Università degli Studi di Napoli

Federico II

Facoltà di Scienze MM. FF. NN.

Corso di Laurea in Informatica

Tesi di Laurea Triennale Sperimentale

Automazione di Sistemi d'integrazione di algoritmi di *Machine Learning* nella web application di data mining astrofisico del progetto DAME

Relatori	**Candidato**
Prof. Massimo Benerecetti	**Sandro Riccardi**
Dott. Massimo Brescia	**Matricola: 566/1499**

Anno Accademico 2010-2011

Indice

Indice delle Figure

Indice delle Tabelle

Introduzione

Con il recente consolidamento del quarto paradigma della scienza moderna, denominato *e-science*, dopo teoria, sperimentazione e simulazione, l'ambiente scientifico ha di fatto accettato il ruolo ed il carattere multi-disciplinare delle nuove discipline emergenti, basati sul paradigma della X-Informatica (dove X sta per Bio, Astro, Geo, etc.), [5]. In pratica, il carattere *data-centrico* della scienza moderna ha evidenziato il ruolo cardine dell'*Information & Communication Technology* (ICT), onde poter organizzare, immagazzinare ed esplorare enormi quantità di dati sperimentali e/o simulati in modo efficiente. Quest'esplosione di dati è stata in parte dovuta all'evoluzione tecnologica degli strumenti di rilevazione, in parte all'intrinseca necessità di "osservare" dati eterogenei, cioè provenienti da diversi settori sperimentali, correlandoli tra loro per incrementare la conoscenza dei fenomeni ad essi legati. Quest'approccio si è rivelato particolarmente importante nel settore della ricerca astrofisica, dove la conoscenza dei fenomeni naturali richiede un'analisi multi-disciplinare e la raccolta di enormi quantità di dati da correlare in modo efficiente e affidabile, [11].

Lo scenario evidenziato implica l'urgente necessità di identificare ed applicare degli standard in grado di rappresentare, archiviare, navigare e esplorare i dati in modo uniforme, garantendo ed ottenendo l'interoperabilità richiesta tra le differenti discipline.
Tuttavia gli scienziati non possono e non vogliono diventare degli esperti nel campo dell'ICT, per poter soddisfare questa esigenza.
L'approccio alla standardizzazione dell'interoperabilità dei dati in Astrofisica è stato definito dal *Virtual Observatory* (VO), una federazione che raccoglie sotto standard comuni tutti gli archivi astronomici disponibili in tutto il mondo. L'utilità principale alla base di tale sforzo è che una volta che l'infrastruttura sarà completa, essa consentirà un nuovo tipo di approccio alla scienza che può essere solo minimamente immaginato, [2].
Il progetto DAME, *Data Mining & Exploration*, estende l'obiettivo di realizzare strumenti di basso livello e la definizione di standard, unendo software *service-oriented* con hardware *resource-oriented*, includendo l'implementazione di strumenti avanzati per l'esplorazione di *Massive Data Sets* (MDS), tra i quali il *Data Mining*.
Il Data Mining può essere semplicemente definito come l'estrazione di informazioni da dati a priori sconosciuti.

Le tecniche di *data mining* permettono quindi di identificare e descrivere delle strutture ordinate d'informazioni all'interno degli MDS, essenzialmente strutture nei *pattern* dei dati mescolati con il rumore, insieme alla capacità di predire il comportamento di un sistema complesso reale, utilizzando metodi di *Machine Learning*, ossia metodi che permettano ad una macchina di modificare il proprio comportamento autonomamente, al fine di poter ottenere migliori prestazioni in termini di risposta agli stimoli esterni, [12].

In un tale complesso contesto scientifico e tecnologico, il lavoro originale di questa tesi è consistito nell'ideare, progettare ed implementare un sistema d'integrazione automatica di modelli di *Machine Learning* all'interno di un'infrastruttura di calcolo specializzata in *data mining* su grossi moli di dati. Tale infrastruttura fa riferimento ad una *web application*, inserita come progetto nel più vasto programma DAME, denominata DAMEWARE[1] (*DAME Web Application REsource*).

Le prerogative principali di tale risorsa sono la scalabilità degli algoritmi di *data mining* rispetto alla mole dei dati processabili, lo sfruttamento implicito di grandi risorse di calcolo (su piattaforma ibrida *CLOUD/GRID*) e l'estrema portabilità, essendo accessibile tramite un semplice browser web, [3].

Il lavoro ha dunque riguardato l'elaborazione di procedure software per fornire all'utenza esterna la possibilità di integrare in modo semplice e affidabile i propri algoritmi di analisi ed esplorazione dati all'interno del *framework* DAMEWARE (*plugin*), senza doverne conoscere i meccanismi ingegneristici interni e ottenendo come importante valore aggiunto la possibilità di eseguire le proprie routine su risorse di calcolo distribuite senza particolari esperienze informatiche e/o sistemistiche. Una parte non secondaria del lavoro ha anche riguardato la progettazione e realizzazione di un software di *wrapping* di un modello di *data mining*, una rete neurale di tipo MLP (*Multi Layer Perceptron*) addestrato tramite le regole di apprendimento basate sulla tecnica Quasi Newton, ideato ed implementato all'interno del gruppo di lavoro del progetto DAMEWARE, per integrarlo nella *web application*. Ciò ha rappresentato l'importante fase preliminare a tutto il successivo lavoro, in particolare per studiare ed analizzare su un caso reale tutti i requisiti necessari alla procedura automatica di *plugin*.

Nel capitolo 1 verrà descritto il progetto DAMEWARE in tutte le sue componenti, focalizzando in modo particolare l'attenzione su quelle direttamente interessate da questo lavoro di tesi. Nel secondo capitolo saranno descritte tutte le fasi di integrazione del modello MLPQNA partendo dalla specifica dei suoi input e output, le operazioni sui file di input e su quelli di output, le modalità di

[1] http://dame.dsf.unina.it/beta_info.html

interazione del modello con le altre componenti del sistema, fino ad arrivare alla definizione delle classi implementate.

Nel capitolo 3 verranno brevemente evidenziate le principali differenze dei modelli esistenti all'interno della suite DAMEWARE al fine di poter astrarre la struttura di un generico *plugin*.

Il capitolo 4 descriverà la progettazione dell'integrazione automatica specificando in dettaglio tutti i requisiti che la procedura automatica dovrà soddisfare sfruttando la struttura generale di un *plugin* descritta nel precedente capitolo. La conseguente realizzazione della procedura è descritta nel quinto capitolo, dove saranno specificate le modifiche da apportare ai componenti del sistema esistente, la realizzazione della procedura automatica, fino alla descrizione dell'interfaccia grafica realizzata.

Il capitolo 6 permetterà di confrontare gli output prodotti con *plugin* relativi ai modelli esistenti con quelli prodotti dalla procedura automatica. In particolare i due modelli utilizzati in fase di *testing* sono MLPQNA e MLPGA, due varianti del noto modello *Multi Layer Perceptron*, [9], addestrato, rispettivamente, dal metodo del *Quasi Newton* e da algoritmi genetici.

1 Il progetto DAMEWARE

DAMEWARE è una *web application* basata su un insieme di strumenti di *Machine Learning* per la gestione e la manipolazione di dati in vari formati, specializzati per il *data mining* su *dataset* di grandi dimensioni (in particolare in ambito astrofisico).

Per *Machine Learning* s'intende una disciplina scientifica che si occupa della progettazione e dello sviluppo di algoritmi che permettono ai calcolatori di imparare a riconoscere modelli complessi e a prendere decisioni intelligenti a partire dai dati in esame. Le due principali tecniche di apprendimento sono : Supervisionato e Non Supervisionato, [13].

Nell'apprendimento *Supervisionato* (Figura 1) si ha a disposizione un insieme di dati, o di osservazioni, per i quali si conosca l'output corrispondente. Tale output può assumere uno o più valori chiamati classi o etichette. L'obiettivo finale è quello di fornire un certo livello di controllo utilizzato dall'algoritmo di apprendimento per regolare i parametri al fine di prendere decisioni in grado di predire le etichette o le classi per nuovi dati.

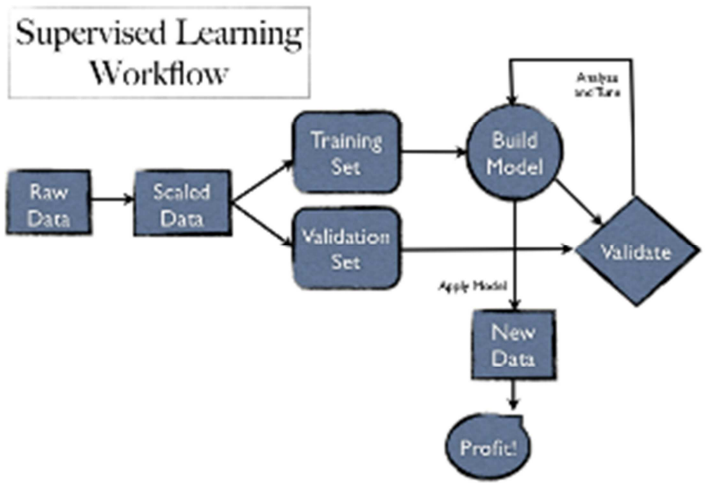

Figura 1 – Apprendimento Supervisionato – Schema teorico

Gli algoritmi basati sull'apprendimento non supervisionato cercano di raggruppare i dati input e di individuare delle classi rappresentative (*cluster*) dei dati stessi, facendo uso di metodi topologici o

probabilistici. Un tipo di rete non supervisionato è quindi in grado di interagire con i dati addestrando se stessa senza un "supervisore" che fornisca soluzioni in punti specifici nello spazio dei parametri.

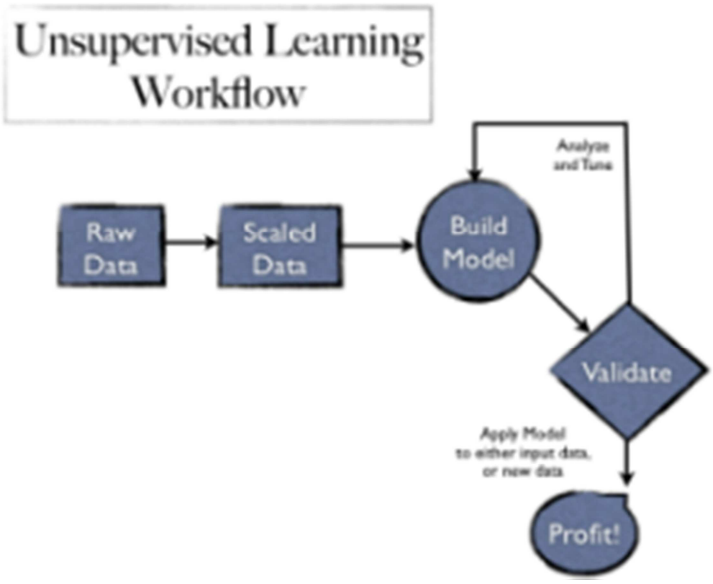

Figura 2 – Apprendimento Non Supervisionato – Schema teorico

Generalmente un generico modello di *Machine Learning* prevede quattro fondamentali funzionalità (casi d'uso):

- Addestramento (***TRAIN***): durante questa fase l'algoritmo riceve in ingresso un insieme di dati (*training set*) contenente per ogni particolare combinazione dei parametri di input, uno o più risultati osservati. Lo scopo di questa funzionalità è minimizzare l'errore commesso tra i risultati forniti dalla rete (gli output) e i dati osservati. L'errore generato non è misurato sul *training set*, ma su un diverso insieme di dati (*validation set*), al fine di evitare che la rete diventi talmente tanto aderente ai dati del *training set*, da risultare inutilizzabile al di fuori di questo insieme di dati, ossia perda la capacità di generalizzazione per la quale era stata progettata (*overfitting*).
- Valutazione dell'accuratezza (***TEST***): durante questa fase si verifica il livello di apprendimento della rete neurale supervisionata. In questa fase è passato in input all'algoritmo un insieme di dati non utilizzato nella fase di *TRAIN*, ma di cui si conosce l'output per valutare l'accuratezza dell'addestramento in termini di percentuale di risposte corrette.
- Esecuzione (***RUN***): questa fase consiste nell'utilizzo vero e proprio dell'algoritmo. Sono forniti in input dei dati di cui non si conoscono i risultati a priori. Sarà compito dell'algoritmo fornire l'output corrispondente, calcolato in base ai dati forniti nella fase di addestramento.

- Addestramento e valutazione consecutiva (***FULL***): è la successione delle fasi *TRAIN* e *TEST*. Consiste nel dare in input alla rete prima un *set* di dati di cui si conosce l'output per addestrare la rete e poi, tramite un *set* di dati, non utilizzato in precedenza ma di cui si conosce l'output, si valuta l'accuratezza della rete con la fase di *TEST*;

In particolare, gli algoritmi di apprendimento supervisionati si prestano alla risoluzione di problemi di classificazione e di regressione.
In tale contesto, per classificazione intendiamo una procedura in cui elementi campione di un determinato problema siano raggruppati in base alle loro intrinseche caratteristiche e attraverso un processo di addestramento del classificatore. Quest'ultimo è un sistema identificato da una funzione che trasforma lo spazio input in uno spazio costituito da categorie (classi), rappresentate attraverso etichette (*label*). Nel caso della varietà funzionale all'interno delle tecniche di *Machine Learning*, un problema di classificazione può assumere una delle seguenti forme:

a) Classificazione *crispy*: dato un campione input (*pattern*), il classificatore restituisce univocamente la corrispondente etichetta (*label*) di appartenenza ad una categoria;
b) Classificazione probabilistica: dato un campione input (*pattern*), il classificatore restituisce un array di probabilità, ciascuna associata ad una particolare categoria;

Naturalmente entrambe le tipologie si possono applicare a problemi a due classi (*binary classification*) o multi-classe (*multi-variate classification*)
Data la natura supervisionata della fase di addestramento di un sistema di classificazione, le prestazioni del classificatore possono essere successivamente misurate, mediante uso di un *set* di dati di *TEST*, in cui quindi campioni di dati extra sono sottoposti al processo di attribuzione di una classe. In tal caso s'innesca il cosiddetto processo di validazione dell'addestramento supervisionato, il cui esito stabilisce il grado di affidabilità e robustezza (generalizzazione) del classificatore.
I metodi di regressione permettono altresì di individuare relazioni funzionali analitiche tra i parametri dei campioni input, originariamente note solo per determinati gruppi di campioni di esempio del problema. In altre parole, permette di estrapolare relazioni matematiche a priori non note tra i dati del problema.
Tuttavia non esiste una definizione univoca di regressione. Ne esistono due particolarmente adatte al caso di sistemi supervisionati di *Machine Learning*:

a) Correlazione statistica: individuazione di una relazione tra i dati su base probabilistica, mediante attribuzione di una distribuzione statistica (calcolo dei momenti di primo e secondo ordine), alla funzione analitica, che meglio rappresenti l'associazione tra le coppie input-target a disposizione (*data table statistical correlation*);
b) *Fitting* polinomiale: individuazione di una funzione d'interpolazione non lineare che rappresenti la relazione tra i dati, il cui grado di approssimazione dipende dal numero di coppie input-target a disposizione per il problema in esame.

Da un lato, gli algoritmi supervisionati minimizzano gli errori nelle osservazioni di classificazione, mentre l'apprendimento non supervisionato non ha categorie (*label*) associate ai *pattern* di input. Gli algoritmi non supervisionati cercano di creare cluster di dati che siano intrinsecamente simili (*clustering*). In alcuni casi non si conosce necessariamente ciò che li rende simili, ma gli algoritmi sono in grado di trovare queste relazioni tra i dati, raggruppandoli in modo significativo. Mentre gli algoritmi supervisionati mirano a minimizzare l'errore di classificazione o regressione, gli algoritmi non supervisionati cercano di creare sottoinsiemi di dati i cui elementi di ogni singolo sottoinsieme siano il più possibile simili tra di loro, [2].

1.1 Architettura

DAME comprende diversi progetti, ideati per essere ospitati su differenti piattaforme di calcolo. Da un punto di vista funzionale, è formato da una serie di servizi ed applicazioni *web based*. In altre parole, la suite DAME è un'infrastruttura SOA (*Service Oriented Architecture*), perfettamente corrispondente al paradigma di calcolo *CLOUD* (Figura 3).
Il *CLOUD computing* descrive un nuovo modello di integrazione, utilizzo e fornitura di servizi per l'ICT , basati su internet, permettendo la fornitura di risorse dinamicamente scalabili e spesso virtualizzate sottoforma di servizi *web*, quindi di facile accesso, anche da semplici *smartphone* o *tablet pc*, [14].
Il termine *CLOUD* è usato come una metafora per internet, e si riferisce ad una "nuvola" computazionale, usata in passato per rappresentare la rete telefonica, e successivamente per rappresentare internet in un diagramma di una rete di calcolatori.
I tipici fornitori di *CLOUD computing* offrono delle comuni applicazioni scientifiche o commerciali *on line*, a cui si accede tramite un altro servizio *web* o un comune software come un *web browser*, mentre il software e i dati sono memorizzati sui *server*.

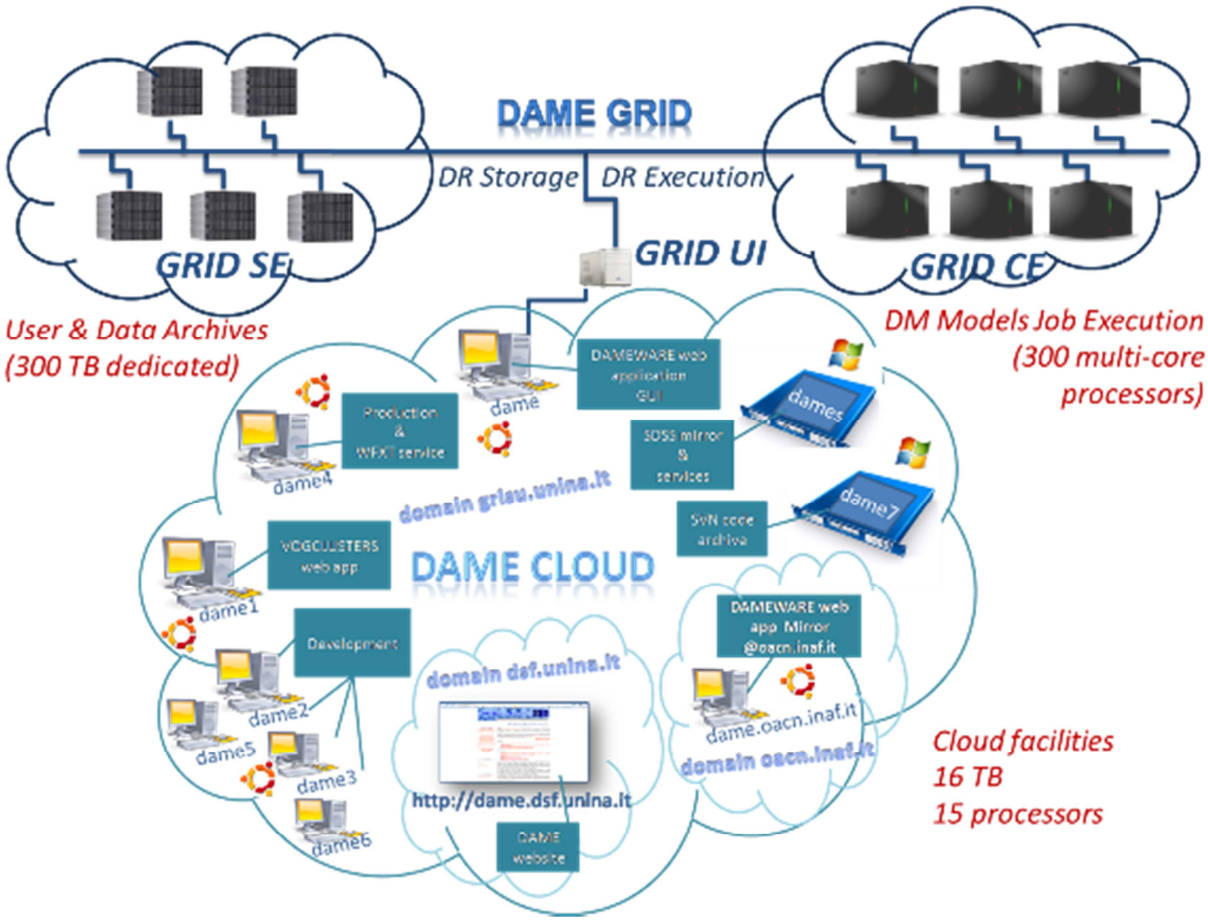

Figura 3 – Architettura Hardware *CLOUD/GRID* di DAME

Nell'attuale configurazione il progetto DAMEWARE, oggetto del presente lavoro di tesi, è basato su sei principali componenti software che interagiscono strettamente per svolgere le operazioni.
I componenti della suite sono:

- ***Front-End* (FE)**: front end della web-application, composta da una GUI (*Graphic User Interface*) realizzata con tecnologia GWT (*Google Web Toolkit*) e da un sistema *client*/*server* (basato su Java *servlet*) responsabile della gestione delle interazioni tra l'utente finale ed il sistema interno;
- ***Database Management System* (REDB)**: una libreria di classi, responsabile della gestione delle interazioni con la base dati (informazioni utente ed I/O degli esperimenti);
- ***Driver Management System* (DRMS)**: una libreria di classi che ha il compito di astrarre l'applicazione rispetto all'infrastruttura HW sottostante (*GRID, CLOUD, Stand-alone*);
- ***Framework* (FW)**: il crocevia dell'applicazione, un componente *restful stateless*, con il compito di gestire il flusso di comandi/dati tra tutti gli altri componenti;

- ***Data Mining Models* (DMM)**: modulo comprendente le librerie software (sottoforma di API) che implementano i vari algoritmi di *data mining* integrati nella *web application*;
- **DM*Plugin***: sub-componente (tra FW e DMM) responsabile dell'elaborazione e setup degli esperimenti utente, in cui è implementata l'associazione funzionalità-modello per ogni esperimento (attraverso l'interpretazione dei parametri scelti) e la relativa esecuzione con salvataggio dell'output e notifica all'utente dello stato dell'esperimento.

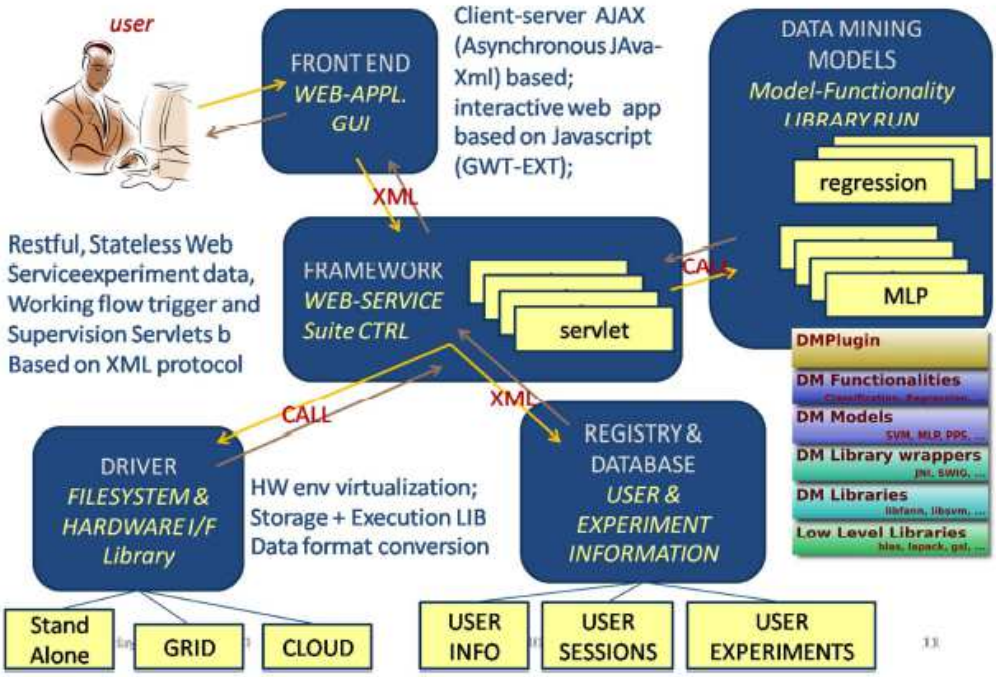

Figura 4 – Architettura Software

L'architettura di DAMEWARE (Figura 4) è stata progettata seguendo gli standard LAR (*Layered Application Architecture*), che prevede la realizzazione del sistema software con una struttura a strati, dove i differenti livelli comunicano tra loro tramite standard semplici e ben definiti.

L'applicazione web ha come principale requisito di progettazione di essere perfettamente compatibile con il paradigma Web 2.0, che ha portato all'avvento delle *web application*, come per esempio le applicazioni accessibili via *browser*, [15].

Una diretta evoluzione di questi tipi di strumenti è il *Rich Internet Application* (RIA), che consiste in applicazioni con le tradizionali caratteristiche interattive e di interfaccia, ma in grado di essere utilizzate tramite un semplice *web browser*, [16]. Uno dei principali vantaggi di questo strumento è, per esempio, la possibilità di eseguire delle applicazioni senza la necessità di installare le applicazioni sul computer dell'utente.

Gli strumenti RIA sono particolarmente apprezzati in termini di efficienza e velocità di esecuzione. Questo evidente vantaggio ha spinto a progettare DAMEWARE come una RIA.
Il Front End (FE) è il componente della web Application che interagisce direttamente con l'utente, basato sulla tecnologia RIA. Questa scelta è stata adottata per ottenere la massima efficienza tra l'interazione tra utente e applicazione, soprattutto per quanto riguarda l'esigenza di nascondere all'utente i pesanti aspetti di elaborazione dei MDS (*Massive Data Sets*), fornendo un accesso remoto rapido e semplice attraverso un web browser evitando qualsiasi installazione sulla macchina dell'utente. Tale scelta permette un facile ed efficiente sviluppo ed estendibilità del codice sorgente.
Una delle tecnologie impiegate per creare il FE come una RIA è AJAX che permette un uso asincrono delle tecniche *Javascript* tramite un'interfaccia basata su XML. Questo ottimizza il flusso delle informazioni tra utente e il *server* in termini di velocità e efficienza. AJAX ha cambiato la natura interna delle applicazioni web, la cui tendenza è quella di diventare disponibile on line fornendo l'uso trasparente delle potenti risorse di calcolo remote, situate in data center che non si possono comparare con le disponibilità hardware dell'utente, [17]. AJAX adotta un approccio basato su molte tecnologie considerate "vecchie", come HTML, CSS, DOM, e *Javascript*, aggiungendo l'uso di linguaggi lato *server* come per esempio le *Servlet*.
La GUI della *web application* DAMEWARE è attualmente l'unico modo in cui l'utente può accedere alla *web application*. Essa è basata su pagine web dinamiche gestite tramite GWT che fungono da interfaccia tra l'utente e le applicazioni, i modelli e le funzionalità per lanciare un esperimento scientifico.
La tecnologia GWT, [18], permette di scrivere applicazioni web utilizzando la piattaforma indipendente del linguaggio Java convertito in codice *Javascript* compatibile con il codice di un *browser* HTML[2].

1.2 Il paradigma Funzionalità – Modello

DAMEWARE offre una serie di funzionalità che possono essere scelte dall'utente per lanciare un esperimento. Tali funzionalità sono implementate come *plugin*, ossia come componenti non autonomi che interagiscano con un programma per ampliarne le funzionalità Ciascun *plugin* ha la necessità di comunicare con un modello di *data mining* durante il suo ciclo di vita. Ogni *plugin* è composto da una coppia Funzionalità – Modello.
La suite DAMEWARE offre varie funzionalità che possono essere scelte dall'utente per eseguire un esperimento. Esse sono rappresentate tramite i cosiddetti DM*Plugin* , in modo tale da permettere ad

[2] http://dame.dsf.unina.it/project.html

uno sviluppatore esterno di inserire nuove funzionalità all'interno della suite. I DM*Plugin* sono blocchi implementati come librerie condivise che sono caricate dinamicamente a *runtime* dal programma principale e che durante il ciclo di vita comunicano con uno dei modelli del DMM.

Il DM*Plugin* in generale ha il compito di creare una directory temporanea, creare un oggetto concernente la funzionalità richiesta (classificazione, regressione, *clustering*) e avviare l'esecuzione del caso d'uso richiesto (1).

Il DM*Plugin* prima di tutto crea due liste per ogni funzionalità, che rappresentano l'insieme di parametri necessari per ogni caso d'uso, e l'insieme dei file di input. Per ciascun elemento della lista dei parametri, il DM*Plugin* raccoglie informazioni come tipo, nome, precisione, possibili valori e una *flag* che stabilisce l'identità di un campo (opzionale/necessario).

Per ciascun elemento della lista dei file, il DM*Plugin* raccoglie nome, descrizione, URI (*Uniform Resource Identifier*) di locazione, formato ed una *flag* che stabilisce l'identità di un file (parziale/completo).

I file creati con l'esecuzione del modello sono contenuti in una directory temporanea creata dal DM*Plugin*. Oltre a questi file, tale directory contiene anche i file di input necessari all'esecuzione dell'esperimento e gli eventuali file di *plot* generati dal *Plugin*.

Il *DMPlugin* è un modulo del componente FW, progettato per fornire all'utente degli avanzati strumenti considerati come *plugin*. In questo senso è quindi possibile estendere la *web application* senza alterarne la struttura.

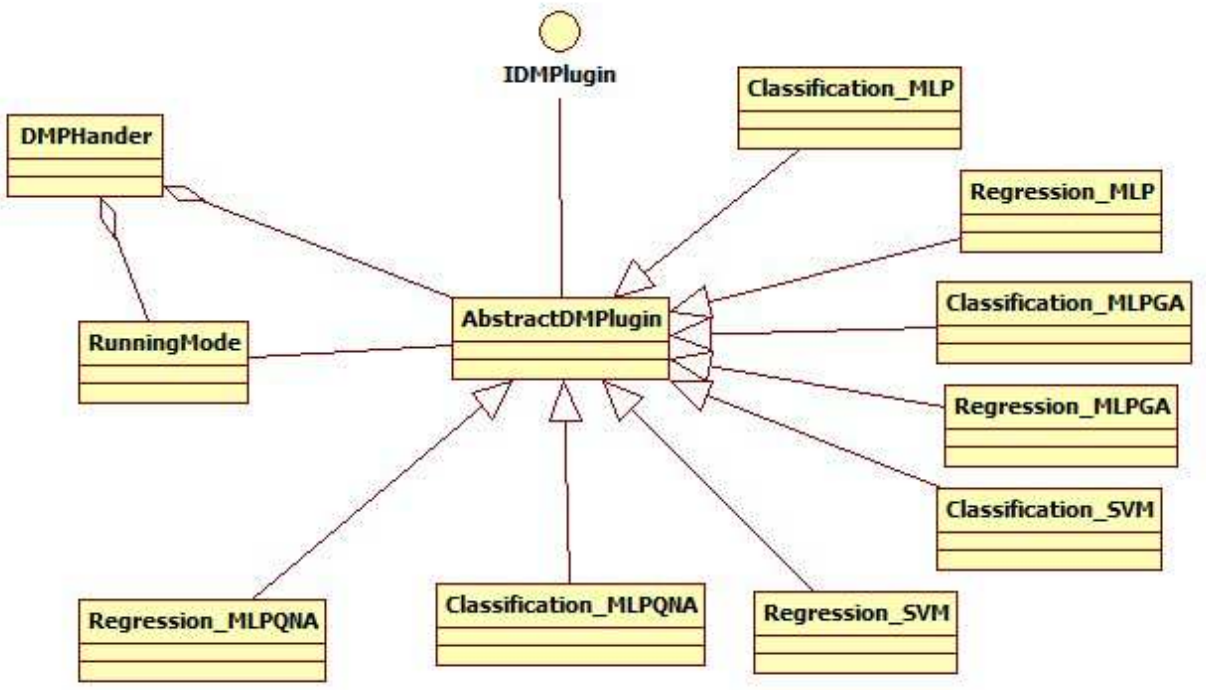

Figura 5 – Diagramma delle classi del DM*Plugin*

Ciascun DM*Plugin* (Figura 5) è specializzato da una classe astratta chiamata **AbstractDM*Plugin*** che contiene i metodi che ciascun DM*Plugin* deve implementare (*TRAIN*, *TEST*, *RUN* e *FULL*, relativi alle funzionalità di un generico modello di *Machine Learning*).

Quindi, il DM*Plugin* rappresenta l'implementazione di una funzionalità richiesta dal Front End, il *Framework* conosce solo la generica interfaccia del *plugin*.

Il fulcro del DM*Plugin* è il metodo che esegue, utilizzando il relativo modello di *data mining*, l'esperimento (DMM) .

Il componente DMM (Data Mining *Models*) racchiude le API degli algoritmi utilizzati per effettuare esperimenti di *data mining* all'interno della *web application* (modelli). Ogni funzionalità utilizza uno o più modelli per completare un esperimento. La libreria delle classi è stata realizzata in modo tale da favorire il riutilizzo del codice ed il polimorfismo (derivato dal paradigma orientato agli oggetti o *Object Oriented Programming*). Nella programmazione ad oggetti con il nome di polimorfismo si indica il fatto che lo stesso codice eseguibile può essere utilizzato con istanze di classi diverse, aventi una superclasse comune, [8].

L'architettura generale ha infatti evitato la duplicazione del codice tramite l'implementazione di un *pattern* strutturale, detto *Bridge Pattern* (1.2), che permette di separare l'interfaccia di una classe dalla sua implementazione in modo tale da renderle indipendenti (Figura 7).
Il componente *DMM*, quindi, è il *package* che implementa tutti i modelli di elaborazione dei dati e gli algoritmi presenti nella *web application.*

Il DMM è strutturato tramite pacchetti o librerie che comprendono i seguenti elementi:

- Librerie di modelli di Data Mining: *Multi Layer Perceptron* (addestrato da diverse regole di apprendimento, quali *Back Propagation*, *Quasi-Newton* e algoritmi genetici), *Support Vector Machine, Genetic Algorithm, Self Organizing Feature Map, Principal Probabilistic Surface, K-Means,* etc.;
- Strumenti per la visualizzazione e la manipolazione dei dati (*pre- e post-processing*);
- Strumenti statistici (matrice di confusione, *cross entropy*);
- Funzionalità di esplorazione: classificazione, regressione, *clustering*, riduzione delle dimensioni dello spazio dei parametri;
- Librerie personalizzate richieste dall'utente.

Il DMM è composto da diversi modelli divisi per categorie, mettendo quindi a disposizione strumenti per entrambi i paradigmi su citati del *Machine Learning* (supervisionato e non supervisionato).
Il componente DMM è stato progettato per essere modulare, quindi facile da estendere aggiungendovi nuove funzionalità e/o nuovi modelli di *data mining* e algoritmi, senza modifiche strutturali dei componenti interni.
Il DMM consiste internamente in un'interfaccia chiamata *DMMInterface,* che rappresenta un generico modello di *data mining*, integrabile nella *web application* DAMEWARE.
La soluzione progettuale originaria, per riutilizzare il codice e ottenere una combinazione coerente e ottimizzata delle classi appartenenti ai modelli e alle loro funzionalità, consiste nell'impiego di un noto *design pattern*, il *Bridge Pattern*.
Il *Bridge Pattern* è un particolare *design pattern* (modello di progettazione) strutturale che permette di separare un'astrazione di una classe (la sua interfaccia) dalla sua implementazione, evitando la duplicazione del codice. In tal modo si può sfruttare l'ereditarietà tra le classi per farne evolvere l'interfaccia o l'implementazione in modo separato.
La Figura 6 mostra il diagramma UML del *Bridge Pattern*.

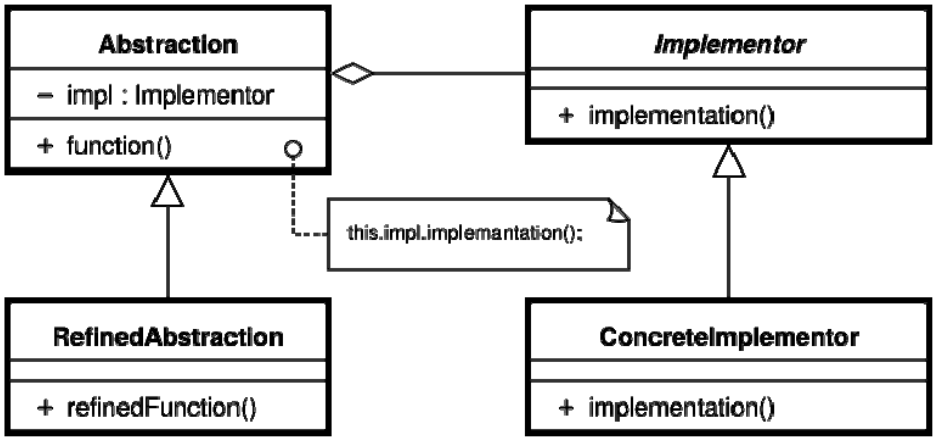

Figura 6 – *Bridge pattern*

La classe astratta "**Abstraction**" rappresenta il livello funzionale del componente, che sarà estesa dalle classi che rappresentano le funzionalità dell'intera suite (*Classification*, *Regression*, *Clustering*, ecc.); mentre la classe astratta "**Implementator**" rappresenta il livello di implementazione dei modelli; infine la classe "**ConcreteImplementator**" rappresenta l'implementazione di un modello.

Il livello funzionale è rappresentato dalle funzionalità, mentre il livello d'implementazione è rappresentato dai modelli.

Nel caso concreto si ha un'interfaccia che generalizza un modello supervisionato chiamato DMMInterface, in cui ogni modello ha una classe con responsabilità mirate esclusivamente a quel particolare modello, mettendo a disposizione tutte le funzionalità del modello stesso.

Quindi, Il livello funzionale consiste in due classi astratte denominate "**Supervised**" e "**Unsupervised**". La classe **Supervised** è estesa da due classi concrete denominate "**Classification**" e "**Regression**".

Il livello di implementazione consiste in una interfaccia denominata "**DMMInterface**" che rappresenta la generalizzazione dei modelli implementati dalla suite.

L'interfaccia prevede tre metodi corrispondenti alle 3 possibili funzionalità su cui si basano i modelli di apprendimento automatico supervisionato (1), tali modelli ricevono in input dei parametri e dei file:

- *Public void TRAIN (DMMFieldParam[] dmmparam, DMMFileParam[] File_in) throws IOException*: è la firma esatta del metodo che corrisponde all'addestramento del modello generico;

- *Public void TEST (DMMFieldParam[] dmmparam, DMMFileParam[] File_in) throws IOException*: è la firma esatta del metodo che corrisponde a valutazione dell'apprendimento del modello sulla base dei parametri di input;
- *Public void RUN (DMMFieldParam[] dmmparam, DMMFileParam[] File_in) throws IOException*: è la firma esatta del metodo che corrisponde all'utilizzo del modello;

L'interfaccia "**DMMInterface**" è implementata dalle classi relative ai modelli.

La Figura 7 mostra il diagramma UML dell'intero pacchetto DMM.

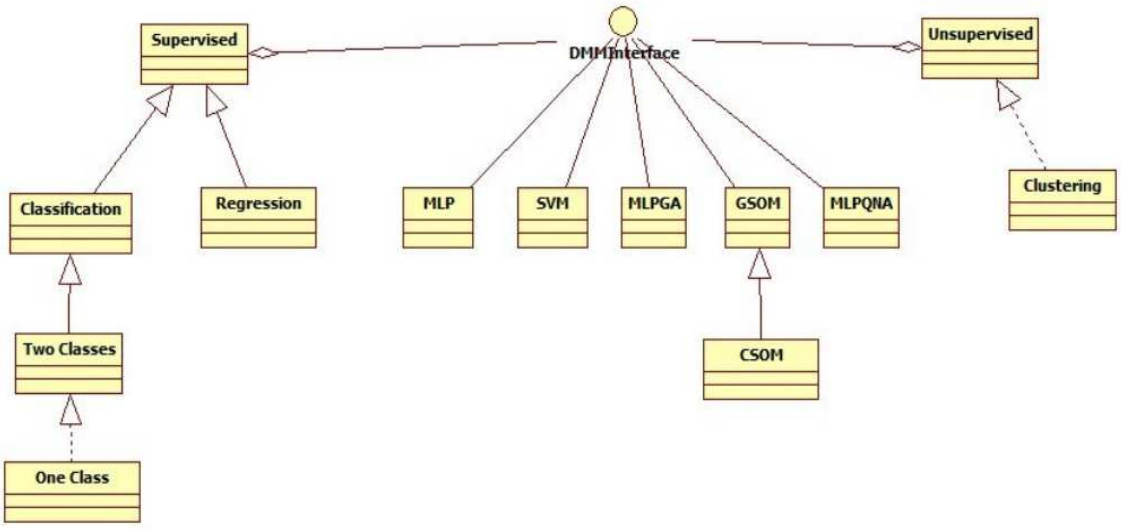

Figura 7 – DMM Struttura interna

Il pacchetto DMM comprende anche due classi non ancora citate:

- **Visualization:** responsabile del *plot* dei file passati come input;
- **Statistics:** responsabile della generazione di statistiche, analizzando le colonne di un *dataset* passato come input;

Il *pattern* strutturale *Bridge* è, nell'attuale architettura, sfruttato in particolare per separare il livello di astrazione dal livello d'implementazione delle associazioni funzionalità-modello, in modo cioè da poterli gestire indipendentemente. Il *Bridge Pattern* è stato dunque concepito per evitare la duplicazione del codice, ottimizzando quindi il debug e l'utilizzo delle risorse.

2 Integrazione di un modello di *Machine Learning* in DAMEWARE

Per poter acquisire la necessaria competenza circa la struttura interna dell'applicazione web e per poter estrapolare in modo preciso e circostanziato i principali requisiti della procedura di *plugin*, una parte preliminare del lavoro è stata dedicata all'operazione di *wrapping* manuale di un modello di *Machine Learning* già in dotazione al gruppo di lavoro di DAME. Tale lavoro, oltre a permettere la piena comprensione dell'architettura sottostante, ha anche avuto come valore aggiunto l'integrazione di un nuovo modello di *data mining* all'interno della *web application*.

2.1 Wrapping manuale di un modello

Di seguito verrà descritta la procedura di *wrapping* manuale, prendendo ad esempio un modello di *data mining* noto come *Multi Layer Perceptron* (MLP), [9], addestrato con il metodo del *Quasi Newton* (MLPQNA), [10]. Per tale modello, originariamente implementato in C++, una porzione del presente lavoro di tesi è consistita nel costruire un sistema di *wrapping* in Java, fase preliminare e propedeutica all'integrazione standardizzata di un qualsiasi modello di *Machine Learning* nella piattaforma DAMEWARE.

2.2 Il modello MLPQNA di data mining

MLPQNA è un modello di reti neurali di tipo *feed-forward*.
Per *rete neurale (artificiale)* s'intende un sistema di elaborazione delle informazioni che cerca di simulare, all'interno di un sistema informatico, il funzionamento del sistema nervoso biologico, costituito da un gran numero di cellule nervose (neuroni) collegati tra loro tramite una complessa rete. Il comportamento intelligente emerge dalle numerose interazioni tra i vari neuroni interconnessi. Alcuni di questi neuroni ricevono informazioni dall'esterno (neuroni input), altri emettono all'esterno informazioni (neuroni di output), altri ancora comunicano con altri neuroni all'interno della rete (neuroni nascosti). Il modo in cui, le informazioni in input corrispondano a quelle in output per ciascun neurone, è ottenuto tramite un processo di apprendimento (1).

Il termine *feed-forward* è usato per indicare che gli impulsi all'interno della rete sono propagati attraverso i suoi strati di neuroni, ad esempio dai neuroni del livello input, fino a quelli del livello output, passando per uno o più livelli intermedi (*hidden layers*).

I neuroni della rete sono quindi organizzati in livelli, il segnale input, trasmesso a tutti i neuroni del livello input, è usato per stimolare i successivi livelli, fino a quello output.

L'output di ciascun neurone è ottenuto tramite una funzione di attivazione, applicata alla somma pesata dei suoi input. Possono essere applicate forme differenti di funzioni di attivazione, dalla più semplice, quella lineare, fino a quella di tipo sigmoidale.

Il numero di livelli intermedi rappresenta il grado di complessità dello spazio delle soluzioni in cui si ricerca la soluzione migliore. Ciò che differenzia in genere le reti neurali MLP è l'algoritmo supervisionato di apprendimento usato per addestrare la rete. Infatti, in ogni caso la rete deve essere prima addestrata: alla rete sono passati delle coppie di valori che rappresentano l'input e l'output desiderato; viene dunque propagato il segnale fino al livello output, ove il risultato prodotto è confrontato con l'output atteso, al fine di valutare l'apprendimento. La valutazione può avvenire attraverso il calcolo del classico MSE (*Mean Square Error*) o su base statistica (*cross entropy*), [19]. Se tale valore rimane sotto a una soglia, predefinita dall'utente in fase di setup del modello, l'errore viene retro-propagato attraverso gli strati della rete, modificando opportunamente i pesi delle connessioni tra i neuroni, seguendo la direzione ottimale del gradiente o hessiano (come appunto nel caso del metodo *Quasi Newton* del modello preso in esame), dell'errore.

Al termine della fase di apprendimento, la rete dovrebbe essere in grado non solo di riconoscere l'output corretto per ciascun input usato nella fase di *training*, ma anche raggiungere un certo grado di generalizzazione al fine di ottenere un output corretto anche per gli input non usati in precedenza.

Dopo la fase di *training*, la rete può essere vista come una *black box*, in grado di eseguire una particolare funzione la cui forma analitica non era a priori nota.

Al fine di compiere un buon addestramento della rete, i dati devono poter descrivere una gran varietà di campioni, cioè devono costituire un valido campione statistico del problema in esame.

Il modello MLPQNA si presta alla risoluzione di problemi di classificazione e di regressione.

Il modello MLPQNA, così come tutti gli algoritmi di *Machine Learning* (1), comprende le funzionalità *TRAIN TEST* e *RUN*.

Tutti i parametri necessari all'esecuzione di un esperimento sono passati al modello tramite linea di comando.

2.2.1 Caso d'uso *TRAIN*

Per il caso d'uso *TRAIN*, il modello accetta in ingresso la seguente sequenza di parametri:

1. **Funzionalità**: Valore numerico di tipo intero che indica il tipo di funzionalità da eseguire (classificazione o regressione), il modella accetta il valore 10 per la classificazione e 20 per la regressione;
2. **Caso d'uso**: Valore numerico di tipo intero che indica il caso d'uso da eseguire (*TRAIN*, *TEST*, *RUN*). Per il caso d'uso *TRAIN*, questo parametro assume valore 3;
3. **Decadimento**: Valore numerico di tipo reale. Il valore consigliato per questo parametro è 0.001;
4. **Restarts:** Valore numerico di tipo intero, rappresenta il numero massimo di cicli di *training* eseguiti, il valore consigliato è 20;
5. **Peso minimo di *step*:** Valore numerico di tipo reale, rappresenta un criterio di arresto, l'algoritmo si arresta se il peso dello *step* è più piccolo del valore rappresentato da questo parametro. Il valore consigliato è 0.01;
6. **Iterazioni massime:** Valore numerico di tipo intero, rappresenta un criterio di arresto dell'algoritmo. Se il numero di iterazioni supera il valore di questo parametro, l'algoritmo si arresta. Il valore consigliato è 1500;
7. **Numero di neuroni input:** Valore numerico intero, stabilisce il numero di colonne del *dataset* di input;
8. **Numero di neuroni output:** Valore numerico intero, stabilisce il numero di colonne di target del *dataset* in input; Nel caso della regressione il valore di questo parametro vale sempre 1.
9. **Numero di livelli intermedi:** Valore intero che stabilisce il numero di livelli nascosti della rete, i possibili valori sono 0, 1 e 2;
10. **Numero di neuroni del primo livello nascosto:** Valore numerico intero, se il parametro 9 assume valore 0, questo parametro non viene considerato;
11. **Numero di neuroni del secondo livello nascosto:** Valore numerico intero, se il parametro 9 assume valore 0 o 1, questo parametro non viene considerato;
12. ***Flag* per la *cross entropy*:** Usato solo nel caso della classificazione, nel caso della regressione questo parametro non viene considerato. I possibili valori sono 0 e 1. Se assume valore 0, non viene utilizzata la *cross entropy*;
13. **Nome del dataset in input:** E' importante che questo file sia in formato CSV (*Comma Separated Values*), non contenga *header*, tantomeno *carriage return***;**

14. ***Flag* per la *cross validation*:** Valore numerico intero che può assumere solo i due valori 0 e 1. Il valore 1 indica l'esecuzione del *TRAIN* con una sequenza di validazione;
15. **Valore per la *cross validation*:** Valore intero, se il parametro 14 è 0, questo parametro non viene considerato. Il valore consigliato per questo parametro è 5;
16. **Modalità di calcolo per la matrice di confusione:** *Flag* che indica la modalità di calcolo della matrice di confusione.
17. **Modalità di scelta dei pesi della rete:** Valore numerico intero. Questo parametro può assumere solo due possibili valori: 702 indica che i pesi della rete sono caricati da un file, 704 indica che i pesi sono inizializzati in modo casuale;
18. **Nome del file dei pesi:** Se il parametro 17 assume valore 704, questo parametro indica il nome del file contenente i valori dei pesi della rete, se il parametro 17 assume valore 702, questo parametro non viene considerato;

Un esempio di esecuzione del caso d'uso *TRAIN* è il seguente:
./mlpqna 10 3 0.001 10 0.01 1000 7 1 1 15 0 1 *dataset*/agn_7_stat.txt 1 10 1 702 none

Al termine dell'esecuzione del caso d'uso *TRAIN*, il modello produce in output una serie di file contenuti in una sottodirectory a partire dalla *path* di esecuzione del modello. Tali sottodirectory sono differenti a seconda che l'esperimento esegua una classificazione o una regressione:

- Per la classificazione: ./ex/cl/;
- Per la regressione: ./ex/re/;

Il modello non genera automaticamente le sottodirectory citate, sarà quindi necessario accertarsi della loro esistenza prima dell'avvio di un esperimento.
I file di output prodotti dal caso d'uso *TRAIN* sono:

1. **errorLog.txt:** file di report di errori contenente informazioni relative a condizioni non corrette o eccezioni che causano la terminazione non corretta del modello. Questo file non viene creato se il programma termina in modo corretto;
2. ***train*Log.txt:** file con informazioni sulla configurazione dell'esperimento;
3. ***train*PartialError.txt:** file contenente i valori parziali per ciascuno *step* di *train*ig dell'algoritmo. Questo file è composto da tre colonne, la prima indica lo *step* (l'indice del ciclo di *training* corrente), la seconda indica il numero di iterazioni per lo *step* corrente, la terza indica l'errore di *training*;

4. ***train*edWeights.txt:** pesi finali della rete alla fine della fase di *TRAIN*. **frozen_train_net.txt:** valori dei nodi interni della rete alla fine della fase di *TRAIN*. Questo file può essere usato come input nella fase di *TEST* o *RUN*;
5. ***trainTest*OutLog.txt:** valori di output calcolati alla fine della fase di *TRAIN* con i rispettivi target. Questo file può essere utilizzato per valutare gli output della rete per ciascun input;
6. **output.txt:** versione semplificata del file *trainTest*OutLog.txt;
7. ***trainTest*ConfMatrix.txt:** matrice di confusione calcolata alla fine della fase di *TRAIN*, utilizzando i valori presenti nel file *trainTest*OutLog.txt.

2.2.2 Caso d'uso *TEST*

Per il caso d'uso *TEST*, il modello accetta in ingresso la seguente sequenza di parametri:

1. **Funzionalità**: Valore numerico di tipo intero che indica il tipo di funzionalità da eseguire (classificazione o regressione), il modella accetta il valore 10 per la classificazione e 20 per la regressione;
2. **Caso d'uso**: Valore numerico di tipo intero che indica il caso d'uso da eseguire (*TRAIN*, *TEST*, *RUN*). Per il caso d'uso *TRAIN*, questo parametro assume valore 4;
3. **Numero di neuroni input**: Valore numerico intero, stabilisce il numero di colonne del *dataset* di input;
4. **Numero di neuroni output:** Valore numerico intero, stabilisce il numero di colonne di target del *dataset* in input. Nel caso della regressione il valore di questo parametro è sempre 1;
5. **Numero di livelli intermedi:** Valore intero che stabilisce il numero di livelli nascosti della rete, i possibili valori sono 0, 1 e 2;
6. **Numero di neuroni del primo livello nascosto:** Valore numerico intero, se il parametro 9 assume valore 0, questo parametro non viene considerato;
7. **Numero di neuroni del secondo livello nascosto:** Valore numerico intero, se il parametro 9 assume valore 0 o 1, questo parametro non viene considerato;
8. **Nome del dataset in input:** E' importante che questo file sia in formato CSV (*Comma Separated Values*), non contenga *header* tantomeno *carriage return***;**
9. **Modalità di calcolo per la matrice di confusione:** *Flag* che indica la modalità di calcolo della matrice di confusione.
10. **Nome del file dei pesi;**

11. **Nome del file di configurazione della rete:** Nome del file con i valori interni della rete assunti alla fine della fase di *TRAIN*;

Un esempio di esecuzione del caso d'uso *TRAIN* è il seguente:
./mlpqna 10 4 7 1 1 15 0 dataset/TEST_agn_ridotto.txt 1 /ex/cl/trainedWeights.txt /ex/cl/frozen_TRAIN_net.txt

anche in questo caso, come per il caso d'uso *TRAIN*, il modello produce in output una serie di file contenuti in una sottodirectory a partire dal percorso (*path*) di esecuzione del modello e dipendono dal tipo di funzionalità richiesta (classificazione o regressione).
I file di output prodotti dal caso d'uso *TEST* sono:

1. **errorLog.txt:** file di report di errori contenente informazioni relative a condizioni non corrette o eccezioni che causano la terminazione non corretta del modello. Questo file non viene creato se il programma termina in modo corretto;
2. ***test*OutLog.txt:** valori di output calcolati alla fine della fase di *TEST* con relativi target.
3. **output.txt**: file con gli output della rete e relativi target, versione semplificata del file *test*OutLog.txt, per uso interno.
4. ***test*ConfMatrix.txt:** matrice di confusione calcolata alla fine della fase di *TEST*, utilizzando i valori presenti nel file *trainTest*OutLog.txt.

2.2.3 Caso d'uso *RUN*

Per il caso d'uso *RUN*, il modello accetta in ingresso la seguente sequenza di parametri:

1. **Funzionalità**: Valore numerico di tipo intero che indica il tipo di funzionalità da eseguire (classificazione o regressione), il modella accetta il valore 10 per la classificazione e 20 per la regressione;
2. **Caso d'uso**: Valore numerico di tipo intero che indica il caso d'uso da eseguire (*TRAIN*, *TEST*, *RUN*). Per il caso d'uso *TRAIN*, questo parametro assume valore 5;
3. **Numero di neuroni di input**: Valore numerico intero, stabilisce il numero di colonne del *dataset* di input;

4. **Numero di neuroni di output:** Valore numerico intero, stabilisce il numero di colonne di target del *dataset* in input. Nel caso della regressione il valore di questo parametro è sempre 1;
5. **Numero di livelli intermedi:** Valore intero che stabilisce il numero di livelli nascosti della rete, i possibili valori sono 0, 1 e 2;
6. **Numero di neuroni al primo livello nascosto:** Valore numerico intero, se il parametro 9 assume valore 0, questo parametro non viene considerato;
7. **Numero di neuroni al secondo livello nascosto:** Valore numerico intero, se il parametro 9 assume valore 0 o 1, questo parametro non viene considerato;
8. **Nome del dataset in input:** E' importante che questo file sia in formato CSV, non contenga *header*, tantomeno *carriage return***;**
9. **Modalità di calcolo per la matrice di confusione:** *Flag* che indica la modalità di calcolo della matrice di confusione.
10. **Nome del file dei pesi;**
11. **Nome del file di configurazione della rete:** Nome del file con i valori interni della rete assunti alla fine ella fase di *TRAIN*;

Un esempio di esecuzione del caso d'uso *RUN* è il seguente:
./mlpqna 10 5 7 1 1 15 0 dataset/TEST_agn_ridotto.txt 1 /ex/cl/edWeights.txt /ex/cl/frozen_TRAIN_net.txt

I vincoli sulle sottodirectory nelle quali sono creati i file di output sono gli stessi per i casi d'uso *TRAIN* e *TEST*.
I file di output prodotti dal caso d'uso *TEST* sono:

1. **errorLog.txt:** file di report di errori contenente informazioni relative a condizioni non corrette o eccezioni che causano la terminazione non corretta del modello. Questo file non viene creato se il programma termina in modo corretto;
2. ***run*OutLog.txt:** valori di output calcolati alla file della fase di *RUN* con relativi target.

2.3 Interazione con le altre componenti

Nell'esecuzione di un esperimento con la *web application* DAMEWARE l'utente, tramite il Front End (FE), seleziona la funzionalità scegliendo tra classificazione o regressione. Il modulo FE

mostra i casi d'uso disponibili per la funzionalità scelta. Con la scelta da parte dell'utente del caso d'uso, il componente FE mostra l'elenco dei parametri richiesti per avviare l'esperimento.

L'elenco dei parametri richiesti per ciascun modello è contenuto all'interno di un file XML che ha proprio il compito di permettere al componente FE di mostrare graficamente l'insieme dei parametri richiesti per un modello. Esiste quindi un file XML adibito a questo scopo per ciascuna coppia Funzionalità – Modello (1.2).

Con l'inserimento dei parametri, il FE crea un file in formato XML contenente le informazioni sui parametri da passare al *Framework* per dare il via all'esperimento (Figura 8).

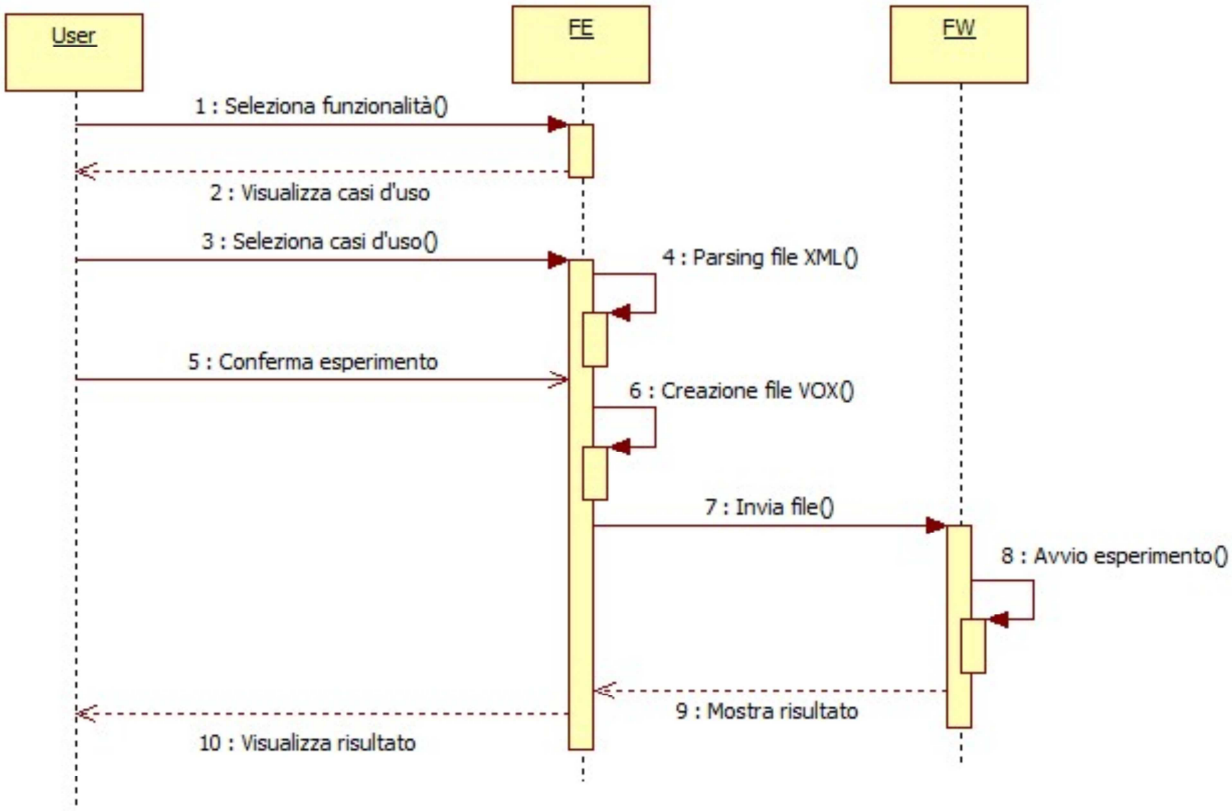

Figura 8 - Interazione utente – *Framework*

Per avviare un esperimento il componente del DM*Plugin* utilizza un oggetto del *package* DMM.

Il componente del DMM ha il compito di analizzare i parametri in ingresso, riordinarli secondo le esigenze (*parsing*) e passarli all'eseguibile per avviarne l'esecuzione.

Nell'esecuzione dell'esperimento, il FW effettua il *parsing* del file XML ricevuto dal modulo FE e imposta un ID relativo all'esperimento tramite una comunicazione con il REDB. Se l'operazione è andata a buon fine, il *Framework* interroga nuovamente il componente REDB per ottenere un identificativo dell'esperimento. Successivamente viene caricata la classe del DM*Plugin* relativa alla funzionalità richiesta dall'utente e vengono istanziate tutte le strutture dati necessarie a tenere

traccia di tutti i parametri necessari (parametri di input, file di input e di output). Il *Framework* avvia l'esperimento utilizzando l'oggetto DM*Plugin* istanziato. Tale oggetto crea una directory temporanea e univoca dedicata all'esecuzione dell'esperimento; effettua il *parsing* dei parametri; avvia l'esperimento e infine effettua tutte le operazioni sugli output prodotti. Al termine della sua esecuzione i file di output prodotti sono registrati nella base di dati (Figura 9).

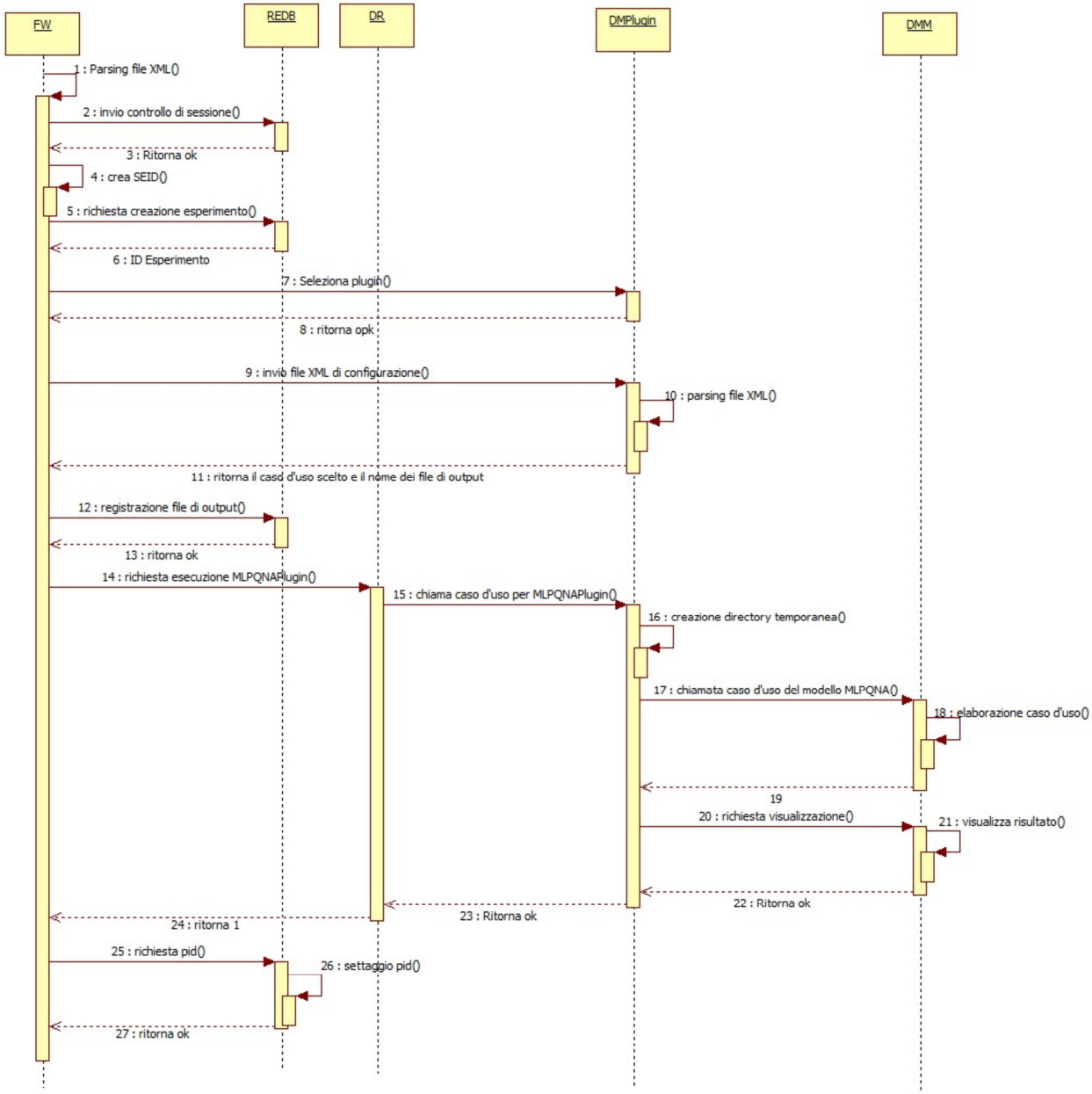

Figura 9 - Interazione FW – DMM

La Figura 10 mostra il ciclo di vita di un esperimento.

Un oggetto del componente DM*Plugin*, nel momento in cui viene istanziato dal FW, è nello stato di "*Created*". Con il metodo configure() della classe AbstactDM*Plugin*, l'oggetto DM*Plugin* passa

allo stato di "*Configured*". Il metodo *run*() permette la selezione del caso d'uso desiderato per il relativo modello, con la chiamata a questo metodo, l'oggetto DM*Plugin* passa allo stato di "*Running*". Al termine dell'esecuzione del modello, l'oggetto passa allo stato di "*Completed*".

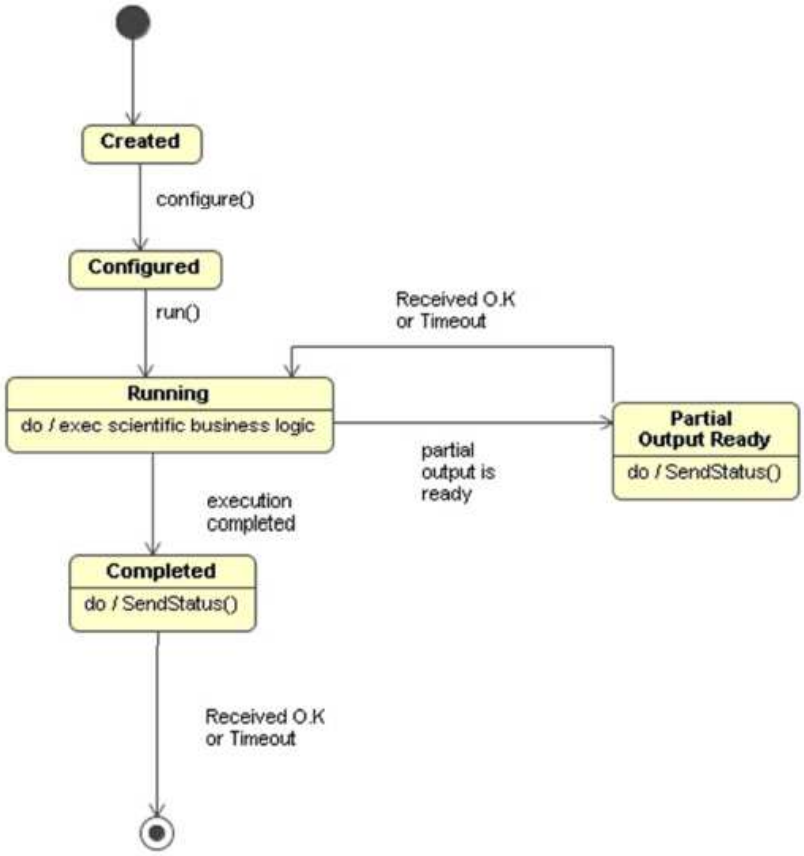

Figura 10 – Diagramma di stato del DM*Plugin*

2.4 Le componenti per il modello MLPQNA

Il compito di avviare l'esecuzione di una funzionalità associata ad un *plugin* è assegnato ad una *servlet* del componente FW.

Una *servlet* è un oggetto che accetta una richiesta da un *client* e crea una risposta basata su questa richiesta.

L'utente, tramite il file XML d'interfaccia con il componente FE, inserisce i parametri con cui avviare un esperimento, il FE quindi, invia i parametri inseriti alla *servlet* del FW che carica il l'opportuno *plugin* (Figura 11).

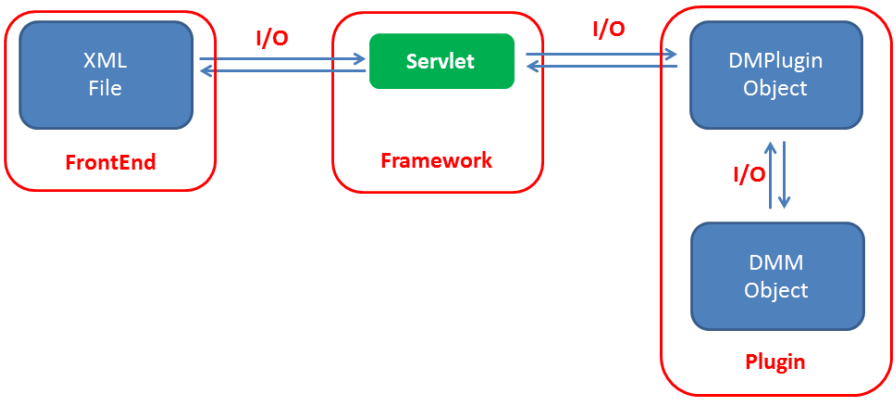

Figura 11 - Comunicazione con la servlet del FW

Quindi, al fine in integrare il modello MLPQNA e le sue rispettive due funzionalità all'interno della *web application* DAMEWARE, sono stati creati i) un file per il componente DMM, ii) due file per il componente DM*Plugin*, rappresentanti le due funzionalità associate al modello (classificazione e regressione), e iii) due file XML, relativi alle due medesime funzionalità.

Il modello MLPQNA, come specificato in precedenza, accetta in input un file *dataset* in formato CSV senza *header* e senza *carriage return* finale.
E' quindi compito del DM*Plugin* convertire i file dei *dataset* input nel formato CSV.
Sia nel caso della classificazione, che della regressione, durante l'esecuzione del caso d'uso *TRAIN*, il modello MLPQNA produce il file *train*PartialError.txt, composto da tre colonne: lo *step* di *training* (l'indice del ciclo di *training* corrente), il numero di iterazioni per lo *step* corrente e l'errore di *training*. E' utile rappresentare graficamente tali informazioni (seconda e terza colonna), utilizzando i metodi messi a disposizione dalla classe **Visualization** del componente DMM. Solo per la funzionalità di regressione invece, per quanto riguarda il caso d'uso *TEST*, è richiesto il *plot* delle colonne 2 e 3 del file di output output.txt. tali colonne rappresentano rispettivamente la colonna di output (per la regressione sempre una) e la colonna dei target.
Il caso d'uso *FULL* prevede una serie di operazioni aggiuntive da effettuare dopo l'esecuzione della fase di *TRAIN* e prima dell'esecuzione della fase di *TEST*.
Sia il caso d'uso *TRAIN* che quello di *TEST* producono in output due file denominati output.txt e errorLog.txt (quest'ultimo generato solo in casi particolari di errore).
Per evitare che nella fase di *TRAIN* vengano sovrascritti i file output.txt e errorLog.txt prodotti dal *TRAIN*, è necessario rinominarli prima di eseguire il caso d'uso *TEST*.

Un altro importante aspetto che caratterizza il caso d'uso *FULL* è il passaggio di alcuni file di output prodotti dal *TRAIN*, come input per la fase di *TEST*.

In modo più specifico durante la fase di *TRAIN* sono prodotti i file *train*edWeights.txt e output.txt descritti in precedenza. Questi file vengono passati in input durante la fase di *TEST* rispettivamente in corrispondenza dei parametri in posizione 10 e 11 della linea di comando (2.2.2).

La suite DAMEWARE prevede degli standard per quanto riguarda il nome dei file di output. Per ciascuno di essi il formato definito è <Nome modello>_<Caso d'uso>_<nome del file>.<estensione>.

I file di output prodotti dal modello MLPQNA (2.2) non rispettano tale formato; inoltre alcuni file necessitano di essere rinominati in quanto i nomi ad essi relativi prodotti dal modello possono risultare poco chiari all'utente.

Ad esempio, il file output.txt , prodotto nelle fasi di *TRAIN* e di *TEST*, rappresenta l'output effettivo della rete e quindi, per lo standard definito, i rispettivi file output, per i casi *TRAIN* e *TEST* saranno denominati MLPQNA_*TRAIN*_output.txt e MLPQNA_*TEST*_output.txt.

Le funzionalità di classificazione e regressione attualmente sono relative solo ai modelli SVM, MLP e MLPGA, già implementati e disponibili all'interno della *web application*. L'integrazione del modello MLPQNA richiede la modifica delle classi implementate nei file **Classification**.java e **Regression**.java del *package* DMM, Tali classi infatti, implementano tre metodi (*TRAIN*, *TEST* e *RUN*), che in base ad un indice passato come parametro, identificano il corrispondente metodo da eseguire del modello appropriato.

L'identificativo di un modello è un valore intero (attualmente il valore 1 per il modello SVM, 2 per MLP, e 3 per il modello MLPGA). Il modello MLPQNA sarà dunque individuato dal valore intero successivo, cioè 4.

Per il modello MLPQNA sono state implementate due classi java, ciascuna relativa ad una funzionalità, per le quali sia possibile utilizzare tale modello. Queste due classi sono state denominate **Classification_MLPQNA** e **Regression_MLPQNA**.

La Figura 12 mostra uno schema rappresentativo della struttura interna del file **Classification_MLPQNA**.java, relativo alla funzionalità di classificazione. La struttura della classe **Regression_MLPQNA** è analoga a quella mostrata per la funzionalità di classificazione. La struttura dei metodi *trainRun*, *testRun* e *runRun* è simile a quella rappresentata per *fullRun*, che è composto dall'unione delle operazioni di *trainRun* e *TestRun*.

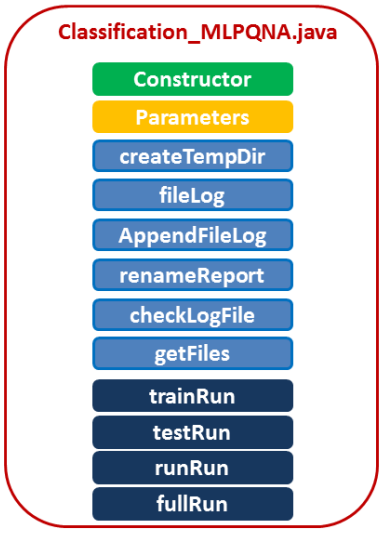

Figura 12 - Struttura della classe Classification_MLPQNA

In entrambe le classi è stato implementato un costruttore che ha lo scopo di inizializzare tutte le strutture dati necessarie all'esecuzione del DM*Plugin*.
Le operazioni svolte dal costruttore sono:

- Creare un oggetto di tipo *Functionality* della classe **DataTypes** che raccoglie informazioni relative a nome ed e-mail dell'implementatore del *plugin*; la versione, il riferimento dove trovare la documentazione relativa al *plugin* o al modello, il nome del modello, la data di creazione e le funzionalità supportate;
- Per ciascuna funzionalità implementata, sono istanziate le strutture dati necessarie a tenere traccia dei parametri, file di input e di output utilizzati dal *plugin*.

La Figura 13 mostra la struttura del costruttore per la classe **Classification_MLPQNA**:
Per ciascun caso d'uso è implementato un blocco come quello mostrato. I blocchi "*Input Fields for UseCase*", "*Input files for UseCase*", "*Output files for UseCase*" e "*Partial files for UseCase*" sono descritti nel paragrafo 3.1.

```
public Classification_MLPQNA(IDrStorage dr, String ip, Integer experimentId, String seid, String userName, Integer sessionId) throws Exception{
        super(dr,ip,experimentId,seid,userName,sessionId);
        Functionality f=new Functionality("Sandro Riccardi","0.1",
        "http://voneural.na.infn.it","Classification_MLPQNA.java","Classification_MLPQNA","12-01-27",
        "sandroriccardi@hotmail.com","Classification",true,true,true,true);
        HashMap<String,RunningMode> runningModes=new HashMap<String,RunningMode>();

        HashMap<String,DMMFieldParam> fields; HashMap<String,DMMFileParam> iFiles;
        HashMap<String,DMMOutputFileParam> oFiles; HashMap<String,DMMOutputFileParam> pOFiles;
        List<Comparable> l; Map<String,Tag> tags; fields=new HashMap<String,DMMFieldParam>(); iFiles=new
        HashMap<String,DMMFileParam>(); oFiles=new HashMap<String,DMMOutputFileParam>(); pOFiles=new
        HashMap<String,DMMOutputFileParam>();

        Input Fields for UseCase
        Input Files for Use Case
        Output files for Use Case
        Partial files for Use Case

        runningModes.put("Test", new RunningMode(fields,iFiles,oFiles,pOFiles,"Test","http://voneural.na.infn.it",0));

        setInformations(f);
        setRunningModes(runningModes);
}
```

Figura 13 – Il costruttore della classe Classification_MLPQNA

La Figura 14 mostra la struttura del metodo *fullRun* per la classe **Classification_MLPQNA**. Anche in questo caso, la descrizione dettagliata dei blocchi è rimandata al paragrafo 3.1.

```
protected void fullRun(){
        UseCase="FULL";
        dirExp=getFiles(UseCase);
        FileLog(uri_out + dirExp + "/mlpqna_" + UseCase + ".log", "START");
        Classification c= new Classification();
        c.train(3, inputField, inputFiles);
        Plot images
        Renaming train output files
        Updating input files list for test
        c.test(3, inputField, newInputFiles);
        FileLog(uri_out + dirExp + "/mlpqna_" + UseCase + ".log", "END");
        Appending files to log
        checkLogFile(uri_out + dirExp + "/mlpqna_" + UseCase+".log");
        Renaming and reporting files
}
```

Figura 14 – Il metodo *fullRun* della classe Classification_MLPQNA

I metodi *trainRun*, *testRun*, *runRun* e *fullRun* hanno il compito di eseguire rispettivamente di casi d'uso *TRAIN*, *TEST*, *RUN* e *FULL* usufruendo dei seguenti metodi dei quali è riportata la firma:

- **private File createTempDir (String useCase)**: ha il compito di creare una directory temporanea univoca all'interno del quale il *plugin* andrà a inserire i file di input necessari per l'esperimento. La directory conterrà anche tutti i file prodotti dal modello, sui quali sarà possibile effettuare altre operazioni prima di spostarli nella directory dell'utente. La directory prodotta sarà denominata seguendo la sintassi <Nome del modello>_<useCase>_<anno><mese><giorno><ora><minuti><secondi> così da garantire l'univocità della directory;
- **private void FileLog (String fileName, String status)**: Crea il file di *log* di un esperimento. Tale metodo è eseguito due volte, prima e dopo l'esecuzione dell'esperimento, e permette di poter quantificare e registrare la durata dell'esperimento. Il file *fileName* conterrà quindi, al termine delle due esecuzioni di questo metodo, le informazioni relative all'inizio e alla fine dell'esperimento;
- **private void AppendFileLog (String fileName1, String fileName2)**: Generalmente un modello di *data mining* produce in output un file con informazioni sull'esito dell'esperimento. Le informazioni presenti in questo file vengono inserite all'interno del file *log*. Questo metodo inserisce il contenuto del file *fileName2* all'interno del file *fileName1*, in *append*. Per il modello MLQNA;
- **private boolean RenameReport(DMMOutFileParam outParam, String name)**: Ha il compito di rinominare il file *outParam* prodotto dal modello ed informare il *Framework* che si occuperà di spostare il file dalla directory temporanea a quella dell'utente;
- **private void checkLogFile()**: Un esperimento non andato a buon fine può non generare in output tutti i file richiesti. Tale metodo ha il compito di controllare il contenuto del file *log*. Se il file *log* contiene solo le informazioni relative all'ora di inizio e fine dell'esperimento, senza nessuna ulteriore informazione, allora indica, tramite un messaggio all'interno del file *log*, l'occorrenza di un errore nell'esecuzione dell'esperimento;
- **private File getFiles (String useCase)**: ha il compito di inizializzare le strutture dati che tengono traccia dei file di input e di output relativi al modello di *Machine Learning*. In particolare, per i file input, se il formato del file non corrisponde a quello accettato dal modello, si occupa di convertirlo. Il metodo restituisce il nome della directory temporanea creata (Figura 15);

```
private java.io.File getFiles(String outputFileName) {
                RunningMode r = this.getChosenMode();
                String name = r.getName();
                IDrStorage dr = getDr();
                inputFiles = new DMMFileParam[r.getInputFiles().values().size()];

                Output files for plot
                 dirExp = createTempDir(outputFileName);
                 java.io.File tempDir = new java.io.File(dirExp+subdir);
                 tempDir.mkdirs();

                for (DMMFileParam f : inputFiles) {
                      switch (f.getName().charAt(0)) {

                      Input files uri
                      }//END SWITCH
                       if (!f.getName().equals(" ")) {
                         dr.download(f.getUri(), uri);

                      Translate input files
                      } else {//END IF
                                f.setUri(uri);
                      }
                } //END FOR
                for (DMMOutputFileParam f : outputFiles) {
                      switch (f.getName().charAt(0)) {
                       Output files uri
                      } //END SWITCH
                }END FOR
}
```

Figura 15 – Il metodo getFiles della classe Classification_MLPQNA

Al fine di completare il *wrapping* del modello MLPQNA nella suite DAMEWARE, è stata creata una classe java denominata **MLPQNA**.java.

La Figura 16 mostra uno schema rappresentativo della struttura interna del file **MLPQNA.java**. La figura mostra l'insieme delle fasi per il metodo *train*. I metodi *test* e *run* contengono le medesime operazioni relative, ovviamente al corrispondente caso d'uso.

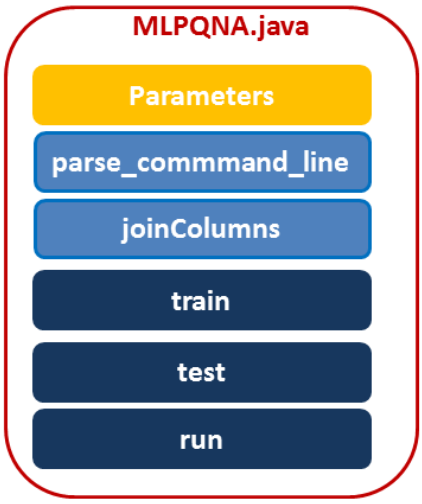

Figura 16 – Schema generale della classe MLPQNA

La classe **MLPQNA** ha il compito di caricare ed inizializzare tutti i parametri passati dal *plugin*, assegnando i valori di *default* ai parametri per i quali non sia stato stabilito un valore dal *Plugin* (scelta dell'utente), ed eseguire un esperimento tramite un comando lanciato utilizzando la *shell* linux.

La suite DAMEWARE prevede un'interfaccia chiamata "**DMMInterface**", che rappresenta un generico modello di *Machine Learning* (1). Quindi, la classe **MLPQNA**.java implementa l'interfaccia **DMMInterface**, che prevede i seguenti metodi:

- **public void *train*(DMMFieldParam[] dmmparam, DMMFileParam[] File_in)**: fase di addestramento (Figura 17);
- **public void *test*(DMMFieldParam[] dmmparam, DMMFileParam[] File_in)**: fase di verifica e validazione dell'addestramento;
- **public void *run*(DMMFieldParam[] dmmparam, DMMFileParam[] File_in)**: esecuzione del modello già addestrato;

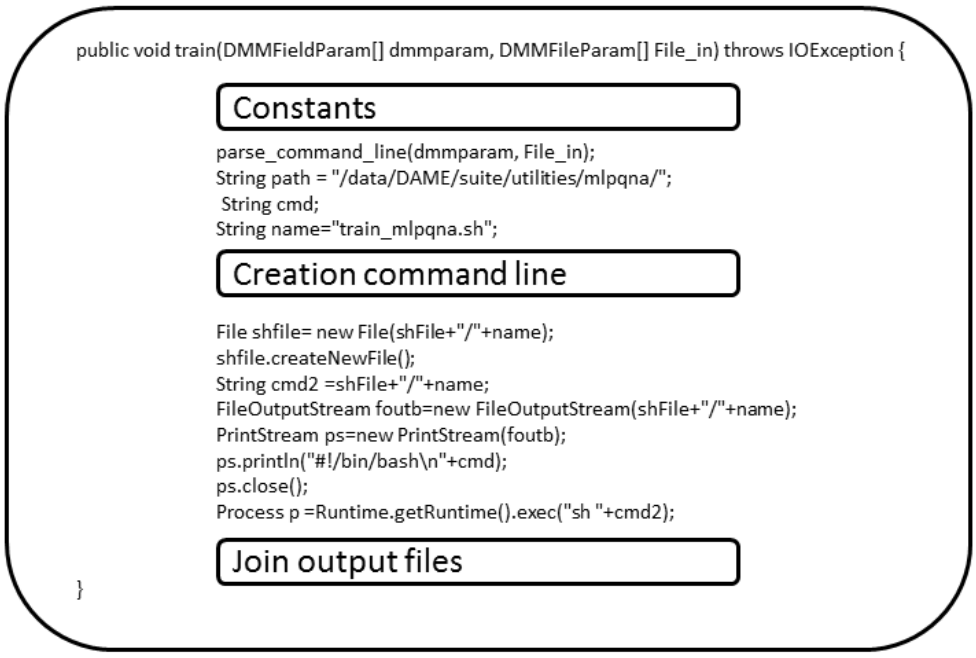

Figura 17 – Il metodo *TRAIN* della classe MLPQNA

Il componente "DMMInterface" non prevede un metodo relativo al caso d'uso *FULL*, in quanto tale caso d'uso, come detto in precedenza, è l'esecuzione in sequenza dei casi d'uso *TRAIN* e *TEST*. Quindi, l'implementazione del caso d'uso *FULL* è assegnata solo al componente DM*plugin* (2.4).

L'implementazione di questi tre metodi prevede, per quanto riguarda il modello MLPQNA, la creazione di un comando *shell* che avvia l'esecuzione del modello di *Machine Learning* con i parametri passati dal componente DM*Plugin*.
Il componente del DMM non prevede l'implementazione di alcun costruttore.
Oltre all'implementazione dei metodi previsti nell'interfaccia **DMMInterface** (*TRAIN*, *TEST* e *RUN*), la classe **MLPQNA**.java prevede i seguenti metodi aggiuntivi:

- **private void parse_command_line (DMMFieldParam[] dmmparam, DMMFileParam[] File_in, int model)**: ha il compito di ricevere le informazioni relative ai parametri di input e ad i file di input passati dal componente del DM*Plugin*, nonché di assegnare i valori di *default* ai parametri per i quali non sia stato ricevuto un valore (Figura 18);
- **private void *join*Columns()**: Il modello MLPQNA produce in output dei file, tra cui uno in particolare, contenente informazioni sugli output prodotti e sui relativi target per ogni *pattern* in input. Il compito di questo metodo è di aggiungere a questo file i valori delle colonne del *dataset* di input, in modo da rendere omogenea e completa l'informazione per l'utente.

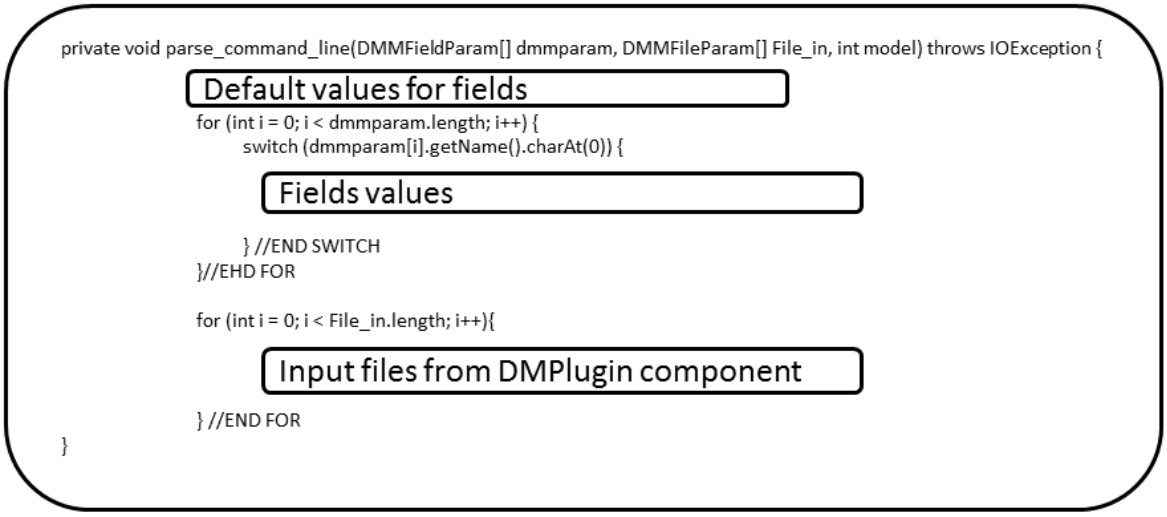

Figura 18 – Il metodo parse_command_line della classe MLPQNA

Come descritto nel paragrafo 2.3, il modello si interfaccia con l'utente per mezzo del componente FE, utilizzando un file XML che specifica l'insieme di tutti i parametri di input, dei file di input, e dei file di output per ciascun caso d'uso.
Per ciascuna funzionalità implementata dal modello MLPQNA è stato quindi creato un file XML:

Classification_MLPQNA.xml per la classificazione e Regression_MLPQNA.xml per la regressione, contenuti rispettivamente nelle directory /data/DAME/suite/*plugin*s/Classification_MLPQNA/ e /data/DAME/suite/*plugin*s/Regression_MLPQNA/.
Lo standard seguito da questi file XML, e da tutti gli altri file XML che interfacciano gli altri *plugin*, è quello messo a disposizione dal *Virtual Observatory*. Tale formato prende il nome di VOTable ed è stato definito per permettere principalmente lo scambio dei dati in forma tabellare[3].

La procedura d'integrazione di un nuovo *plugin* all'interno della suite prevede la registrazione di alcune informazioni ad esso relative all'interno del componente REDB.
Il componente REDB (Figura 19), oltre che gestire tutte le informazioi relative agli utenti, alle loro sessioni di lavoro, ai vari dati di input e di output di ogni esperimento e ai dati temporanei e finali dei vari processi lanciati dall'utente, ha anche il compito di tenere traccia dei *Plugin*s installati nella web aplication e dei formati dei *dataset* che ognuno di essi accetta in input.
A tal fine, l'integrazione di un nuovo *Plugin* all'interno della suite richiede, quindi, anche l'aggiunta di informazioni ad esso relative, all'interno della base di dati.
In particolare, è necessario aggiungere informazioni sul nuovo *Plugin* nelle entità FUNCTIONALITY e FILE FORMAT.
La tabella FUNCTIONALITY raccoglie le seguenti informazioni:

- *name*: Nome del *plugin* che verrà visualizzato nell'interfaccia grafica del Front End;
- *docsLink*: *link* in cui è possibile trovare la documentazione relativa al *plugin*;
- *ver*: versione del *plugin*;
- *creationDate*: data di creazione del *plugin*;
- *ownName*: nome del proprietario del *plugin*;
- *ownMail*: e-mail del proprietario del *plugin*;
- *status*: *flag* che stabilisce se il *plugin* è diponibile per l'utilizzo oppure no;
- *domain*: Classificazione, Regressione, *Clustering* o *Feature Extraction*;
- *fileName*: nome del file java del *plugin*;
- trainSupported: *Flag* che stabilisce se è implementato il caso d'uso *TRAIN*;
- testSupported: *Flag* che stabilisce se è implementato il caso d'uso *TEST*;
- runSupported: *Flag* che stabilisce se è implementato il caso d'uso *RUN*;
- fullSupported: *Flag* che stabilisce se è implementato il caso d'uso *FULL*.

[3] http://www.ivoa.net/cgi-bin/twiki/bin/view/IVOA/IvoaVOTable

Per ciascun *Plugin* è richiesta una sola *query* di inserimento dei dati all'interno della tabella FUNCTIONALITY.
La tabella FILE FORMAT raccoglie le seguenti informazioni:

- *formatName*: formato del file;
- *owner*: *plugin* a cui è associato il formato del file;
- *extension*: estensione del file;
- *description*: una breve descrizione;
- *isEditable*: *flag* che indica se il file sia editabile con i *tool* messi a disposizione dalla *web application*.

Per ciascun *Plugin* è richiesta una *query* di inserimento dei dati all'interno della tabella FILE FORMATS per ogni formato di *dataset* in input previsto dalla suite.

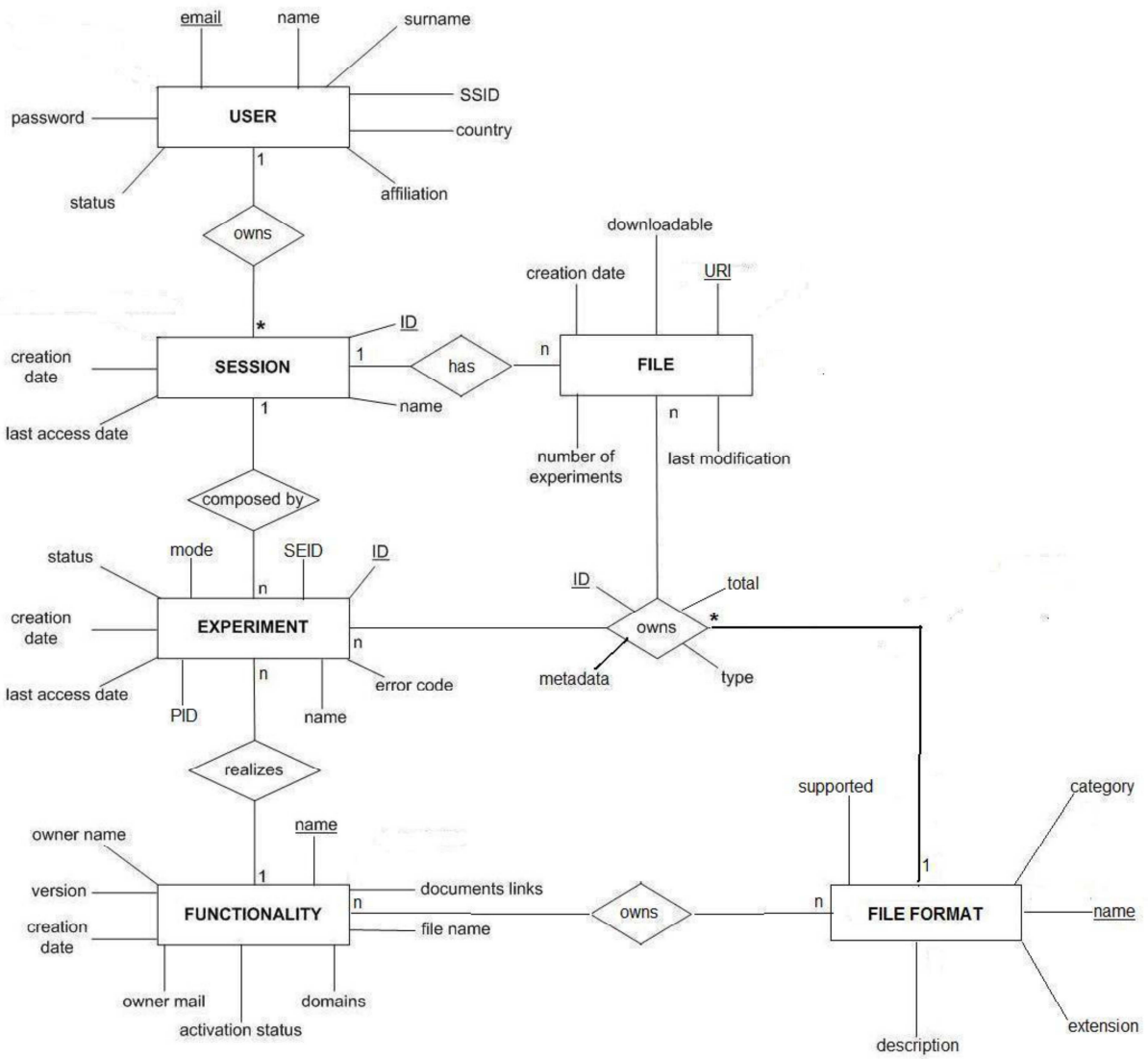

Figura 19 – Diagramma Entità-Relazione del componente REDB

3 L’integrazione automatica

Col passare del tempo, i servizi messi a disposizione da DAMEWARE sono utilizzati sempre con maggiore frequenza da utenti che intendono eseguire esperimenti di *data mining*.
Il costante utilizzo di tale applicazione e l’esigenza di aggiungere sempre nuovi modelli di *data mining* all’interno della suite, ha portato alla necessità di creare uno strumento software che permetta ad un utente di poter integrare all’interno della *web application* un proprio algoritmo di *Machine Learning* e renderlo disponibile all’intera comunità. Affinché tale obiettivo sia possibile, è scaturita la necessità di standardizzare la creazione di un elemento DM*Plugin* e di stabilire dei vincoli da rispettare per aggiungere nuove funzionalità alla *web application* in modo semplice.
A tal proposito, partendo da una breve analisi dei modelli presenti attualmente all’interno della suite DAMEWARE (3.1) , è stata definita una struttura generale di un *plugin* e delle relative classi da implementare, rispettando il più possibile la struttura dei plugin già implementati nella suite.

3.1 Confronto dei modelli presenti nella suite

Il modello MLPQNA è stato già descritto nel capitolo 2. In questo capitolo verranno descritti i modelli attualmente presenti all’interno della suite DAMEWARE focalizzando l’attenzione sulle operazioni da effettuare sugli input e output nonché sulla struttura dei *plugin* ad esso relativi.
Al momento dell’inizio del presente lavoro, la *web application* DAMEWARE espone i seguenti modelli di *data mining*:

- MLP addestrata con l’algoritmo di *Back Propagation* e con algoritmi genetici (MLPGA);
- SVM (*Support Vector Machine*);

Il modello MLP prevede, come il modello MLPQNA, il passaggio dei parametri e dei file di input tramite la linea di comando. Il componente del DMM, oltre ai metodi *TRAIN TEST* e *RUN* che eseguono il comando *shell* per l’esecuzione dell’esperimento, implementa un metodo dedicato al *parsing* dei parametri passati dal componente DM*Plugin*. Il *dataset* input deve essere in formato FITS. Il modello MLP è associato alle funzionalità di classificazione e regressione.
Per quanto riguarda le operazioni sui file di output prodotti, per la classificazione, il caso d’uso *TRAIN* effettua solo il *plot* del file dell’errore di *training*, il caso d’uso *TEST* crea una matrice di

confusione a partire dal file degli output delle rete prodotto. Inoltre unisce (*join*) le colonne del *dataset* di input al file relativo agli output della rete. Anche il caso d'uso *RUN*, analogamente a quello di *TEST*, al termine dell'esecuzione di un esperimento, effettua una concatenazione (*join*) tra le colonne del *dataset* di input e le colonne del file degli output della rete. Per quanto riguarda il caso d'uso *FULL*, oltre alle operazioni definite nelle fasi di *TRAIN* e *TEST*, richiede di passare alla fase di *TEST*, fornendo in input un file generato dalla fase di *TRAIN* relativo ai pesi della rete. Nel caso della regressione, invece, il caso d'uso *TRAIN* prevede il *plot* del file dell'errore di *training*. I casi d'uso *TEST* e *RUN*, al termine dell'esecuzione di un esperimento, effettuano la concatenazione tra le colonne del *dataset* di input e le colonne del file degli output della rete. Per quanto riguarda il caso d'uso *FULL*, oltre alle operazioni definite nelle fasi di *TRAIN* e *TEST*, richiede di passa
re alla fase di *TEST*, in input, un file generato dalla fase di *TRAIN* relativo ai pesi della rete.
Un altro modello presente all'interno della suite è denominato MLPGA. Esso, a differenza dei modelli MLPQNA e MLP, prevede che tutti i parametri di input siano inseriti all'interno di due distinti file di configurazione: il primo raccoglie i parametri per la configurazione della rete neurale, il secondo i parametri relativi al caso d'uso in questione.
I nomi dei file di configurazione sono poi inseriti all'interno di un altro file, un semplice file di testo, in cui: il primo rigo contiene il nome del file di configurazione della rete neurale; il secondo contiene un'etichetta che identifica il caso da eseguire ("*TRAIN*","*TEST*","*RUN*"); il terzo rigo contiene il nome del file di configurazione del caso d'uso.
Il modello prevede due *dataset* in input: uno riguardante gli input della rete, uno relativo ai suoi target.
Tuttavia, in generale un *dataset* di input è composto da colonne di valori che rappresentano gli input della rete, uniti a colonne di valori che rappresentano i target. Il *plugin* quindi, in questo caso, deve poter separare il *dataset* input in due file, uno per gli input, uno per i target.
Anche il modello MLPGA è utilizzato per problemi di classificazione e regressione. Per la classificazione, nel caso d'uso *TRAIN*, il *plugin* genera due *plot* relativi entrambi al file contenente gli errori di *training*. Questo file contiene, per ciascuno *step*, due valutazioni dell'errore di *training* (*best* e *worst*). Nel caso d'uso *TRAIN* il file degli output della rete è utilizzato per generare una matrice di confusione concernente i dati che esso contiene; un'altra operazione riguarda l'unione delle colonne del *dataset* in input con i valori di output prodotti durante questa fase; per il caso d'uso *RUN* è prevista solo l'unione delle colonne del *dataset* in input con i valori di output prodotti; il caso d'uso *FULL*, oltre ad implementare le medesime operazioni dei casi d'uso *TRAIN* e *TEST*, riorganizza, al termine della fase di *TRAIN*, l'insieme dei file di input ricevuti dal FW, aggiungendo il file di output prodotto dal *TRAIN*, relativo ai pesi della rete, da passare al caso di *TEST*.

Per la regressione, sono effettuate le stesse operazioni definite per la classificazione, a meno della generazione della matrice di confusione nel caso di *TEST*, che non avrebbe senso per un problema di regressione.
Infine il modello SVM, per compiere un esperimento, utilizza degli oggetti della classe DMM, appositamente creati per il modello in questione. Inoltre i metodi di queste classi utilizzano come parametri le stesse strutture dati ricevute dal FW per quanto riguarda i parametri e i file di input. La struttura di tale modello quindi non è stata presa in considerazione, poiché il modello SVM è strettamente legato all'architettura del componente DMM, mentre l'obiettivo prefissato è relativo a modelli non dipendenti dalla struttura della suite.
Gli originali requisiti funzionali del componente DM*Plugin* erano:

- Comunicare con il componente Driver per registrare i file prodotti in output e ottenere le informazioni relative ai dati di input;
- Comunicare il suo stato al componente FW, in particolare la disponibilità dei dati di output;
- Comunicare con il componente Driver per convertire i dati input e output, in base al formato richiesto dall'utente all'atto della configurazione del *plugin*;
- Effettuare il *parsing* dei file XML di configurazione;
- Eseguire il flusso logico di un esperimento, sulla base della configurazione utente;
- Sincronizzare il termine di un esperimento con i tempi di attesa del componente FW;
- Effettuare la ripartizione dei dati in input, sulla base delle richieste definite dall'utente (*dataset* di *TRAIN* e *TEST*);

Inoltre, il DM*Plugin* doveva rispettare i seguenti requisiti non funzionali:

- Ogni *plugin* deve mantenere uno standard d'interfaccia;
- Ogni *plugin* deve avere associata una priorità d'esecuzione, in caso di esecuzione di diversi casi d'uso;
- Ogni *plugin* deve comunicare con il componente FW tramite *socket* dedicati;

Saranno ora analizzati i blocchi di codice relativi ai costruttori e ad alcuni metodi dei modelli citati.
La Figura 20 mostra il costruttore della classe **Classification_MLP** relativo al modello MLP per la funzionalità classificazione. Il codice contenuto nel riquadro grande è implementato per ciascun caso d'uso.
Per ciascun caso d'uso, sono disponibili quattro insiemi di operazioni:

- In "*Input Fields for UseCase*": vengono definiti i parametri di ingresso previsti, per la funzionalità corrente. Questo blocco contiene per ciascun parametro il codice:

  fields.put(<label parametro>, new DMMFieldParam(<tipo del parametro>, "", "", "", "",<label del parametro> , <vincolo sui valori possibili>, <lista valori>,<descrizione>,<flag per parametro opzionale>));

  dove il campo "vincolo sui valori possibili" è un oggetto che può assumere tre valori: NO_CONSTRAIN, VALUES, RANGE. Nel primo caso, la "lista valori" sarà vuota, nel secondo conterrà l'insieme dei possibili valori, nel terzo, gli estremi dell'intervallo dell'insieme dei valori assegnabili al parametro.

- In "*Input Files for UseCase*": sono definiti i file di input. Questo blocco contiene il codice *iFiles.put(<etichetta>, new DMMFileParam(<etichetta>,"", <descrizione>,<formato>,<flag per stabilire se è un file opzionale>));*

- In "*Output files for UseCase*": sono definiti i file di output: Per ciascun file di output previsto, il blocco contiene il codice *oFiles.put(<etichetta>, new DMMOutputFileParam(<etichetta>,<descrizione>,<flag per file parziale>, <uri>, <formato>));*

- In "*Partial files for UseCase*" vengono definiti i file parziali. Il codice contenuto nel blocco è: *pOFiles.put(<etichetta>, new DMMOutputFileParam(<etichetta>,<descrizione>,<flag per file parziale>, <uri>, <formato>));*

```
public Classification_MLP (IDrStorage dr, String ip, Integer experimentId, String seid, String userName, Integer sessionId) throws Exception) {
        super(dr, ip, experimentId, seid, userName, sessionId);
        Functionality f = new Functionality("Alessandro Di Guido & Stefano Cavuoti", "1.0.0", "http://voneural.na.infn.it/", "MLPClassification.java",
        "MLPClassification", "10-03-12 ", "ale.diguido@gmail.com", "classification", true, true, true, true);

        HashMap<String,DMMFieldParam> fields; HashMap<String,DMMFileParam> iFiles;
        HashMap<String,DMMOutputFileParam> oFiles; HashMap<String,DMMOutputFileParam> pOFiles;
        List<Comparable> l; Map<String,Tag> tags; fields=new HashMap<String,DMMFieldParam>(); iFiles=new
        HashMap<String,DMMFileParam>(); oFiles=new HashMap<String,DMMOutputFileParam>(); pOFiles=new
        HashMap<String,DMMOutputFileParam>();

        Input Fields for UseCase
        Input Files for Use Case
        Output files for Use Case
        Partial files for Use Case

        runningModes.put("Test", new RunningMode(fields,iFiles,oFiles,pOFiles,"Test","http://voneural.na.infn.it",0));

        setInformations(f);
        setRunningModes(runningModes);

}
```

Figura 20 - Costruttore per il modello MLP - classificazione

La Figura 21 mostra il costruttore della classe **Classification_MLPGA** relativo al modello MLPGA per la funzionalità classificazione. Il codice contenuto nel riquadro grande è implementato per ciascun caso d'uso previsto dal modello.

Per ciascuna funzionalità, sono disponibili quattro insiemi di operazioni visti nel caso precedente.

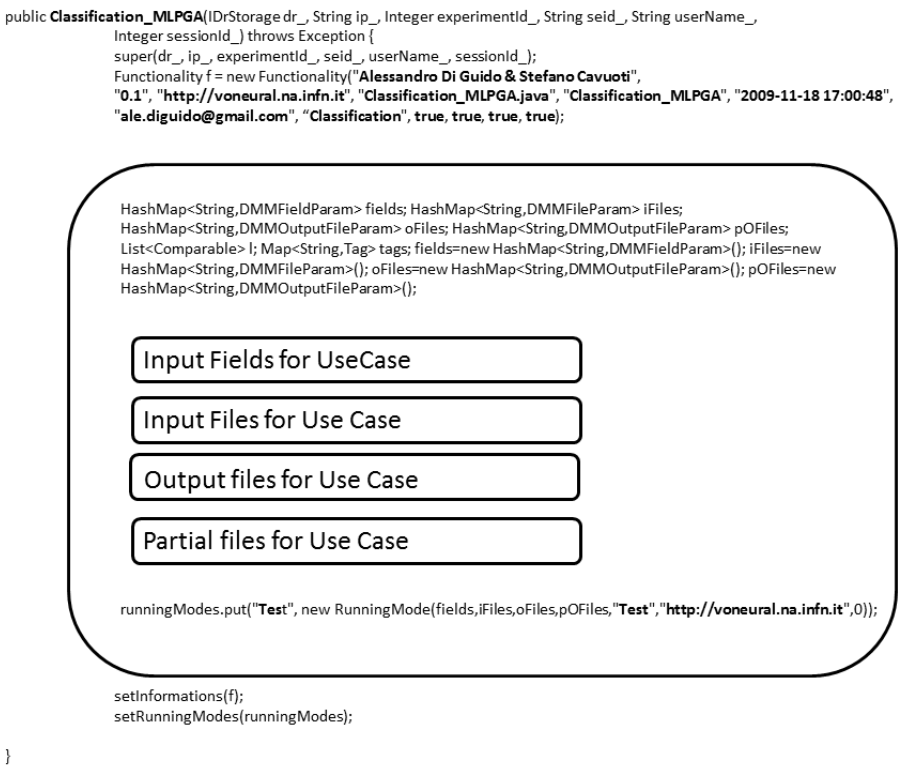

Figura 21 - Costruttore per lil modello MLPGA - classificazione

Il costruttore per la classe **Classification_MLPQNA** (Figura 13) ha una struttura analoga ai costruttori dei modelli descritti precedentemente.

Nella Figura 22 sono evidenziati i blocchi di codice per il metodo *getFiles* della classe Classification_MLP. I blocchi evidenziati sono:

- "*Input files uri*": Il blocco contiene, per ciascun parametro di input il codice:

  case <nome etichetta>:

  uri = uri_out + dirExp + "/<nome del file>";

  break;

  All'interno del ciclo mostrato in figura, associa alla variabile uri, il nome del file relativo al file di input corrente

- "*Translate input files*": Il blocco contiene per ciascuno dei file di input da tradurre:

  if ((f.getName().charAt(0) != <etichetta del file di input>)) {

  Dataset t = new Dataset(uri, f.getFormat());

  t.translate(uri + ".<estensione del file>", "<formato del file>", true);

  f.setUri(uri + ".<estensione del file>");

  }

  Il blocco traduce il file di input corrente, nel formato indicato;

- "*Output files uri*": Per ciascun file di output:

case <etichetta file di output>':
<nome parametro associato al file>= f;
<nome parametro associato al file>.setUri(uri_out + dirExp + "/" + dirExp + "<nome file di output>";
break;

Associa ad ogni parametro, relativo ad un file, il corrispondente uri;

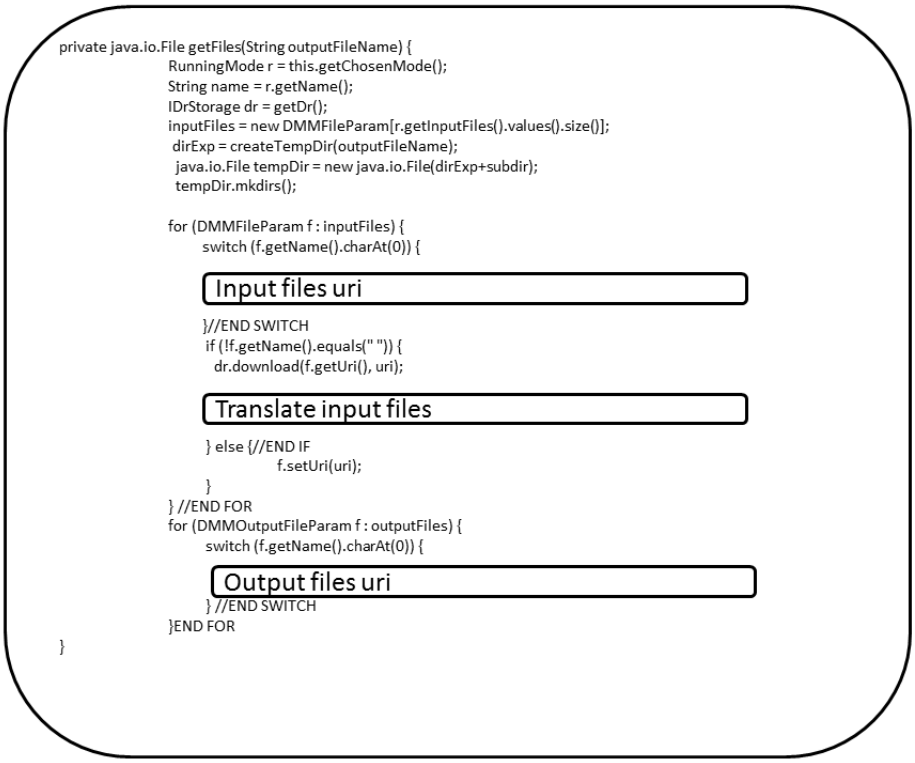

Figura 22 - Il metodo getFiles della classe Classification_MLP

I blocchi del metodo *getFiles* della classe **MLPGA** (Figura 23) corrispondono agli stessi blocchi del caso precedente.

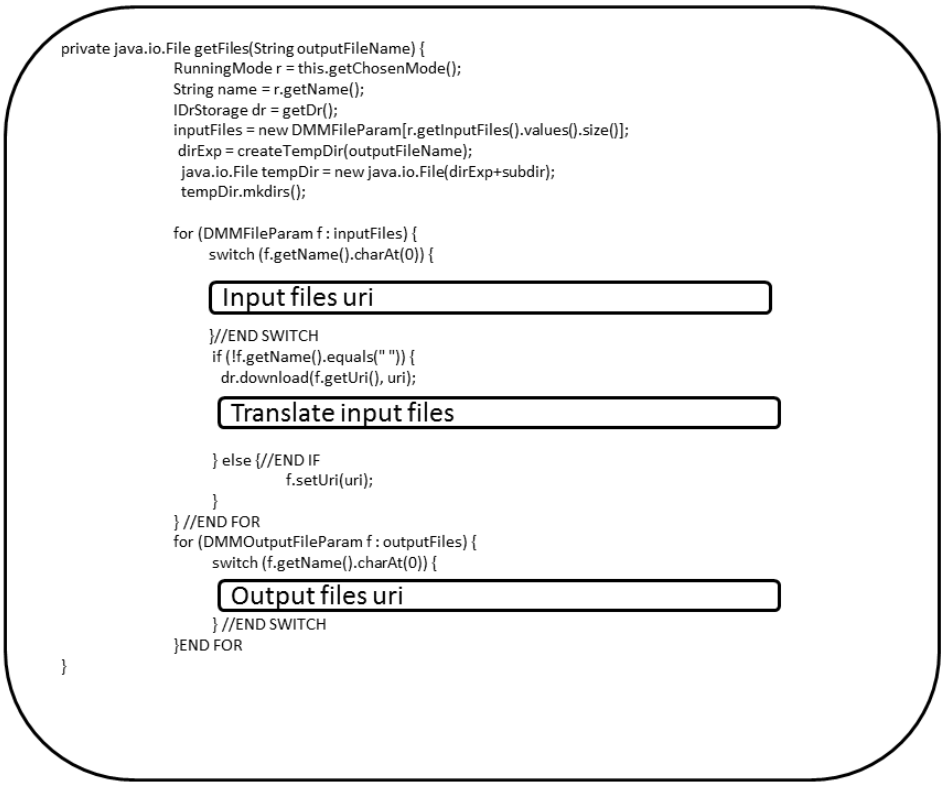

Figura 23 - Il metodo getFiles della classe Classification_MLPGA

Il metodo *getFile* della classe **Classification_MLPQNA** (Figura 15) comprende i medesimi blocchi citati.

La Figura 24 mostra la struttura del metodo *fullRun* del modello MLP. I blocchi di codice individuati sono:

- "*Renaming train Output files*": per ciascun file da rinominare, cioè i files i cui nomi corrispondono con quelli prodotti durante la fase di test, il blocco prevede il codice:

  Rename(uri_out + dirExp + "/" + dirExp + <nome del file>, uri_out + dirExp + "/" + dirExp + "Train"+<nome del file>);

  che rinomina un file;

- "*Updating input files for test*": associa dei files di output prodotti durante il train, come input durante la fase di test:

  DMMFileParam <nome parametro>= new DMMFileParam(<etichetta>, null, null, null, false);
  <nome parametro>.setUri(netTrain.getUri());
  DMMFileParam[] newInputFiles = new DMMFileParam[inputFiles.length + <numero dei parametric da aggiungere>];
  boolean found = false;
  for (DMMFileParam file : inputFiles) {

```
 if (file.getName().equals<etichetta>)) {
  file = <nome parametro>;
  found = true;
  break;
 }
  …
}
if (!found) {
 for (int i = 0; i < inputFiles.length; i++) {
  newInputFiles[i] = inputFiles[i];
 }
 newInputFiles[newInputFiles.length - <valore del ciclo corrente>] = N;
} else {
 newInputFiles = inputFiles;
}
```

- "*Plot images*": il blocco contiene il codice relative alle immagini da *plottare*. Per ciascuna immagine da *plottare*:

  Visualization v = new Visualization();
  v.plot2d(<parametro del file di output da plottare>.getUri(), <valore ascissa>, <valore ordinata>, "<formato del file>", "<estensione immagine>");

- "*Generate confusion matrix*": genera una matrice di confusione a partire dalle informazioni su un file di output:

  Dataset d = new Dataset(<parametro del file di output>.getUri(),(<parametro del file di output>.getFormat());
  d.translate((<parametro del file di output>..getUri() + "<.estensione>", "<formato desiderato>", true);
  c.GetConfusionMatrix(<parametro del file di output>.getUri() + ".ascii");

- "*Join output files*": il blocco contiene il codice che unisce le colonne di un file di input ad un file di output:

  Dataset d = new Dataset(<parametro del file di input>.getUri(),(<parametro del file di output>.getFormat());
  boolean[] columnTestout = new boolean[d.getColumns()];
  for (int i = 0; i <numero di colonne> ; i++) {
  columnTestout[i] = true;
  }
  d.columnSubset(columnTestout);
  Dataset d2 = new Dataset(uri_out + dirExp + "/(<nome del fole di output>.", "<formato>");
  d2.joinColumn(d);

- "*Append files to log*": comprende il codice per aggiungere il contenuto di un file all'interno del file *log*;

AppendFileLog(uri_out + dirExp + "/mlp_" + <nome del file log>, uri_out + dirExp + "/" + dirExp + "<nome del file da appendere>");

- "*Renaming and reporting files*": contiene il codice per rinominare un file di output e caricare il file nella cartella utente. Per ciascun file di output, il codice contenuto nel blocco è

  RenameReport(test_out, uri_out + dirExp + "/<modello>_" + <caso d'uso> + "_<nuovo nome>");

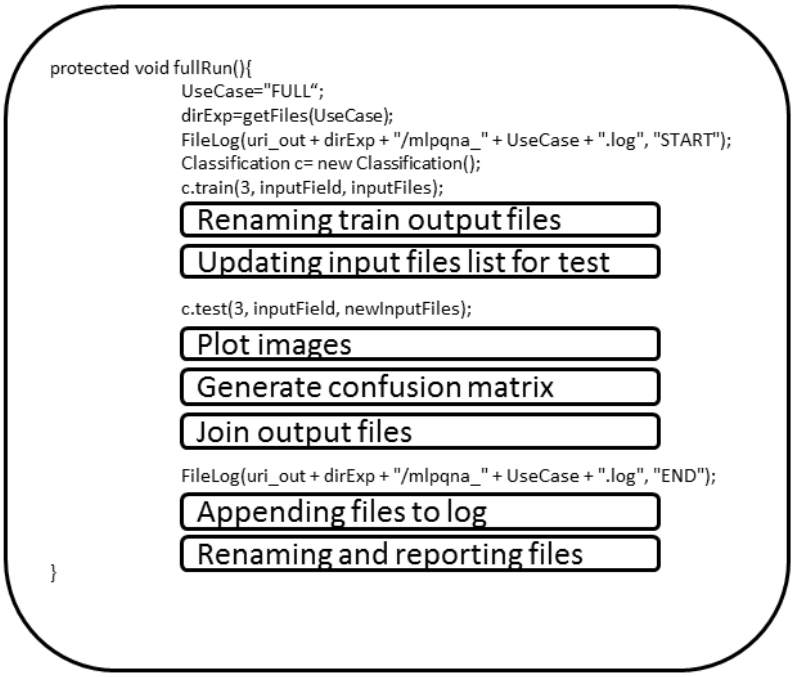

Figura 24 - *fullRun* per Classification_MLP

La Figura 25 mostra il metodo *fullRun* per la classe **Classification_MLPGA**. I blocchi di codice evidenziati coincidono con quelli del metodo *fullRun* per il modello MLP.

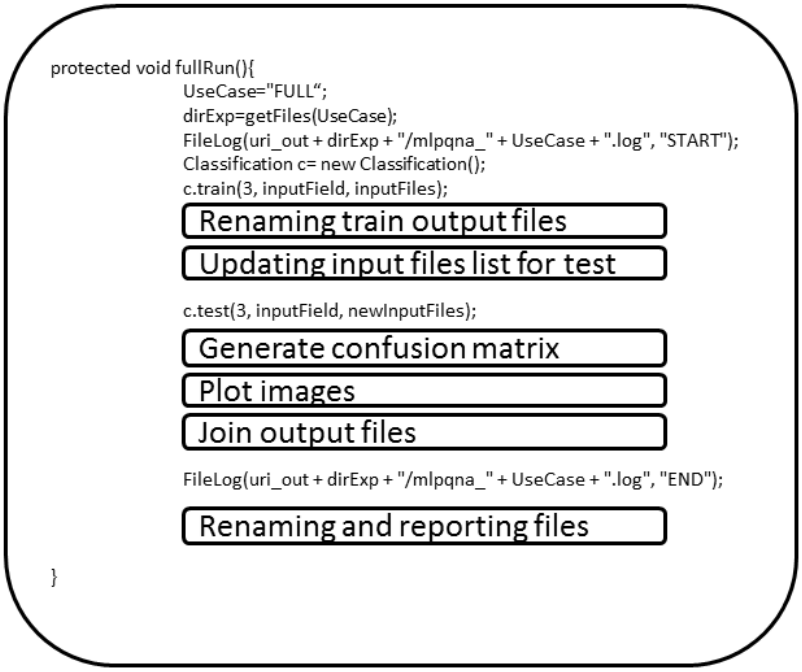

Figura 25 - *fullRun* per Classification_MLPGA

Il metodo *parse_command_line* della classe **MLP** (Figura 26) prevede tre blocchi:

- "*Default values for fields*": comprende l'associazione per ciascun parametro ad un valore di default:

  per ciascun parametro, per il quale è previsto un valore di default:

  <nome parametro> = <valore>;

- "*Fields values*", per ciascun parametro ricevuto dalla classe relativa alla funzionalità il blocco, comprende il codice:

  case <etichetta>:
  <nome parametro> = (<tipo del parametro>) dmmparam[i].getValue();
  break;

- "*input files from DMPlugin component*" comprende il codice, per ciascun file di input:
  case '<etichetta>':
  <nome del parametro> = File_in[i].getUri();
  break;

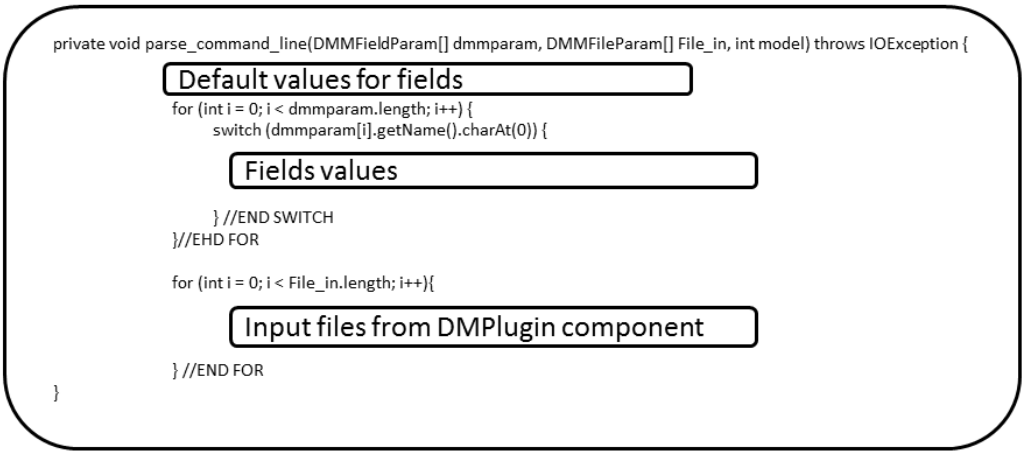

Figura 26 – Il metodo parse_command_line per MLP

Il metodo *parse_file* della classe **MLPGA** (Figura 27), come il metodo *parse_command_line* della classe **MLP,** ha il compito di associare una variabile a ciascun parametro di input ed a ciascun file di input. I blocchi "*Default values for fields*", "*Fields values*" e "*input files from DMPlugin component*" coincidono con quelli visti in precedenza. Questo metodo prevede un blocco aggiuntivo che si occupa di dividere (*split*) un file di input in due parti:

- "Split input dataset:

  Dataset d = new Dataset(<uri del file di input>, "<formato>");
  int nColumns = d.getColumns();
  boolean[] columnMap = new boolean[nColumns];
  for (i = 0; i < <parametro utilizzato per la divisione delle colonne>; i++) {
  columnMap[i] = true;
  }
  d.columnSubset(columnMap);
  d.translate(dirExp + <nome del primo file splittato>, "<formato di conversione>", true);
  i = 0;
  Dataset t = new Dataset((<uri del file di input>,, "<formato del file>");
  for (i = 0; i < n_input_neurons; i++) {
  columnMap[i] = false;
  }
  for (i = n_input_neurons; i < nColumns; i++) {
  columnMap[i] = true;
  }
  t.columnSubset(columnMap);
  t.translate(dirExp + <nome del second file splittato>, "<formato di conversione>", true);

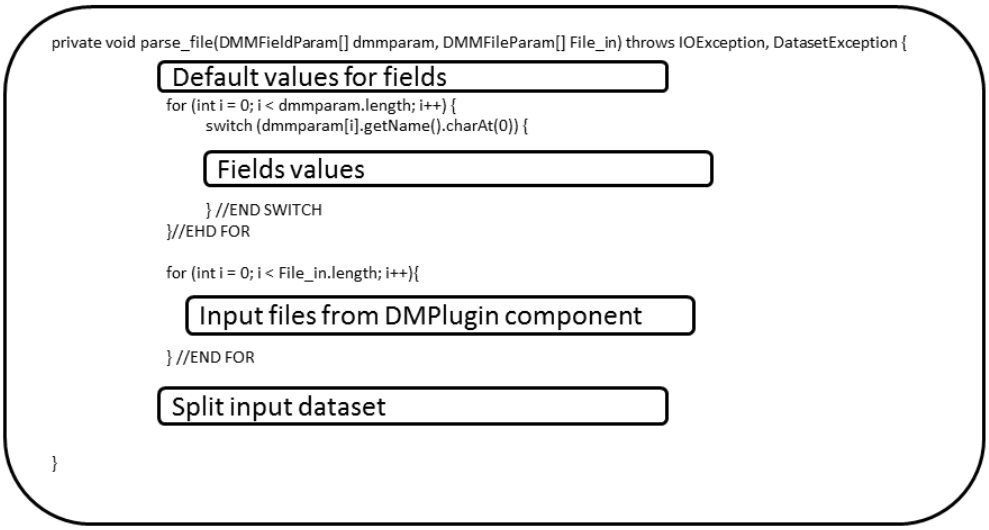

Figura 27 – Il metodo parse_file della classe MLPGA

Il metodo *train* prevede un solo blocco che si occupa della creazione della linea di comando per eseguire l'esperimento:

- "*Constants*": Associa a ciascun parametro relativo ad una costante, il corrispondente valore.

  <nome parametro > = <valore del parametro>;

- "*Creation command line*":

  String path = "/data/DAME/suite/utilities/<nome dell'eseguibile>";
  String cmd = path + " " + <parametro di posizione 1>+ " " +…+ " " + <parametro di posizione n>;
  String cmd2 = "sh " + <directory dell'esperimento> +"/fout.sh";
  FileOutputStream foutb=new FileOutputStream(directory dell'esperimento> +"/fout.sh");
  PrintStream ps=new PrintStream(foutb);
  ps.println("#!/bin/bash\n"+cmd);
  Process p = r.exec(cmd2);

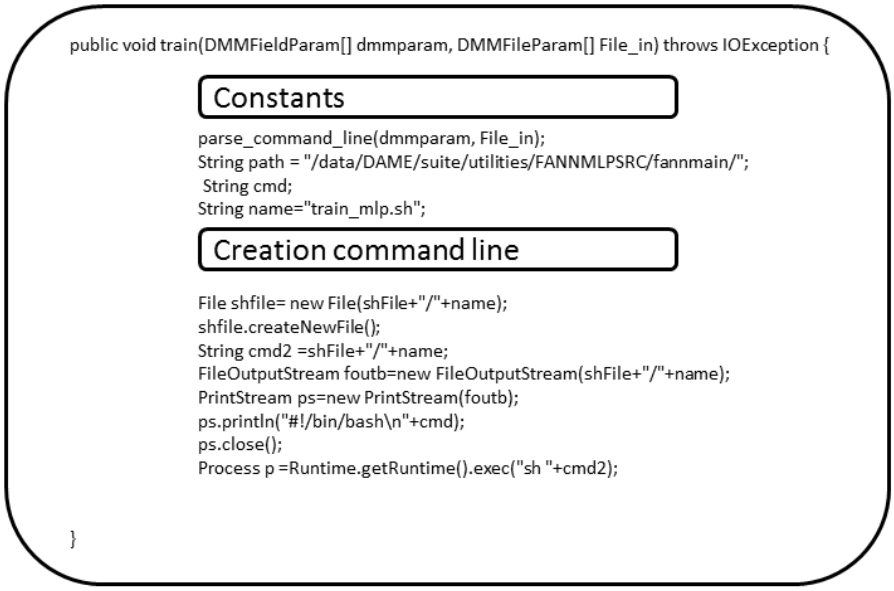

Figura 28 – Il metodo *train* della classe MLP

Il metodo *train* per la classe **MLPGA** (Figura 29) comprende il blocco "*Creation configuration files*" con il compito di creare i file di configurazione ed eseguire l'esperimento; il blocco "*Constants*" è analogo al caso precedente.

- "*Creation configuration files*":

```
File specific_experiment_conf = new File(dirExp + "training_configuration.txt");
File specific_use_case = new File(dirExp + "experiment_configuration.txt");
for (int i = 0; i < dmmparam.length; i++) {
        switch (dmmparam[i].getName().charAt(0)) {
        //PARSING SPECIFIC EXPERIMENT CONFIGURATION FILE
        ...
        case <etichetta>:
        <parametro> = (<tipo>) dmmparam[i].getValue();
        break;
        ...
        }
}
//WRITING SPECIFIC EXPERIMENT CONFIGURATION FILE
PrintStream p = new PrintStream(specific_experiment_conf);
 ...
 p.println(<valore del parametro i-mo per specific_experiment_conf >);
...
//WRITING NEURAL NETWORK CONFIGURATION FILE
PrintStream p = new PrintStream(dirExp  + neural_net_conf);
...
p.println(<valore del parametro i-mo per neural_net_conf >);
...
//WRITING SPECIFIC USE CASE CONFIGURATION FILE
PrintStream u = new PrintStream(specific_use_case);
```

```
u.println(dirExp + "neural_net_conf.txt");
u.println("<valore parametro i-mo per specific_use_case >");
u.println(dirExp + "training_configuration.txt");
Runtime r = Runtime.getRuntime();
pr = r.exec(path + "MLPGA " + specific_use_case);
```

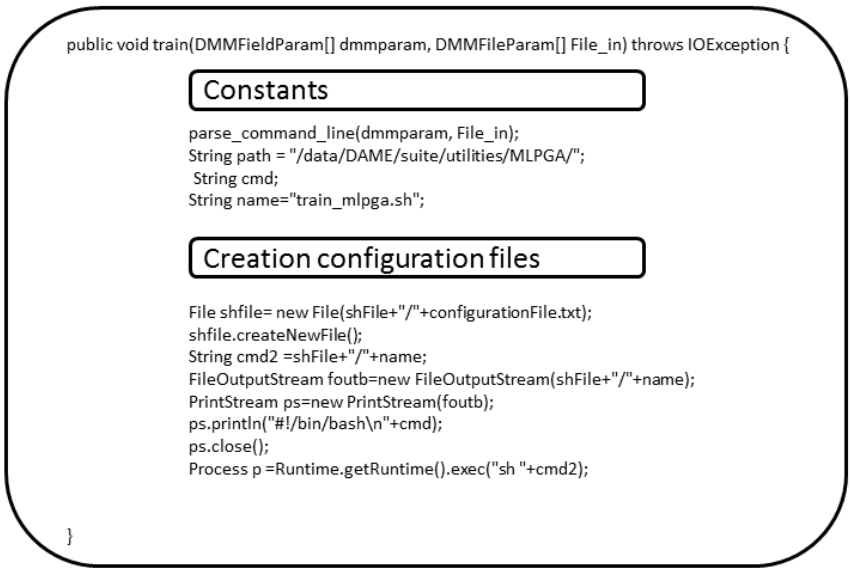

Figura 29 – Il metodo *train* per la classe MLPGA

3.2 La generalizzazione di un plugin

Come descritto in precedenza ciascun *plugin* è rappresentato da una coppia Funzionalità–Modello, di conseguenza, l'integrazione di un nuovo *plugin* richiede la creazione di una nuova coppia di classi che rispettino quest'associazione.

Le figure del capitolo precedente permettono di analizzare il codice dei due modelli MLP e MLPGA, oltre al modello MLPQNA, presenti nella suite.

Dal punto di vista implementativo, ciascun elemento del componente DM*Plugin* relativo ad una funzionalità contiene:

- Un **costruttore** che crea un oggetto *Functionality* che raccoglie le informazioni relative al nome dell'utente, la versione del modello, la documentazione del modello, il nome della classe java della funzionalità, il nome del *plugin*, la data di creazione e la mail dell'utente; per ciascun caso d'uso da implementare, poi, istanzia le strutture dati in grado contenere tutti gli oggetti relativi ai parametri di input, file di input e file di output;
- Un insieme di **parametri,** che tengano traccia dei file di input e dei file di output previsti;

- Un metodo **checkLogFile,** con il compito di controllare il contenuto del file *log* generato dal *plugin*. Generalmente, se un modello produce in output un file sull'esito dell'esperimento, queste informazioni potrebbero essere inserite all'interno del file *log*. Questo metodo controlla che le informazioni siano effettivamente contenute all'interno del file *log*, oppure inserisce, all'interno del file, un messaggio di errore;
- Un metodo **getFiles** che effettua una scansione dei file di input passati dal FE. Ciascun file ricevuto, viene poi spostato nella cartella temporanea dedicata all'esecuzione dell'esperimento. In particolare in questa fase vengono anche tradotti i file ricevuti nel formato appropriato per il modello. Successivamente effettua una scansione di tutti i file di output previsti dal modello e imposta i relativi URI per poterli poi gestire una volta prodotti, alla fine dell'esecuzione del modello;
- Un metodo **renameReport,** con il compito di rinominare il file e di spostare il file in questione dalla directory temporanea, relativa all'esperimento in esecuzione, all'interno della directory dell'utente;
- Un metodo **createTempDir,** per generare una directory temporanea univoca relativa all'esecuzione di un esperimento;
- Un metodo **FileLog** eseguito prima dell'avvio di un esperimento e al termine dell'esperimento scrive, all'interno di un file di testo, la data e l'ora di avvio e terminazione dell'esperimento;
- Un metodo **appendFileToLog** che permette di aggiungere ulteriori informazioni al contenuto del file *log*: un algoritmo di *Machine Learning* potrebbe prevedere un file di *log* in output contenente. Questo metodo permette di aggiungere il suo contenuto all'interno del file di *log* del *plugin*.
- Un metodo ***join*Columns** con il compito di unire le colonne di un file, con quelle di un altro file. Questo metodo deve poter definire anche l'indice della colonna del file, dopo la quale è necessario inserire le colonne dell'altro file;
- Il metodo ***trainRun*** che ha il compito di (i) passare i parametri ricevuti dal componente FW al modello per eseguire il caso d'uso *TRAIN*; (ii) creare la directory temporanea per l'esecuzione dell'esperimento; (iii) se necessario, convertire i file di input nel formato previsto dal modello; (iv) eseguire il metodo *TRAIN* del modello; (v) effettuare eventuali *plot*; (vi) generare l'eventuale matrice di confusione; (vii) eseguire *join* delle righe di diversi file e infine (viii) rinominare i file di output secondo lo standard previsto;
- Il metodo ***testRun*** per il caso d'uso *TRAIN*: ha il compito di (i) passare i parametri ricevuti dal componente FW al modello, per eseguire il caso d'uso *TEST*; (ii) creare la directory

temporanea per l'esecuzione dell'esperimento; (iii) se necessario, convertire i file di input nel formato previsto dal modello; (iv) eseguire il metodo *TEST* del modello; (v) effettuare eventuali *plot*; (vi) generare l'eventuale matrice di confusione; (vii) eseguire *join* delle righe di diversi file e infine (viii) rinominare i file di output secondo lo standard previsto;

- Il metodo ***runRun*** per il caso d'uso *RUN*: ha il compito di (i) passare i parametri ricevuti dal componente FW al modello per eseguire il caso d'uso *RUN*; (ii) creare la directory temporanea per l'esecuzione dell'esperimento; (iii) se necessario, convertire i file di input nel formato previsto dal modello; (iv) eseguire il metodo *RUN* del modello; (v) effettuare eventuali *plot*; (vi) generare l'eventuale matrice di confusione; (vii) eseguire *join* delle righe di diversi file e infine (viii) rinominare i file di output secondo lo standard previsto;
- Il metodo ***fullRun*** per il caso d'uso *FULL*: ha il compito di (i) passare i parametri ricevuti dal componente FW al modello per creare la directory temporanea per l'esecuzione dell'esperimento; (ii) se necessario, convertire i file di input nel formato previsto dal modello; (iv) eseguire il metodo *TRAIN* del modello; (v) rinominare i file che hanno lo stesso nome dei file di output del *TEST*; (vi) aggiornare l'insieme dei parametri di input per il *TEST*, aggiungendo i file di output creati dal *TRAIN* da passare a *TEST*; (vii) eseguire il caso d'uso *TEST*; (viii) effettuare eventuali *plot*; (ix) generare l'eventuale matrice di confusione; (x) eseguire la *join* delle righe di diversi file e infine, (xi) rinominare i file di output secondo lo standard previsto;

La Figura 30 mostra la struttura generale di un oggetto relativo alla funzionalità. I blocchi tratteggiati rappresentano le porzioni di codici la cui presenza all'interno del metodo dipende dalle informazioni presenti all'interno dell'albero delle informazioni del plugin. Ad esempio, se per alcun file di output non è prevista la generazione di alcuna immagine, allora il blocco "Plot images" non sarà presente all'interno della classe da implementare.

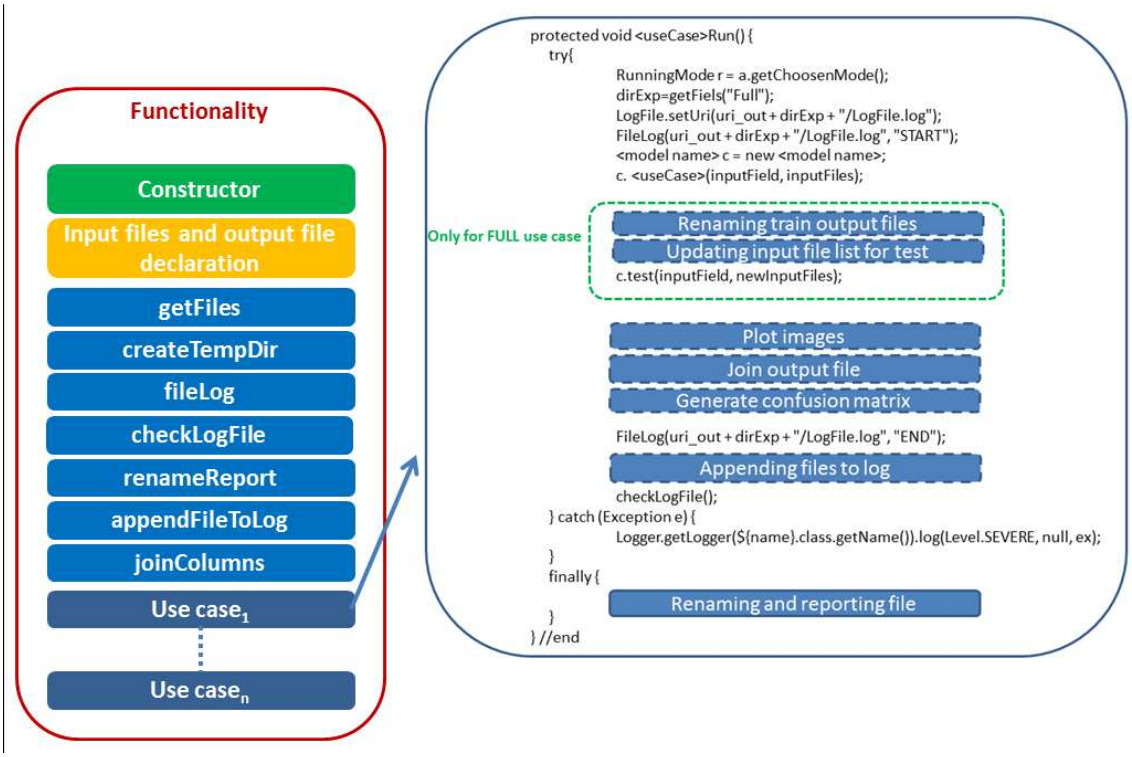

Figura 30 - Generico oggetto relativo alla Funzionalità

Per quanto riguarda un componente del DMM, ciascun suo nuovo componente prevede:

- Un metodo ***TRAIN***: se il modello prevede il passaggio di tutti parametri a linea di comando, crea il comando *shell* con l'ordine dei parametri definito dal modello ed esegue l'esperimento. Se invece il modello prevede che i parametri siano contenuti all'interno di uno (o due) file di configurazione, allora inserisce i parametri all'interno del (o dei) file di configurazione, rispettando anche in questo caso l'ordine definito dal modello; inserisce i percorsi (URI) dei file di configurazione all'interno di un file di testo ed esegue il caso d'uso *TRAIN*, passando come unico argomento alla linea di comando il nome del file di testo che contiene tutte le informazioni. Ai parametri, per i quali l'utente non ha definito un valore, sono assegnati i valori di *default*;
- Un metodo ***TEST***: sono eseguite le stesse operazioni definite per il caso d'uso *TRAIN*, utilizzando però i parametri e il loro ordine definiti per il caso d'uso *TEST*;
- Un metodo ***RUN***: sono eseguite le stesse operazioni definite precedentemente, utilizzando però i parametri e il loro ordine definiti per il caso d'uso *TEST*;
- Un insieme di **parametri** che tengano traccia dell'insieme dei parametri di input e dei nomi dei file di input;

- Un metodo **parse_command_line** che scansiona l'insieme dei parametri e dei file di input ricevuti da un oggetto DM*Plugin* e associa ciascuno di essi ad una variabile locale;

La Figura 31 mostra la struttura generale di una classe relativa al modello. Anche in questo caso i blocchi con contorni tratteggiati indicano la loro presenza in dipendenza dalle informazioni presenti all'interno dell'albero rappresentativo di un *plugin*. La classe finale conterrà uno tra i blocchi "*Creation command line*" e "*Creation configuration files*", a seconda che il modello preveda rispettivamente, i parametri passati tramite linea di comando, oppure tramite file di configurazione.

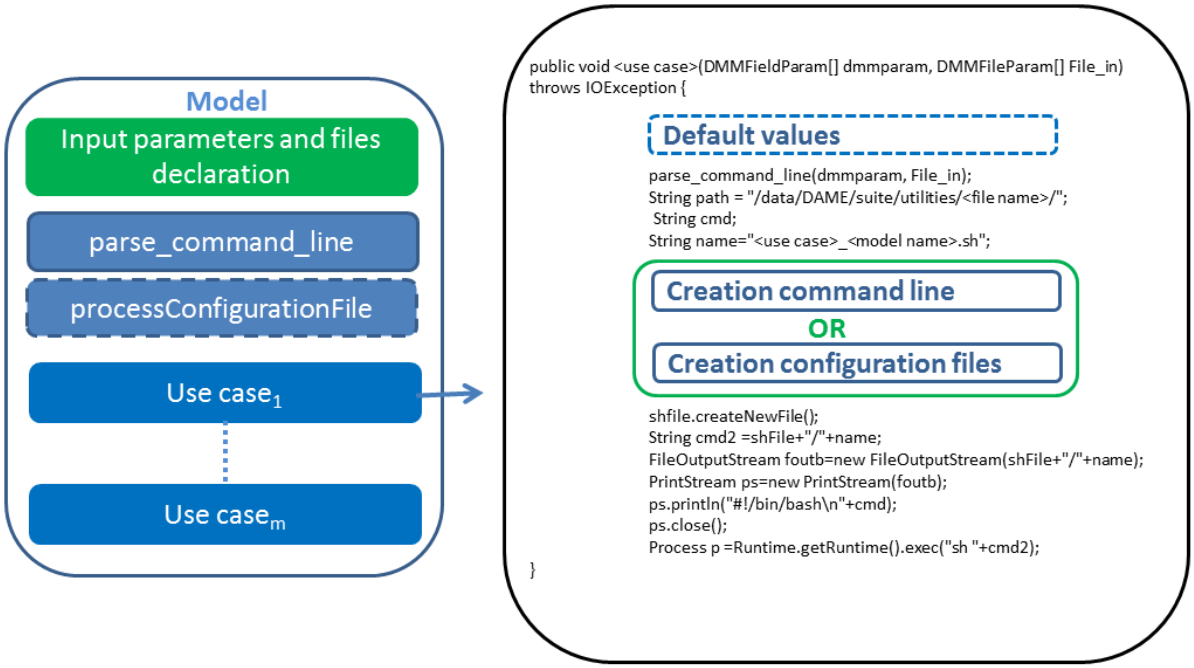

Figura 31 - Generico oggetto relativo al Modello

Quindi, la struttura interna del *plugin* è stata resa indipendente dalla funzionalità e dal modello. Ciò che differenzia un *plugin* da un altro, sono solo le informazioni dei parametri di input, dei file di input e di output, strettamente dipendenti dal modello:

- Funzionalità;
- Modello;
- Casi d'uso da implementare;
- Parametri di input per ciascun caso d'uso;
- File di input per ciascun caso d'uso;
- File di output per ciascun caso d'uso;

- Costanti relative a ciascun caso d'uso;

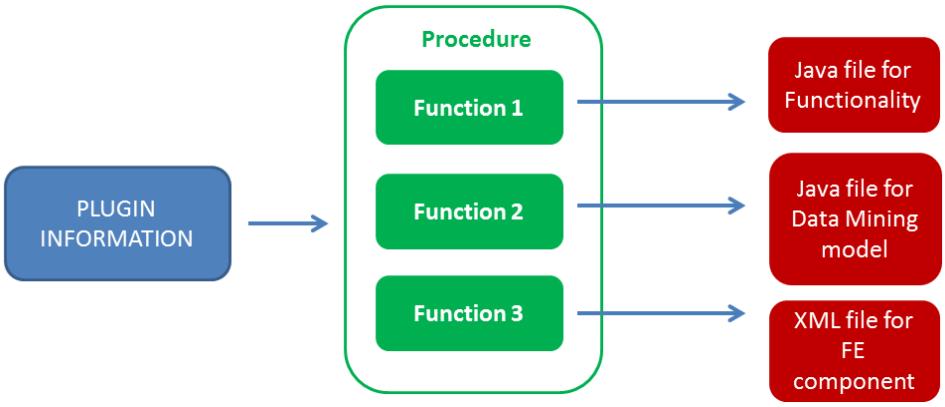

Figura 32 - Funzioni della procedura automatica

La generazione automatica di un *plugin* deve produrre in output due file Java, il primo appartenente al componente DM*Plugin*, il secondo appartenente al DMM, oltre a un file XML di interfaccia con il componente FE. Quindi, è necessario definire delle funzioni che, dato in input un insieme di informazioni relative alla configurazione di un *plugin*, restituiscano in output i file Java e XML rappresentanti il *plugin* (Figura 32).

Ciascun *plugin*, quindi, prevede una serie di casi d'uso, e per ciascuno di essi, è possibile definire dei parametri di input, dei file di input, di output e delle costanti (Figura 33).

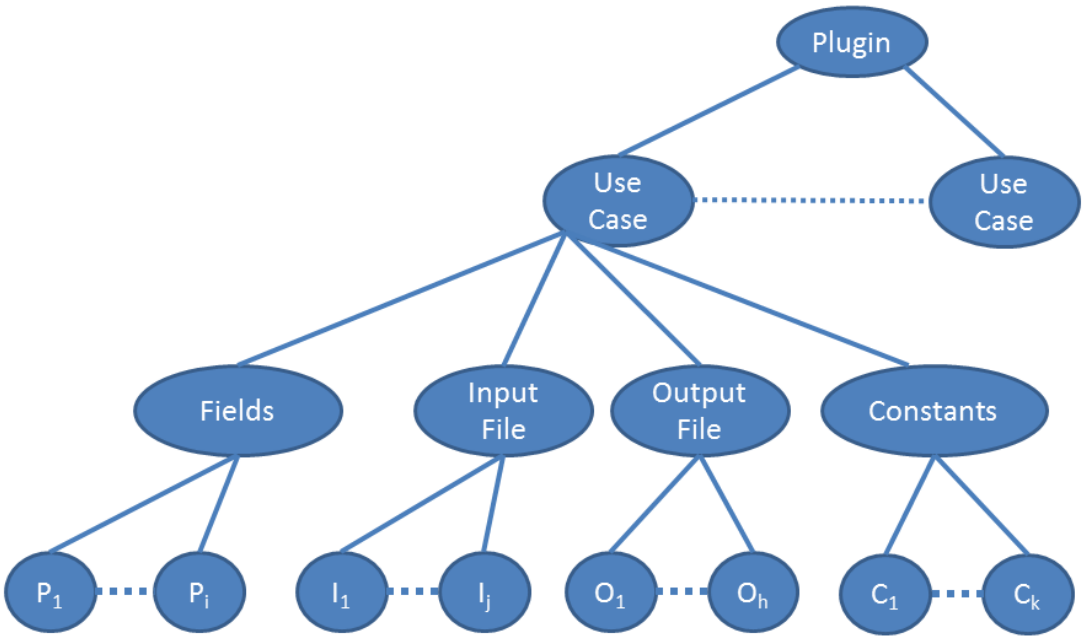

Figura 33 - Struttura delle informazioni relative ad un *plugin*

Ogni *plugin* è caratterizzato dai seguenti attributi:

- *modelName*: nome del modello da implementare;
- *functionality*: la funzionalità da implementare;
- *version*: la versione del modello;
- *owner*: il proprietario del modello;
- *ownerMail*: l'*e-mail* del proprietario;
- *documentation*: un *link* in cui trovare la documentazione sul modello;
- *dateCreation*: la data di creazione del *plugin*;
- *execName*: nome dell'eseguibile relativo all'algoritmo di *Machine Learning.*

Ciascun *plugin* prevede uno o più casi d'uso da implementare, ciascuno di essi è caratterizzato da:

- *useCaseName*: nome del caso d'uso (*TRAIN*, *TEST* o *RUN*);
- *useCaseDocumentation*: *link* alla documentazione sul particolare caso d'uso.

Ciascun caso d'uso prevede un insieme di parametri con i seguenti attributi:

- *namePar: label* con la quale il parametro sarà istanziato nel codice finale;
- *nameFE*: *label* con la quale il parametro sarà visualizzato all'utente;
- *description*: una breve descrizione relativa al parametro;
- *type*: il tipo associato al parametro (Intero, Stringa, Reale, Booleano);
- *constrain*: vincoli di assegnazione di valori al parametro. Il parametro potrebbe assumere qualsiasi valore, oppure un numero ben definito di valore, oppure potrebbe essere compreso in un particolare intervallo di valori;
- *constrainVal*: qualora fosse stato imposto un vincolo sui valori associabili al parametro, come descritto nel punto precedente, deve essere possibile definire i parametri consentiti o l'intervallo consentito;
- *isOptional*: *flag* per stabilire se è obbligatorio assegnare un valore al parametro oppure no;
- *defVal*: stabilire un valore di *default* se è un parametro opzionale;
- *posCommandLine*: se il parametro è contenuto all'interno della linea di comando, deve essere possibile definirne la posizione;
- *isInUseCaseConfFile*: Se il parametro deve essere aggiunto in un file di configurazione;

- *posUseCaseConfFile*: se il parametro è contenuto all'interno di un file di configurazione, deve essere possibile definirne anche in questo caso la posizione;
- *isInConfFile*: Se il parametro deve essere aggiunto in un file di configurazione diverso dal precedente;
- *posConfFile*: se il parametro è contenuto in un file di configurazione diverso dal precedente, deve essere possibile indicarne la posizione;
- *posFE*: la posizione nella quale comparirà il parametro all'utente;
- *passing*: parametro utilizzato solo nel caso d'uso *FULL*. Permette di poter associare una sola *label* a livello di FE a due parametri di input (uno per *TRAIN* e uno per *TEST*). Il valore assegnato al parametro, verrà assegnato, in fase di *TEST*, al parametro il cui nome è contenuto in questo attributo.

Per i file di input invece, le informazioni sono:

- *namePar: label* con la quale il file di input sarà istanziato nel codice finale;
- *nameFE*: *label* con i quale il parametro è visualizzato dal FE;
- *description*: Una breve descrizione;
- *isOptional*: *flag* per stabilire se il file è opzionale oppure no;
- *convertTo*: stabilisce se è necessario convertire il file in un particolare formato prima di avviare un esperimento;
- *posCommandLine*: Se il nome dei file di input è passato al modello tramite la linea di comando, deve essere possibile indicarne la posizione al fine di un ordinamento dei parametri;
- *posUseCaseConfFile*: Se il file di input deve essere contenuto in un file di configurazione, deve essere possibile indicarne la posizione;
- *posConfFile*: Se il file di input deve essere contenuto in un file di configurazione diverso dal precedente, deve essere possibile indicarne la posizione;
- *posFE*: la posizione del file di input nella lista di parametri visualizzata dall'utente;
- *isInUseCaseConfFile*: *flag* che stabilisce se il nome del file deve essere inserito all'interno del file di configurazione del caso d'uso;
- *toUseCaseConfFile*: se il nome del file deve essere aggiunto in un file di configurazione;
- *isInConfFile*: *flag* che stabilisce se il nome del file deve essere inserito all'interno di un altro file di configurazione, diverso dal precedente.
- *toConfFile*: se il nome del file deve essere aggiunto in un altro file di configurazione, diverso dal precedente;

- *subSet*: Se le colonne del file devono essere suddivise all'interno di due file separati;
- *using*: la suddivisione delle colonne di un *dataset* è effettuata per mezzo del valore di un altro parametro. Questo attributo raccoglie il nome del parametro dal quale andare a prendere il valore per effettuare la suddivisione del file.

OutputFile:

- *namePar*: *label* con la quale il file di output sarà istanziato nel codice finale;
- *name*: Nome del file;
- *URI*: Il *path* relativo a partire dalla directory di esecuzione nel quale verrà creato il file;
- *format*: Il formato del file;
- *description*: Descrizione del file;
- *renameIn*: Nome con cui rinominare il file, se necessario;
- *isPartial*: Se il file contiene informazioni parziali;
- *isOptional*: Se il file è creato solo in casi particolari;
- *plot*: Se è richiesto di *plottare* il contenuto del file;
- *xVal*: La colonna che corrisponderà all'ascissa del grafico, se si desidera *plottare* il contenuto del file;
- *yVal*: La colonna che corrisponderà all'ordinata del grafico, se si desidera *plottare* il contenuto del file;
- *appendLog*: Se il file deve essere inserito all'interno del file di *log*;
- *confMatrix*: *flag* che indica se i valori del file di output devono essere espressi tramite matrice di confusione;
- *appendToFile*: Il nome del file a cui aggiungere il contenuto di questo file;
- *passTo*: nome del file di input per *TEST* a cui associare il file di output (solo per *FULL*);
- *appendFile*: Il nome del file le cui colonne devono essere aggiunte a questo file;
- *using*: Se al file di output devono essere aggiunte delle colonne, deve essere necessario definire la posizione dalla quale aggiungerle;

Costanti:

- *namePar*: del parametro;
- *type*:tipo (Intero, Stringa, Reale, Booleano);
- *value*: valore;
- *posCommandLine*: posizione nella linea di comando, se necessario;

- *posUseCaseConfFile*: posizione nel file di configurazione, se previsto;
- *posConfFile*: posizione in un altro file di configurazione se previsto;
- *isInUseCaseConfFile*: se deve essere aggiunto ad un file di configurazione;
- *isInConfFile*: se deve essere inserito in un file di configurazione diverso dal precedente;

Ciascun metodo delle due classi Java rappresentanti il *plugin*, e del file XML d'interfaccia con il componente FE, conserva una struttura che è indipendente dalle informazioni relative ai parametri, ai file di input e ai file di output. A tal proposito, le funzioni su citate che permettono la creazione dei file Java a partire dall'insieme delle informazioni di un *plugin*, possono essere implementate con l'ausilio di *template* che rappresentino le generiche strutture dei file.
Questo tipo di approccio prevede quindi che le funzioni precedentemente citate accettino in ingresso, oltre alle informazioni sul *plugin*, anche un apposito *template* con cui processare i dati (Figura 34).

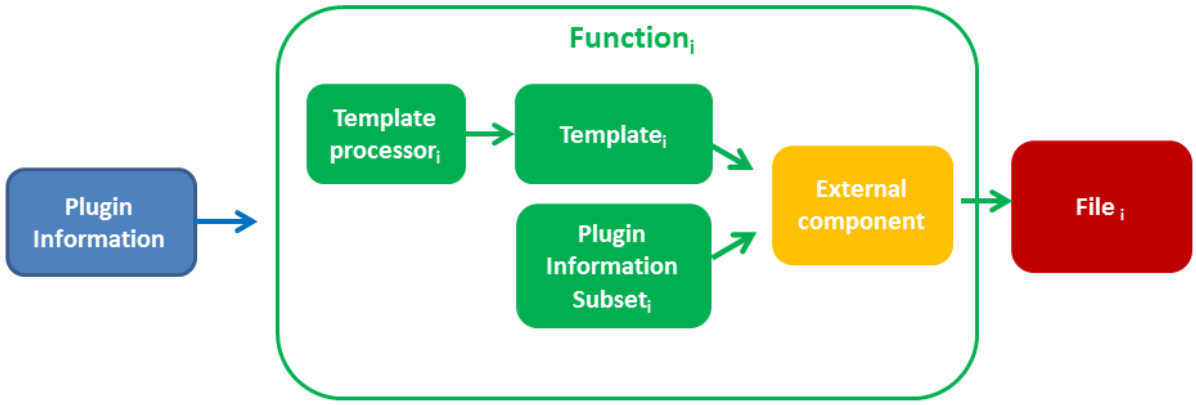

Figura 34 – La funzione di generazione automatica del codice

La scelta di utilizzare un *template* per la generazione del codice, anziché la sua generazione diretta, è dettata dal fatto che quest' approccio permette, oltre all'alto grado di riutilizzabilità del codice creato, una più agile gestione del codice java da generare, riducendo quindi la possibilità di generare errori ed il tempo di *debugging*. Un'altra vantaggiosa caratteristica seguita da tale approccio è una migliore leggibilità del codice che genera i file finali.
Per quanto riguarda la generazione di un *template*, esso è creato con l'ausilio di una funzione che prende in input le informazioni relative ad un *plugin* e genera in output un *template*. La scelta di questo tipo di strategia, in alternativa alla creazione manuale di un *template*, ad esempio all'interno di un file di testo, è stata adottata per evitare che una modifica volontaria o involontaria al contenuto del file del *template* possa compromettere la generazione dei file finali. Tuttavia, la generazione

automatica di un *template* e la successiva elaborazione può sembrare una scelta meno vantaggiosa ed efficiente, rispetto a quella legata alla generazione diretta del codice finale. In realtà la strategia scelta unisce diversi vantaggi, dalla facilità di formattazione del codice, al riutilizzo di alcuni blocchi, all'efficienza legata alla creazione diretta del codice. Dunque, il *template*, generato dalla funzione citata in precedenza, contiene la parte del codice finale che è indipendente dalle caratteristiche più specifiche di ciascun *plugin*, come ad esempio le operazioni sui file di output, a meno delle informazioni più generali, come il nome del modello e la funzionalità ad esso associata da implementare. L'elaborazione del *template* con il sottoinsieme di informazioni del *plugin,* per ciascuna funzione, è eseguito da un componente esterno.
Per la generazione del file java riguardante la funzionalità, la funzione di generazione utilizza:

- Nel **Costruttore**: le informazioni generali del *plugin* (*owner* ,*ownerMail*, *version*, *documentation*, *modelName, creationDate*), nonché le informazioni relative ai casi d'uso implementati (useCaseName, *useCaseDocumentation*) e per ciascuno di essi, l'insieme dei parametri (*namePar*, *description*, *constrain*, *constrainVa*l, *isOptional*), sui file di input (*namePar*, *description*, *format*, *isOptional*), dei file di output (*uri*, *description*, *format*, *isOptional*);
- Nella definizione dei **parametri**: parametri di input (*namePar, type*), file di input (*namePar*), file di output (*namePar, uri*);
- Nel metodo ***createTempDir***: le informazioni generali (*modelName*);
- Nel metodo FileLog: come per il metodo precedente, solo le informazione generali(*functionality*, *modelName*);
- Nel metodo ***renameReport***: le informazioni generali (*functionality*, *modelName*);
- Nel metodo ***checkLogFile***: le informazioni generali(*functionality, modelName*) ;
- Nel metodo ***getFiles***: le informazioni relativi ai file di input (*namePar*, *format*, *convertIn*), ai parametri di input (*namePar, type*), costanti (*namePar*, *value, type*) e ai file di output (*nemPar, uri*);
- Nei metodi ***trainRun***, ***testRun***, ***runRun*** e ***fullRun***: le informazioni generali (*modelName*), informazioni sui file di output (*uri*, *isPlot*,*xVal*, *yVal*, *isJoin*, *using*, *isAppendLog*, *renameTo*, *isPartial*,*isOptional*,*confMatrix*);
- Nel metodo ***fullRun***: anche altre informazioni relative ai parametri di input (*namePar*) e ai file di output (*passTo*);

Per la generazione del file java riguardante il modello, la funzione di generazione utilizza:

- Nei metodi ***TRAIN, TEST*** e ***RUN***: informazioni generali (*execName*), informazioni sui parametri di input (*namePar*, *posCommandLine*, *isInConfFile*, *posConfFile*, *isInUseCaseConfFile*, *posUseCaseConfFile*) , sui file di input (*namePar, posCommandLine*, *isInConfFile, posConfFile*, *isInUseCaseConfFile*, *posUseCaseConfFile*) e sulle costanti (*namePar, type, value*, *posCommandLine*, *isInConfFile, posConfFile, isInUseCaseConfFile*, *posUseCaseConfFile*);
- Nel metodo ***parse_command_line***: informazioni sui parametri di input (*namePar*, *type*, *value*) e sui file di input (*namePar*);

Riassumendo, ciascun *plugin* è composto da due file Java, uno relativo alla Funzionalità, che si occupa di tutte le informazioni sui casi d'uso implementati, sui file di input e sulle operazioni su di essi, sulle costanti e sui file di output e sulle operazioni ad essi relativi, e uno relativo al Modello con il compito di ordinare i parametri in ingresso, ed avviare un esperimento di *Machine Learning*. La struttura di questi file comprende costruttori e metodi strutturati sempre nella stessa forma, ma il cui contenuto varia in base al *plugin*, alle informazioni su di esso, e in particolar modo ai parametri di input ed ai file di input e output.
La struttura definita per raccogliere le informazioni relative ad un *plugin*, insieme con i *template* creati tramite le apposite funzioni, permettono la generazione del codice java finale associato ad un *plugin*.
In particolare, in una prima analisi, confrontando le strutture dei *template* e dei dati relativi ad un *plugin*, con le classi definite nel capitolo 2 e le informazioni sui parametri di input, sui file di input e sui file di output per il modello MLPQNA, è possibile stabilire che la procedura genera un codice java compatibile con il codice implementato per il *wrapping* del modello MLPQNA.

3.3 La generazione del file d'interfaccia per il componente FE

Anche il file XML utilizzato dal componente FE per interfacciare i parametri di un modello con un utente, ha una struttura dipendente solo dai casi d'uso previsti dal modello e dai parametri richiesti per ognuno di essi.

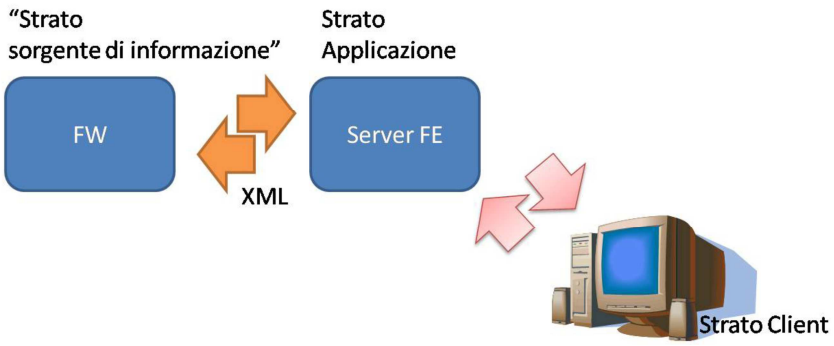

Figura 35 – Scambio informazioni fra FE e FW, basato su documenti XML

La figura sopra riportata mostra il meccanismo generale di comunicazione tra il componente FE e l'infrastruttura interna della *web application* (attraverso il componente FW). Tale flusso di comunicazione avviene mediante specifici documenti XML:

- **XML FW Functionality Description**: descrizione dei parametri relativi alla funzionalità richiesta;
- **XML FW Interactive Report**: descrizione dello stato di un esperimento;
- **XML FW Session *Manager***: descrizione delle informazioni utente relative all'accesso ed alla gestione di sessioni di lavoro;
- **XML FW Meta-Data**: descrizione dei metadati relativi ai file caricati dall'utente;
- **XML FW Functionalities List**: descrizione dei parametri relativi alle funzionalità esposte dalla *web application*;
- **XML Error Report**: report delle eccezioni verificatesi a seguito di malfunzionamenti interni al flusso dati/comandi tra i vari componenti software ed agli esperimenti eseguiti;
- **XML FE Dataset operation**: descrizione delle operazioni di manipolazione dei dati, richiesti dall'utente;

Per quanto concerne il passaggio dei parametri di configurazione, immessi dall'utente, esso avviene tramite il primo dei documenti XML, elencati sopra. La figura successiva mostra l'interazione tra FE e FW durante tale scambio.

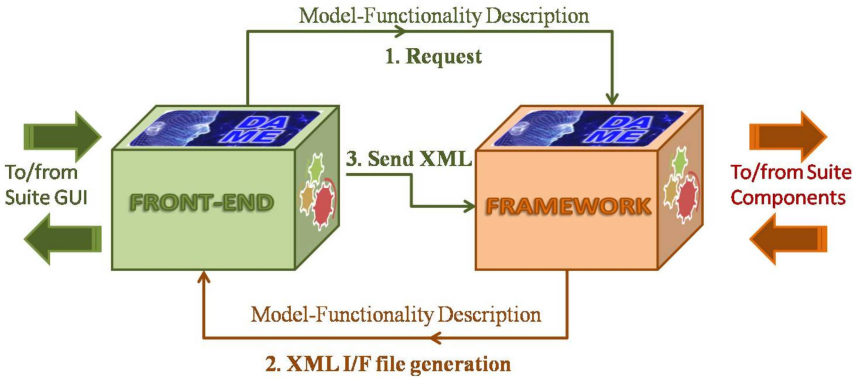

Figura 36 - Flusso informazioni esperimento fra FE e FW

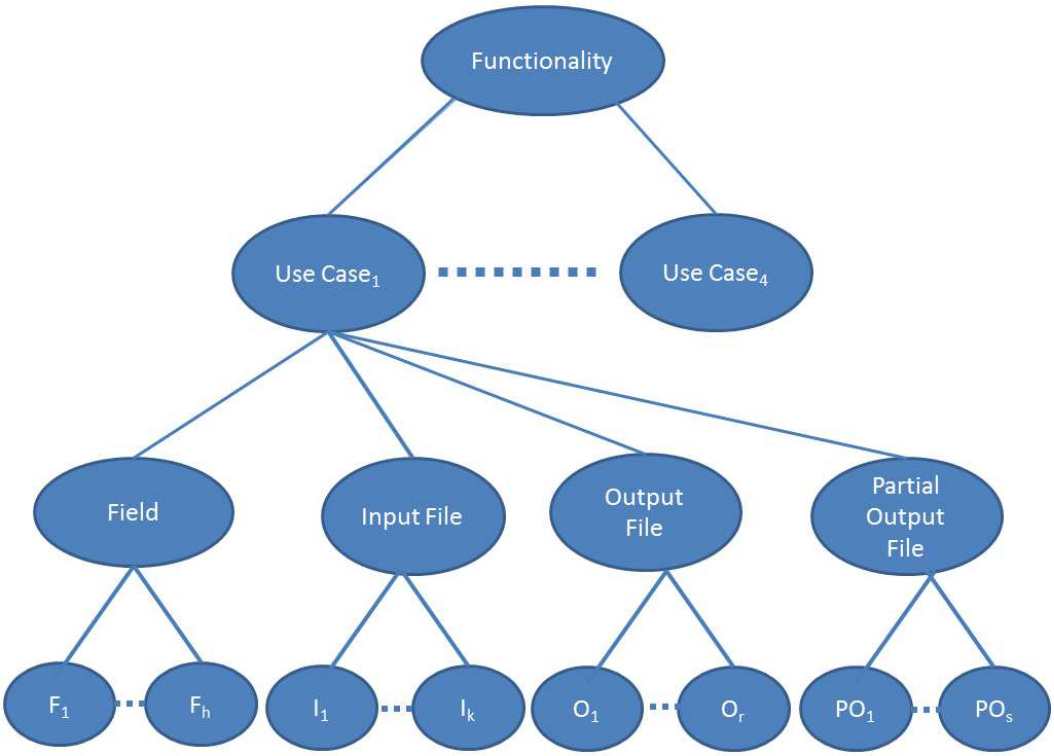

Figura 37 - Struttura file XML FW Functionality Description

Partendo dalla Figura 37 il file XML contiene i seguenti attributi.

Funzionalità:

- *Functionality*: nome della funzionalità;
- *documentation*: *link* dove trovare la documentazione;
- *version*: la versione del *plugin*;
- *creationDate*: data di creazione del *plugin*;

Caso d'uso:

- *name*: nome della funzionalità;
- *value*: *link* alla documentazione relativa al caso d'uso

Parametri di input:

- *nome*: nome del parametro;
- *ucd*: per uniformità con gli standard VOTABLE;
- *unit*: specifica l'unità di misura del parametro;

- *datatype*: tipo del parametro (Intero, Stringa, Booleano, Reale);
- *precision*;
- *isOptional*: *flag* che stabilisce se è un parametro opzionale, cioè che non è strettamente necessario l'inserimento di un valore ad esso associato da parte dell'utente;

File di input:

- *name*: nome del parametro che identifica il file di input;
- *ucd*: per uniformità con gli standard VOTABLE;
- *datatype*: lasciato vuoto;
- *arraysize*: posto sempre a 200;
- *utype*: indica il formato del file;
- *isOptional*: *flag* che stabilisce se è un parametro opzionale, cioè che non è strettamente necessario l'inserimento di un valore ad esso associato da parte dell'utente;

File di output:

- *name*: nome del parametro che identifica il file di output;
- *datatype*: campo lasciato vuoto;

File di output parziali:

- *name*: nome del parametro che identifica il file di output;
- *datatype*: campo lasciato vuoto;

4 Progettazione dell'integrazione automatica di modelli in DAMEWARE

L'attuale configurazione di DAMEWARE permette di integrare agevolmente una nuova funzionalità al suo interno, ma non in modo automatico: l'integrazione di un nuovo modello, richiede una nuova compilazione dell'intera *web application*.
Il principale problema di questo tipo di approccio è che la ricompilazione dell'intera applicazione richiede l'interruzione di tutti gli esperimenti che sono in esecuzione, nonché il loro riavvio dopo la ricompilazione. Questo problema avrebbe una scarsa rilevanza se tutti gli esperimenti lanciati tramite DAMEWARE riguardassero solamente piccoli *dataset* e quindi, se fossero relativi ad esperimenti con un ridotto tempo di esecuzione. Tuttavia l'utilità principale di DAMEWARE è proprio quella di mettere a disposizione della comunità scientifica uno strumento in grado di poter effettuare esperimenti su enormi *dataset* (dell'ordine cioè dei TB). L'interruzione di un esperimento con queste caratteristiche porterebbe quindi alla perdita di molti giorni di lavoro.
Nasce quindi l'esigenza di poter aggiungere funzionalità alla suite, nel modo meno invasivo possibile.

4.1 Analisi del progetto

Oltre alla necessità di aggiungere nuove funzionalità all'interno della *web application* DAMEWARE, evitando reiterate compilazioni dei codici sorgente, un altro importante vincolo da rispettare è quello di permettere l'integrazione di un modello di *data mining* ad un utente finale, senza richiedere la conoscenza dell'architettura interna ed i meccanismi di flusso delle informazioni tra i componenti dell'infrastruttura (2.3).
Lo strumento da realizzare deve cioè permettere all'utente di configurare tutti i parametri per gli algoritmi di *Machine Learning*, gli use case (*TRAIN*, *TEST*, *RUN*, *FULL*), nonché le informazioni di I/O, con cui generare il codice sorgente da integrare, per rendere fruibile la nuova risorsa di *Machine Learning dalla comunità di utenti registrati*.
Naturalmente ciò implica la necessità di avere un rigido controllo sul codice dell'utente e sul *plugin* generato, per evidenti motivi di sicurezza: un utente male intenzionato potrebbe lanciare in esecuzione del software potenzialmente dannoso.

Il componente DM*Plugin* di DAMEWARE ha il compito di costruire un *wrapping* dell'applicazione utente, al fine di permetterne l'esecuzione coerente all'interno della *web application* e, se necessario, effettuare operazioni sugli output prodotti.
Dunque, per integrare un nuovo algoritmo di *Machine Learning*, deve essere creata una nuova coppia di classi secondo il Paradigma Funzionalità-Modello (1.2), insito nei requisiti base dell'applicazione.
Per interfacciare il nuovo algoritmo di *Machine Learning* con la *web application*, è necessario un ulteriore file che ha lo scopo di esporre all'utente l'insieme di tutti i parametri configurabili. Tale file è di tipo XML ed è ovviamente utilizzato dal componente Front End (Figura 4).
Una volta creato tale file, è necessario aggiungere le informazioni, relative al nuovo modello, all'interno del database, tramite componente REDB (Figura 4), compilando il codice sorgente del modello di *Machine Learning,* per ottenere una versione compatibile con le caratteristiche hardware dell'infrastruttura sulla quale utilizzarlo.
Al termine di tutte le operazioni definite in precedenza, sarà possibile avviare esperimenti in DAMEWARE con il nuovo modello di *Machine Learning* completamente integrato.

4.2 Analisi dei Requisiti

Un utente, che desideri integrare il proprio algoritmo all'interno della suite, deve avere a disposizione un codice sorgente, sviluppato in un qualsiasi linguaggio di programmazione, la cui compilazione produca un eseguibile avviabile da linea di comando, nella quale siano definiti i relativi parametri, necessari ad eseguire un esperimento. Un esempio di una linea di comando compatibile è:

./mlpqna 10 3 0.001 10 0.01 1000 7 1 2 15 6 0 datasets/agn_7_stat_full.txt 0 10 1 704 ./trainedWeights.txt

Oppure (in alternativa al passaggio dei parametri a linea di comando), il programma dell'utente deve poter accettare tutti i parametri confezionati sottoforma di un file esterno di configurazione, ad esempio:

./mlpga confFile.txt

dove confFile.txt è un file di configurazione formattato in semplice codice ASCII.

Il compito dello strumento da implementare è dunque quello di creare un componente in grado di eseguire il *wrapping* relativo ad un modello di *Machine Learning* al fine di poterlo integrare all'interno dell'infrastruttura DAMEWARE.
E' noto che un tipico flusso computazionale per un esperimento basato sul *Machine Learning* richieda una serie di fasi (*Use Case*) da eseguire in sequenza.

All'inizio di un esperimento è richiesta una fase di *training* per addestrare il modello. La seconda fase generalmente è quella di *TEST*. L'ultima fase è denominata "*RUN*", in cui il modello, addestrato e testato, è usato come una generica funzione applicata a nuovi dati. E' possibile definire anche una fase chiamata "*FULL*", che automaticamente esegue un *workflow* includente la fase di *TRAIN* e di *TEST* in sequenza. L'utente deve poter scegliere i passi validi per il modello da integrare (generalmente un modello di *Machine Learning* prevede tutti i passi citati).
Per ciascuno degli *use case* menzionati, l'utente deve inserire le informazioni sui parametri richiesti per avviare un esperimento. L'applicazione, tramite schermate differenti distingue le informazioni relative ai parametri, file e constanti in input, oltre ai file di output.
Ogni caso d'uso di un modello prevede, naturalmente, di specificare i file di input da utilizzare (come *dataset* di input, file di configurazioni esterni, ecc.). Ogni volta che un esperimento è completato, ha come risultato una serie di file di output, sotto forma di immagini, *plottaggi*, diagrammi, tabelle, file di *log*, ecc..
Il caso d'uso *FULL* è una particolare fase di un esperimento in cui i casi d'uso *TRAIN* e *TEST* sono eseguiti insieme in sequenza. In questo caso è necessario specificare le operazioni da effettuare alla fine del caso d'uso *TRAIN* e prima di avviare il caso d'uso *TEST*. Inoltre, in un comune esperimento di *Machine Learning*, alcuni output prodotti dal *TRAIN*, costituiscono gli input della successiva fase di *TEST* (ad esempio in una rete neurale il file dei pesi della rete salvati in un file di output durante il caso *TRAIN*, devono essere passati come input nella fase di *TEST*).
La procedura deve quindi raccogliere informazioni generali come il nome del modello da integrare, un URL dove reperire eventuali informazioni sul modello, la versione, il dominio di funzionalità, il nome dell'utente e la sua *e-mail*.
L'utente deve poter quindi definire i casi d'uso previsti e implementati dal proprio algoritmo di *Machine Learning*, e per ognuno di essi deve poter specificarne i parametri di input, eventuali costanti, file di input e di output previsti.
In particolare per i parametri di input è richiesta la definizione di:

- Un'etichetta con la quale sarà rappresentato il parametro: il valore definito in questo campo corrisponderà alla *label* utilizzata dal componente Front End;
- Una breve descrizione relativa al parametro in questione. Questa informazione sarà contenuta in un *tooltip* a disposizione dell'utente, così da poter mettere a disposizione maggiori informazioni sul parametro e guidare nella compilazione;
- Il tipo di parametro: I parametri possono essere di tipo Stringa, Intero, Reale o Booleano;

- Stabilire se il parametro sia obbligatorio. In caso contrario, si dovrà far uso di un valore di *default*;
- Stabilire, se necessario, l'insieme dei valori ammessi o l'intervallo di valori ammessi: è possibile che un parametro possa assumere sono un numero prestabilito di valori, oppure dei valori all'interno di uno o più intervalli.
- Indicare se il campo è passato tramite linea di comando, o all'interno di un file di configurazione.

Per i file di input l'utente può definire:

- L'etichetta con la quale sarà rappresentato il parametro: come nel caso precedente, tale valore rappresenta la *label* utilizzata dal componente Front End;
- Una breve descrizione che specifichi con maggiore dettaglio le caratteristiche del parametro, Tale informazione sarà utilizzata dal FrontEnd per descrivere con maggior dettaglio le caratteristiche del *dataset* in input;
- Definire se il file possa essere omesso;
- Il formato nel quale il file di input deve essere convertito per poter essere passato all'eseguibile (la *web application* mette a disposizione la scelta arbitraria tra ASCII, CSV, FITS o VOTABLE);
- Indicare se l'URI del file di input è passato all'eseguibile tramite linea di comando oppure attraverso un valore all'interno di un file di configurazione.

Le informazioni richieste per i file di output sono:

- *path* relativo di destinazione del file di output e nome corrispondente;
- Una breve descrizione;
- Il formato (ASCII, CSV, FITS o VOTABLE);
- Rinominare il file di output: alcuni modelli possono produrre file di output con nomi che non rappresentano correttamente il loro contenuto; a tal proposito deve essere necessario poter rinominare il file di output;
- Definire se il file sia parziale (ossia acquisito in *streaming* durante l'esecuzione);
- Indicare se il file sia sempre prodotto, oppure creato solo in casi particolari, come ad esempio un file di report degli errori nei parametri;

- Richiesta di grafici circa le informazioni contenute nel file di output; in tal caso, deve essere possibile poter inserire gli indici delle colonne che rappresenteranno l'ascissa e l'ordinata nel grafico prodotto, nonché il nome dell'immagine generata;
- Richiesta di inserire il file di output nel file di *log*;
- Unire le informazioni presenti in file diversi (*join*);
- Inserire all'interno del file di output il contenuto di un file di input utilizzato per l'esperimento.

Infine, le informazioni relative ad eventuali costanti sono:

- Il nome del parametro che identifica la costante;
- Il tipo (Stringa, Intero, Reale, Booleano);
- Il valore;
- Stabilire se è un valore passato alla riga di comando o all'interno di un file di configurazione.

L'utente deve poter specificare se gli output prodotti siano creati all'interno di una directory definita a *runtime*, oppure se siano prodotti a partire dal *path* corrente di esecuzione dell'esperimento.
Per i parametri inseriti, l'utente deve poter anche definire l'ordine con il quale essi saranno visualizzati all'interno della *web application*.
Un altro ordinamento richiesto è relativo all'ordine dei parametri passati all'applicazione dalla riga di comando o all'interno dei file di configurazione.
I file di configurazione utilizzati da alcuni modelli di *Machine Learning*, non rispettano uno standard comune per quanto riguarda il loro formato. Essi quindi possono essere di qualsiasi tipo e formattare il proprio contenuto nel modo anche diverso da qualsiasi altro modello. Per risolvere questo problema, è possibile fornire dei *template* per ciascun modello che definiscano la struttura dei file di configurazione.
Dopo aver configurato le informazioni sui file di input e di output, costanti, e parametri per la linea di comando per i casi d'uso previsti dal modello da essere integrato, è necessario definire il nome dell'applicazione per lanciare effettivamente un esperimento.

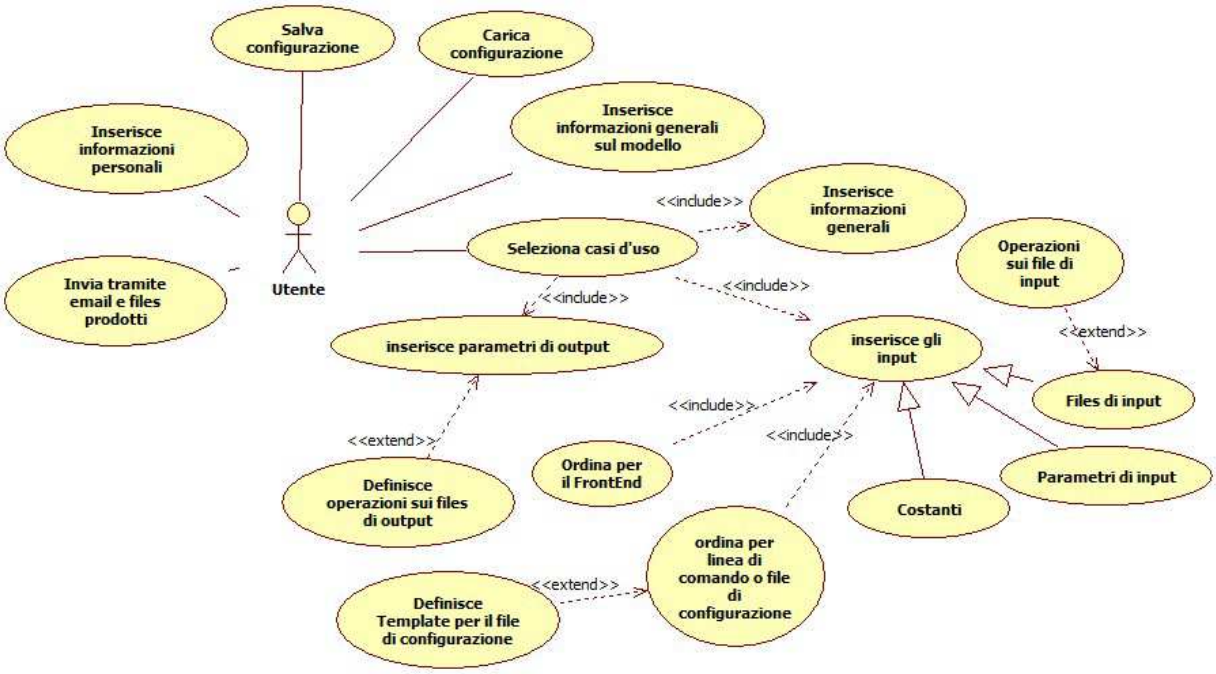

Figura 38 – *Use Case Diagram* per l'utente

Alcuni modelli prevedono la definizione di numerosi elementi tra parametri di input, file di input, costanti e altre informazioni. A tale scopo, è indispensabile che un utente possa salvare le informazioni già inserite, in modo da poterle ricaricare nella procedura e completare le operazioni in un secondo momento.

Per i motivi di sicurezza citati in precedenza, non è possibile fare in modo che un utente possa completamente integrare all'interno della suite il proprio modello, ma affinché ciò avvenga, è necessario che il modello e il relativo codice di *wrapping* prodotto siano inizialmente testati da un amministratore, in grado di valutarne la correttezza e la stabilità.

Per queste ragioni, è necessario dividere la procedura automatica in due procedure indipendenti, una utilizzabile dall'utente, che permetta di creare il codice Java dedicato al *wrapping* del nuovo modello e il file XML di interfaccia con il FrontEnd, ed un'altra, utilizzabile esclusivamente da un amministratore DAMEWARE, in grado di integrare effettivamente il modello nella *web application*, solo dopo averne verificato il funzionamento su un sistema dedicato al *testing* del modello.

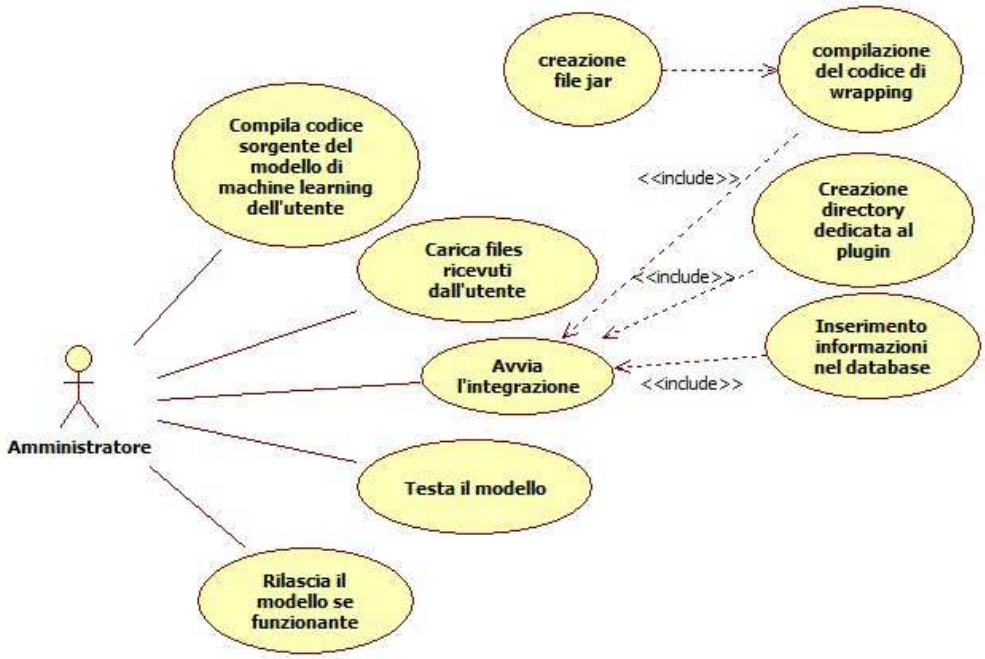

Figura 39 – *Use Case diagram* lato amministratore

Quindi, l'applicazione utilizzabile da un amministratore ha il compito di:

- Caricare il file *Zip* ricevuto dall'utente e verificarne il contenuto;
- Creare le directory necessarie dove immagazzinare il file contenuti nel file *Zip*;
- Compilare il codice Java ricevuto, e creare un file *Jar* contenente il *bytecode* compilato;
- Inserire le informazioni del nuovo modello all'interno della base di dati.

4.3 Class Diagram

L'applicazione deve essere in grado di raccogliere informazioni relative ad i parametri di input che comprendono file, parametri e costanti, oltre ai parametri di output. Tutte le informazioni raccolte devono quindi permettere la creazione del codice java di *wrapping* del nuovo modello, del file XML per interfacciare il nuovo modello con il FE della *web application*, e di un altro file XML con i compito di salvare le informazioni inserite, così da poter riprendere il lavoro dal pinto in cui lo si era lasciato. Tutti i file prodotti e gli eventuali *template* per i file di configurazione previsti dal modello, devono essere archiviati all'interno di un file *zip* ed inviati allo staff di DAME.
La Figura 40 mostra le entità individuate nella fase di analisi e la relazione tra di esse per l'applicazione DAMEWARE *Plugin Wizard*.

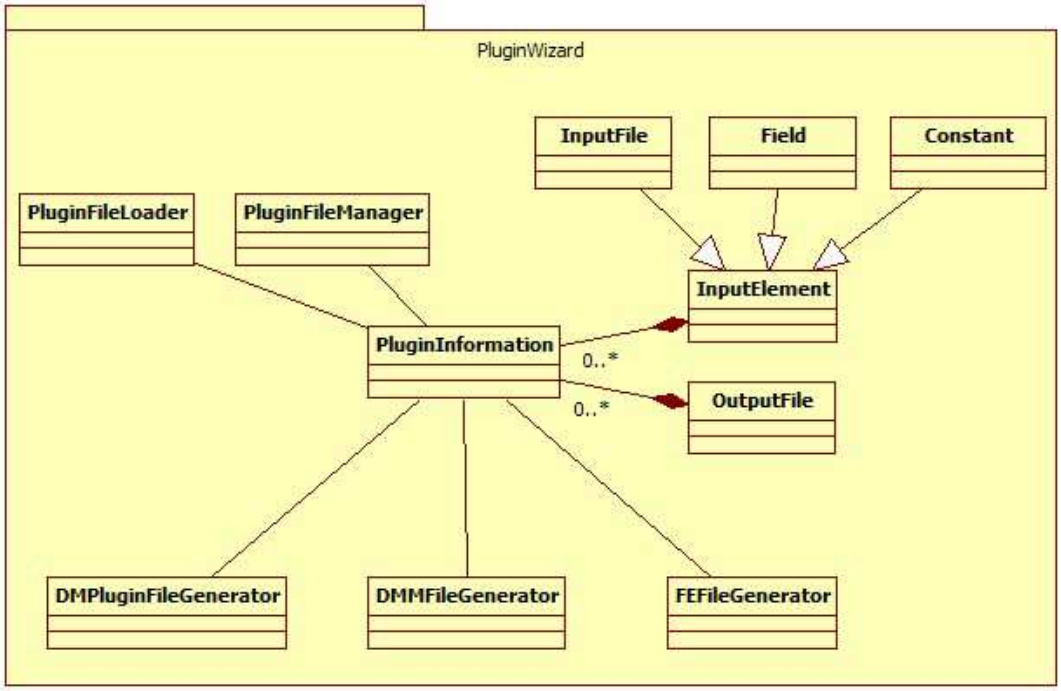

Figura 40 – Diagramma delle classi *PluginWizard*

In particolare, la classe ***Plugin*Information** raccoglie tutte le informazioni relative ad i parametri di input e i parametri di output di un modello da integrare, relativi ad una funzionalità.
La classe **InputElement** rappresenta un generico componente in input e contiene i seguenti attributi:

- *label*: la *label* con la quale il parametro verrà visualizzato dal FE;

- *pos*: la posizione che il parametro assume all'interno della linea di comando o del file di configurazione;
- *posFE*: la posizione che assume nella lista dei parametri visualizzati dal FE;
- *description*: la descrizione relativa al parametro;

La classe **Field** raccoglie informazioni come il tipo, il valore, se è obbligatorio che l'utente debba assegnarvi un valore, e un valore di *default* nel caso contrario;

La classe **InputFile** che estende **InputElement** raccoglie le informazioni relative ad un file di input come il formato in cui deve essere convertito, se e un file opzionale oppure no e se il file deve essere diviso in due file (nel caso di un *dataset* in input che debba essere diviso in due file separati come nel caso del modello MLPGA, par. 3.1);

La classe **Constant** tiene traccia di informazioni come il tipo e il corrispondente valore ;

La classe **OutputFile** raccoglie la descrizione, il nome con cui è creato dal modello, il formato, il nome con cui deve essere rinominato, se è un file parziale, se deve generare dei *plot*, relative ad un file di output;

Il file *zip* generato dal *Plugin Wizard* e inviato dall'utente, deve essere scompattato ed i file presenti al suo interno devono essere spostati in directory ad esse dedicate.

La Figura 41 mostra le entità individuate nella fase di analisi e la relazione tra di esse per l'applicazione *Plugin Wizard*.

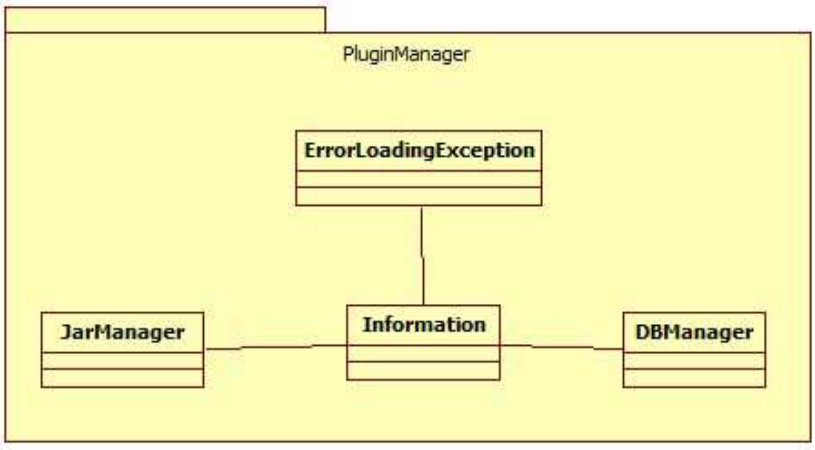

Figura 41 – Diagramma delle classi *Plugin Manager*

La classe **Information** ha il compito di scompattare il file *zip* ricevuto dall'utente, verificarne il contenuto e spostare i file in esso contenuti all'interno della cartella creata, dedicata al *plugin* da integrare.
La classe **Jar*Manager*** compila i file java contenuti nell'archivio *zip* e crea il file *jar* contenente le classi del *plugin*.
Infine, la classe **DB*Manager*** si occupa di inserire le informazioni relative al *plugin* all'interno del database REDB.

4.4 Requisiti non funzionali

L'applicazione è rivolta a utenti in genere non necessariamente esperti di ICT e certamente non a conoscenza dei meccanismi interni alla *web application* DAMEWARE, dovendo peraltro permettere di eseguire le operazioni di configurazione del *plugin* in modo "*user friendly*", tramite una serie di passi successivi. Tale applicazione deve avere le seguenti principali caratteristiche:

- Facile ed intuitiva da usare;
- Assista l'utente durante il suo utilizzo;
- Limiti le possibilità di scelta da parte dell'utente in modo da ridurre gli errori;
- Permetta di modificare le informazioni già inserite;
- Disponga di un' interfaccia grafica.

Tali caratteristiche sono alla base di strumenti software noti con il nome comune di "*Wizard*".
Inoltre l'applicazione deve poter essere eseguito su ogni elaboratore, indipendentemente dal sistema operativo in esecuzione.

5 Realizzazione dell'implementazione dell'integrazione automatica

Il sistema d'integrazione di modelli di *data mining* all'interno della *web application* DAMEWARE è basato su due strumenti software, il "*Plugin Wizard*" per l'utente che voglia integrare il modello ed il "*Plugin Manager*", dedicato all'amministratore dell'infrastruttura per rendere operativa l'integrazione.

Il ruolo cardine dei due strumenti è rendere il più possibile autonomo l'utente dall'infrastruttura e l'amministratore dal modello nuovo, permettendo però a entrambi di compiere facilmente le operazioni d'integrazione, garantendo anche il necessario livello di protezione e sicurezza del sistema.

Lo strumento software in grado di creare tutti gli elementi necessari per l'integrazione di un modello di *Machine Learning* all'interno della *web application* DAMEWARE è dunque denominato "DAMEWARE *Plugin Wizard*".

La menzionata prerogativa "*Wizard*" di tale strumento implica il suo utilizzo tramite una procedura guidata, attraverso cui configurare il sistema di integrazione del modello utente all'interno dell'infrastruttura di *data mining*. Ciò in modo interattivo e tale da poter procedere arbitrariamente in avanti e/o a ritroso per modificare qualunque elemento della procedura.

Tutti i file generati dal *Wizard* sono inseriti in un archivio compattato (*zip* file). Sarà poi compito dell'utente, al termine, inviare il file *zip* creato e il codice sorgente del modello di *Machine Learning* da integrare agli amministratori DAMEWARE.

Parallelamente, il componente software denominato "DAMEWARE *Plugin Manager*" permette ad un amministratore di integrare fisicamente il codice del modello di *Machine Learning* di un utente all'interno dell'infrastruttura DAMEWARE.

L'applicazione ha il compito di estrarre dal file *zip* ricevuto dall'utente il file di configurazione e il codice java di *wrapping*, compilare e creare il file .jar che rappresenterà effettivamente il *plugin* da eseguire.

In seguito l'applicazione aggiungerà le informazioni necessarie all'interno del database e sposterà i file restanti all'interno di una directory prestabilita. Sarà invece compito dell'amministratore, compilare il codice sorgente ricevuto dall'utente e spostare il file eseguibile creato nella prestabilita cartella di destinazione. Al termine di queste operazioni il modello sarà effettivamente utilizzabile dall'intera utenza.

L’applicazione deve essere compatibile con diversi sistemi operativi, e deve essere facilmente installabile su ognuno di essi. A tale scopo, si è scelto di implementarla con tecnologia Java.
Infatti il compilatore Java genera un formato di file neutrale rispetto all’architettura del sistema ospite, cioè tale che il codice generato possa essere eseguito su diversi processori purché dotati del JRE (*Java Runtime Environment*) installabile gratuitamente da internet. Il compilatore Java permette di ottenere questa neutralità generando istruzioni *bytecode* che non dipendono da una determinata architettura del computer. [1]

5.1 Superamento dei vincoli progettuali iniziali

Affinché i nuovi modelli siano effettivamente *pluggabili*, è indispensabile che il componente FW possa caricare a *runtime* nuove funzionalità e poterle eseguire. L’idea seguita è quella di compilare il codice di *wrapping* ed inserirlo in un archivio *jar*, eseguibile su richiesta.
La necessità di caricare nuove classi a *runtime* ha evidenziato l’esigenza di apportare delle modifiche alle relazioni che legano il componente *Framework* con le altre componenti dell’architettura del sistema, nonché al modo in cui il *Framework* carica ed esegue le classi richieste per un esperimento.
L’attuale architettura utilizza le potenzialità offerte dal *Bridge Pattern* per favorire il riutilizzo del codice. Il principale svantaggio di tale approccio deriva dalla necessità di ricompilare il codice, in quanto l’aggiunta di una nuova funzionalità comporta la modifica delle classi **Classification**.java, **Regression**.java o **Clustering**.java del *package* DMM (Figura 6), che gestiscono la corrispondenza Funzionalità – Modello (1.2).
Non è quindi più possibile utilizzare questo tipo di approccio per le finalità prefissate.
Le classi del nuovo *plugin* devono essere strutturate in modo che siano il più indipendenti possibile dalle altre componenti dell’architettura, e allo stesso tempo devono poter utilizzare i servizi messi a disposizione dalle altre componenti del sistema.
Il *package* DMM originario (Figura 7) non risulta quindi più efficace agli scopi prefissati. I nuovi *plugin* non sfrutteranno più il *Bridge Pattern*, e di conseguenza, le classi create per implementarlo. Del componente DMM, l’unica classe ancora utilizzabile dai nuovi *plugin* sarà la classe **Visualization**, responsabile dei *plot* dei file del *plugin* .

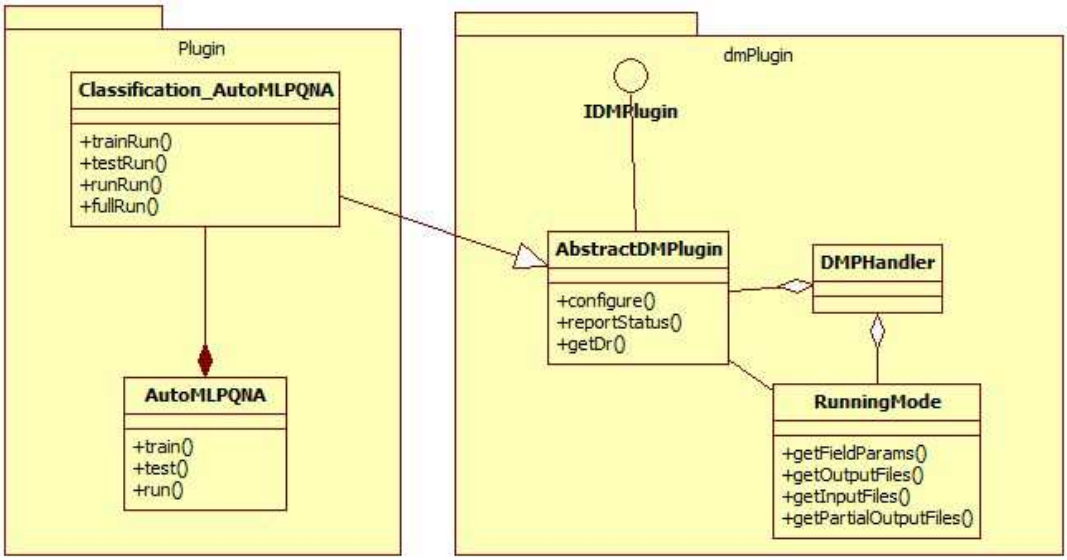

Figura 42 - Struttura di un *Plugin* e relazione con il componente DM*Plugin*

Ciascun *plugin* contiene due classi, una relativa alla funzionalità e l'altra relativa al modello.

L'associazione Funzionalità – Modello, quindi, è ancora valida, ma si tratta di un tipo di associazione diretta.

Nella precedente versione il FW caricava una funzionalità che, tramite le classi ***Supervised*** e ***Unsupervised*** del componente DMM, utilizzava l'opportuna classe relativa ad un modello.

Nel caricamento dei nuovi *plugin* (Figura 43), invece, il file *jar* che viene caricato dal FW contiene sia la classe che implementa la funzionalità, sia la classe che implementa il modello, senza quindi passare tramite le classi del *bridge pattern*.

Questo tipo di approccio non favorisce il riutilizzo del codice, a vantaggio del superamento dei vincoli sulle modifiche delle classi ***Supervised*** e ***Unsupervised,*** imposti dal *Bridge pattern*.

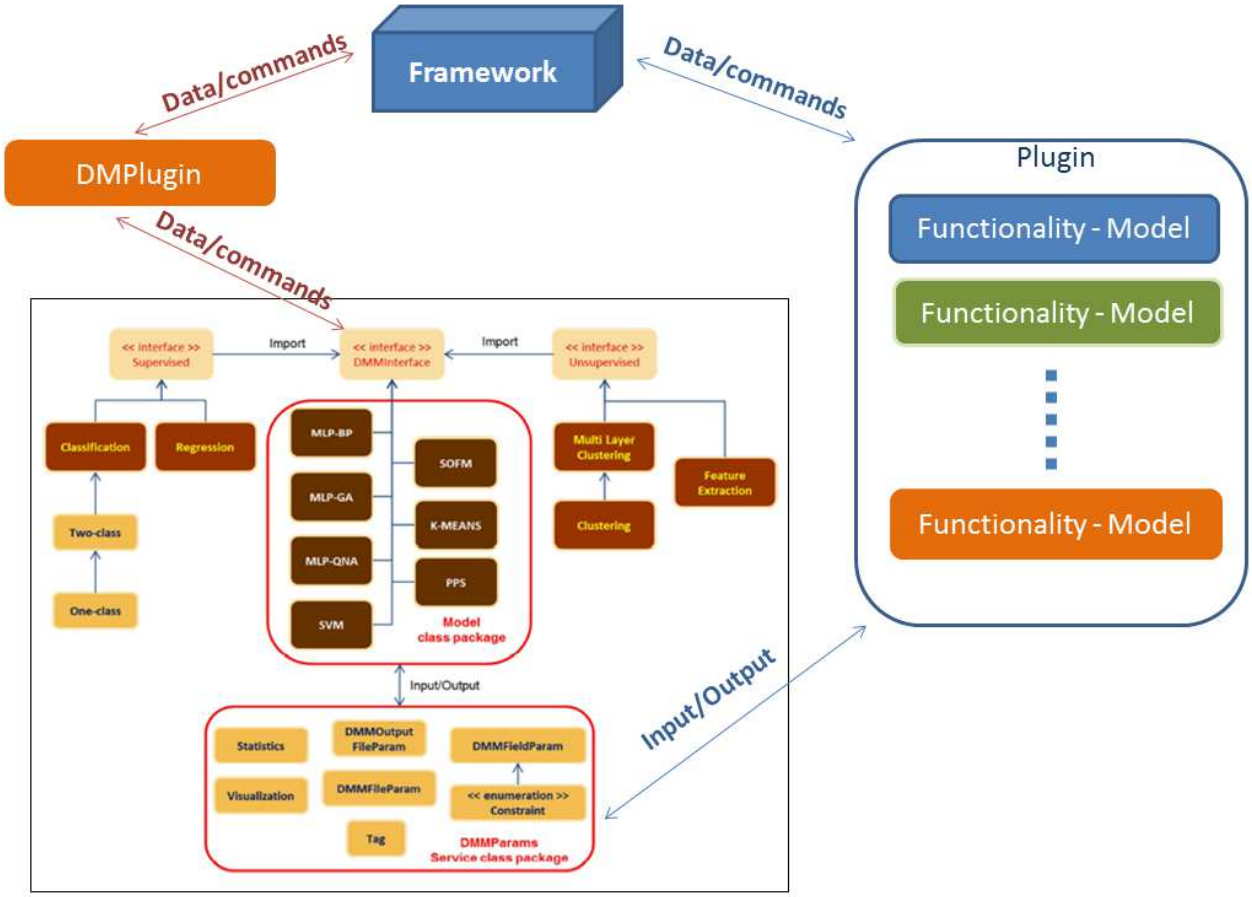

Figura 43 - Confronto tra vecchia e nuova strategia di *plugin*

5.2 Modifiche al componente Framework

La *Riflessione* è uno strumento messo a disposizione dal linguaggio Java, per poter scrivere programmi che elaborano dinamicamente il codice Java.

Il meccanismo della riflessione è estremamente efficace per permettere in particolar modo di analizzare le funzionalità delle classi a *runtime* e di ispezionarne gli oggetti [1].

Nel *Framework* della *web application* questo potente meccanismo è stato impiegato per poter istanziare il costruttore di un oggetto caricato a *runtime* ed eseguirne i suoi metodi.

La precedente versione del *Framework* adottava già questo meccanismo, per stabilire le classi da istanziare per avviare un esperimento, ma non era in grado di caricare le nuove classi relative ad i nuovi *plugin* a *runtime*, richiedendo necessariamente la ricompilazione dell'intera Web Application. Il caricamento a *runtime* di nuove classi è ottenuto combinando la potenzialità della

riflessione con un *Class Loader*, utilizzando la classe **URLClassLoader** della libreria standard Java.

Un *Class Loader*[4] è un oggetto che è responsabile del caricamento delle classi.

Dato un nome di una classe, un *Class Loader* tenta di individuare i dati che costituiscono una classe. Tutti gli oggetti di tipo *Class* contengono un riferimento al *Class Loader* che li definisce.

La classe **URLClassLoader**[5] è utilizzata per caricare classi a partire da *path* che possono riferirsi sia a file *jar*, sia a directory.

5.3 La generazione del codice Java

L'applicazione produce in output, tra gli altri, due file Java, uno denominato <Nome modello>.java con il compito principale di caricare e ordinare tutti i parametri ricevuti ed eseguire l'esperimento, l'altro, denominato <Funzionalità>_<Nome Modello>.java con il compito di ricevere le informazioni passata dal componente FE, ed effettuare le operazioni richieste sia su file di input (ad esempio tradurre il file di input da un formato ad un altro), sia sui file di output (*plot*, generazione della matrice di confusione, unione di più file, creazione del file *log*, ecc..).

La generazione dei file citati, è eseguita con l'ausilio del *package Freemarker* al fine di avere un maggior controllo da un punto di vista implementativo sul codice generato.

Freemarker è un "*template engine*": uno strumento per la generazione di file di output utilizzando dei *Template*.

Freemarker è stato progettato principalmente per la generazione di pagine web HTML, in modo particolare per le applicazione *server-based* che seguono il *pattern* MVC (*Model View Controller*). L'idea seguita da tale *pattern* è la possibilità di separare il *design* di una pagina (ad esempio il codice HTML) dal suo contenuto.

La scelta di sfruttare le potenzialità rese disponibili dal *package Freemarker* per la generazione dei file Java, è stata dettata dal fatto che ciascun *plugin* conserva una struttura interna, dal punto di vista implementativo, indipendente dai modelli.

Quindi è possibile separare la struttura di un *plugin* (costruttore e metodi, la cui struttura è indipendente dal modello) dal suo contenuto (informazioni relative ad i parametri di input, file di input, costanti e file di output, dipendenti da ogni singolo modello) come mostrato in Figura 44.

[4] http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/ClassLoader.html
[5] http://docs.oracle.com/javase/1.4.2/docs/api/java/net/URLClassLoader.html

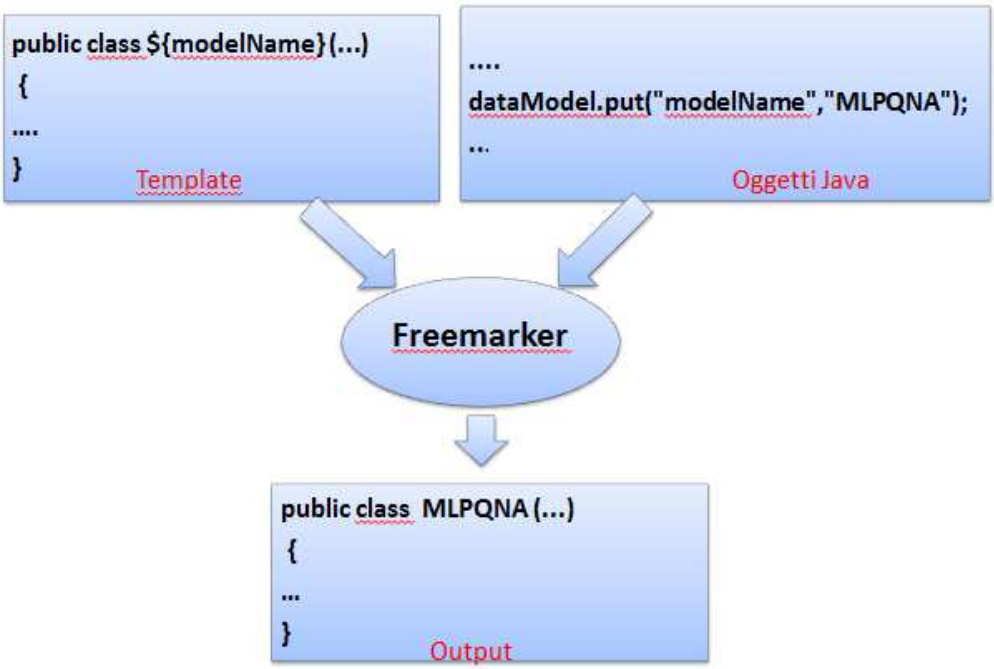

Figura 44 – *Freemarker*

L'approccio utilizzato per la generazione di file Java è stato utilizzato anche per la generazione del file XML d'interfaccia con il componente FE, in quanto tale file presenta le medesime caratteristiche descritte in precedenza.

5.4 Salvataggio e caricamento delle informazioni di configurazione di un modello

Le informazioni inserite all'interno del *Plugin Wizard* possono essere in qualunque momento salvate all'interno di un file XML (Figura 33) al fine di poter essere caricate in un secondo momento e continuare con l'aggiunta di nuove informazioni.
Tale file viene gestito con l'ausilio del *package* org.jdom.
Il *package jdom* fornisce una soluzione completa e *Java-Based* per l'accesso, la manipolazione e la creazione di file in formato XML.

5.5 L'interfaccia grafica

L'attuale versione implementata dell'applicazione DAMEWARE *Plugin Wizard* permette, tramite una procedura guidata, di configurare tutti i parametri di un modello di *Machine Learning*, nonché i casi d'uso e le informazioni di I/O ad esso relativi. L'applicazione genera il codice sorgente da inserire fisicamente nell'infrastruttura della *web application* per poter offrire alla comunità una nuova risorsa di *Machine Learning*.

I modelli che possono essere integrati utilizzando la procedura sono degli eseguibili forniti di parametri da linea di comando oppure con parametri contenuti all'interno di file di configurazione.
Un utente, per eseguire l'applicazione, è necessario che abbia installato sulla propria macchina l'ambiente di esecuzione JRE (*Java Runtime Environment*).
Dopo l'installazione di JRE è possibile scaricare il *Plugin Wizard* dall'indirizzo http://dame.dsf.unina.it/beta_info.html**,** ed eseguirla con un *click* col pulsante destro del mouse sul file *jar* e selezionando la voce "*Open with*" e "*Java Platform SE binary*". In alternativa è possibile eseguire il file *jar* direttamente dal *prompt* dei comandi, eseguendo il comando:

> java –jar DAMEWAREPlugin.jar [return]

5.5.1 DAMEWARE *Plugin Wizard*

Di seguito è descritta la GUI di DAMEWARE *Plugin Wizard*. La Figura 45 mostra la schermata principale dell'applicazione. Nella parte bassa delle finestra sono presenti i pulsanti "*Back*" e "*Next*", inizialmente disabilitati, che permettono di procedere (rispettivamente in avanti e indietro) lungo i passi della procedura.
Il pulsante "*New*" permette di avviare una nuova configurazione di un *plugin*, mentre il pulsante "*Load*" permette di caricare una configurazione salvata in precedenza.
Infine il pulsante "*Save*" permette di salvare in qualsiasi momento le informazioni inserite.

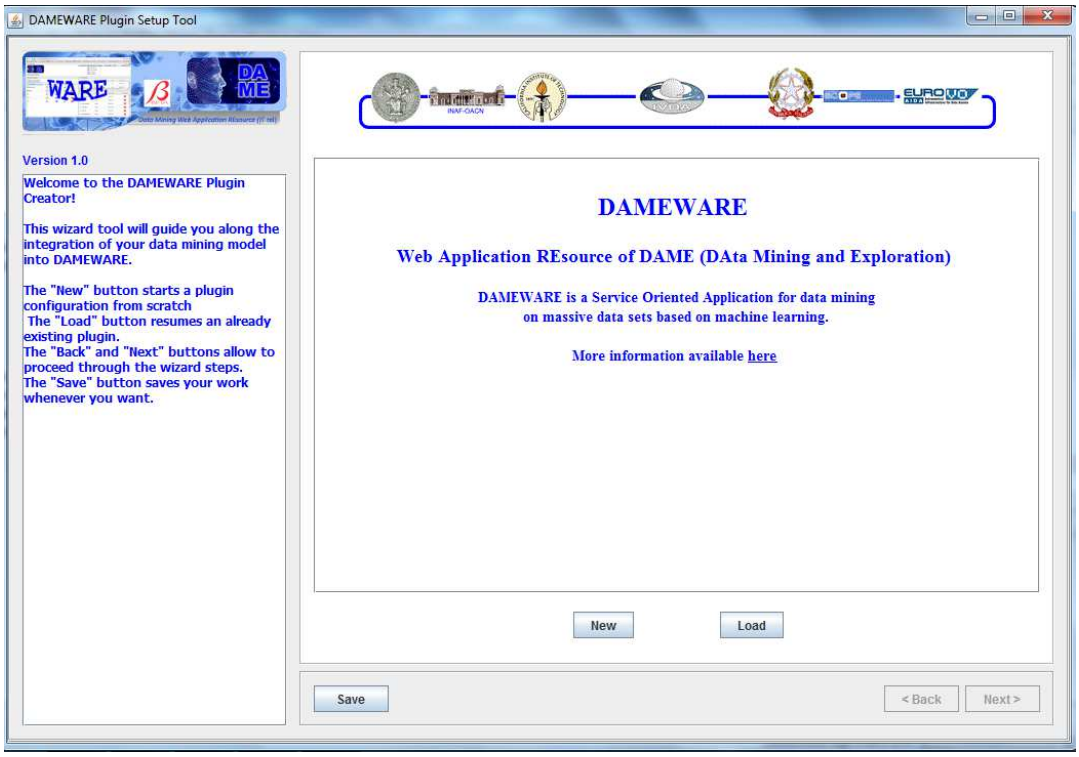

Figura 45 - DAMEWARE *Plugin Wizard*

Il secondo *step* della procedura (Figura 46) prevede l'inserimento delle informazioni generali, in particolare:

- *Model name*: generalmente un acronimo che rappresenti il nome del modello, senza spazi (ad esempio MLP per *Multi Layer Perceptron*);
- *Documentation*: un URI dove trovare la documentazione relativa al modello;
- *Version*;
- *Functionality*: il dominio funzionale da associare al modello (ad esempio classificazione o regressione per i modelli supervisionati, *clustering* per modelli non supervisionati, ecc..);
- *Owner name*: Nome dell'utente;
- *Owner e-mail*: *e-mail* dell'utente;

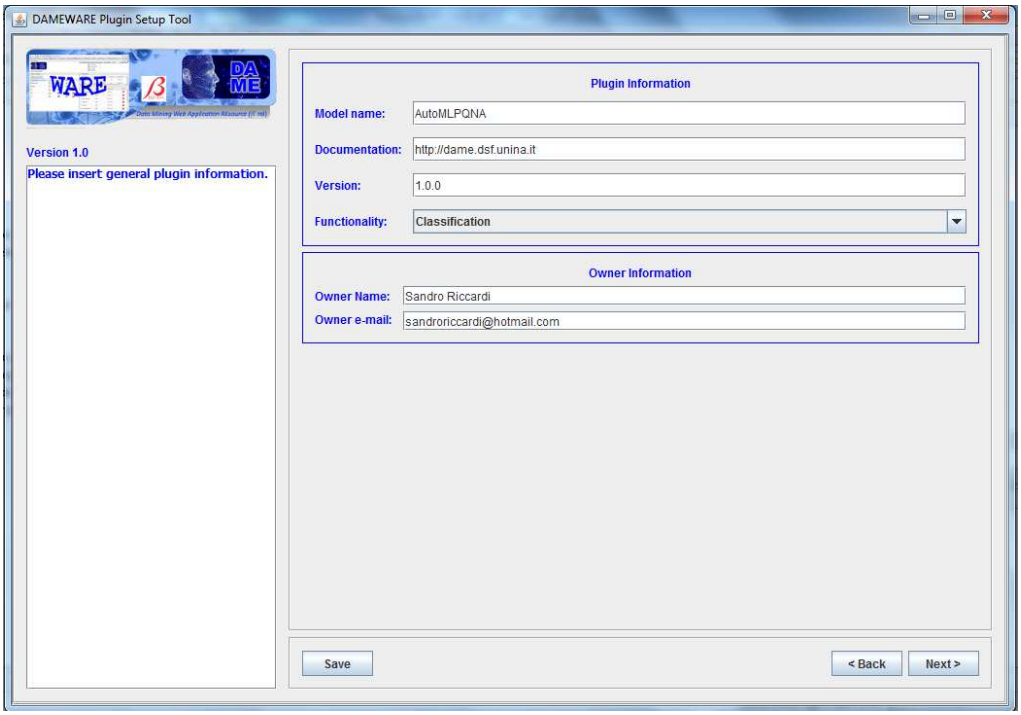

Figura 46 - DAMEWARE *Plugin Wizard*, Informazioni generali

La fase successiva (Figura 47) permette di selezionare i casi d'uso previsti per il modello. Generalmente un esperimento richiede una serie di fasi (casi d'uso) da eseguire. All'inizio di un esperimento è richiesta una fase di *training* (sia nel caso di modelli supervisionati che per i modelli non supervisionati). La seconda fase è generalmente quella di *TEST*; l'ultima, invece, è la fase di *RUN* in cui il modello addestrato e *testato* è usato come una generica funzione applicata a nuovi dati di un problema. E' prevista anche una fase di *FULL* che esegue automaticamente e in sequenza prima la fase di *TRAIN* e consecutivamente quella di *TEST*.

In questa fase, quindi, l'utente può selezionare i casi d'uso che devono essere implementati per il modello da integrare (generalmente un modello di *Machine Learning* prevede tutti e quattro i casi citati), e per ciascuna fase, specificare la documentazione di riferimento.

E' possibile selezionate il caso d'uso *FULL* solo se prima siano stati selezionati i casi d'uso *TRAIN* e *TEST*.

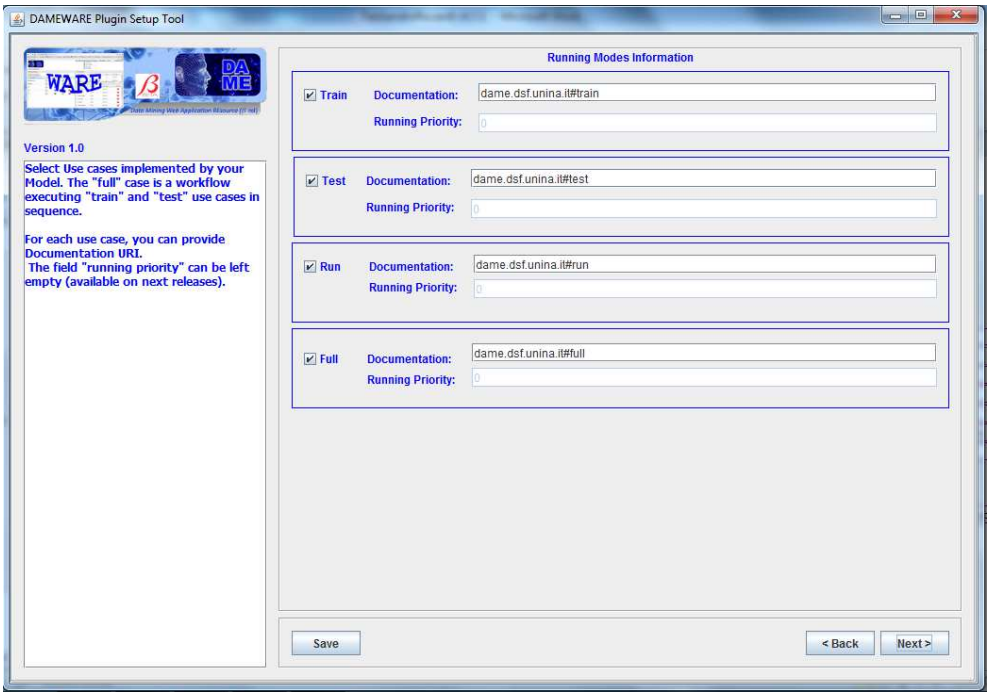

Figura 47 - DAMEWARE *Plugin Wizard*, selezione casi d'uso

Per ciascun caso d'uso selezionato (escluso il caso d'uso *FULL*), la procedura permette di inserire informazioni relative ad i parametri di input, file di input, costanti e file di output.

La Figura 48 mostra il pannello relativo ad i parametri di input per il caso d'uso *TRAIN* (lo stesso pannello verrà mostrato per i casi d'uso *TEST* e *RUN*).

Il pulsante "*Add field*" permette di aggiungere le informazioni relative ad un nuovo parametro in ingresso, "*Edit field*" permette la modifica delle informazioni di un parametro precedentemente inserito, mentre il pulsante "*Remove field*" permette di eliminare un parametro.

I pulsanti "*Up*" e "*Down*" consentono di definire l'ordine con cui i parametri inseriti verranno mostrati all'utente tramite il componente FE.

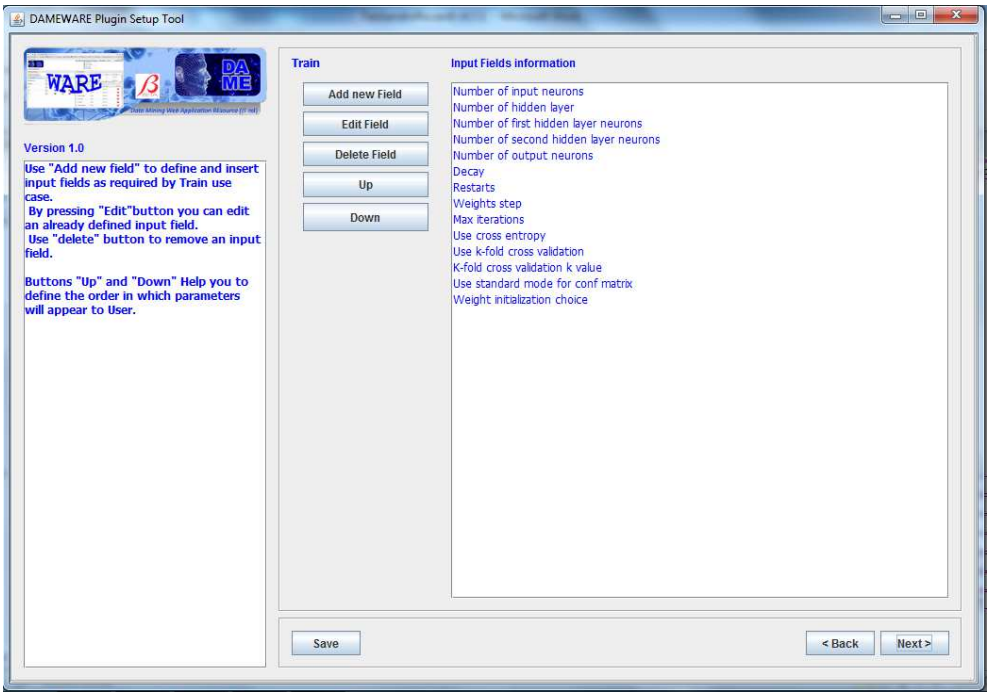

Figura 48 - *Plugin Wizard*, pannello dei parametri di input (esempio per il caso d'uso *TRAIN*)

Selezionando il pulsante "*Add new field*", una nuova finestra permetterà all'utente di inserire le nuove informazioni relative ad un nuovo parametro di input, in particolare:

- *Visualized label*: l'etichetta del parametro che comparirà nella *web application* DAMEWARE (ad esempio "*Number of input neurons*");
- *Description*: Una breve descrizione del parametro che comparirà come *tooltip* nella *web application*;
- *Type*: Il tipo del parametro (*Integer*, *Double*, *String*, *Boolean*);
- *Is Optional*: stabilisce se il parametro non sia obbligatoriamente richiesto (se omesso, il parametro assumerà il valore di *default* specificato nel campo "*Default value*";
- *Default value*: il valore di *default* che assume il parametro di tipo definito nel campo Type
- *Ucd* (opzionale): per uniformità con gli standard VOTABLE;
- *Unit* (opzionale): specifica l'unità di misura del parametro, ad esempio unit = m2 p"r m^2, unit="cm"2.s-1.keV-1" for $cm^{-2}s^{-1}keV^{"1}$, or "unit="erg/s" for erg s^{-1};
- *Precision* (opzionale): il valore *floating point* che indica la precisione della rappresentazione interna del parametro;
- *Constrain*: Se il parametro prevede un insieme di valori consentiti, scegliendo tra un insieme di valori, un intervallo, oppure nessun vincolo (NO CONS*TRAIN*);
- *Constrain values*: Se è stata selezionata l'opzione "*VALUES*" questo campo conterrà l'insieme dei possibili valori separati da un virgola (per esempio 0,1,2): Se è stata

selezionata l'opzione "*RANGE*" questo campo conterrà gli estremi dell'intervallo separato da ":" (per esempio 0.1 : 0.9);

- *Add to Use Case Configuration File***:** permette di definire se il parametro debba essere inserito all'interno di un file di configurazione del caso d'uso;
- *Add to Neural Network Configuration File*: permette di definire se il parametro debba essere inserito all'interno di un altro file di configurazione;

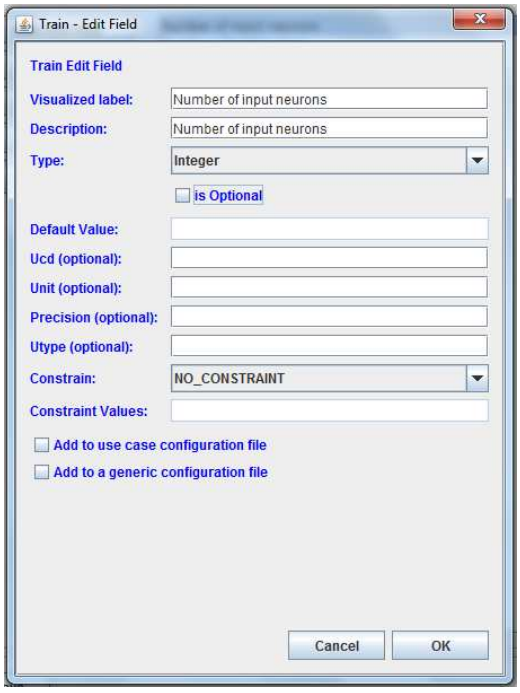

Figura 49 - *Plugin Wizard*, inserimento dei parametri di input

Un altro pannello della GUI permette l'inserimento delle informazioni relative ad i file di input (Figura 50).

Il pulsante "*Add input file*" permette di aggiungere le informazioni relative ad un nuovo file di input, "*Edit input file*" permette la modifica delle informazioni di un file di input precedentemente inserito, mentre il pulsante "*Remove input file*" permette di eliminare un file di input.

I pulsanti "*Up*" e "*Down*" consentono di definire l'ordine con i quali i file di input verranno mostrati all'utente tramite il componente FE.

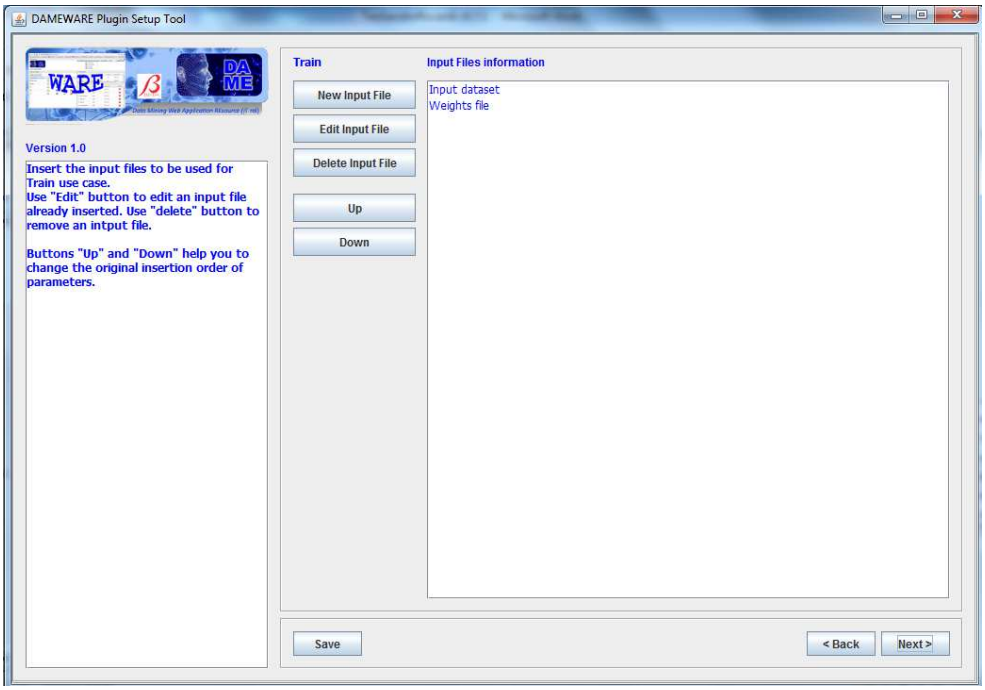

Figura 50 - *Plugin Wizard*, pannello dei file di input

Il pulsante "*Add new input file*" permette di aggiungere le informazioni relative ad un file di input (Figura 51), in particolare:

- *Visualized label*: l'etichetta del file di input che comparirà nella *web application* DAMEWARE (ad esempio "Input *dataset*");
- *Description*: Una breve descrizione del file di input che comparirà come *tooltip* nella *web application*;
- *Is optional*: Se selezionato, il file di input verrà considerato come non necessariamente richiesto
- *Convert to*: Se il modello richiede un particolare formato per il file di input (ad esempio il file di input deve essere necessariamente in formato CSV), l'utente può scegliere il corretto formato selezionando uno dei formati disponibili;
- *Split input columns from target columns*: se il modello richiede di separare le colonne di un file di input (ad esempio può essere necessario dividere un *dataset* di input in *inputs* e *targets* utilizzando il numero di neuroni in input come criterio);
- *Add to Use Case Configuration File*: permette di definire se il nome del file di input debba essere inserito all'interno di un file di configurazione del caso d'uso;
- *Add to Neural Network Configuration File*: permette di definire se il nome del file di input debba essere inserito all'interno di un altro file di configurazione;

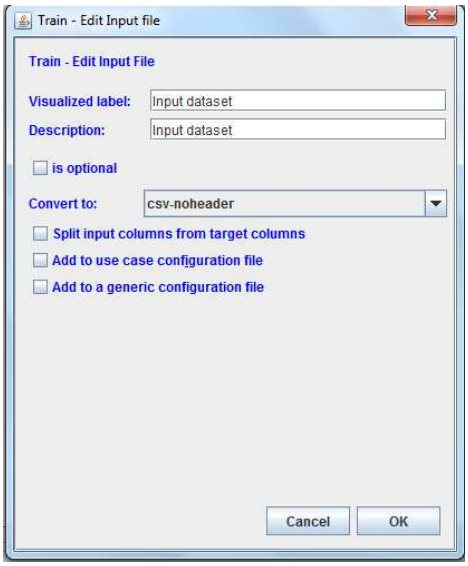

Figura 51 - *Plugin Wizard*, inserimento dei file di input

Ogni volta che un esperimento termina, produce in output dei file. Il pannello mostrato in Figura 52 permette di definire i file di output previsti per ciascun caso d'uso.

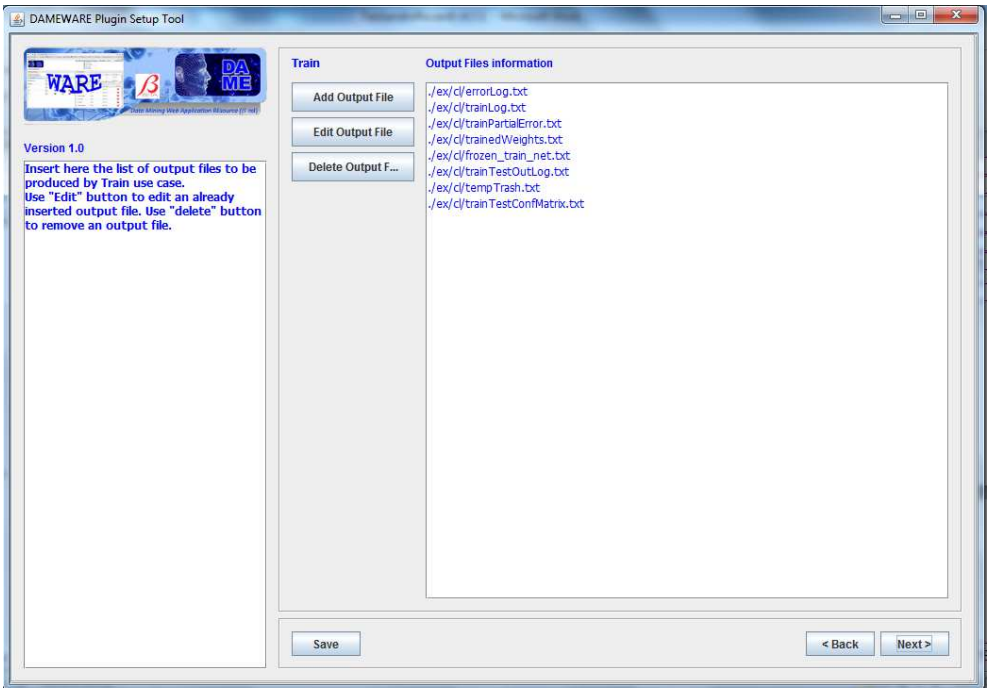

Figura 52 - *Plugin Wizard*, pannello dei file di output

Il pulsante "*Add new output file*" permette l'inserimento delle informazioni relative ad i file di output (Figura 53).

Par ciascun file di output, le informazioni da inserire sono:

- *URI*: il *path* relativo in cui sarà creato il file di output;

- *Description*: una breve descrizione del file;
- *Format*: il formato del file;
- *Rename in*: il nome finale del file;
- *Is Partial*: se il file conterrà informazioni parziali;
- *Plot this file*: permette di plottare il contenuto del file ;
- *Append to log file*: specifica se il contenuto del file deve essere inserito nel file di *log*;
- *Generate confusion matrix in output*: i valori di output di un esperimento possono essere rappresentati tramite matrice di confusione;
- *Join this file with another output file*: è possibile unire le colonne del file con quelle di un altro file di output;
- *Join this file with an input file*: è possibile aggiungere alle colonne del file di output, delle colonne di un file di input;

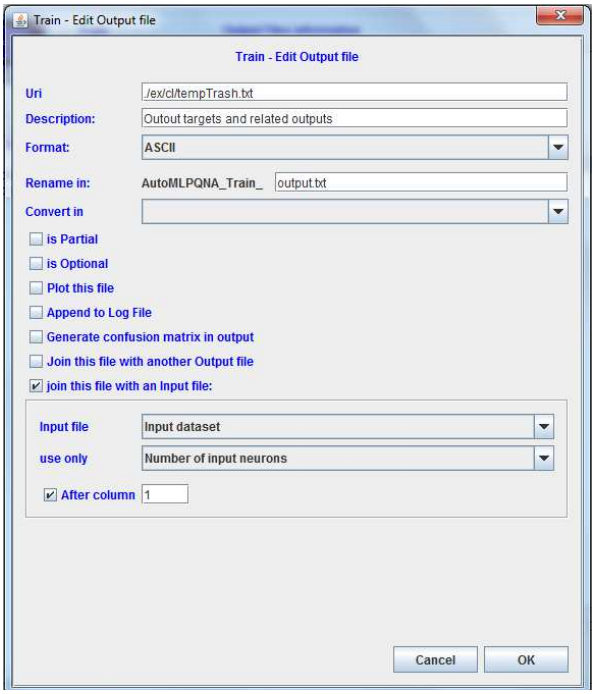

Figura 53 - *Plugin Wizard*, inserimento dei file di output

L'ultimo pannello relativo a ciascun caso d'uso permette di definire l'ordine con cui i parametri saranno passati a linea di comando (Figura 54) oppure all'interno dei file di configurazione (Figura 55). In questo *step* è possibile anche aggiungere eventuali costanti.

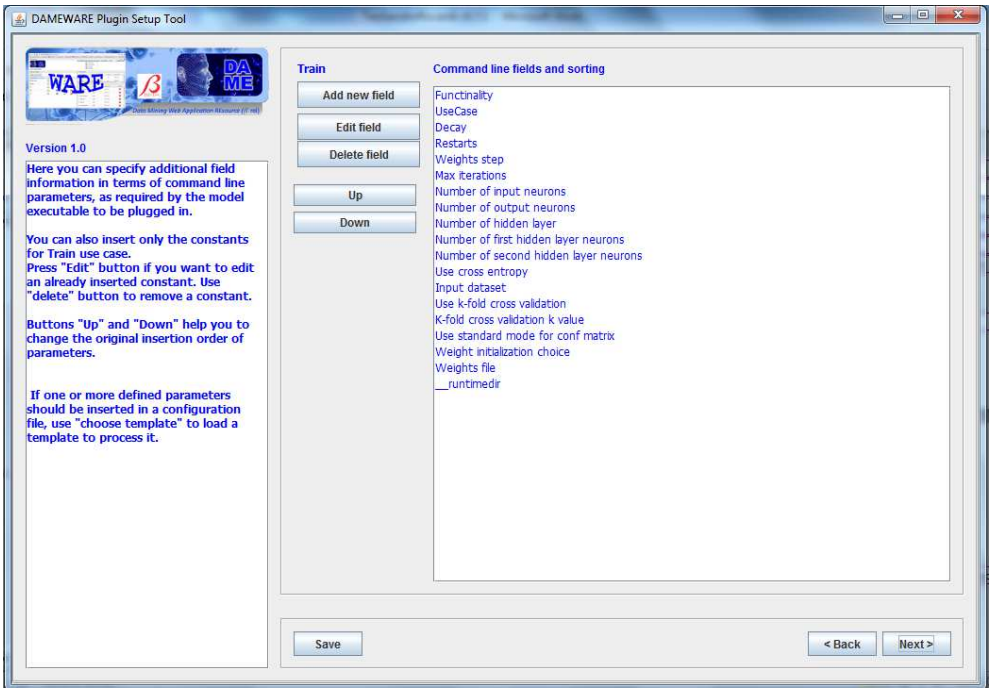

Figura 54 - *Plugin Wizard*, caso parametri a linea di comando

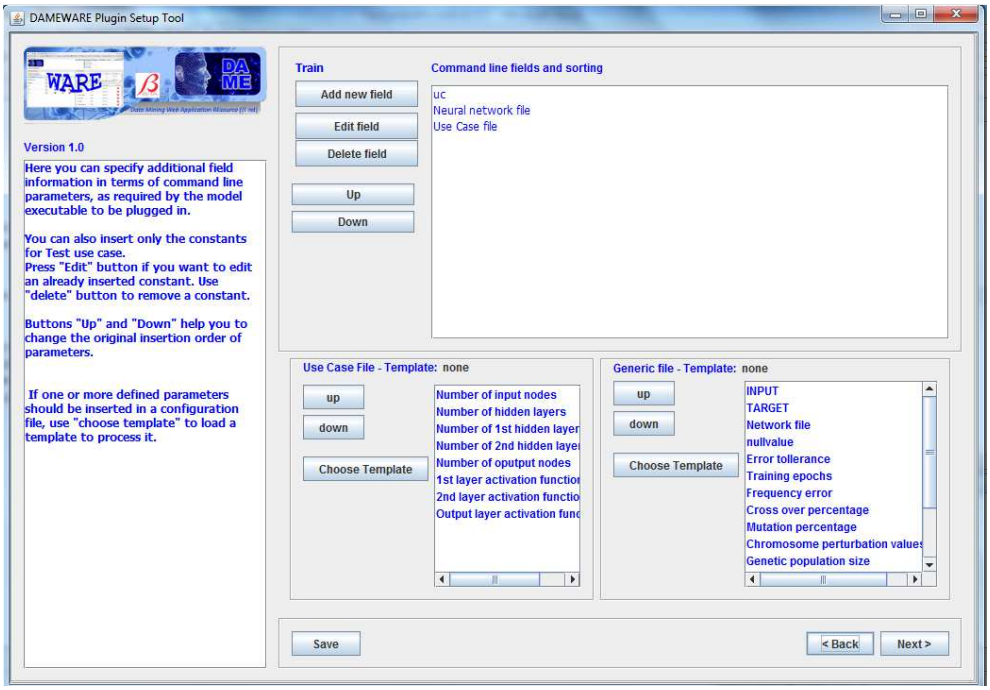

Figura 55 – *Plugin Wizard*, caso parametri in file di configurazione

Il pulsante "*Add constant*" permette, quindi, di aggiungere le informazioni relative ad una costante (Figura 56):

- *Parameter name*: il nome del parametro;
- *Type*: il tipo (*Integer, Double, String, Boolean*);

- *Value*: il valore della costante;
- *Add to Use Case Configuration File*: permette di definire se la costante debba essere inserita all'interno di un file di configurazione del caso d'uso;
- *Add to Neural Network Configuration File*: permette di definire se la costante debba essere inserita all'interno di un altro file di configurazione;

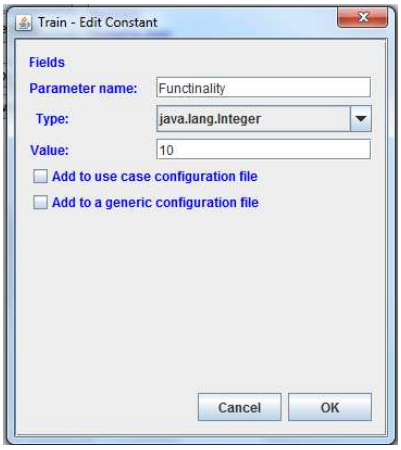

Figura 56 – *Plugin Wizard*, inserimenti delle costanti

Il caso d'uso *FULL*, come detto in precedenza, è una fase particolare di un esperimento in cui le fasi *TRAIN* e *TEST* sono eseguite in sequenza. Se è stato selezionato il caso d'uso *FULL*, il pannello mostrato in Figura 57 permette di specificare la corrispondenza dei parametri di input di *TRAIN* con quelli di *TEST* e quali file di output del caso d'uso *TRAIN* vanno passati come file di input al caso d'uso *TEST*. In un comune esperimento di *Machine Learning*, infatti, alcuni output della fase di *TRAIN* diventano gli input della fase di *TEST* (per esempio in una rete neurale, i pesi della rete salvati durante la fase di *TRAIN* in un file di output, sono passati in input ad un esperimento di *TEST*).

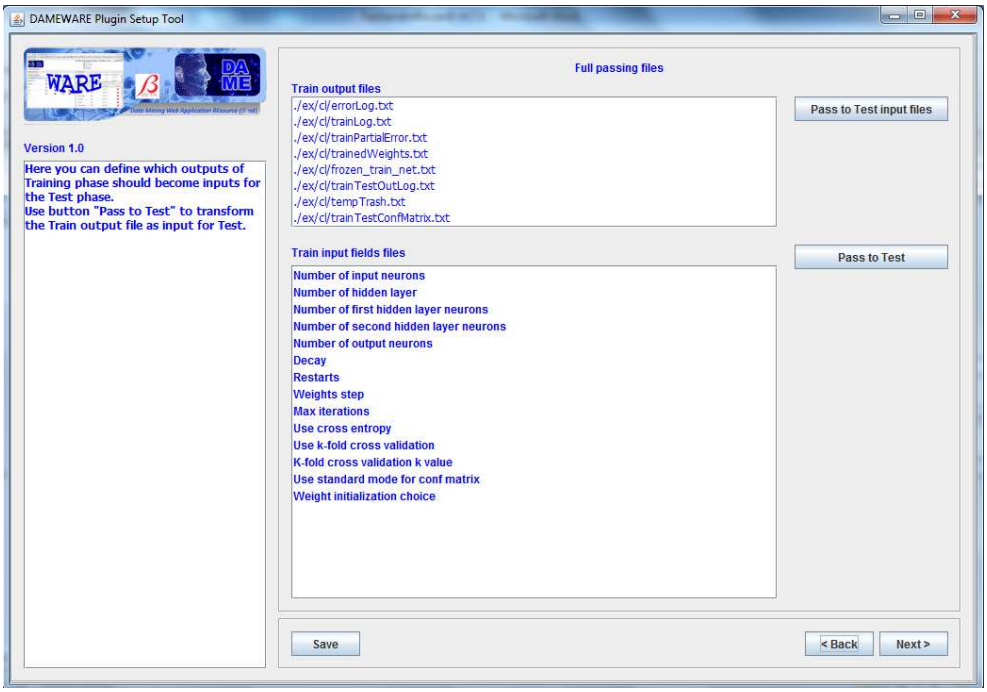

Figura 57 – *Plugin Wizard*, pannello per caso d'uso *FULL*

Alla fine della procedura, dopo aver configurato tutti i file di input e di output, tutti i parametri e le costanti, è necessario definire il nome dell'eseguibile da lanciare (Figura 58).

Il pulsante "*Finish*" permette di terminare la procedura con la creazione del file *zip* di nome <funzionalità>_<nome modello>.*zip* contenente:

- Un file Java denominato <funzionalità>_<nome modello>.java
- Un file Java denominato <nome modello>.java
- Un file XML denominato <funzionalità>_<nome modello>.xml
- Un file XML denominato ConfigurationFile.xml utile per replicare le fasi della procedura;
- Un eventuale *Template* per il file di configurazione del caso d'uso;
- Un eventuale *Template* per un generico file di configurazione;

Un messaggio confermerà la creazione del file *zip* (Figura 59).

L'utente infine, invia il file *zip* creato dalla procedura, insieme al codice sorgente del modello (da ricompilare sulla macchina DAME), agli amministratori di DAMEWARE (helpdame@gmail.com) che provvederanno a completare l'integrazione.

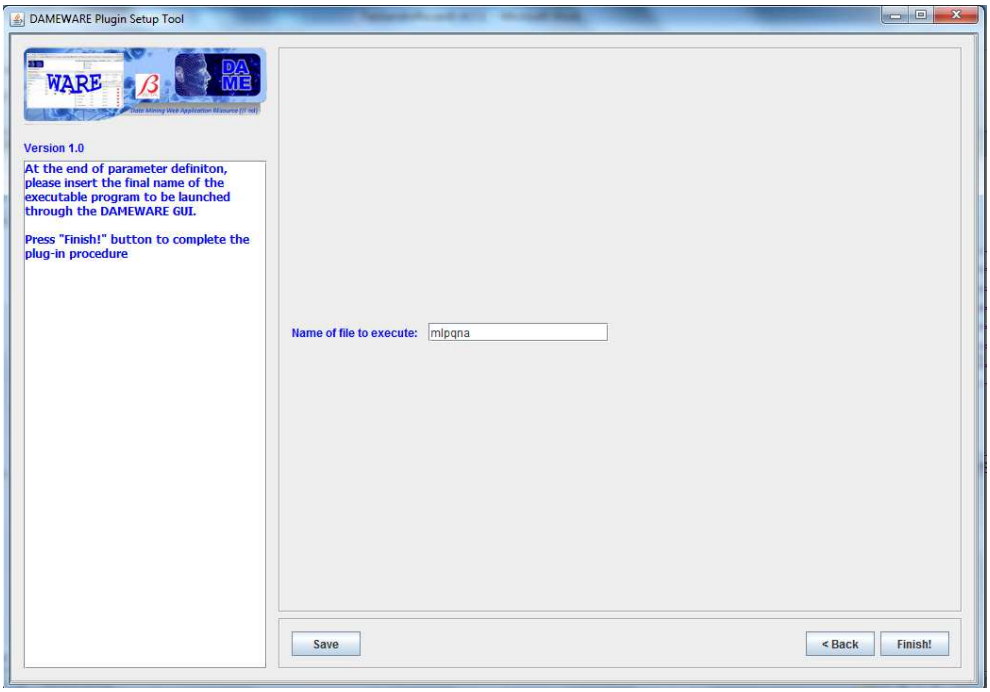

Figura 58 – *Plugin Wizard*, pannello finale

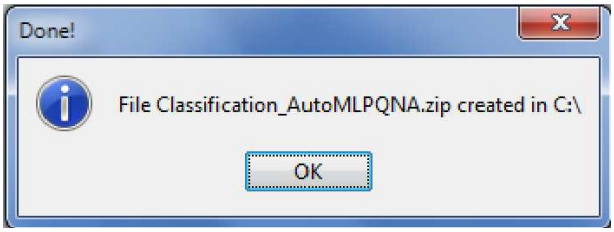

Figura 59 – *Plugin Wizard*, avviso di terminazione della procedura

5.5.2 DAMEWARE *Plugin Manager*

DAMEWARE *Plugin Manager* è l'applicazione che si occupa della compilazione del codice Java ricevuto dall'utente, e dell'integrazione fisica del *plugin* all'interno della suite DAMEWARE.
Di seguito è descritta la GUI del *Plugin Manager*.
La Figura 60 mostra la schermata che compare con l'esecuzione del file Dameware*PluginManager*.jar.

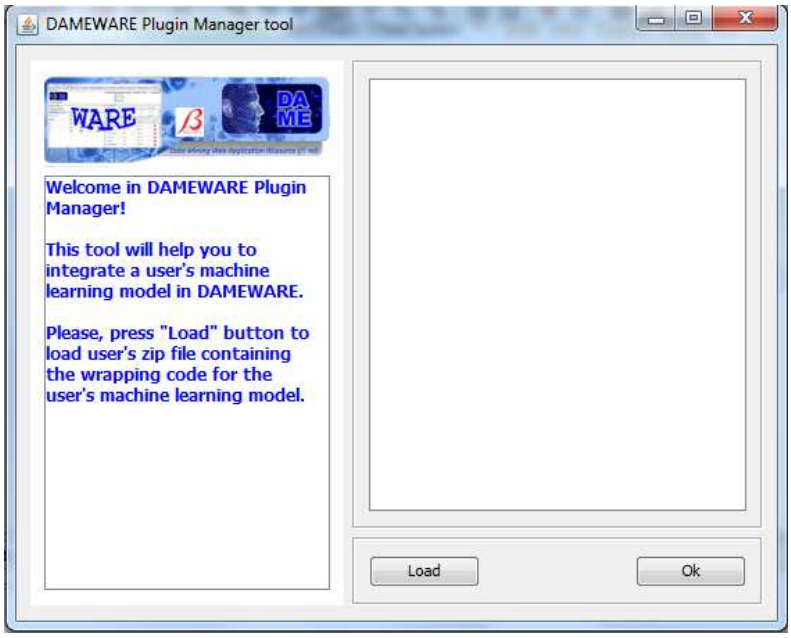

Figura 60 – *Plugin Manager*

Nella parte bassa della finestra, il pulsante "*Load*" permette di caricare il file *zip* ricevuto dall'utente contenente il file ConfigurationFile.xml, i due file Java del *plugin*, il file XML di interfaccia con il FE e gli eventuali *Template* per i file di configurazione, se previsti dal modello.
Dopo aver scompattato il file, l'applicazione utilizza il file ConfigurationFile.xml per raccogliere le informazioni relative al *plugin* da integrare come:

- Nome della funzionalità;
- Nome del modello associato;
- Il *link* relativo alla documentazione sul modello;
- Il nome e *l'e-mail* del proprietario;
- I casi d'uso che saranno implementati;

Inoltre è controllata l'effettiva esistenza dei file Java e del file XML per il FE.
Saranno mostrate nel quadrante bianco le informazioni lette, o gli eventuali errori riportati (Figura 61).

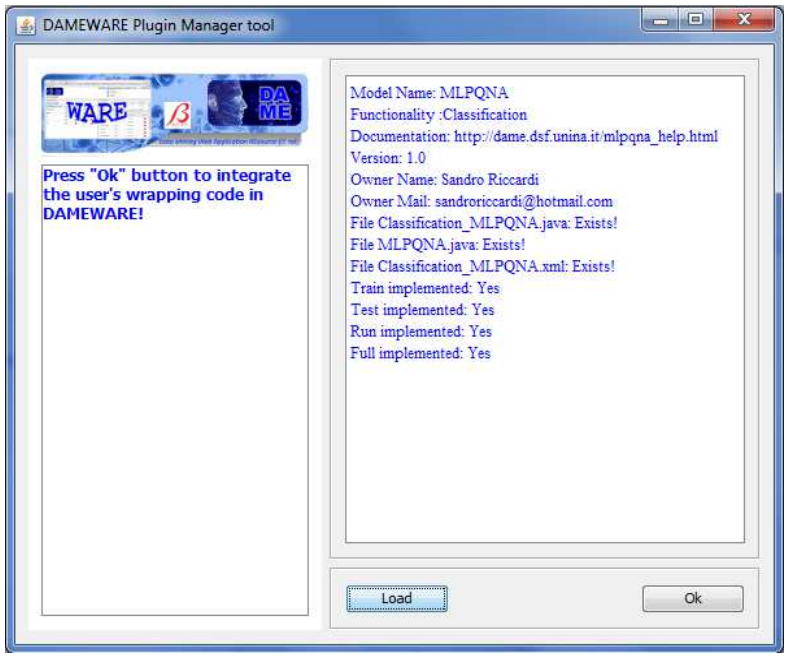

Figura 61 – *Plugin Manager*, controllo file zip

Se sono presenti tutti i file e le informazioni necessarie, il pulsante "OK" sarà abilitato e permetterà la compilazione del codice Java e l'effettiva integrazione del *plugin* all'interno della suite DAMEWARE (Figura 62).

Figura 62 - *Plugin Manager*, messaggio d'integrazione terminata

6 *Testing*

Le seguenti tabelle permettono di confrontare gli output prodotti dai modelli già pre-esistenti all'interno della *web application* DAMEWARE (integrati cioè con la tecnologia tradizionale), con quelli prodotti dai modelli integrati utilizzando il *Plugin Wizard* per ciascuna funzionalità (classificazione e regressione) e per ciascun caso d'uso (*TRAIN*, *TEST*, *RUN* e *FULL*) per i modelli MLPQNA e MLPGA.
In sintesi questi modelli sono due varianti della nota rete neurale *Multi Layer Perceptron* (MLP), rispettivamente, addestrata con la regola del *Quasi Newton* (QNA, approssimazione della matrice Hessiana dell'errore di apprendimento) e con un algoritmo genetico (GA, algoritmo evolutivo ispirato alle leggi darwiniane di ottimizzazione della specie). Entrambi i modelli mantengono la medesima struttura topologica e meccanismo di flusso delle informazioni, variando ovviamente la modalità di apprendimento, con conseguente diversità nel numero e nella quantità di informazioni di setup. Quest'ultimo particolare ha ovviamente un impatto sulla valutazione degli strumenti di *plugin*, permettendone quindi la verifica esaustiva per tutti i casi d'uso previsti.

Il modello MLPQNA (2.2) è un tipico modello i cui parametri sono tutti passati a linea di comando all'applicazione.
L'ordine dei parametri e la loro descrizione sono riportate nei paragrafi 2.2.1, 2.2.2 e 2.2.3, rispettivamente per i caso d'uso *TRAIN*, *TEST* e *RUN*.
Un esempio di una esecuzione di una fase di *TRAIN* con ilo modello MLPQNA è:

./mlpqna 10 3 0.001 10 0.01 1000 7 1 2 15 6 0 datasets/agn_7_stat_full.txt 0 10 1 704 ./trainedWeights.txt

Le tabelle Tabella 1, Tabella 2, Tabella 3, Tabella 4 mettono a confronto gli output prodotti effettuando una classificazione con il modello MLPQNA, rispettivamente per i casi d'uso *TRAIN*, *TEST*, *RUN* e *FULL*, per il *plugin* pre-esistente nella suite (colonne di sinistra) e il *plugin* creato con DAMEWARE *Plugin Wizard*.

<table>
<tr><td colspan="4">CLASSIFICAZIONE
TRAIN con MLPQNA</td></tr>
<tr><td colspan="4">Parametri di input:
Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii
Number of input neurons: 4
Number of first hidden layer neurons: 9
Number of output neurons: 1
Tutti gli altri parametri assumono valori di *default* assegnati dal sistema</td></tr>
<tr><td colspan="2">Versione Manuale</td><td colspan="2">Con *WIZARD*</td></tr>
<tr><td colspan="2">Nome esperimento: ClmlpqnaTR01</td><td colspan="2">Nome esperimento: CLwizMLPQNA*train*01</td></tr>
<tr><td>Nome:</td><td>Descrizione:</td><td>Nome:</td><td>Descrizione:</td></tr>
<tr><td>mlpqna_*TRAIN*_error.txt</td><td>File degli errori parziali, 3 colonne: *Step* di *training*, numero di iterazioni, errore</td><td>AutoMLPQNA_*Train*_error.txt</td><td>File degli errori parziali, 3 colonne: *Step* di *training*, numero di iterazioni, errore</td></tr>
<tr><td>mlpqna_*TRAIN*_output.txt</td><td>Valori di output con in aggiunta i valori del *dataset* di input</td><td>AutoMLPQNA_*Train*_output.txt</td><td>Valori di output con in aggiunta i valori del *dataset* di input</td></tr>
<tr><td>mlpqna_*TRAIN_trainTest*OutLog.txt</td><td>Valori di output calcolati al termine della fase di *training* con rispettivi target</td><td>AutoMLPQNA_*Train_trainTest*OutLog.txt</td><td>Valori di output calcolati al termine della fase di *training* con rispettivi target</td></tr>
<tr><td>mlpqna_*TRAIN*_frozen_net.txt</td><td>Valori dei nodi interni della rete da dare in input alla rete nei casi di *TEST* e *RUN*</td><td>AutoMLPQNA_*Train*_frozenNet.txt</td><td>Valori dei nodi interni della rete da dare in input alla rete nei casi di *TEST* e *RUN*</td></tr>
<tr><td>mlpqna_*TRAIN*_weights.txt</td><td>Pesi finali della rete</td><td>AutoMLPQNA_*Train*_weights.txt</td><td>Pesi finali della rete</td></tr>
<tr><td>mlpqna_*TRAIN.log*</td><td>File di *log* sullo stato dell’esperimento, contiene data e ora di inizio e fine esperimento e il contenuto del file</td><td>AutoMLPQNA_*Train.log*</td><td>File di *log* sullo stato dell’esperimento, contiene data e ora di inizio e fine esperimento e il</td></tr>
</table>

	*train*Log.txt prodotto dal modello		contenuto del file *train*Log.txt prodotto dal modello
mlpqna_*TRAIN*_errorPlot.txt	*Plot* della prima e della terza colonna del file mlpqna_*TRAIN*_error.txt	AutoMLPQNA_*Train*_errorPlot.txt	*Plot* della prima e della terza colonna del file AutoMLPQNA_*Train*_error.txt
*dataset_test*_20.ascii_mlpqna_*Train*_ConfMatrix.txt	Matrice di confusione	AutoMLPQNA_*Train*_ConfMatrix.txt	Matrice di confusione
MLPQNA_*Train*_params.xml	File di configurazione dell'esperimento	AutoMLPQNA_*Train*_params.xml	File di configurazione dell'esperimento

Tabella 1 - Classificazione, *TRAIN* conMLPQNA e AutoMLPQNA – confronto

CLASSIFICAZIONE *TEST* con MLPQNA			
Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii *Train*ed network setup file: ./mlpqna_*TRAIN*_frozen_net.txt (prodotto dal caso d'uso *TRAIN*) *Train*ed network weights file: ./mlpqna_*TRAIN*_weights.txt (prodotto dal caso d'uso *TRAIN*) Number of input neurons: 4 Number of first hidden layer neurons: 9 Number of output neurons: 1 Tutti gli altri parametri assumono valori di *default* assegnati dal sistema			
Versione Manuale		Con *WIZARD*	
Nome esperimento: ClmlpqnaTE01		Nome esperimento: CL_WIZ_MLPQNA01	
Nome output prodotto	Descrizione	Nome output prodotto	Descrizione
Mlpqna_*TEST*_*test*ConfMatrix.txt	Matrice di confusione calcolata alla fine della fase di *TEST*	AutoMLPQNA_*TEST*_confMatrix	Matrice di confusione calcolata alla fine della fase di *TEST*
Mlpqna_*TEST*_output.tx	Valori di output con	AutoMLPQNA_*TEST*_outp	Valori di output

t	*dataset* di input	ut.txt	con *dataset* di input
Mlpqna_*TEST.log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.	AutoMLPQNA_*TEST.log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.
Mlpqna_*TEST*_*Test*OutLog.txt	Valori di output calcolati dopo la fase di *TEST* con i rispettivi target	AutoMLPQNA_*TEST*_OutLog.txt	Valori di output calcolati dopo la fase di *TEST* con i rispettivi target
Mlpqna_*TEST*_params-XML	File di configurazione dell'esperimento	AutoMLPQNA_params.XML	File di configurazione dell'esperimento

Tabella 2 - Classificazione, *TEST* conMLPQNA e AutoMLPQNA – confronto

CLASSIFICAZIONE *RUN* con MLPQNA			
Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii *Train*ed network setup file: ./mlpqna_*TRAIN*_frozen_net.txt (prodotto dal caso d'uso *TRAIN*) *Train*ed network weights file: ./mlpqna_*TRAIN*_weights.txt (prodotto dal caso d'uso *TRAIN*) Number of input neurons: 4 Number of first hidden layer neurons: 9 Number of output neurons: 1 Tutti gli altri parametri assumono valori di *default* assegnati dal sistema			
Versione Manuale		Con *WIZARD*	
Nome esperimento: ClmlpqnaRU01		Nome esperimento: CL_WIZ_MLPQNA01	
Nome output prodotto	Descrizione	Nome output prodotto	Descrizione
mlpqna_*RUN.log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine	AutoMLPQNA_*RUN.log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine

	esperimento.		esperimento.
Mlpqna_*RUN*_output.txt	Valori di output con *dataset* di input	AutoMLPQNA_*RUN*_output.txt	Valori di output con *dataset* di input
mlpqna_*RUN*_params.XML	File di configurazione dell'esperimento	AutoMLPQNA_*RUN*_params.XML	File di configurazione dell'esperimento

Tabella 3 - Classificazione, *RUN* conMLPQNA e AutoMLPQNA – confronto

CLASSIFICAZIONE *FULL* con MLPQNA			
Input *dataset*: http perTRAIN://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii Input *dataset*: http per TEST://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii Number of input neurons: 4 Number of first hidden layer neurons: 9 Number of output neurons: 1 Tutti gli altri parametri assumono valori di *default* assegnati dal sistema			
Versione Manuale		Con *WIZARD*	
Nome esperimento: CL_MLPQNA01		Nome esperimento: CL_WIZ_MLPQNA01	
Nome output prodotto	Descrizione	Nome output prodotto	Descrizione
mlpqna_*FULL_TRAIN*_output.txt	Output con relativi target prodotto durante la fase di *TRAIN*	AutoMLPQNA_*FULL_TRAIN*_output.txt	Output con relativi target prodotto durante la fase di *TRAIN*
mlpqna_*FULL.log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.	AutoMLPQNA_*FULL.log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.
Mlpqna_*FULL*_error.txt	File degli errori parziali, 3 colonne: *Step* di *training*, numero di iterazioni, errore	AutoMLPQNA_*FULL*_error.txt	File degli errori parziali, 3 colonne: *Step* di *training*, numero di iterazioni, errore
mlpqna_*FULL_tes*	Matrice di confusione	AutoMLPQNA_*FUL*	Matrice di confusione

*t*ConfMatrix.txt		*L_test*ConfMatrix	
mlpqna_*FULL_TEST*_output.txt	Output con relativi target prodotto durante la fase di *TEST*	AutoMLPQNA_*FULL*_output.txt	Output con relativi target prodotto durante la fase di *TEST*
mlpqna_*FULL_trainTest*OutLog.txt	Valori di output con relativi target	AutoMLPQNA_*FULL_trainTest*OutLog.txt	Valori di output con relativi target
mlpqna_*FULL*_frozen_net.txt	File di configurazione della rete	AutoMLPQNA_*FULL*_frozenNet.txt	File di configurazione della rete
mlpqna_*FULL*_weights.txt	File dei pesi della rete	AutoMLPQNA_*FULL*_weights.txt	File dei pesi della rete
mlpqna_*FULL*_errorPlot.txt	*Plot* della prima e della terza colonna del file mlpqna_*FULL*_error.txt	AutoMLPQNA_*FULL*_errorPlot.txt	*Plot* della prima e della terza colonna del file mlpqna_*FULL*_error.txt
mlpqna_*FULL_Test*OutLog.txt	Output di *TEST* con relativi target	AutoMLPQNA_*FULL_Test*OutLog.txt	Output di *TEST* con relativi target
MLPQNA_*FULL*_params.XML	File di configurazione dell'esperimento	AutoMLPQNA_*FULL*_params.XML	File di configurazione dell'esperimento

Tabella 4 - Classificazione, *FULL* conMLPQNA e AutoMLPQNA – confronto

Le tabelle Tabella 5, Tabella 6, Tabella 7, Tabella 8 mettono a confronto gli output prodotti effettuando una regressione con il modello MLPQNA, rispettivamente per i casi d'uso *TRAIN*, *TEST*, *RUN* e *FULL*, per il *plugin* pre-esistente nella suite (colonne di sinistra) e il *plugin* creato con DAMEWARE *Plugin Wizard*.

REGRESSIONE *TRAIN* con MLPQNA
Parametri di input: Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii Number of input neurons: 4 Number of first hidden layer neurons: 9 Tutti gli altri parametri assumono valori di *default* assegnati dal sistema

Versione Manuale		Con *WIZARD*	
Nome esperimento: RemlpqnaTE01		Nome esperimento: RE_WIZ_MLPQNA01	
Nome output prodotto	Descrizione	Nome output prodotto	Descrizione
mlpqna_*TRAIN*_error.txt	File degli errori parziali, 3 colonne: *Step* di *training*, numero di iterazioni, errore	AutoMLPQNA_*TRAIN*_error.txt	File degli errori parziali, 3 colonne: *Step* di *training*, numero di iterazioni, errore
*dataset_TEST*_20.ascii_mlpqna_*TRAIN*_output.txt	Output file con input *dataset* in append	AutoMLPQNA_*TRAIN*_output.txt	Output file con input *dataset* in append
*dataset_TEST*_20.ascii_mlpqna_*TRAIN_TrainTest*OutLog.txt	Valori di output con telativi target	AutoMLPQNA_*TRAIN_TrainTest*OutLog.txt	Valori di output con telativi target
mlpqna_*TRAIN*_frozen_net.txt	File di configurazione della rete	AutoMLPQNA_*TRAIN*_frozenNet.txt	File di configurazione della rete
mlpqna_*TRAIN*_weights.txt	File dei pesi	AutoMLPQNA_*TRAIN*_weights.txt	File dei pesi
mlpqna_*TRAIN.log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.	AutoMLPQNA_*TRAIN.log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.
Mlpqna_*TRAIN*_errorPlot.jpeg	*Plot* della prima e della terza colonna del file mlpqna_*TRAIN*_error.txt	AutoMLPQNA_*TRAIN*_errorPlot.jpeg	*Plot* della prima e della terza colonna del file AutoMLPQNA_*TRAIN*_error.txt
mlpqna_*TRAIN*_params.xml	File di configurazione dell'esperimento	AutoMLPQNA_*TRAIN*_params.xml	File di configurazione dell'esperimento

Tabella 5 - Regressione, *TRAIN* conMLPQNA e AutoMLPQNA – confronto

REGRESSIONE
TEST con MLPQNA

<table>
<tr><td colspan="4">Input dataset: http://dame.dsf.unina.it/documents/alpha_sample_data/dataset_test_20.ascii
Trained network setup file: ./mlpqna_TRAIN_frozen_net.txt (prodotto dal caso d’uso TRAIN)
Trained network weights file: ./mlpqna_TRAIN_weights.txt (prodotto dal caso d’uso TRAIN)
Number of input neurons: 4
Number of first hidden layer neurons: 9
Tutti gli altri parametri assumono valori di default assegnati dal sistema</td></tr>
<tr><td colspan="2">Versione Manuale</td><td colspan="2">Con WIZARD</td></tr>
<tr><td colspan="2">Nome esperimento: RemlpqnaRU01</td><td colspan="2">Nome esperimento: RE_WIZ_MLPQNA01</td></tr>
<tr><td>Nome output prodotto</td><td>Descrizione</td><td>Nome output prodotto</td><td>Descrizione</td></tr>
<tr><td>mlpqna_TEST_outputPlot.jpeg</td><td>Plot dei valori di output</td><td>AutoMLPQNA_TEST_outputPlot.jpeg</td><td>Plot dei valori di output</td></tr>
<tr><td>dataset_TEST_20.ascii_mlpqna_TEST_output.txt</td><td>Valori di output con input dataset in append</td><td>AutoMLPQNA_TEST_output.txt</td><td>Valori di output con input dataset file</td></tr>
<tr><td>mlpqna_TEST.log</td><td>File di log sullo stato dell’esperimento, contiene data e ora di inizio e fine esperimento.</td><td>AutoMLPQNA_TEST.log</td><td>File di log sullo stato dell’esperimento, contiene data e ora di inizio e fine esperimento.</td></tr>
<tr><td>MLPQNA_TEST_params.xml</td><td>File di configurazione dell’esperimento</td><td>AutoMLPQNA_TEST_params.xml</td><td>File di configurazione dell’esperimento</td></tr>
</table>

Tabella 6 - Regressione, *TEST* conMLPQNA e AutoMLPQNA – confronto

<table>
<tr><td colspan="2">REGRESSIONE
RUN con MLPQNA</td></tr>
<tr><td colspan="2">Input dataset: http://dame.dsf.unina.it/documents/alpha_sample_data/dataset_test_20.ascii
Trained network setup file: ./mlpqna_TRAIN_frozen_net.txt (prodotto dal caso d’uso TRAIN)
Trained network weights file: ./mlpqna_TRAIN_weights.txt (prodotto dal caso d’uso TRAIN)
Number of input neurons: 4
Number of first hidden layer neurons: 9
Tutti gli altri parametri assumono valori di default assegnati dal sistema</td></tr>
<tr><td>Versione Manuale</td><td>Con WIZARD</td></tr>
<tr><td>Nome esperimento: RE_MLPQNA01</td><td>Nome esperimento: RE_WIZ_MLPQNA01</td></tr>
</table>

Nome output prodotto	Descrizione	Nome output prodotto	Descrizione
mlpqna_*RUN*.*log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.	AutoMLPQNA_*RUN*-*log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.
Mlpqna_*RUN*_output.txt	Output file con *dataset* in input	AutoMLPQNA_*RUN*_output.txt	Output file con *dataset* in input
MLPQNA_*RUN*_params.xml	File di configurazione dell'esperimento	AutoMLPQNA_*RUN*_params.xml	File di configurazione dell'esperimento

Tabella 7 - Regressione, *RUN* conMLPQNA e AutoMLPQNA – confronto

REGRESSIONE *FULL* con MLPQNA			
Input *dataset* per *TRAIN*: //dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii Input *dataset* per *TEST*: //dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii Number of input neurons: 4 Number of first hidden layer neurons: 9 Tutti gli altri parametri assumono valori di *default* assegnati dal sistema			
Versione Manuale		Con *WIZARD*	
Nome esperimento: RE_MLPQNA01		Nome esperimento: RE_WIZ_MLPQNA01	
Nome output prodotto	Descrizione	Nome output prodotto	Descrizione
mlpqna_*FULL*_error.txt	File degli errori parziali, 3 colonne: *Step* di *training*, numero di iterazioni, errore	AutoMLPQNA_*FULL*_error.txt	File degli errori parziali, 3 colonne: *Step* di *training*, numero di iterazioni, errore
mlpqna_*FULL_TEST*_outputPlot.jpeg	*Plot* del file di output	AutoMLPQNA_*FULL*_outputPlot.jpeg	*Plot* del del file di output per *TEST*
mlpqna_*FULL_TEST*_output.txt	File contenente il *dataset* in input e gli output della	AutoMLPQNA_*FULL*_output.txt	File contenente il *dataset* in input e gli

	rete		output della rete
mlpqna_*FULL*_frozen_net.txt	File di configurazione della rete	AutoMLPQNA_*FULL*_frozenNet.txt	File di configurazione della rete
mlpqna_*FULL*_weights.txt	File dei pesi della rete	AutoMLPQNA_*FULL*_weights.txt	File dei pesi della rete
mlpqna_*FULL*.*log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.	AutoMLPQNA_*FULL*.*log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.
Mlpqna_*FULL*_errorPlot.jpeg	*Plot* degli errori parziali	AutoMLPQNA_*FULL*_errorPlot.txt	*Plot* degli errori parziali
*dataset*_*TEST*_20.ascii_mlpqna_*FULL*_*Test*OutLog.txt	Valori di output con relative target	AutoMlPQNA_*FULL*_*Test*OutLog.txt	Valori di output con relative target
MLPQNA_*FULL*_params.xml	File di configurazione dell'esperimento	AutoMLPQNA_*FULL*_params.xml	File di configurazione dell'esperimento

Tabella 8 - Regressione, *FULL* conMLPQNA e AutoMLPQNA – confronto

Il modello MLPGA è un tipico modello i cui parametri sono compresi all'interno di due file di configurazione.
Più precisamente, l'unico parametro passato a linea di comando al modello è il nome di un file di testo contenente:

- Nome del file di configurazione della rete neurale al primo rigo;
- Nome del caso d'uso ("*TRAIN*","*TEST*","*RUN*") al secondo rigo;
- Nome del file di configurazione del caso d'uso al terzo rigo;

Un esempio del lancio in un esperimento è quindi:
./mlpga confFile.txt

Il file di configurazione della rete neurale contiene i parametri:

1. File contenente gli input: File contenente il *dataset* in input;
2. File contenente i target (solo per *TRAIN*): File contenente i target;
3. File dei pesi: se non inserito il file conterrà la stringa "*RANDOM*";
4. Stringa "*none*" relativa ad un parametro non più utilizzato (solo per *TRAIN*);
5. Grandezza dell'errore (solo per *TRAIN*): Valore di *default* 0.001;
6. Epoche di *training* (solo per *TRAIN*): Valore di *default* 1000;
7. Frequenza dell'errore (solo per *TRAIN*): Valore di *default* 10;
8. Percentuale di *cross over* (solo per *TRAIN*): Valore di *default* 0.7;
9. Percentuale di mutazione (solo per *TRAIN*): Valore di *default* 0.6;
10. Modalità di perturbazione dei cromosomi (solo per *TRAIN*): Valore di *default* 1;
11. Grandezza della popolazione genetica (solo per *TRAIN*): Valore di *default* 20;
12. Numero di cicli (solo per *TRAIN*): Valore di *default* 2;
13. Elitismo (solo per *TRAIN*): Valore di *default* 3;
14. Nome del file dei pesi da produrre (solo per *TRAIN*): Stringa "weights.txt";
15. Nome del file degli errori da produrre (solo per *TRAIN*): Stringa "error.txt";
16. Nome del file con i valori di output da produrre: Stringa "output.txt";

Il file di configurazione del caso d'uso contiene i parametri:

1. Numero di neuroni in input
2. Numero di livelli intermedi
3. Numero di neuroni al primo livello intermedio
4. Numero di neuroni al secondo livello intermedio
5. Numero di neuroni di output

6. Funzione di attivazione per il primo livello intermedio
7. Funzione di attivazione per il primo secondo intermedio
8. Funzione di attivazione per il livello di output

Le tabelle Tabella 9, Tabella 10, Tabella 11 e Tabella 12 mettono a confronto gli output prodotti effettuando una classificazione con il modello MLPGA, rispettivamente per i casi d'uso *TRAIN*, *TEST*, *RUN* e *FULL*, per il *plugin* pre-esistente nella suite (colonne di sinistra) e il *plugin* creato con DAMEWARE *Plugin Wizard*.

<table>
<tr><td colspan="4">CLASSIFICATIONE
TRAIN con MLPGA</td></tr>
<tr><td colspan="4">Parametri di input:
Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii
Number of input neurons: 4
Number of first hidden layer neurons: 9
Number of output neurons: 1
Tutti gli altri parametri assumono valori di *default* assegnati dal sistema</td></tr>
<tr><td colspan="2">Versione Manuale</td><td colspan="2">Con *WIZARD*</td></tr>
<tr><td colspan="2">Nome esperimento: ClmlpgaTR01</td><td colspan="2">Nome esperimento:</td></tr>
<tr><td>Nome: RemlpgaTE01</td><td>Descrizione</td><td>Nome:</td><td>Descrizione:</td></tr>
<tr><td>mlpga_*TRAIN*_output.csv</td><td>File contenente il *dataset* in input e gli output della rete</td><td>AutoMLPGA_*TRAIN*_output.txt</td><td>File contenente il *dataset* in input e gli output della rete</td></tr>
<tr><td>mlpga_*TRAIN*_error.csv</td><td>File sull'errore di *training* per ogni epoca</td><td>AutoMLPGA_*TRAIN*_error.txt</td><td>File sull'errore di *training* per ogni epoca</td></tr>
<tr><td>mlpga_*TRAIN*_weights</td><td>File dei pesi della rete</td><td>AutoMLPGA_*TRAIN*_weights.txt</td><td>File dei pesi della rete</td></tr>
<tr><td>mlpga_*TRAIN*_worstPlot.jpeg</td><td>*Plot* della prima e nona colonna del file mlpga_*TRAIN*_error.csv</td><td>AutoMLPGA_*TRAIN*_worstPlot.txt</td><td>*Plot* della prima e nona colonna del file AutoMLPGA_*TRAIN*_error.txt</td></tr>
</table>

mlpga_*TRAIN*_bestPlot.jpeg	*Plot* della prima e quinta colonna del file mlpga_*TRAIN*_error.csv	AutoMLPGA_*TRAIN*_bestPlot.txt	*Plot* della prima e quinta colonna del file AutoMLPGA_*TRAIN*_error.txt
mlpga_*TRAIN.log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.	AutoMLPGA_*TRAIN*.*log*	File di *log* sullo stato dell'esperimento, contiene data e ora di inizio e fine esperimento.
MLPGA_*TRAIN*_params.xml	File di configurazione dell'esperimento	AutoMLPGA_*TRAIN*_params.xml	File di configurazione dell'esperimento

Tabella 9 - Classificazione, *TRAIN* con MLPGA – AutoMLPGA – confronto

CLASSIFICATIONE *TEST* con MLPGA			
Parametri di input: Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii Number of input neurons: 4 Number of first hidden layer neurons: 9 Number of output neurons: 1 Tutti gli altri parametri assumono valori di *default* assegnati dal sistema			
Versione Manuale		Con *WIZARD*	
Nome esperimento: ClmlpgaTE01		Nome esperimento:	
Nome:	Descrizione	Nome:	Descrizione:
mlpga_*TEST*_output.csv	File contenente il *dataset* in input e gli output della rete	AutoMLPGA_*TEST*_output.csv	File contenente il *dataset* in input e gli output della rete
mlpga_*TEST*_error.csv	File sull'errore di *training* per ogni epoca	AutoMLPGA_*TEST*_error.txt	File sull'errore di *training* per ogni epoca
mlpga_*TEST*_weigh	File dei pesi della rete	AutoMLPGA_*TES*	File dei pesi della

ts		*T*_weights.txt	rete
mlpga_*TEST*_bestPlot.jpeg	*Plot* della prima e quinta colonna del file mlpga_*TRAIN*_error.csv	AutoMLPGA_*TEST*_bestPlot.txt	*Plot* della prima e quinta colonna del file mlpga_*TRAIN*_error.csv
mlpga_*TEST*_worstPlot.jpeg	*Plot* della prima e nona colonna del file mlpga_*TRAIN*_error.csv	AutoMLPGA_*TEST*_worstPlot.jpeg	*Plot* della prima e nona colonna del file AutoMLPGA_*TEST*_error.txt
mlpga_*TEST.log*	File di *log* contenente data ed ora di inizio e fine esperimento	AutoMLPGA_*TEST.log*	File di *log* contenente data ed ora di inizio e fine esperimento
MLPGA_*TEST*_params.xml	File di configurazione dell'esperimento	AutoMLPGA_*TEST*_params.xml	File di configurazione dell'esperimento

Tabella 10 - Classificazione, *TEST* con MLPGA – AutoMLPGA – confronto

CLASSIFICATIONE *RUN* con MLPGA			
Parametri di input: Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii Number of input neurons: 4 Number of first hidden layer neurons: 9 Number of output neurons: 1 Tutti gli altri parametri assumono valori di *default* assegnati dal sistema			
Versione Manuale		Con *WIZARD*	
Nome esperimento: ClmlpgaRU01		Nome esperimento:	
Nome	Descrizione	Nome:	Descrizione:
mlpga_*RUN*_output.dat	File contenente il	AutoMLPGA_*RUN*_	File contenente il *dataset*

	dataset in input e gli output della rete	output.txt	in input e gli output della rete
mlpga_*RUN*.*log*	File di *log* contenente data ed ora di inizio e fine esperimento	AutoMLPGA_*RUN*.*log*	File di *log* contenente data ed ora di inizio e fine esperimento
MLPGA_*RUN*_params.xml	File di configurazione dell'esperimento	AutoMLPGA_*RUN*_partams.xml	File di configurazione dell'esperimento

Tabella 11 - Classificazione, *RUN* con MLPGA – AutoMLPGA – confronto

CLASSIFICATIONE *FULL* con MLPGA			
Parametri di input: Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii Number of input neurons: 4 Number of first hidden layer neurons: 9 Number of output neurons: 1 Tutti gli altri parametri assumono valori di *default* assegnati dal sistema			
Versione Manuale		Con *WIZARD*	
Nome esperimento: ClmlpgaFU01		Nome esperimento:	
Nome:	Descrizione	Nome:	Descrizione:
mlpga_*FULL*_*train*Output.txt	Output file del *TRAIN*	AutoMLPGA_*FULL*_output.txt	Output file del *TRAIN*
mlpga_*FULL*_error.csv	File sull'errore di *training* per ogni epoca	AutoMLPGA_*FULL*_error.txt	File sull'errore di *training* per ogni epoca
mlpga_*FULL*_weights	File dei pesi della rete	AutoMLPGA_*FULL*_weights.txt	File dei pesi della rete
mlpga_*FULL*_worstPlot.jpeg	*Plot* della prima e nona colonna del file mlpga_*TRAIN*_error.csv	AutoMLPGA_*FULL*_worstPlot.jpeg	*Plot* della prima e nona colonna del file AutoMLPGA_*FULL*_error.txt

mlpga_*FULL*_bestPlot.jpeg	*Plot* della prima e quinta colonna del file mlpga_*TRAIN*_error.csv	AutoMLPGA_*FULL*_bestPlot.txt	*Plot* della prima e quinta colonna del file AutoMLPGA_*FULL*_error.txt
mlpga_*FULL*_output.dat	File contenente il *dataset* in input e gli output della rete	AutoMLPGA_*FULL*_output_*TEST*.txt	File contenente il *dataset* in input e gli output della rete
mlpga_*FULL.log*	File di *log* contenente data ed ora di inizio e fine esperimento	AutoMLPGA_*FULL.log*	File di *log* contenente data ed ora di inizio e fine esperimento
mlpga_*FULL*_ConfusionMatrix.txt	Matrice di confusione	AutomLPgA_*FULL*_ConfusionMatrix.txt	Matrice di confusione
MLPGA_*FULL*_params.txt	File di configurazione dell'esperimento	AutoMLPGA_*FULL*_params.xml	File di configurazione dell'esperimento

Tabella 12 - Classificazione, *FULL* con MLPGA e AutoMLPGA – confronto

Le tabelle Tabella 13, Tabella 14, Tabella 15 e Tabella 16, mettono a confronto gli output prodotti effettuando una regressione con il modello MLPGA, rispettivamente per i casi d'uso *TRAIN*, *TEST*, *RUN* e *FULL*, per il *plugin* pre-esistente nella suite (colonne di sinistra) e il *plugin* creato con DAMEWARE *Plugin Wizard*.

REGRESSIONE *TRAIN* con MLPGA			
Parametri di input: Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii Number of input neurons: 4 Number of first hidden layer neurons: 9 Number of output neurons: 1 Tutti gli altri parametri assumono valori di *default* assegnati dal sistema			
Versione Manuale		Con *WIZARD*	
Nome esperimento: ClmlpgaTR01		Nome esperimento:	
mlpga_*TRAIN.log*	File di *log* contenente data	AutoMLPGA_*TRAIN.l*	File di *log*

	ed ora di inizio e fine esperimento	*og*	contenente data ed ora di inizio e fine esperimento
mlpga_*TRAIN*_output.csv	File contenente il *dataset* in input e gli output della rete	AutoMLPGA_*TRAIN*_output.txt	File contenente il *dataset* in input e gli output della rete
mlpga_*TRAIN*_error.csv	File sull'errore di *training* per ogni epoca	AutoMLPGA_*TRAIN*_error.txt	File sull'errore di *training* per ogni epoca
mlpga_*TRAIN*_bestPlot.jpeg	*Plot* della prima e quinta colonna del file mlpga_*TRAIN*_error.csv	AutoMLPGA_*TRAIN*_bastPlot.jpeg	*Plot* della prima e quinta colonna del file AutoMLPGA_*TRAIN*_error.txt
mlpga_*TRAIN*_worstPlot.jpeg	*Plot* della prima e nona colonna del file mlpga_*TRAIN*_error.csv	AutoMLPGA_*TRAIN*_worstPlot.jpeg	*Plot* della prima e nona colonna del file AutoMLPGA_*TRAIN*_error.txt
mlpga_*TRAIN*_weights	File dei pesi della rete	AutoMLPGA_*TRAIN*_weights.txt	File dei pesi della rete
MLPGA_*TRAIN*_params.xml	File di configurazione dell'esperimento	AutoMLPGA_*TRAIN*_params.xml	File di configurazione dell'esperimento

Tabella 13 - Regressione, *TRAIN* con MLPGA – AutoMLPGA – confronto

REGRESSIONE *TEST* con MLPGA
Parametri di input: Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii Number of input neurons: 4 Number of first hidden layer neurons: 9 Number of output neurons: 1 Tutti gli altri parametri assumono valori di *default* assegnati dal sistema

Versione Manuale		Con *WIZARD*	
Nome esperimento: ClmlpgaTE01		Nome esperimento:	
mlpga_*TEST.log*	File di *log* contenente data ed ora di inizio e fine esperimento	AutoMLPGA_*TEST.log*	File di *log* contenente data ed ora di inizio e fine esperimento
mlpga_*TEST*_output.txt	File contenente il *dataset* in input e gli output della rete	AutoMLPGA_*TEST*_output.txt	File contenente il *dataset* in input e gli output della rete
mlpga_*TEST*_outputPlot.jpeg	*Plot* delle ultime due colonne del file mlpga_*TEST*_output.txt	AutoMLPGA_*TEST*_outputPlot.jpeg	*Plot* delle ultime due colonne del file AutoMLPGA_*TEST*_output.txt
MLPGA_*TEST*_params.xml	File di configurazione dell'esperimento	AutoMLPGA_*TEST*_params.xml	File di configurazione dell'esperimento

Tabella 14 - Regressione, *TEST* con MLPGA – AutoMLPGA – confronto

REGRESSIONE *RUN* con MLPGA			
Parametri di input: Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii Number of input neurons: 4 Number of first hidden layer neurons: 9 Number of output neurons: 1 Tutti gli altri parametri assumono valori di *default* assegnati dal sistema			
Versione Manuale		Con *WIZARD*	
Nome esperimento: RemlpgaRU01		Nome esperimento:	
mlpga_*RUN.log*	File di *log* contenente data ed ora di inizio e fine esperimento	AutoMLPGA_*RUN.log*	File di *log* contenente data ed ora di inizio e fine esperimento
mlpga_*RUN*_output.txt	File contenente il *dataset* in input e gli output della rete	AutoMLPGA_*RUN*_output.txt	File contenente il *dataset* in input e gli output della rete
MLPGA_*RUN*_params.xml	File di configurazione dell'esperimento	AutoMLPGA_*RUN*_params.xml	File di configurazione dell'esperimento

Tabella 15 - Regressione, *RUN* con MLPGA – AutoMLPGA – confronto

<table>
<tr><td colspan="4">REGRESSIONE
FULL con MLPGA</td></tr>
<tr><td colspan="4">Parametri di input:
Input *dataset*: http://dame.dsf.unina.it/documents/alpha_sample_data/*dataset_test*_20.ascii
Number of input neurons: 4
Number of first hidden layer neurons: 9
Number of output neurons: 1
Tutti gli altri parametri assumono valori di *default* assegnati dal sistema</td></tr>
<tr><td colspan="2">Versione Manuale</td><td colspan="2">Con *WIZARD*</td></tr>
<tr><td colspan="2">Nome esperimento: RemlpgaFU01</td><td colspan="2">Nome esperimento:</td></tr>
<tr><td>mlpga_*FULL*.*log*</td><td>File di *log* contenente data ed ora di inizio e fine esperimento</td><td>AutoMLPGA_*FULL*.*log*</td><td>File di *log* contenente data ed ora di inizio e fine esperimento</td></tr>
<tr><td>mlpga_*FULL*_output.txt</td><td>File contenente il *dataset* in input e gli output della rete</td><td>AutoMLPGA_*FULL*_output.txt</td><td>File contenente il *dataset* in input e gli output della rete</td></tr>
<tr><td>mlpga_*FULL*_error.txt</td><td>File sull’errore di *training* per ogni epoca</td><td>AutoMLPGA_*FULL*_error.txt</td><td>File sull’errore di *training* per ogni epoca</td></tr>
<tr><td>mlpga_*FULL*_bestPlot.jpeg</td><td>*Plot* della prima e quinta colonna del file mlpga_*TRAIN*_error.csv</td><td>AutoMLPGA_*FULL*_bestPlot.txt</td><td>*Plot* della prima e quinta colonna del file AutoMLPGA_*FULL*_error.txt</td></tr>
<tr><td>mlpga_*FULL*_worstPlot.jpeg</td><td>*Plot* della prima e nona colonna del file mlpga_*TRAIN*_error.csv</td><td>AutoMLPGA_*FULL*_worstPlot.txt</td><td>*Plot* della prima e nona colonna del file AutoMLPGA_*FULL*_error.txt</td></tr>
<tr><td>mlpga_*FULL*_outputPlot.jpeg</td><td>*Plot* delle ultime due colonne del file mlpga_*FULL*_output.txt</td><td>AutoMLPGA_*FULL*_outputPlot.txt</td><td>*Plot* delle ultime due colonne del file AutoMLPGA_*FULL*_output.txt</td></tr>
<tr><td>mlpga_*FULL*_</td><td>File dei pesi della rete</td><td>AutoMLPGA_*FULL*_</td><td>File dei pesi della rete</td></tr>
</table>

		weights.txt	
MLPGA_*FULL*_params.xml	File di configurazione dell'esperimento	AutoMLPGA_*FULL*_params.xml	File di configurazione dell'esperimento

Tabella 16 - Regressione, *FULL* con MLPGA – AutoMLPGA – confronto

7 Conclusioni e sviluppi futuri

L'infrastruttura DAMEWARE, sin dalle prime fasi della sua progettazione, aveva tra gli obiettivi principali la necessità di dotarsi di un sistema d'integrazione di nuovi modelli di *data mining* e funzionalità di trattamento dati, basato sulla tecnica *plug-and-play*. Ciò al fine di renderla capace di venire incontro alla molteplicità delle esigenze dei potenziali *stakeholders*, permettendone il facile ed intuitivo uso anche da parte di utenti non esperti in tecnologie informatiche.
Tale obiettivo finora era stato parzialmente raggiunto rispettando il requisito minimo di assicurare al contempo l'associabilità e l'indipendenza tra funzionalità di analisi e modelli di *data mining*. Questo requisito era stato rispettato attraverso l'implementazione del *design pattern Bridge*.
Tuttavia il completamento necessario riguardava il superamento dell'intrinseca limitazione di dover modificare manualmente alcune parti di codice interno alle classi di *wrapping* del modello da integrare, nonché l'esigenza di ricompilare l'intera infrastruttura della *web application* per sincronizzare tutti i componenti rispetto alle nuove informazioni iniettate dal modello nuovo.
I due strumenti progettati e realizzati per rendere realmente *pluggabile* l'infrastruttura, oggetto principale del presente lavoro, hanno dunque garantito il raggiungimento dell'obiettivo preposto.
All'interno del lavoro qui presentato, si è anche proceduto in corso d'opera al *wrapping* di un nuovo modello elaborato e implementato dai colleghi del gruppo DAME, il modello di rete neurale MLP addestrato con l'algoritmo del *Quasi-Newton* (QNA). Il *wrapping* è stato effettuato impiegando entrambi i metodi, tradizionale e automatico. In tal modo, durante le fasi del lavoro, si è potuto studiare in dettaglio ogni singolo aspetto della procedura e verificarne direttamente l'impatto.
L'attuale versione delle procedura automatica permette quindi l'integrazione di modelli i cui parametri di input siano passati da linea di comando, oppure tramite file di configurazione esterno. Un ampliamento delle potenzialità di tale procedura prevede la possibilità di gestire i parametri di input come una combinazione delle due modalità supportate. Inoltre, l'attuale procedura, permette di poter definire fino ad un massimo di due file di configurazione. Un successivo sviluppo dell'applicazione consisterà nel dare la possibilità di ampliare la varietà e la quantità di tali file.
Un altro previsto sviluppo dell'applicazione permetterà la combinazione di due o più algoritmi di *Machine Learning* all'interno di un unico *plugin* (*workflow*).
Per quanto riguarda la funzione di generazione automatica del codice, un successivo sviluppo permetterà un miglioramento delle funzioni di generazione ed un'ottimizzazione del codice generato, convertendo i blocchi di codice in metodi delle classi.

Bibliografia

[1] G. C. Cay S. Horstmann, Core Java 2, Volume I - Fondamenti.

[2] M. Brescia, G. Longo e F. Pasian, Mining Knowledge in Astrophysical Massive Data Sets, 2010.

[3] M. Brescia, S. Cavuoti, A. Nocella, M. Garofalo e S. Riccardi, «designSummary_DAMEWARE-SDD-NA-0018-Rel1.0,» Napoli, 2010.

[4] M. Brescia, «http://dame.dsf.unina.it/project.html,» 2010. [Online].

[5] T. Hey, S. Tansley e K. Tolle, The Fourth Paradigm: Data-Intensive Scientific Discovery, Redmond, Washington: Microsoft Research, 2009.

[6] M. Brescia e S. Riccardi, « MLPQNA_DAME-SRS-NA-0009-Rel_1.0,» 2011.

[7] M. Brescia, « mlpGP_DAME-MAN-NA-0008-Rel1.,» 2010.

Ringraziamenti

Al Dottor Massimo Brescia, per la cortesia, la disponibilità e per la pazienza.
Al Professor Giuseppe Longo e a tutti i ragazzi del gruppo DAME per tutto l'aiuto e la disponibilità.
Al Professor Massimo Benerecetti, per la disponibilità.

Ai miei genitori per avermi permesso di raggiungere questo traguardo.
A Grazia, per essersi presa sempre cura di me.
A Mario, per avermi sempre portato con se.

Ad Anya, per tutto l'aiuto che mi ha dato in questi anni.

A Mara, per avermi sempre incoraggiato, sostenuto e per essermi sempre stata vicina, soprattutto nei momenti difficili.