UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

Progetto e Sviluppo di un Componente Finalizzato

Professore:

Ch.mo Prof.
Enrico Vicario

Candidato:

Hamidreza Hashemi
Mat. 5996260

Corso:

Ingegneria del Software

ANNO ACCADEMICO 2020/2021

Dedicata a coloro che
ci provano ogni giorno.

INDICE

INTRODUZIONE

In questo esercizio introdurremo un'esigenza a livello industriale e lo risolviamo con le tecniche introdotte nel corso di Ingegneria del Software. I campi relativi sono :

- Machine Learning
- Statistica
- Informatica Industriale
- IOT (*internet of things)*

Sarebbe utile avere una conoscenza sulla piattaforma Niagara NX [1]

[1]Piattaforma Niagara NX è una piattaforma sviluppato da Tridium (*un'azienda sotto Honeywell*) per la gestione IOT. Questa piattaforma permette di programmare i vari controllori attraverso dei blocchi logici e ha dei vari servizi per poter monitorare andamento energetico dell'impianto e creare la supervisione per un smart building.[1]

CAPITOLO 1

ANALISI DEI REQUISITI

L'azienda Calosi *s.r.l* è un'azienda che si occupa nell'ambito di Smart Building e la gestione degli impianti. Loro usano la piattaforma Niagara NX per poter comunicare attraverso la rete internet con vari dispositivi nel campo. In questo momento loro hanno bisogno delle analisi predittive dinamiche (*cambio di setpoint della temperatura monitora al cliente in un modo dinamico l'andamento predittivo di energia.*) in modo da garantire un'efficienza energetica su un asset specifico. Per fare questo abbiamo bisogno dei dati relativi a quel asset e abbiamo bisogno di un servizio sulla piattaforma niagara per poter elaborare con i dati. Bisogna prendere in considerazione la stagione in cui stiamo facendo la predizione. L'energia può dipendere da più di una variabile quindi si può decidere il numero di variabili dipendenti che in questo momento al massimo possono essere 3 variabili. Per fare l'analisi conviene usare una libreria di terze parti che ha tanti algoritmi Machine learning. Al momento una libreria scritto in java è la libreria WEKA che ha tante funzionalità per poter filtrare e predire il dato. Weka usa un formato suo che si chiama ARFF (*Attribute-Relation File Format*). Il servizio deve prendere un file csv con un formato particolare (*la prima riga deve essere la data, la seconda energia, dalla terza riga in poi devono essere i variabili dipendenti*), fa

i filtraggi necessari e salva il risultato nel formato ARFF. Ci sono 3 algoritmi di predizione che sono "Regresione Lineare", "Rete neurali" e "SMOreg". Il servizio deve scegliere l'algoritmo migliore rispetto al dato e salva il modello statistico già preparato per poter predire l'energia. Il servizio deve predire l'energia su ogni cambimento dell'input e deve sapere anche la stagione, così prende il modello adatto per quella stagione.

CAPITOLO 2

USE CASE DIAGRAMS

Per poter gestire meglio il progetto, definiamo due attori che sono "System Integrator" che può essere un'ingegnere nell'azienda che gestisce la prorammazione logica della supervisione e i controllori e il "Service" che è il servizio nuovo sulla piattaforma niagara per la previsione.

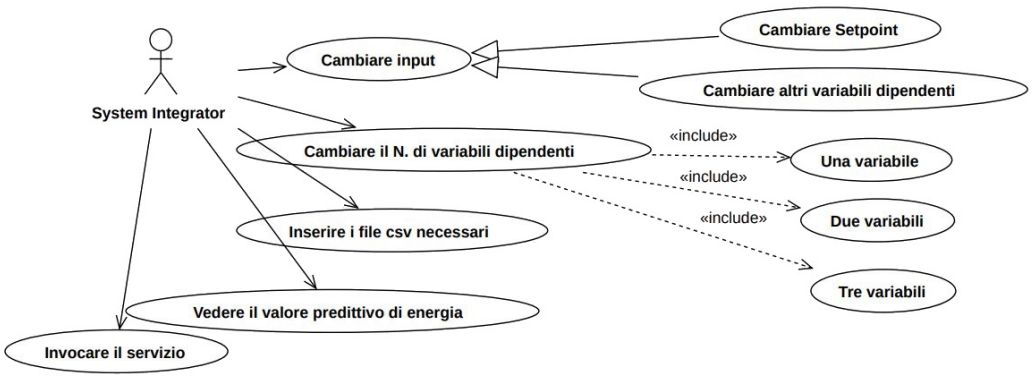


Figura 2.1: Use case diagram del System Integrator

System Integrator è una persona che può cambiare l'input (*es.cambiare gradi giorno*) e prendere come output il risultato di predizione. Lui può decidere il numero di variabili dipendenti e dare come l'impasto un file csv con tutti gli attributi dipendenti e il loro valori. Per risolvere questo problema è stato pensato di sviluppare un'estensione per il punto[1] numerico di niagara e avere tutte le funzionalità relative all'utente sull'estensione.

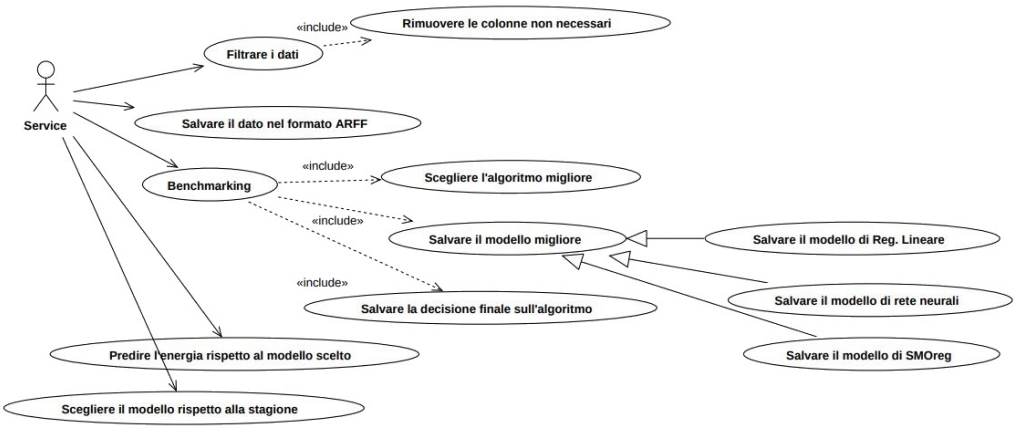


Figura 2.2: Use case diagram del Service

Service ha il ruolo di filtrare i dati, fare il benchmarking per trovare l'algoritmo più preciso tenendo il conto di coefficiente di correlazione, salvare il file nel formato ARFF e predire il consumo rispetto all'input. Per non caricare troppe funzionalità sul servizio sarebbe utile definire più classi al momento della progettazione.

[1]Un punto è un componente logico sulla piattaforma niagara che contiene un valore e lo stato del valore. Possono essere aggiunti certe estensioni sul punto per dare più funzionalità al punto stesso.(*es. estensione allarme*)

CAPITOLO 3

PROGETTAZIONE

In questo capitolo introdurremmo weka-api e baja-api che sono due API che useremmo nel nostro modulo.

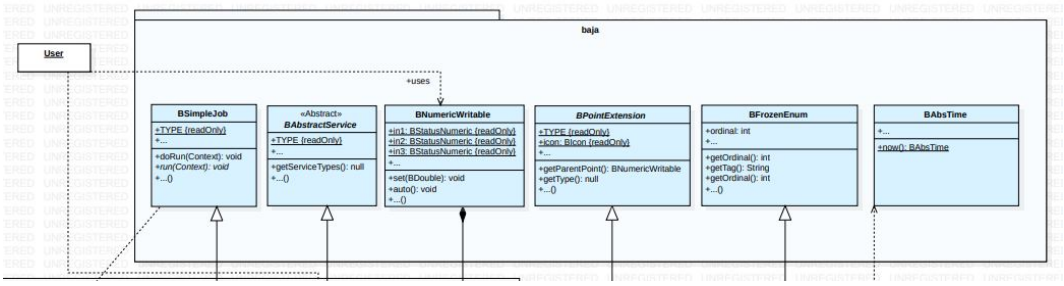


Figura 3.1: UML diagram di Baja api

Per poter sviluppare la piattaforma niagara e creare i componenti adatti per questa piattaforma bisogna interfacciare il nostro modulo con baja-api (*Building Automation Java Architecture*). Sulla piattaforma niagara qualsiasi oggetto è un "BObject" (*invece di Object in java*) che comunque il padre di BOject è "Object" di java. Per questa definizione qualsiasi altro tipo ha un carattere "B" all'inizio della parola (*es. BAbstractService, BNumericWritable, etc...*). Questo è per il fatto che la piattaforma niagara al

momento di esecuzione deve conoscere tutti i tipi relativi.

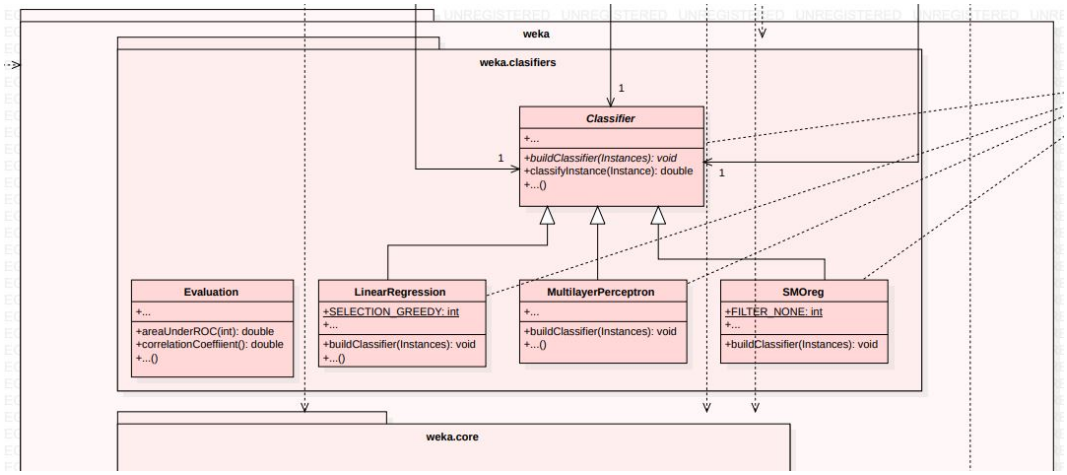


Figura 3.2: UML diagram di Weka api

Per poter usare gli algoritmi di Machine learning abbiamo bisogno di interfacciare il modulo con weka-api che è una libreria scritto in java. Estendiamo certe classi di baja-api per poter creare componenti sulla piattaforma niagara e usiamo certi metodi su weka-api per poter fare la previsione.

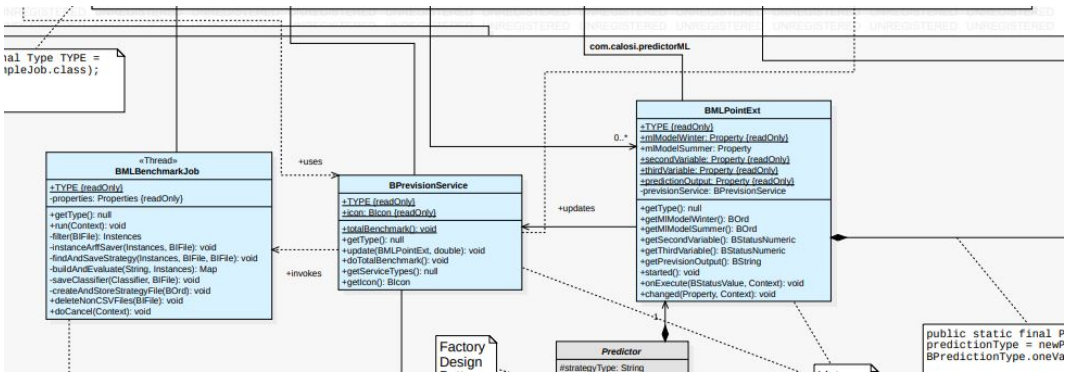


Figura 3.3: UML diagram di PredictorML

Tutto quello che è stato realizzato in questo progetto, si trova nel pacchetto grigio. PredictorML è il pacchetto che realizza il nostro progetto ed è un'interfaccia tra due mondi di machine learning e automazione.

il colore blu indica una classe del pacchetto Baja oppure una classe esteso da un tipo del pacchetto Baja; il colore grigio indica una classe che è stato realizzato per dividere meglio i ruoli e il colore rosso indica una classe del pacchetto Weka.

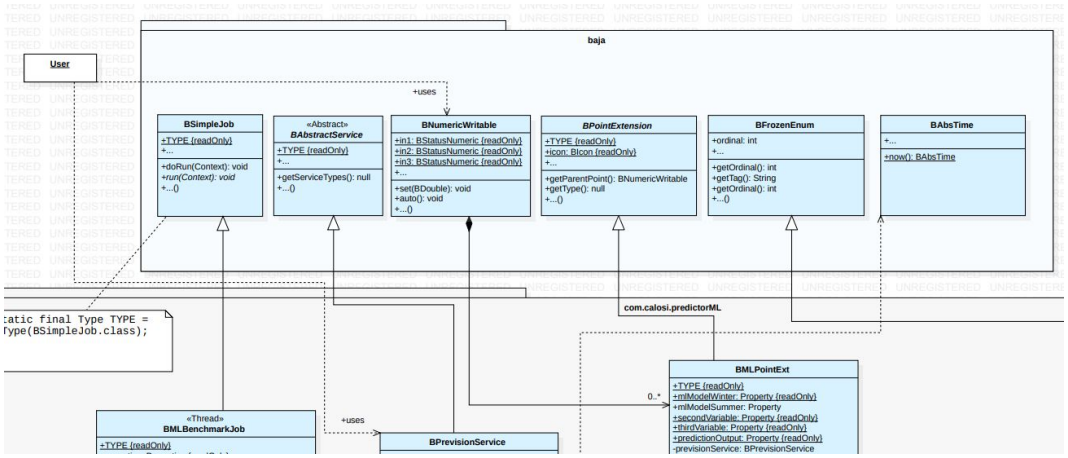


Figura 3.4: UML diagram Utente

Come è mostrato nella figura sopra utente può usare un punto numerico e può interagire con il servizio di previsione sulla piattaforma niagara.

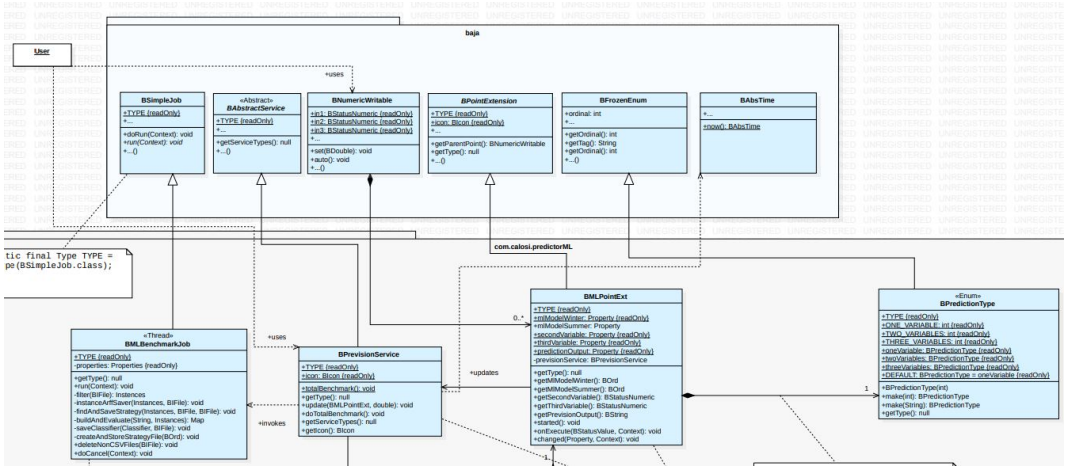
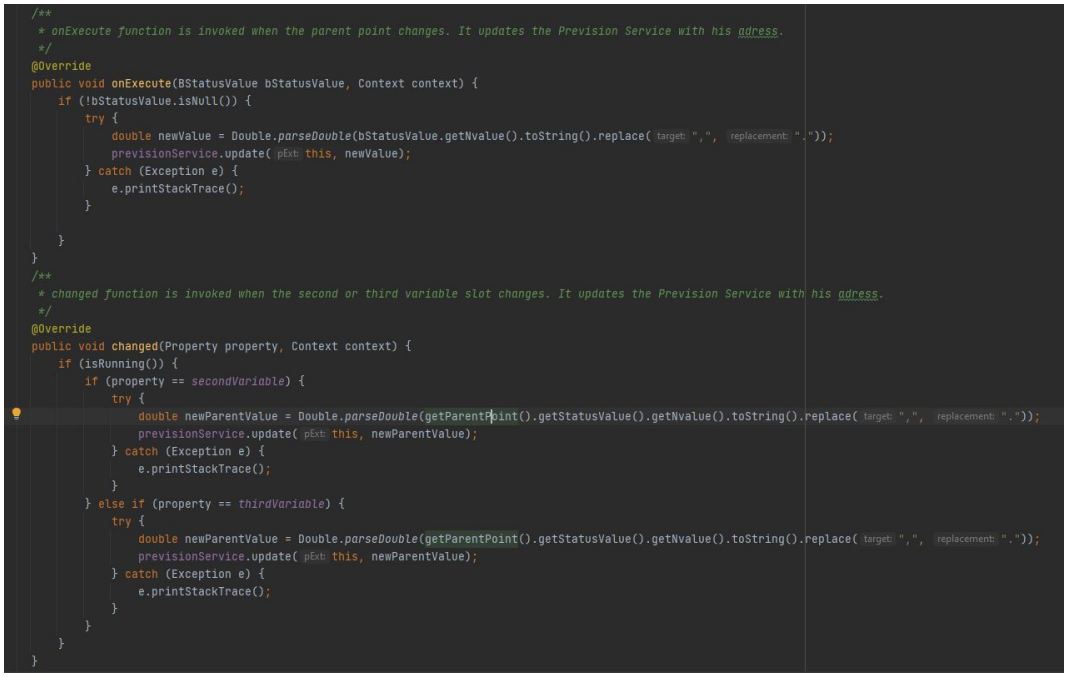


Figura 3.5: UML diagram di classi sotto il tipo baja

Ogni variabile numerico potrebbe avere un'estensione del tipo **BMLPointExt**.



```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();
            }
        }
    }
}
```

Figura 3.6: BMLPointExt

Questo tipo è un sottotipo di BPointExtension. Ogni volta che la nostra variabile numerica cambia il valore, andrà in esecuzione un blocco del codice nell'estensione. L'estensione contiene un tipo particolare di enumerato. Questo tipo serve per dire al servizio quante variabili dipendenti ci sono nel nostro modello. Nel caso in cui viene scelto il tipo sbagliato, il servizio lancia un'errore. Ogni volta che la variabile cambia, l'estensione aggiorna il nostro servizio con il suo indirizzo e il valore nuovo. Il tipo del nostro servizio si chiama **BPrevisionService**.

```java
/**
 * 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();
    }

}
```

Figura 3.7: BPrevisionService

Questo servizio estende da BAbstractService che è il padre di tutti i servizi sul niagara. Il servizio ha la capacità di fare benchmarking totale. Questa funzione contiene filtraggio, salvataggio nel formato giusto e la scelta della strategia migliore. Il legame tra l'estensione e il servizio è stato realizzato con listener design pattern. Non si può usare l'Observer design pattern per il fatto che la classe subject ha già un padre. Ogni volta che l'utente chiama il benchmarking totale, il servizio lancia un thread con il tipo **BMLBenchmarkjob** per non bloccare tutto il resto del framework.

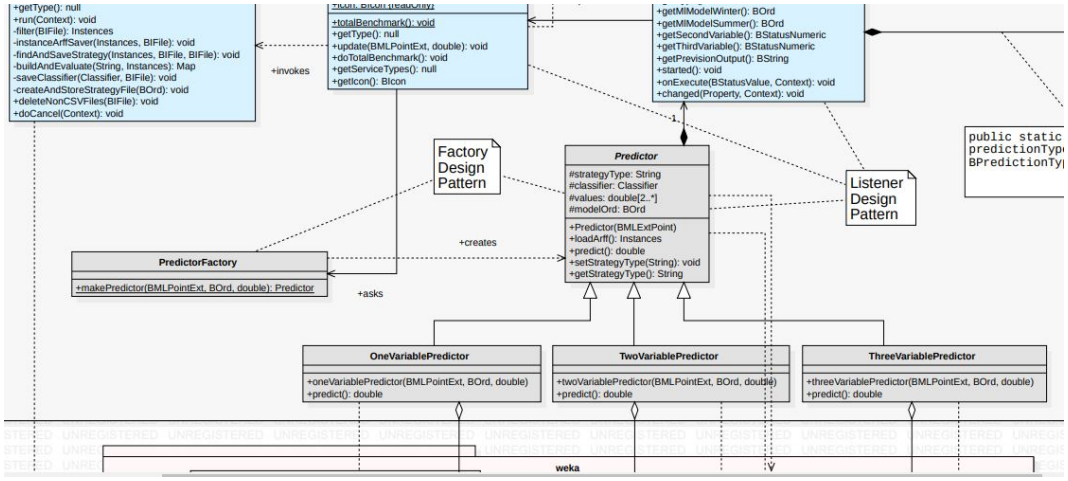


Figura 3.8: UML Diagram di Factory Design Pattern

Una volta finito il benchmarking e abbiamo il modello salvato c'è la possibilità di fare la previsione.

```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();
    }
```

Figura 3.9: Predictor

Questo è stato realizzato con il fatto che ogni volta che c'è una variazione sul punto e il servizio lo conosce, verrà creato un **Predictor** per fare la

previsione. Il predictor deve essere di tipo una variabile, due variabili o tre variabili a seconda quella che l'utente ha scelto. È utile notare che la classe cambia totalmente rispetto al numero di variabili.

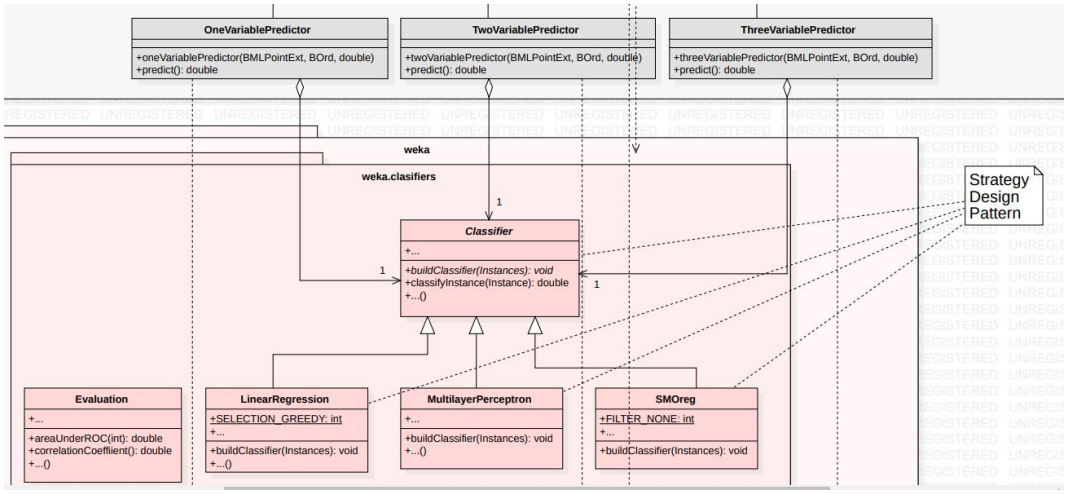


Figura 3.10: UML Diagram di Strategy Design Pattern

Il predictor andrà a creare un **Classifier** di un tipo particolare a seconda quella strategia che è stato deciso al momento di benchmarking. Questo è stato realizzato con il Factory design pattern e Strategy design pattern.

```
public static final Property predictionType
```

Figura 3.11: Static Property

C'è da notare che fare un modulo per la piattaforma niagara, porta alcuni vincoli sul codice. Un vincolo che vediamo quasi sempre è che ogni proprietà sul nostro componente è un attributo statico.

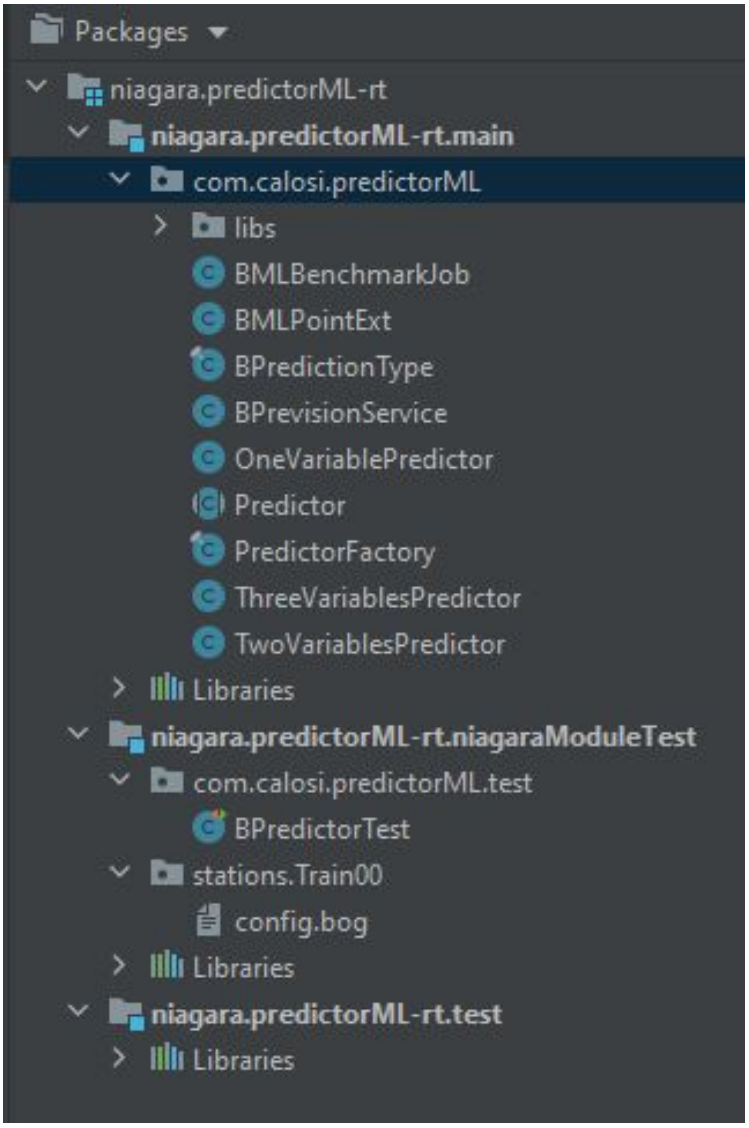


Figura 3.12: Packages

Nella figura di sopra possiamo vedere un riassunto dei pacchetti e i classi fatti in questo progetto.

CAPITOLO 4

MOCKUPS & ESPERIMENTAZIONE

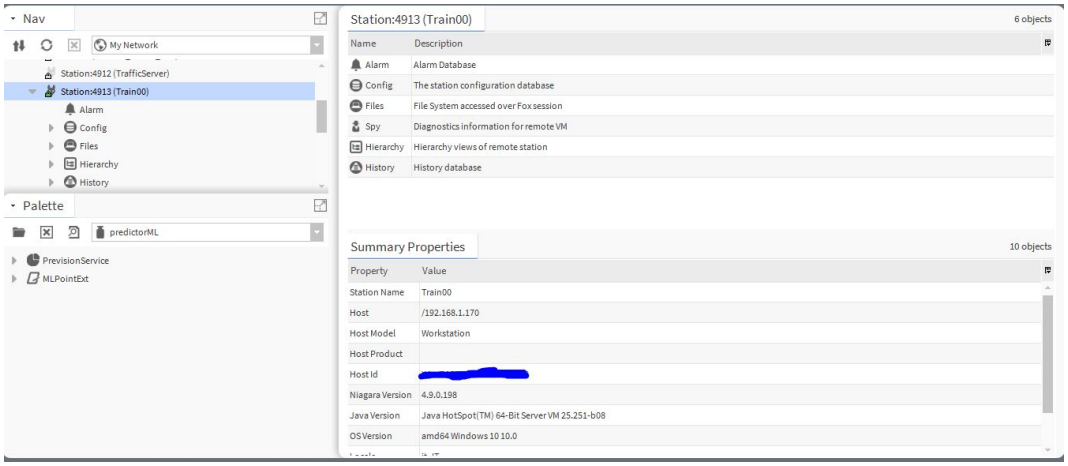


Figura 4.1: Niagara Framework

Questo è una figura generale del Niagara framework. Sulla parte basso in sinistra possiamo trovare il nostro modulo nuovo che è costruito dal servizio e dall'estensione. Possiamo trascinare il servizio sotto tutti i servizi della nostra station e possiamo aggiungere l'estensione sotto il punto richiesto.

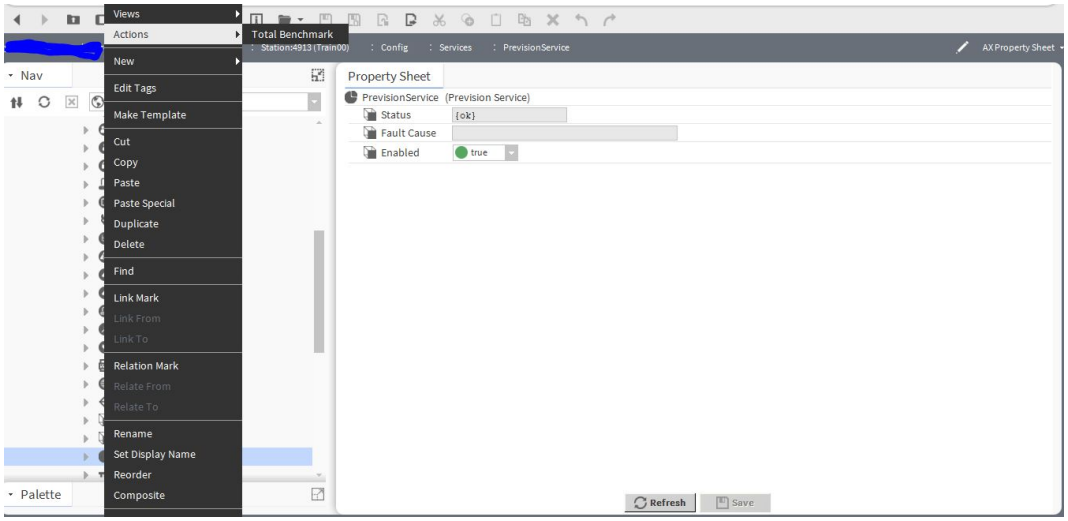


Figura 4.2: Total Benchmarking

Una volta è stato aggiunto il servizio, dobbiamo mettere i file csv sotto la cartella MLFiles e cliccare il tasto destro sul servizio e fare il benchmarking.

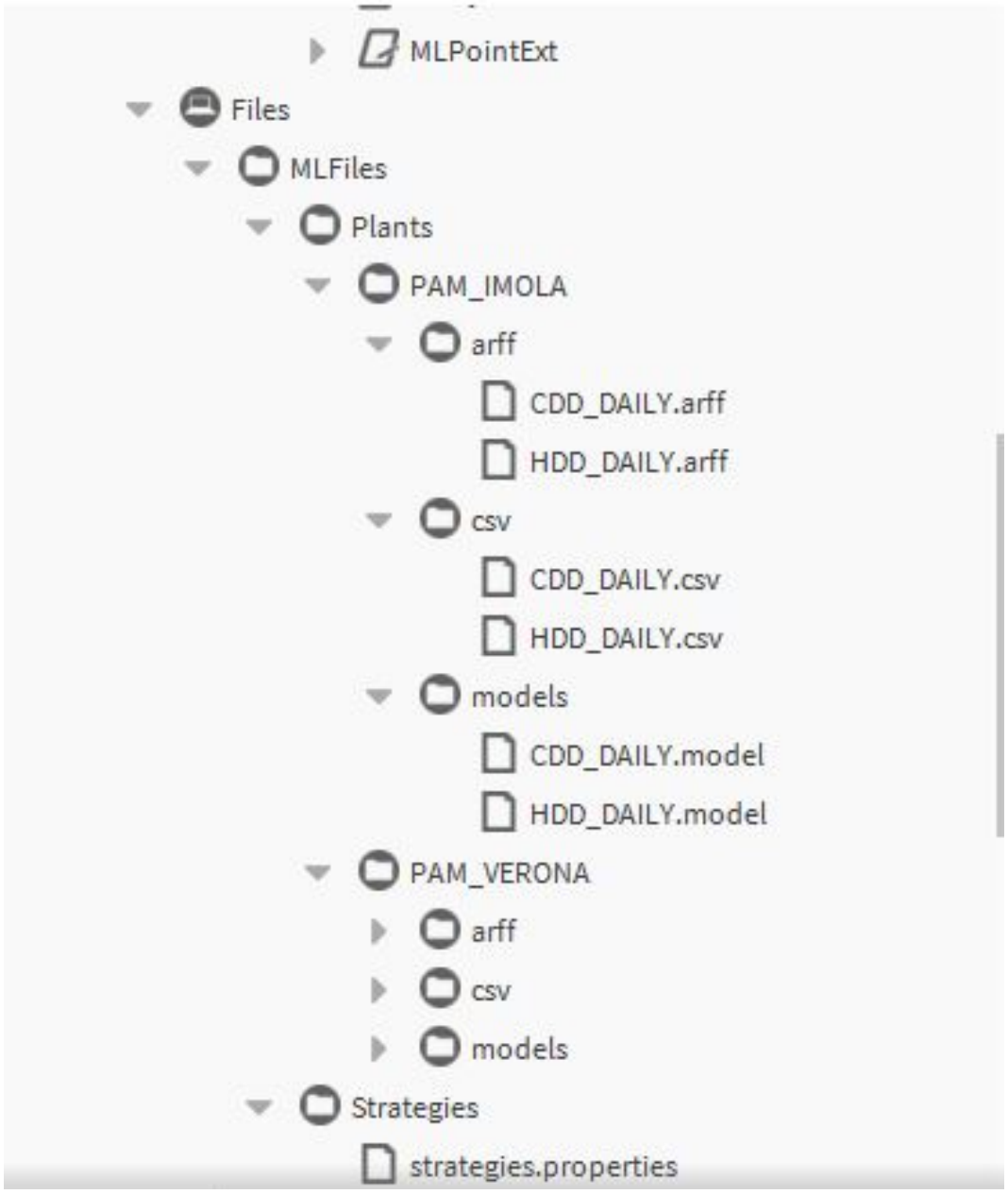


Figura 4.3: Dopo benchmarking

Quando è finito il processo di benchmarking, verrà salvato tutti i file necessari come è mostrato nella figura di sopra.

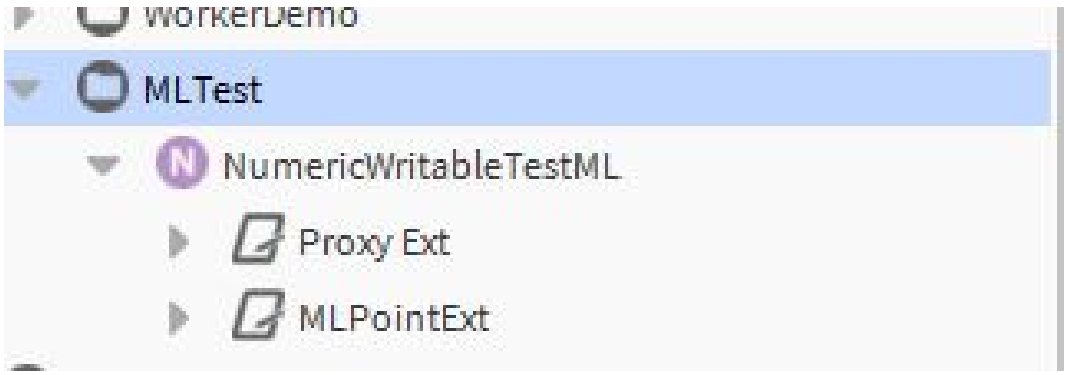


Figura 4.4: Estensione

Questo è un punto numerico che l'estensione verrà montata sul punto.

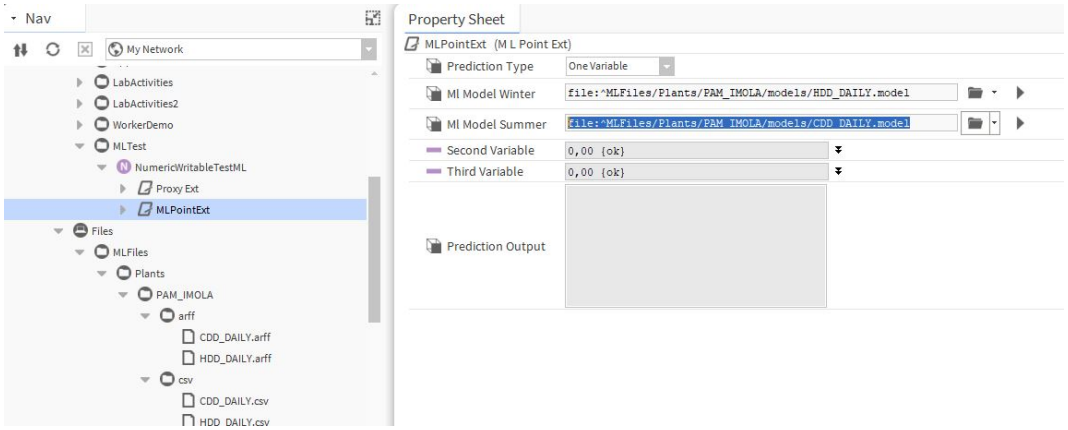


Figura 4.5: Proprietà dell'estensione

In questo passo bisogna inserire il modello giusto sulla proprietà dell'estensione e selezionare il numero di variabili. Il modello viene creato nella fase di benchmarking.

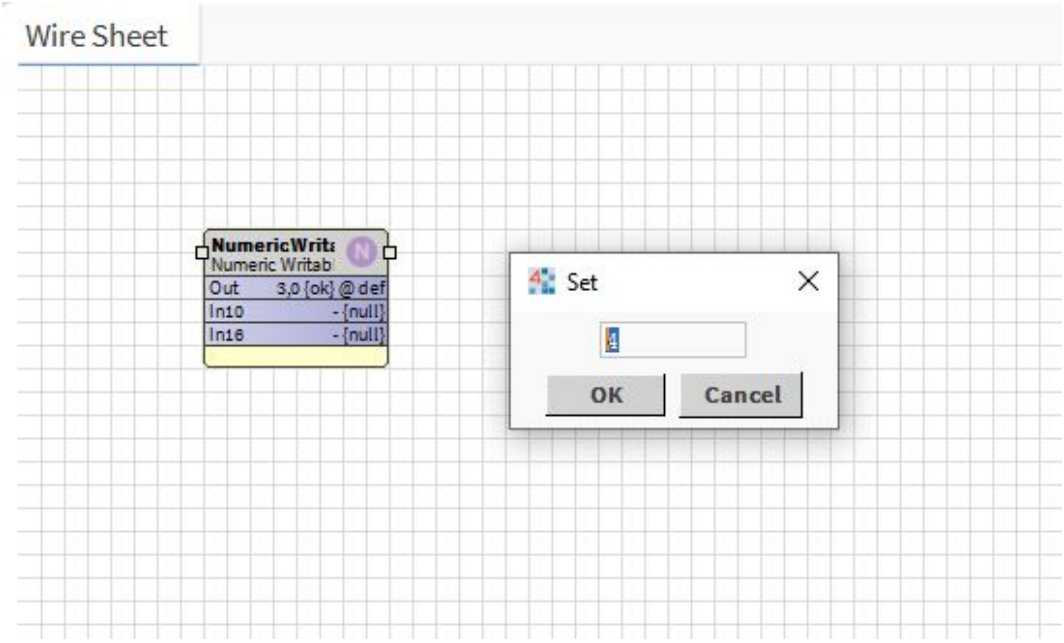


Figura 4.6: Settaggio

A questo punto ogni volta che il punto cambia, il servizio si aggiorna e mette in disposizione il valore predittivo calcolato rispetto alla stagione.

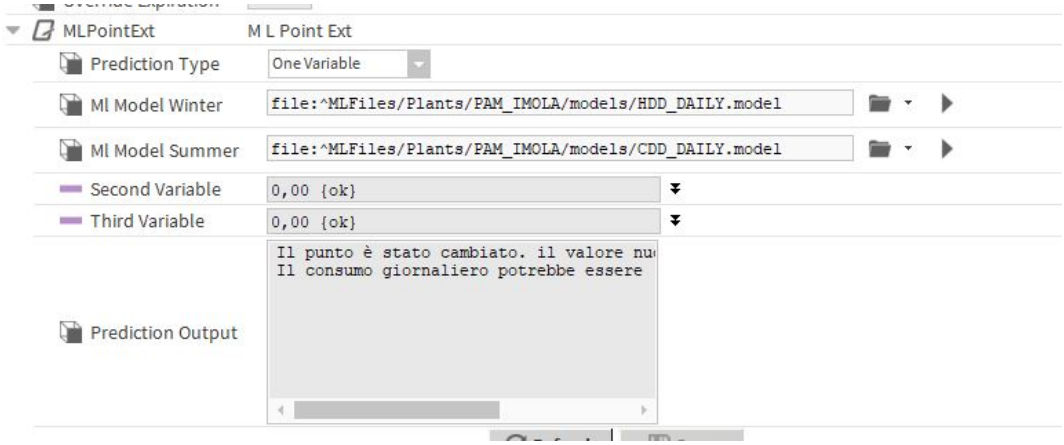


Figura 4.7: Il risultato di previsione

In questa figura potete vedere il risultato ottenuto nel formato stringa. Teniamo in conto che noi abbiamo provato con una sola variabile però potrebbe essere provato anche con più variabili.

CAPITOLO 5

TESTING

Per testare un modulo baja bisogna usare una libreria particolare di test che è stato esteso dalla libreria TestNg di java.

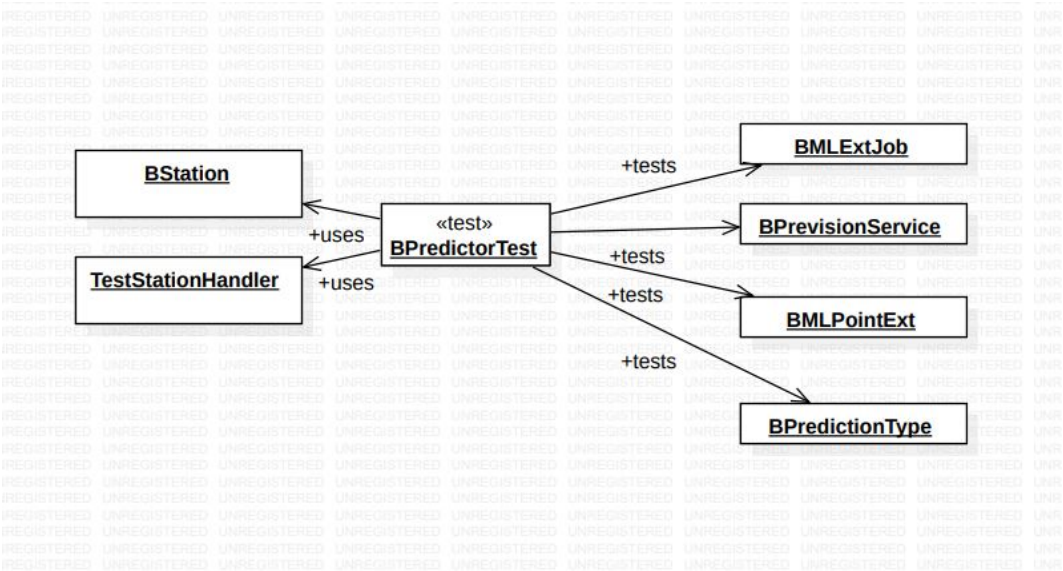


Figura 5.1: Object Diagram del Testing

Questa libreria ci permette di creare un'ambiente di niagara per fare un test. Cioè si può inizializzare un station a livello del codice e fare il test sulla nostra station. Per inizializzare uno station si può caricare un file di configurazione per avere uno station già preparato. È stato usato la

configurazione dello station usato nel capitolo *Mockups* ma senza il servizio, l'estensione del punto e altri file.

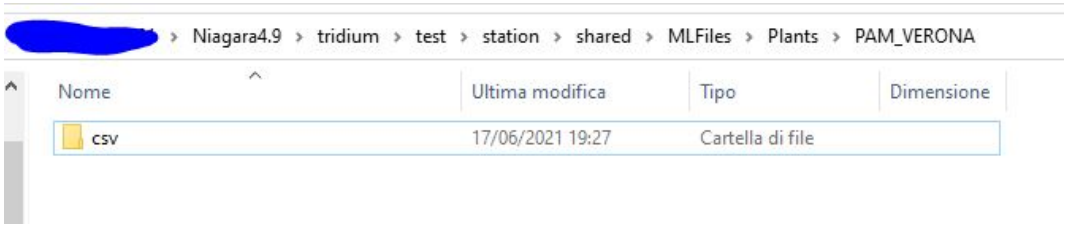


Figura 5.2: I file CSV nella fase di testing

Abbiamo solo i file CSV sulla nostra station. Abbiamo creato una sola classe di test per verificare le funzionalità principali. Questo include testare il benchmarking, la classe enumerato e l'estensione del punto.

```
/**
 * 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");
}
```

Figura 5.3: Sleep function

Nel test possiamo vedere che è stato messo alcuni ritardi (*sleep(3000)*). Questo è per aspettare che il thread finisca il suo lavoro.

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

Figura 5.4: Test Result

Come vediamo tutti i test sono andati a buon fine. È stato fatto tanti altri test a livello pratico che non sono stati messi in questo documentazione.

REFERENCES

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

ELENCO DELLE FIGURE