UNIVERSITY OF FLORENCE

DEPARTMENT OF INFORMATION ENGINEERING


Degree Course in Computer Engineering

# Design and Development of a Finalized Component

Teacher:
Ch.mo Prof.
Enrico Vicario

Candidate:
Hamidreza Hashemi
Mat. 5996260

Course:
Software engineering

ACADEMIC YEAR 2020/2021

*Dedicated to those*
*who try every day.*

# INDEX

# INTRODUCTION

In this exercise we will introduce an industrial level need and solve it with the techniques introduced in the Software Engineering course. The related fields are:

- Machine Learning
- Statistics
- Industrial Informatics
- IOT (*internet of things)*

It would be helpful to have some knowledge about the Niagara NX platform [1]

[1] Niagara NX platform is a platform developed by Tridium (*a company under Honeywell*) for IOT management. This platform allows you to program the various controllers through logic blocks and has various services to monitor the energy performance of the system and create the supervision for a smart building. [1]

# CHAPTER 1

# REQUIREMENTS ANALYSIS

The Calosi company *srl* And a company that deals with Smart Building and plant management. They use the Niagara NX platform to be able to communicate over the internet with various devices in the field. Right now they need dynamic predictive analytics (*change of temperature setpoint monitors the predictive trend of energy to the customer in a dynamic way*.) in order to guarantee energy efficiency on a specific asset. To do this we need the data related to that asset and we need a service on the niagara platform to be able to process with the data. We have to take into account the season in which we are making the prediction. Energy can depend on more than one variable so you can decide the number of dependent variables which at this moment can be at most 3 variables. To do the analysis it is better to use a third-party library that has many Machine learning algorithms. At the moment a library written in java is the WEKA library which has many features to be able to filter and predict the data. Weka uses its own format called ARFF (*Attribute-Relation File Format*). The service must take a csv file with a particular format (*the first row must be the date, the second energy, from the third row onwards must be the dependent variables*), does

the necessary filters and save the result in ARFF format. There are 3 prediction algorithms which are "Linear Regresion", "Neural Network" and "SMOreg". The service must choose the best algorithm with respect to the data and save the statistical model already prepared in order to predict the energy. The service has to predict the energy on each change of input and has to know the season as well, so it picks the model suitable for that season.

# CHAPTER 2

# USE CASE DIAGRAMS

In order to better manage the project, we define two actors who are "System Integrators" who can be an engineer in the company who manages the logical scheduling of supervision and controllers and the "Service" which is the new service on the niagara platform for forecasting. .

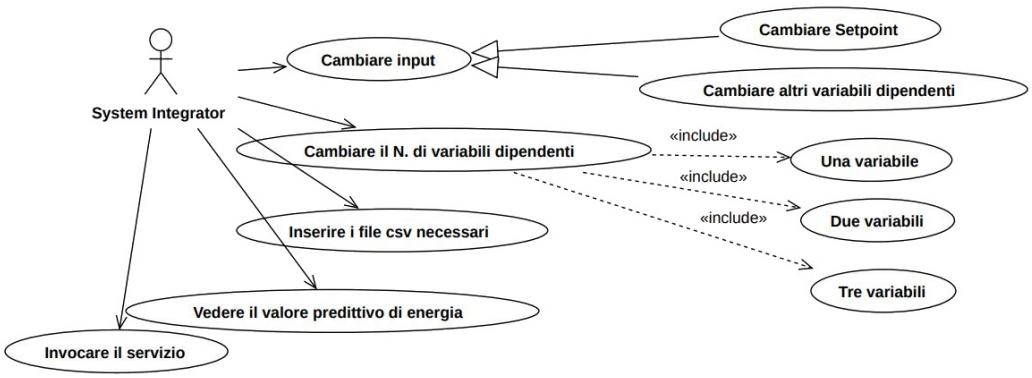



Figure 2.1: System Integrator use case diagram

System Integrator is a person who can change the input (*e.g. change degree days*) and take the prediction result as the output. He can decide the number of dependent variables and give as the dough a csv file with all the dependent attributes and their values. To solve this problem it was thought to develop an extension for the point[1] numeric of niagara and have all user related functionality on the extension.

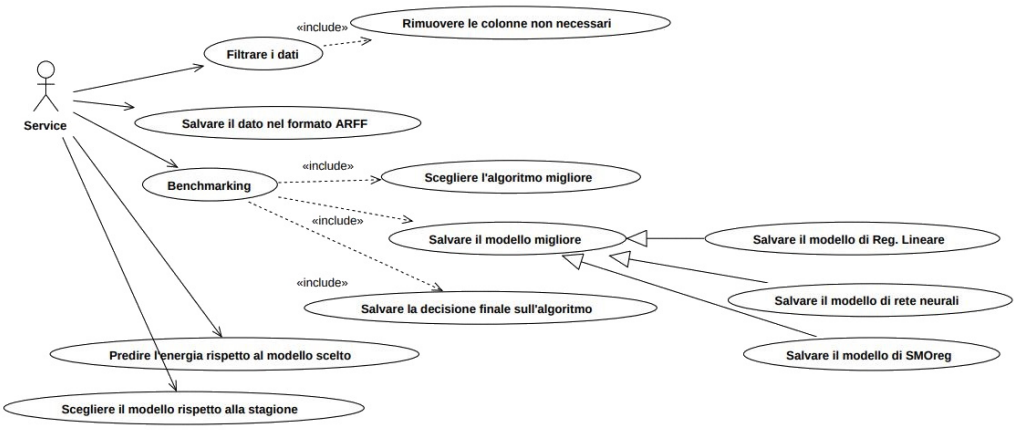



Figure 2.2: Use case diagram of the Service

Service has the role of filtering the data, benchmarking to find the most accurate algorithm taking into account the correlation coefficient, saving the file in the ARFF format and predicting consumption with respect to the input. In order not to load too many features on the service it would be useful to define more classes at design time.

[1]A point is a logical component on the niagara platform that contains a value and the status of the value. Certain extensions may be added on the point to give more functionality to the point itself. (*ex. alarm extension*)

# CHAPTER 3

# DESIGN

In this chapter we will introduce weka-api and baja-api which are two APIs that we would use in our module.

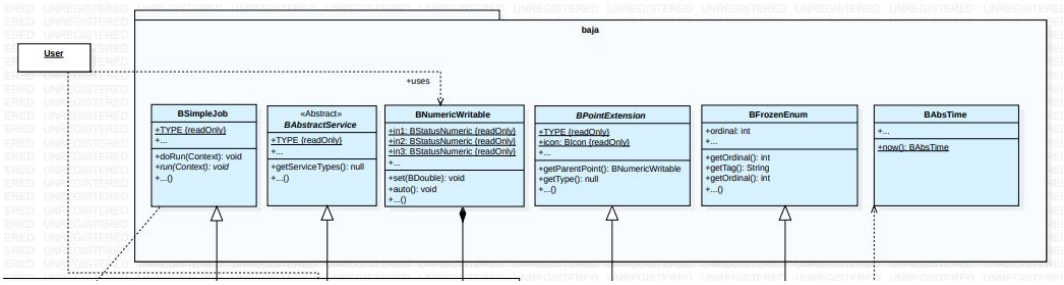



Figure 3.1: UML diagram of Baja api

In order to develop the niagara platform and create the components suitable for this platform, our module must be interfaced with baja-api ( *Building Automation Java Architecture*). On the niagara platform any object is a "BObject" (*instead of Object in java*) which in any case the father of BOject is "Object" of java. For this definition any other type has a "B" character at the beginning of the word (*ex. BAbstractService, BNumericWritable, etc ...*). This is due to the fact that the niagara al

execution time must know all related types.

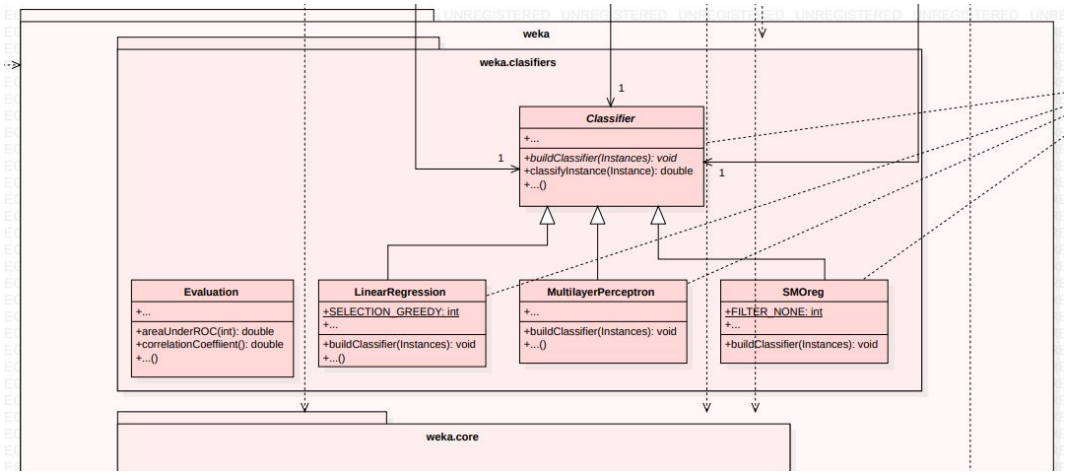



Figure 3.2: Weka api UML diagram

In order to use the Machine learning algorithms we need to interface the module with weka-api which is a library written in java. We extend certain classes of baja-api to be able to create components on the niagara platform and we use certain methods on weka-api to be able to do the prediction.

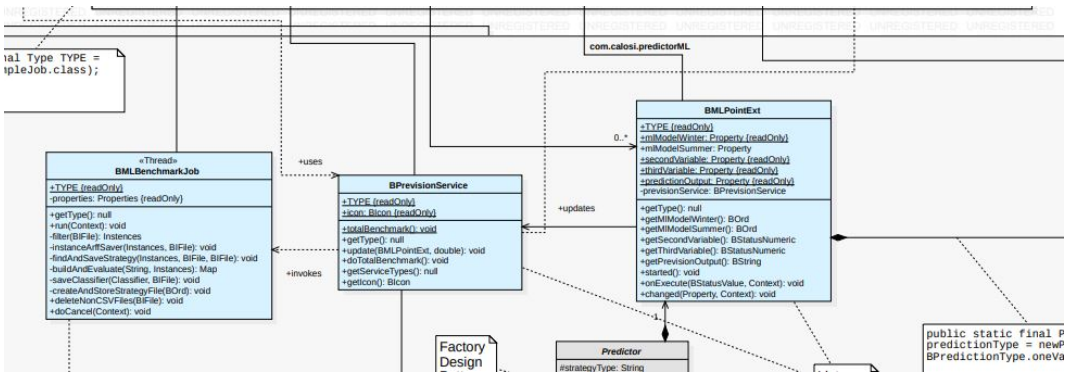



Figure 3.3: PredictorML UML diagram

Everything that has been done in this project is found in the gray package. PredictorML is the package that realizes our project and is an interface between two worlds of machine learning and automation.

the blue color indicates a class of the Baja package or a class extended by a type of the Baja package; the gray color indicates a class that has been designed to better divide the roles and the red color indicates a class of the Weka package.

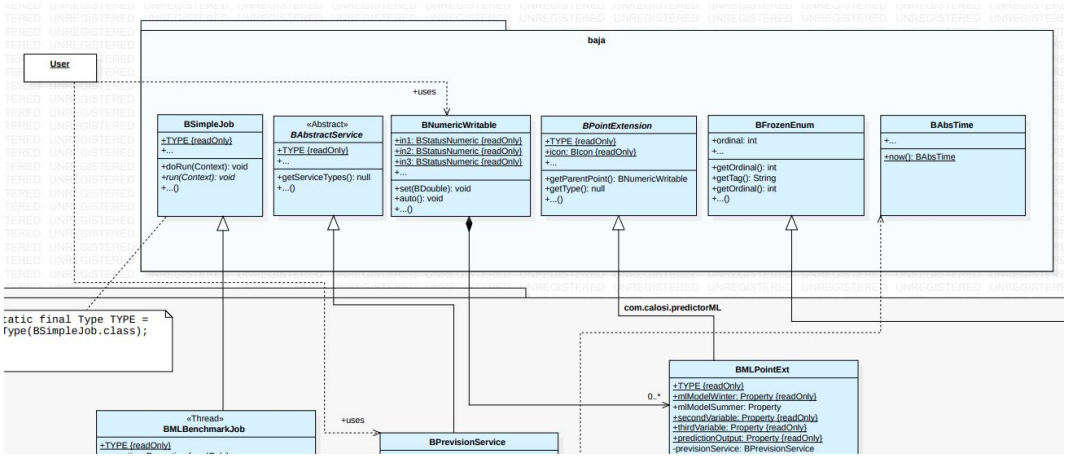



Figure 3.4: UML diagram User

As shown in the figure above user can use a numerical point and can interact with the forecast service on the niagara platform.

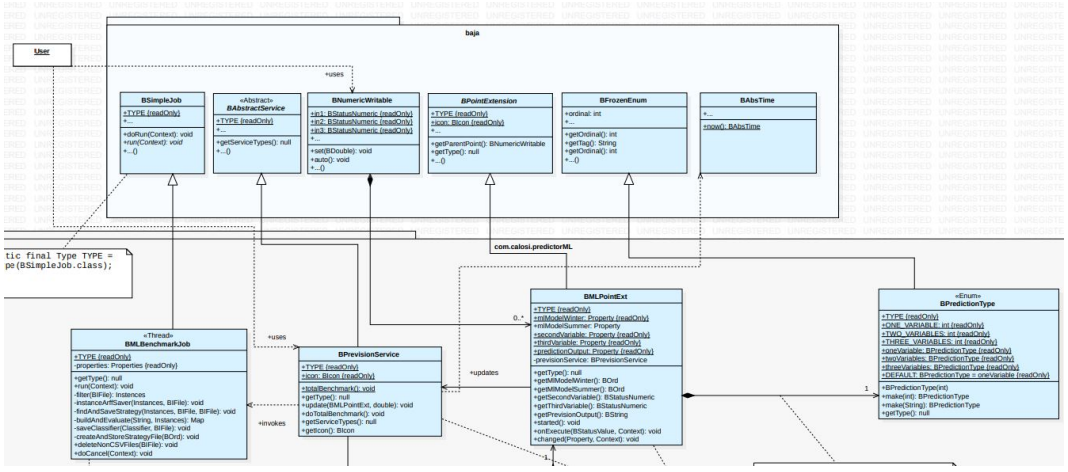
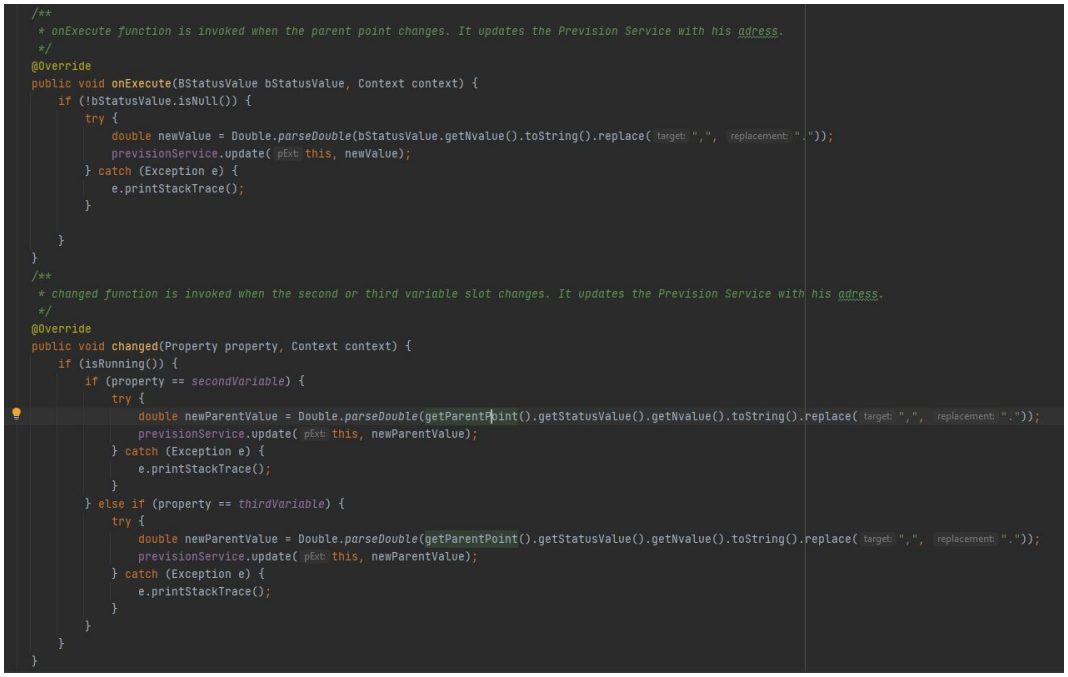



Figure 3.5: UML class diagram under the baja type

Each numeric variable could have a type extensionBMLPointExt.

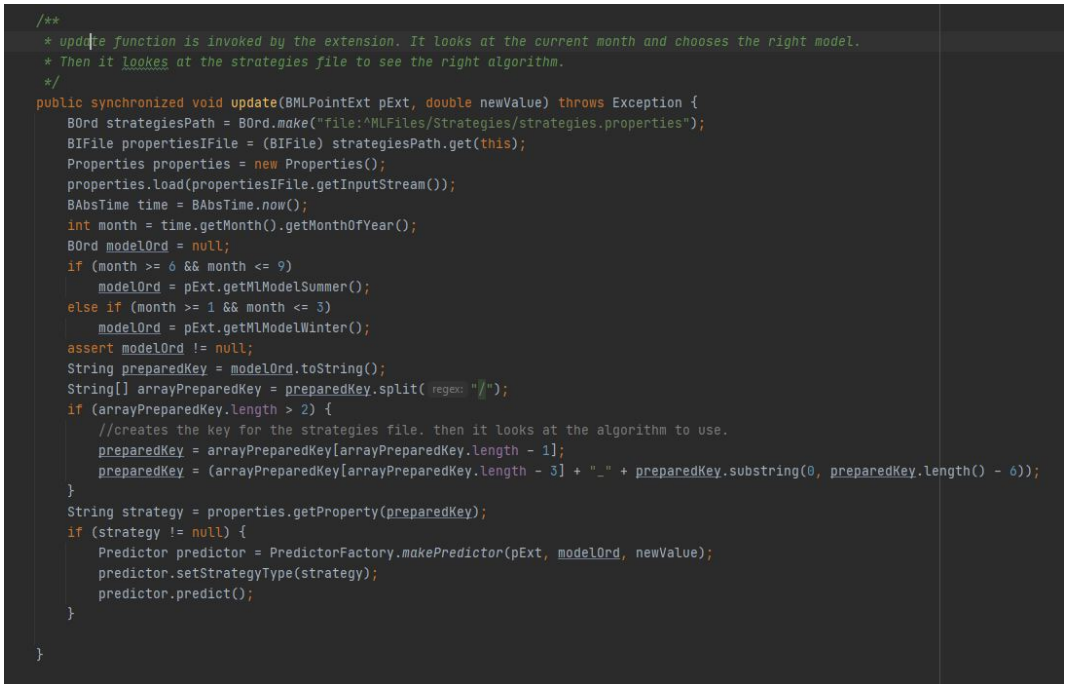


```java
/**
 * onExecute function is invoked when the parent point changes. It updates the Prevision Service with his adress.
 */
@Override
public void onExecute(BStatusValue bStatusValue, Context context) {
    if (!bStatusValue.isNull()) {
        try {
            double newValue = Double.parseDouble(bStatusValue.getNvalue().toString().replace( target: ",",  replacement: "."));
            previsionService.update( pExt: this, newValue);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
/**
 * changed function is invoked when the second or third variable slot changes. It updates the Prevision Service with his adress.
 */
@Override
public void changed(Property property, Context context) {
    if (isRunning()) {
        if (property == secondVariable) {
            try {
                double newParentValue = Double.parseDouble(getParentPoint().getStatusValue().getNvalue().toString().replace( target: ",",  replacement: "."));
                previsionService.update( pExt: this, newParentValue);
            } catch (Exception e) {
                e.printStackTrace();
            }
        } else if (property == thirdVariable) {
            try {
                double newParentValue = Double.parseDouble(getParentPoint().getStatusValue().getNvalue().toString().replace( target: ",",  replacement: "."));
                previsionService.update( pExt: this, newParentValue);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Figure 3.6: BMLPointExt

This type is a subtype of BPointExtension. Whenever our numeric variable changes its value, it will execute a block of code in the extension. The extension contains a particular type of enumerate. This type is used to tell the service how many dependent variables there are in our model. In case the wrong type is chosen, the service throws an error. Whenever the variable changes, the extension updates our service with its address and new value. The type of our service is calledBPrevisionService.



```
/**
 * update function is invoked by the extension. It looks at the current month and chooses the right model.
 * Then it lookes at the strategies file to see the right algorithm.
 */
public synchronized void update(BMLPointExt pExt, double newValue) throws Exception {
    BOrd strategiesPath = BOrd.make("file:^MLFiles/Strategies/strategies.properties");
    BIFile propertiesIFile = (BIFile) strategiesPath.get(this);
    Properties properties = new Properties();
    properties.load(propertiesIFile.getInputStream());
    BAbsTime time = BAbsTime.now();
    int month = time.getMonth().getMonthOfYear();
    BOrd modelOrd = null;
    if (month >= 6 && month <= 9)
        modelOrd = pExt.getMlModelSummer();
    else if (month >= 1 && month <= 3)
        modelOrd = pExt.getMlModelWinter();
    assert modelOrd != null;
    String preparedKey = modelOrd.toString();
    String[] arrayPreparedKey = preparedKey.split( regex: "/");
    if (arrayPreparedKey.length > 2) {
        //creates the key for the strategies file. then it looks at the algorithm to use.
        preparedKey = arrayPreparedKey[arrayPreparedKey.length - 1];
        preparedKey = (arrayPreparedKey[arrayPreparedKey.length - 3] + "_" + preparedKey.substring(0, preparedKey.length() - 6));
    }
    String strategy = properties.getProperty(preparedKey);
    if (strategy != null) {
        Predictor predictor = PredictorFactory.makePredictor(pExt, modelOrd, newValue);
        predictor.setStrategyType(strategy);
        predictor.predict();
    }

}
```

Figure 3.7: BPrevisionService

This service extends from BAbstractService which is the father of all services on niagara. The service has the ability to do full benchmarking. This feature contains filtering, saving in the right format and choosing the best strategy. The link between the extension and the service was created with listener design patterns. The Observer design pattern cannot be used due to the fact that the subject class already has a parent. Whenever the user calls total benchmarking, the service launches a thread with the type BMLBenchmarkjob not to block all the rest of the framework.

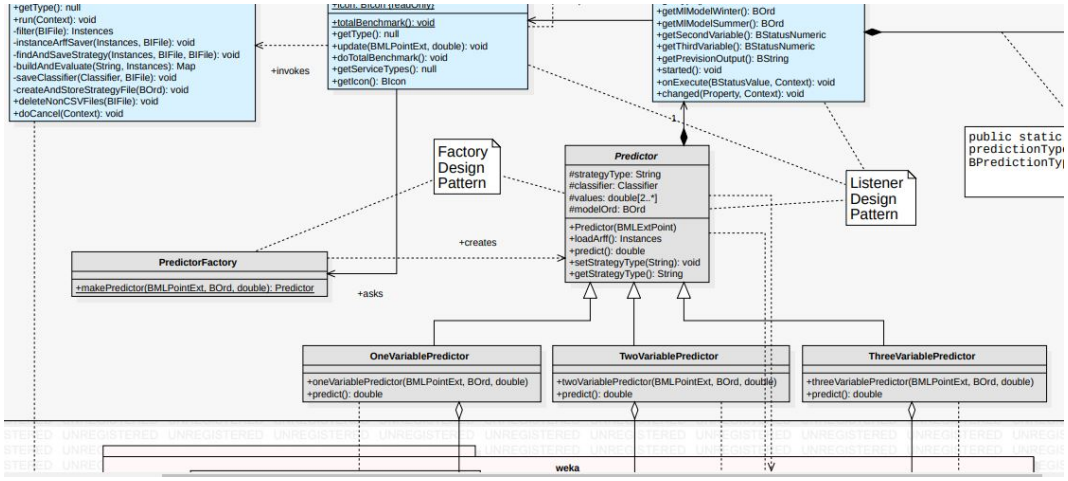



Figure 3.8: UML Diagram of Factory Design Pattern

Once the benchmarking is finished and we have the model saved there is the possibility to make the prediction.

```java
/**
 * Predictor class is the parent class for each predictor. Every predictor has a strategy type and has a predict function to see the future.
 * They also have the loadArff method to load the data for the Weka api. The values attribute is an array of new values.
 */
public abstract class Predictor {
    Predictor(BMLPointExt pointExt, BOrd modelOrd) {
        this.pointExt = pointExt;
        this.modelOrd = modelOrd;
    }

    String getStrategyType() { return strategyType; }

    void setStrategyType(String strategyType) throws Exception {
        this.strategyType = strategyType;
        BIFile modelFile = (BIFile) modelOrd.get(pointExt);
        switch (strategyType) {
            case "SMOreg":
                classifier = (SMOreg) SerializationHelper.read(modelFile.getInputStream());
                break;
            case "MultilayerPerceptron":
                classifier = (MultilayerPerceptron) SerializationHelper.read(modelFile.getInputStream());
                break;
            default:
                classifier = (LinearRegression) SerializationHelper.read(modelFile.getInputStream());
        }
    }

    double predict() throws Exception {
        return -99;
    }

    Instances loadArff() throws IOException {
        ArffLoader arffLoader = new ArffLoader();
        String modelOrdString = modelOrd.toString();
        modelOrdString = modelOrdString.replace( target: "models",  replacement: "arff");
        modelOrdString = modelOrdString.replace( target: ".model",  replacement: ".arff");
        BOrd arffOrd = BOrd.make(modelOrdString);
        BIFile arffFile = (BIFile) arffOrd.get(pointExt);
        arffLoader.setSource(arffFile.getInputStream());
        return arffLoader.getDataSet();
    }
```

Figure 3.9: Predictor

This was accomplished with the fact that whenever there is a variation on the point and the service knows about it, a Predictor to do the

forecast. The predictor must be of type one variable, two variables or three variables depending on which one the user has chosen. It is worth noting that the class totally changes with respect to the number of variables.

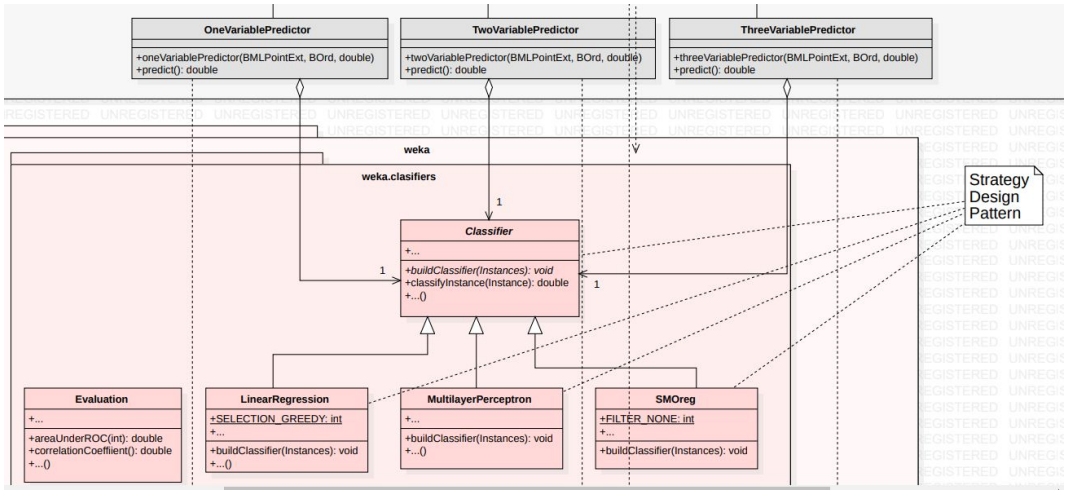



Figure 3.10: UML Diagram of Strategy Design Pattern

The predictor will go to create a Classifier of a particular type depending on the strategy that was decided at the time of benchmarking. This was done with the Factory design pattern and Strategy design pattern.

```
public static final Property predictionType =
```

Figure 3.11: Static Property

It should be noted that making a module for the niagara platform carries some constraints on the code. One constraint we almost always see is that every property on our component is a static attribute.

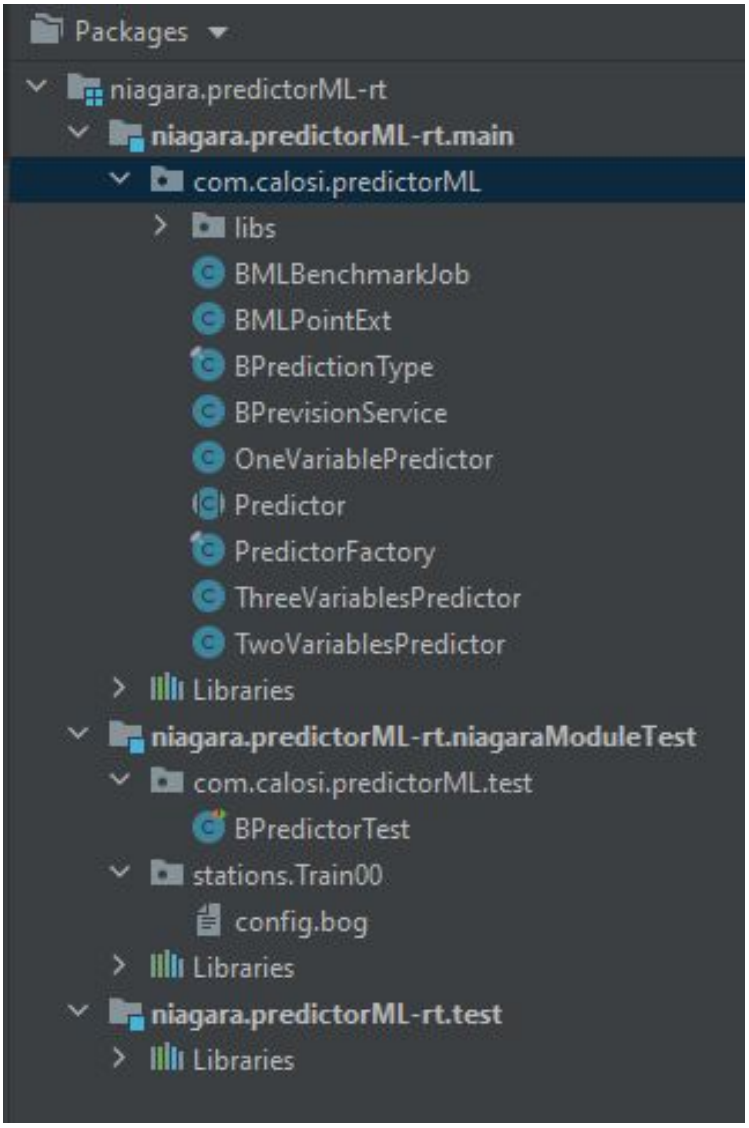



Figure 3.12: Packages

In the figure above we can see a summary of the packages and classes made in this project.

# CHAPTER 4

# MOCKUPS & EXPERIMENTATION

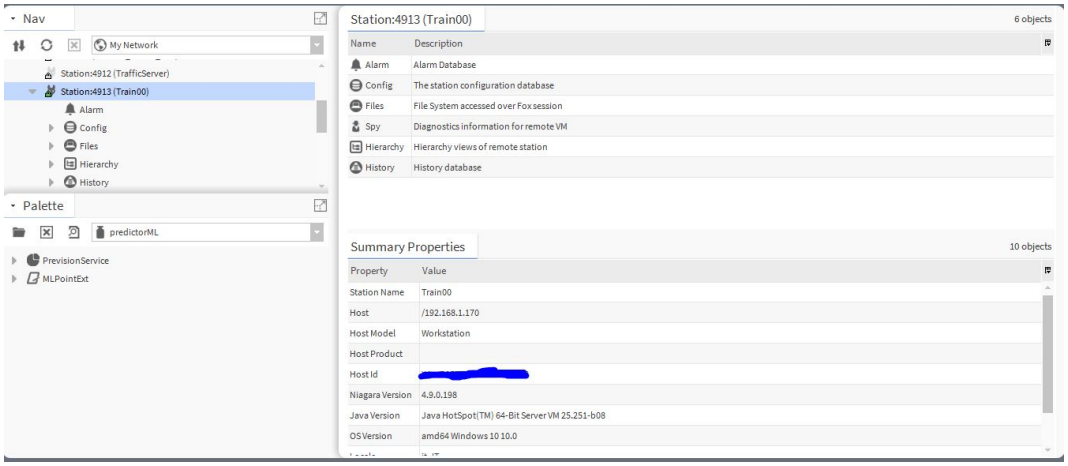



Figure 4.1: Niagara Framework

This is a general figure of the Niagara framework. On the lower left we can find our new module which is built from the service and the extension. We can drag the service under all the services of our station and we can add the extension under the required point.

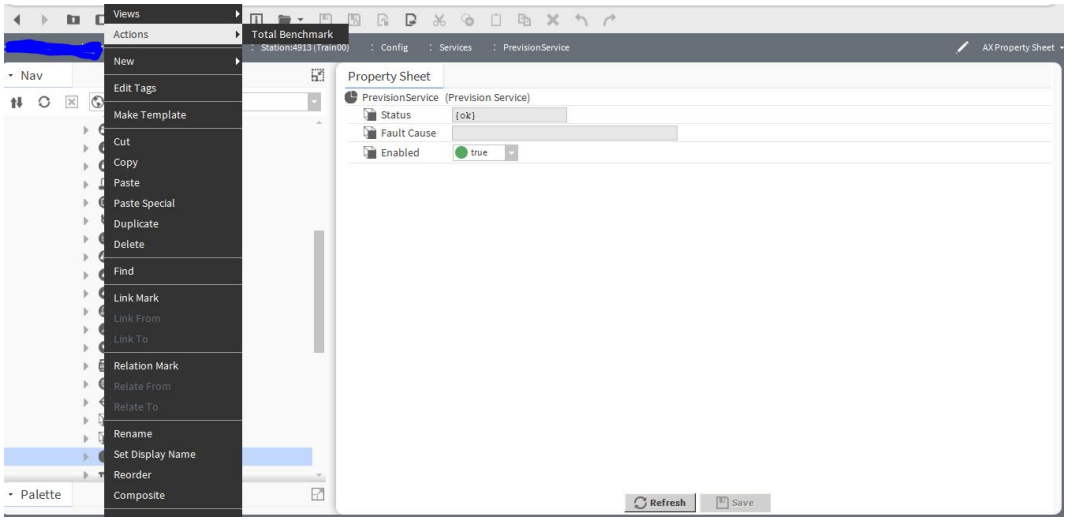



Figure 4.2: Total Benchmarking

Once the service has been added, we need to put the csv files under the MLFiles folder and right click on the service and benchmark.

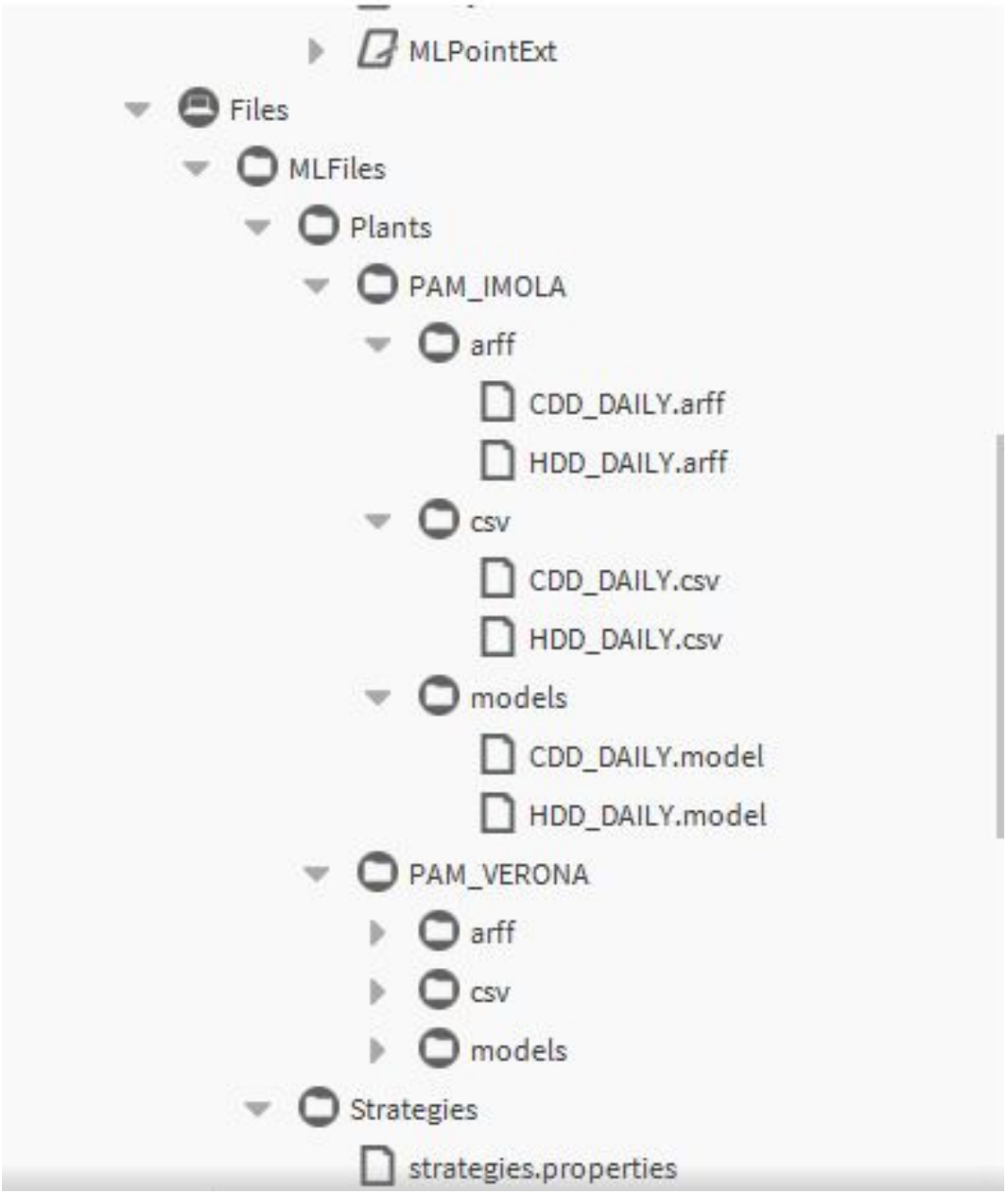



Figure 4.3: After benchmarking

When the benchmarking process is finished, all necessary files will be saved as shown in the figure above.

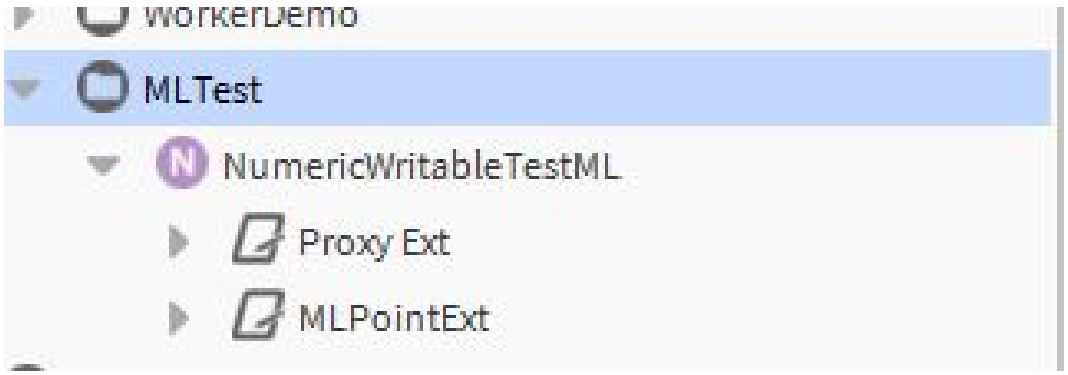



Figure 4.4: Extension

This is a numeric point that the extension will fit over the point.

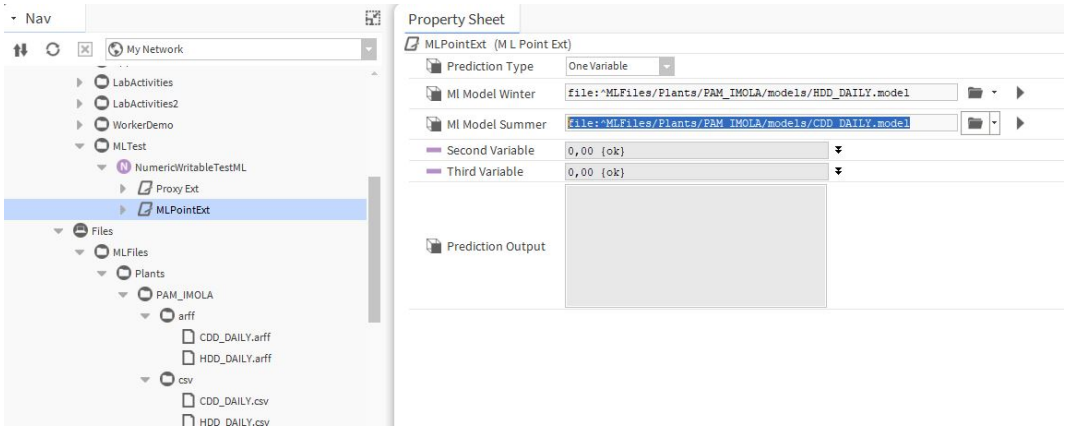



Figure 4.5: Extension properties

In this step you have to insert the right model on the extension property and select the number of variables. The model is created in the benchmarking phase.

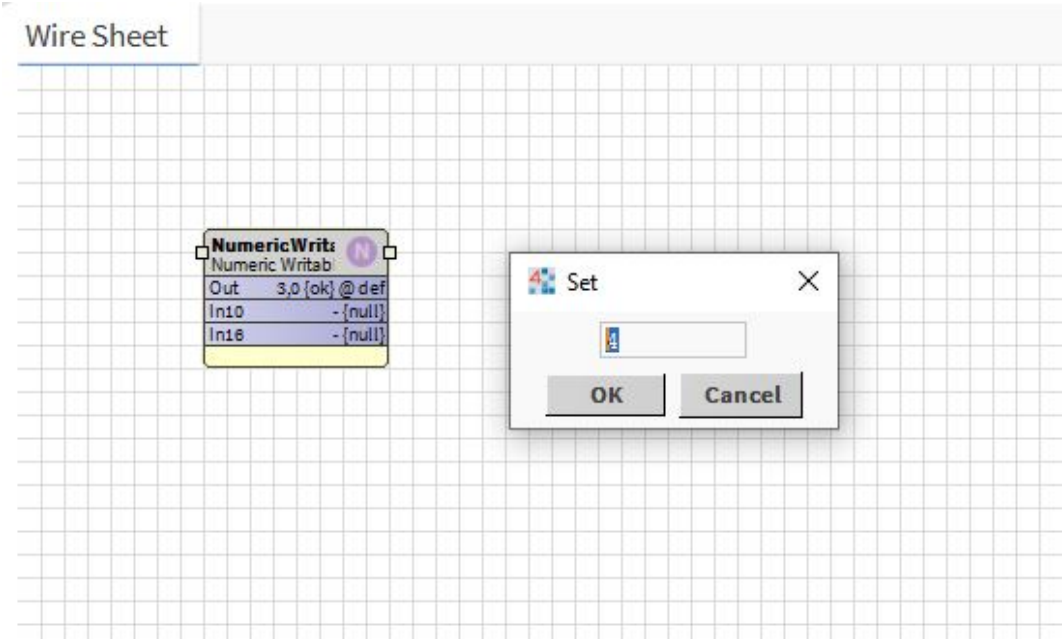



Figure 4.6: Setting

At this point, every time the point changes, the service updates and makes available the predictive value calculated with respect to the season.

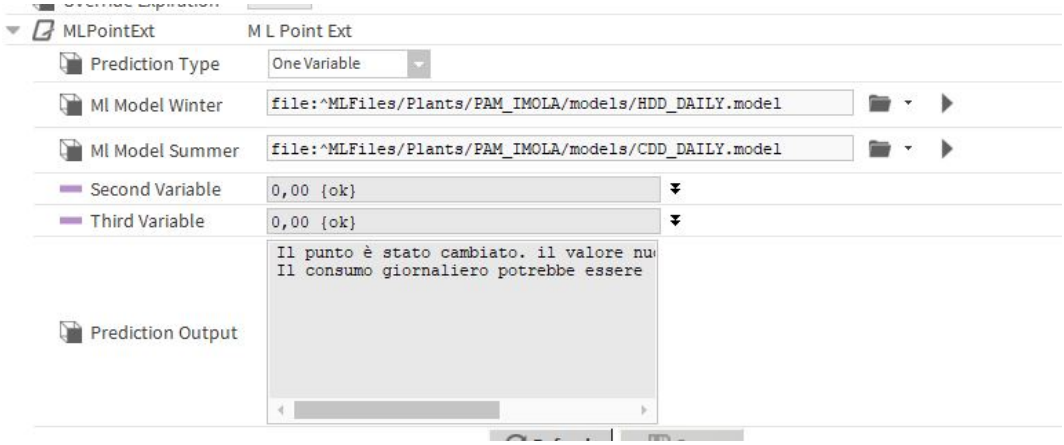



Figure 4.7: The forecast result

In this figure you can see the result obtained in the string format. Keep in mind that we have tried with only one variable but it could also be tested with more variables.

# CHAPTER 5

# TESTING

To test a baja module you need to use a special test library which has been extended by java's TestNg library.

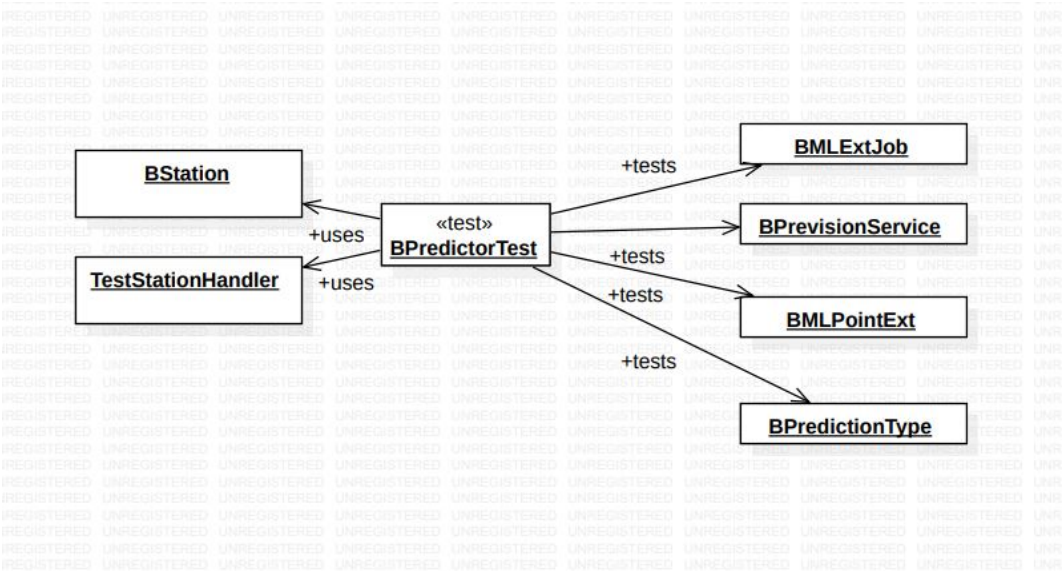



Figure 5.1: Testing Object Diagram

This library allows us to create a niagara environment to do a test. That is, you can initialize a station at the code level and do the test on our station. To initialize a station you can load a configuration file to have a station already prepared. The

configuration of the station used in the chapter *Mockups* but without the service, dot extension and other files.

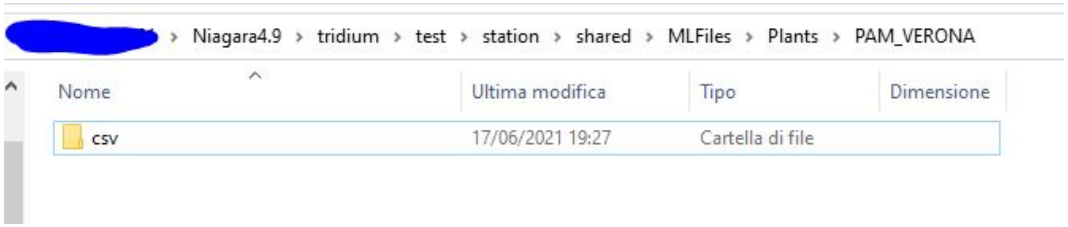



Figure 5.2: CSV files in the testing phase

We only have the CSV files on our station. We have created a single test class to test core functionality. This includes testing the benchmarking, the enumerated class, and the point extension.

```
/**
 * totalBenchmark test case creates a prediction service, adds it on a test station.
 * Then it invokes the total benchmark to see if it creates the right model.
 */
@Test(groups = "a")
public void totalBenchmark() throws InterruptedException {
    BPrevisionService previsionService = new BPrevisionService();
    BServiceContainer services = station.getServices();
    services.add( name: "PrevisionService", previsionService);
    previsionService.doTotalBenchmark();
    sleep( millis: 3000);
    BOrd bOrd = BOrd.make("file:^MLFiles/Plants/PAM_IMOLA/models/CDD_DAILY.model");
    BIFile file = (BIFile) bOrd.get(station);
    assertEquals(file.getFileName(),  expected: "CDD_DAILY.model");
}
```

Figure 5.3: Sleep function

In the test we can see that some delays have been put (*sleep (3000)*). This is to wait for the thread to finish its work.

| Test | Methods Passed | Scenarios Passed | # skipped | # failed | Total Time | Included Groups | Excluded Groups |
|---|---|---|---|---|---|---|---|
| Command line test | 4 | 4 | 0 | 0 | 22,6 seconds | | |

| Class | Method | # of Scenarios | Start | Time (ms) |
|---|---|---|---|---|
| **Command line test — passed** | | | | |
| com.calosi.predictorML.test.BPredictorTest | stationName (a) | 1 | 1624108274908 | 13 |
| | totalBenchmark (a) | 1 | 1624108274931 | 3080 |
| | checkPredictionType | 1 | 1624108278015 | 4 |
| | setAndCheckOutput | 1 | 1624108278020 | 3033 |

Figure 5.4: Test Result

As we can see, all the tests were successful. So many other practical tests have been done that have not been put into this documentation.

# REFERENCES

[1] Tridium. *Niagara NX website*. url: https://www.tridium.com/us/en/Products / niagara.

# LIST OF FIGURES