Università degli Studi di Salerno

Dipartimento di Informatica

Tesi di Laurea Triennale in

Informatica

UN ALGORITMO PER L'INFERENZA DI DIPENDENZE FUNZIONALI RILASSATE: IDENTIFICAZIONE DI SOGLIE OTTIME

Relatore

Chiar.mo Prof. Vincenzo Deufemia

Secondo Relatore

Dott.ssa Loredana Caruccio

Candidato

Giovanni Leo

Matr. 0512103062

Anno Accademico 2016-2017

Alla mia famiglia, per aver creduto in me e per avermi sostenuto.

A Carmela, per avermi supportato e sopportato. MODIFICARE

Abstract

Nella progettazione di una base di dati ci sono aspetti essenziali da prendere in considerazione per assicurare un servizio quanto più efficiente possibile. Considerato il netto aumento del flusso di dati degli ultimi anni, la *data quality* è divenuta una materia estremamente interessante vista la cospicua presenza di dati "sporchi" nelle basi di dati. Per ridurre anomalie ed inconsistenze ci vengono incontro le *Dipendenze funzionali*, utilizzate ampiamente per stabilire vincoli di integrità tra i dati. La grande mole di dati, però, ha reso necessario un riadattamento delle dipendenze funzionali rendendole in grado di catturare inconsistenze più ampie nei dati. Le *Dipendenze funzionali rilassate o approssimate* **(RFD)** sono da considerarsi come una naturale evoluzione o generalizzazione delle *dipendenze funzionali canoniche*. Il concetto più importante introdotto dalle RFD è quello della *similarità*. Nelle dipendenze funzionali classiche esisteva soltanto il concetto di uguaglianza tra dati, nelle RFD espandiamo questo concetto ad una similarità, questo ci permetterà di coprire una quantità di dati maggiore. Tuttavia le RFD possono fornire vantaggi solo se possono essere scoperte automaticamente dai dati. Il lavoro di tesi si è basato su questo ultimo concetto di ottenere le RFD in seguito

ad una procedura automatizzata. Durante le varie fasi di studio si è pensato ed implementato un algoritmo che permette, attraverso tre fasi intermedie, la scoperta di RFD di un dataset dato come input. Per questo lavoro di tesi mostreremo l'idea dell'algoritmo generale ed entreremo nel dettaglio dell' ultima fase di sviluppo(RFD Discovery), mostrando, infine, i risultati della sperimentazione.

Indice

Elenco delle tabelle

Snippet di codice

Introduzione

1.1 Incipit

Nella progettazione di una base di dati ci sono aspetti essenziali da prendere in considerazione per assicurare un servizio quanto più efficiente possibile. Uno di questi servizi è certamente la *qualità dei dati*, una base di dati con questa caratteristica farà si che le inconsistenze tra i dati siano il minor numero possibile. Negli ultimi anni la crescita delle reti ha portato ad un aumento considerevole del flusso di dati rendendo la *data quality* una materia estremamente interessante vista la cospicua presenza di dati "sporchi" proveniente da fonti differenti. Per ridurre questo tipo di anomalie è impensabile tentare di eliminare le *inconsistenze* manualmente, una procedura di questo tipo può essere facilmente incline ad errori sopratutto con la quantità di dati precedentemente citata. In questo lavoro ci vengono incontro le *Dipendenze funzionali*, utilizzate ampiamente per stabilire vincoli di integrità tra i dati e ridurre anomalie e inconsistenze all'interno della nostra base di dati. La grande mole

di dati, però, ha reso necessario un riadattamento delle dipendenze funzionali rendendole in grado di catturare inconsistenze più ampie nei dati. Le *Dipendenze funzionali rilassate o approssimate* **(RFD)** sono da considerarsi come una naturale evoluzione o generalizzazione delle *dipendenze funzionali canoniche*. Questo nuovo strumento ci permette di adattare le semplici dipendenze funzionali a diversi contesti applicativi, infatti, le RFD possono applicarsi anche solo ad una porzione di database. Il concetto più importante introdotto dalle RFD è quello della *similarità*. Nelle dipendenze funzionali classiche esisteva soltanto il concetto di uguaglianza tra dati, nelle RFD espandiamo questo concetto ad una similarità, questo ci permetterà di coprire una quantità di dati maggiore. Tuttavia le RFD possono fornire vantaggi solo se possono essere scoperte automaticamente dai dati. Il lavoro di tesi si è basato su questo ultimo concetto di ottenere le RFD in seguito ad una procedura automatizzata. Durante le varie fasi di studio si è pensato ed implementato un algoritmo che permette, attraverso tre fasi intermedie, la scoperta di RFD di un dataset dato come input. Le tre fasi di questo algoritmo sono: *Feasibility,Minimality,RFD Discovery* . Per questo lavoro di tesi mostreremo l'idea dell'algoritmo generale ed entreremo nel dettaglio dell'ultima fase di sviluppo(RFD Discovery), mostrando, infine, i risultati della sperimentazione. Per questo algoritmo, particolare attenzione è stata posta sull'efficienza, oltre che sull'efficacia, studiando un'implementazione basata sul multithreading e predisponendola ad eventuale adattamento parallelo.

1.2 Nozioni Preliminari

Prima di esporre le RFD è necessario introdurre alcuni concetti preliminari.

1.2.1 Schema di relazione

Uno schema di relazione è costituito da un simbolo R, detto nome della relazione, e da un insieme di attributi $X = \{A_1, A_2, ..., A_n\}$, di solito indicato con $R(X)$. A ciascun attributo $A \in X$ e associato un dominio $dom(A)$. Uno schema di base di dati è un insieme di schemi di relazione con nomi diversi:

$$R = \{R_1(X_1), R_2(X_2), \ldots, R_n(X_n)\}.$$

Una relazione su uno schema $R(X)$ è un insieme r di tuple su X. Per ogni istanza $r \in R(X)$, per ogni tupla $t \in r$ e per ogni attributo $A \in X$, $t[A]$ rappresenta la proiezione di A su t. In modo analogo, dato un insieme di attributi $Y \subseteq X$, $t[Y]$ rappresenta la proiezione di Y su t.[1]

Matricola	Cognome	Nome	Data di nascita
123456	Rossi	Maria	25/11/1991
654321	Neri	Anna	23/04/1992
456321	Verdi	Fabio	12/02/1992

Tabella 1.1: Esempio di schema di relazione

1.2.2 Dipendenze funzionali canoniche

Una *dipendenza funzionale*, abbreviata in FD, è un vincolo di integrità semantico per il modello relazionale che descrive i legami di tipo funzionale tra gli attributi di una relazione.

Data una relazione r su uno schema $R(X)$ e due sottoinsiemi di attributi non vuoti Y e Z di X, diremo che esiste su r una dipendenza funzionale tra Y e Z, se, per ogni coppia di tuple t_1 e t_2 di r aventi gli stessi valori sugli attributi Y, risulta che t_1 e t_2 hanno gli stessi valori sugli attributi Z:

$$\forall t_1, t_2 \in r, t_1[Y] = t_2[Y] \implies t_1[Z] = t_2[Z] \tag{1.1}$$

Una dipendenza funzionale tra gli attributi Y e Z viene indicata con la notazione $Y \to Z$ e viene associata ad uno schema.

Se l'insieme Z è composto da attributi $A_1, A_2, \ldots, A_k$, allora una relazione soddisfa $Y \to Z$ se e solo se essa soddisfa tutte le k dipendenze $Y \to A_1$, $Y \to A_2$,..., $Y \to A_k$. Di conseguenza, quando opportuno, possiamo assumere che le dipendenze abbiano la forma $Y \to A$, con A singolo attributo.

Una relazione funzionale è *non banale* se A non compare tra gli attributi di Y.

Data una chiave K di una relazione r, si può facilmente notare che esiste una dipendenza funzionale tra K ed ogni altro attributo dello schema di r. Quindi una dipendenza funzionale $Y \to Z$ su uno schema $R(X)$ degenera nel vincolo di chiave se l'unione di Y e Z è pari a X. In tal caso Y è superchiave per lo schema $R(X)$.

Con la notazione $\langle R(X), F \rangle$ indicheremo uno schema $R(X)$ su cui è definito un insieme di dipendenze funzionali F. Un'istanza r di $R(X)$ viene detta *istanza legale* di $\langle R(X), F \rangle$ se soddisfa tutte le dipendenze funzionali in F. Infine, data una relazione funzionale $Y \to Z$, se ogni istanza legale r di $\langle R(X), F \rangle$ soddisfa anche $Y \to Z$, allora diremo che F *implica logicamente* $Y \to Z$,

indicato come $F \models Y \rightarrow Z$.

1.2.3 Dipendenze funzionali rilassate

In alcuni casi per risolvere dei problemi in alcuni di domini di applicazioni, come l'identificazione di inconsistenze tra i dati, o la rilevazione di relazioni semantiche fra i dati, è necessario rilassare la definizione di dipendenza funzionale, introducendo delle approssimazioni nel confronto dei dati. Invece di effettuare dei controlli di uguaglianza, si utilizzano dei controlli di similarità. Inoltre spesso si potrebbe desiderare che una certa dipendenza valga solo su un sottoinsieme di tuple che su tutte. Per questo motivo sono nate delle dipendenze funzionali che rilassano alcuni dei vincoli delle FD, prendono il nome di Dipendenze Funzionali Rilassate o Approssimate [1]. Esistono differenti tipi di RFD, ciascuna di esse rilassa uno o più vincoli delle FD, si possono dividere in due macro aree:

1. Confronto di attributi: La funzione di uguaglianza delle FD canoniche viene sostituita da una funzione di similarità , ciò implica che l'AFD deve descrivere una soglia di rilassamento per ogni attributo.

2. Estensione: Permette che il vincolo non sia valido su tutte le tuple, ma solo su di un sottoinsieme di esse.

Le RFD sono utilizzate in attività di: data cleaning, record matching e di rilassamento delle query. Le definizione formale di una RFD è la seguente:

[1] RFD abbreviazione di Relaxed Functional Dependency.

Teorema 1 *Sia R uno schema relazionale definito su di un insieme di attributi finito, e sia $R = (A_1, A_2, ..., A_k)$ uno schema relazionale definito su R. Una RFD φ su R viene rappresentata come:*

$$D_c : (X)_{\Phi_1} \xrightarrow{\Psi(X,Y) \leq \epsilon} (Y)_{\Phi_2}$$

dove:

- $\mathbb{D}_c = \{(t) \in dom(R) | (\bigwedge_{i=1}^{k} c_i(t[A_i]))$, *dove* $c = (c_1, ..., c_k)$ *con* c_i *è un predicato sul* $dom(A_i)$, *utilizzato per filtrare le tuple a cui* φ *va applicata;*
- $X, Y \subseteq attr(R)$ *tali che* $X \cap Y = 0$*;*
- Φ_1*(*Φ_2 *rispettivamente) è un insieme di vincoli* $\phi[X](\phi[Y])$ *definito sull'attributo* X *e* (Y *rispettivamente). Per qualsiasi coppia di tuple* $(t_1, t_2) \in \mathbb{D}_c$ *il vincolo* $\phi[X](\phi[Y]$ *rispettivamente) restituisce vero se la similarità fra* t_1 *e* t_2 *sugli attributi* X *e* (Y *rispettivamente) concordano con i vincoli specificati da* $\phi[X](\phi[Y]$ *rispettivamente);*
- Ψ : *rappresenta una misura di copertura su* $\mathbb{D}_c$ *e indica il numero di tuple che violano o soddisfano* φ*;*
- ϵ *è la soglia che indica il limite superiore o inferiore per il risultato della misura di copertura;*

Nel lavoro di tesi vengono trattate solo le RFD che rilassano il vincolo di uguaglianza. Data RFD $X \rightarrow Y$ essa vale su una relazione r se e solo se la distanza fra due tuple t_1 e t_2, i cui valori sui singoli attributi A_i non superano una certa soglia β_i, è inferiore ad una certa soglia a_A su ogni attributo $A \in X$, allora la distanza fra t_1 e t_2 su ogni attributo $B \in Y$ è minore di una certa

soglia a_B.

La struttura delle RFD utilizzate è la seguente:

$$attr_1(\leq soglia_1), \ldots, attr_n(\leq soglia_n) \rightarrow RHS$$

Gli attributi che si trovano a sinistra della freccia costituiscono la parte LHS[2], l'attributo che invece si trova dopo la freccia costituisce l'RHS[3]. È importante focalizzare l'attenzione su questo concetto in quanto le dipendenze funzionali hanno un verso, ed è quello indicato dalla freccia. Qualsiasi operazione effettuata con le RFD deve sempre tener conto del verso, le RFD non forniscono conoscenza nel verso opposto. Questa non è una proprietà riguardante solo le RFD, bensì riguarda qualsiasi tipo di dipendenza funzionale. Ad esempio consideriamo la relazione in questa tabella:

Impiegato	Stipendio	Progetto	Bilancio	Funzione
Rossi	20000	Sito web	2000	tecnico
Verdi	35000	App Mobile	15000	progettista
Verdi	35000	Server	15000	progettista
Neri	55000	Server	15000	direttore
Neri	55000	App Mobile	15000	consulente
Neri	55000	Sito web	2000	consulente
Mori	48000	Sito web	15000	direttore
Mori	48000	Server	15000	progettista
Bianchi	48000	Server	15000	progettista
Bianchi	48000	App Mobile	15000	direttore

Tabella 1.2: Esempio di Relazione con anomalie

Si può osservare che lo stipendio di ciascun impiegato è unico, quindi in ogni tupla in cui compare lo stesso impiegato verrà riportato lo stesso stipendio. Possiamo dire che esiste una Dipendenza Funzionale: *Impiegato* $\rightarrow$

[2]Left Hand Side o lato sinistro.
[3]Right Hand Side o lato destro.

Stipendio. Si può fare lo stesso discorso tra gli attributi Progetto e Bilancio, quindi anche qui abbiamo una dipendenza funzionale $Progetto \rightarrow Bilancio$. Non si può dire che di conseguenza vale anche il verso opposto:

$$Impiegato \rightarrow Stipendio \neq Stipendio \rightarrow Impiegato$$

Infatti percepiscono 48000 di stipendio sia Mori che Bianchi.[1]

1.2.4 Scoperta di RFD

Data una relazione r, la scoperta di una RFD è il problema di trovare un *minimal cover set* di RFD che si verificano per r. Questo problema rende ancor più complesso il problema della scoperta delle dipendenze dei dati visto l'ampio spazio di ricerca dei possibili vincoli di similarità. Dunque è necessario trovare algoritmi efficienti in grado di estrarre RFD con vincoli di similarità significativi.

Se i vincoli di similarità e le soglie sono noti per ogni attributo del dataset, scoprire le RFD si riduce a trovare tutte le possibili dipendenze che soddisfano la seguente regola:

Lemma 1 *Le partizioni di tuple che sono simili sugli attributi contenuti nel lato sinistro o LHS della dipendenza, devono corrispondere a quelle che sono simili nel lato destro o RHS.*

Questo problema è simile a trovare le FD, dove bisogna trovare le partizioni di tuple che condividono lo stesso valore sull'RHS quando esse condividono lo stesso valore sull'LHS. Il problema viene reso più semplice dal fatto che, nel

caso della scoperta delle FD, tali partizioni sono disgiunte, cosa che però non vale nelle RFD in quanto uno stesso valore può essere simile a valori differenti. Ciò impedisce quindi di sfruttare gli algoritmi utilizzati nella scoperta delle FD, nella scoperta delle RFD

1.2.5 Dominanza

Nel corso del nostro lavoro, abbiamo applicato alcuni risultati dell'intelligenza artificiale al campo della discovery delle *Dipendenze Funzionali Rilassate*. In particolare, cercando di individuare le dipendenze funzionali rilassate ci siamo serviti di un importante risultato nella succitata materia: la dominanza stretta *(strict dominance)*.

Teorema 2 *Dato un vettore di attributi* $\mathbf{X} = X_1, X_2, \ldots, X_n$*, siano* $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ *e* $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ *due vettori di assegnamenti definiti sugli attributi di* $\mathbf{X}$*, dove l'i-esimo elemento* x_i *o* y_i *può essere sia un valore numerico sia un valore discreto con un assunto ordinamento su tali valori. Diremo che* $\mathbf{x}$ *domina strettamente (o deterministicamente)* $\mathbf{y}$ *se e solo se*

$$y_i \leq x_i \quad i = 1, 2, \ldots, n,$$

ovvero

$$\mathbf{y} - \mathbf{x} \leq \mathbf{0}$$

1.3 Studi preliminari

Prima di iniziare lo sviluppo dell'algoritmo per la scoperta di RFD si è reso necessario uno studio approfondito di un algoritmo precedentemente sviluppato per un progetto di IA [2]. Tale algoritmo è stato sviluppato in Python, pertanto, abbiamo effettuato uno studio del linguaggio precedentemente citato. Oltre le principali caratteristiche di questo linguaggio, è stato fatto uno studio anche delle librerie utilizzate all'interno del progetto:

- ***Pandas***: È una libreria che include delle strutture dati e tool di analisi facili da usare e fortemente ottimizzate.
- ***Numpy***: È un package dedicato all'elaborazione scientifica sul linguaggio Python.

Una volta concluso questo tipo di studio si è cominciato a pensare allo sviluppo dell'algoritmo in un ambiente differente. La scelta è ricaduta su *Java*, tale scelta è dovuta, oltre che alla già piena conoscenza del team di questo linguaggio, alla potenza e versatilità che questo linguaggio ci offre, oltre che al gran numero di framework presenti per la gestione di parallelizzazione e concorrenza, essendo quest'ultimo un aspetto molto importante per l'efficienza dell'algoritmo. Le librerie esterne studiate ed utilizzate saranno ben approfondite nel capitolo 4(*Implementazione*) di questo elaborato. Le sopracitate librerie esterne utilizzate sono:

- ***AKKA***: Framework per la gestione del parallelismo e concorrenza.

- ***Joinery Dataframe***: Struttura dati simile al dataframe presente in *Pandas* di Python.

All'infuori delle conoscenze legate ai linguaggi di programmazione, è stato necessario leggere e studiare vari documenti legati al mondo delle dipendenze funzionali.

Stato dell'Arte

Esistono svariati metodi per scoprire le RFD data una determinata soglia ϵ, un esempio è il metodo *top-down*.

I metodi di discovery *top-down* effettuano una generazione di possibili FD livello per livello e controllano se queste si verificano. L'algoritmo inizia generando un grafo di attributi, con una struttura a lattice, dove vengono considerati tutti i possibili sottoinsiemi di attributi. Dato uno schema relazionale $R = (A_1, A_2, \cdots, A_n)$, il livello 0 del lattice non contiene nessun attributo, il livello 1 contiene tutti i singleton dei singoli attributi dello schema relazionale R, il livello due tutte le possibili coppie di attributi in R fino ad arrivare all'ultimo livello, l'n-esimo, che contiene un unico insieme con tutti gli attributi di R al suo interno. Ogni sottoinsieme contenuto nel lattice rappresenta un candidato per una possibile FD.

Generato il lattice, l'algoritmo parte dal livello 0 fino ad arrivare all'ultimo, e per ogni livello verifica, per tutti i possibili sottoinsiemi $X \in L_r$[1], l'esistenza

[1] livello r-esimo

di possibili dipendenze funzionali. Nello specifico, per ogni attributo $A \in X$ si cerca di verificare se la FD $X \setminus \{A\} \rightarrow A$ vale. Per ridurre il tempo di esecuzione esponenziale, assieme alla verifica avviene una potatura del grafo sfruttando la scoperta di nuove FD.

Inoltre negli ultimi anni c'è stata una proliferazione delle RFDs di cui solo alcune di loro erano dotate dell'algoritmo per la scoperta dai dati. Mostriamo adesso alcune di esse.

2.1 AFD Discovery

Una *dipendenza funzionale approssimata*(AFD) è una canonica FD che deve essere soddisfatta da 'più' tuple, piuttosto che 'tutte', di una relazione r. In altre parole, una AFD permette a una piccolissima porzione di tuple di r di violarla. Diversi approcci sono stati proposti per calcolare il grado di soddisfacibilità di una AFD. Gli approcci principali sono basati su una piccola porzione di tuple $s \subset r$ per decidere se una AFD esiste su r. Come conseguenza, le AFDs che esistono su s possono anche esistere su r, con una data probabilità. Alcuni metodi sfruttano la misurazione dell'errore della super chiave per determinare la soddisfacibilità approssima delle AFDs.

2.2 MD Discovery

Matching dependencies(MDs) sono delle RFD proposte recentemente per l'object identification. Sono definite in termini di predicati di similarità per adeguarsi agli errori e a differenti rappresentazioni di dati in sorgenti inaffida-

bili. Infatti è stato proposto un algoritmo che ha a che fare con la valutazione dell' utilità delle MDs in una data istanza di un database e la determinazione del pattern di similitudine delle MDs. L'utilità è misurata considerando la convenienza e il sostegno delle MDs, mentre le soglie sono determinate in base alla distribuzione statistica dei dati. Inoltre sono state introdotte delle strategie di Pruning per filtrare i pattern con un basso sostegno.

2.3 DD Discovery

Differential dependencies(DDs) sono delle RFD che specificano vincoli sulla differenza dei valori degli attributi invece delle corrispondenze esatte delle FD canoniche. Il discovery delle DDs eredita la complessità esponenziale dal problema del discovery delle FD.

Un algoritmo per il discovery delle DDs si basa sugli algoritmi di riduzione, il quale una volta fissate le funzioni di differenza per l' RHS per ogni attributo della relazione r, l'insieme delle funzione di differenza per gli LHS ridotti viene cercato per formare le DDs. Le strategie di pruning sono state proposte per migliorare le performance del discovery.

Un algoritmo alternativo riduce lo spazio di ricerca per mezzo di limiti superiori alle soglie di distanza per gli intervalli di LHS specificati dall' utente.

Un ulteriore proposta per il DD discovery è un algoritmo che estrae un minimal cover di DDs, basato su regole di associazione. In particolare l'algoritmo estrae una classe di regole di associazione non ridondanti le quali verranno trasformate in DDs.

Infine è stato proposto un algoritmo per ottenere delle soglie adatte per una

data DD. In particolare data una istanza di un database e una DD su di esso, l'algoritmo determina le soglie di distanza per la DD al fine di massimizzare la sua utilità.

Algoritmo

In questo capitolo saranno mostrati i passi da effettuare per ottenere, partendo da un dataset rappresentante una relazione, una lista di dipendenze funzionali rilassate. La sequenza di passi che l'algoritmo affronterà sono:

- ***Feasibility***
- ***Minimality***
- ***RFD Generation***

Per fare in modo che la prima fase(*Feasibility*) abbia inizio, ci dobbiamo creare la matrice delle distanze, che, insieme ad alcune informazioni aggiuntive verranno date in input alla suddetta fase. In seguito di passerà alla fase di *Minimality* la quale fornirà in output, l'input dell' ultima fase ovvero, *RFD Generation*. In questo capitolo verrà descritta nello specifico la fase di *RFD Generation* che è oggetto di questo lavoro di tesi.

3.1 Matrice delle distanze

Il primo passo consiste nel calcolo delle distanze tra ogni coppia di tuple del dataset. Questo passaggio viene fatto utilizzando diverse funzioni di distanza, a seconda del tipo di RFD che si vuole ricavare.

3.1.1 Funzione di distanza

Date due tuple t_i e t_j tali che $t_i, t_j \in \mathbf{r}$, risulta necessario definire un metodo per capire quanto queste siano distanti tra di loro. Definiamo quindi una funzione

$$\mathbf{d} : \mathbf{r} \to \mathbb{R}^{k+1}$$

tale che, date due tuple $t_i, t_j \in \mathbf{r}$,

$$\mathbf{d}(t_i, t_j) = [(d_1, d_2, ..., d_{k+1}) \mid d_p = d(t_i[p], t_j[p])],$$

dove $t_i[p]$ rappresenta il p-esimo elemento della tupla t_i e $d(x, y)$ è una funzione di distanza tra gli elementi x e y che cambia in base al tipo di elemento.
Una funzione $\mathbf{d}(t_i, t_j)$ può quindi essere ibrida, ossia composta da diverse distanze su attributi di tipi diversi: per confrontare tuple composte da valori misti (e.g. numerici, stringhe e date) è necessario utilizzare diverse distanze. Quindi, una generica funzione distanza $\mathbf{d}$ operante su due tuple, è composta al suo interno da altre funzioni di distanza d_t, anche diverse tra di loro, operanti su singoli attributi.
Durante la progettazione del nostro algoritmo abbiamo considerato le seguenti

funzioni distanza, in base ai diversi tipi di elemento che potevano far parte di una tupla:

Distanza euclidea mono-dimensionale tale funzione viene utilizzata quando x e y sono entrambi tipi numerici:

$$d_{eucl}(x, y) = \sqrt{(x - y)^2} = |x - y|. \tag{3.1}$$

Distanza di Levenshtein è una metrica per misurare la differenza fra due stringhe (o sequenze in generale). Tale distanza indica il minimo numero di caratteri singoli che è necessario modificare (e.g. inserimenti, cancellazioni, sostituzioni) per far sì che le due stringhe siano uguali. È quindi definita come:

$$d_{lev}(x, y) = lev_{x,y}(|x|, |y|) \tag{3.2}$$

dove

$$lev_{a,b}(i, j) = \begin{cases} \max(i, j) & \text{se } min(i, j) = 0 \\ \min \begin{cases} lev_{a,b}(i - 1, j) + 1 \\ lev_{a,b}(i, j - 1) + 1 \\ lev_{a,b}(i - 1, j - 1) + 1_{(a_i \neq b_j)} \end{cases} & \text{altrimenti} \end{cases}$$

dove $1_{(a_i \neq b_j)}$ è una *funzione indicatrice*, uguale a 0 se $a_i = b_j$ e uguale a 1 altrimenti, e $lev_{a,b}(i, j)$ è la distanza di Levenshtein tra i primi i caratteri di a e i primi j caratteri di b;

Distanza tra date quando gli attributi da confrontare sono di tipo data (a prescindere dal formato di questa), viene utilzzata una semplice funzione di distanza che permette di calcolare i secondi trascorsi dalla data meno recente a quella più recente.

3.1.2 Calcolo della matrice delle distanze

Dal confronto tra tutte le coppie di tuple della relazione ricaviamo una tabella delle distanze DT tale che $|DT| = \binom{n}{2} = \frac{n(n-1)}{2}$.

La tabella delle distanze (supponendo senza perdere di generalità di utilizzare l'attributo X_{k+1} come attributo RHS) ricavata è quindi:

	RHS	X_1	X_2	$\dots$	X_k
1	d_1^{RHS}	d_{11}	d_{12}	$\dots$	d_{1k}
2	d_2^{RHS}	d_{21}	d_{22}	$\dots$	d_{2k}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
m	d_m^{RHS}	d_{m1}	d_{m2}	$\dots$	d_{mk}

dove $m = \frac{n(n-1)}{2}$ e ogni riga rappresenta il vettore distanza ottenuto confrontando due diverse tuple facenti parte della relazione, ad esempio, date due tuple $t_r = (x_1, x_2, \dots, x_{k+1})$ e $t_l = (y_1, y_2, \dots, y_{k+1})$ tali che $t_r \in \mathbf{r}$ e $t_l \in \mathbf{r}$, calcoliamo la distanza tra queste due tuple e poniamo il vettore

$$d(t_r, t_l) = [d(x_1, y_1), d(x_2, y_2), \dots, d(x_{k+1}, y_{k+1})]$$

in $DT[i]$. Il generico d_{ij} rappresenta la distanza sull'attributo X_i calcolato su una coppia di tuple.

La colonna

$$DT[\cdot]['RHS'] = d_1^{RHS} d_2^{RHS} : d_m^{RHS}$$

rappresenta la colonna delle distanze tra tutte le n tuple rispetto all'attributo RHS.

Dopo aver calcolato la tabella DT, occorre ordinarla utilizzando come pivot la colonna $DT[\cdot]['RHS']$, ottenendo il seguente data frame:

	RHS	X_1	X_2	$\dots$	X_k
1	d_i^{RHS}	d_{i1}	d_{i2}	$\dots$	d_{ik}
2	d_j^{RHS}	d_{j1}	d_{j2}	$\dots$	d_{jk}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
m	d_r^{RHS}	d_{r1}	d_{r2}	$\dots$	d_{rk}

dove $d_i^{RHS} \leq d_j^{RHS} \leq \ldots \leq d_r^{RHS}$.

A questo punto occorre dividere e raggruppare tutte le righe del data frame: le righe aventi lo stesso valore($ClusterID$) sulla colonna $DT[\cdot]['RHS']$ saranno raggruppate in sezioni di data frame disgiunte. Questo data frame finale può finalmente essere utilizzato dalle successive fasi dell'algoritmo.

3.2 Feasibility e Minimality

La fase di *Feasibility* sottopone i pattern ad un test e restituisce l'insieme dei pattern che li hanno superati. Ce ne sono tanti quanti sono gli RHS. Invece la fase si *Minimality* restituisce in output un certo numero di sottopattern minimi.

3.3 RFD Generation

3.3.1 Nozioni Preliminari

Dalla fase di *Minimality* si otterranno un certo numero di sottopattern minimi(ovvero minimali[1] e ammissibili[2]) i quali possono essere denominati S_k. Essendo che ogni RHS contiene un certo numero di C_i[3] (con $i = 1, \ldots, m$) dove ogni C_i contiene un certo numero di pattern P_j (con $j = 1, \ldots, h$) e per ognuno di questi pattern si otterrà un S_k

3.3.2 Identificazione di soglie ottime

L'idea generale è quella di trovare i più grandi pattern di thresholds tali che questi non dominano i pattern che non dominano. Quindi dato un S_k ammissibile dobbiamo generare le soglie ottime.

L'idea di base è una volta che ho trovato un minimo per un cluster, devo generare le RFD per il cluster successivo, poiché una volta raggiunto il minimo viene trovata una violazione alla dipendenza candidata questo implica che la soglia valida per il cluster successivo non deve raggiunge il minimo. La regola generale è che ogni volta che trova una coppia che è simile sull' LHS(in questa sezione lo chiameremo X) questa deve essere simile secondo quella soglia sull'RHS. Possiamo distinguere 3 casi.

- **Caso base:** Consideriamo un unico attributo candidato per $X(|X| =$

[1]Non è possibile eliminare nessun attributo da tale pattern poiché eliminandolo la distanza di un attributo diventa dominante.

[2]Tali pattern non dominano altri pattern.

[3]Gli insiemi C_i sono l'output della fase di *Feasibility*.

1).

Dato un *clusterID* $k > 0$ dell'attributo A, se esiste trova il minimo valore m tale che

$$m = min(k) < min(prev(k)^4) \tag{3.3}$$

$(m - \epsilon)$ rappresenta la migliore soglia per il cluster $next(k)^5$, in questo caso $\epsilon = 1$. Se $m > 0$ (non avrebbe senso imporre che la soglia sia minore o uguale ad un numero negativo) allora genero

$$X_{(\leq m-\epsilon)} \rightarrow A_{(\leq next(k))} \tag{3.4}$$

nel caso in cui $m = min(prev(k))$ devo aggiornare

$$X_{(\leq m-\epsilon)} \rightarrow A_{(\leq k)} \text{ in } X_{(\leq m-\epsilon)} \rightarrow A_{(\leq next(k))}$$

in altre parole quando ho due o più RFD sullo stesso LHS devo scegliere quella con RHS minore.

- **Caso con $|X| = 2$:** Dato un *clusterID* $k > 0$ dell'attributo A, se esiste trova le coppie (m_1, m_2) non dominanti rispetto all'insieme di tuple $S = nonDominating(k)$, quindi $(m_1 - \epsilon, \alpha_2)$ rappresenta una delle migliori soglie per il cluster $next(k)$.

  Se $m_1 > 0$ genera

$$(X_1)_{(\leq m_1-\epsilon)}(X_2)_{(\leq \alpha_2)} \rightarrow A_{(\leq next(k))} \tag{3.5}$$

[4]Indica il *clusterID* precedente
[5]Indica il *clusterID* successivo

invece, $(\alpha_1, m_2 - \epsilon)$ rappresenta una delle migliori soglie per il cluster $next(k)$.

Se $m_2 > 0$ genera

$$(X_1)_{(\leq \alpha_2)}(X_2)_{(\leq m_2 - \epsilon)} \rightarrow A_{(\leq next(k))} \tag{3.6}$$

il tutto è possibile se α_j(in questo caso $j = 1, 2$) esiste. A questo punto devo determinare α_j per un X_j di cui viene fatto il dicovery secondo m_i di X_i ($j = 1, 2, i = 1, 2$ in particolare quando $j = 1$, $i = 2$ e viceversa)per un dato pattern di tuple t.

Consideriamo i pattern di tuple non dominanti $S = t_1, ..., t_n$ con $t_l \neq t$ per $l = 1, ..., n$.

Se esiste, trova il minimo valore p_j del pattern di tuple t_l tale che

1. $m_j < p_j$, e
2. $m_{i-\epsilon} \geq p_i$ con $p_i = t_l[X_i]$

e genera $\alpha_j = p_j - \epsilon$.

In altre parole il minimo valore fra questi è dominato da $m_{i-\epsilon}$

3.3.3 Generazione di RFD

Implementazione

4.1 Tecnologie utilizzate

Al fine di testare l'algoritmo progettato, abbiamo sviluppato una applicazione scritta in Java. La scelta nell'utilizzare Java, come detto negli studi preliminari, è nata dalla necessità di ottenere maggiori prestazioni, dalla potenza del linguaggio e dal gran numero di librerie e framework utilizzabili al nostro scopo. Nella seguente sezione analizzeremo quelle che sono le librerie utilizzate per la realizzazione del seguente progetto.

4.1.1 AKKA

Akka è un insieme di strumenti per la realizzazione di applicazioni Java o Scala altamente concorrenti e distribuite. La scelta di questo framework è stata fatta in seguito alla necessità di rendere l'algoritmo quanto più efficiente possibile. La soluzione migliore si è rivelata essere quella del multithreading. A questo punto si è pensato che una impostazione *low level* dei thread fosse

poco sicura, quindi ci siamo orientati verso l'utilizzo di un container che gestisse le operazioni fondamentali di coordinazione. Oltre alla coordinazione del multithreading **AKKA** ci offre la possibilità di creare una applicazione distribuita. Questo argomento è stato trattato in fase di studio preliminare e temporaneamente accantonato, ciò non toglie che l'utilizzo di questo framework renda l'applicazione predisposta alla distribuzione. AKKA ci offre un *ambiente sicuro* su cui eseguire i nostri thread chiamato **Actor system**. L'Actor system è un container che prevede la gestione dei vari thread, chiamati **Actor**, garantendo servizi come la scalabilità della nostra applicazione, gestione della concorrenza e massime prestazioni. Per questa tecnologia si può fare riferimento alla documentazione ufficiale: `https://akka.io`

4.1.2 FastUtil

In fase di implementazione si è reso necessario un boost nelle prestazioni ed un uso efficiente della memoria da parte delle principali strutture dati utilizzate da Java. Per garantire questi servizi ci viene incontro una libreria chiamata **fastutil**. Questa libreria re-implementa le principali strutture di java garantendoci i servizi succitati. Le strutture sfruttate nella nostra applicazione sono:

- ***ObjectArrayList***: Re-implementazione della classe ArrayList di Java.
- ***Object2ObjectHashMap***: Re-implementazione della classe HashMap di Java.

4.1.3 Joinery Dataframe

Il DataFrame di joinery è una struttura dati simile a dataframe presente nella libreria *pandas* del linguaggio Python. Essa ci permette di memorizzare velocemente un *dataset* ed effettuare operazioni su di esso come se fosse una semplice tabella. La scelta di questa libreria è stata fatta in seguito alla valutazione dei vantaggi che offrivano i vari metodi implementati da Joinery. Si sono mostrati essere molto vantaggiosi i metodi che garantiscono l'import di un dataset da file csv e la gestione diretta ai dati in esso contenuti. Si può fare riferimento a tale libreria sulla documentazione ufficiale: `https://cardillo.github.io/joinery/`.

4.2 Struttura del progetto

Tutto il progetto è strutturato in 4 *package(Dataset,RFD,Actors,Utility)*. Nel corso della descrizione del codice, assumeremo che il lettore possieda già basi del linguaggio di programmazione *Java*. Per cui, sarà omessa la differenziazione tra metodi pubblici, metodi privati e metodi statici delle singole classi. Sia la **matrice delle distanze**, sia il **dataset** sono implementati utilizzando due diversi data frame (classe `DataFrame` della libreria DataFrame di Joinery citata nella sezione precedente). Per gli **insiemi C**, invece, è stata utilizzata una lista (classe `ObjectArrayList` della libreria FastUtil) avente come elementi i vettori contenenti le distanze e le coppie di identificativi delle due tuple da cui tale vettore è stato ricavato. Le **RFDs** sono state inserite in una classe di utility chiamata **RFDMap**.

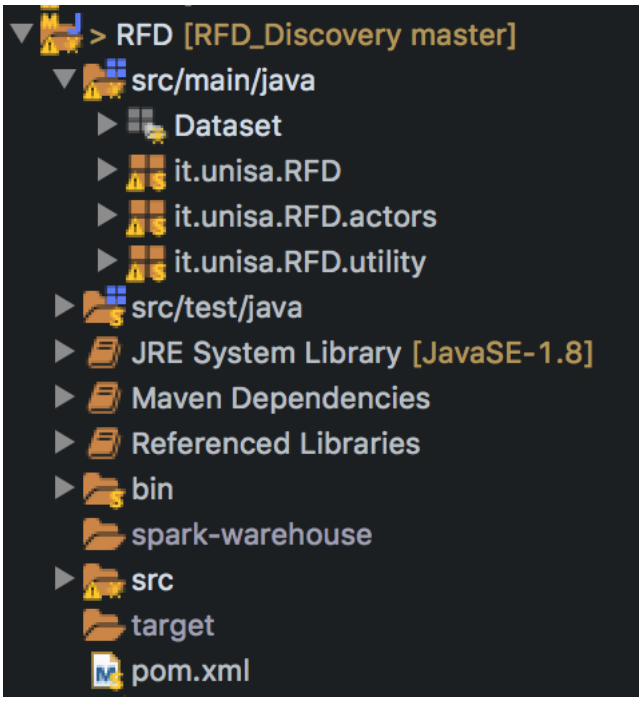

Figura 4.1: Struttura del Progetto

4.2.1 Package DataSet

Il package DataSet include una serie di DataSet in formato *csv*. I DataSet inclusi nel progetto sono:

- adult.csv
- balance-scale.csv
- breast-cancer-wisconsin.csv
- bridges.csv
- chess.csv
- Citiseer.csv
- cora.csv
- crawled-tweets.csv
- dataset.csv
- dataset_string.csv

- echocardiogram.csv
- hepatitis.csv
- horse.csv
- iris.csv
- restaurant.csv

4.2.2 Package RFD

Questo pacchetto contiene le classi che vengono utilizzate per le fasi principali del nostro algoritmo.

Figura 4.2: Package RFD

Il contenuto di questo pacchetto è:

DistanceMatrix.java: Questa classe contiene alcuni metodi statici fondamentali per il caricamento del dataset nella struttura dati *DataFrame* e per la creazione della Distance Matrix,tali metodi verranno spiegati di seguito.

```java
/**
 * Metodo che riceve in input nome del file csv e lo carica in un DataFrame
 * @param nameCSV nome file CSV
 * @param separator separatore di colonne utilizzato nel file
 * @param naString stringa nulla
 * @param hasHeader presenza di header nel file
 * @return Dataframe  DataFrame caricato da file
 * @throws IOException
 */
public static DataFrame<Object> loadDF(String nameCSV,String separator,
                                       String naString,boolean hasHeader,
                                                            int righeTaglio)
```

Listing 1: Metodo loadDF

Il metodo **loadDF** è utilizzato per importare il dataset dato in input come file csv. Per importare tale file ci siamo affidati al metodo messo a disposizione dalla libreria DataFrame di Joinery.Tale metodo ha il compito di caricare il file CSV e di trasformarlo in un oggetto della classe `DataFrame` di Joinery. Tutti i parametri relativi al CSV specificati nella firma saranno usati per interpretare correttamente tale file. Restituisce un oggetto di tipo `DataFrame` contenente il dataset.

Gli argomenti da passare al metodo sono i seguenti:

- **nameCSV** : il percorso dal quale caricare il file CSV;
- **separator**: il separatore di campi del file CSV;
- **naString**: il carattere di valore nullo presente all'interno del file CSV;
- **hasHeader**: presenza o meno dell'header come prima riga nel file CSV;
- **righeTaglio**: permette di effettuare un taglio sulle righe, inserire "0" se non si vuole effettuare alcun taglio. Tale parametro è stato inserito al fine di effettuare testing sulle grandezze dei dataset.

Durante l'esecuzione del nostro algoritmo è necessario calcolare un secondo dataset che nel nostro caso andrà ad essere inserito in una istanza della classe DataFrame di joinery. Questo secondo dataset è la matrice delle distanze che conterrà, per ogni coppia di pattern nel dataset originale, la differenza dei valori dei singoli attributi. Per calcolare la differenza fra i valori di un attributo, occorre prima di tutto capire il tipo dell'attributo in questione. Per questo scopo viene utilizzato il metodo offerto dalla classe DataFrame di Joinery: **types()**. A differenza del dataset originale, la matrice delle distanze avrà un campo in più inserito alla fine che rappresenterà la coppia dei pattern da cui è nata quella riga. Tale processo di creazione farà in modo di inserire una sola riga tra una coppia di tipo (1,2) e (2,1). Infine, effettuiamo un taglio delle righe duplicate. Tutto questo viene effettuato dal metodo **concurrentCreateMatrix**.

```
/**
 * @param inizio Indice di riga iniziale da confrontare
 * @param dimensione Indice di numero righe da confrontare
 * @param df dataFrame in input
 * @return Matrice Delle Distanze
 * Metodo per la creazione della matrice delle distanze.
 * Effettua anche taglio dei duplicati.
 * Versione per la parallelizzazione.
*/
public static DataFrame<Object> concurrentCreateMatrix
                                        (int inizio,int dimensione,
                                            DataFrame<Object> completeDF)
```

Listing 2: Metodo CreateDistanceMatrix

Gli argomenti da passare al metodo sono i seguenti:

- **inizio** : indice di riga di inizio del processo di creazione della matrice delle distanze su cui il thread corrente dovrà operare;

- **dimensione**: numero di righe su cui operare per il thread corrente;
- **completeDF**: dataframe precedentemente importato.

Come ultimo metodo richiamato dalla classe **DistanceMatrix** c'è **createOrderedDM**, questo metodo viene utilizzato per creare informazioni sulla matrice delle distanze al variare di tutti gli *RHS*. Queste informazioni saranno memorizzate come istanza di una classe di utility chiamata **OrderedDM**

```
/**
        * Metodo statico per la creazione di una DM ordinata
        * in base a RHS dato come parametro
        * @param indiceRHS colonna RHS
        * @param dm distance matrix
        * @return orderedDM DM ordinata
*/
public static OrderedDM createOrderedDM(int indiceRHS,DataFrame<Object> dm)
```

Listing 3: Metodo OrderedDMMethod

Gli argomenti da passare al metodo sono i seguenti:

- **indiceRHS** : indice di colonna del corrente RHS;
- **dm**: matrice delle distanze creata in precedenza .

FeasibilityOrderedDM.java: Questa classe contiene i metodi per l'esecuzione della fase di *Feasibility*, che ricordiamo essere la prima fase dell'algoritmo di RFD Discovery.

Il metodo più importante di questa classe è senz'altro il metodo statico **feasibilityTest**, esso darà inizio alla fase vera e propria di Feasibility. Questo metodo restituirà gli insiemi C fondamentali per l'inizio della prossima fase di *Minimality*.

```
/**
 * Metodo che permette di calcolare l'insieme c
 * dell'orderedDM dato come parametro
 * @param orderedDM
 * @return hashMap contenente l'insieme c
 */
public static Object2ObjectOpenHashMap
                          <String,ObjectArrayList<Tuple>>
                             feasibilityTest(OrderedDM orderedDM,
                                       DataFrame<Object> dmGenerale)
```

Listing 4: Metodo FeasibilityTest

Gli argomenti da passare al metodo sono i seguenti:

- **orderedDM** : istanza di una classe di utility che contiene informazioni sulla matrice delle distanze per un dato RHS;
- **dmGenerale**: matrice delle distanze creata in precedenza .

Questo metodo richiamerà, come visto nel capitolo precedente la dominanza tra due righe. La dominanza è definita attraverso un altro metodo implementato in questa classe. Il metodo è chiamato **dominance** e restituisce true se la prima tupla passata come parametro *domina* la seconda.

```
/**
 * Metodo che permette di verificare se tupla1
 * domina tupla2, in questo caso ritorniamo ture
 * @param tupla1
 * @param tupla2
 * @param dm
 * @return boolean
 */
        private static boolean dominance(int tupla1, int tupla2,
                                         DataFrame<Object> dm,int rhs)
```

Listing 5: Metodo Dominance

Gli argomenti da passare al metodo sono i seguenti:

- **tupla1** : una riga per il confronto;
- **tupla2** : seconda riga per il confronto;
- **dmGenerale**: matrice delle distanze creata in precedenza .
- **rhs** : indice di colonna che rappresenta l'RHS attualmente considerato;

MinimalityAndGenerationRFD.java: Questa classe contiene il metodo per l'inizio delle ultime due fasi finali. Durante l'esecuzione si cercano i minimi attraverso la fase di *Minimality* e, successivamente, vengono trovate le *RFD* durante la fase di *RFD Generation*(Le prime due fasi non sono parte dello studio di questo lavoro di tesi). Il Metodo che si occupa di dare inizio a questi ultimi due step è **startMinimalityAndGeneration**.

```
/**
* Metodo che da inizio alla fase di minimality e RFD discovery
**/
public static ObjectArrayList<RFDMap> startMinimalityAndGeneration
          (ObjectArrayList<ObjectArrayList<String>> allC,int colonne,
                                    OrderedDM orderedDM,DataFrame<Object> dM)
```

Listing 6: Metodo MinimalityAndGenerationRFD

Gli argomenti da passare al metodo sono i seguenti:

- **allC** : insiemi C trovati nella fase precedente di Feasibility;
- **colonne** : numero colonne del dataset;
- **orderedDM**: istanza di una classe di utility che contiene informazioni sulla matrice delle distanze per un dato RHS;
- **dM** : matrice delle distanze.

MainClass.java: Classe che da il via al nostro algoritmo, si occupa di creare il container per i nostri attori(*ActorSystem*) e di mandare un messaggio all'*Actor principale* per l'avvio delle varie fasi dell'algoritmo. Di seguito vediamo uno snippet che mostra quanto appena descritto.

```
df = DistanceMatrix.loadDF
        (args[0],args[3],args[4],true,Integer.parseInt(args[1]));

ActorSystem system = ActorSystem.create("SistemaAttoriRDF");

ActorRef act=system.actorOf
        (MainActor.props(df,Integer.parseInt(args[2])),"AttorePrincipale");

act.tell(new MainActor.ConcurrenceDistanceMatrix(), ActorRef.noSender());
```

Listing 7: MainClass

OrderedDM.java: Classe di utility per mantenere in memoria informazioni sulla matrice delle distanze per i vari RHS.

Le variabili di istanza di tale classe sono le seguenti:

- **allC** : insiemi C trovati nella fase precedente di Feasibility;
- **rhs** : indice di colonna che rappresenta l'RHS attualmente considerato;
- **lhs** : lista degli indici colonna degli LHS.

4.2.3 Package Actors

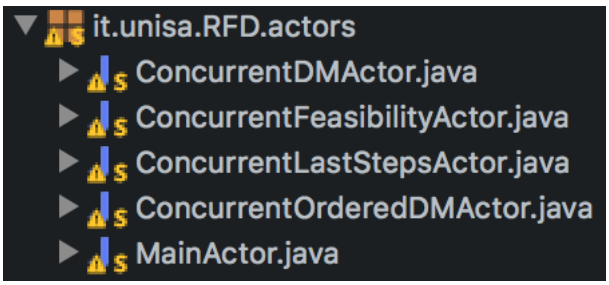

Figura 4.3: Package Actors

Questo pacchetto contiene le classi che rappresentano gli *Actors*. Queste classi si occupano di richiamare i metodi presenti nel package RFD in modo concorrente. Verrà istanziato un attore presente in questo pacchetto per quanti thread vogliamo far lavorare in concorrenza per ogni fase dell'algoritmo. Le classi per questo pacchetto sono:

MainActor.java: Classe principale che coordina il numero di attori che devono essere istanziati. Questa classe da il via ad ogni singola fase dell'algoritmo richiamando i metodi principali situati nelle classi del pacchetto precedentemente visto(RFD). Per ogni inizio di fase istanzia un numero di attori(thread) deciso come parametro da parte dell'utente(default:2).

Di seguito è riportato un esempio di lancio di messaggio da parte del *MainActor*, in particolare, in questo esempio viene lanciato un thread per la creazione di una parte di matrice delle distanze.

```
ActorRef actor=this.getContext().actorOf(ConcurrentDMActor.props());
actor.tell(new CreateConcurrentDM
                (inizioCorrente,dimension,this.df), this.getSelf());
```

Listing 8: Esempio invio messaggio da MainActor

ConcurrentDMActor.java: Questa classe, istanziata in numero pari ai thread desiderati in input, è la responsabile della creazione della matrice delle distanze. Essa riceve un messaggio da parte del *MainActor* e richiama il metodo **concurrenceCreateMatrix**. A lavoro ultimato invierà la sua parte di matrice delle distanze appena calcolata al MainActor. Il MainActor, infine, ricompone tutte le parti ricevute dai diversi thread ed elimina eventuali pattern ripetuti. Al termine otterremo la matrice delle distanze completa che

verrà utilizzata nelle fasi successive.

```
/*
 * Chiama metodo per la creazionedella DM parziale e
 * la invia al MainActor
*/
return receiveBuilder().match(CreateConcurrentDM.class, c->
{
   this.getSender().tell(new ReceivePartDM(DistanceMatrix
          .concurrentCreateMatrix(c.inizio,c.dimensione,c.completeDF)),
                                                          this.getSelf());
}).build();
```

Listing 9: Chiamata metodo concurrentCreateMatrix

ConcurrentOrderedDMActor.java: Questa classe attende un messaggio da parte del *MainActor* per dare il via alla fase di creazione degli *OrderedDM* richiamando l'apposito metodo nella classe DistanceMatrix.

```
//crea DM ordinata e la spedisce al mittente
return receiveBuilder()
.match(CreateOrderedDM.class, c->
{
     this.getSender().tell(new ReceiveOrderedDM
        (DistanceMatrix.createOrderedDM(c.indiceRHS, c.dm)),
                                                 this.getSelf());
}).build();
```

Listing 10: Chiamata metodo createOrderedDM

ConcurrentFeasibilityActor.java: Analogamente alle classi precedenti di attori, essa viene istanziata in numero pari ai thread desiderati. Queste istanze attendono un messaggio per dare il via alla fase di *Feasibility* chiamando il metodo della classe *FeasibilityOrderedDM*.

```
//Gestione feasibility test e risposta al mittente
return receiveBuilder()
   .match(CreateFeasibiity.class, cf->
   {
       OrderedDM dm = cf.orderedDM;
       Object2ObjectOpenHashMap<String,ObjectArrayList<Tuple>> hMap =
                   FeasibilityOrderedDM.feasibilityTest(dm,cf.dmGenerale);
       dm.setInsiemeC(hMap);
       this.getSender().tell(new MainActor.ReciveFeasibility(dm),
                                                       this.getSelf());
         }).build();
```

Listing 11: Chiamata metodo feasibilityTest

ConcurrentLastStepsActor.java: Questa classe attende un messaggio da parte del *MainActor* per dare il via alle ultime due fasi di *Minimality* e *RFD Generation*. Le due fasi hanno inizio chiamando il metodo apposito nella classe *MinimalityAndGenerationRFD*.

```
//Gestione last steps e risposta al mittente
return receiveBuilder()
   .match(CreateLastSteps.class, lsps->
   {
      this.getSender().tell(new MainActor
        .ReceiveLastSteps(MinimalityAndGenerationRFD
          .startMinimalityAndGeneration
            (lsps.allC,lsps.colonne,lsps.orderedDM,lsps.dM),
                      lsps.orderedDM.getRhs()),this.getSelf());
   }).build();
```

Listing 12: Chiamata metodo startMinimalityAndGeneration

4.2.4 Package Utility

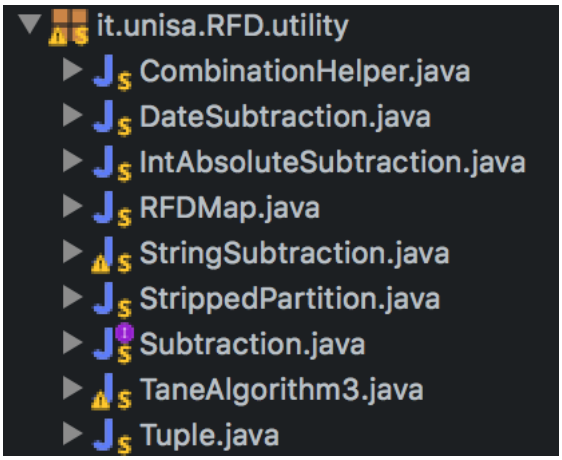

Figura 4.4: Package Utility

Questo pacchetto contiene le classi di utility utilizzate durante i 3 processi dell'algoritmo. Tale package comprende l'interfaccia per i vari tipi di differenza tra i campi e le relative implementazioni. In particolare, possiamo notare la presenza di 3 diversi tipi di differenze: *Interi*,*Date*,*String*. Particolare attenzione può essere posta sulla differenza tra stringhe. Quest'ultima è effettuata tramite l'algoritmo di *levenshtein*.

4.3 Requisiti

La versione di Java utilizzata è la 8 ed è stata utilizzata su di un architettura a x64. Il progetto è stato basato su un *maven project*, quindi, è necessaria l'installazione di maven. L'IDE utilizzato prevalentemente è *Eclipse*. Il progetto contiene le seguenti dipendenze:

- ***FastUtil***
- ***DataFrame Joinery***

- ***AKKA-actor***

- ***DataFrame Joinery***

Bibliografia

[1] F. P. P. S. T. R. Atzeni P., Ceri S., *Basi di dati: Modelli e linguaggi di programmazione*. McGraw Hill, 2013.

[2] T. M. Altamura A., Di Pasquale D., "Relaxed functional dependencies discovery," *Unisa*, 2017.