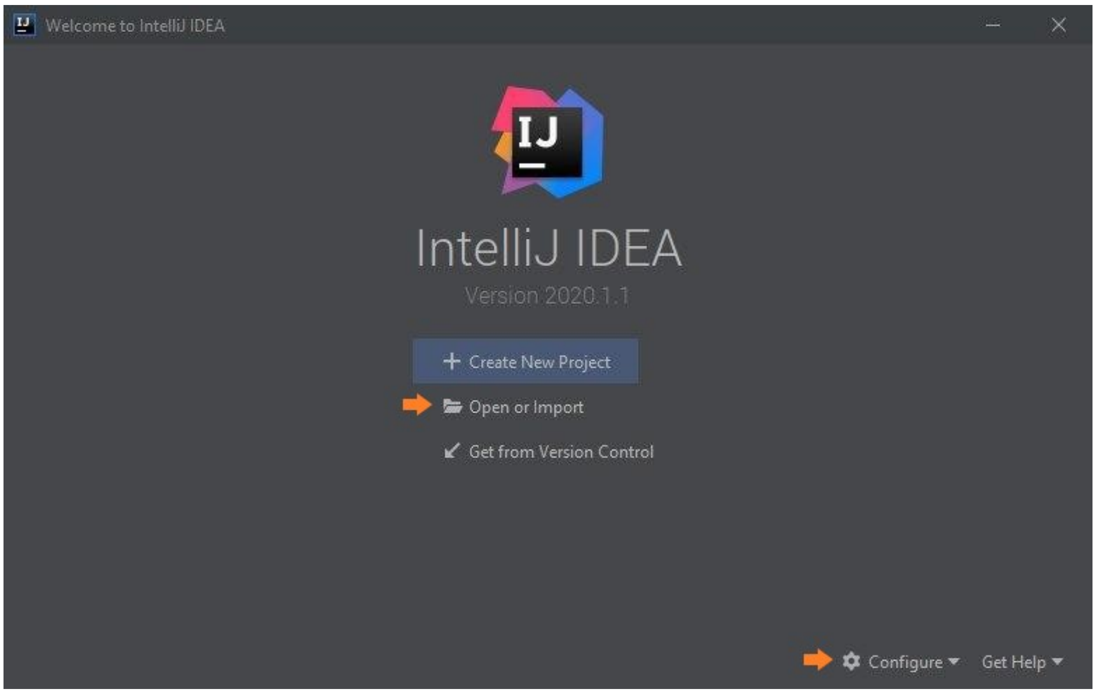
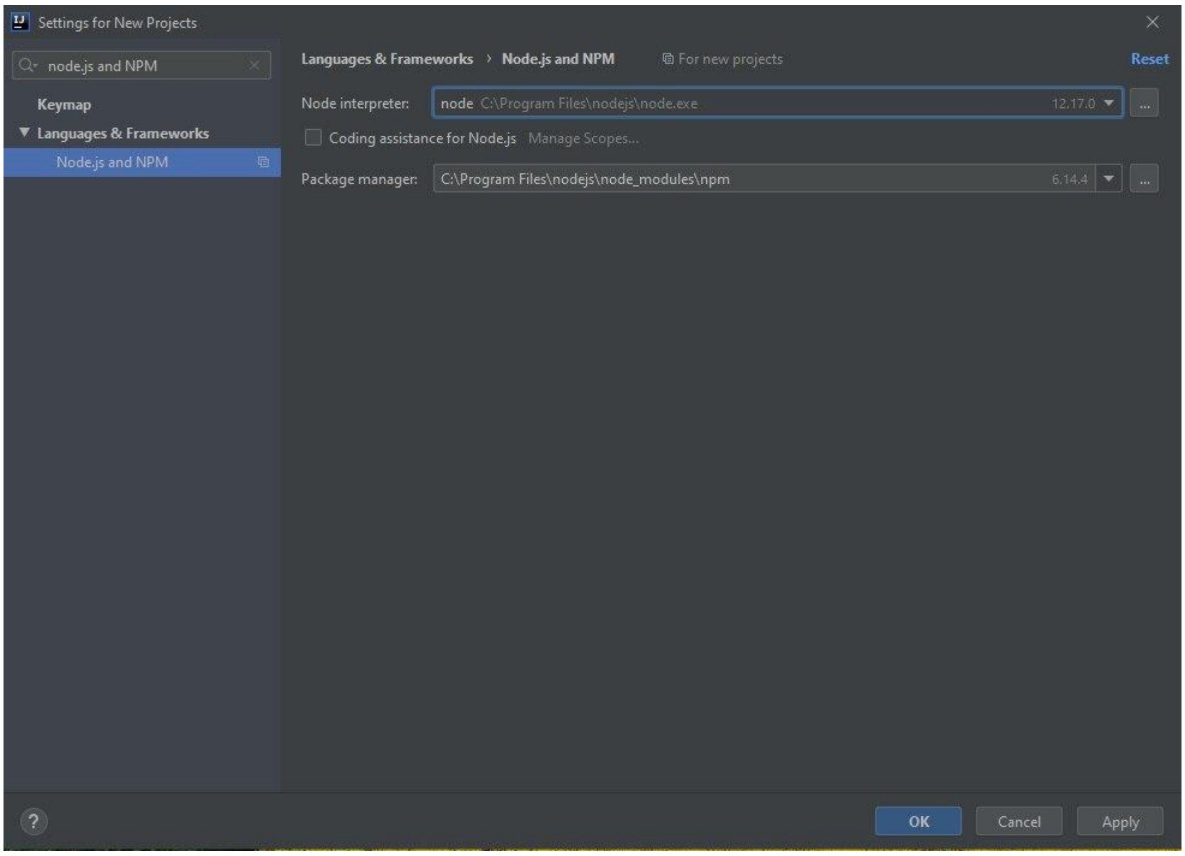


Figure 4.1.1: IntelliJ IDEA first execution

4.1.1 Path system configuration

Once you run IntelliJ IDEA move to "Configure" then "Settings"in the lower-right corner. Write "Node.js and NPM" in the search bar and check for the correct settings of fields "Node interpreter" and "Package manager", otherwise update them with the correct paths.



Node interpreter: node C:\Program Files\nodejs\node.exe 12.17.0

Coding assistance for Node.js

Package manager: C:\Program Files\nodejs\node_modules\npm 6.14.4

Figure 4.1.2: Node.js and NPM settings

4.2 Project import

From IntelliJ IDEA main window click on "Open or Import" and select the root directory of our project repository folder.

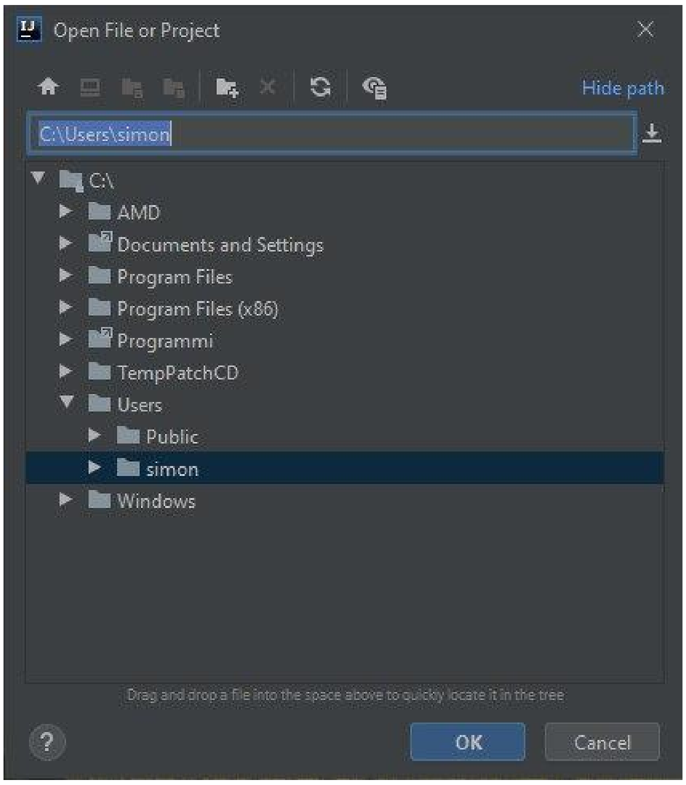
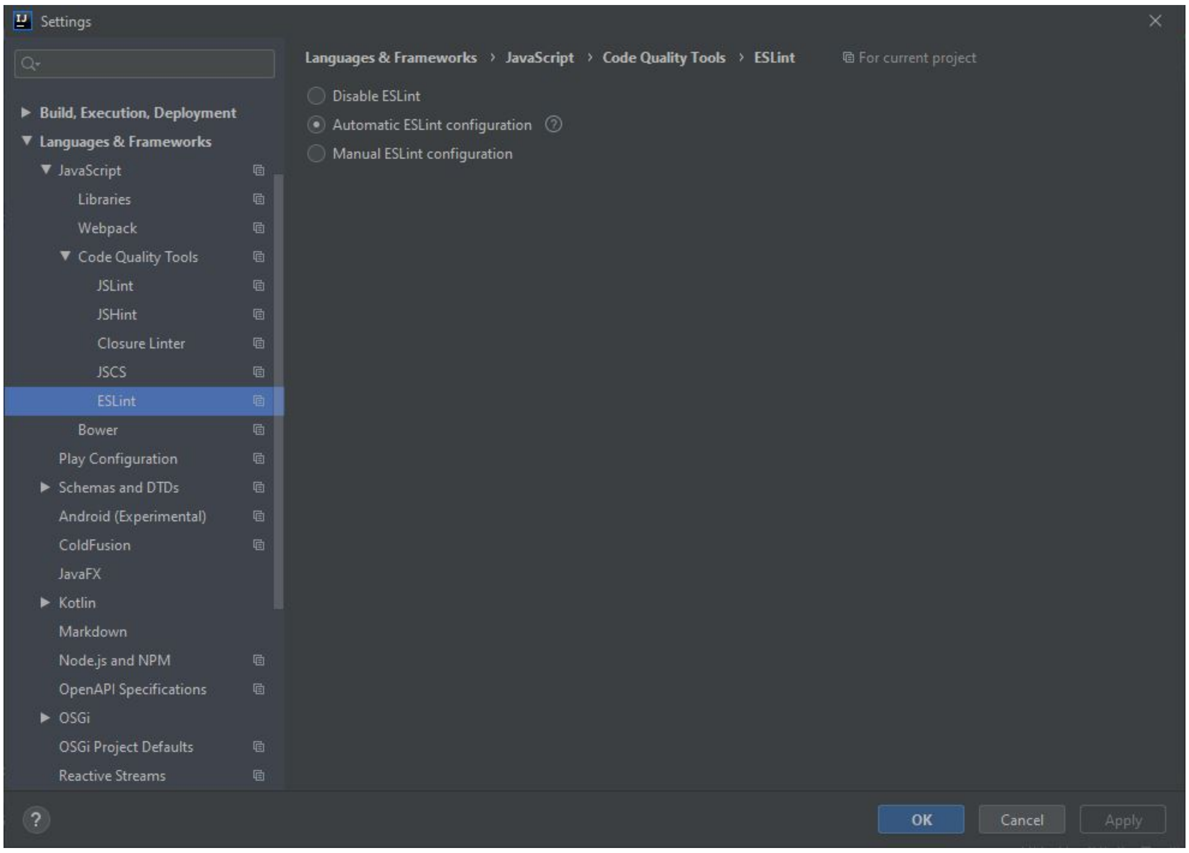


Figure 4.2.1: Project path on opening

4.3 ESLint configuration

4.3.1 Automatic configuration

Since ESLint is listed as a dependency in our project IntelliJ IDEA automatically configures it. Move to "File" | "Settings" | "Languages and Frameworks" | "JavaScript" | "Code Quality Tools" | "ESLint" and check that Automatic ESLint configuration option is enabled.



Languages & Frameworks › JavaScript › Code Quality Tools › ESLint   For current project

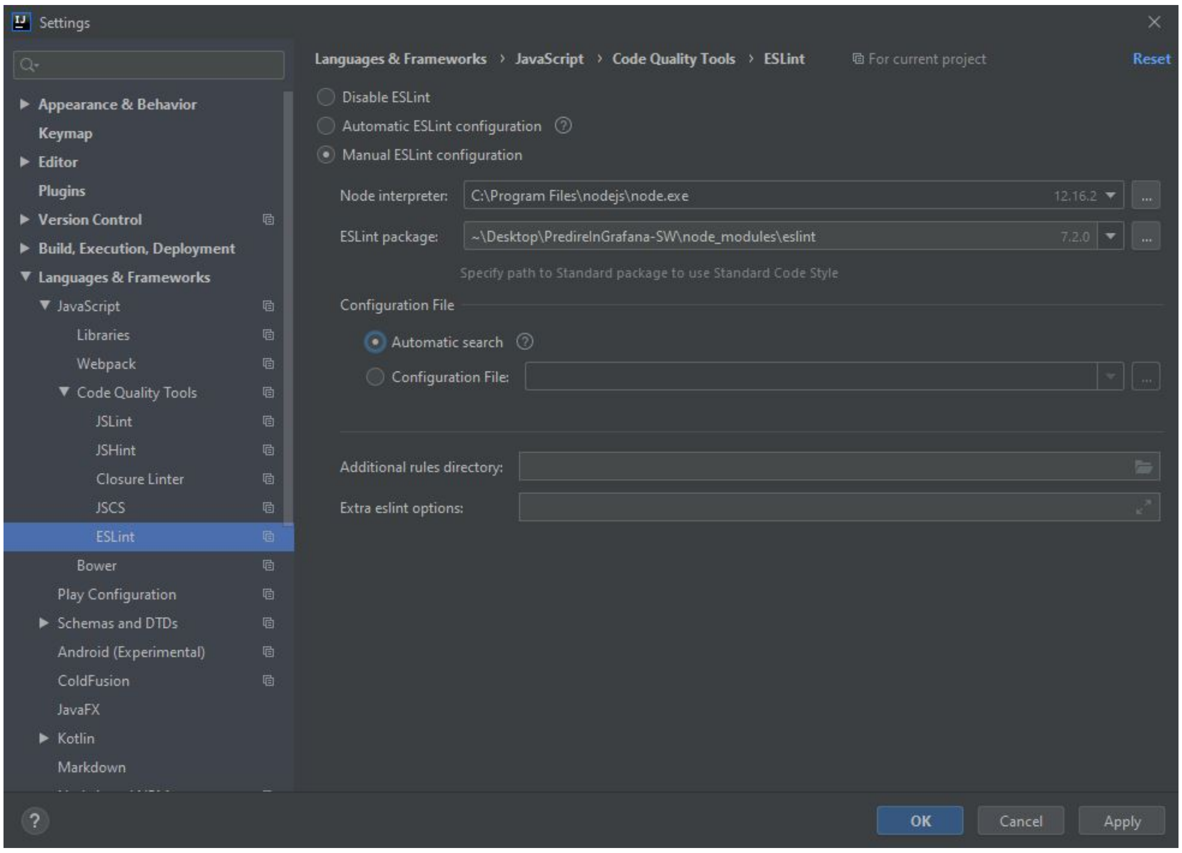
○ Disable ESLint
● Automatic ESLint configuration
○ Manual ESLint configuration

Figure 4.3.1: Automatic ESLint configuration

4.3.2 Manual configuration

You can also configure ESLint manually checking the Manual ESLint configuration option and complete fields as it follows:

1. in the "Node Interpreter" field, specify the path to Node.js;
2. in the "ESLint Package" field, specify the location of the eslint or standard package;
3. choose the configuration file to use checking "Automatic search" if you want to let IntelliJ IDEA do it for you or specify the file location in the path field;
4. optionally specify additional command-line options to run ESLint in "Extra ESLint Options" field and specify the location of the files with additional code verification rules in the "Additional Rules Directory" field then confirm the whole configuration.



Disable ESLint
Automatic ESLint configuration
[✓ Manual ESLint configuration]

Node interpreter: C:\Program Files\nodejs\node.exe 12.16.2
ESLint package: ~\Desktop\PredireInGrafana-SW\node_modules\eslint 7.2.0
Specify path to Standard package to use Standard Code Style

Configuration File
[✓ Automatic search]
Configuration File:

Additional rules directory:
Extra eslint options:

Figure 4.3.2: Manual ESLint configuration

4.4 Grafana plugin panel enviroment configuration

4.4.1 package.json content

In package.json file you can find all the app informations and requirements needed for its proper functioning. Attributes which represent important information are listed below:

- **scripts**: it contains all CLI command lines used from the developer:
  - **build**: this command creates a production-ready build of your plugin. It generates a folder named dist which contains our plugin production files ;
    `npm run build`
  - **test**: this command runs Jest against your codebase (used in automatic tests);
    `npm run test`
  - **dev** : this command creates a development build that's easy to play with and debug using common browser tooling;
    `npm run dev`
  - **watch**: this command run development task in a watch mode.
    `npm run dev -watch`

- **dependencies**: it contains the following packages needed for our app proper functioning:
  - **@influxdata/influxdb-client**: the reference javascript client for InfluxDB. Both node and browser environments are supported;
  - **axios**: promise based HTTP client for the browser and node.js;
  - **react-files**: a file input (dropzone) management component for React we use when loading JSON files.
- **devDependencies**: it contains the following packages needed for our app proper functioning during development:
  - **@grafana/data**: a library containing most of the core functionality and data types used in Grafana.
  - **@grafana-toolkit**: a CLI$_G$ that enables efficient development of Grafana plugins.
  - **@grafana/ui**: a library containing the different design components of the Grafana ecosystem;
  - **webpack**: used to compile JavaScript modules. Once installed, you can interface with webpack either from its CLI or API.

4.5 Training tool enviroment configuration

4.5.1 package.json content

In package.json file you can find all the app informations and requirements needed for its proper functioning. Attributes which represent important information are listed below:

- **scripts**: it contains all CLI command lines used from the developer:
  - **start**: the command runs the app in development mode. Open `http://localhost:3000` to view it in the browser. The page will automatically reload if you make changes to the code. You will see the build errors and lint warnings in the console;
    `npm start`
  - **build**: this command builds the app for production to the build folder. It correctly bundles React in production mode and optimizes the build for the best performance. The build is minified and the filenames include the hashes. In the end our app is ready to be deployed;
    `npm run build`
  - **test**: this command runs the test watcher in an interactive mode. By default, runs tests related to files changed since the last commit.
    `npm test`
  - **eject**: this command will copy all the configuration files and the transitive dependencies into our project as dependencies in package.json ;
    `npm run eject`

- **dependencies**: it contains the following packages needed for our app proper functioning:
  - **@testing-library/react**: simple and complete React DOM testing utilities that encourage good testing practices. ;
  - **@testing-library/user-event**: simulate user events for react-testing-library;
  - **bootstrap**: sleek, intuitive, and powerful front-end framework for faster and easier web development;
  - **chart.js**: simple yet flexible JavaScript charting for designers and developers;
  - **is-promise**: test whether an object looks like a promises-a+ promise;
  - **libsvm-js**: c++ library that allows to do Support Vector Machine classification and regression;
  - **ml-svm**: Support Vector Machine in Javascript;
  - **react**: JavaScript library for creating user interfaces. The react package contains only the functionality necessary to define React components;
  - **react-bootstrap**: bootstrap components built with React;
  - **react-chartjs-2**: React wrapper for Chart.js 2;
  - **react-csv-reader**: React component that handles csv file input. It handles file input and returns its content as a matrix;
  - **react-dom**: package that serves as the entry point to the DOM and server renderers for React. It is intended to be paired with the generic React package, which is shipped as react to npm;
  - **react-scripts**: this package includes scripts and configuration used by Create React App;
  - **svm**: is a lightweight implementation of the SMO algorithm to train a binary Support Vector Machine. As this uses the dual formulation, it also supports arbitrary kernels.
- **devDependencies**: it contains the following packages needed for our app proper functioning during development:
  - **@testing-library/jest-dom**: custom jest matchers to test the state of the DOM;
  - **csv-parser**: streaming CSV parser that aims for maximum speed as well as compatibility with the csv-spectrum CSV acid test suite. Csv-parser can convert CSV into JSON;
  - **eslint-plugin-react-hooks**: this ESLint plugin enforces the Rules of Hooks. It is a part of the Hooks API for React.

5 Test

A unity test is the operation that aims to check the correct execution of functions of a component. The run tests in this project, the developer has to open the terminal and to move inside the folder of the product that needs to be tested (in this case `PredireInGrafanaSW/my-plugin` or `PredireInGrafanaSW/prediction-tool`). Inside the folder, the developer has to run the command `npm run test` inside the terminal.

5.1 Edit tests

Inside the project, every components need to be tested. For this reason the developer has to name a test with **<component name>.test.ts**. Doing so a single component can be checked by a single file.

5.2 Code Coverage

Code coverage is the percentage of the project's code that is analysed by tests. This operation is made by Coverall which is an external tool linked to the code through the project's repository. So if the developer wants to check coverage's value, he/she has to reach the READ.me file inside the GitHub repository at `https://github.com/teamafkSWE/PredireInGrafana-SW`. Inside that file there are two badges and the one with the label "coverage" shows the value of code coverage until the last build. This operation can be done locally, by checking the READ.me in the local repository.

6 Technologies

6.1 ESLint

ESLint is a plugin used for code static analysis, so it checks for JavaScript syntax errors.

6.2 InfluxDB

InfluxDB it's an open source time series database which is used to store and manage data inside the Prediction Plugin.

6.3 Telegraf

Telegraf is a plugin-driven server agent for collecting and sending metrics and events from databases, systems, and IoT sensors.

6.4 NodeJS

NodeJS it's a runtime environment that execute JavaScript code outside a web browser. NodeJS it's useful for the tool's developing.

6.5 NPM

NPM or Node Package Manager it's used to manage dependencies and build automation. All its functionalities are available by the presence of the file package.json. In this project there are two copies of it because each product needs a dedicated package.json file. So tool's configuration and management are independent from those required for the plugin.

6.6 React

React it's a JavaScript library for building user interfaces for both plugin and tool.

6.7 JSX

JSX it's an extension of JavaScript language syntax. This extension helps to create HTML components for the Training Tool.

6.8 TypeScript

TypeScript it's a JavaScript superset that provides static typing to the language. Since Grafana is actually based on this language, TypeScript is used to develop the plugin.

6.9 JSON

JSON it's an open standard file format used to store the training data obtained from the Training Tool. So JSON files allows to pass data between tool and plugin.

6.10 CSV

CSV it's a file format used to store data for tool's training tests.

6.11 Travis CI

Travis i is an hosted continuous integration service used to build and test software hosted on GitHub.

6.12 Linear Regression

The Linear regression it's a linear approach to modelling the relationship between dependent and independent variable through points setted on a plan.

6.13 Support Vector Machine

A Support Vector Machine it's a supervised learning model used in machine learning. With a SVM input data that are analysed and classified in two different classes.

7 Product architecture

Our product consists of a plug-in developed for the Grafana platform and a Training Tool external to this platform. Therefore the analysis of the architecture is divided into these two components.

7.1 Training tool

The Training tool deals with training an SVM or LR algorithm using a dataset inserted by the user, to then generate a JSON file containing the necessary information to perform the prediction. This module has been developed following the *MVVM* behavioural design pattern.

7.1.1 Architectural design

We decided to implement this pattern because React$_G$ was used to build the component and we believed that this pattern coupled well with the structure of the latter. Moreover, it allows to divide the *Presentation Logic*$_G$ and the *Business Logic*$_G$ and it allows to reuse some components in other contexts, without having to change them.

As can be seen in the following figure, we have the View that exchanges information on user interactions with the ViewModel, which in turn transforms them into actions on the data performed by the Model. The data transition from the View to the Model occurs through the instantiation of the ViewModel inside the View. Through this instance, the ViewModel calls the correct functions when the user interacts with the View. The Model provides functionality for the management of the algorithms through the `SVMtrain` and `RLtrain` classes that will be used by the ViewModel. Finally there is a communication between Model and View for the constant updating of the latter, thanks to a functionality provided by the Grafana platform.

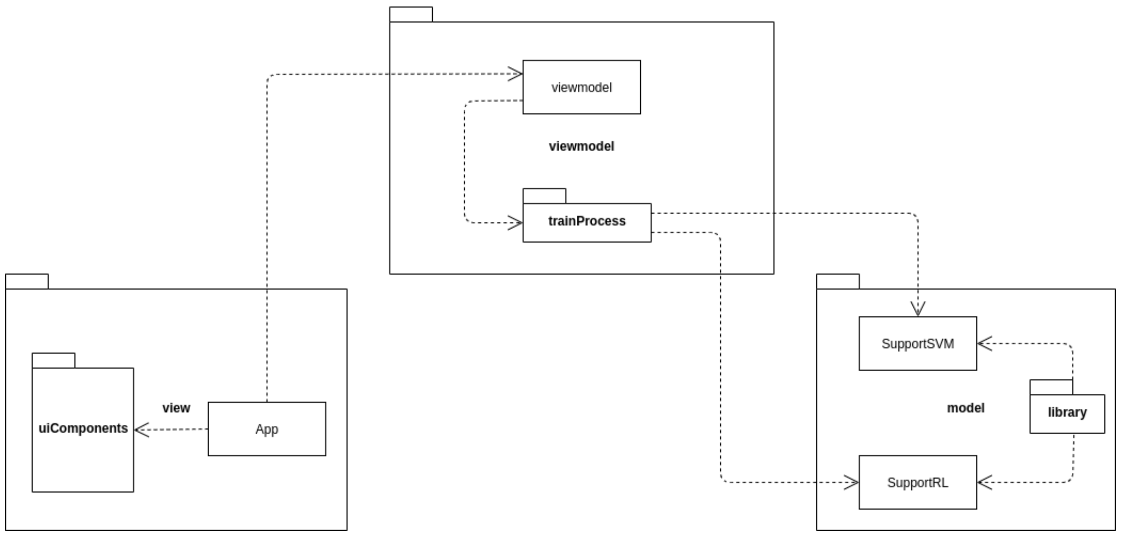


Figure 7.1.1: Training tool package diagram

Analyzing the specific components, our architecture is structured as follows:

- **Model**: it manages the *business logic*, in detail everything concerned the algorithms (such as libraries and their own methods);
- **View**: it manages the *presentation logic* through React, specifically what the user will see;
- **ViewModel**: it manages the *application logic$_G$* . The ViewModel is an abstraction of the View exposing public commands, and it has a binder, which automates communication between the View and its bound properties in the ViewModel.

7.1.2 Detailed design

7.1.2.1 Model

The main component of the Model is the abstract class of the prediction algorithms called Train. Starting with this, we have implemented two algorithms: Support Vector Machine and Linear Regression. They are represented respectively by the concrete classes `SupportSVM` and `SupportRL`. We have found that, for families algorithms such as Support Vector Machine and Linear Regression algorithms, it is possible to trace them to a single abstract class as we have provided.

SupportSVM

To implement the SVM algorithm we developed the concrete class `SupportSVM`. This class performs the prediction on the dataset. It also makes use of components specific to the SVM class imported from the SVM library to perform the real prediction. SupportSVM also has these fields data, each one instantiated by the constructor:

- `dataSVM`: reference to the dataset;
- `svm`: the SVM object;
- `weights`: the weights are the SVM coefficients that will be used to make the prediction.

The constructor also defines the SVM options, such as the kernel (linear) and the karpathy (true). This class includes the method to perform the SVM train (`trainSVM()`) and the method to set the JSON file SVM structure up (`JSONData()`).

SupportRL

To implement the Linear Regression algorithm we developed the concrete class `SupportRL`. This class performs the prediction on the dataset. It also makes use of components specific to the RL class imported from the RL library to perform the real prediction. SupportRL also contains these private fields data:

- `dataRL`: reference to the dataset, instantiated by the constructor;
- `numOfX`: the numbers of X founded on the CSV file;

- `reg`: the RL object, which has 2 parameters: the X numbers (`dataRl.length`) and Y numbers (1);
- `coefficients`: the coefficients needed to make the prediction, set as null by the constructor.

This class includes the method to perform the RL train (`trainRl()`) and the method to set the JSON file RL structure up (`JSONData()`).

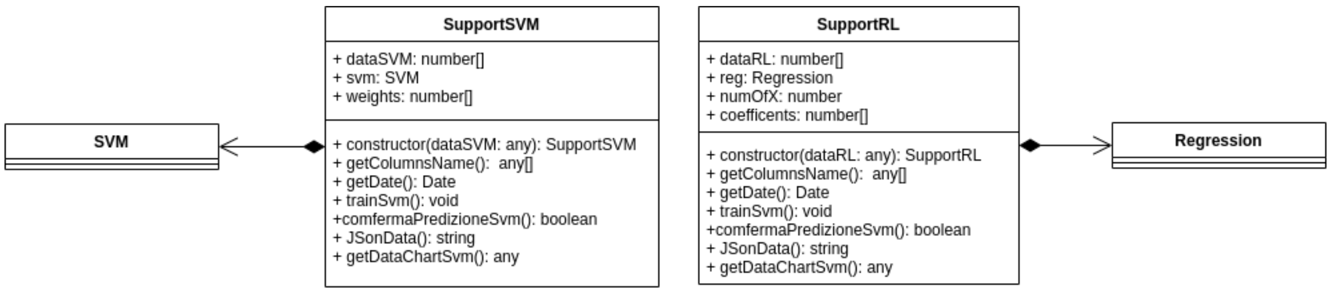


Figure 7.1.2: Training tool Model class diagram

7.1.2.2 View

Inside the View there are all the view components. The `App` class is the main component and it represents the tool's entry point. Each part of the View has been divided into components and rendered by the App. The communication between these components and the App takes place through the `props` (conceptually, the components are like JavaScript functions): it accepts arbitrary data (under the name of "props") and returns React elements that describe what should appear on the screen. Inside there is an instance of the ViewModel for data-binding.

App

The `App` class is the main component and it represents the tool's entry point. Each part of the View has been divided into components and rendered by the App. This class has just one parameter: `viewModel` is an instance of the ViewModel. The constructor contains the properties in common with the ViewModel and the `viewModel` instance. Moreover, it has the following methods:

- `changeAlgorithm(event)`: it manages the algorithm user's choice, setting the `algorithm` state value after the event was made;
- `resetAlgorithm(algorithm)`: it sets the `algorithm` state parameter to the default value;
- `setDataFromFile(data, fileInfo)`: it sets the `data` state parameter and the `fileName` state parameter with the inserted file's name;
- `handleTraining()`: it sends the data to the ViewModel and if the `performTraining()`

method success, it calls the function to write the information into the JSON file, otherwise it resets the algorithm;

- `downloadJsonData()`: it creates the DownloadJson component;
- `render()`: it creates the InsertCsvButton, ComboBoxAlgorithm, TrainButton and Chart components and it calls the `downloadJsonData()` function;
- `changeXAxis(event)`: it manages the xAxis user's choice to be displayed, setting the `xAixs` state value after the event was made;
- `selectAxisX()`: if a file has been imported, it allows to change the X axis to be displayed;
- `handleNotes(event)`: it manages the notes that have been written by the user, setting the `notes` state value after the event was made;
- `handleName(event)`: it manages the file name that has been written by the user, setting the `changeName` state value after the event was made;
- `show()`: it shows the "Information" page, which includes a step-by-step guide and the bugs report section;
- `setDefaultName()`: if the new file name hasn't been written on the specific textArea, this method sets the default name of that file to: *predictorsRL* if the CSV was a linear regression or *predictorsSVM* if it was a SVM file.

combo_box_algorithm

It renders the combo box for the algorithm's choice.
The `render()` method renders the JSX$_G$ element wanted.

download_json

It manages and render the "Download" button.
The `downloadJsonFile()` method allow to download the file, making the connection between the browser and the local pc. The `render()` method renders the "Download" button.

chart

It renders the chart. This component contains the following parameters:

- `data`: contains the data that must be shown;
- `options`: contains the various options that can be applied to the graph, such as colors, axes and the regression line.

It also implements the following methods:

- `formatData()`: it associates and manages the data to the graph;
- `render()`: it renders the chart graph.

insert_csv_button

It renders the insert button for the CSV file.
The `render()` method renders the "Select file" button.

header

It renders the header.
The `render()` method renders the site's header.

train_button

It renders the train button, that will start training.
The `render()` method renders the "Start training" button.

combo_box_x_axis

It renders the combo box the axis's choice. The `onChange` and `value` properties are `props` managed by the viewModel.

footer

It renders the footer of the page.

information

It renders the information page. It includes a brief guide for the tool and the email address to which bug reports should be sent.

textArea

It renders the textArea, which notes must be added to the JSON file produced.

textAreaFileName

It renders the textArea above the notes textArea, which information must be set as the JSON file name.

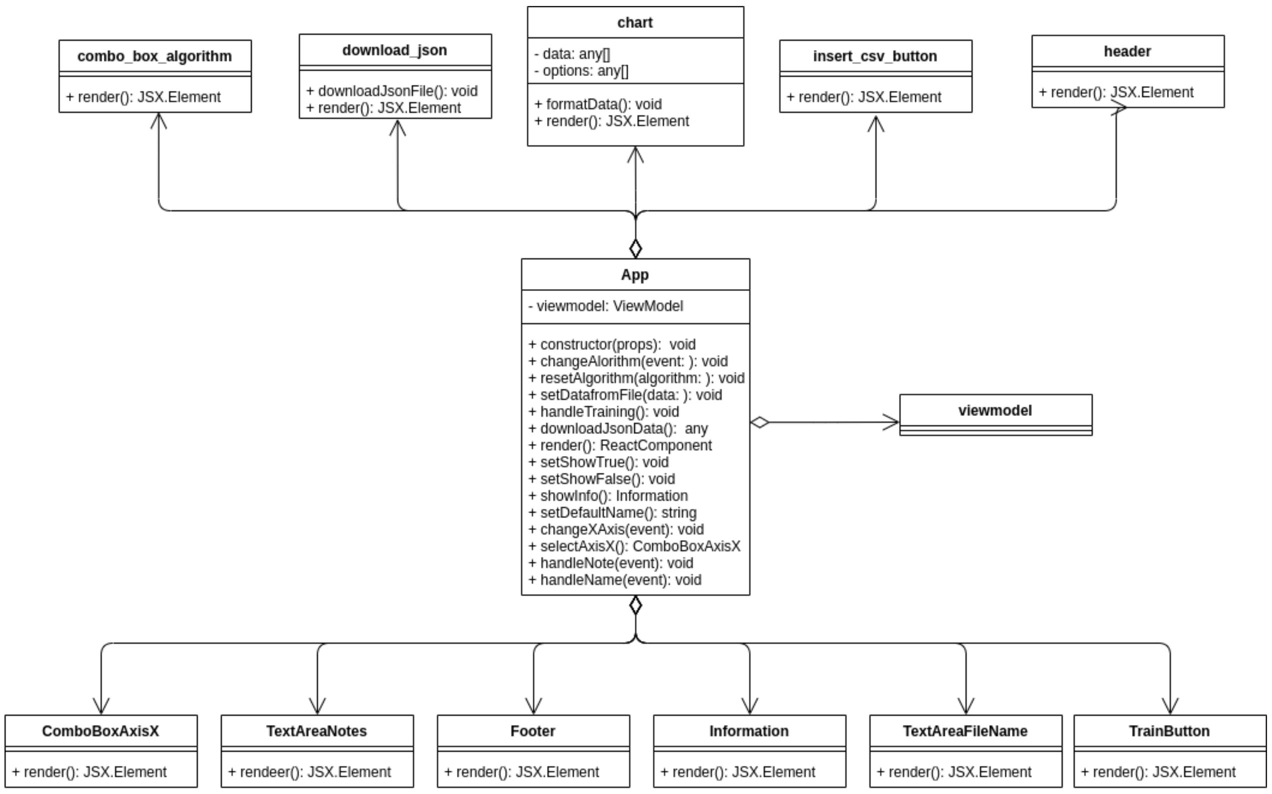


Figure 7.1.3: Training tool Model class diagram

7.1.2.3 ViewModel

The data transition from the views to the model occurs through an instance of the ViewModel inside the View. Through this instance, the ViewModel calls the correct functions when the user interacts with the View. The ViewModel has functions that interact with both the View and the Model (e.g. `RLChart()` and `performTraining()`). To communicate with the Model, there is an `RLTrain` or `SVMTrain` instance. In order to instantiate the correct algorithm, the behavioural design pattern Strategy is applied.
The ViewModel contains the following private variables:

- `algorithm`: it contains the string of the chosen algorithm (combo-box component);
- `file`: it contains the data of the inserted CSV;
- `hasFile`: true if the file CSV has been inserted, false otherwise;
- `STrain`: it contains the strategy object;
- `xAxis`: it contains the X axis of the CSV file;
- `indexOfMax`: max index for the regression line to be displayed;
- `indexOfMin`: min index for the regression line to be displayed;
- `maxXAxis`: max X axis value;
- `minXAxis`: min X axis value;

- `notes`: it contains the notes that has been written on the textArea.

The `constructor()` set these variables to `null`. Each variable has its own `set«Variable»()` function, that sets `this.«Variable»` to that variable.
The `checkAlgorithm()` method perform controls to the uploaded file:

- if it is a SVM CSV file, the `verifyAlgorithm` calls the `isSVM` method;
- if it is a RL CSV file, the `verifyAlgorithm` calls the `isRL` method;
- if the CSV file does not match any prediction algorithm structure, the function returns an alert;
- if any algorithm hasn't been chosen and we are trying to start training, the function returns an alert;
- if the algorithm has been chosen but any file hasn't been inserted, the function returns an alert.

There are also the chart methods, that handle the datas to be displayed:

- `straightLine()`: it uses the axis variables (xAxis, indexOfMax, indexOfMin, maxXAxis and minXAxis) to display the regression line correctly in the chart;
- `chart()`: it sets the chart with the correct informations obtained by the uploaded file, in detail both X and Y axis;
- `chartAxisX(label, dataX, dataY)`: it sets the color of the datas to be displayed: red if the data is "bad" (so miscalculated by the prediction algorithm) and green if the data is "good" (so correctly calculated);
- `getPredictorsName()`: it returns the CSV columns names.

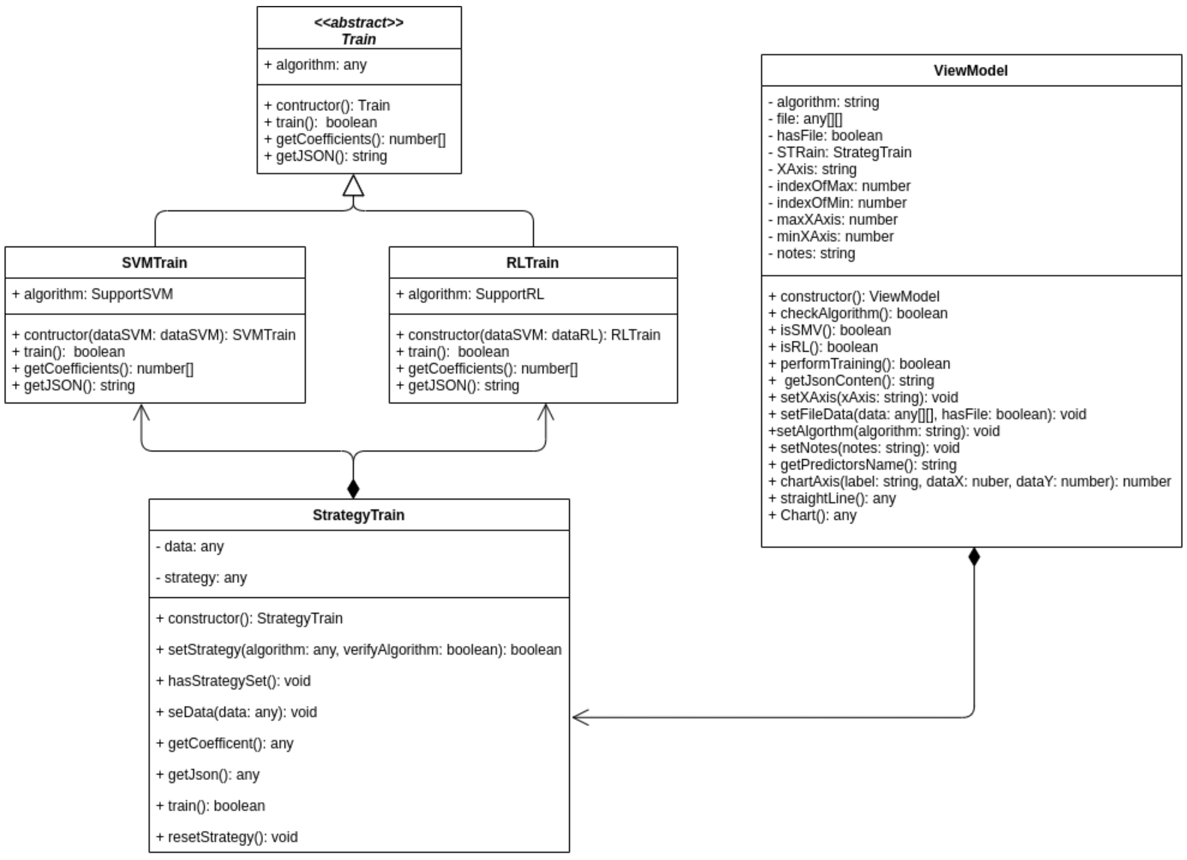

<<abstract>>
Train
+ algorithm: any
+ contructor(): Train
+ train(): boolean
+ getCoefficients(): number[]
+ getJSON(): string
SVMTrain
+ algorithm: SupportSVM
+ contructor(dataSVM: dataSVM): SVMTrain
+ train(): boolean
+ getCoefficients(): number[]
+ getJSON(): string
RLTrain
+ algorithm: SupportRL
+ constructor(dataSVM: dataRL): RLTrain
+ train(): boolean
+ getCoefficients(): number[]
+ getJSON(): string
StrategyTrain
- data: any
- strategy: any
+ constructor(): StrategyTrain
+ setStrategy(algorithm: any, verifyAlgorithm: boolean): boolean
+ hasStrategySet(): void
+ seData(data: any): void
+ getCoefficent(): any
+ getJson(): any
+ train(): boolean
+ resetStrategy(): void
ViewModel
- algorithm: string
- file: any[][]
- hasFile: boolean
- STRain: StrategTrain
- XAxis: string
- indexOfMax: number
- indexOfMin: number
- maxXAxis: number
- minXAxis: number
- notes: string
+ constructor(): ViewModel
+ checkAlgorithm(): boolean
+ isSMV(): boolean
+ isRL(): boolean
+ performTraining(): boolean
+ getJsonConten(): string
+ setXAxis(xAxis: string): void
+ setFileData(data: any[][], hasFile: boolean): void
+setAlgorthm(algorithm: string): void
+ setNotes(notes: string): void
+ getPredictorsName(): string
+ chartAxis(label: string, dataX: nuber, dataY: number): number
+ straightLine(): any
+ Chart(): any

Figure 7.1.4: Training tool Model class diagram

As it can be seen in this diagram, the ViewModel has a child class `StrategyTrain`, which is an implementation of the Strategy design pattern, that instantiate the correct algorithm whenever it's needed. Both `SVMTrain` and `RLTrain` classes are children of the *abstract* `Train` class, so adding new algorithms in the future is simplified.

SVMTrain

This class contains the methods to perform the SVM train, such as:

- `train()`: it calls `algorithm.trainSvm()` to perform the train and it returns a boolean if the train has been done correctly (true) or it failed (false);
- `getCoefficients()`: it gets the `algorithm.Weights()`;
- `getJSON()`: it gets the `algorithm.JSONData()`.

The constructor instantiate the private variable `algorithm` to a `new SupportSvm(dataSVM)` object.

RLTrain

This class contains the methods to perform the RL train, such as:

- `train()`: it calls `algorithm.insert()` to push the data and `algorithm.trainRl()` to perform the train and it returns a boolean if the train has been done correctly (true) or it failed (false);
- `getCoefficients()`: it gets the `algorithm.getCoefficientsRL()`;
- `getJSON()`: it gets the `algorithm.JSONData()`.

The constructor instantiate the private variable `algorithm` to a `new SupportRl(dataRl)` object.

StrategyTrain

This class is the implementation of the Strategy design pattern, so it sets the strategy according to the CSV file inserted and the user's choice.
The constructor instantiates the `strategy` variable and the `data` variable to `null`. Therefore, the assignment of the strategy algorithm is deferred to method `setStrategy(algorithm, verifyAlgorithm`, which returns true if the strategy was set correctly or false if not. There are also the following methods:

- `train()`: it calls the `train()` method of the specific strategy algorithm or it returns false;
- `getJson()`: it calls the `getJson()` method of the specific strategy algorithm or it returns null;
- `getCoeff()` it calls the `getCoefficients()` method of the specific strategy algorithm;
- `resetStrategy()`: it sets the `strategy` variable to `null`;
- `hasStrategySet()`: it return true if the `strategy` is not null, false if not;
- `setData()`: it sets the private variable `data` to the specific `data`, which will be used for the prediction.

abstractTrain

The base class, useful to add new prediction algorithms in the future.
It represents the contract that all concrete classes must abide to be able to characterize a prediction algorithm. It contains only one parameter: `algorithm`, set to `null`.

Sequence diagram

To better explain the set of actions performed in order to process the data for predict algorithms, the following picture illustrate a sequence diagram which examines SVM. The procedure is also indicative for the others algorithms.

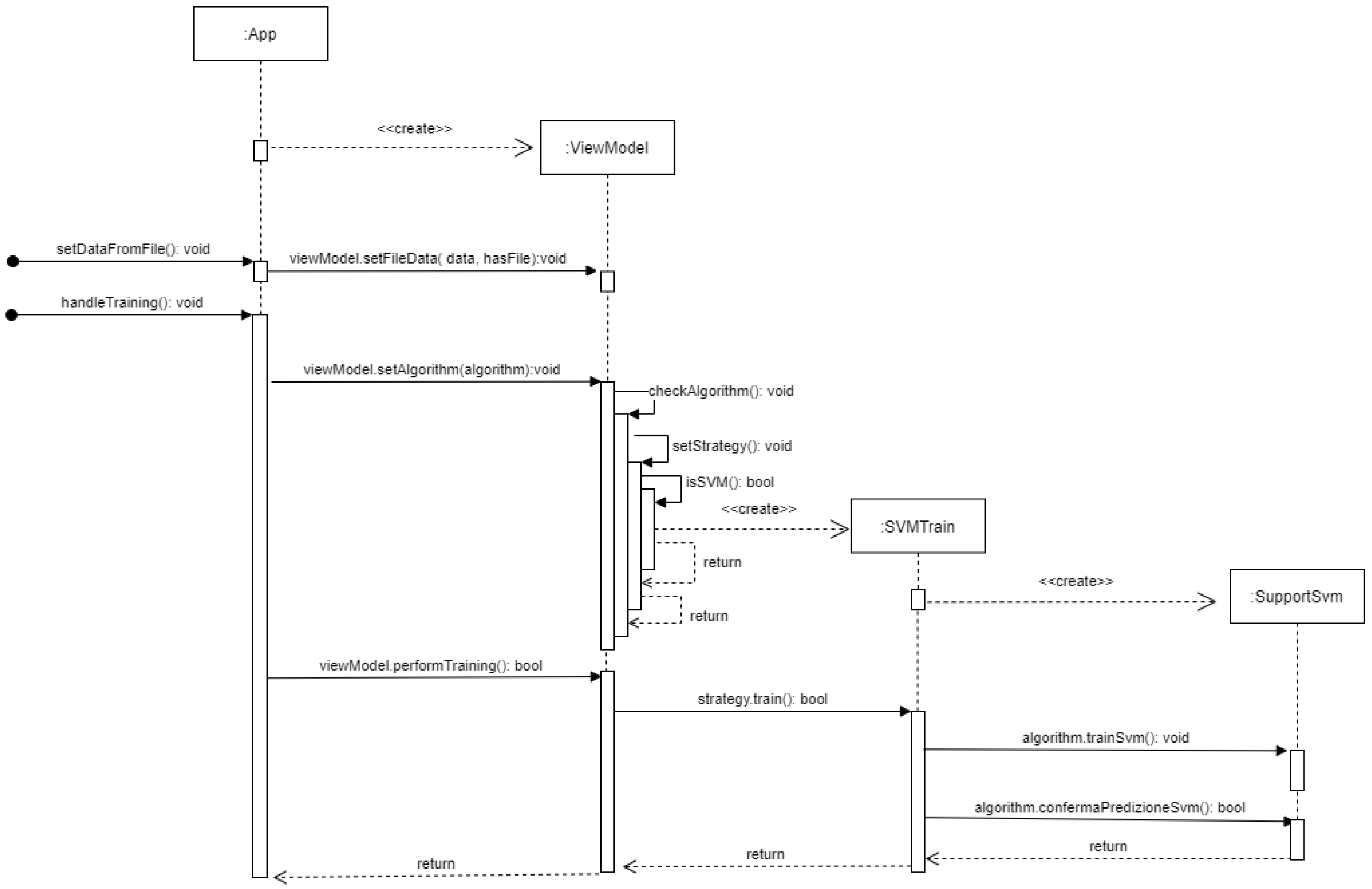


Figure 7.1.5: TrainSVM sequence diagram

7.2 Prediction plug-in

The Prediction Plug-in will take care of receiving the json input and once the predictors have been connected to a data flow, will allow you to start making calculations forecasting. This module has been developed following the *MVC* design pattern.

7.2.1 Architectural design

We felt that this model coupled well with the Grafana plug-in structure. Moreover, it allows to divide the *presentation logic* and the *business logic* and it allows to reuse some components in other contexts, without having to change them (e.g. the View). In this module, consisting of several panels within Grafana, the predictor, produced by the external tool, will be associated with the data flow monitored in Grafana. For this module the *MVC* architectural pattern was used. In this architecture the `Editor` and the `Panel` share the variable `props`, instantiated by Grafana.

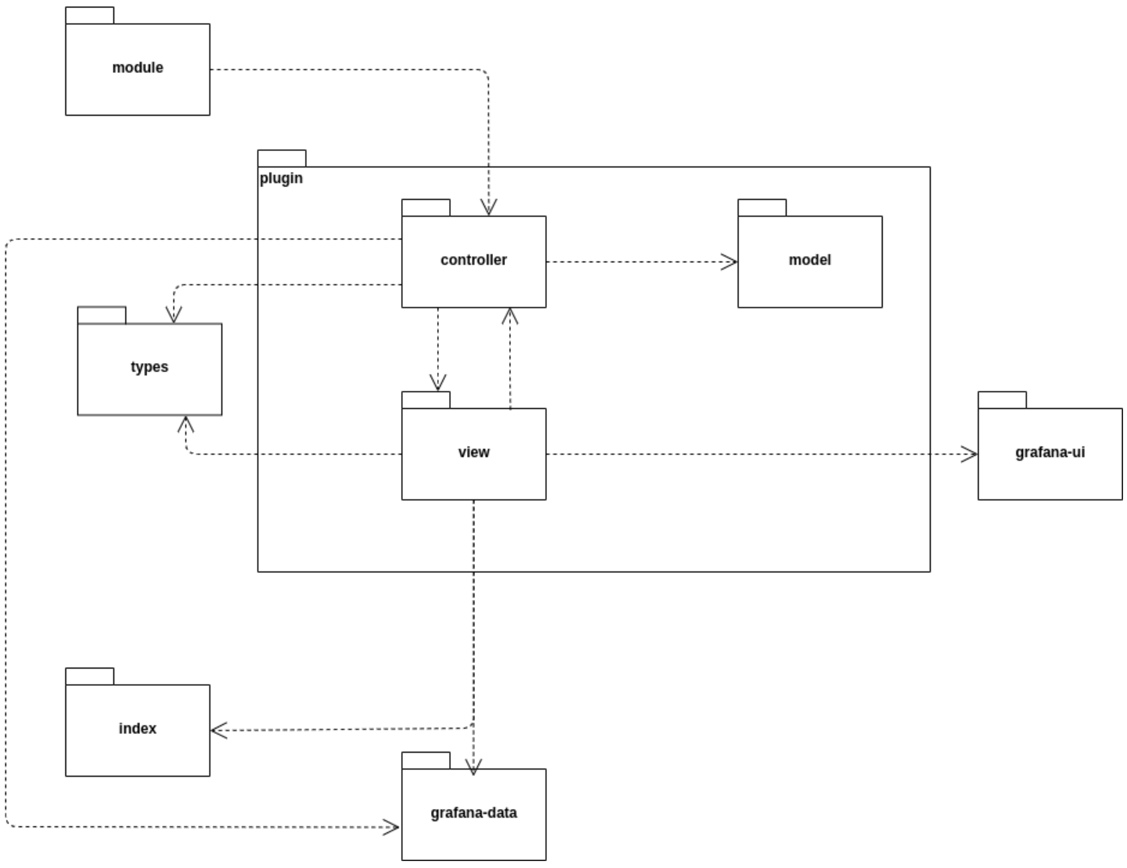


Figure 7.2.1: Prediction plug-in package diagram

Analyzing the specific components, our architecture is structured as follows:

- **Model**: it manages the *business logic*. It contains the data prediction algorithms that have been presently implemented and writing the result of the predictions to an Influx database;

- **View**: it manages the *presentation logic*. It allows the creation of a customized graphic panel within a Grafana dashboard. With this panel the user can select prediction algorithm settings, incoming data streams and the minimum and maximum thresholds;
- **Controller**: it manages the *application logic*. It transforms the data obtained from user interactions and data flows obtained by Grafana in a format suitable for carrying out the actions by the Model.

7.2.1.1 Grafana's role

Grafana plays a fundamental role in the architecture as it allows continuous updating of the View to change the data in the Model. In particular, whenever there are updates on the data in the Model, Grafana makes them available to View through specific methods, together with the configuration of the queries with which this data was obtained. Moreover, through React, it manages the two-way data binding between HTML View and Javascript code, in addition to offering most of the necessary graphic components for the realization of the plug-in View.

7.2.2 Detailed design

7.2.2.1 Model

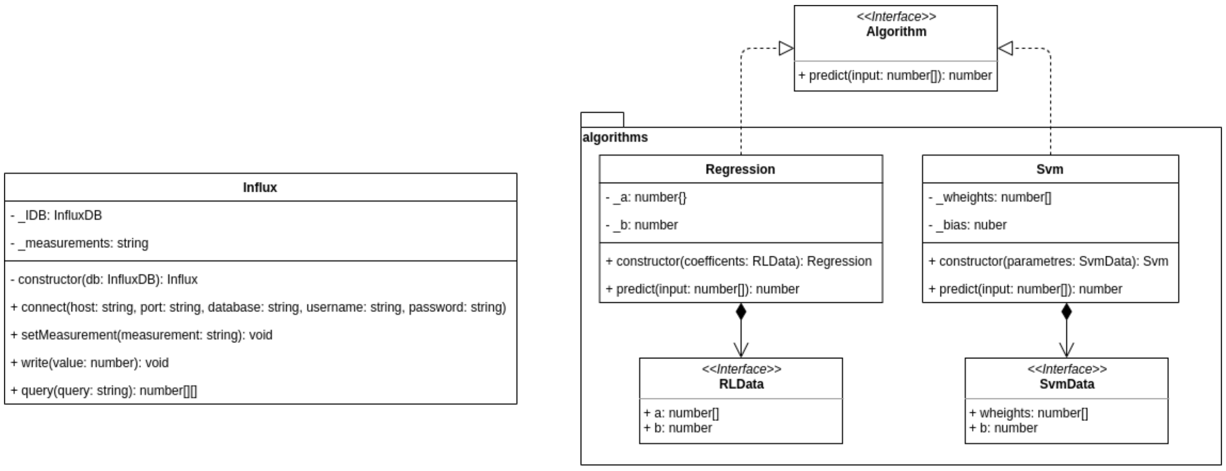
The Model is the implementation of the *Strategy* pattern. Unlike the tool, in this case the pattern is used to implement the forecast calculation algorithms. The algorithms are instantiated by the Controller when a JSON file containing the definition of the algorithm to be used is read, therefore extrapolating the values to be passed to the constructors always from the file just inserted.
The Model's main component is the interface of the prediction algorithms called `Algorithm`. Starting with this, we have implemented two algorithms: Support Vector Machine and Linear Regression. They are represented respectively by the concrete classes Svm and Regression. We have found that, for families of Support Vector Machine and Regression algorithms, it is possible to trace them to a single interface as we have provided.

Algorithm

The interface of the prediction algorithms is called *Algorithm* and it represents the contract that all concrete classes must abide by to be able to characterize a prediction algorithm. It contains only one method: `predict(input)`. This allows you to perform the prediction on a specific dataset of type `number`.

Influx

This class is used to interface and write data to the Influx database WriteInflux. `InfluxDB` contains an instance of the Influx DB and the `write()` method writes the data to the Influx database. The async method `connect(host, port, database, username?, password?)` it's used to connect the database to the plug-in. Any other "connection-handling method", `setMeasurement` and `setDatasource` methods are handled by the *controller*.



<<Interface>>
Algorithm
+ predict(input: number[]): number
algorithms
Regression
- _a: number{}
- _b: number
+ constructor(coefficents: RLData): Regression
+ predict(input: number[]): number
Svm
- _wheights: number[]
- _bias: nuber
+ constructor(parametres: SvmData): Svm
+ predict(input: number[]): number
<<Interface>>
RLData
+ a: number[]
+ b: number
<<Interface>>
SvmData
+ wheights: number[]
+ b: number
Influx
- _IDB: InfluxDB
- _measurements: string
- constructor(db: InfluxDB): Influx
+ connect(host: string, port: string, database: string, username: string, password: string)
+ setMeasurement(measurement: string): void
+ write(value: number): void
+ query(query: string): number[][]

Figure 7.2.2: Prediction plug-in Model class diagram

7.2.2.2 View

The main node of the View is the `Editor` class, which is the composition of different graphic objects such as CollegamentoView, PrevisioneView, ListaCollegamentiView and CaricamentoJsonView. The latter correspond to the panels and internal objects that will be shown by the plug-in.
All these classes are subclasses that extend the React class `React.PureComponent` which allows you to create components in React:

- `CaricamentoJsonView`: it is a view that takes care of loading the JSON file containing the definition of the algorithm previously trained by the training tool;
- `CollegamentoView`: this view allows the user to insert a new link, connect a data flow node to the latter and set the thresholds;
- `ListaCollegamentiView`: this view lets the user to modify or delete the previous links created;
- `PrevisioneView`: this view allows the user to start or stop the monitoring and save the prediction on the Influx database.

The *Observer* pattern was used to simplify communication between the Controller and the View components that need to be updated as a result of changing the state of the Controller. In our case the subject class, when it undergoes a status change, it notifies the other components via the `update()` method.

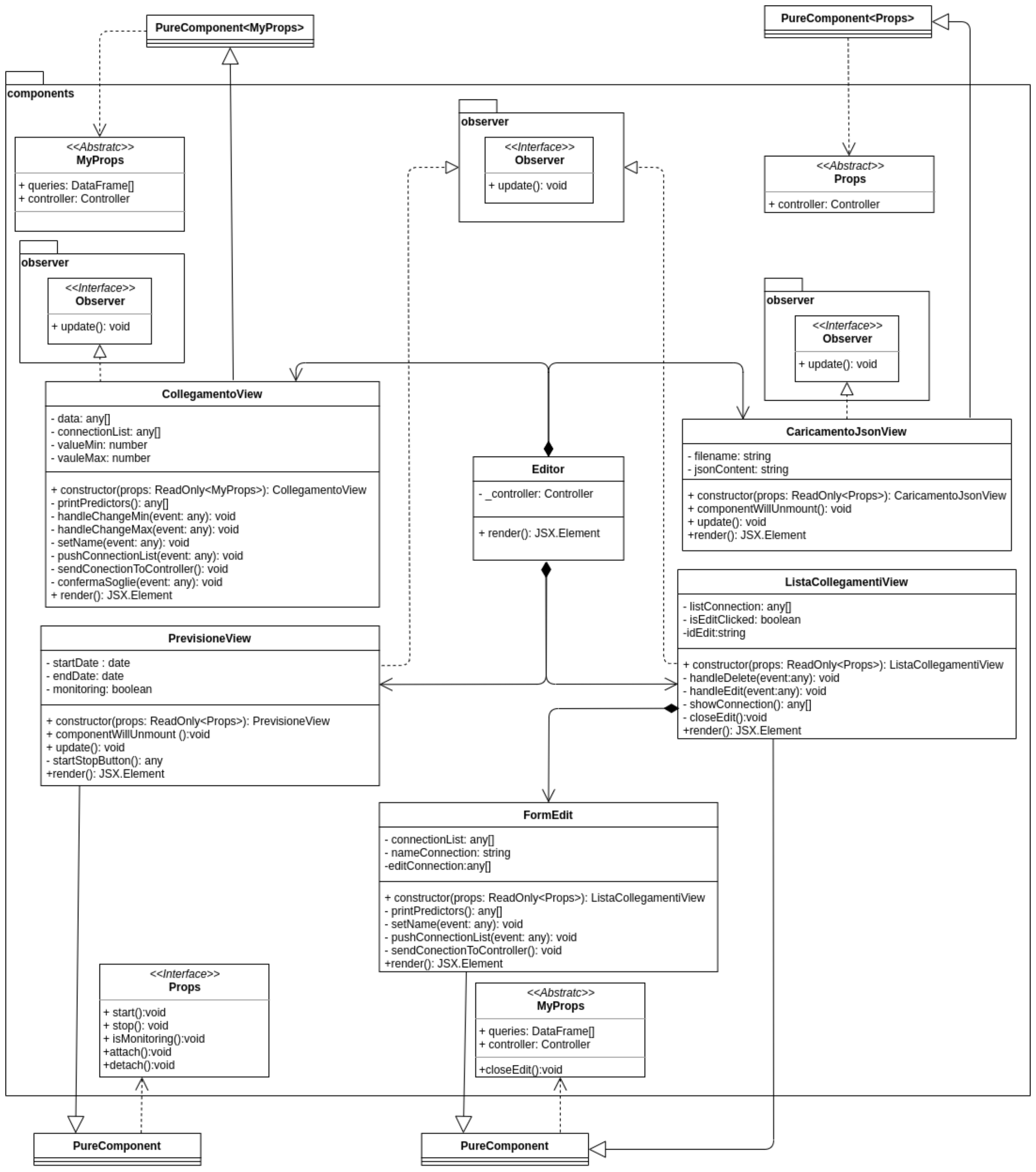


Figure 7.2.3: Prediction plug-in View class diagram

7.2.2.3 Controller

The controller implements the *Observer* pattern; in this case the controller class extends the abstract observable class which therefore represents the subject to be observed, the observer will instead be the view classes. The Controller will notify the views whenever it changes its state, particularly when the JSON file is inserted and read and when the monitoring is started or stopped.
When importing the JSON file, the controller will take care of:

- read its contents through the `setJson(file)` function. Once the Controller finishes reading the file, it notifies the View via the `update()` method, showing the new information obtained from the file;
- save a copy of the predictors;
- apply the correct forecasting algorithm through the `setStrategy()` method;
- notify the JSON file upload success views.

Moreover, it manages the database functions, which allow to add, edit or remove a connection, set new measurements, update and set datasources and save predictions on the Influx database.

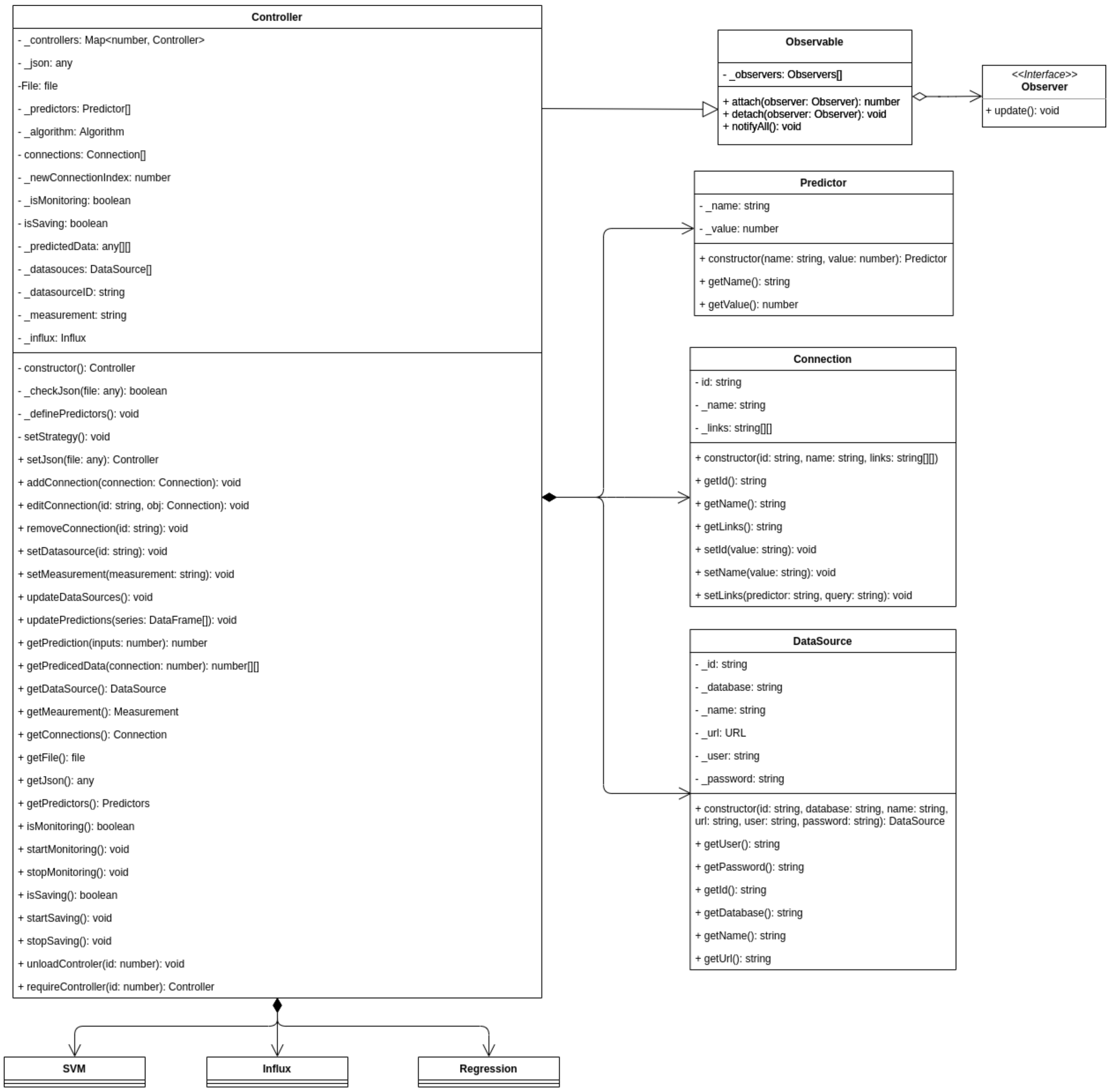

Controller
- _controllers: Map<number, Controller>
- _json: any
-File: file
- _predictors: Predictor[]
- _algorithm: Algorithm
- connections: Connection[]
- _newConnectionIndex: number
- _isMonitoring: boolean
- isSaving: boolean
- _predictedData: any[][]
- _datasouces: DataSource[]
- _datasourceID: string
- _measurement: string
- _influx: Influx
- constructor(): Controller
- _checkJson(file: any): boolean
- _definePredictors(): void
- setStrategy(): void
+ setJson(file: any): Controller
+ addConnection(connection: Connection): void
+ editConnection(id: string, obj: Connection): void
+ removeConnection(id: string): void
+ setDatasource(id: string): void
+ setMeasurement(measurement: string): void
+ updateDataSources(): void
+ updatePredictions(series: DataFrame[]): void
+ getPrediction(inputs: number): number
+ getPredicedData(connection: number): number[][]
+ getDataSource(): DataSource
+ getMeaurement(): Measurement
+ getConnections(): Connection
+ getFile(): file
+ getJson(): any
+ getPredictors(): Predictors
+ isMonitoring(): boolean
+ startMonitoring(): void
+ stopMonitoring(): void
+ isSaving(): boolean
+ startSaving(): void
+ stopSaving(): void
+ unloadControler(id: number): void
+ requireController(id: number): Controller
Observable
- _observers: Observers[]
+ attach(observer: Observer): number
+ detach(observer: Observer): void
+ notifyAll(): void
<<Interface>>
Observer
+ update(): void
Predictor
- _name: string
- _value: number
+ constructor(name: string, value: number): Predictor
+ getName(): string
+ getValue(): number
Connection
- id: string
- _name: string
- _links: string[][]
+ constructor(id: string, name: string, links: string[][])
+ getId(): string
+ getName(): string
+ getLinks(): string
+ setId(value: string): void
+ setName(value: string): void
+ setLinks(predictor: string, query: string): void
DataSource
- _id: string
- _database: string
- _name: string
- _url: URL
- _user: string
- _password: string
+ constructor(id: string, database: string, name: string, url: string, user: string, password: string): DataSource
+ getUser(): string
+ getPassword(): string
+ getId(): string
+ getDatabase(): string
+ getName(): string
+ getUrl(): string
SVM
Influx
Regression

Figure 7.2.4: Prediction plug-in Controller class diagram

Sequence diagram

The following sequence diagram describes the `viewGraph()` method, that process the data and updates the Grafana's graph with the correct algorithm prediction.

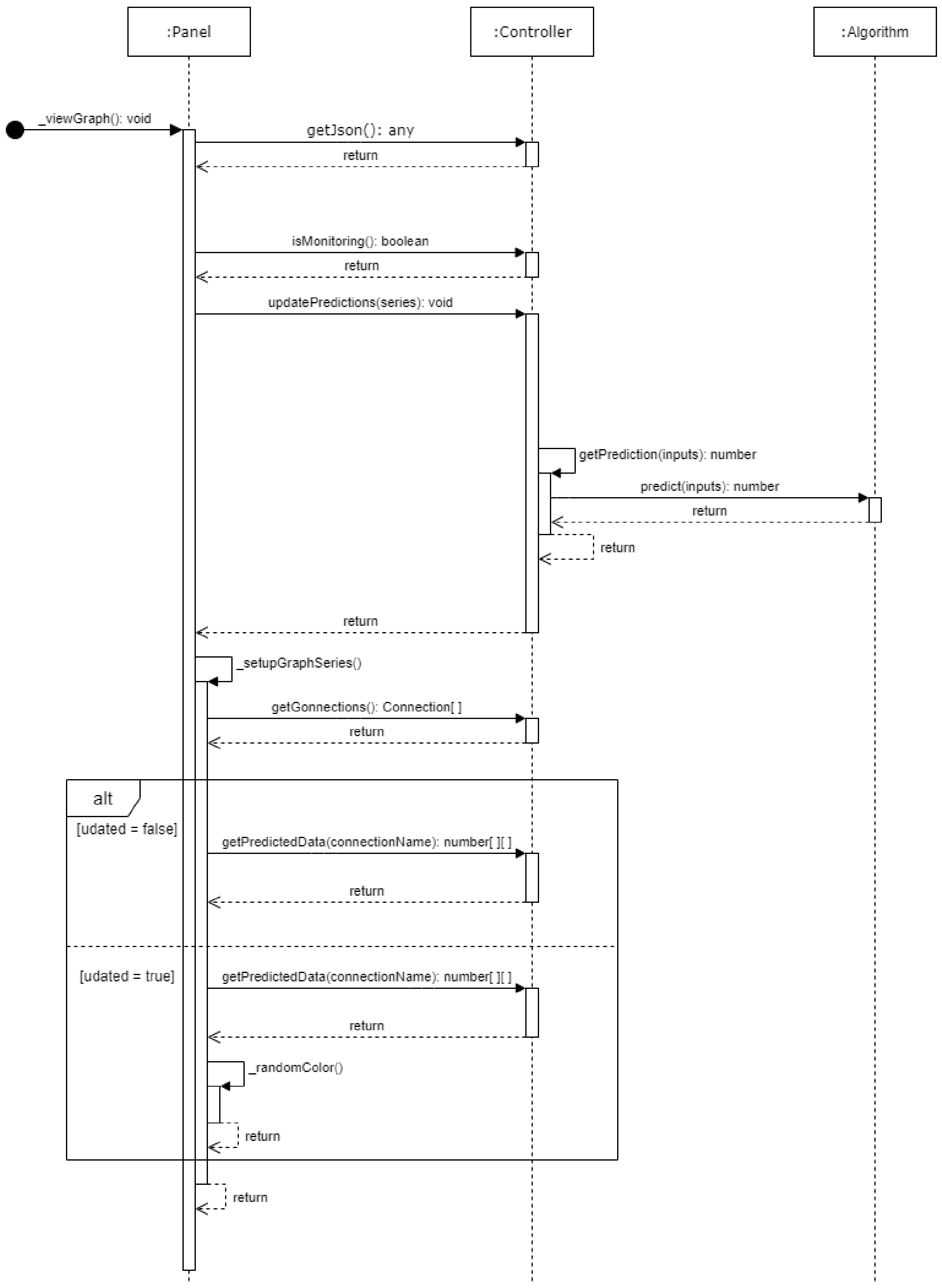


Figure 7.2.5: `viewGraph()` sequence diagram

8 Product extensibility

8.1 Integration of new prediction algorithm

With this release the product performs trainings and predictions through Support Vector Machine or Linear Regression. But the product can support new procedures because its architecture allows to extend the list of algorithm available by following some steps.
Inside Training Tool:

- in *model*'s *library* package, the developer has to implement the effective algorithm;
- in *model*'s *train* package the developer has to implement a support class;
- the developer has to extend the abstract class *Train* located in *model* package, and has to implement the all abstract methods. Then the developer has to import the class inside the *viewmodel*.

Inside Prediction Plugin:

- inside the *model* package, the developer has to extend the class *algorithm*, which is the base for the prediction operation.

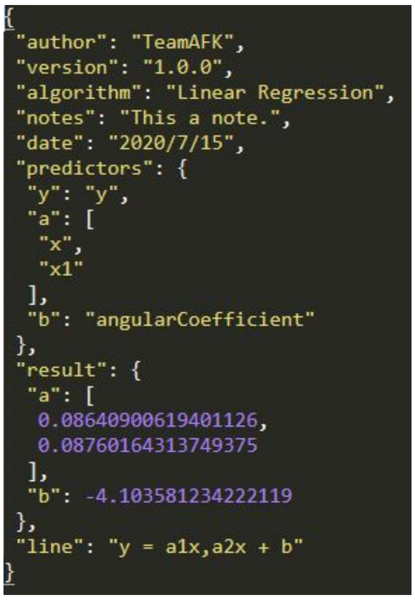
9 File structure

9.1 JSON file structure

JSON files definition, which contain trained algorithm, must be structured as it follow.

9.1.1 Linear Regression

- **author**: the file's author. Set in default as "TeamAFK";
- **version**: the training tool application version. Set in default as "1.0.0";
- **algorithm**: the trained algorithm . Set in default as "Linear Regression";
- **notes**: additional notes added to the file;
- **date**: the date when the file was created;
- **predictors**: list of all predictors' labels which are taken from CSV files;
- **result**: list of obtained coefficients from the training. In detail:
  - **a**: list of coefficients values of the regression line formula;
  - **b**: bias of the regression line formula;
- **line**: regression line formula.



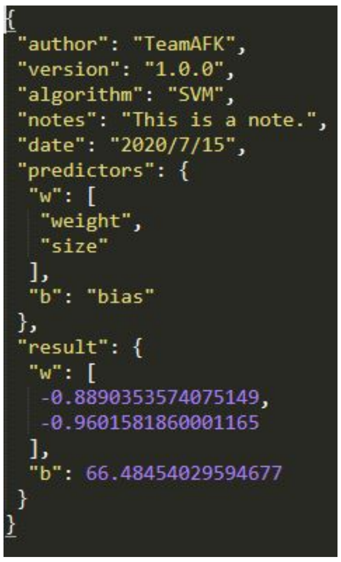
```
{
 "author": "TeamAFK",
 "version": "1.0.0",
 "algorithm": "Linear Regression",
 "notes": "This a note.",
 "date": "2020/7/15",
 "predictors": {
  "y": "y",
  "a": [
   "x",
   "x1"
  ],
  "b": "angularCoefficient"
 },
 "result": {
  "a": [
   0.08640900619401126,
   0.08760164313749375
  ],
  "b": -4.103581234222119
 },
 "line": "y = a1x,a2x + b"
}
```

Figure 9.1.1: Example of Linear Regression JSON file

9.1.2 Support Vector Machine

- **author**: the file's author. Set in default as "TeamAFK";
- **version**: the training tool application version. Set in default as "1.0.0";

- **algorithm**: the trained algorithm . Set in default as "SVM";
- **notes**: additional notes added to the file;
- **date**: the date when the file was created;
- **predictors**: list of all predictors' labels which are taken from CSV files;
- **result**: list of obtained coefficients from the training. In detail:
  - **a**: list of weights values of the regression line formula;
  - **b**: bias of the regression line formula.



```
{
 "author": "TeamAFK",
 "version": "1.0.0",
 "algorithm": "SVM",
 "notes": "This is a note.",
 "date": "2020/7/15",
 "predictors": {
  "w": [
   "weight",
   "size"
  ],
  "b": "bias"
 },
 "result": {
  "w": [
   -0.8890353574075149,
   -0.9601581860001165
  ],
  "b": 66.48454029594677
 }
}
```

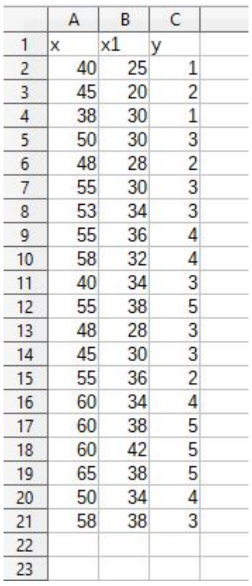
Figure 9.1.2: Example of Support Vector Machine JSON file

9.2 CSV file structure

Whether you choose to train one algorithm or another, CSV file has a common structure:

- **first columns till y column**: each column represents a predictor;
- **y**: this column contains the results.

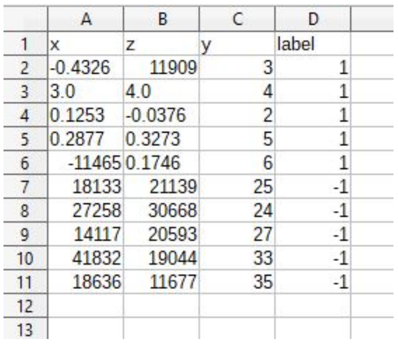
First line of CSV files must contain predictors tags while the other lines contain values. If you want to add a predictor you must insert a column, before y column, with the predictor tag as first line.



	A	B	C
1	x	x1	y
2	40	25	1
3	45	20	2
4	38	30	1
5	50	30	3
6	48	28	2
7	55	30	3
8	53	34	3
9	55	36	4
10	58	32	4
11	40	34	3
12	55	38	5
13	48	28	3
14	45	30	3
15	55	36	2
16	60	34	4
17	60	38	5
18	60	42	5
19	65	38	5
20	50	34	4
21	58	38	3
22			
23			

Figure 9.2.1: Example of Linear Regression CSV file

In SVM CSV files you can find one more column named "label" which contains the data group.



	A	B	C	D
1	x	z	y	label
2	-0.4326	11909	3	1
3	3.0	4.0	4	1
4	0.1253	-0.0376	2	1
5	0.2877	0.3273	5	1
6	-11465	0.1746	6	1
7	18133	21139	25	-1
8	27258	30668	24	-1
9	14117	20593	27	-1
10	41832	19044	33	-1
11	18636	11677	35	-1
12				
13				

Figure 9.2.2: Example of Support Vector Machine CSV file

A Glossary

A

Application Logic
Application Logic is any logic that is specific to some application.

B

Business Logic
It is the processing logic, in the form of source code, which manages the communication between a user interface and a database. It therefore contains the information that defines or they bind the way a company operates.

C

CSV
CSV (comma-separated values) is a simple file format used to store tabular data, such as a spreadsheet or database.

D

DOM
The Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document.

L

Linear Regression (LR)
Linear regression (LR) is a linear approach to modeling the relationship between a scalar response (or dependent variable) and one or more explanatory variables (or independent variables). The case of one explanatory variable is called simple linear regression.

G

Grafana
Grafana is a multi-platform open source analytics and interactive visualization web application. It provides charts, graphs, and alerts for the web when connected to supported data sources. It is expandable through a plug-in system. End users can create complex monitoring dashboards using interactive query builders.

GUI
A graphical user interface (GUI) is an interface through which a user interacts with electronic devices such as computers, hand-held devices and other appliances.

J

JavaScript
JS is a programming language that conforms to the ECMAScript specification. JavaScript is high-level, often just-in-time compiled, and multi-paradigm.

JSON
JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for machines to parse and generate.

JSX
JSX is an extension of the JavaScript language based on ES6, and is translated into regular JavaScript at runtime.

M

Machine learning
Machine learning (ML) is the study of computer algorithms that improve automatically through experience. Machine learning algorithms, such as LR or SVM, build a mathematical model based on sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to do so.

N

Node.js
Node.js is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside a web browser. Node.js lets developers use JavaScript to write command line tools and for server-side scripts to produce dynamic web page content before the page is sent to the user's web browser.

NPM
NPM (Node Package Manager) is a package manager for the JavaScript programming language. It is the default package manager for the JavaScript runtime environment Node.js.

O

OS
An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices.

P

Package
A package (java package) is used to group related classes. They are used to avoid name conflicts, and to write a better maintainable code.

Presentation Logic
The presentation logic of an application is where occurs almost of all interactions with the end user. Manages the reception of user inputs and the presentation of the output of the application to the latter.

R

React
React is a JavaScript library for building user interfaces.

S

Support Vector Machine (SVM)
In machine learning, Support Vector Machines (SVM) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis.

T

Typescript
TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.