UNIVERSITÀ DEGLI STUDI DI TRENTO

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Progettazione e sviluppo di Interfacce Intelligenti basate sul paradigma della Cultura Implicita

Relatore:

Prof. Paolo Giorgini

Tesi di laurea di:

Fabiano Dalpiaz

Anno Accademico 2002-2003

Indice

Introduzione

Negli ultimi anni si sta osservando, nell'area dei sistemi informativi e delle basi di dati, il fenomeno dell'esplosione dei dati (*data explosion*). Esso consiste nell'avere a disposizione un numero sempre maggiore di testi, immagini, video e quant'altro sotto forma digitale, cioè memorizzati su computer.

Se, da un lato, questo fa sì che attraverso qualsiasi computer sia possibile avere a disposizione rapidamente un'enorme quantità di dati, comporta anche degli aspetti negativi: sostanzialmente essi consistono in una crescente difficoltà nell'individuare ciò che si sta cercando. Alcune stime indicano che un impiegato può occupare una percentuale di tempo fino al 25 - 35% nel cercare ciò che gli serve per il suo lavoro (una settimana ogni mese!) [23]. Il fenomeno descritto, inizialmente definito come data explosion, può essere ora ricondotto all'overload di informazioni (*information overload*): esso avviene quando la quantità di dati a disposizione fa sorgere dei problemi per gli utilizzatori.

Per cercare di porre rimedio a questa situazione, stanno nascendo molti applicativi che permettono la gestione di grandi quantità di informazioni. Per riuscire in questo obiettivo vengono proposte diverse alternative per esaminare i dati (all'interno di uno stesso programma): se da un lato questo è positivo, permettendo all'utente di scegliere la modalità di ricerca che ritiene migliore, dall'altro tali funzionalità incrementano la complessità dell'interfaccia del software. Con questo si intende sottolineare che l'utente (soprattutto se è la prima volta che utilizza l'interfaccia) potrebbe rimanere disorientato dalle tante opzioni e per assurdo non sapere come procedere.

Sono stati condotti molti studi con lo scopo di migliorare le interfacce utente, e sono state proposte diverse linee guida per la loro progettazione. Una soluzione particolare consiste nelle *interfacce intelligenti* (*intelligent (user) interface – I(U)I*), le quali hanno lo scopo di adattarsi al software e all'utilizzo da parte degli utenti dello stesso, modificando l'interfaccia sulla base delle osservazioni effettuate sull'utilizzo del software.

Per creare un'interfaccia definibile intelligente possono essere utilizzate diverse tecniche, anche non studiate appositamente per risolvere questo problema. Un primo insieme di metodologie che forniscono strumenti validi è il *data mining* [11] [19] [40] [42]. Esso è definibile come "*Processo di estrazione di conoscenza da banche dati di grandi dimensioni tramite l'applicazione di algoritmi che individuano le associazioni "nascoste" tra le informazioni e le rendono visibili*" [11].

Alle classiche tecniche di data mining (che sono degli ottimi strumenti per l'analisi delle informazioni raccolte sull'utilizzo del software) si affiancano altri strumenti, che cercano di assistere l'utente nell'utilizzo delle funzionalità offerte dalle applicazioni create per gestire le informazioni. Il *collaborative filtering* implica la realizzazione di sistemi che suggeriscono degli elementi sulla base delle valutazioni espresse da altri utenti considerabili simili a quello corrente. Esistono numerosi esempi reali dell'applicazione di tecniche di collaborative filtering, fra i quali il più conosciuto (uno dei primi anche cronologicamente) è probabilmente il sito di e-commerce Amazon. Il concetto di *cultura implicita* (*Implicit Culture*) è invece stato formulato inizialmente da Blanzieri e Giorgini [2], ed approfondito in vari altri lavori principalmente legati all'Università di Trento. La cultura implicita può essere introdotta con la seguente definizione: "*Un agente che interagisce con un ambiente avendo poca conoscenza dello stesso agisce in maniera non ottimale. Se in tale ambiente agisce anche un gruppo di altri agenti, dovrebbe essere utile utilizzare le informazioni derivanti dalle loro azioni allo scopo di migliorare le conoscenze ed il comportamento del singolo agente*" [2]. Si dice che è in atto un *fenomeno di cultura implicita* quando un gruppo di agenti agisce in modo conforme alla cultura di un altro gruppo senza dover essere a conoscenza della cultura stessa. Un *sistema di supporto alla cultura implicita (SICS)* è definito come un sistema (tipicamente software) che ha lo scopo di facilitare la realizzazione di fenomeni di cultura implicita, inducendo un utente ad eseguire azioni conformi alla cultura dell'ambiente nel quale sta operando.

L'obiettivo principale di questa tesi è quello di progettare e sviluppare sistemi di supporto alla cultura implicita. Verrà presentato in dettaglio un modulo effettivamente sviluppato che si integra con un software web per la consultazione di basi di dati cartografiche (3wGIS). Tale modulo suggerisce all'utente l'utilizzo di determinate funzionalità, specificandone anche i parametri qualora fossero necessari. Saranno inoltre presentati dei risultati sperimentali realizzati allo scopo di valutare la reale efficacia del sistema.

La tesi si articola in quattro capitoli, oltre all'introduzione ed alle conclusioni. Il capitolo 1 riguarda il problema affrontato nel resto della tesi (l'analisi dei *sistemi per la gestione dell' overload di informazioni)*: dopo averlo definito, si illustrano le tecniche utilizzate per risolverlo (fra queste il data mining, il collaborative filtering e la cultura implicita). Sono inoltre analizzati i problemi correlati alla presenza di numerose funzionalità all'interno dei software di collegamento ai dati e i metodi per risolvere questa situazione (*interfacce intelligenti*). Il capitolo 2 introduce il framework realizzato, presentandone dapprima la struttura generale, per analizzare poi più in dettaglio varie soluzioni utilizzabili per l'effettiva implementazione dei moduli che compongono tale architettura. Nel successivo terzo capitolo si prende in esame il software effettivamente sviluppato, esaminandolo nei dettagli; particolare attenzione viene dedicata alle scelte effettuate, sia in relazione all'architettura del software, che per quanto riguarda gli algoritmi utilizzati. Infine

(capitolo 4) si affronta il problema della valutazione dell'efficacia dell'interfaccia sviluppata, attraverso l'esecuzione di alcune serie di test.

Si conclude questa introduzione segnalando che il lavoro svolto è stato realizzato presso l'azienda *Informatica & Servizi (I&S)* di Trento, impresa operante nel campo della cartografia. Il modulo sviluppato è infatti un'integrazione per un software realizzato da I&S chiamato *3wGIS*, che permette la consultazione di basi di dati cartografiche tramite un'interfaccia web. La permanenza in I&S è iniziata a maggio 2003 nell'ambito di uno stage, per poi proseguire a partire da inizio agosto con la realizzazione della presente tesi.

Capitolo 1

Sistemi per la gestione dell'overload delle informazioni

1.1 L'overload delle informazioni

Quando si parla di overload di informazioni si intende considerare un fenomeno la cui nascita è molto recente: la presenza di un numero di informazioni così elevato da far sì che gli utilizzatori abbiano dei problemi nel loro utilizzo. Infatti, è ormai comprovato che il numero di informazioni disponibili sia in continua crescita, e che tale aumento non si fermerà. Ci si può chiedere quali sia il motivo di tale proliferazione di contenuti. La risposta è piuttosto articolata, dato che non esiste un unico fattore che abbia portato a tutto ciò.

Sicuramente l'evoluzione tecnologica ha un posto rilevante, visto che in pochissimi anni si è passati dall'utilizzo di informazioni prettamente testuali (anni '80), all'era multimediale, in cui immagini, musica e video possono essere riprodotti non solo sui personal computer casalinghi, ma persino su dispositivi portatili quali palmari e telefoni cellulari. Oggi l'utilizzo di fotocamere e videocamere digitali è frequente: basti pensare a quante persone scelgano di archiviare le proprie fotografie su supporto magnetico (CD o DVD) anziché farle stampare su carta lucida da uno studio fotografico. Tutto questo è supportato da una crescita parallela dell'hardware di tipo storage, le cui capacità crescono ed i cui costi per megabyte (anche se ormai sarebbe meglio parlare di costo per gigabyte) decrescono. Per capire quanto queste informazioni incidano dal punto di vista dell'occupazione di memoria fisica, basti pensare che una singola fotografia digitale è grande quanto un testo contenente 18.000 parole [23]. Se si considerasse solamente l'occupazione di memoria che necessitano non si avrebbe però un fenomeno di overload di informazioni: esso è dovuto al fatto che il numero di questi contenuti è in costante aumento.

Un mezzo che permette l'accesso a queste informazioni da parte di svariate persone ubicate ovunque sul globo terrestre è il *computer networking*, ed in particolare la rete globale *Internet*. Le reti locali sono sempre più numerose e funzionano ad alta velocità nei collegamenti interni (lo standard Ethernet è arrivato al Gigabit/sec), e una parte

consistente di tali reti ha un accesso ad Internet, permettendo la visione di informazioni pubbliche presenti all'interno della rete a chiunque disponga di un accesso al web.

Anche per quanto riguarda i database la situazione è la stessa. Infatti, il livello tecnologico raggiunto dagli odierni database permette la gestione di grandi quantità di informazioni mantenute su elaboratore sotto forma di dati; essi, oltre ai classici formati numerici e testuali, possono essere anche informazioni multimediali. Le tecniche di accesso ai dati utilizzate sfruttano al meglio gli algoritmi esistenti, permettendo un reperimento rapido delle informazioni.

Certamente la presenza di un numero sempre maggiore di informazioni può essere considerato un fatto positivo, dato che in tal modo è molto più facile sfruttare la conoscenza degli altri. Esso rappresenta però anche un problema, poiché risulta sempre più difficile riuscire ad accedere in modo efficiente a queste enormi masse di informazioni. E' infatti comune la presenza di informazioni "spazzatura" che non interessano a nessuno, che possono essere più facili da individuare rispetto a quelle realmente rilevanti.

D'altra parte una delle caratteristiche peculiari degli umani sta nella nostra capacità di adattarci alle informazioni e nel notare fatti inusuali, riuscendo ad esprimere giudizi rapidi su ciò che ci interessa e ciò che non riteniamo utile (basti pensare alla capacità di giudicare un libro dalla copertina o da una breve introduzione) [17]. Nel caso delle informazioni odierne (specialmente quelle digitali), la loro grande quantità fa però in modo che non sia così facile riuscire ad esprimere un giudizio su esse senza averle prima visionate.

Esistono anche problemi relativi alle interfacce utente dei software creati per l'accesso a grandi quantità di informazioni: tali applicazioni riescono spesso a gestire le informazioni in modo efficace, ma offrono un'interfaccia ostica e troppo ricca di opzioni che tende a confondere gli utilizzatori. Infatti, per poter esaminare al meglio le informazioni, è comune l'utilizzo di funzioni molto complesse, che possono utilizzare diversi parametri (da chiedere all'utente) o necessitare di svariati passi per la loro esecuzione, rendendo l'interfaccia poco user-friendly e richiedendo un alto livello di conoscenza del dominio da parte degli utenti.

Diverse soluzioni sono state realizzate per riuscire a facilitare l'utente nella ricerca delle informazioni che più gli servono. Inoltre, è da tenere in considerazione un ulteriore utilizzo di banche dati di grandi dimensioni: la scoperta di associazioni "nascoste" fra gli elementi presenti all'interno delle informazioni. Molte relazioni sono infatti di facile rilevamento o già note a priori, ma altre vengono individuate solamente con l'utilizzo delle informazioni e la loro analisi sistematica: esse, se scoperte attraverso appositi strumenti, possono essere utilizzate per facilitare l'utente nell'esecuzione delle future ricerche.

1.2 Interfacce intelligenti

In questo paragrafo si introduce il concetto di *interfaccia intelligente*. Dapprima sono individuati i motivi per cui sono richieste interfacce di questo tipo; viene poi definito il concetto di *interfaccia intelligente*, concludendo con l'analisi delle tecniche utilizzabili in questo campo.

1.2.1 Perché le interfacce intelligenti?

E' ormai assodato il fatto che, con il passare del tempo, i sistemi software stiano divenendo sempre più potenti e veloci: questo avviene sia tramite l'utilizzo di algoritmi efficienti che grazie al rapido sviluppo dell'hardware. Tale fenomeno è certamente positivo se si considera che l'utilizzo del software permette la riduzione di costi e tempo in moltissimi campi, ad esempio nei casi in cui esso aiuti il personale nello svolgimento delle proprie mansioni. D'altro canto è innegabile la necessità di utilizzare diverse funzionalità per l'esecuzione completa di attività complesse: i requisiti che devono essere soddisfatti dalle nuove versioni dei software sono in costante crescita, e la soluzione più efficace per soddisfarli consiste nell'offrire molti strumenti all'interno dello stesso software. Nonostante sia possibile raggruppare le funzioni in gruppi omogenei tramite utilizzando dei menu risulta comunque difficile per gli utenti individuare lo strumento adatto in una certa situazione e l'utilizzo corretto dello stesso. Un qualsiasi utilizzatore conosce il proprio obiettivo nell'utilizzo del software (ha un'idea di quale debba essere il risultato finale), però non necessariamente è a conoscenza dei passi necessari per ottenere l'output atteso utilizzando il programma. Il problema è quindi duplice: la ricerca della funzionalità adatta in una determinata situazione ed il suo corretto utilizzo. E' necessaria una certa *esperienza (expertise)* per sfruttare efficacemente ed efficientemente il software: in caso contrario l'utente è inesperto, e necessita di una soluzione al proprio problema. Nella vita di tutti i giorni, in una situazione simile (anche non legata all'ambito informatico) si ricorre ad un *intermediario umano* [12], il quale ha maggiore esperienza nell'ambiente in cui stiamo operando e riesce a darci dei suggerimenti per procedere. Nel campo dei sistemi per l'accesso alle informazioni questa non è certamente la soluzione migliore, sia perché richiede la presenza di un'apposita persona, sia perché sarebbe molto meglio se fosse lo stesso software a fornire aiuto. L'interrogativo da porsi è ora come questo software possa aiutare; la risposta è scontata: basta che l'interfaccia utente sia migliore e dia consigli per la ricerca anziché ostacolare presentando innumerevoli funzioni fra le quali si è costretti a districarsi. La risoluzione di questo problema non è però un'attività semplice: non si tratta infatti solamente di implementare un modo "ordinato" per l'accesso alle funzionalità del software, ma bisogna fornire loro qualcosa in più, ovvero un qualche tipo di intelligenza. E' proprio dall'unione dei due concetti di *interfaccia utente* e di *intelligenza* che nasce il concetto di *interfaccia intelligente*.

1.2.2 Definizione di interfaccia intelligente

Come per molti altri termini, non esiste una definizione universalmente accettata di *interfaccia intelligente*, nel senso che diversi studiosi le definiscono differentemente. In questo caso si è pensato di utilizzare lo stesso approccio presentato in [43], ovvero fornire prima una descrizione di **cosa è** un'interfaccia intelligente, per poi darne una **non definizione**, elencando cosa non è considerabile tale. La definizione del concetto, secondo P.A.Hancock e M.H.Chignell è la seguente:

"*Un'entità intelligente che media fra due o più agenti che interagiscono fra loro, i quali sono in possesso di una comprensione incompleta delle reciproche conoscenze"*

Si indica quindi fortemente il rapporto esistente fra i vari agenti, nel senso che nell'ambito di riferimento opera più di un soggetto, ma essi non sono così in contatto da poter sfruttare le reciproche conoscenze: proprio per questo esiste l'interfaccia intelligente, ovvero un'entità intelligente che interagisce con loro fornendo le conoscenze presenti ma che non verrebbero sfruttate altrimenti.

Definiamo ora cosa non intendiamo per interfaccia intelligente [45] e [44]:

- *Un sistema intelligente non ha necessariamente un'interfaccia intelligente*: i sistemi di ragionamento basati sulla conoscenza sono certamente intelligenti, perché danno dei suggerimenti agli utenti, però l'intelligenza risiede negli algoritmi di intelligenza artificiale che contengono, anziché nell'interfaccia;
- *Un'interfaccia ben progettata non è intelligente*: esse sono spesso progettate seguendo delle linee guida, ma non tenendo in considerazione la diversità degli utenti (diversi utenti si comportano diversamente) mancano del requisito dell'intelligenza.

E' quindi necessario che l'interfaccia intelligente agisca fra utente umano e computer, permettendo di colmare la distanza fra queste entità.

1.2.3 Tecniche e classificazioni

In questo paragrafo si propongono delle tecniche utilizzate nelle interfacce intelligenti. E' bene sottolineare che esse possono essere utilizzate sia separatamente che contestualmente, a seconda dell'ambito in cui si opera. Prima di presentare le tecniche utilizzabili si può abbozzare una prima di classificazione relativa ai tipi di aiuti proponibili [12]:

1. *Aiuto concettuale*: permette di superare i problemi che nascono durante lo svolgimento della ricerca, sia relativi alla terminologia (*aiuto terminologico*) attraverso la presentazione di sinonimi che permettono l'arricchimento del

vocabolario dell'utente, che in relazione alla strategia di ricerca (*aiuto strategico*), aiutando l'utente nei casi in cui la ricerca non dia l'esito auspicato (per esempio non si ottiene alcun risultato);

2. *Aiuto tecnico*: tale tipo di aiuto è legato alla conoscenza dell'interfaccia utente-sistema, e mira a fare in modo che l'utente conosca progressivamente meglio l'interfaccia in cui opera, permettendogli la realizzazione di ricerche più complete ed efficienti.

Vengono ora riportate le principali tecniche utilizzabili per lo sviluppo di interfacce intelligenti [44]:

1. *Adattamento all'utente*: queste tecniche fanno sì che l'interazione utente – sistema si adatti a differenti utilizzatori e a distinte situazioni di utilizzo. Il sistema prende decisioni relative a come presentare l'interfaccia a run-time. Esempi di queste tecniche sono modifiche nella disposizione del layout, l'uso di cursori e combinazioni di colori personalizzati nell'interfaccia;
2. *Modelli di utente*: si ricorre a questi metodi nei casi in cui si vogliano mantenere informazioni relative ai vari utenti. Un *modello di utente* è definibile come la rappresentazione del sistema della conoscenza dell'utente. Esistono tre tipi di modelli:
    a. *Modello che ha il disegnatore del sistema*: chi progetta l'interfaccia ha un'idea dell'utente tipico, e si basa su questo stereotipo. Questo modello ha il problema che tutti gli utenti sono considerati uguali;
    b. *Modello fornito al sistema dall'utente stesso*: l'interfaccia si configura in base al modello legato all'utente. Un esempio tipico sono i profili di UNIX, i quali caricano delle impostazioni personali quando un utente effettua il login (i files *.profile*);
    c. *Sistemi adattativi*: il sistema crea un modello dell'utente basato sulla sua interazione con il sistema. Questo modello ha il problema di richiedere un tempo di setup (in cui l'utente utilizza un sistema di default), però è quello migliore, maggiormente dinamico ed effettivamente utilizzato.
3. *Tecniche di linguaggio naturale*: esse permettono al sistema di interpretare o generare linguaggio naturale sotto forma di testo o voce. Tali sistemi utilizzano vocabolari e modelli di dominio. Questo tipo di tecniche è forse la via più interessante, ed è soggetto di numerose ricerche: chiaramente vi sono grandi problemi, fra i quali l'ambiguità è uno fra i principali. Infatti la macchina necessita di dati non ambigui, mentre il linguaggio umano ha molte sfaccettature e

particolarità difficili da interpretare correttamente (ed univocamente) in modo automatico;

4. *Modellazione del dialogo*: queste tecniche permettono al sistema di mantenere un dialogo in linguaggio naturale con l'utente. Esse sono ancora più complesse che quelle legate al linguaggio naturale, e richiederanno diversi anni prima di essere applicate effettivamente e con buoni risultati.

1.3 Software per la gestione dell'overload di informazioni

Finora sono stati esaminati distintamente due concetti: *l'overload di informazioni* e le *interfacce intelligenti*. In questo paragrafo si mostra la relazione che intercorre fra essi.

I programmi sviluppati per gestire il problema dell'overload di informazioni riescono tutt'oggi a svolgere il proprio compito in maniera efficace: essi soffrono però dello stesso problema della maggior parte dei software moderni di grandi dimensioni, cioè risultano difficili da utilizzare. Le tecniche utilizzate per affrontare con successo la gestione e la ricerca di informazioni sono piuttosto complesse, e per produrre il risultato migliore devono essere configurate in modo appropriato tramite la specifica di un gran numero di parametri. Per un comune utilizzatore del software non è facile riuscire in questo compito, soprattutto nel caso in cui non conosca approfonditamente l'ambito in cui agisce e/o le tecniche utilizzate per la gestione dell'overload di informazioni.

E' quindi compito del produttore del software fornire dei mezzi per l'utilizzo delle funzionalità proposte, e le interfacce intelligenti risultano la soluzione appropriata, attraverso la modifica della visione dell'ambiente software allo scopo di facilitare l'utente (soprattutto se inesperto) nell'utilizzo dell'applicazione. Questo può essere fatto proponendo lo strumento presumibilmente idoneo per l'utente, ed eventualmente mostrandone diverse variazioni relative ai parametri da impostare. Per riuscire in questo compito è necessaria la presenza di almeno due attività distinte: una relativa all'osservazione del comportamento degli utenti ed una che aiuti gli utenti meno esperti nell'utilizzo del software. L'idea è quindi di creare un tramite software che apprenda le conoscenze dell'ambiente possedute dagli utenti più esperti e che le trasferisca nei momenti più opportuni a quelli che posseggono minore conoscenza.

1.4 Un esempio di software: 3wGIS

Per comprendere al meglio il problema delle interfacce utente complesse in relazione ai software per la gestione dell'overload di informazioni si propone ora un esempio di software che affronti positivamente le informazioni ma che non proponga in modo chiaro le proprie funzionalità agli utenti. Esso è innanzi tutto esaminato in generale, per passare

in seguito ad un'analisi più legata alla complessità dell'interfaccia utente ed alla necessità di trovare una soluzione a tale problema.

1.4.1 Introduzione a 3wGIS

Il software che si è deciso di esaminare è prodotto da Informatica & Servizi S.r.l. di Trento, e si chiama 3wGIS. Esso è un applicativo web che ha lo scopo di permettere la consultazione di banche dati cartografiche, ovvero legate a mappe digitalizzate.

Tale software è utilizzabile tramite un comune web browser, e suddivide l'area di visualizzazione in due parti principali:

- da un lato viene mostrata la mappa di riferimento;
- dall'altro è presente una barra di navigazione che ha lo scopo di fornire un'interfaccia fra la banca dati cartografica e la mappa stessa;

In figura 1.1 è mostrata un'immagine ottenuta durante l'esecuzione del software.

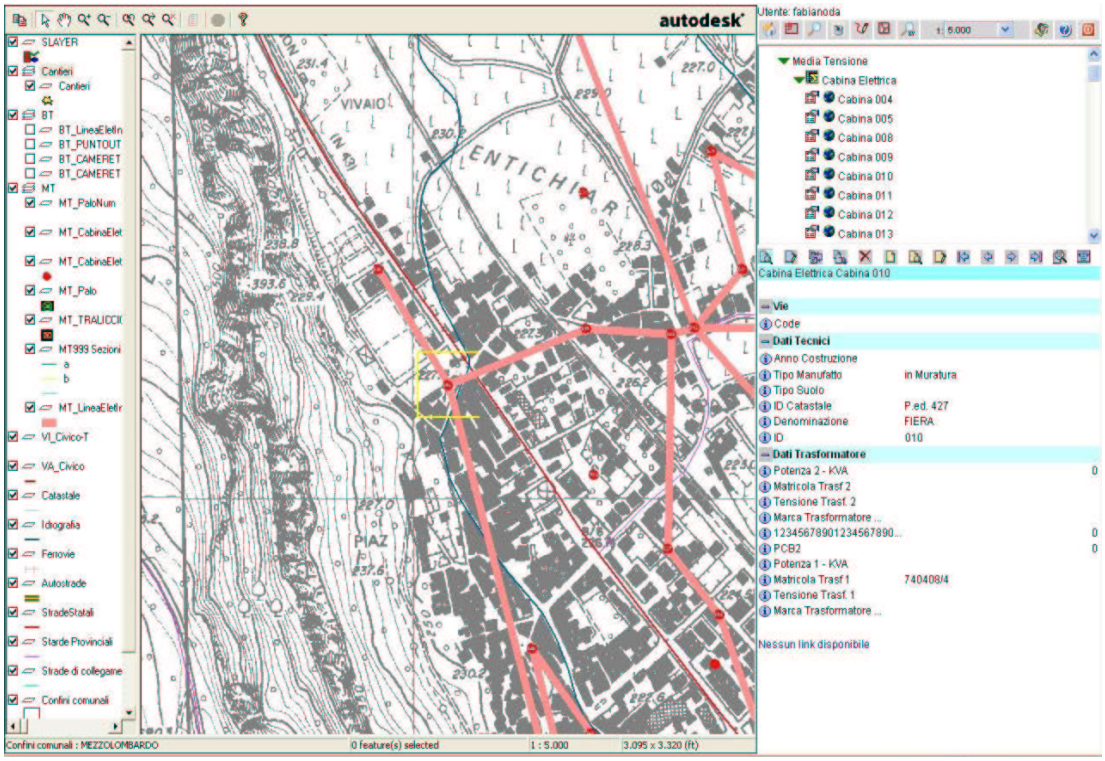

Figura 1.1: 3wGIS in esecuzione

La mappa è visualizzata attraverso l'utilizzo di un apposito plug-in di tipo ActiveX prodotto da Autodesk, chiamato MapGuide. Questo è un software di consultazione di file

cartografici nel formato *mwf*, ormai uno standard per quanto riguarda le mappe da visualizzare tramite web.

Ogni mappa è costituita da uno o più layer, o livelli, ognuno dei quali rappresenta un tipo di elemento presente nella cartografia. Se si considera per esempio il campo dei trasporti, è possibile pensare all'esistenza di un layer per la rete ferroviaria, uno per quella autostradale, un terzo per le altre strade.

I layer possono essere inoltre raggruppati per similarità: infatti si può decidere di creare un gruppo che comprenda tutte le strade (sia autostrade che non). I livelli possono essere in un dato momento accesi o spenti, a seconda che essi vengano visualizzati sul monitor o meno.

La base di dati che contiene i dati relativi alla cartografia è consultabile attraverso la parte di schermo contenente l'interfaccia di navigazione. I tipi di database per ora supportati sono Microsoft Access e Microsoft SQL Server; è in progetto il supporto anche a banche dati di tipo Oracle.

All'interno del database possono essere presenti dati relativi ad elementi che appaiono nella mappa (o che comunque sono relativi alla mappa) e ad altri che invece non sono direttamente correlati con la cartografia. Per quanto riguarda i tipi di elementi raggiungibili attraverso l'interfaccia di navigazione, essi sono classificati utilizzando due tabelle accessorie presenti nello stesso database, che suddividono i vari tipi gerarchicamente:

- Un primo livello è fornito dalle *categorie*. Questo ha lo scopo di effettuare una prima selezione fra gli elementi ad un livello più grezzo: ad esempio si può pensare di suddividere gli elementi legati alla viabilità da ciò che invece riguarda gli immobili.
- Il secondo livello separa gli elementi della stessa categoria in *classi* omogenee. Tutti gli elementi appartenenti alla stessa classe sono simili fra loro, ovvero possono essere racchiusi nella stessa struttura (ad esempio una tabella di un database). Proseguendo con l'esempio iniziato in precedenza, possiamo suddividere la categoria viabilità in strade principali, strade secondarie, autostrade, ferrovie, … mentre gli immobili possono essere suddivisi in fabbricati, palazzine, condomini, ville, …

Le operazioni eseguibili all'interno di 3wGIS per la ricerca di elementi sono molte:

- Effettuare una ricerca attraverso l'ausilio di una vista ad albero, la quale per prima cosa permette di navigare fra le categorie, e successivamente consente la scelta della classe;

- Utilizzare una comune query. La costruzione di tale stringa è facilitata tramite l'ausilio di apposite caselle di testo e combo box che mostrano le alternative disponibili (spesso indicando la descrizione anziché i nomi dei campi), permettendo anche a chi non conosca la sintassi sql di eseguire queries personalizzate;
- Navigare fra i record seguendo un ordinamento determinato dai campi chiave delle classi: una volta che si seleziona un elemento di una certa classe ci si può muovere fra gli elementi della classe stessa avanti, indietro, oppure verso il primo o l'ultimo record;
- Si possono memorizzare le query effettuate o creare/utilizzare delle strutture predefinite, in cui è sufficiente impostare i valori da assegnare ai vari parametri della query: ad esempio si può effettuare una ricerca di strade a seconda del loro numero, predisponendo una certa struttura nella quale inserire solo i valori al momento della ricerca;
- Per operazioni ancora più complesse è possibile estendere il software creando delle procedure personalizzate (dette *procedure custom*), le quali mostrano un pulsante che rimanda ad una nuova pagina che contiene il tipo di ricerca personalizzato. Tali procedure custom possono essere utilizzate anche per fini diversi dalla ricerca (creazione di funzioni personalizzate relative all'ambito considerato).

Il software è utilizzato da diversi tipi di clienti che agiscono in ambiti diversi: ciò che li accomuna è il collegamento con il territorio (ovvero la necessità di utilizzare la cartografia): alcune istanze del software sono state proposte per enti pubblici quali comuni, altre per enti di distribuzione dell'energia elettrica e di altri servizi.

1.4.2 3wGIS e interfacce utente complesse

Dopo aver esaminato cosa sia il software 3wGIS e le sue funzionalità principali (quelle maggiormente legate alla ricerca), si passa ad un'analisi del software ancora più legata al problema delle interface complesse. Saranno analizzati gli aspetti positivi di 3wGIS, ovvero le caratteristiche che permettono di affrontare in modo efficace il problema, ma verranno evidenziati anche gli aspetti negativi, e le eventuali carenze dello stesso software.

Si può considerare 3wGIS un valido esempio di software che riesce ad affrontare e gestire con successo una grande quantità di informazioni, offrendo diversi strumenti che permettono la ricerca e l'esplorazione: il problema dell'overload di informazioni legato ai dati cartografici è gestito quindi in modo soddisfacente (non per questo non migliorabile ovviamente).

Uno degli aspetti maggiormente positivi è la suddivisione su due livelli degli elementi. La presenza di una categoria e di una classe per identificare un certo oggetto è sicuramente utile per aiutare l'utente a districarsi all'interno di una grande quantità di dati, permettendo anche ad una comune interfaccia di organizzare in gruppi omogenei gli elementi.

Altra caratteristica positiva sta nel supporto di 3wGIS ad un gran numero di sistemi differenti di ricerca. Infatti l'utente può scegliere quello che più gradisce per effettuare le proprie ricerche, e non è per forza costretto ad utilizzare un solo metodo.

Proprio la presenza di molti tipi di ricerca, che aiuta notevolmente un utente esperto, risulta un grosso ostacolo per chi non ha conoscenza sufficiente dell'ambiente, cioè per coloro che hanno poca esperienza. In particolare, non esiste allo stato attuale alcun mezzo che permetta di sfruttare l'esperienza acquisita dagli altri utilizzatori del software.

Un altro punto debole dell'applicazione sta nella ricerca ad albero. Nonostante permetta una discreta scrematura all'interno degli elementi presenti nel database, si arriva ad un punto in cui tutti gli elementi della stessa classe vengono visualizzati (potrebbero essere molti).

Questi problemi fanno sì che il software necessiti di essere integrato tramite l'aggiunta di un modulo che offra un'interfaccia intelligente. Sono individuabili quindi problemi legati sia alla quantità di funzionalità (nel caso specifico i vari tipi di ricerca) che connessi con i parametri per la ricerca (nella ricerca classica – tramite query – sono da specificare alcune impostazioni).

1.5 Tecniche per affrontare le interfacce utente complesse

Dopo aver analizzato il problema delle interfacce utente troppo complesse, si cerca ora di fare una panoramica sulle principali tecniche utilizzabili per affrontarlo. Verranno evidenziati i loro aspetti positivi e le loro lacune, prestando particolare attenzione agli ambiti ai quali tali metodi si adattano meglio o sono già stati utilizzati in passato. Nella prima sottosezione si esamina la scienza definita come *data mining*, che è la madre di tutte le tecniche utilizzate in seguito. Si passa poi ad una panoramica sul *collaborative filtering* (noto anche come *recommender systems*), ovvero i sistemi che cercano di facilitare le ricerche degli utenti sulla base delle precedenti valutazioni espresse da altri soggetti. Infine si introduce il concetto di *cultura implicita*, evidenziandone il collegamento con il collaborative filtering.

1.5.1 Data mining

1.5.1.1 Definizione: data mining

Il termine data mining è molto generale e per esso sono presenti un gran numero di definizioni in diversi libri e pubblicazioni. Esse hanno sostanzialmente lo stesso significato; per mostrare quali sono gli aspetti più rilevanti ne sono ora riportate alcune, in cui verranno sottolineate le parole (i termini) chiave:

- *Le tecniche di ricerca dell'informazione nascosta nei dati di una Data Warehouse* [1];
- *L'estrazione di informazioni o sequenze (patterns) interessanti (non banale, implicita, precedentemente sconosciuta e potenzialmente utile) dai dati in database di grandi dimensioni* [19];
- *L'estrazione automatizzata di informazioni nascoste e predittive da (grandi) database* [40];
- *Processo di estrazione di conoscenza da banche dati di grandi dimensioni tramite l'applicazione di algoritmi che individuano le associazioni "nascoste" tra le informazioni e le rendono visibili* [11].

E' sufficiente dare una rapida occhiata a queste definizioni per notare come esse siano somiglianti ed accorgersi che in parte si integrano; a questo scopo si cerca di estrarre le parole chiave individuate e di esaminarle, in modo da trovare le caratteristiche del data mining:

- *Informazione nascosta, implicita, associazioni "nascoste"*: questo è uno dei requisiti fondamentali di una buona definizione di data mining, dato che indica con esattezza quello che si vuole trovare, cioè qualcosa che non risalti all'occhio senza alcuna analisi, ma che richieda un processo di estrazione di conoscenza per essere riscontrato. E' inoltre importante la presenza del termine *associazione*, indicante che particolare attenzione è dedicata alle sequenze o alle regole ricavabili dai dati, e non solamente alle informazioni singole;
- *Data WareHouse, database di grandi dimensioni, grandi database*: viene subito richiamato il concetto di una grande quantità di informazioni. Se avessimo solo una piccola banca dati non sarebbe necessaria l'esistenza di alcuna tecnica per esaminarla, ma qui è focalizzato il termine "grande dimensione", che implica appunto l'esistenza di metodologie apposite per l'analisi;

Data warehouse: una collezione di dati orientata al soggetto, integrata, non volatile e dipendente dal tempo [W.H. Inmon].

- *Non banale e Potenzialmente utile*: con questo termine si fa riferimento al fatto che, avendo a disposizione un'enorme quantità di dati, è necessario estrarre solamente le informazioni che sono realmente importanti, escludendo quelle "visibili ad occhio nudo", ovvero senza alcuna analisi, e non considerando nemmeno quelle inutili (ovviamente);
- *Predittiva*: lo scopo ultimo del data mining è di essere una tecnica utile per futuri miglioramenti nel campo di analisi. E' quindi necessario che vengano estratte informazioni "predittive", ovvero che diano delle indicazioni ben precise relative al business che si sta esaminando, e che permettano di individuare rapidamente ed efficacemente le soluzioni per migliorarlo.

Una definizione completa di *data mining* è quindi l'insieme di queste parole chiave, che rappresentano in modo sintetico ma completo il concetto.

Esistono molti sinonimi in letteratura utilizzati al posto del termine *data mining*; si ritiene importante elencare almeno i più noti nel caso in cui vengano utilizzati in seguito: Knowledge discovery in databases (KDD), knowledge extraction, data/pattern analysis, Knowledge mining in databases [19].

1.5.1.2 Campi di applicazione

E' bene considerare ora le aree in cui è possibile utilizzare tecniche di data mining (DM). Infatti, sono presenti alcuni campi in cui l'utilizzo di strumenti di KDD è ormai comune, esistono altri ambiti in cui sono finora state realizzate poche applicazioni reali ma la cui relazione con il data mining sta diventando più stretta, ed aree in cui si vedono pochi se non nessun collegamento.

Viene ora presentato un elenco dei campi in cui l'utilizzo di tecniche di DM è più frequente:

- *Analisi e gestione di mercato*: partendo dai dati ottenuti dalle transazioni con le carte di credito, con le carte fedeltà (ad esempio quelle dei supermercati), o dai reclami dei clienti è possibile effettuare diversi tipi di analisi. Innanzi tutto è realizzabile il *target marketing*, cioè la ricerca di classi di consumatori con le stesse caratteristiche; un altro tipo di analisi è la cosiddetta *cross-market analysis*, la quale individua associazioni fra vari prodotti, cercando di effettuare delle previsioni basate sulle relazioni trovate. Un terzo tipo di analisi è la ricerca dei

requisiti dei vari clienti: in base al tipo di cliente considerato si cercano i prodotti che potrebbero interessargli/le maggiormente;

- *Analisi aziendale e gestione dei rischi*: in campo finanziario è possibile effettuare l'analisi e le previsioni sui flussi di cassa, oltre a studi sulle serie temporali (*trend analysis*). Anche per quanto riguarda la pianificazione delle risorse (*resource planning*) le tecniche di data mining possono essere molto utili per riassumere e comparare le risorse. Inoltre l'analisi competitiva può trarre indubbi vantaggi monitorando i concorrenti per poter agire di conseguenza, raggruppando i clienti in classi allo scopo di stabilire prezzi in base alla categoria di riferimento;
- *Rilevamento e gestione delle frodi (fraud detection)*: il rilevamento delle frodi relative all'utilizzo della carta di credito è possibile e può dare grandi benefici monetari. Lo stesso procedimento può essere effettuato per gli acquisti on-line tramite carta di credito; inoltre, in questo campo, è possibile offrire servizi di gestione del rischio per i siti di e-commerce. Sempre per quanto riguarda il web, tramite appositi meccanismi di *logging* e tecniche di data mining, le intrusioni in reti di computer possono essere rilevate. Anche in campo assicurativo e medico il data mining può essere davvero utile, visto che si possono individuare frodi nei confronti delle compagnie di assicurazione e, in campo medico, è possibile scoprire diagnosi errate che comportano costi per il sistema sanitario di uno stato;
- *Analisi del web (web analysis)*: sono ormai comuni analisi dei visitatori di particolari siti web, basate sulla ricerca di sequenze (*patterns*) all'interno dei log web. Sempre a questo proposito, è possibile effettuare una classificazione dei visitatori e degli acquirenti di siti di e-commerce ed analizzare il comportamento degli acquirenti on-line. Sono utilizzate spesso tecniche di data mining per inviare e-mail pubblicitarie basate sugli interessi dei destinatari in modo automatico.

1.5.1.3 Estensione dell'applicabilità del data mining oltre ai database

Da quanto proposto finora, è evidenziabile lo stretto legame delle tecniche di data mining con l'utilizzo dei database. Questo perché l'applicazione classica di questa disciplina ne prevede la presenza, visto che essi riescono ad organizzare le informazioni attraverso una struttura omogenea ed offrono diversi strumenti che permettono la loro analisi (si pensi all'efficacia delle "semplici" query). Non è però esclusa la possibilità di riferire ciò che si è finora detto e quello che viene esposto più avanti ad una qualsiasi *collezione di informazioni*. Per riuscire nell'estensione del concetto di data mining ad informazioni non connesse con i database si può considerare la definizione del termine, in cui una delle parole chiave era *Database di grandi dimensioni*. Sostituendo questo concetto con quello di *collezione di informazioni* si ottiene una definizione equivalente estesa anche ad

informazioni non strutturate all'interno di database: quello che manca, dopo la sostituzione, è un'organizzazione efficace delle stesse, problema comunque affrontabile positivamente. E' sufficiente effettuare delle operazioni ulteriori atte all'organizzazione delle informazioni di modo che siano assimilabili ad un database. Tutto ciò che è esposto in seguito viene analizzato non solamente in relazione ai comuni database ma a qualsiasi gruppo di informazioni.

1.5.1.4 Il processo di estrazione di conoscenza

Per capire al meglio la collocazione precisa delle applicazioni che sfruttano il data mining, si è deciso di visualizzare ora un quadro generale (sotto forma di diagramma) che rappresenti il flusso dei dati, a partire dalle collezioni di informazioni originali (che possono essere appunto più di una), fino a raggiungere l'obiettivo prefissato, ovvero la conoscenza (*knowledge*). Questo processo è universalmente noto con il nome di *Knowledge Discovery in Databases Process*, ovvero processo di scoperta della conoscenza nei database, e la figura 1.2 (da [19]), che lo rappresenta in formato grafico, è utilizzata comunemente.

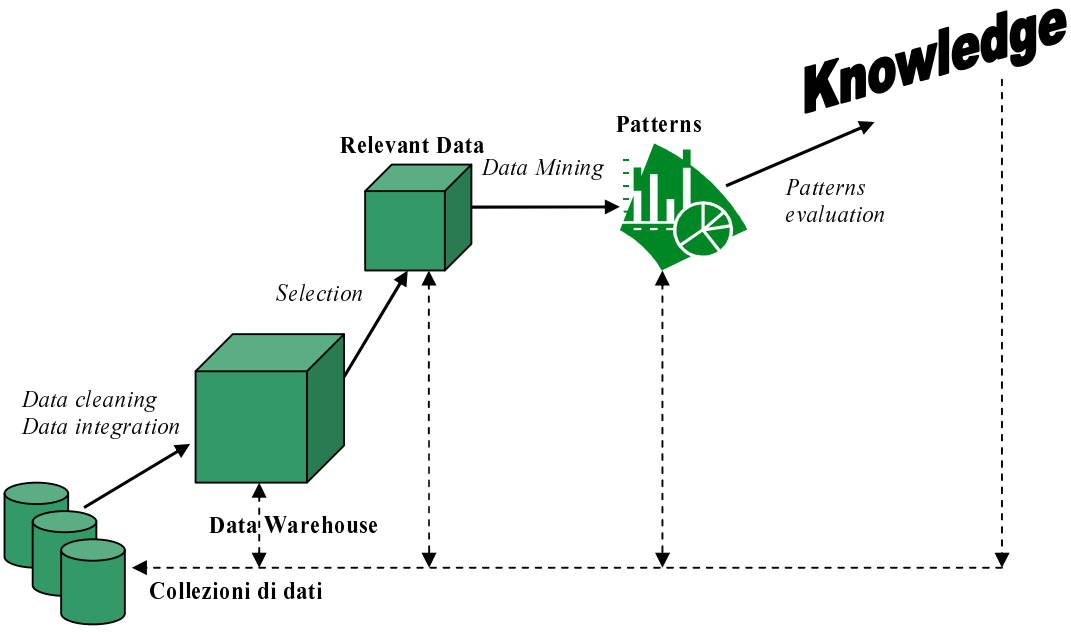

Figura 1.2: Processo di KDD

Nello schema proposto in figura 1.2 le azioni sono evidenziate in *corsivo*, mentre ogni stato dei dati è rappresentato da una scritta in **grassetto**. E' bene però analizzare nel

dettaglio il significato dei vari termini presentati, e per fare questo il modo migliore è spiegare il funzionamento del processo di KDD.

Il primo passo da eseguire è capire su quali collezioni di dati si vuole lavorare: per fare questo si selezionano quelle ritenute interessanti nell'ambito di riferimento. Allo scopo di avere una gestione migliore dei dati contenuti all'interno delle collezioni individuate è necessaria l'esecuzione di due tipi di operazioni, chiamate rispettivamente *data cleaning* e *data integration.*

Il *data cleaning*, detto anche *data scrubbing*, è il processo di rimozione di informazioni incorrette, incomplete, impropriamente formattate o duplicate [37]. E' infatti indispensabile rimuovere tutte quelle informazioni che potrebbero trarre in inganno le fasi successive, le quali potrebbero considerare rilevanti degli elementi errati e trarre di conseguenza delle false conclusioni.

Per *data integration* si intende l'attività che mira all'integrazione delle varie collezioni all'interno di una singola struttura, la quale deve essere coerente e permettere la gestione efficiente di tutte queste banche dati. L'input del data integration sono quindi uno o più collezioni di informazioni, mentre l'output è la restituzione di una *Data Warehouse*.

Questi primi passi (*data cleaning* e *data integration*) sono quelli che richiedono maggiori modifiche in relazione all'utilizzo di collezioni di informazioni al posto di database. Serve infatti qualche passo in più per raggiungere la *Data Warehouse*, dato che non si parte (come accade con i database) da informazioni già organizzate e strutturate, ed è richiesto un processo di integrazione più complesso che nel caso dell'utilizzo di banche dati.

Il passo successivo è una scrematura della Data Warehouse appena ottenuta, al fine di mantenere solamente le informazioni che si ritengono utili per l'analisi che si vuole effettuare. Questo è necessario anche per risparmiare tempo nell'esecuzione delle tecniche di data mining (riducendo la quantità di elementi), ma soprattutto per evitare che delle informazioni non rilevanti ai fini dell'analisi che si effettua "ingannino" i tool utilizzati e li inducano a trarre conclusioni errate. I dati rimasti sono solo quelli effettivamente rilevanti (*relevant data*).

A questo punto entrano in gioco le funzioni di data mining. Infatti siamo in presenza dei soli dati che realmente interessano, e saranno questi che dovranno essere esaminati. La prima scelta da fare sta nel decidere che tipo di funzioni utilizzare; un elenco di quelle disponibili è il seguente:

1. *Class Description*: detta anche *Summarization*, fornisce una rappresentazione schematica e concisa dei dati presenti. Essa si basa su una distinzione fra i dati presenti sulla base dei loro attributi;

2. *Association*: la ricerca di *regole associative* permette la creazione di regole del tipo Antecedente ➔ Conseguente, a partire dai dati rilevanti che si hanno a disposizione;
3. *Classification*: vengono prima di tutto esaminati dei dati che servono per istruire il software di classificazione (essi sono chiamati *training data*), allo scopo di creare delle classi. Attraverso un albero di decisione si riescono poi a classificare i dati reali, decidendo a quale classe dovranno appartenere;
4. *Prediction*: le funzioni di predizione hanno lo scopo di cercare di prevedere quali valori dovrebbero assumere i record mancanti, o di riempire dei campi il cui valore non è stato assegnato;
5. *Clustering*: quest'attività è simile alla classificazione, con la differenza che in questo caso le classi non sono stabilite a priori, bensì dinamicamente, sulla base degli attributi presenti nei vari elementi;
6. *Time series analysis*: l'analisi di serie temporali viene eseguita su dati che hanno uno o più attributi di tempo che li identificano. Vengono cercate delle sequenze o delle associazioni interessanti fra i dati.

Dopo aver scelto quale fra queste tecniche utilizzare è necessario compiere una scelta ulteriore: decidere quale algoritmo adottare. Tutte queste tecniche, infatti, propongono una serie di algoritmi più o meno standard che possono essere utilizzati per analizzare i dati. E' molto frequente che sorga la necessità di apporre modifiche (anche sostanziali) agli algoritmi standard per adattarli alla situazione che si vuole esaminare, o può anche essere richiesto lo sviluppo di algoritmi ad-hoc.

A questo punto si sono ottenuti dei risultati, spesso esprimibili in *pattern di interesse*. Il processo di KDD non è comunque esaurito, visto che è necessario esaminare i pattern che sono stati scoperti. In particolare le operazioni che si possono effettuare sono la visualizzazione degli stessi, la loro trasformazione in altre forme per permetterne un esame migliore, e la rimozione dei pattern ridondanti. Una volta eseguite le operazioni necessarie si può utilizzare la *conoscenza* ottenuta.

Tutti i passi (*step*) che vengono eseguiti nel processo di KDD prevedono la possibilità di tornare indietro (in figura 1.2 le frecce tratteggiate). Infatti, può accadere che in una delle azioni eseguite vengano impostati dei parametri che eliminano dei dati interessanti, o che non conducono a risultati significativi; in tal caso si può tornare al passo precedente ed eseguire nuovamente l'azione modificandone i parametri.

1.5.1.5 Tecniche di data mining

In questo paragrafo saranno esaminate più nel dettaglio le singole tecniche di data mining, al fine di poter individuare con chiarezza quale sia la quella più idonea per la risoluzione del problema che si affronta. Per avere informazioni ulteriori si vedano [18] e [22].

1. *Class Description*

   La descrizione delle classi fornisce un *riassunto conciso e succinto di un insieme di dati e li distingue fra loro* [18]. Il riassunto di un insieme di dati è detto *class characterization*, mentre il confronto fra vari insiemi di dati è chiamato *comparison / discrimination*.

   Un esempio di utilizzo di tecniche di data description è il confronto fra le vendite europee e quelle asiatiche di una multinazionale, che può identificare i fattori più importanti che distinguono le due classi (cioè i due mercati), e presentare una panoramica sintetica dei risultati ottenuti.

2. *Association*

   Per associazione si intende *la scoperta di relazioni di associazioni (association relationships) o correlazioni fra un insieme di elementi* [18].

   Questo tipo di tecnica [43] è molto utilizzata, e l'esempio di applicazione più noto è la *basket data analysis*, un'istanza della quale è rappresentata dalle relazioni fra i prodotti acquistati in un supermercato da uno stesso cliente (cioè da ciò che mette nel carrello della spesa).

   Le regole che si individuano sono nella forma **Antecedente → Conseguente**, in cui sia antecedente che conseguente possono essere formati da uno o più atomi. Nel caso della basket data analysis, ad esempio, si può individuare la seguente relazione:

   Spaghetti AND Peperoncino → Acqua minerale

   Questa associazione, detta anche *regola associativa* (*association rule*), indica che la presenza contemporanea nel carrello della spesa di spaghetti e di peperoncino implica anche la presenza di acqua minerale. L'antecedente è formato da due atomi, mentre il conseguente da uno.

   Al fine di identificare le regole più importanti e significative, sono presenti due misure chiave, che permettono di definire delle soglie minime, facendo in modo che le regole che stanno al di sotto di tali soglie (*threshold*) non vengano considerate: tali misure sono il *supporto* (*support*) e la *confidenza* (*confidence*) [34]. Definiamole:

Sia data la seguente regola X AND Y → Z

Il *supporto* S è definito come la probabilità che una transazione contenga X, Y e Z sul totale delle transazioni.

La *confidenza* C è la probabilità condizionale che una transazione che abbia X e Y contenga anche Z. Se X ed Y fossero presenti in 10 transazioni, ed in queste Z fosse presente solamente 2 volte, la confidenza C varrebbe in questo caso il 20%.

L'algoritmo più utilizzato per la scoperta di regole associative si chiama *Apriori* [20]. Esso viene esaminato nel dettaglio più avanti.

3. *Classification*

Quando si parla di classificazione si intende la *catalogazione di un fenomeno in una classe predefinita* [1]. Questo significa che è necessario avere un qualche tool contenente un algoritmo che effettui tale catalogazione: in particolare l'algoritmo è detto genericamente *classificatore* (*classifier*). La sua realizzazione è automatica, cioè il classificatore è generato in maniera automatica esaminando un certo numero di dati di prova (i cosiddetti *training data*). Ci si può chiedere come sia possibile la generazione automatica del classificatore: la risposta sta nella struttura dell'algoritmo, che è in sostanza un *albero di decisione*. Ai nodi dell'albero di decisione, che permettono la classificazione degli elementi presenti nella banca dati, corrispondono dei predicati corrispondenti agli attributi degli oggetti esaminati. E' possibile decidere di considerare anche solamente gli attributi rilevanti, nei casi in cui il numero totale di attributi sia troppo elevato. Un esempio può essere utile per chiarire il concetto di decision tree.

Si consideri una tabella di un DBMS che identifichi i clienti di un consulente finanziario. Gli attributi dei clienti possono essere il reddito, l'età, il sesso, la professione, lo stato coniugale. Al fine di proporre un investimento il consulente potrebbe basarsi su queste informazioni, per capire quale offerta sia la migliore fra quelle disponibili (basso rischio, medio rischio, alto rischio). Potrà quindi costruire un albero basandosi sugli attributi che considera importanti, ad esempio il reddito, l'età, lo stato coniugale:

Investimento(Reddito, Età, Stato Coniugale).

In figura 1.3 è mostrata la rappresentazione schematica del caso appena presentato, raffigurando l'algoritmo di classificazione tramite albero di decisione.

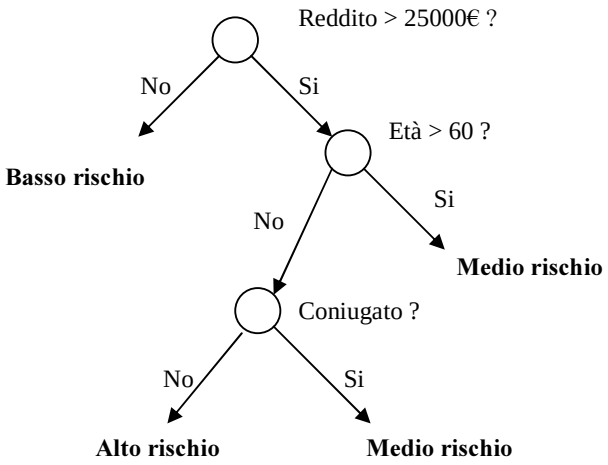

Figura 1.3: Esempio di Decision Tree

Questo esempio è chiaramente molto semplificato, ma dovrebbe essere sufficiente per illustrare il funzionamento di un tipico albero di decisione (*decision tree*). Per la costruzione di algoritmi di classificazione si utilizzano spesso tecniche di *machine learning*, che si basano sull'apprendimento dai dati esistenti, e che dovrebbero migliorare le proprie prestazioni con l'uso (si parla infatti di *sistemi esperti*).

4. *Prediction*

   Questo tipo di funzione di data mining ha lo scopo di *predirre i possibili valori di alcuni dati mancanti o la distribuzione del valore di certi attributi in un insieme di oggetti* [18]. Per funzionare correttamente è necessaria la ricerca degli attributi più rilevanti, attraverso l'utilizzo di metodologie statistiche. Un tipico esempio può essere il prezzo da fissare per un determinato prodotto: una soluzione semplificata può consistere nel basarsi sulla media dei prezzi proposti nei negozi concorrenti. Strumenti molto utilizzati per la predizione sono tecniche di *regressione*, *alberi di decisione*, *reti neurali* ed *algoritmi genetici*.

5. *Clustering*

   Il clustering è definibile come *l'individuazione di cluster presenti nei dati* [18], dove per *cluster* si intende un insieme di oggetti simili fra loro. Da questa prima definizione potrebbe sembrare che parlare di *classification* o di *clustering* sia esattamente la stessa cosa. In effetti, questi due procedimenti sono molto simili, dato che entrambi mirano a suddividere gli oggetti in classi; la differenza sta nel fatto che il *clustering* non definisce preventivamente quali siano i *clusters*, ovvero i gruppi di oggetti, ma questi vengono individuati dinamicamente durante

l'esecuzione dell'algoritmo prescelto. Per questa somiglianza il clustering è anche definito come *unsupervised classification* (*classificazione senza supervisione*) [19]. Una buona metodologia di clustering deve fare in modo che sia presente un basso grado di *inter-cluster similarity*, cioè due elementi appartenenti a diversi gruppi siano il più possibile dissimili; allo stesso tempo è necessario che vi sia un alto coefficiente di *intra-cluster similarity*, cioè due elementi appartenenti al medesimo cluster siano molto simili.

La *cluster analysis* è utilizzata sia in tool che precedono una successiva analisi (*preprocessing step*) che in software stand alone che hanno lo scopo di raggruppare i dati in classi omogenee.

Gli approcci più comuni per effettuare il clustering sono gli algoritmi di partizionamento (*partitioning algorithms*), i quali costruiscono varie partizioni e le valutano secondo alcuni criteri, gli algoritmi gerarchici (*hierarchy algorithms*) che decompongono gerarchicamente i dati, tecniche basate sulla densità (*density based*), metodi a griglia (*grid based*) utilizzanti una struttura con diversi livelli di granularità, i metodi basati su modello (*model based*), i quali ipotizzano un modello per ogni cluster ed utilizzano dei modelli matematici.

6. *Time-series analysis*

   Lo scopo dell'analisi delle serie temporali (*time series analysis*) è l'esame di grandi insiemi di dati temporali cercando comportamenti regolari e caratteristiche interessanti, fra le quali la ricerca di sequenze simili o sottosequenze, trend e deviazioni [18].

   Il tipico esempio pratico di utilizzo di tecniche di time-series analysis consiste nell'effettuare una scansione di dati relativi alle quotazioni di azioni o di titoli quotati in borsa, per valutarne l'andamento nel corso del tempo e probabili cicli che permettano di prevedere il futuro andamento della quotazione.

Chiaramente non è possibile elencare tutte le tecniche di data mining esistenti in questa tesi; si ritiene comunque sufficiente l'elenco delle tecniche appena riportato.

1.5.2 Collaborative Filtering

Dopo aver definito cos'è il data mining ed averne discusso le principali tecniche, si passa ora all'esame di una differente metodologia, la quale, nonostante sia trattata separatamente, utilizza delle tecniche spesso simili (se non le stesse) di alcune funzioni di data mining. Questa tecnica si chiama *collaborative filtering* (filtraggio collaborativo).

Il collaborative filtering cerca di aiutare l'utente nella ricerca delle informazioni: infatti un altro nome per indicare tale tecnica è *recommender systems*, ovvero sistemi di

raccomandazione. Questo tipo di sistemi sfruttano, per raggiungere il loro scopo, le opinioni degli utenti che fanno uso del servizio del quale si vuol facilitare l'utilizzo. Ciò che contraddistingue il collaborative filtering da degli approcci più semplici sta nel considerare, per aiutare l'utente, non solo le proprie preferenze, ma anche quelle espresse da utenti considerati simili. Un ulteriore sinonimo per collaborative filtering è *social filtering*.

L'approccio di questa metodologia è molto simile a quello della società moderna, in cui un utente (ovvero una persona) si basa sui consigli ed i gusti degli utenti simili (i famigliari e gli amici) per compiere le proprie scelte.

Esistono diversi esempi di sistemi che sfruttano il collaborative filtering: esso viene utilizzato, attraverso algoritmi più o meno sofisticati, in molti siti che effettuano vendite on-line. Fra i siti web più conosciuti si possono citare Amazon, CDNow, eBay, Levis, MovieFinder, Reel. Per un approfondimento sui sistemi commerciali si vedano [33] [36].

Esistono due tipologie differenti di collaborative filtering: *passivo* ed *attivo* [14].

- *CF passivo*: si intende un sistema in cui gli utenti non interagiscono direttamente fra loro: quello che hanno da fare è solo dare dei giudizi sugli oggetti che esaminano, le loro preferenze vengono memorizzate in un database ed utilizzate dal sistema per facilitare altri utenti nella ricerca di oggetti di loro gradimento;
- *CF attivo*: in questo caso si parla di un tipo di sistema molto più simile alla realtà della società umana, in cui vi è interazione fra gli utenti, ovvero chi ha un opinione su un oggetto va a riferirla ad altri soggetti.

Un altro modo di classificare gli algoritmi di collaborative filtering è secondo il modo in cui le informazioni sono ottenute dagli utenti, ovvero se in modo esplicito o implicito [25] :

- *Explicit CF*: ogni volta che l'utente esamina un determinato elemento, gli viene chiesto di darne una valutazione, la quale verrà memorizzata nel database. Questo tipo di filtraggio è detto esplicito, perché richiede la collaborazione diretta dell'utente;
- *Implicit CF*: in questo caso l'utente può anche non essere a conoscenza del fatto di essere monitorato. Infatti in base alle azioni eseguite si cerca di fare una valutazione del comportamento dell'utente, senza che nulla venga richiesto esplicitamente all'utente stesso [27].

Da questa breve introduzione si può già fare una valutazione relativa all'utilità del collaborative filtering rispetto al data mining. Mentre il data mining effettua una serie di operazioni atte ad estrarre delle relazioni ed informazioni nascoste, per poi utilizzare ciò

che si è appreso in seguito (magari modificando o creando un software di accesso ai dati per far sì che queste associazioni vengano proposte ed eseguite dall'utente), il collaborative filtering è una tecnica che cerca di aiutare direttamente l'utilizzatore dei dati, dandogli dei consigli sugli elementi da visualizzare.

Un altro modo di classificare il collaborative filtering riguarda il tipo di algoritmo che viene utilizzato per espletare il processo di filtraggio collaborativo. Esistono sostanzialmente due classi di algoritmi fra i quali si può scegliere:

- *Memory based CF*: sono mantenute in un database le informazioni relative a tutte le preferenze espresse dagli utenti; ogni volta che si ritiene necessario aiutare l'utente si esaminano tutti questi dati;
- *Model based CF*: prima di consigliare l'utente viene creato un modello che tiene conto degli utenti, degli oggetti e delle preferenze. I suggerimenti vengono effettuati in base al modello;

Visto che la classificazione più comune per i recommender systems è proprio questa, si esamineranno ora in dettaglio gli algoritmi e le tecniche sia relative al collaborative filtering basato sul modello che basato sulla memoria.

1.5.2.1 Memory based

Come già accennato, questo tipo di algoritmi utilizza, allo scopo di predirre le preferenze di un certo utente, un insieme di pesi ottenuti dal database che registra tutte le preferenze degli utenti .

Esistono due sottocategorie di algoritmi memory-based: *user-based* e *item-based.*

User-based algorithms

In questo caso si agisce sul database delle preferenze, ma si considera anche un fattore relativo allo stesso utente in relazione alle precedenti preferenze espresse su altri oggetti. Gli algoritmi di questo tipo si basano sulle preferenze degli utenti più vicini (simili) a quello corrente: per questo motivo sono anche detti *neighbourhood algorithms* [32]. Il database è formato da un insieme di records, ognuno dei quali esprime la preferenza di un utente *i* in relazione ad un oggetto *j*: chiameremo questa preferenza *voto* e la indicheremo con $v_{i,j}$.

Sia I_i l'insieme degli oggetti votati dall'utente *i*; possiamo ora definire il voto medio per tale utente:

$$\bar{v}_i = \frac{1}{|I_i|}\sum_{j \in I_i} v_{i,j} \qquad (1.1)$$

Si assume quindi che il voto da predirre $p_{a,j}$, relativo all'utente corrente *a* ed all'oggetto *j*, *sia* una somma pesata dei voti espressi dagli altri utenti [13]:

$$p_{a,j} = \bar{v}_a + k\sum_{i=1}^{n} w(a,i)(v_{i,j} - \bar{v}_i) \qquad \textbf{(1.2)}$$

- *n* rappresenta il numero di utenti nel database con pesi diversi da zero;
- *w(a,i)* indica la distanza, correlazione o similarità fra l'utente *i* e l'utente corrente *a*. Questo è il fattore più importante, dato che identifica la similarità fra l'utente corrente e l'utente esaminato, e riesce quindi a dare un peso maggiore o minore alla preferenza espressa da tale utente;
- *k* è un fattore di normalizzazione per fare in modo che i valori assoluti dei pesi vengano relazionati con l'unità;

E' quindi osservabile come la formula generale 1.2 tenga in considerazione sia la media dei voti relativi all'utente corrente, che un valore ottenuto dai voti degli altri utenti in relazione con l'oggetto del quale si vuole predirre la preferenza per l'utente corrente.

La formula 1.2 è modificabile a seconda del campo di utilizzo, ed in particolare i diversi tipi di valutazione per *w(a,i)* permettono di utilizzare diverse tecniche che possono portare a risultati differenti.

Per determinare la similarità fra due utenti sono utilizzabili diversi metodi, fra cui i più importanti sono certamente:

- *coefficiente di correlazione di Pearson* fra l'utente *a* e l'utente *i* $w_{a,i}$ [32] [25]

$$w_{a,i} = \frac{\sum_{j=1}^{n}(v_{a,j} - \bar{v}_a)(v_{i,j} - \bar{v}_i)}{\sigma_a * \sigma_i} \qquad \textbf{(1.3)}$$

  con σ_a che indica la varianza dei voti di *a*. Questo coefficiente è, almeno allo stato attuale, quello che fornisce i risultati migliori (almeno secondo [21]); per questo motivo molte applicazioni reali ne fanno utilizzo;

- *coefficiente di correlazione di Spearman* [25]*:* anziché utilizzare i voti espressi per calcolare la similarità, li raggruppa dapprima in classi chiamate *rank*. Il coefficiente di correlazione di Spearman fra *a* ed *i* è:

$$w_{a,i} = \frac{\sum_{j=1}^{n}(rank_{a,j} - \overline{rank}_a)(rank_{i,j} - \overline{rank}_i)}{\sigma_a * \sigma_i} \qquad \textbf{(1.4)}$$

- *coseno dell'angolo tra vettori* [25]

$$w_{a,i} = \sum_{j=1}^{n} \frac{v_{a,j}}{\sqrt{\sum_{k=1}^{n} v_{a,k}^2}} * \frac{v_{i,j}}{\sqrt{\sum_{k=1}^{n} v_{i,k}^2}} \qquad \textbf{(1.5)}$$

  In questo caso i denominatori sono utilizzati per fare in modo che la similarità non venga definita in base al numero di oggetti votati da un singolo utente;

- *differenza quadratica media*: anche questa misura prettamente legata a concetti di probabilità e statistica, è utilizzata per verificare la similarità fra due utenti. La similarità è calcolata come:

$$w_{a,i} = \frac{\sum_{j=1}^{n} (v_{a,j} - v_{i,j})^2}{n} \qquad \textbf{(1.6)}$$

  Questa misura è qualitativamente inferiore a quelle precedenti, anche se in casi particolari può essere sufficientemente efficace.

Esistono però due problemi generici relativi agli algoritmi *user-based.* Questi sono la *sparsità (sparsity)* e la *scalabilità (scalability)* dei dati.

Il database contenente i voti degli utenti potrebbe essere molto *sparso*, cioè gli utenti potrebbero aver espresso un numero troppo piccolo di voti relativamente ad una quantità enorme di oggetti. In questo caso gli utenti saranno difficilmente simili fra loro e le previsioni sulle preferenze poco credibili, visto il basso numero di voti per ciascun oggetto.

Il secondo problema è la *scalabilità*: potrebbero esserci troppi dati, cosicché un approccio basato sull'utente potrebbe essere troppo difficile da gestire per il motore software che ha il compito di scorrere il database: infatti è caratteristica dei sistemi user based la necessità di esaminare tutto il database contenente i voti.

Item-based algorithms

Questo tipo di approccio ha come idea basilare il fatto che un utente si interesse di oggetti simili a quelli che in passato ha già votato positivamente e che, all'opposto, cercherà di evitare oggetti che in precedenza non sono stati graditi (verso i quali avrà dato una preferenza negativa).

Gli algoritmi basati sugli oggetti dovrebbero riuscire a superare il problema della scalabilità, non andando ad esaminare le preferenze di tutti gli utenti. Infatti vengono considerati solo gli elementi in precedenza votati dall'utente, e si cerca di stabilire un grado di similarità fra essi e gli oggetti che ha ancora da visualizzare.

Sono presenti due passi distinti che devono essere compiuti sequenzialmente: il calcolo della similarità fra gli oggetti (*item similarity computation*) e la predizione (*prediction computation*).

- *Calcolo della similarità fra oggetti*: Questo è decisamente il passo più importante e difficile per poter aiutare l'utente nella scelta di un oggetto da visualizzare. Infatti, se tale computazione non avvenisse in modo corretto, il passo successivo, ovvero la predizione, non produrrebbe risultati effettivamente significativi per l'utente. Vengono spesso utilizzate le stesse tecniche del modello user-based e del data mining, ovviamente adattate allo scopo presente. Le tecniche più frequenti sono la *cosine-based similarity*, la *correlation-based similarity* e la *adjusted cosine similarity*.

  Prima di esaminarle si consideri la matrice che contiene i voti di dimensioni *n x m*, in cui *n* indica il numero di oggetti, mentre *m* rappresenta il numero di utenti.

  Cosine-based similarity: due oggetti sono considerati come due vettori di dimensione *m* (nei vettori sono presenti i voti di ciascun utente), e la similarità fra i due vettori è calcolata determinandone il coseno dell'angolo che tali vettori formano [35]:

$$sim(i,j) = \cos(\vec{i},\vec{j}) = \frac{\vec{i} \cdot \vec{j}}{\left\|\vec{i}\right\|_2 * \left\|\vec{j}\right\|_2} \qquad \textbf{(1.7)}$$

  Correlation-based similarity: la similarità fra gli oggetti j e k è calcolato trovando il coefficiente di correlazione di Pearson fra essi. La formula è quella già proposta parlando della user-based similarity e per questo non è riportata.

  Adjusted cosine similarity: è un metodo che unisce i due metodi appena riportatati; risolve il problema del primo di non tener in considerazione la differenza nella scala di gradimento fra un utente e l'altro. Per fare questo si sottrae la media di voto di ciascun utente quando si analizza un voto di quell'utente, determinando quindi di quanto sia sopra o sotto la media [35]:

$$sim(i,j) = \frac{\sum_{k=1}^{n}(v_{k,i} - \bar{v}_k)(v_{k,j} - \bar{v}_k)}{\sqrt{\sum_{k=1}^{n}(v_{k,i} - \bar{v}_k)^2}\sqrt{\sum_{k=1}^{n}(v_{k,j} - \bar{v}_k)^2}} \qquad \textbf{(1.8)}$$

- *Predizione*: Le tecniche più utilizzate per effettuare questo passo sono due: la somma pesata (*weighted sum*) e la regressione (*regression*).

La *somma pesata* calcola la predizione per un oggetto *j* ed un utente *a* calcolando la somma dei voti dati dall'utente agli oggetti simili a *j*. Ogni votazione è pesata in base alla similarità $s_{j,similarItems[k]}$ fra gli oggetti *j* ed il *k-esimo* fra quelli simili (*similarItems*):

$$P_{i,j} = \frac{\sum_{similarItems} (s_{j,similarItems[k]} * v_{i,similarItems[k]})}{\sum_{similarItems} | s_{j,similarItems[k]} |} \quad \textbf{(1.9)}$$

La *regressione* è molto simile al metodo precedente: ciò che le differenzia è che, anziché utilizzare direttamente i voti degli oggetti simili, usa un'approssimazione dei voti basati su un *modello di regressione*. La formula rimane quella precedente ma, anziché utilizzare direttamente il voto $v_{i,similarItems[k]}$ se ne approssima il valore calcolando il $v'_{i,similarItems[k]}$ attraverso un modello di regressione lineare. Dati R_j e $R_{similarItems[k]}$ i vettori di voto per gli oggetti *j* e *similarItems[k]*, siano α e β i coefficienti per il modello e sia ε l'errore del modello di regressione, indichiamo la formula:

$$\bar{v}'_{similarItems[k]} = \alpha \bar{v}_j + \beta + \varepsilon \quad \textbf{(1.10)}$$

1.5.2.2 Model based

Questo tipo di collaborative filtering crea un modello utilizzato per generare raccomandazioni basandosi su un insieme di dati di prova (*training data*). Una volta che è finita la fase di generazione del modello è possibile utilizzarlo per raccomandare l'utente sull'oggetto da visualizzare.

Gli algoritmi appartenenti a questa categoria sfruttano un approccio di tipo probabilistico considerando il collaborative filtering come la computazione del valore atteso (*expected value*) di una predizione per l'utente, dati i suoi voti relativi agli altri oggetti. La formula generale per la predizione può essere espressa in questo modo [13]:

$$P_{a,j} = E(v_{a,j}) = \sum_{i=0}^{m} \Pr(v_{a,j} = i \mid v_{a,k}, k \varepsilon I_a) i \quad \textbf{(1.11)}$$

L'espressione di probabilità è la probabilità che l'utente dia un particolare voto fra 0 e *m* per l'oggetto *j*, dati i voti precedentemente osservati ($v_{a,k}$) moltiplicati per il valore particolare *i*. La somma di queste probabilità dà il valore atteso. Questa è precisamente la definizione di valore atteso in probabilità.

Le tecniche applicate a questa formula generale sono solitamente le reti Bayesiane (*Bayesian network*), il *clustering* e gli approcci basati su regole (*rule based*).

- *Clustering*: questa tecnica è stata esaminata anche per quanto riguarda il data mining (e questo mostra la stretta relazione esistente fra collaborative filtering e data mining); qui viene utilizzata raggruppando gli utenti in varie classi *C* in base alle loro caratteristiche. Viene quindi calcolata la probabilità che un particolare utente appartenga ad una certa classe *C*, dato l'insieme dei suoi voti. Il *classificatore bayesiano* è un modello probabilistico molto utilizzato e la formula [13] è la seguente:

$$\Pr(C = c, v_1, ..., v_n) = \Pr(C = c)\prod_{1}^{n} \Pr(v_i \mid C = c) \qquad \textbf{(1.12)}$$

- *Reti bayesiane*: è un'alternativa all'utilizzo del clustering. Una rete bayesiana è formata da nodi, ognuno dei quali rappresenta un oggetto nel dominio. Gli stati di ciascun nodo corrispondono ai possibili voti di ogni elemento. Esiste anche uno stato particolare che definisce "nessun voto", per i domini in cui non esiste una non esiste alcun altro modo per esprimere l'assenza di dati.

  Viene quindi eseguito l'algoritmo di apprendimento (*learning algorithm*), il quale crea una rete a partire dai nodi. Essa è caratterizzata dal fatto che ogni nodo (elemento) ha un insieme di elementi padre che sono i migliori predittori per i suoi voti. Ogni tabella che rappresenti la probabilità condizionata può essere codificata con un albero di decisione per quel nodo. L'albero presentato in figura 1.4 è un esempio tipico tratto da [13], relativo alla visione di serie televisive: è l'albero di decisione legato alla probabilità di visione di "Melrose Place" in relazione con "Beverly Hills, 90201" e "Friends":

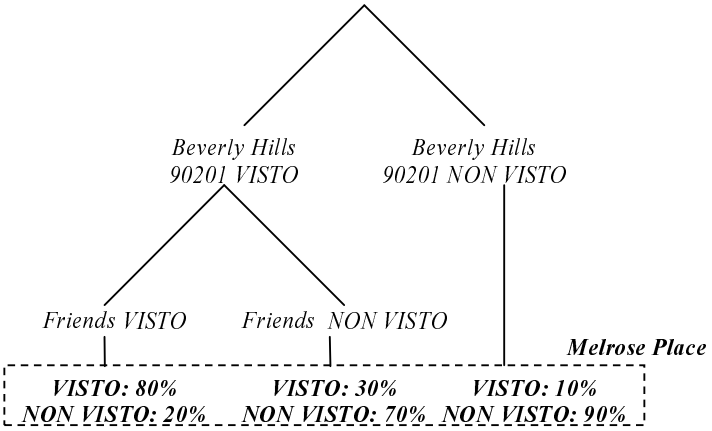

Figura 1.4: Reti Bayesiane

- *Rule-based*: questo tipo di approccio utilizza algoritmi che permettono la scoperta di regole di associazione al fine di individuare relazioni fra elementi già valutati, quindi genera una raccomandazione sulla base della forza dell'associazione individuata. Anche le regole associative sono state già esaminate nell'ambito del data mining e non viene ripetuta la descrizione.

1.5.3 Implicit Culture

Concludiamo questa panoramica sulle tecniche utilizzabili per risolvere i problemi derivanti dall'overload di informazioni introducendo un terzo concetto: quello di *cultura implicita.*

L'idea che sta alla base del *collaborative filtering* ha una valenza maggiore rispetto al semplice filtraggio di informazioni. A questo proposito nasce l'idea di creare una nuova metodologia, che riesca a sfruttare questa potenzialità: tale tecnica è stata chiamata *Cultura Implicita* (*Implicit Culture*); essa tende a fare in modo che un agente che opera un ambiente senza conoscerlo bene si comporti in modo ottimale, al contrario di ciò che succede normalmente (opera in modo non ottimale). Per definire con esattezza il concetto si ritiene utile riportare un paragrafo da [2]:

"Quando un agente inizia ad interagire con un ambiente senza sufficienti conoscenze o abilità, il suo comportamento sarà lontano dall'ottimale. I problemi che tale agente deve affrontare sono ancora più complessi se altri agenti sono attivi nello stesso ambiente. Essi avranno probabilmente più conoscenze e saranno più esperti. Inoltre, potrebbero non voler condividere le loro conoscenze e qualche volta non essere in grado di rappresentarle e di comunicarle".

Ovvero il nuovo agente si può trovare in una situazione in cui vorrebbe sfruttare le conoscenze degli altri agenti attivi, ma in cui non è in grado di farlo, per i motivi suddetti. Un esempio comune è un utente che ricerca nel web informazioni relative ad un argomento del quale ha poche conoscenze: sarà difficile per lui inserire le parole chiave (che utenti più esperti hanno già acquisito) e trovare le risorse rilevanti.

Allo scopo di migliorare il proprio comportamento, il nuovo agente dovrebbe agire in modo conforme alla cultura del gruppo degli agenti che conoscono l'ambiente. Questo è fattibile, in linea generale, migliorando le capacità degli agenti in termini di comunicazione, conoscenza ed apprendimento. Una soluzione può essere che il nuovo agente *chieda a qualcuno*; non è certo uno scenario facile da realizzarsi, dato che esistono i problemi relativi a *cosa chiedere* (conoscenza del problema), *come chiederlo* (necessità di un linguaggio), *a chi chiederlo*. Una seconda alternativa è rappresentare la conoscenza rilevante e fornirla all'agente (con qualche tramite); potrebbe essere una soluzione efficace, ma solo nel caso in cui la conoscenza sia oggettiva e piuttosto statica. La

soluzione migliore è fornire all'agente abilità di osservazione ed apprendimento (imitare il comportamento degli altri agenti); essa è completa, però le capacità richieste per l'agente sono piuttosto complesse e richiedono molte risorse.

Nel caso in cui l'ambiente sia (anche parzialmente) sotto controllo, si può ricorrere ad una strada completamente diversa: anziché agire sulle capacità dell'agente, si modifica la visione che egli ha dell'ambiente, cercando di indurlo ad eseguire delle determinate azioni (sulla base delle conoscenze degli altri agenti).

Esistono altre tecniche che, al pari della cultura implicita, hanno cercato di estendere le potenzialità del collaborative filtering, e diversi documenti hanno cercato di proporre delle soluzioni che vadano oltre il semplice filtraggio di informazioni [24] [17]. Il limite di queste proposte è però che sono delle soluzioni per un caso specifico (ovvero delle applicazioni singole), e non è stato proposto alcun framework, al contrario della cultura implicita che fa della propria architettura ben definita ed estensibile uno dei propri punti di forza. Per questo motivo si è scelto di analizzare proprio l'*implicit culture* anziché queste altre soluzioni.

1.5.3.1 Cosa è la cultura implicita?

Cerchiamo in questo paragrafo di mostrare più nel dettaglio il problema appena definito e di proporne una soluzione.

Si assuma che l'ambiente (in cui agisce il nuovo agente) sia composto da oggetti ed altri agenti. Le azioni possono avere nei propri argomenti sia agenti che oggetti. Ad esempio *offri(libro1, prezzo1)* contiene solo oggetti, mentre *invia(lettera, ricevente)* contiene un oggetto (*lettera*) ed un agente (*ricevente*).

Prima di eseguire un'azione, un agente si trova a fronteggiare una scena formata da una porzione dell'ambiente. Per esempio, l'agente *acquirente* può affrontare *venditore1, venditore2, libro1, gadget1, prezzo1, prezzo2* ed eseguire *acquistaDa(venditore1, libro1, prezzo1)*, oppure *acquistaDa(venditore2, gadget1, prezzo2)* o ancora *acquistaNulla()*. Un agente esegue un'azione in una determinata *situazione*, ovvero affronta una *scena* ad un tempo determinato, così l'agente esegue *azioni situate*.

Subito dopo l'esecuzione di un'azione situata, l'agente affronta una nuova scena. Ad un certo istante, la nuova scena dipende dall'ambiente e dalle azioni situate eseguite. Se l'agente *acquirente1* avesse eseguito *acquistaDa(venditore1, libro1, prezzo1)* e *acquirente2* non avesse acquistato nulla (*acquistaNulla()*), il nuovo ambiente non conterrebbe più *libro1* in vendita.

Le azioni eseguite situate che un agente sceglie dipendono ovviamente da stati interni ed inaccessibili dall'esterno e non è possibile predirli deterministicamente. E però possibile

cercare di caratterizzarli attraverso tecniche probabilistiche e di valore atteso. Ad esempio, avendo l'agente *acquirente* che affronta una scena in cui può scegliere fra: *acquistaDa(venditore1, libro1, prezzoAlto)* e *acquistaDa(venditore2, libro1, prezzoBasso*) e *acquistaNulla()*, è possibile predirre che l'azione situata sarà la seconda (l'acquisto dal *venditore2* al prezzo *prezzoBasso*).

Dato un gruppo di agenti, si suppone che esista una teoria sulle loro azioni eseguite attese. Se tale teoria è consistente con le azioni eseguite dal gruppo, si dice che essa è un *vincolo culturale validato* per il gruppo. Un esempio è il seguente:

$$\forall x, y \in Gruppo, libro \in Libri:$$
$$esegui(x, acquistaDa(y, libro, prezzo)) \wedge esegui(y, vendiA(x, libro, prezzo))$$
$$\rightarrow prezzo \in [prezzoBasso, prezzoAlto]$$

Questa vincolo dice che, nel caso in cui l'agente *x* acquisti il libro *libro* da *y*, il *prezzo* sarà ragionevole, ovvero compreso fra un *prezzoBasso* (che non andrebbe bene a *y*) ed un *prezzoAlto* (che non va bene per *x*).

Se un gruppo di agenti esegue azioni che soddisfano i vincoli culturali validati del gruppo, il problema del loro comportamento subottimale è risolto. Avere un gruppo di agenti tale che le loro azioni soddisfino i vincoli culturali validati di un altro gruppo senza necessità di saperlo crea ciò che si chiama *Cultura Implicita.*

Un *Sistema per il Supporto della Cultura Implicita (SICS)* ha lo scopo di stabilire un fenomeno di cultura implicita. Questo avviene creando vincoli culturali validati dalle osservazioni sulle azioni eseguite situate, presentando poi scene agli agenti in modo tale che le loro azioni situate attese soddisfino i vincoli culturali.

1.5.3.2 Definizione dei concetti chiave relativi alla cultura implicita

Siano *nome_agente*, *nome_azione* e *nome_oggetto* delle stringhe. Definiamo l'*insieme degli agenti* P come l'insieme delle stringhe *nome_agente*; l'*insieme degli oggetti* O come l'insieme delle stringhe *nome_oggetto*, infine sia l'*ambiente* ε un sottoinsieme dell'unione dell'insieme degli agenti e di quello degli oggetti, ovvero $\varepsilon \subseteq O \cup P$

Azione: sia E un sottoinsieme dell'ambiente ($E \subseteq \varepsilon$) ed s una stringa *nome_azione*, si dice azione α la coppia $<s, E>$, dove E è detto l'argomento di α ($E = arg(\alpha)$).

Scena: sia A un insieme di azioni, $A \subseteq A$ ed $E \subseteq \varepsilon$, si definisce la scena σ come la coppia $<E, A>$ in cui, per qualunque $\alpha \in A$, $arg(\alpha) \subseteq E$, si dice che α *è possibile in* σ. Si definisce inoltre lo *spazio delle scene* $S_{\varepsilon, A}$ come l'insieme di tutte le scene.

Situazione: sia $a \in P$, α un'azione e σ una scena: una *situazione* al tempo discreto *t* è la tripla $<a, \sigma, t>$. Si dice che *a* affronta la scena σ al tempo *t*.

Esecuzione: sia $a \in P$, α un'azione: un'*esecuzione* al tempo *t* è la tripla $<a, \alpha, t>$. Si dice che *a* esegue α al tempo *t*.

Azione eseguita situata: Sia $a \in P$, sia α un'azione e σ una scena. Un'azione α è detta *azione eseguita situata* se esiste una situazione $<a, \sigma, t>$ in cui *a* esegue α al tempo *t* e α è possibile in σ . Si dice che *a* esegue α nella scena σ al tempo *t*.

Quando un agente esegue un'azione in una scena, l'ambiente reagisce proponendo all'agente una nuova scena. La relazione fra azioni eseguite situate e nuova scena dipende dalle caratteristiche dell'ambiente, e in particolare dalle leggi che descrivono la sua dinamica. Si suppone che sia possibile descrivere tale relazione attraverso una funzione dipendente dall'ambiente definita come segue:

$F_\varepsilon : A \times S_{\varepsilon, A} \times T \rightarrow S_{\varepsilon, A}$

Data l'azione situata α_t eseguita dall'agente *a* nella scena σ_t al tempo *t*, F_ε determina la nuova scena σ_{t+1} ($= F_\varepsilon(\alpha_t, \sigma_t, t)$) che verrà affrontata al tempo *t+1* dall'agente *a*.

Mentre si suppone che F_ε sia una funzione deterministica, l'azione che l'agente *a* esegue al tempo *t* è la variabile casuale $h_{a,t}$ che assume valori in A .

Azione attesa: Sia $a \in P$ e $<a, \sigma, t>$ una situazione. L'*azione attesa* dell'agente *a* è il valore atteso della variabile $h_{a,t}$, che è $E(h_{a,t})$.

Azione situata attesa: Sia $a \in P$ un agente e $<a, \sigma, t>$ una situazione. Definiamo *azione situata attesa* di *a* al tempo *t* il valore atteso della variabile $h_{a,t}$ condizionato alla situazione $<a, \sigma, t>$, ovvero $E(h_{a,t} | <a, \sigma, t>)$.

Sia L il linguaggio utilizzato per descrivere l'ambiente (agenti ed oggetti), azioni, scene, situazioni, azioni eseguite situate e azioni situate attese, e sia Σ_0 una teoria "*a priori*" che descrive l'ambiente e le relazioni tra agenti ed oggetti, in termini di azioni, scene, situazioni e azioni eseguite situate.

Teoria culturale: sia $G \subseteq P$ un insieme di agenti, si dice *teoria culturale* Σ per *G* la teoria espressa nel linguaggio L che predica sulle azioni situate attese dei membri di *G*.

Teoria culturale validata: sia Σ una teoria culturale. Σ si dice *validata* se un'azione situata attesa, stimata dalle azioni eseguite situate dei membri di *G*, soddisfa tale teoria. In tal caso *G* è detto gruppo rispetto a Σ.

Gruppo: un insieme di agenti $G \subseteq P$ è definito come *gruppo* se esiste una teoria culturale Σ tale che *G* è un gruppo rispetto a tale Σ.

Azione culturale: Sia *G* un gruppo rispetto a Σ, si definisce *azione culturale* rispetto a *G* un'azione eseguita che soddisfa Σ.

Cultura implicita: si definisce *cultura implicita* una relazione $\bowtie$ tra due insiemi di agenti *G* e *G'* ($G \bowtie G'$) tali che *G* è un gruppo e i membri di *G'* eseguono azioni culturali rispetto a *G*.

Fenomeno di cultura implicita: si definisce *fenomeno di cultura implicita* quello che si verifica tra due gruppi *G* e *G'* legati da una relazione di cultura implicita.

Il termine implicito non è riferito agli stati interni degli agenti, bensì al fatto che in generale gli agenti non fanno niente di esplicito allo scopo di produrre un fenomeno di cultura implicita.

I gruppi *G* e *G'* possono essere sovrapposti fino a coincidere; nel caso in cui i due gruppi si sovrappongano si parla di cultura in senso generale.

1.5.3.3 Architettura generale di un SICS

Per introdurre un'architettura generale di un sistema per il supporto della cultura implicita, è opportuno ricordare l'obiettivo principale di un simile sistema: stabilire un fenomeno di cultura implicita, fare cioè in modo che un nuovo agente in un determinato ambiente agisca in modo coerente con la cultura posseduta dagli altri agenti operanti nello stesso ambiente.

In figura 1.5 è mostrata un'architettura generale di un SICS, i cui obiettivi sono:

- Elaborare una teoria di vincoli culturali validati Σ dati una teoria di dominio ed un insieme di *situazioni eseguite situate* eseguite da un gruppo *G*;
- Proporre ad un gruppo *G'* un insieme di scene tali che le scene situate attese dell'insieme di agenti *G'* soddisfi Σ.

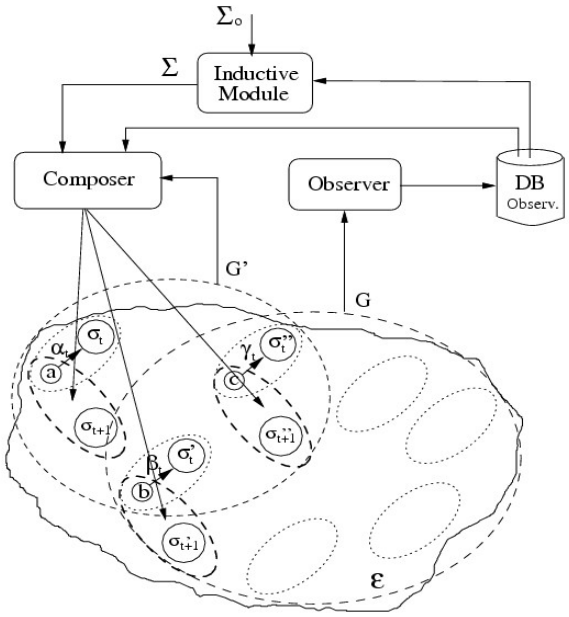

Figura 1.5: Architettura generale di un SICS

I componenti basilari di un SICS sono tre:

- *Observer*: la sua funzione è, come dice il nome, osservare le azioni eseguite situate degli agenti di *G*. Inoltre esse vengono memorizzate all'interno di un database (DB Observ.), allo scopo di poter essere riutilizzate;
- *Modulo induttivo* (*inductive module*): utilizzando le azioni situate attese contenute all'interno del database DB Observ. e la teoria di dominio Σ_0, induce la teoria culturale validata Σ;
- *Composer*: propone ad un gruppo di agenti *G'* un insieme di scene $\overline{\sigma}'_{t+1} \neq \overline{\sigma}_{t+1} = \{\overline{\sigma}_t[i] = F_\varepsilon(\alpha_t[i], \sigma_t[i]), t\}$, tale che le loro azioni situate attese $\bar{e}_{t+1}$ soddisfino Σ.

Nella figura il composer propone agli agenti *a, b* e *c* rispettivamente le scene σ_{t+1}, σ'_{t+1}, σ''_{t+1}. Si noti che in questo caso gli agenti *b* e *c* appartengono sia a *G* che a *G'*; questo significa che le loro azioni situate sono memorizzate in DB Observ., e quindi sono usate per elaborare la teoria Σ e le nuove scene.

1.5.4 Relazioni fra Data Mining, Collaborative Filtering e Implicit Culture

Nonostante siano state presentate separatamente, queste tre tecniche generali hanno molti aspetti in comune. In particolare, esse non rappresentano tre insiemi separati, ma esistono molti punti di intersezione, nel senso che alcuni algoritmi e soluzioni vengono condivisi da tutte le tecniche.

Il data mining è la più estesa e studiata delle tre metodologie, ed è stata applicata ormai in molti campi. Essa riesce a trovare delle associazioni fra i dati altrimenti quasi impossibili da individuare, ed è certamente la soluzione migliore per effettuare un'analisi di tipo off-line. Questa è proprio la sua grande limitazione, dato che i tempi richiesti sono alti e l'applicazione delle regole scoperte attraverso l'analisi deve essere effettuata in seguito e non avviene in modo automatico.

Per questo motivo si è considerato il collaborative filtering, il quale agisce on-line dando delle raccomandazioni all'utente attivo relativamente alla ricerca che sta effettuando. Le tecniche proprie dei recommender systems sono però insite nell'algoritmo che viene eseguito, e quindi piuttosto statiche, al contrario di ciò che propone la cultura implicita, in cui esiste una *teoria culturale* che può essere aggiornata automaticamente o manualmente a seconda delle esigenze. Inoltre, i sistemi di raccomandazione agiscono relativamente ad un solo utente, mentre la cultura implicita può, nei casi in cui vi sia la necessità, proporre delle azioni ad un gruppo di utenti.

Si può vedere quindi la cultura implicita come un'architettura generale, un'istanza della quale è individuabile nei recommender systems. E' bene sottolineare che gli algoritmi utilizzati nel collaborative filtering e nella cultura implicita derivano da quelli standard nati in relazione con il data mining, eventualmente adattati al particolare campo di applicazione.

1.5.4.1 Il collaborative filtering come istanza della cultura implicita

Un SICS che si basi sull'architettura appena presentata permette ad un agente di avere un comportamento migliore in un nuovo ambiente. Esempi di sistemi per il supporto della cultura implicita possono essere trovati in software reali. In questo paragrafo si esamineranno i sistemi che sfruttano tecniche di collaborative filtering.

Innanzi tutto, il *collaborative filtering* può essere considerato un'istanza dell'architettura. Infatti lo scopo di tale tecnica è il filtraggio di informazioni (*information filtering*), allo scopo di estrarre da una grande lista di oggetti quelli che l'utente preferisce. Vediamo quindi come la definizione di SICS si adatta a questo caso.

L'ambiente ε è composto di oggetti e preferenze. Gli agenti appartenenti a P sono utenti. Un agente può esplicitamente valutare un oggetto: *vota(oggetto1, voto1)*, o eseguire altre azioni come *scegli(oggetto1)* o *acquista(oggetto2)*, … che il sistema considera come votazioni, ad esempio associando *acquista(oggetto2)* con *vota(oggetto2, voto1)*. Indichiamo con A l'insieme di queste azioni. La teoria di dominio *a priori* Σ_0 nel caso di un sistema di collaborative filtering è composta da:

$$\forall x \in P, \exists \sigma_x : \forall \sigma'_x \neq \sigma_x \in S_{\varepsilon,A}$$

$$E(h_{x,t} \mid \langle x, \sigma_x, t \rangle) = vota(o, r) \wedge E(h_{x,t} \mid \langle x, \sigma'_x, t \rangle) = vota(o', r') \rightarrow r' < r$$

in cui le scene σ_x e σ'_x contengono *o* e *o'* rispettivamente e

$$\forall \sigma \in S_{\varepsilon,A}, \exists K \subseteq G \subseteq P :$$

$$\forall x, y \in K (x \neq y), E(h_{x,t} \mid \langle x, \sigma, t \rangle) = E(h_{y,t} \mid \langle y, \sigma, t \rangle)$$

La prima formula dice che, dato un certo utente, esiste una scena tale che il voto associato con l'azione situata attesa è massimo, maggiore di tutte le altre possibili scene.

La seconda indica esprime invece gli utenti sono classificati in base alle loro preferenze.

Da questa riformulazione del concetto di *collaborative filtering* in termini di *cultura implicita* è facile notare la relazione esistente fra i due termini. Come precedentemente affermato e già dimostrato in [25], è possibile considerare il *collaborative filtering* come un'istanza della *cultura implicita*, ovvero pensare a quest'ultima come una sua generalizzazione.

1.5.5 Tecniche per l'information overload ed interfacce intelligenti

Nel precedente paragrafo sono state esaminate le relazioni tra le soluzioni proposte per i problemi derivanti dalle interfacce complesse originate dai software che gestiscono l'overload di informazioni (data mining, collaborative filtering, implicit culture) in linea generale; qui verranno invece sottolineati i possibili collegamenti fra esse e lo sviluppo di interfacce intelligenti.

Il data mining non propone strumenti specifici utilizzabili direttamente per la progettazione e l'implementazione di interfacce intelligenti; nonostante questo gli algoritmi tipici di questa tecnica sono facilmente applicabili. Ad esempio la scoperta di regole associative può essere molto utile per prevedere i bisogni dell'utente (per determinare quale funzionalità utilizzerà), e le tecniche di clustering e classificazione sono altrettanto necessarie per capire a quale categoria appartenga un utente e quindi che tipo di interfaccia adottare.

Per quanto concerne il collaborative filtering, le relazioni presenti fra esso e le interfacce intelligenti sono molto più strette: infatti esso è uno strumento che si può presentare proprio sotto forma di interfaccia intelligente (si adatta all'utente e/o alla situazione specifica, raggruppa gli utenti in gruppi, …). Sono stati scritti diversi white papers relativi alla progettazione di interfacce intelligenti connesse al collaborative filtering: *AlteredVista* [30] è un esempio di sistema di questo tipo.

Dato che si può considerare la cultura implicita come generalizzazione del collaborative filtering, è anche possibile affermare che si può creare un'interfaccia intelligente basata sulla cultura implicita. Essendo inoltre la cultura implicita più generale del collaborative filtering, è possibile creare delle interfacce ancora migliori, sfruttando tutte le caratteristiche assenti nei recommender systems: la presenza di una teoria culturale, ad esempio, permette un grado di adattabilità e dinamicità molto maggiore.

Capitolo 2

Progettazione di un Framework per interfacce intelligenti basate su SICS

Dopo aver analizzato, nel capitolo precedente, gli strumenti utilizzabili al fine di affrontare con successo il problema dell'utilizzo di interfacce utente complesse, ed aver evidenziato quali tecniche siano le più efficaci, si propone in questo capitolo una soluzione generale per la realizzazione di interfacce intelligenti il cui funzionamento sia basato sulla cultura implicita. In altre parole si progetta un Framework riutilizzabile ed adattabile alle esigenze specifiche, capace di offrire le funzionalità tipiche di un sistema di supporto per la cultura implicita e quindi di interfaccia intelligente.

L'implementazione di istanze di questo framework porta alla creazione sia di interfacce software, da integrare all'interno di applicazioni la cui interfaccia utente sia difficilmente utilizzabile in modo efficace (soprattutto per utenti con poca esperienza) che di una componente server separata, con il compito di effettuare le computazioni per la facilitazione dell'utilizzo del software.

Il presente capitolo, interamente dedicato all'analisi del framework, inizia con un paragrafo relativo all'analisi dei requisiti, attività fondamentale nell'Ingegneria del Software per la realizzazione di qualsiasi sistema software, proseguendo poi con la realizzazione di un diagramma delle classi che potrà in futuro essere utilizzato come base di partenza per l'implementazione di un sistema di questo tipo. Si effettua infine un'analisi delle scelte progettuali ed implementative per le parti principali del framework, mostrando vantaggi e svantaggi di ogni alternativa.

2.1 Analisi dei requisiti

In questa fase si devono innanzitutto identificare quali siano gli *attori* presenti nel contesto esaminato, ovvero quali entità (sia persone fisiche che agenti software) utilizzino o siano in qualche modo correlate al software. Una volta che sono stati individuati gli attori, è necessario determinare ciò che essi richiedono al software, ovvero i modi con cui necessitano di interagirvi: si dice che devono essere individuati i *casi d'uso*.

Nel framework esaminato gli attori sono solo due: l'*amministratore del sistema* e l'*utente*. Si ricorda che un attore raggruppa una classe di soggetti, ossia non identifica una singola

entità (il signor Mario Rossi), rendendo possibile la presenza di più amministratori e/o più utenti.

L'*amministratore del sistema* è l'attore che richiede dal software il maggior numero di funzionalità, dato che è colui che deve controllarne il corretto funzionamento ed impostarne i parametri nel caso in cui non svolga a dovere la propria funzione.

Nella figura 2.1 possiamo vedere i casi d'uso relativi all'amministratore di sistema: questo attore si occupa principalmente degli aspetti di gestione del software, ovvero della gestione della teoria culturale, dell'amministrazione delle applicazioni da gestire e dello stato del sistema (avviato / arrestato).

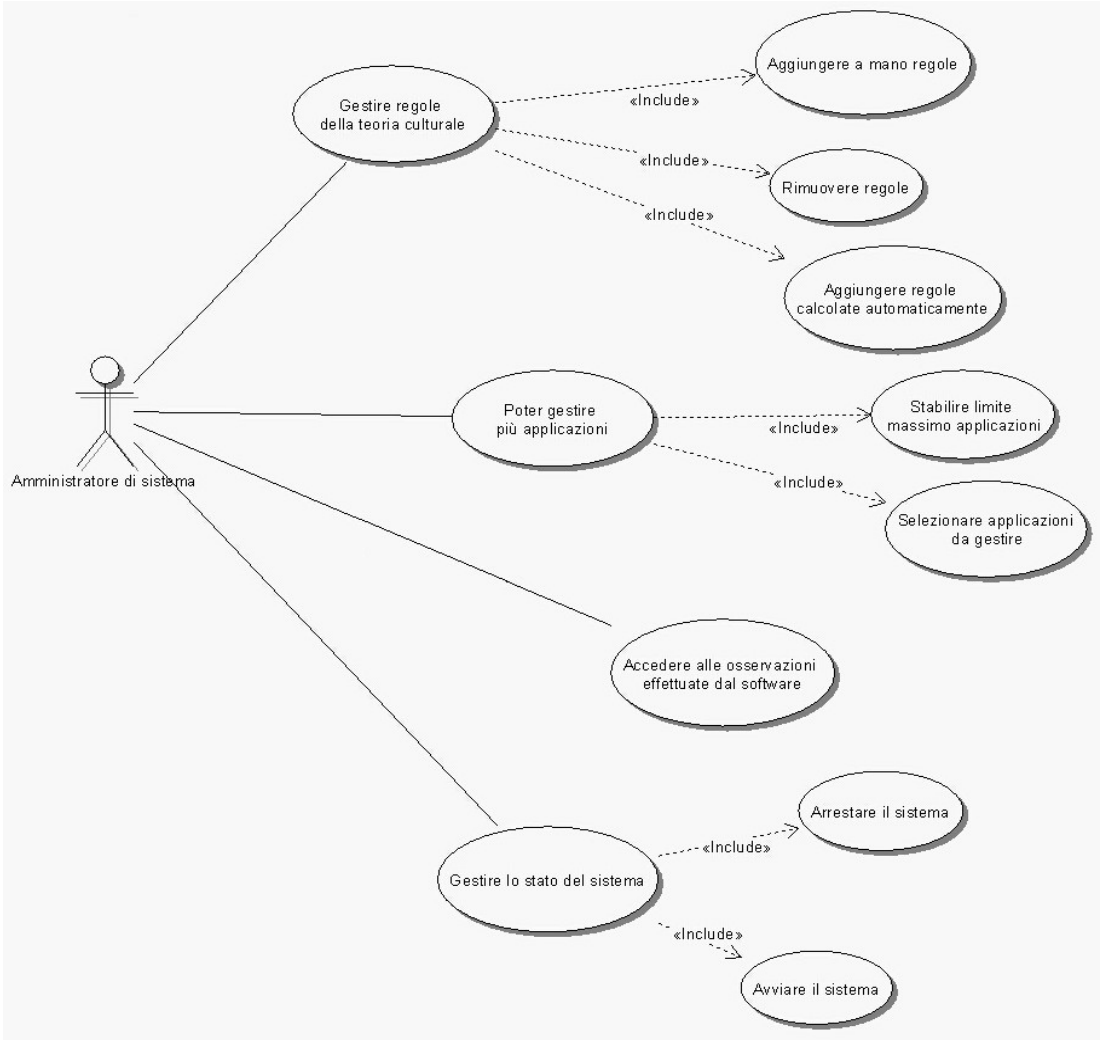

Figura 2.1: Use-cases diagram relativo all'amministratore di sistema

Esaminiamo ora in dettaglio il significato dei casi d'uso individuati:

- *Gestione regole della teoria culturale*: il software deve permettere all'amministratore di gestire a proprio piacimento le regole appartenenti alla teoria culturale, determinando in tal modo il comportamento del sistema. Questo caso d'uso include diversi sotto casi d'uso richiesti per il suo soddisfacimento :
    - *Aggiungere a mano regole*: avere la possibilità di inserire manualmente le regole per indirizzare il sistema in una certa direzione e poterne aggiungerne alla lista di quelle già attive;
    - *Rimuovere regole*: essere in grado di rimuovere una o più regole fra quelle attive;
    - *Aggiungere regole calcolate automaticamente*: il modulo induttivo è in grado, a partire da una teoria a priori Σ_0 ed utilizzando le osservazioni effettuate, di elaborare una nuova teoria culturale: questo deve essere ovviamente supportato dal sistema tramite la realizzazione di un apposito modulo.
- *Poter gestire più applicazioni*: qualsiasi sistema basato sul framework deve poter supportare in contemporanea più applicazioni, tramite l'utilizzo di *interfacce intelligenti* su diversi sistemi. Deve essere possibile effettuare due tipi di interventi:
    - *Stabilire limite massimo di applicazioni*: si può decidere di limitare il numero di applicazioni da gestire per non sovraccaricare troppo il sistema;
    - *Selezionare applicazioni da gestire*: il software deve fornire un meccanismo che permetta di stabilire, per le applicazioni supportate, se esse debbano o meno utilizzare l'interfaccia intelligente.
- *Accedere alle osservazioni effettuate dal software*: il modulo *Observer* ha lo scopo di effettuare osservazioni circa il comportamento degli utilizzatori del software; tali dati devono essere situati in qualche luogo (tipicamente in un database), e spesso l'amministratore può necessitare di esaminarli. Il software deve mantenerli in un formato il più possibile comprensibile all'interno di un database accessibile dagli amministratori;
- *Gestire lo stato del sistema*: ultimo, ma non per importanza, requisito del sistema in relazione all'utilizzo da parte dell'amministratore è la possibilità di gestire lo stato dell'intero sistema. Per fare questo deve essere possibile:

- *Arrestare il sistema*: nel caso sorgano complicazioni o vi siano richieste particolari il sistema deve prevedere un modo per venire arrestato;
- *Avviare il sistema*: l'amministratore può decidere di avviare il sistema quando lo ritenga opportuno.

Per l'utente di sistema, l'unico use-case è la ricezione di suggerimenti circa l'utilizzo (figura 2.2):

Figura 2.2: Casi d'uso per l'utente di sistema

- *Ottenere suggerimenti*: l'interazione fra software ed utente (non amministratore) consiste nell'ottenere suggerimenti relativi all'utilizzo del software. Stabilire con quali modalità e secondo quali tecniche debbano essere presentati non è compito dell'utente, l'importante è solo che egli riceva tali consigli.

2.2 Architettura di sistema

Dopo aver definito quali siano le funzionalità richieste dagli attori che interagiscono con il sistema, si propone un'architettura generale, attraverso la definizione di un diagramma delle classi, individuando in tal modo un framework riutilizzabile per varie applicazioni che, sfruttando la tecnica della cultura implicita, offrano un'interfaccia intelligente. Il *class diagram* è rappresentato nello standard UML [28] in figura 2.3.

Gli elementi fondamentali legati al concetto di cultura implicita sono l'*Observer*, l'*Inductive Module* ed il *Composer*. Nel framework proposto richiedono più di una classe ciascuno: questo soprattutto perché vi è la necessità di utilizzare una struttura di tipo client / server fra le interfacce per le applicazioni che utilizzano il sistema ed il corpo principale del sistema stesso (quello contenente la parte server), richiedendo in tal modo la presenza di classi di interfaccia da realizzare separatamente dai moduli componenti il server. Prima di esaminare nel dettaglio le singole classi presenti in figura 2.3 si ritiene giusto identificare la composizione dei tre moduli dell'Implicit Culture all'interno del framework proposto:

1. ***Observer***: la parte client è nella classe di interfaccia *ObserverInterface*, che invia delle osservazioni di tipo *Observer* al gestore delle comunicazioni client / server, ovvero a *ClientManager*. Questa classe comunica a sua volta con *Actions*, che contiene le osservazioni utilizzate per fornire suggerimenti attraverso interfacce

intelligenti, inviando l'azione effettuata. L'interfaccia *ObserverInterface* memorizza poi all'interno del *Database* l'azione svolta. La scelta di mantenere le informazioni in memoria, oltre che nel database, è dovuta al fatto che deve essere possibile, attraverso apposite strutture dati, accedere rapidamente alle osservazioni ritenute rilevanti al momento (ad esempio quelle più recenti) senza dover considerare tutte le osservazioni del database (che comporterebbe un appesantimento dei compiti del sistema);

2. ***Inductive module***: anche qui è presente una parte client per la gestione delle funzionalità offerte dal modulo induttivo: *InductiveModuleInterface* comunica con *ClientManager* che, interpretando le richieste relative al modulo induttivo, interagisce con la classe *InductiveModule*. Questa restituisce poi il risultato della richiesta, ovvero fornisce la nuova teoria culturale, per la determinazione della quale è necessario che vengano utilizzate le osservazioni mantenute in memoria, ovvero la classe *Actions*.
3. ***Composer*:** il terzo componente della cultura implicita, con il compito di facilitare l'utente proponendogli le azioni più probabili, utilizza alcune classi facenti parte degli altri moduli. La classe principale è quella chiamata *Composer*, la quale comunica con l'interfaccia di osservazione *ObserverInterface* proponendole una o più azioni da eseguire. Tale interfaccia riceve dei suggerimenti in un formato standard, che verranno poi interpretati e formattati da un'ulteriore classe chiamata *SuggesterFormatter*.

Si procede quindi con la descrizione delle singole classi presenti nel diagramma 2.3, non prima di aver effettuato una dovuta premessa: alcune classi saranno effettivamente da sviluppare all'interno del corpo principale del sistema (quelle dal lato server). Altre sono delle interfacce da aggiungere alle applicazioni che vogliono utilizzare le funzionalità offerte dal sistema, sviluppate differentemente a seconda della natura dell'applicazione (come plug-in per applicazioni stand-alone, integrando pagine web nel caso di software web, …). Infine esistono delle classi di supporto, non necessariamente implementate come tali: ad esempio le informazioni relative alle comunicazioni client / server non sono obbligatoriamente rappresentate da classi.

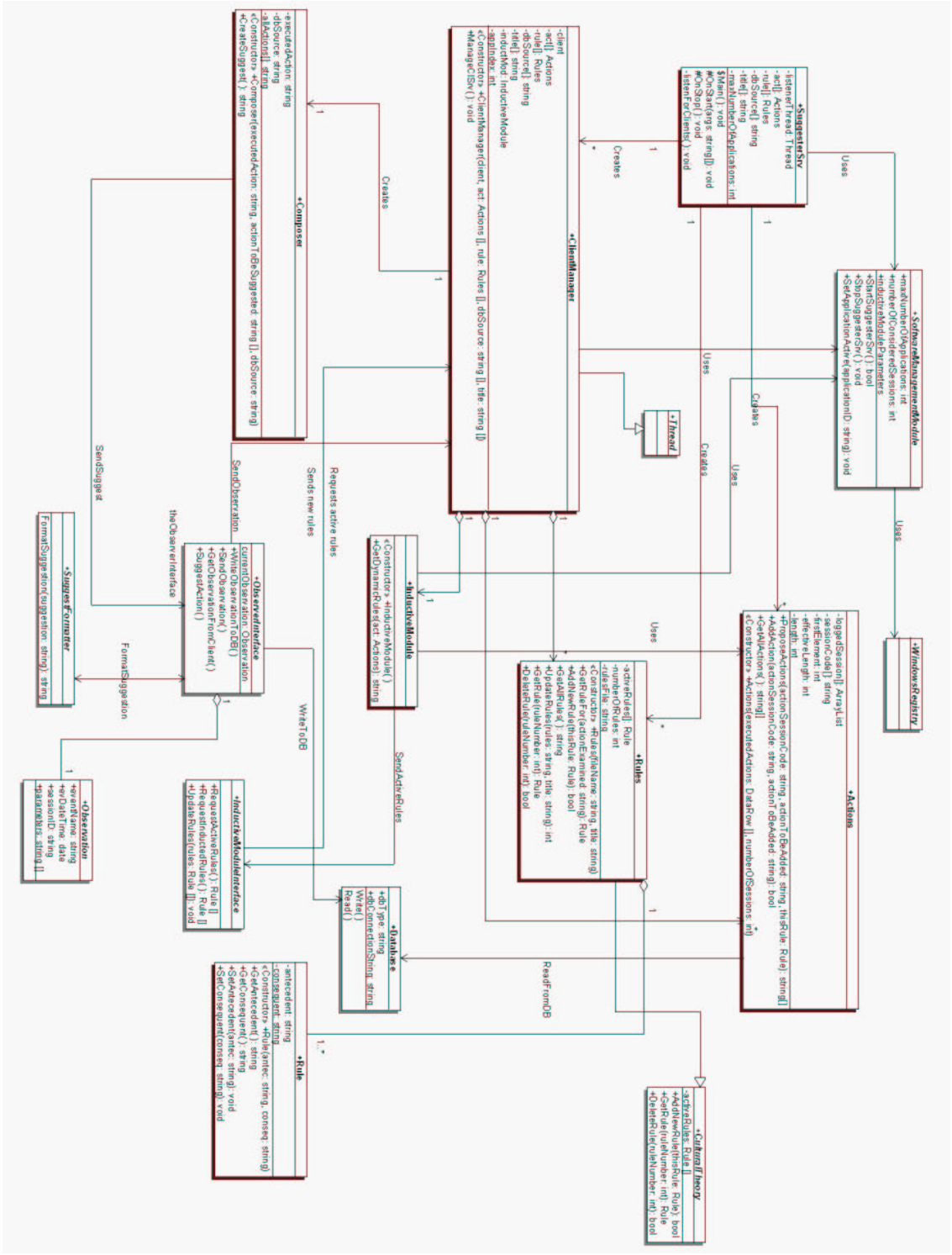

Figura 2.3: Diagramma delle classi del framework

Vengono ora descritte le classi effettivamente da sviluppare nel corpo principale del sistema:

- *SuggesterSrv*: tale classe è quella principale, a partire dalla quale vengono create istanze di tutte le altre. Contiene diversi attributi indispensabili, fra cui *listenerThread* di tipo *Thread* utilizzato per la creazione di un thread che resta in attesa della connessione di nuovi client (interfacce) che comunichino con il sistema. Per questo motivo esiste il metodo *listenForClients*, il quale viene eseguito in un thread separato dal corpo principale del programma. La variabile intera *maxNumberOfApplications* rappresenta il numero massimo di applicazioni supportate (ognuna delle quali necessita di apposite strutture dati), è inizializzata tramite l'ausilio del *SoftwareManagementModule*, ed il suo valore serve per dimensionare alcuni array. Questi, condivisi con le altre classi, sono relativi alle informazioni sulle applicazioni attive: *act* di tipo *Actions* contiene le osservazioni ritenute importanti per la creazione di suggerimenti, *rule* di tipo *Rules* comprende la teoria culturale per ogni applicazione, la stringa *dbSource* rappresenta le stringhe di connessione ai database delle varie applicazioni, *title* (anch'esso stringa) contiene gli identificatori delle applicazioni. Sono poi presenti il metodo *Main*, e due funzioni di reazione agli eventi relativi all'intero sistema: *OnStart* definisce il comportamento al momento dell'avvio (tipicamente si inizializzano alcuni parametri utilizzando i dati nel *SoftwareManagementModule*) ed *OnStop* controlla l'arresto del sistema, facendo pulizia nella memoria e chiudendo connessioni e files;
- *ClientManager*: la funzione di questa classe è la gestione di un client connesso al sistema (per client si intende un'interfaccia che invia dati ed attende risposte). Per fare questo deve prevedere la distinzione fra vari tipi di richiesta, a seconda che sia effettuata da parte di un'*interfaccia di osservazione*, che invia le azioni effettuate attendendo suggerimenti, o da un'*interfaccia del modulo induttivo*, la quale può richiedere la teoria culturale o inviarne una nuova. E' presente un attributo *client*, il quale è il mezzo per identificare l'interfaccia con la quale l'istanza della classe sta comunicando; il tipo non è definito, permettendo l'utilizzo di diverse modalità di comunicazione. Anche qui sono presenti gli stessi array di *SuggesterSrv* (*act*, *rule*, *dbSource* e *title*), i quali vengono passati al momento della creazione dell'istanza di *ClientManager* come parametri del costruttore. Vi è anche un'istanza di *InductiveModule* chiamata *inductMod*, dato che, nel caso in cui la presente classe sia connessa con un'interfaccia relativa al modulo induttivo, sarà indispensabile comunicare con la classe principale di tale modulo. Un ultimo attributo intero è *appIndex*: gli array contengono informazioni relative a tutte le applicazioni connesse, e serve un modo per identificare a quale applicazione ci si debba riferire,

in particolare un indice relativo agli array. Il metodo contenente le funzionalità della classe è *ManageClSrv*, il quale gestisce la comunicazione fra client e server, individuando il tipo di richiesta e soddisfacendola tramite l'ausilio delle altre classi;

- *Actions*. serve per memorizzare le osservazioni effettuate dalle interfacce. Visto che il framework è rivolto all'implementazione di sistemi che offrano interfacce intelligenti, e quindi verranno effettuate osservazioni sull'utilizzo delle funzionalità di un software (i quali hanno caratteristiche in comune), si ritiene necessario porre un vincolo almeno relativamente alle strutture dati da utilizzare per la memorizzazione delle osservazioni. Esse sono contenute all'interno di un array circolare *loggedSession* di dimensione massima *length* (passato con il costruttore) e dimensione effettiva *effectiveLength*. Non è infatti necessario utilizzare tutte le sessioni presenti nel database (che rallenterebbe troppo il software); si considerano invece solamente le ultime *k*. Per identificare le sessioni all'interno dell'array *loggedSession* e verificare se una nuova azione appartenga ad una sessione memorizzata o ad una nuova sessione, si utilizza un array di appoggio (di stringhe) chiamato *sessionCode*. Ogni elemento dell'array è un'istanza della classe ArrayList (una lista concatenata realizzata utilizzando array), per permettere la presenza di sessioni di dimensione arbitraria. I metodi presenti nella classe sono il costruttore che inizializza le strutture dati, *AddAction* per aggiungere una nuova azione passando fra i parametri il codice della sessione e l'azione da aggiungere, un metodo (*GetAllActions*) che permetta di ottenere tutte le azioni in un formato standard (utilizzato per il modulo induttivo). Troviamo infine una fra le funzioni più complesse dell'intero framework: *ProposeActions*. Essa ha lo scopo di proporre un suggerimento per l'utente (da rappresentare poi attraverso l'interfaccia del composer): i parametri sono la stringa relativa alla sessione, l'ultima azione eseguita e la regola da soddisfare (già determinata attraverso un apposito metodo della classe *Rules*). Attraverso l'utilizzo delle tecniche più opportune, è necessario che il metodo *ProposeActions* restituisca una o più azioni che soddisfino la regola, ordinate dalla più probabile alla meno probabile;
- *InductiveModule*: scopo di questa classe è l'induzione di teorie culturali. Per questo motivo è presente, oltre al costruttore, un solo metodo: *GetDynamicRules*. Esso può divenire algoritmicamente piuttosto complesso, visto che deve, utilizzando qualche tecnica, generare una teoria culturale relativa all'ambiente di riferimento. Per fare questo necessita di un parametro contenente le osservazioni dell'applicazione che dovrà considerare. E' probabile che sia necessario ricorrere al *SoftwareManagementModule*, in cui sono presenti dei parametri per il corretto funzionamento dell'algoritmo di induzione;

- *Rules*: questa classe deriva da *CulturalTheory*, e quindi la sua funzione è l'implementazione della gestione della teoria culturale. Al suo interno è presente un array chiamato *activeRules* contenente elementi di tipo *Rule* (regole singole). La dimensione di tale array è definita da *numberOfRules*, ed il file relativo alle regole dell'applicazione di riferimento è indicato nella stringa *rulesFile*. Il costruttore inizializza le strutture dati e la teoria culturale di partenza, andando a leggere il file *fileName* passato come parametro. Il metodo *GetRuleFor* riceve in input l'azione da considerare e deve restituire la prima regola in cui l'antecedente sia soddisfatto da tale azione (viene restituito un oggetto di tipo *Rule*). *AddNewRule* permette l'inserimento di una nuova regola all'interno della teoria culturale, *GetAllRules* ritorna una stringa in formato standard contenente tutte le regole, *UpdateRules* permette l'aggiornamento completo delle regole, *GetRule* ritorna la regola presente alla posizione *ruleNumber* dell'array, mentre *DeleteRule* ne permette la rimozione;

- *Rule*: per rappresentare una singola regola si è deciso di creare questa struttura dati. Gli attributi sono due stringhe, rappresentanti rispettivamente l'antecedente ed il conseguente della regola. E' presente un costruttore ed i metodi *get* e *set* relativi sia all'antecedente che al conseguente;

- *Composer*: tale classe rappresenta parte del modulo Composer della cultura implicita. Il suo compito è di suggerire le azioni all'utente. Per fare questo bisogna formattare le stringhe ottenute attraverso il metodo *ProposeActions* di *Actions* in un formato standard, comprensibile dalla parte client del Composer (*SuggestFormatter*). In alcuni casi può essere necessario effettuare alcuni controlli: ad esempio, se si vuole suggerire di esaminare il primo record di una tabella, si può pensare di determinarlo in modo dinamico (visto che il contenuto dei database cambia); gli attributi sono *executedAction*, una stringa che indica l'ultima azione eseguita (utile nel caso in cui i parametri relativi all'azione da proporre debbano essere valutatati in relazione all'azione precedente), *dbSource* per la stringa di connessione al database dell'applicazione considerata ed *allActions*, un array contenente la lista delle azioni da proporre. Oltre al costruttore è necessario il metodo *CreateSuggest*, che si occupa della formattazione standard delle azioni da proporre (ed eventualmente effettua dei controlli sui parametri dell'azione da proporre);

Si passa ora all'esame delle altre classi, che comprendono le interfacce per le applicazioni e le classi accessorie (che possono rappresentare anche solamente delle entità – non legate allo sviluppo).

- *ObserverInterface*: uno degli elementi più importanti dell'intero sistema. Il suo compito è infatti quello di osservare il comportamento degli utenti, di memorizzarlo nel database e di inviarlo al *ClientManager* che, dopo averlo interpretato, restituirà, nel caso in cui lo ritenga necessario (se esiste una regola soddisfacibile nella teoria culturale), una o più azioni in un formato standard. E' presente una variabile chiamata *currentObservation* di tipo *Observation*, rappresentante l'osservazione appena effettuata. I metodi sono *WriteObservationToDB*, per memorizzare la *currentObservation* nel database dell'applicazione di riferimento, *SendObservation*, che la invia al *ClientManager*, *GetObservationFromClient*, per osservare il comportamento dell'utente, e *SuggestAction*. Quest'ultima funzione suggerisce all'utente l'azione (o una lista di azioni) da eseguire; per fare questo ci si appoggia alla classe *SuggestFormatter*;
- *SuggestFormatter*: il suo compito è solamente di convertire una stringa (che è in formato standard) in un nuovo formato comprensibile direttamente dall'applicazione. Per fare questo bisogna chiamare il metodo *FormatSuggestion*; è possibile pensare alla creazione di diversi tipi di classi *SuggestFormatter*, per venire incontro a differenti tipi di applicazioni che necessitano di conversioni diverse;
- *Observation*: rappresenta una singola osservazione. Essa è identificata dagli attributi *eventName* (il nome dell'evento), *evDateTime* (quando è stato eseguito), *sessionID* (l'identificatore della sessione corrente), *parameters* (un array che indica eventuali parametri aggiuntivi). Ovviamente la struttura di questa classe è molto generica, ed andrà adattata alle varie situazioni;
- *InductiveModuleInterface*: questa interfaccia serve per la gestione del modulo induttivo. Come previsto dagli use-cases solo l'amministratore di sistema ha a disposizione la possibilità di utilizzare questo modulo (la gestione del modulo induttivo non è un caso d'uso dell'utente). I metodi sono *RequestActiveRules*, il quale ritorna l'intera teoria culturale attiva, *RequestInductedRules*, che fa in modo che la classe *InductiveModule* esegua un'induzione sulla teoria culturale, e *UpdateRules* con parametro *rules* di tipo array di *Rule*, per aggiornare la teoria culturale con le azioni selezionate dall'utente;
- *SoftwareManagementModule*: raggruppa tutto ciò che serve per modificare o accedere alle impostazioni relative al sistema. Sono presenti gli attributi *maxNumberOfApplications* (il numero massimo di applicazioni gestibili), *numberOfConsideredSessions* (quante sessioni sono da considerare per le varie operazioni), *inductiveModuleParameters* (le impostazioni per il modulo induttivo, che variano a seconda dell'algoritmo scelto). Fra i metodi *StartSuggesterSrv* e

StopSuggesterSrv, hanno lo scopo di avviare ed arrestare il sistema, e *SetApplicationActive* permette di aggiungere un'applicazione (il parametro *applicationID*) a quelle gestite dal software. Visto che parte del sistema è composto di interfacce, è probabile che in un'implementazione effettiva non tutte queste informazioni possano essere raggruppate in un unico modulo, ma debbano essere divise collegandole in qualche modo alle diverse interfacce. Dal diagramma delle classi si evidenzia inoltre la necessità di accedere al registro di sistema (*WindowsRegistry* supponendo di lavorare in Windows) per ottenere alcune variabili;

- *CulturalTheory*: è la classe da cui deriva *Rules*. Essa comprende solamente gli attributi ed i metodi fondamentali per gestire la cultura implicita, ovvero la lista delle regole (l'array *activeRules* di tipo *Rule*) ed i metodi per aggiungere (*AddNewRule*), togliere (*DeleteRule*) ed ottenere (*GetRule*) specifiche regole;
- *Database*: è la rappresentazione tramite classe dei databases utilizzati dalle varie applicazioni e per mantenere le osservazioni dell'Observer. E' caratterizzata dal tipo di database (*dbType*), dalla stringa di connessione (*dbConnectionString*) e dai metodi per l'accesso in lettura (*Read*) ed in scrittura (*Write*);

2.3 Analisi dei componenti base

Dopo aver presentato, tramite il diagramma delle classi, una struttura generale per il framework che si vuole proporre, si passa ora ad un'analisi dettagliata degli aspetti più importanti relativi al framework stesso. In particolare si evidenzieranno le scelte strutturali ed algoritmiche effettuabili, mostrandone pregi e difetti.

2.3.1 Struttura generale

Questa prima sezione è interamente dedicata a problemi ed aspetti che riguardano il sistema nella propria completezza. Si analizzeranno quindi le caratteristiche globali, senza scendere troppo nel dettaglio dei vari moduli (questo è fatto in seguito).

Il primo aspetto da affrontare consiste nel **tipo di applicazione server** da realizzare. Il requisito fondamentale è che l'applicazione possa essere eseguita automaticamente all'avvio del sistema, e che per l'amministratore sia possibile gestirne lo stato (riavvio / arresto). Per fare questo è quindi necessaria un'applicazione che agisca in background: se si vuole sviluppare in ambiente Windows la scelta migliore consiste nella creazione di un *servizio di MS Windows*, mentre per Linux sarà sufficiente un applicativo eseguito in background e che venga avviato con l'entrata nel sistema. Scegliendo come linguaggio di sviluppo Java è possibile realizzare un'applicazione funzionante in entrambi i sistemi.. Una soluzione alternativa è la creazione di un *web service*, cioè un sistema che offra

funzionalità richiedibili tramite il web. Affinché tale applicazione funzioni è però necessaria la presenza di un web server; questa scelta è idonea per i software con interfaccia web, mentre non è il modo migliore per comunicare con le classiche applicazioni.

Per quanto concerne la **comunicazione fra client e server**, cioè fra le interfacce (sia di osservazione che relative al modulo induttivo) ed il sistema di suggerimento, sono possibili diverse scelte. Una prima soluzione consiste nell'utilizzo di primitive di sincronizzazione per la gestione dei dati in comune. Per fare questo si può ricorrere alla memoria condivisa (*shared memory*) ed alla sincronizzazione dell'accesso tramite *semafori*. Certamente questa non è la soluzione migliore: un primo motivo sta nell'impossibilità di utilizzare memoria condivisa ed i semafori fra macchine diverse (situazione che avviene utilizzando la parte server dell'applicazione su una macchina e quella client su altri computer); un ulteriore impedimento è legato alla necessità di programmare a livello piuttosto basso (ad esempio il framework .NET non supporta nativamente la memoria condivisa). Una soluzione migliore consiste nell'utilizzare comunicazione attraverso lo standard TCP/IP: in tal modo si possono utilizzare macchine diverse, oltre che poter agire sullo stesso computer. Il TCP/IP è ormai ampiamente diffuso ed adottato per la maggior parte delle comunicazioni fra diverse macchine.

Le **impostazioni generali** relative al sistema devono essere memorizzate su disco, affinché vengano conservate anche in caso di riavvio del sistema. Diverse alternative sono applicabili, partendo dalle più banali, come l'utilizzo di un semplice file di testo in un formato proprietario. Tale scelta è però discutibile per l'assenza di uno standard nella rappresentazione delle informazioni. Per porre rimedio a tale problema si può pensare all'utilizzo di un file *xml* (*eXtensible Markup Language*), il quale è composto da uno schema specifico utilizzabile, nel caso particolare, per mantenere le impostazioni del software. Nel caso in cui si decida di propendere per uno sviluppo legato a Windows, si può creare un'apposita sezione nel registro di sistema contenente una chiave per ogni elemento di configurazione.

Un problema simile è quello legato a dove mantenere il **Log con le osservazioni** effettuate dal modulo Observer. Infatti, queste informazioni vengono utilizzate da diversi moduli del sistema, ed è necessario definirne la collocazione. Una prima soluzione consiste nella creazione di un database apposito per i log. Senza dover necessariamente creare un database apposito si può aggiungere una tabella al database al quale l'applicazione si collega (nei casi in cui ne sia presente uno). Da scartare è l'utilizzo di un file di testo, visto che l'accesso allo stesso risulta molto più difficoltoso e complesso (manca la possibilità di effettuare queries, per esempio). Si potrebbe casomai pensare alla scelta di un file di testo

solamente come supporto aggiuntivo per l'amministratore, facendo accedere comunque il software al database.

Un ulteriore aspetto è la **struttura del lato client**, ovvero delle interfacce. In altre parole bisogna determinare come possa l'interfaccia intercettare gli eventi derivanti dall'utilizzo del software da parte dell'utente, e come faccia a suggerirgli le azioni da eseguire. Bisogna tenere in considerazione la difficoltà nel reagire agli eventi di un qualsiasi software: sarà necessario che il software al quale si vuole applicare il sistema sia sviluppato in modo modulare. Ciò significa che, considerando un applicativo stand-alone, dovrà fornire la possibilità di inserire dei plug-in e disporre di apposite API (Application Programming Interface) che permettano di intercettare gli eventi. Nel caso in cui si tratti di un software web, la situazione è diversa: si possono integrare delle funzioni scritte in linguaggi per pagine web dinamiche (Asp o Php) che gestiscano gli eventi e che comunichino con il server. In tal caso bisogna determinare come possa avvenire tale comunicazione: molto probabilmente sarà richiesto l'utilizzo di un apposito oggetto da integrare all'interno della pagina che supporti il TCP/IP (ad esempio un oggetto COM per Windows). Infine, ci si potrebbe trovare in un caso (raro) in cui esista già un database (o un file) che effettui il log dell'applicazione. Qui bisognerà monitorare continuamente tale log per cercare di fornire un'interfaccia intelligente.

2.3.2 Teoria culturale

Prima di esaminare nel dettaglio ciò che riguarda la gestione del sistema per indurre una teoria culturale e per suggerire le azioni all'utente è necessario esaminare varie soluzioni relative alla rappresentazione della teoria culturale.

Come definito nel paragrafo 1.5.3.2, per teoria culturale si intende una "*Teoria espressa in un linguaggio L che predica sulle azioni situate attese dei membri di G*".

Per fare questo bisogna quindi definire il linguaggio migliore per stabilire tale teoria. Certamente l'utilizzo di un qualche tipo di *regole* nel formato *antecedente* → *conseguente* è la scelta più opportuna, visto il loro elevato grado di leggibilità e la facilità di trovare metodologie già esistenti che le gestiscano. Inoltre è necessario tenere in considerazione l'importanza della sequenzialità delle azioni eseguite: se in molti contesti può avere un'importanza relativa l'ordine in cui vengono eseguite le azioni, nel caso dell'utilizzo di un software tale ordinamento assume un'importanza fondamentale. Per quanto riguarda il formalismo utilizzabile per la rappresentazione delle regole una buona scelta è l'uso di *Regole associative (Association Rules)*, già presentate in 1.5.1.4.

Esse sono delle regole nella forma A_1 *and* A_2 *and ... and* $A_n \rightarrow C_1$ *and* C_2 *and ... and* C_m, e rappresentano una soluzione molto efficace nel caso esaminato. Chiaramente sarà necessario che la temporalità di antecedenti e conseguenti venga rispettata, nel senso che

$t(A_i) < t(A_j) \ \forall i<j$ ed allo stesso modo $t(C_i) < t(C_j) \ \forall i<j$. Tenendo in considerazione lo scopo di un'interfaccia intelligente, si può considerare la possibilità di ridurre il conseguente ad un solo atomo: se nel passato sono state eseguite delle azioni in successione, allora sarà probabile che la prossima azione da eseguire sia quella riscontrata nella scena precedente. In alcuni casi, si può ridurre in forma atomica anche l'antecedente. Questo fa perdere qualcosa relativamente alle potenzialità del linguaggio scelto, ma permette all'intero sistema di reagire in modo molto più rapido (potrebbe essere utile soprattutto nel caso dell'applicazione di interfacce ad un software web o in presenza di molti utenti in contemporanea). Bisognerà quindi che si determini il livello di formalità (e quindi di dinamicità ed espressività) del linguaggio a seconda dell'implementazione particolare prescelta, trovando un compromesso fra potenza espressiva e rapidità di esecuzione. Nel caso in cui l'interfaccia non agisca tempestivamente l'utilizzatore potrebbe preferire andare a tentativi nella scelta delle funzioni anziché attendere i suggerimenti da parte delle interfacce.

2.3.3 Inductive Module

Il modulo induttivo è la parte del framework che lo differenzia maggiormente da un sistema di interfacce intelligenti basato sul collaborative filtering. Questo componente permette infatti la creazione di una teoria culturale a run-time, rendendo il sistema dinamico ed adattabile ai vari tipi di utilizzo dello stesso.

Un primo problema legato all'inductive module è **quando e come deve essere eseguito.** Se da un lato un utilizzo continuo (ad esempio un'induzione per ogni osservazione ricevuta) garantirebbe un grado di elevata dinamicità, dall'altro bisogna tenere in considerazione che l'esecuzione di un algoritmo induttivo può richiedere diverso tempo, e rallentare anche fortemente il sistema. Varie soluzioni sono proponibili per risolvere questa problematica: una prima scelta è decidere se eseguire manualmente il modulo induttivo, ovvero attraverso un'interfaccia per l'amministratore, o se fare in modo che esso operi automaticamente. Nel secondo caso è da decidere ogni quanto tempo la teoria culturale debba essere aggiornata: è possibile definire un intervallo temporale, anche se questo potrebbe essere poco efficace nel caso in cui l'utilizzo del software non sia omogeneo nel tempo, oppure si può scegliere un'esecuzione ogni n sessioni, o ancora un mix di queste due soluzioni.

Per quanto riguarda invece la parte più algoritmica, un primo punto consiste nel decidere se considerare tutte le azioni presenti all'interno del database delle osservazioni per l'applicazione in questione, o se utilizzare solo i dati relativi alle ultime k sessioni. Questa potrebbe essere una soluzione migliore, dato che mantenere troppe informazioni relative all'utilizzo del passato può portare ad avere un'interfaccia poco dinamica, poiché sempre

legata ad osservazioni remote, ed inoltre provocherebbe un rallentamento del sistema. Un problema è però l'individuazione di un valore di *k* adeguato per lo scopo. La sperimentazione di varie alternative è il metodo migliore per ottenere un valore significativo da assegnare a tale variabile.

Si passa ora all'analisi degli algoritmi utilizzabili per il funzionamento del modulo induttivo. Bisogna individuare l'algoritmo ritenuto migliore fra quelli proposti per la scoperta delle regole associative, dopo aver evidenziato (nel paragrafo 2.3.2) che la teoria culturale è esprimibile in modo molto efficace attraverso delle regole.

L'algoritmo standard dal quale derivano quasi tutti gli altri legati alle regole associative è chiamato *A-Priori*. Esso è fortemente basato sui concetti di confidenza e supporto definiti in 1.5.1.4 e [34]. Si parte da uno scenario in cui si hanno dei gruppi di elementi (nel nostro caso un gruppo equivale ad una sessione); l'obiettivo è la ricerca di regole associative considerando quali siano le più frequenti all'interno delle transazioni (dei gruppi di elementi). A-Priori è un algoritmo di tipo *BFS (Breadth First Search)* che determina il supporto degli insiemi di elementi (*itemset*) salendo di livello. Lo pseudo-codice dell'algoritmo è presentato in figura 2.4.

```
Sia C1 l'insieme di tutti gli 1-itemsets, sia k = 1
While Ck ≠ ∅
Esamina il database per determinare il supporto σ(A)  ∀ A∈Ck
Estrai gli itemset il cui supporto è > min_support  da Ck e
inseriscili in Lk
Genera Ck+1
k = k + 1
```

Figura 2.4: Algoritmo A-Priori

Ciò significa che inizialmente viene determinato il supporto degli insiemi di un solo elemento, per poi passare a quelli con due elementi se il supporto degli 1-itemset era soddisfacente, arrivando poi ai 3-itemsets, …

L'algoritmo A-Priori basa la propria efficacia su un'osservazione fondamentale [29]:

Ogni sottoinsieme di un itemset frequente è frequente. Per questo motivo un itemset candidato in C_{k+1} può essere eliminato se uno qualsiasi dei suoi sottoinsiemi non è contenuto in C_k..

Questo principio permette di evitare di dover scorrere tutti gli itemsets di dimensione *k+1*: si esamineranno solamente quelli ottenibili attraverso gli itemsets di dimensione *k* il cui supporto è maggiore di *min_support*.

La rimozione riduce il carico di lavoro, ma è ancora necessario eseguire il controllo dei subset di ogni singola transazione per ogni iterazione, e questo può portare a lunghi tempi

di esecuzione per grandi itemsets (molte iterazioni). Per questo motivo sono state progettate diverse variazioni dell'algoritmo A-Priori, le quali mirano ad un'esecuzione più rapida;

- *A-Priori con tecniche di Hashing*: i candidati vengono memorizzati all'interno di un albero di hashing, riducendo in tal modo la dimensione di C_k al passo *k-1*;
- *Partizionamento*: un itemset che è potenzialmente frequente in un database deve essere frequente almeno in una delle partizioni del database;
- *Campionamento (Sampling)*: utilizzare un sottoinsieme dei dati riduce l'*accuratezza* del calcolo ma aumenta decisamente l'*efficienza*;

Il problema principale legato a questo algoritmo standard sta nel fatto che non tiene in considerazione la sequenzialità degli elementi presenti all'interno di una transazione. Se si vuole porvi rimedio è necessario modificarlo opportunamente.

Nel caso di un sistema di interfacce intelligenti basato sulla cultura implicita si può pensare ad uno scenario ancora diverso: si può assumere la creazione di regole il cui antecedente e conseguente siano formati da un solo atomo, generando regole come *{azione1}* → *{azione2}*.

Questo, nonostante sia una limitazione sulle potenzialità del modulo induttivo nell'individuare regole complesse e precise, può essere considerato positivo in relazione al tempo di esecuzione del software ed all'efficienza nella presentazione di nuove scene.

Fare in questo modo significa determinare gli itemset di dimensione 2 con supporto maggiore della soglia *min_support*. E' possibile correlare il tutto con un vincolo sulla confidenza, escludendo le regole con confidenza minore di una certa soglia *min_confidence* (ovvero determinare se il rapporto fra il numero di volte in cui si presenta una certa regola sul totale delle regole con lo stesso antecedente sia maggiore della soglia). Questa tecnica può essere considerata la creazione di un algoritmo ad-hoc piuttosto che una modifica dell'algoritmo A-Priori.

In figura 2.5 è rappresentata tramite l'utilizzo di pseudo-codice, partendo dalle osservazioni in memoria fino a raggiungere un array contenente le regole ordinate dalla più probabile alla meno probabile.

```
HashTable ht = new HashTable();
string [] allActions = GetObservationsInMemory();
for all action ∈ allActions do
      if IsValidRule (action, action.next) then
            ht.Add(GenerateRule(action, action.next));
      end if
end for
for i = 0 to ht.Keys.Length do
      ht.Keys[i] = GenerateGenericRule (ht.Keys[i]);
      if IsBanalRule(ht.Keys[i]) then
            ht.Values[i] = -1;
      end if
end for
string [] keys = ht.Keys.CopyToArray();
int [] values = ht.Values.CopyToArray();
Array.Sort(keys, values, Ascending);
int diffElements = GroupEqualRules(values, keys);
string [] allRulesString = new string[diffElements];
integer [] allRulesValues = new int[diffElements];
float [] allRulesSupport = new float[diffElement];
CopyNonNullValues(keys, values, allRulesString, allRulesValues);
for k = 0 to allRulesString.Length do
      if Confidence(k) ≥ confidenceThreshold
            if Support(k) ≥ supportThreshold then
                  allRulesSupport[k] = support;
            else allRulesSupport = 0
            end if
      else allRulesSupport = 0;
      end if
end for
Array.Sort(allRulesSupport, allRulesString, Descending);
```

Figura 2.5: Algoritmo custom per il modulo induttivo

Si inizia con la creazione di una tabella hash: essa è utile perché composta da chiavi (qui saranno le regole individuate sotto forma di *azione[n] -> azione[n+1]*) e da valori (il numero di volte in cui tale azione è ripetuta). Allo scopo di inizializzarlo, viene creato un array contenente tutte le azioni (*allActions*), e lo si scorre aggiornando anche il numero di volte in cui è presente. Si esamina nuovamente la tabella hash, allo scopo di generalizzare le regole (sostituendo i valori particolari con delle costanti se possibile) e di eliminare le regole banali. Si copiano sia le chiavi che i valori in due array appositi (più comodi da gestire della tabella hash), e si ordinano i valori in base all'ordine alfabetico delle chiavi. Così si possono raggruppare le regole uguali (l'ordine alfabetico serve a questo), sommandone il numero di presenze nella prima occorrenza e settandolo al valore -1 nelle altre (assimilandole ad una regola banale). Vengono poi creati degli array la cui dimensione è uguale al numero di regole differenti individuate, e al loro interno si copiano le regole ed il numero di volte in cui sono state individuate. Per ogni regola si calcolano confidenza e supporto, settando quest'ultimo a zero nel caso in cui la confidenza o il

supporto siano sotto le rispettive soglie minime (*ConfidenceThreshold* e *SupportThreshold*). A questo punto si possono ordinare gli array in base al supporto, di modo che le regole siano ordinate dalla più probabile alla più remota.

2.3.4 Composer

Relativamente all'esame del composer, un primo elemento è legato alla struttura dello stesso, ossia alla sua **architettura**. Bisogna definire come avvenga il processo di comunicazione fra interfacce dell'applicazione e sistema server in relazione alla ricezione dei suggerimenti. Il funzionamento è mostrato in figura 2.6; per chiarezza è evidenziato anche l'interazione con il modulo observer.

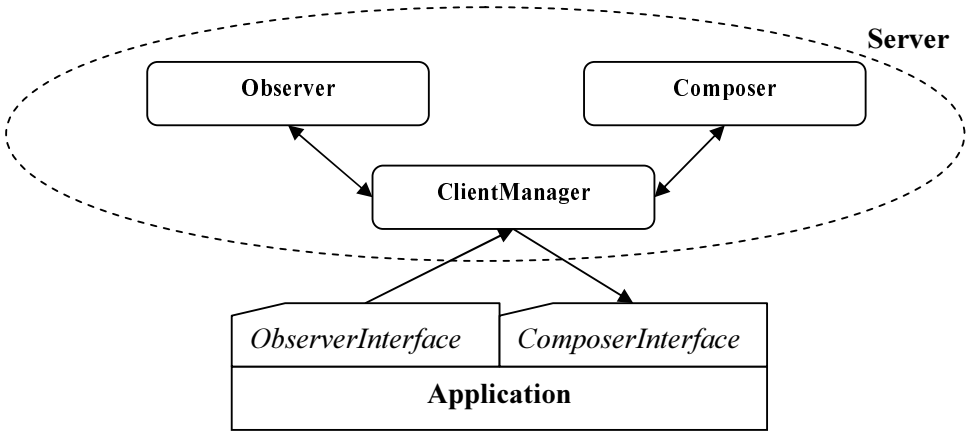

Figura 2.6: Architettura del sistema per observer e composer

Il modulo composer è stato suddiviso in due sottomoduli: una classe appartenente al server chiamata *Composer* e l'interfaccia applicata all'applicazione che sfrutterà il sistema di interfacce intelligenti. La comunicazione fra client e server non avviene, nel server, attraverso l'utilizzo delle classi *Observer* e *Composer*, ma sfrutta la classe *ClientManager*, che ha il compito di gestire le richieste provenienti dalle interfacce e rispondere (utilizza le altre classi per determinare la risposta).

Un ulteriore fattore da considerare è relativo ad un'altra funzionalità richiesta al composer: una volta che si è determinata la regola della teoria culturale utilizzata per suggerire all'utente, è possibile che per la stessa azione vengano individuate diverse varianti da proporre all'utente (in base alle sessioni passate); bisogna determinare il modo di **ordinare queste azioni dalla più probabile alla meno probabile**. Per farlo si può ricorrere a diversi parametri:

- Un *coefficiente di similarità* fra la scena corrente (quella dell'utente attivo che ha inviato le osservazioni ed attende eventuali suggerimenti) e le scene di riferimento delle azioni proposte;

- Un *identificatore temporale* che ordini le azioni proposte in base cronologicamente;
- Il *numero di volte* che l'azione è stata eseguita nelle sessioni memorizzate.

Per l'ordinamento il metodo più corretto è considerare innanzi tutto il coefficiente di similarità fra le scene, ordinando tutte le azioni da quella con maggior grado di similarità a quella con livello più basso. Nel caso in cui siano presenti delle situazioni di parità (ugual grado di similarità) si può ricorrere ad una chiave secondaria di ordinamento basata sul numero di volte in cui l'azione è stata eseguita. Una terza chiave consiste nell'identificatore temporale, ordinando i gruppi con similarità uguale dall'azione più recente alla più remota.

Per quanto riguarda il calcolo del **coefficiente di similarità** sono utilizzabili diverse tecniche del collaborative filtering, che devono però tenere in considerazione l'importanza della sequenzialità di esecuzione delle azioni. Le tecniche standard, utilizzanti i coefficienti di correlazione di Pearson o di Spearman ed il coseno dell'angolo fra vettori, forniscono in generale risultati molto interessanti per quanto riguarda la similarità fra utenti (o sessioni) di dimensione prefissata.. Nel caso di un'interfaccia intelligente, la dimensione dei vettori da esaminare può essere diversa, ed è meglio calcolare la similarità fra due scene in un modo diverso: è possibile ad esempio determinare quante azioni simili siano state eseguite in due scene partendo dall'azione che precede immediatamente l'antecedente della regola, individuando così un coefficiente che stabilisce la similarità.

In figura 2.7 è proposto lo pseudocodice per la valutazione del coefficiente di similarità della scena corrente (*currentScene*) rispetto ad un'altra (*examinedScene*). Nell'algoritmo proposto sono presenti dei controlli per evitare di riferirsi ad un elemento dell'array non esistente (il cui indice è minore di zero).

```
currentScene ed examinedScene sono due array di dimensione n ed
m; gli indici delle azioni simili sono rispettivamente i e j.
similarityCoeff = 0;
for k = i downto 0 do
      if j < 0 then
            return similarityCoeff;
      end if
      if AreSimilar(currentScene[k], examinedScene[j]) then
            similarityCoeff = similarityCoeff + 1;
      else return similarityCoeff;
      end if;
      j = j - 1;
end for
return similarityCoeff;
```

Figura 2.7: Algoritmo per la determinazione della similitudine fra due scene

Viene effettuata una scansione a ritroso delle due scene esaminate, decrementando ad ogni passo i rispettivi indici (*i* e *j*). Per ogni coppia di azioni si determina la similarità tramite la funzione *AreSimilar*: nel caso in cui siano simili si incrementa il coefficiente di similarità, altrimenti si esce dal ciclo ritornando il risultato (il coefficiente). L'altra situazione che implica l'uscita dall'algoritmo consiste nel tentativo di accedere ad un elemento precedente l'inizio di uno dei due array (uno degli indici diviene minore di zero).

Capitolo 3

Applicazione del framework a 3wGIS

Per evidenziare l'effettiva applicabilità del framework proposto nel capitolo precedente si è deciso di realizzarne un'implementazione da applicare ad un sistema reale. Per fare questo si è partiti dalla struttura generale offerta dal framework, operando delle scelte relative alla struttura del software, al tipo di comunicazione client / server, alle strutture dati ed agli algoritmi. Compito di questo capitolo è proprio analizzare l'applicazione realizzata. L'istanza del framework è stata applicata al software 3wGIS, applicativo web la cui interfaccia utente è complessa, come già mostrato nel paragrafo 1.5.1.

Le sezioni seguenti analizzeranno rispettivamente la struttura del sistema nel suo insieme, il diagramma delle classi derivato dal framework ed un'analisi relativa al processo di facilitazione dell'utilizzo del software.

3.1 Struttura del sistema

Un primo aspetto è legato ai **linguaggi di programmazione** da utilizzare ed alle **piattaforme software** supportate. Si sceglie di effettuare un'analisi congiunta di questi concetti perché la scelta dell'uno provoca ripercussioni sull'altro. Occorre evidenziare innanzi tutto che le scelte proponibili sono fortemente legate al software al quale si vuole applicare il sistema. In questo caso, 3wGIS è funzionante solamente utilizzando Internet Explorer in ambiente Windows: il mancato supporto ad altre piattaforme è legato soprattutto all'oggetto ActiveX con il compito di visualizzare la cartografia (la tecnologia ActiveX non è supportata da tutti i browsers), e ad alcune scelte implementative. Il software è stato sviluppato principalmente utilizzando Html ed Asp, con qualche script a lato client scritto in JavaScript. Dal punto di vista delle interfacce è necessario dunque che esse abbiano una parte scritta in Html ed Asp, almeno per quanto riguarda il collegamento visuale con l'utente.

Visto lo stato attuale di 3wGIS non è possibile pensare ad una piena integrazione in un sistema utilizzante Linux o Unix; questione a parte è rappresentata dai sistemi Macintosh, i quali integrano una versione di Internet Explorer. Per quanto riguarda lo sviluppo dell'applicazione dal lato server, pensando ad una possibile estensione del software al fine di permetterne l'utilizzo anche in Linux, l'utilizzo di un linguaggio interpretato da una macchina virtuale (in figura 3.1 è mostrata la sua collocazione rispetto agli altri

componenti di un computer) adattabile a diversi sistemi operativi si ritiene migliore rispetto alla realizzazione di eseguibili tradizionali (che avrebbero vantaggi dal punto di vista dell'efficienza). Allo stato attuale Java è la tecnologia più utilizzata, dato che permette l'implementazione di software eseguibile da una macchina virtuale (chiamata *JVM – Java Virtual Machine*) su sistemi Windows, Linux, Macintosh ed altri. Accanto a questa soluzione prodotta da Sun, sta emergendo una soluzione alternativa proposta da Microsoft, ovvero il .NET Framework. Esso opera in modo simile a Java, offrendo in più la possibilità di effettuare una programmazione in diversi linguaggi (C++, C# , Visual Basic, J#). La scelta è ricaduta sull'utilizzo del .NET Framework poiché offre un supporto migliore per lo sviluppo di software in ambiente Windows (3wGIS allo stato attuale è sostanzialmente un'applicazione solo per Windows). Per quanto riguarda il linguaggio di programmazione, si è escluso immediatamente Visual Basic, il quale non fornisce una sintassi sufficientemente chiara; si è deciso di eliminare anche C++, dati i problemi che può comportare la programmazione tramite puntatori. E' stato deciso di utilizzare C#, linguaggio creato da Microsoft come evoluzione di C++, il quale cerca di unire gli elementi migliori del C++ (potenza) e di unirli a quelli di Java (chiarezza del codice).

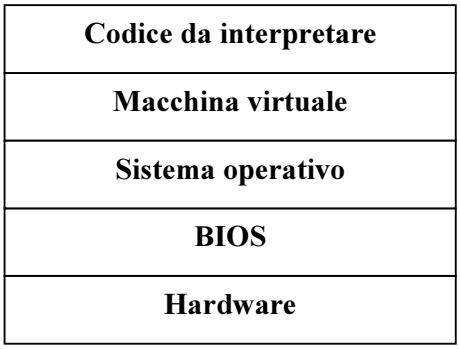

Figura 3.1: Macchina virtuale

Si passa ora all'esame della struttura del sistema in relazione all'**architettura client / server** di cui è composto. Dopo aver definito la necessità dell'utilizzo, come linguaggi di programmazione principali, di C# per il lato server e di Html con Asp per il client, bisogna analizzare in maggior dettaglio la struttura di tali elementi ed il modo in cui essi comunicano. Per quanto riguarda quest'ultima caratteristica (la comunicazione client / server), come già detto in relazione al framework in generale, la scelta migliore consiste nella comunicazione attraverso il protocollo Tcp/Ip, ed in particolare utilizzando i Sockets di Berkeley, una libreria di funzioni per gli sviluppatori che permette la comunicazione fra processi su diverse macchine. Essi sono stati introdotti nel 1982 dal sistema operativo Unix 4.1 BSD. A questo punto sorge però un problema, poiché la programmazione tramite Asp non permette di connettersi ad un'altra macchina su una porta determinata tramite

Sockets: la soluzione scelta è la creazione di un oggetto da inserire all'interno della pagina Asp sotto forma di oggetto COM. Anche per questo componente lo sviluppo è realizzato in C# (altri linguaggi possono andare altrettanto bene, dato che i Sockets sono indipendenti dal linguaggio). Per quanto riguarda il server si sceglie l'implementazione di un servizio di Windows, che può essere reso automaticamente eseguibile all'avvio del sistema e che permette nativamente la gestione dell'avvio ed arresto del processo. In figura 3.2 viene rappresentata schematicamente l'architettura generale del sistema sotto forma di moduli fondamentali, evidenziando i linguaggi utilizzati ed il tipo di comunicazione client / server. I contenitori rettangolari rappresentano le entità che compongono il software, mentre le frecce indicano la comunicazione fra le entità che collegano: esse sono comunicazioni fra elementi dello stesso processo, se non specificato diversamente.

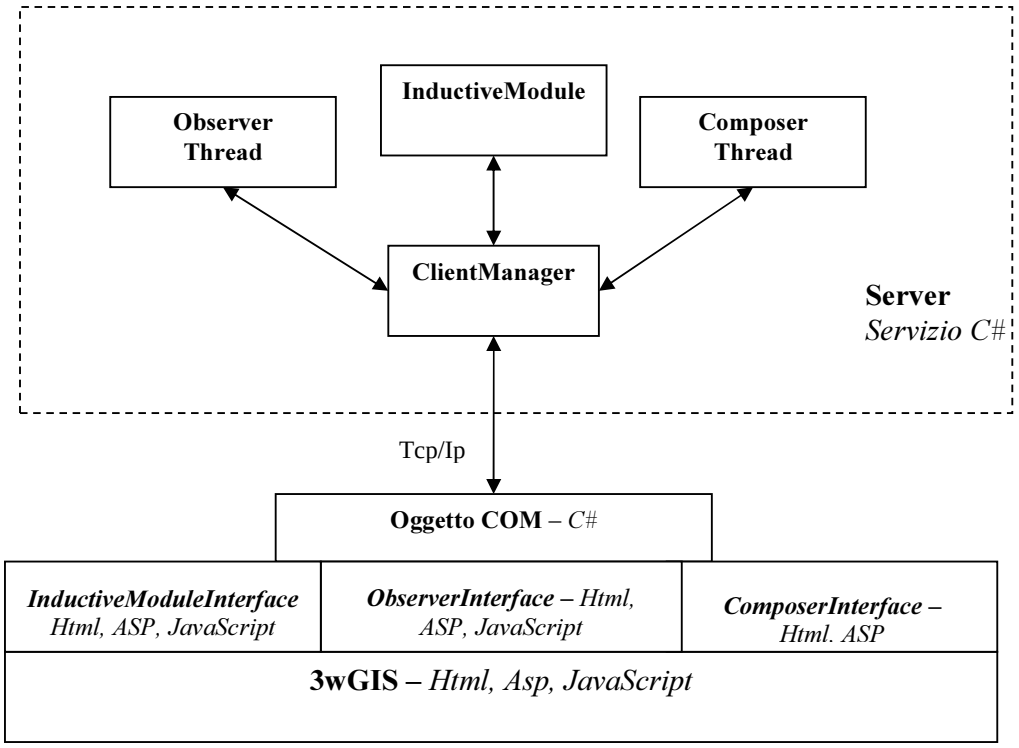

Figura 3.2: Architettura del sistema realizzato

Un terzo punto da tenere in considerazione relativamente al sistema nel suo insieme è la **localizzazione e struttura degli elementi accessori**: bisogna definire dove vengano conservate le informazioni relative al corretto funzionamento del sistema di raccomandazione e come vengano rappresentate. Nel caso specifico si è scelto l'utilizzo del registro di Windows per la memorizzazione delle impostazioni generali relative al

funzionamento del server: si è creata una chiave *SuggesterSrv* che contiene i parametri di funzionamento. La voce *MaxNumberOfApplications* indica il numero massimo di applicazioni gestite dal software: nel caso in cui più applicazioni vogliano connettersi esse verranno respinte (potranno comunque funzionare senza interfaccia intelligente); *NumberOfConsideredSessions* è un intero che permette di decidere quante sessioni debbano essere mantenute in memoria per il funzionamento del composer e del modulo induttivo. Sempre all'interno della chiave *SuggesterSrv* la voce *Port* rappresenta la porta sulla quale il servizio rimane in attesa della connessione di clients; sono poi presenti due voci relative al funzionamento del modulo induttivo (i suoi parametri di funzionamento: *ConfidenceThreshold* e *SupportThreshold* indicano le soglie minime di confidenza e supporto affinché le regole individuate possano essere aggiunte alla teoria culturale (per il modulo induttivo si utilizza l'algoritmo personalizzato proposto in figura 3.5). Attraverso il registro si riesce quindi ad impostare il funzionamento del server; per quanto riguarda i clients (le interfacce) si deve provvedere ad un diverso tipo di soluzione: esse possono risiedere su macchine diverse rispetto al server, e quindi potrebbero non poter accedere al registro di *SuggesterSrv*. L'applicazione 3wGIS è corredata da un file di configurazione di tipo *.ini*: ad esso sono state aggiunte delle opzioni per impostare correttamente il funzionamento delle interfacce intelligenti: *ApplicationID* è un identificatore univoco per l'applicazione utilizzata: tale voce permette di inviare al *SuggesterSrv* una stringa che gli permetta di distinguere fra le applicazioni che gestisce in contemporanea; *SuggesterServer* indica il nome (o l'indirizzo IP) del computer sul quale risiede il modulo server; *SuggesterPort* identifica la porta tramite la quale effettuare la connessione Tcp/Ip e *SuggesterOnOff* permette di attivare (impostando 1) o disattivare (valore $\neq 1$) il sistema di interfacce intelligenti in relazione all'applicazione considerata. Un'ultima considerazione relativa agli elementi accessori riguarda le regole della cultura implicita. Esse sono mantenute sul server nella stessa cartella in cui è installato il software, all'interno di un file di testo in cui, per ogni riga, è contenuta una di esse.

3.2 Diagramma delle classi

Come già fatto per la presentazione del framework generale, anche qui viene riportata la rappresentazione del sistema implementato attraverso l'utilizzo di class diagram.

In questo caso sono mostrate solamente quelle classi che hanno avuto effettiva implementazione, e sono presenti nel sistema all'interno del modulo server o di quello client. Il diagramma generale in figura 3.3, comprendente tutte le classi senza indicarne né attributi né operazioni, è seguito dalle analisi individuali del client e del server.

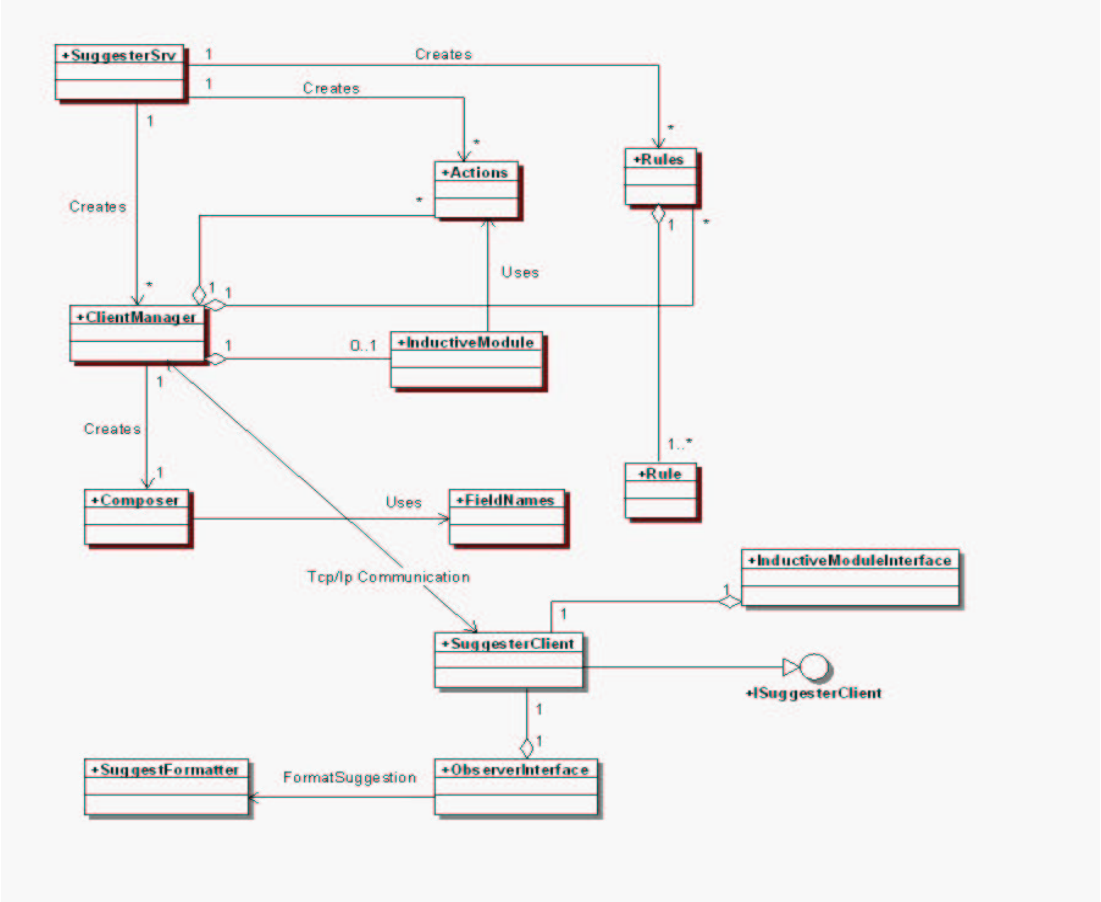

Figura 3.3: Class Diagram generale del sistema

La struttura del lato server è sostanzialmente la stessa proposta nel framework: sono presenti la classe di base *SuggesterSrv*, il gestore delle comunicazioni fra un'interfaccia ed il server *ClientManager*, *Actions* per memorizzare le osservazioni da considerare al momento, *InductiveModule* per le operazioni relative all'induzione della teoria culturale, *Rules* contenente le regole (istanze di *Rule*) e *Composer* per la creazione dei suggerimenti in formato standard. E' presente una sola classe in più rispetto a quelle necessarie proposte dal framework: *FieldNames*. Essa è strettamente legata all'ambito per il quale viene realizzato il sistema (3wGis), e contiene diversi metodi che accedono al database utilizzato da 3wGis per esaminare delle tabelle, ricevendo in input un campo noto e ritornandone uno ignoto. Alcune funzioni non sono al momento utilizzate dal sistema, ma sono state sviluppate in previsione di eventuali utilizzi futuri. Passando al lato client, balza subito all'occhio la presenza di una nuova classe chiamata *SuggesterClient*, che implementa l'interfaccia *ISuggesterClient*. Questa non era prevista nel disegno generale del framework; è stata aggiunta per la gestione delle comunicazioni Tcp/Ip con il server: si

tratta di un oggetto COM sviluppato per permettere l'utilizzo dei sockets di Berkeley in pagine web. Da evidenziare è anche il fatto che la comunicazione fra client e server avviene totalmente fra la classe client *SuggesterClient* e quella server *ClientManager*. Sia l'interfaccia del modulo induttivo che quella relativa all'observer contengano un'istanza dell'oggetto COM, che si adatta quindi ai vari tipi di comunicazione fra client e server (ricevendo parametri diversi). Come previsto dalle specifiche del framework sono state implementate le classi *SuggestFormatter, ObserverInterface* ed *InductiveModuleInterface*, utilizzate rispettivamente per la formattazione in un linguaggio comprensibile dall'applicazione 3wGis dei suggerimenti (Html) e come interfacce per l'osservazione e per la gestione del modulo induttivo.

3.2.1 Analisi del lato server

Per offrire un'analisi completa della componente server del software si ritiene ottimale la presentazione del diagramma delle classi. Esso è mostrato In figura 3.4, e consiste in un servizio di Windows creato con C#: i nomi delle classi e dei tipi di dati rispecchiano la sintassi proposta da questo linguaggio. La sua analisi tiene in considerazione soprattutto le differenze con il modello proposto dal framework: viene esaminata ogni singola classe senza soffermarsi approfonditamente sulle relazioni fra le stesse, visto che sono già state evidenziate nell'analisi del framework.

- *SuggesterSrv*: per quanto riguarda la classe principale del sistema essa differisce in maniera molto lieve da come era stato previsto dal framework. Le modifiche sono dovute sostanzialmente alla decisione di effettuare lo sviluppo di un servizio per Microsoft Windows: è presente infatti un ulteriore attributo chiamato *components* di classe *System.ComponentModel.Container*, il cui scopo è mantenere le informazioni relative al tipo di servizio che si è sviluppato (le impostazioni dello stesso). Per quanto riguarda i metodi sono stati aggiunti *InitializeComponent* e *Dispose*, con lo scopo di inizializzare *components* e di rimuovere ciò che contiene la memoria nel momento della terminazione del servizio. Il resto degli attributi è conforme alle specifiche individuate nel framework; si ricorda la presenza degli array contenenti le informazioni per la gestione delle applicazioni connesse (*act, rule, dbSource, title*), la variabile *maxNumberOfApplications* per indicare il numero massimo di applicazioni gestibili (inizializzata utilizzando il registro di sistema) ed il metodo *listenForClients* che crea un Thread per ogni connessione ricevuta;

Figura 3.4: Class diagram completo del server

- *ClientManager*: anche questa classe, che ha il compito di gestire la comunicazione Tcp/Ip fra client e server, ha un'implementazione che discosta molto poco da quella del framework: sono presenti solamente due metodi in più: *GetRulesFileName* e *IndexInArray*. Il primo ritorna il nome del file delle regole relativo all'applicazione considerata: per ognuna è conservato, sulla macchina che ospita il server, un file contenente la teoria culturale, il cui nome è composto dal prefisso *rules* seguito dal titolo dell'applicazione (l'*applicationID*). Questo metodo ne verifica l'esistenza e, in caso contrario, lo crea copiandolo da un file contenente le regole standard. *IndexInArray* serve invece per determinare quale sia l'indice, all'interno dei vari array per la gestione delle applicazioni, relativo a quella identificata dal parametro *currentTitle*. Gli attributi sono rimasti gli stessi del framework, ovvero gli array contenenti le informazioni relative alle applicazioni gestite, un'istanza del modulo induttivo (da utilizzare nel caso in cui serva), l'indice dell'applicazione corrente *appIndex* (settato proprio con il metodo *IndexInArray*) e la variabile *client* di tipo *Socket* per effettuare la connessione con il client. Il metodo principale è *ManageClSrv* che si occupa appunto della gestione della comunicazione;
- *Actions* è la classe che ha subito il maggior numero di modifiche rispetto a quanto proposto nel capitolo precedente. In realtà la struttura generale è rimasta la stessa (sono presenti tutti i metodi individuati per il framework), ma, vista la complessità di alcuni di essi, sono state realizzate delle funzioni di appoggio allo scopo di creare un codice più leggibile e facilmente modificabile. Gli attributi della classe sono gli stessi identificati nel framework, e consistono nella struttura dati contenente le azioni appartenenti alle sessioni in memoria (*loggedSession*), nella lista dei loro identificatori (*sessionCode*), e in tre interi utilizzati per la navigazione all'interno di *loggedSession*: *length* è la dimensione totale della struttura dati (il numero di sessioni massimo), *effectiveLength* indica il numero di sessioni effettivamente in memoria e *firstElement* rappresenta la posizione del primo elemento nell'array. I metodi sono invece numerosi: fra quelli del framework il costruttore, *AddAction* e *GetAllActions* richiedono poco commento (sono piuttosto semplici), mentre per *ProposeAction* il discorso è differente, visto che sfrutta quasi tutti gli altri metodi per svolgere le proprie funzioni. La funzione *ProposeActions* ha lo scopo di individuare un insieme di azioni che possano essere considerate come conseguente di una regola della teoria culturale (il parametro *thisRule)*. L'esecuzione di questo metodo è seguita dall'ordinamento delle azioni dalla più probabile alla meno probabile tramite la funzione *OrderActions*. Le funzioni accessorie sono *HasNoQuestionMarks*, per determinare se il conseguente della regola sia privo di punti di domanda, ovvero di valori da determinare;

CompleteConsequentConstants è usato per inserire i valori delle costanti nel conseguente di una regola, partendo dall'antecedente e dall'azione eseguita. Per verificare se una coppia di azioni (antecedente e conseguente) verifichi una regola è presente il metodo *IsRuleSatisfied*; allo scopo di effettuare confronti relativi all'antecedente si utilizza *AreAntecedentSimilar*, che imposta anche il valore delle costanti in apposite variabili passategli per riferimento. Tali valori vengono utilizzati per modificare il conseguente (l'azione da proporre) tramite *ModifyConsequent*, per poi controllare se il conseguente sia verificato utilizzando *AreConsequentSimilar*. *AreSimilar* serve per determinare la similarità fra due azioni; *GetCompleteConsequent* ha lo scopo di modificare un conseguente che abbia valori da calcolare, utilizzando *ModifyConsequent* e *SetQuestionMarks* per inserire nel conseguente le costanti ed i valori da calcolare. *PositionInArray* serve per cercare la posizione di un elemento (una stringa) in un array di stringhe – che ritorna la prima posizione libera in caso non sia stato individuato, *ShowSessionsOrderedByTime* per salvare su file le sessioni in memoria (utile per debug) e *GetString* ritorna l'azione numero *index* in una sessione determinata (partendo da 0);

- *InductiveModule*: il modulo induttivo (dal lato server) rispecchia le caratteristiche richieste dal framework, integrandole con alcuni attributi e metodi di supporto. Fra gli attributi *supportThreshold* e *confidenceThreshold* indicano le soglie di supporto e confidenza per l'algoritmo utilizzato (inizializzati attraverso il registro). Il costruttore e *GetDynamicRules* sono i metodi previsti dal framework, ai quali ne sono stati aggiunti altri. *GenerateRule* serve per creare una regola generica, sostituendo i valori specifici con costanti, valori liberi ed altri da calcolare; *AreDuplicates* verifica se due regole siano considerabili equivalenti; *SimilarActions* determina la similarità di due azioni senza considerarne i parametri (solo il tipo di azione); *HasClassParam* è utilizzata per verificare se un'azione abbia o meno fra i propri parametri il riferimento ad una classe specifica (cioè ad una tabella del database contenente i dati dell'applicazione);

- *Rules*: la rappresentazione della teoria culturale (nel diagramma 3.3 si è mostrato come questa classe derivi da quella astratta *CulturalTheory*) è stata implementata rispettando totalmente le specifiche determinate per il framework. Non è stato aggiunto nessun attributo né metodo;

- *Rule*: anche questa classe è stata realizzata esattamente come proposta nel capitolo 2 in occasione della presentazione del diagramma delle classi componenti il framework. Questo è stato possibile anche perché tale classe è soprattutto una

struttura dati, e non richiede operazioni particolari al suo interno (solo i metodi get e set per gli attributi);

- *Composer*: la classe con lo scopo di convertire in un formato standard (comprensibile dalle interfacce) i suggerimenti da proporre all'utente rispecchia la proposta effettuata in relazione al framework. Il metodo principale rimane *CreateSuggest*, il quale ha lo scopo di creare il suggerimento sulle azioni da eseguire;
- *FieldNames*: questa è l'unica novità rispetto alla struttura precedentemente mostrata. Come già detto si tratta di una classe accessoria che permette di ottenere delle descrizioni partendo da una chiave e il contrario, relativamente ad alcune tabelle presenti nel database. Queste tabelle sono accessorie al database, ovvero non rappresentano direttamente dati, ma servono per la loro visualizzazione all'interno di 3wGis. Si ritiene poco utile ai fini della tesi un'analisi approfondita di questa classe in relazione a metodi ed attributi (il legame con il caso specifico è troppo forte: non sono presenti concetti utili per lo scopo della tesi).

3.2.2 Analisi del client (oggetto COM)

Dopo aver analizzato il lato server dell'applicazione (quello più corposo) si passa ora ad un esame della componente client; in questo paragrafo si discute solamente di come sia stato implementato il connettore fra le interfacce (di osservazione e del modulo induttivo) ed il server, analizzando nel seguente la realizzazione delle interfacce.

Lo scopo della creazione di questo oggetto COM è quindi solamente di permettere la comunicazione attraverso l'utilizzo dei Sockets: per questo motivo la composizione del modulo realizzato è piuttosto semplice, come mostrato in figura 3.5.

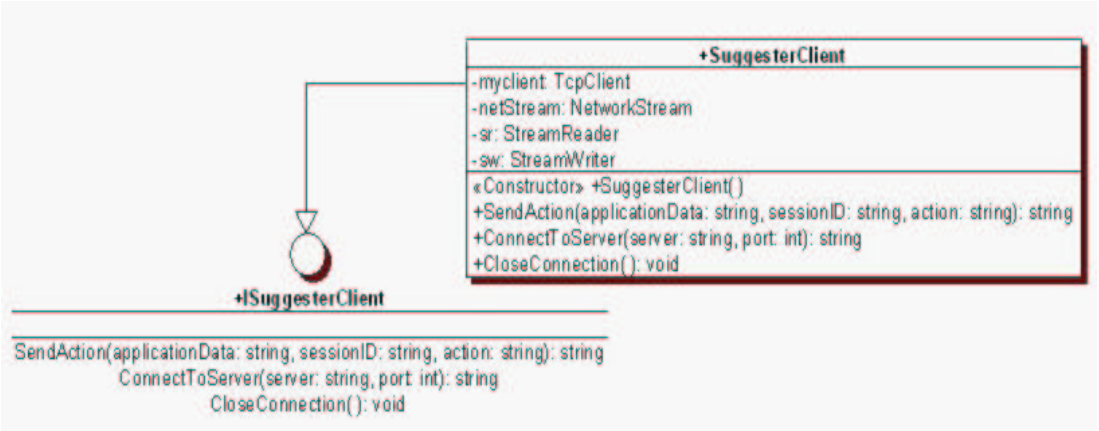

Figura 3.5: Diagramma delle classi relativo al client (oggetto COM)

La caratteristica principale degli oggetti COM sta nel fornire un'interfaccia attraverso la quale è possibile chiamarne i metodi dall'esterno. Proprio per questo motivo è presente una classe di interfaccia chiamata *ISuggesterClient* ed una che la implementa detta *SuggesterClient*. L'interfaccia non richiede spiegazioni, dato che l'implementazione dei metodi è effettuata nella classe *SuggesterClient*, che viene ora descritta.

Un primo attributo è *myClient* di tipo *TcpClient*, utilizzato per identificare il tipo di socket inglobato nell'oggetto COM: si tratta di una connessione dal lato client, che cerca di comunicare con un server (la cui caratteristica è di rimanere in attesa di connessioni). Esistono poi tre variabili più legate alla comunicazione tramite flusso di informazioni (stream), ovvero *netStream* di tipo *NetworkStream* ad indicare la comunicazione in generale fra client e server, *sr* di classe *StreamReader* e *sw* di tipo *StreamWriter*, per la gestione rispettivamente del flusso di dati in lettura ed in scrittura verso il server.

Oltre al costruttore sono presenti tre diversi metodi (come previsto dall'interfaccia): *ConnectToServer*, *SendAction* e *CloseConnection*. Devono essere effettuate tutte, nell'ordine in cui sono state appena presentate, per svolgere correttamente la funzione dell'oggetto COM. *ConnectToServer* cerca di connettersi al server specificato; *SendAction* invia l'ultima azione effettuata (cioè dell'ultima osservazione), specificando l'applicazione dalla quale proviene, la sessione corrente e l'azione stessa e riceve una stringa contenente eventuali azioni da proporre (suggerimenti). E' possibile che *SendAction* sia utilizzata anche per richieste particolari (come l'invio di una nuova teoria culturale). *CloseConnection* chiude la connessione con il server.

3.2.3 Analisi delle interfacce

Qui verranno analizzate le interfacce applicate al software al fine di osservare le azioni eseguite dall'utente, di proporgli dei suggerimenti e di permettergli la gestione del modulo induttivo (ovvero della teoria culturale). Lo sviluppo è avvenuto utilizzando il linguaggio Asp (che si basa su VBScript, un dialetto di Visual Basic), e non sono presenti vere e proprie classi: il corpo principale dei moduli è composto da un insieme di funzioni racchiuse all'interno di un file con estensione *.inc* (nel caso specifico, infatti l'estensione è indifferente).

L'**interfaccia di osservazione** è composta sostanzialmente da due parti differenti: la prima si occupa di intercettare le azioni effettuate dall'utente, mentre l'altra ha il compito di memorizzare tali osservazioni. Per quanto riguarda l'osservazione del comportamento dell'utente, la soluzione riscontrata è stata l'inserimento, all'interno del codice di 3wGis, di chiamate ad una particolare funzione in corrispondenza dell'esecuzione di azioni da parte dell'utente. La funzione richiamata per gestire le osservazioni è chiamata *logEvent*: essa è implementata nella seconda parte dell'interfaccia di osservazione, ovvero nel

modulo creato chiamato *logFunctions.inc*. Le funzioni presenti in questo modulo sono *logEvent* che, come già detto, gestisce l'osservazione delle azioni eseguite; *saveLogToFile* e *saveLogToDB* sono richiamate proprio da *logEvent* per salvare le osservazioni su file e su database. E' possibile utilizzarli entrambi, solo uno dei due o nessuno, a seconda delle impostazioni del software 3wGis. E' poi presente la funzione *logDBVerify*, la quale controlla la presenza di un'apposita tabella nel database dell'applicazione, e la crea se assente o non conforme allo standard. Chiaramente è stata implementata una funzione per connettersi al server tramite l'oggetto COM: essa è chiamata s*endLogToSuggesterServer*, ed è invocata da *logEvent* nel caso in cui si stabilisca che l'applicazione debba essere connessa al server (tramite un'impostazione di 3wGis).

Sempre relativamente alla funzione *logEvent* è da considerare la funzione svolta dal **SuggestFormatter**, ovvero dalla parte del composer con il compito di formattare le azioni proposte in una forma comprensibile dal software 3wGis. Esso gestisce la stringa contenente le azioni suggerite ritornata proprio da *logEvent*. Ogni volta in cui viene memorizzata l'esecuzione di un'azione, viene gestita di seguito la presenza di eventuali suggerimenti. Per formattare il suggerimento è richiamata una funzione presente all'interno del file modulo *Suggester.inc* chiamata *GetSuggestion*, alla quale sono passati come parametri la pagina Html (o Asp) di provenienza dell'osservazione e la lista delle azioni proposte. A seconda del tipo di azione viene richiamata un'altra funzione presente nel modulo: sono implementate ad esempio *SuggestForDetails*, per facilitare la visione dei dettagli di un determinato record, *SuggestForSearch* per aiutare nella ricerca, …

Per quanto riguarda la **gestione del modulo induttivo**, sono state create alcune pagine web da collegare a 3wGis. Le funzionalità offerte riguardano non solo il modulo induttivo, ma tutto ciò che è legato alla gestione della teoria culturale. E' infatti possibile visualizzare le regole attive in un determinato momento, modificarne antecedente e conseguente, eliminarne o aggiungerne (sia manualmente che attraverso l'utilizzo del modulo induttivo), ordinare in modo diverso le regole attive. Tutto questo permette di soddisfare appieno i requisiti dell'amministratore di sistema, per quanto riguarda la teoria culturale (si veda figura 3.1).

3.3 Funzionamento del sistema: azioni attese

In questo paragrafo è esaminata in dettaglio una delle attività più importanti dell'intero sistema, ovvero la proposizione delle azioni attese all'utente. Questa fase non è stata descritta né in occasione della presentazione del framework né nelle prime sezioni di questo capitolo: si analizza qui l'intero processo, partendo dall'invio delle osservazioni (che avviene nell'interfaccia observer), per terminare con la formattazione dei suggerimenti (effettuata dalla componente client del composer).

E’ possibile distinguere in questo processo cinque distinte fasi, nelle quali operano sia la parte client che quella server (non necessariamente entrambe in ogni fase):

1. *Invio e ricezione dell’osservazione*
2. *Scelta dell’eventuale regola da facilitare*
3. *Proposta delle azioni attese*
4. *Creazione del suggerimento in formato standard*
5. *Formattazione del suggerimento*

3.3.1 Invio e ricezione dell’osservazione

In questo primo passo operano sia il client che il server: innanzi tutto sarà compito del client osservare il comportamento dell’utente nell’utilizzo del software. Dopo aver identificato l’azione eseguita è, l’interfaccia di osservazione provvede ad inviarla al server tramite l’utilizzo dell’oggetto COM con il compito di gestire la comunicazione Tcp/Ip.

Il server (in particolare la classe *SuggesterSrv* nel metodo *listenForClients*) attende la connessione di clients attraverso l’utilizzo di un ciclo: quando la richiesta di connessione arriva (la funzione *ConnectToServer* di *SuggesterClient* è eseguita) il compito passa alla classe *ClientManager* ed in particolare al metodo *ManageClSrv*, il quale attende la ricezione di dati (che possono essere sia l’invio di osservazioni che la richiesta di particolari servizi) da parte del metodo *SendAction* del *SuggesterClient*. Nel caso in cui i dati vengano identificati come osservazioni (lo si può dedurre dal formato della stringa inviata), si passa alla loro gestione, descritta nel successivo paragrafo. Il primo passo è mostrato in figura 3.6, evidenziando sia gli elementi del *server* che quelli del client.

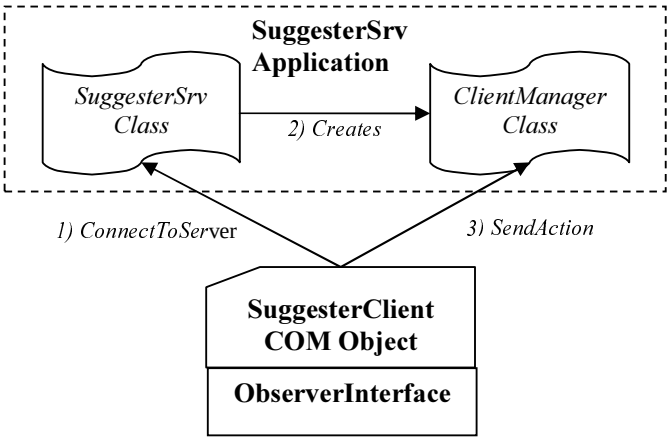

Figura 3.6: Invio e ricezione dell’osservazione

3.3.2 Scelta dell'eventuale regola da facilitare

Il secondo passo è compito esclusivamente della parte server del sistema. Si è nella situazione in cui è stata appena ricevuta l'osservazione, e bisogna determinare l'esistenza di una qualche regola appartenente alla teoria culturale il cui antecedente venga soddisfatto dall'osservazione (cioè dall'azione appena eseguita). Per fare questo è ovviamente necessario che in precedenza la teoria culturale sia stata inizializzata correttamente (in caso contrario la regola soddisfatta potrebbe risultare inutile). Allo scopo di determinare quale sia la prima regola appropriata viene chiamato il metodo *GetRuleFor* della classe *Rules*: il primo parametro è l'osservazione stessa, mentre il secondo è un intero passato per riferimento. Lo scopo di questo valore è permettere la scelta di una regola differente, nel caso in cui, nelle fasi successive del processo, si riscontrino dei problemi (ad esempio non si riescano a determinare con esattezza i parametri del conseguente della regola relativamente all'osservazione, cioè l'azione da proporre). Il metodo *GetRuleFor* scorre la struttura dati contenente le regole (l'array di oggetti *Rule* della classe *Rules*) partendo da quello presente nella posizione rappresentata dall'indice, ritornando la regola e l'indice della stessa.

3.3.3 Proposta delle azioni attese

E' la fase più complessa dell'intero processo di facilitazione dell'utilizzo del software. Per quanto riguarda la comunicazione fra diverse classi la situazione è però molto semplice, dato che questo passo avviene completamente all'interno di *Actions*. E' possibile distinguere due diverse fasi: l'individuazione delle azioni attese ed il loro ordinamento (in ordine di probabilità).

Relativamente alla proposizione delle azioni ritenute probabili il compito è svolto dal metodo *ProposeActions* di *Actions*. Al suo interno contiene però chiamate a diversi altri metodi, utilizzati per lo svolgimento di singole operazioni. I suoi parametri sono *actionSessionCode*, *actionToBeAdded* e *thisRule*, ad indicare rispettivamente il codice della sessione relativa all'azione eseguita, l'osservazione stessa e la regola che si tenta di soddisfare. Il valore di ritorno è un array di stringhe, contenente i suggerimenti. Inizialmente viene verificata la presenza di valori da calcolare nel conseguente della regola individuata (si usa *HasNoQuestionMarks*). Nel caso in cui non ne sia presente alcuno significa che vi sono solamente costanti o valori nulli nel conseguente, e che si può quindi determinare univocamente la struttura del conseguente relativo all'osservazione (l'azione attesa) attraverso il metodo *CompleteConsequentConstants*. Nel caso contrario si deve invece proseguire diversamente: bisogna determinare in quali scene passate la regola sia stata soddisfatta e quanta sia la loro similitudine con la scena corrente. Si individua la sessione relativa all'azione corrente, utilizzando il parametro *actionSessionCode* nello scorrere l'array della classe *Actions*, in modo da identificare la sessione attraverso un

indice. Si effettua poi un ciclo all'interno dell'array *loggedSession* (il contenuto delle sessioni memorizzate), partendo dal primo elemento (in ordine cronologico), esaminando una sessione per volta. Scorrendo il contenuto delle sessioni si verifica la presenza di scene in cui (in passato) sia stata soddisfatta la regola (attraverso il metodo *IsRuleSatisfied*). Nel caso in cui si individui una scena che soddisfi tale regola si passa alla determinazione del coefficiente di similarità con quella corrente: questo è fatto conteggiando il numero di azioni simili (attraverso *AreSimilar*) andando a ritroso partendo dal punto in cui l'antecedente della regola è stato soddisfatto nella situazione precedente ed in quella attuale; chiamiamo tale coefficiente *similarityIndex*. Per ogni scena simile si determina il contenuto dei parametri del conseguente (dell'azione da proporre), basandosi sull'osservazione, sulla regola, e sul conseguente della scena simile (attraverso il metodo *GetCompleteConsequent*). Ogni conseguente individuato è aggiunto poi alla lista, mantenendo un formato *sessionCode / similarityIndex / inSessionIndex / action*, dove i parametri rappresentano il codice della sessione in cui è stata individuata la similitudine, il coefficiente di similarità, l'indice dell'azione proposta nella sessione ed il suggerimento. In figura 3.7 è mostrato lo pseudo-codice di questo algoritmo.

```
if HasNoQuestionMarks(thisRule.GetConsequent()) then
      return "0/1/1" + CompleteConsequentConstants(thisRule,
      actionToBeAdded);
end if
int indexOfAction = 0;
for all session ∈ sessionCode) do
      if session==actionSessionCode then exit for; end if
      indexOfAction++;
end for
int k = firstElement;
do
      int index = 0;
      for all action ∈ loggedSession[k] do
            if IsRuleSatisfied(loggedSession[k], index, thisRule) then
                  similarityIndex = GetSimilarity(loggedSession[k],
                  loggedSession[indexOfAction]);
                  returnArray.Add(GetCompleteConsequent(actionToBeAdded,
                  thisRule, actionString));
            end if
      end for
      if (k==(firstElement+effectiveLength-1)%length) then
            exit do;
      end if
      index = index + 1;
while (1)
return returnArray;
```

Figura 3.7: Algoritmo di proposta azioni attese

In relazione all'ordinamento si è adottato un algoritmo simile a quello proposto nell'analisi del framework. Viene descritto nei dettagli, presentandone lo pseudocodice in figura 3.8. Il metodo si chiama *OrderActions* e fa parte di *Actions*. Le chiavi individuate per l'ordinamento nel capitolo 2 sono il coefficiente di similarità, il numero di occorrenze e l'ordinamento temporale delle azioni eseguite. Nell'implementazione si utilizza un ordinamento che sfrutta solamente le prime due chiavi, tralasciando il coefficiente temporale, visto che l'utilizzo di troppe chiavi può rallentare il sistema nella presentazione dei suggerimenti.

```
string [] actions="", [] similarity="";
for all action ∈ proposedActions do
      actions.Add(GetAction(action));
      similarity.Add(GetSimilarity(action));
end for
InitializeArrays(countedActions, countedActionsKey1, countedActionsKey2);
int ind = 0;
for all action ∈ actions do
      int pos = PositionInArray(thisRecord, countedActions);
      countedActions[pos]=thisRecord;
      if countedActionsKey1[pos] < actions[ind].ToInt() then
            countedActionsKey1[pos]=actions[ind].ToInt();
      end if
      countedActionsKey2[pos] = countedActionsKey2[pos] + 1;
end for
tempArray = countedActionsKey1;
Array.Sort(tempArray, countedActions);
Array.Sort(countedActionsKey1, countedActionsKey2);
firstNonZeroElement = GetFirstNonZeroElement(countedActionsKey1);
for all action in countedActions do
      if index < countedActions.Length-1 then
            if (countedActionsKey1[index]==countedActionsKey1[index+1])
                  endIndex = index + 1;
            else  Array.Sort(countedActionsKey2, countedActions,
                  beginIndex, endIndex-beginIndex+1);
                  endIndex = beginIndex = index +1;
            end if
      else if (index!=beginIndex) {
                  endIndex = index;
                  Array.Sort(countedActionsKey2, countedActions,
                  beginIndex, endIndex-beginIndex+1);
      end if
end for
return Array.Reverse(countedActions).SubArray(0, countedActions.Length-
   firstNonZeroElement);
```

Figura 3.8: Algoritmo per l'ordinamento delle azioni proposte

Per quanto riguarda il funzionamento, innanzi tutto si estraggono le azioni con i rispettivi coefficienti di similarità dall'array ottenuto dal metodo *ProposeActions*; si riempiono poi degli array relativi alle azioni ed alle chiavi (*countedActions, countedActionsKey1,*

countedActionsKey2). Si procede poi con l'ordinamento in base alla chiave principale, per proseguire esaminando la chiave secondaria nei casi in cui vi sia ancora una situazione di parità (delle scene hanno lo stesso coefficiente di similarità). Per fare questo bisogna scorrere all'interno dell'array ordinato in base alla chiave primaria, individuare gli estremi delle sequenze con ugual similarità, e provvedere al loro ordinamento. L'unico parametro è l'array ottenuto da *ProposeActions*, ed il valore di ritorno è un array ordinato.

3.3.4 Creazione del suggerimento in formato standard

Anche questo passo è interamente realizzato dalla componente server. Viene utilizzata la classe *Composer*, che riceve l'array contenente i suggerimenti ordinati da *OrderActions*. Per eseguire la formattazione del suggerimento in un linguaggio comprensibile dalla componente client è necessario per prima cosa creare un'istanza del modulo *Composer* chiamandone il costruttore (i cui parametri sono l'azione eseguita, la lista dei suggerimenti e la stringa di connessione al database), invocando poi il metodo *CreateSuggest*. Esso ritorna una stringa (partendo da un array di stringhe), i cui elementi sono divisi da un separatore (non presente nelle azioni): questa soluzione è adottata per facilitare la comunicazione client / server utilizzando una stringa (si sfruttano le classi di tipo *Stream*). Tale metodo non ha molti compiti: nella maggior parte dei casi formatta la stringa partendo dall'array ricevuto in input dal costruttore; vi sono però alcuni casi in cui esso agisce diversamente. Un esempio è costituito dalla proposizione di un'azione il cui significato sia la visualizzazione del l'ultimo record di una tabella nel database. Qui si determina a run-time quale sia l'ultimo elemento della tabella, cercando in tal modo di venire incontro il più possibile alle esigenze dell'utente ed evitando la presentazione di errori (se fosse stato cancellato il record potrebbero esserci dei problemi). Il suggerimento viene ora restituito al client come valore di ritorno del metodo *SendAction* di *SuggesterClient*.

3.3.5 Formattazione del suggerimento

Quest'ultima fase è invece compito della parte client. E' stata restituita la stringa contenente la lista dei suggerimenti, ed il lavoro rimasto consiste solo nel presentarli a video. Per fare questo è stato creato il modulo *Suggester.inc*, il quale, attraverso la funzione *GetSuggestion*, formatta in Html la lista dei suggerimenti. Inoltre, una caratteristica di questo modulo sta nella possibilità di mostrare diversi tipi di suggerimenti a seconda dell'azione da proporre e della pagina (del frame) in cui devono essere visualizzati. Per esempio, se si vuole proporre la navigazione all'interno dell'albero, sono presenti due scelte: nel caso in cui l'ultima azione eseguita (osservata) sia stata relativa alla pagina contenente la visualizzazione ad albero verranno evidenziate le scelte più

probabili proponendo un albero supplementare, altrimenti verrà mostrato un riquadro con la descrizione del suggerimento che permetta l'utilizzo di un link.

In figura 3.9 è mostrata la ricerca tramite albero utilizzante l'interfaccia intelligente.

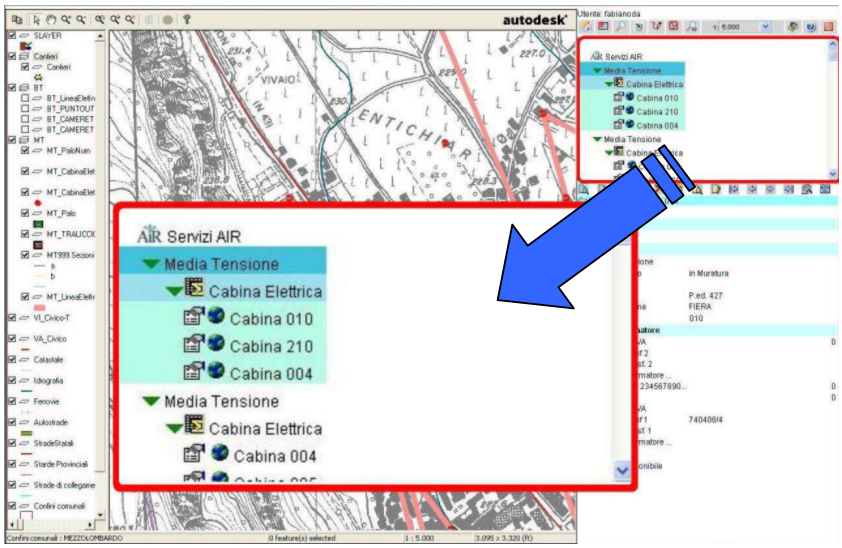

Figura 3.9: Ricerca tramite albero utilizzante interfaccia intelligente

Per facilitare l'utente nella ricerca di un elemento appartenente ad una certa classe il composer fa precedere la visualizzazione classica da un'altra struttura ad albero che visualizza solo gli elementi più probabili nella scena particolare.

In figura 3.10 si mostra la facilitazione relativa alla ricerca tramite query.

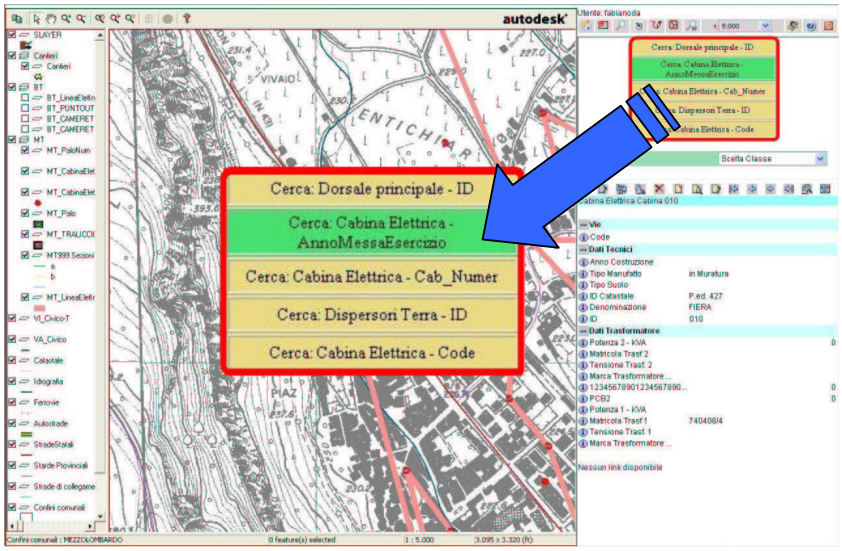

Figura 3.10: Facilitazione della ricerca con query

In questo caso sono proposte alcune alternative relative alle ricerche più comuni effettuate in situazioni simili a quella considerata. Cliccando su una delle proposte viene impostata la struttura della ricerca, permettendo di specificare eventuali parametri aggiuntivi. Nel caso specifico vengono proposte la classe degli oggetti da cercare ed i parametri degli oggetti sui quali si vuole porre una condizione (ad esempio tutte le cabine elettriche la cui data di entrata in funzione sia minore dell'anno 1990).

Capitolo 4

Sperimentazione del software

4.1 Introduzione

Quest'ultimo capitolo della tesi si occupa di valutare l'effettiva efficacia del sistema sviluppato, testando il modulo induttivo ed il composer. Non viene preso in considerazione il modulo di osservazione poiché la sua funzione di mantenere dati in memoria è meno complessa rispetto agli altri elementi di un sistema di supporto alla cultura implicita: è sufficiente che riesca ad intercettare le azioni eseguite dall'utente, e questo requisito è soddisfatto dal modulo sviluppato.

L'attività di valutazione è una fra le più complesse esistenti (non solamente in ambito informatico), poiché richiede l'individuazione di una o più metodologie che permettano di stabilire se il compito di ciò che si valuta sia stato svolto correttamente o meno. Nel caso particolare bisogna esprimere un giudizio sulla teoria culturale ottenuta attraverso l'esecuzione del modulo induttivo e determinare se il composer proponga all'utente ciò di cui realmente necessita.

I paragrafi seguenti utilizzeranno fortemente la codifica delle osservazioni sull'utilizzo del software, ovvero la rappresentazione delle azioni tramite regole. Per questo motivo viene ora presentata una lista delle azioni analizzate in seguito, al fine di permetterne una migliore comprensione:

- *Selection(;a;b;)*: selezione di un elemento sulla mappa da parte dell'utente tramite evidenziazione. I suoi parametri sono la classe dell'elemento (*a)* e l'identificatore dello stesso (*b*);
- *ChangeScale(a;;b;)*: cambiamento della scala di visualizzazione della cartografia. Il parametro *a* indica la pagina Html dalla quale è stata effettuata la variazione, mentre *b* rappresenta il denominatore della scala impostata (ad esempio vale 1000 nel caso in cui la nuova visualizzazione sia 1:1000);
- *MenuServ&Class(a;;;)*: accesso al menu per la visualizzazione ad albero degli oggetti. La pagina che viene aperta è *a*;
- *TreeSearch(a;b;c;)*: esplorazione dell'albero degli oggetti. Il parametro *a* rappresenta la pagina in cui è effettuata la visualizzazione, *b* la classe dell'oggetto

e *c* la sua categoria. La sequenzialità di *b* e *c* non rispecchia correttamente il loro ordinamento gerarchico (*b* è sottoinsieme di *c*), ma è stata utilizzata per mantenere la classe come secondo parametro dell'azione (come in tutti gli altri tipi di azioni che si occupano di oggetti). Nel caso in cui la classe sia vuota significa che si è al primo livello dell'esplorazione dell'albero, altrimenti si è al secondo livello (si è già definita la categoria e bisogna scegliere la classe);

- *Details(a;b;c;)*: visualizzazione dei dettagli di un elemento. Fra i parametri la pagina Html da cui viene eseguita l'azione (*a*), la classe dell'oggetto (*b*) e l'identificatore dello stesso (*c*);
- *MenuSearch(a;;;)*: apertura del menu di ricerca classica, tramite creazione di query;
- *Search(a;b;c;d)*: effettuazione di una ricerca tramite query. I parametri *a, b,c* e *d* indicano rispettivamente la pagina Html di origine, la classe specificata, il tipo di ricerca (che può sfruttare una struttura precedentemente salvata), i campi utilizzati nella query;
- *Zoom(a;b;c)*: zoom su un particolare oggetto nella mappa. Fra i parametri *a* rappresenta la pagina, *b* la classe e *c* l'elemento sul quale è effettuato l'ingrandimento.
- *MenuGlobalView(a*;;;): entrata nel menu che riproduce una miniatura della mappa. Questo è detto anche menu di vista globale;

Si ricorda inoltre la possibilità della presenza, fra i parametri delle azioni, oltre alle costanti (*a, b*, …), dei valori "*" e "*?*". L'asterisco indica, nell'antecedente, la presenza di un qualsiasi valore, mentre il punto di domanda rappresenta, per il conseguente, un parametro il cui valore è da calcolare (può assumere vari valori).

4.2 Valutazione del modulo induttivo

Per riuscire ad esprimere un giudizio sull'efficacia del modulo induttivo si è deciso di mostrarne i risultati dell'esecuzione (ovvero la teoria culturale indotta) impostando diversi parametri ed utilizzando due diverse istanze del software 3wGis. La prima (che viene chiamata *Istanza1*) è utilizzata da un comprensorio di comuni, e contiene classi relative a dati catastali, amministrativi e legati alla toponomastica (vie e civici), ed è stata esaminata durante un periodo in cui i tecnici di I&S la stavano aggiornando inserendo nuovi dati. La seconda (*Istanza2*) è anch'essa riferita ad un comprensorio di comuni, ma ha informazioni relative alla rete elettrica a media tensione, alla dorsale del gas ed ai cantieri aperti; nel periodo di esame l'applicazione era già correttamente funzionante, e quindi il suo utilizzo è stato quello tipico. Si è scelto di esaminare due diverse istanze con caratteristiche

differenti per mostrare al meglio il comportamento del modulo induttivo in situazioni diverse.

Una prima serie di test è stata effettuata considerando i seguenti parametri: una soglia minima per la confidenza di 0.45, per il supporto di 0.04 ed il numero di sessioni considerate uguale a 60. In figura 4.1 è mostrato il raffronto fra i risultati ottenuti nell'esecuzione del modulo induttivo nei due casi.

#	*Istanza1*	*Istanza2*
1	Selection (;a;*;) → Selection (;a;?;)	MenuServ&Class (a;;;) → TreeSearch (a;;?;)
2	ChangeScale (a;;*;) → ChangeScale (a;;?;)	TreeSearch (a;;b;) → TreeSearch (a;?;b;)
3		TreeSearch (*;a;*;) → Details (?;a;?;)
4		MenuGlobalView (*;;;)→ Selection (;?;?;)
5		MenuSearch (*;;;)→ Search (?;?;?;?)

Figura 4.1: Esecuzione del modulo induttivo con confidenza 0.45, supporto 0.04 e 60 sessioni

Da questa prima analisi risalta immediatamente il fatto che nell'*Istanza1* sono state individuate poche regole (solamente due). Questo può essere dovuto a molti motivi, fra cui un indice di confidenza troppo alto (vincolo forte nelle situazioni in cui sia assente un comportamento "standard" dopo l'esecuzione di una certa azione), o il limite del supporto elevato (questo parametro impedisce la proposta di suggerimenti per regole che, nonostante possano avere un'alta confidenza, siano poco presenti in percentuale sul totale). Per l'*Istanza2* si è individuato invece un numero di regole più consistente. Si esaminano quindi le regole riscontrate, distinguendo fra le istanze:

- *Istanza1*: la prima regola (la più frequente) dice che la selezione di un elemento di classe *a* implica la selezione di un altro della stessa classe. L'altra è relativa al cambiamento della scala: una variazione è seguita da un'altra. Questo è il comportamento tipico di un utente che inserisce dati, il quale svolge quest'attività off-line agendo manualmente sul database, e vuole poi verificare nella cartografia il successo del proprio lavoro;
- *Istanza2*: la regola più frequente indica che, dopo essere entrati nel menu per la visualizzazione ad albero degli oggetti, si effettua una ricerca relativa alle categorie (di primo livello). Essa è seguita a ruota dal passo logicamente successivo (regola 2), ovvero il passaggio dall'esplorazione di primo livello a quella a secondo (relativa alle classi della categoria selezionata). La numero 3 indica che, una volta effettuata l'esplorazione di secondo livello, l'utente vuole vedere i dettagli di un elemento della ricerca (appartenente alla classe individuata). Queste prime regole rispecchiano appieno il comportamento individuabile nella classica esplorazione ad albero: una ricerca gerarchica dal livello più generale a quello più specifico. La quarta regola individua nella selezione di un elemento l'azione che segue la

visualizzazione del menu di vista globale: si usa la miniatura della mappa per aiutarsi nella selezione di un oggetto in essa presente. L'ultima regola dice che un utente, dopo essere entrato nel menu per la ricerca classica, effettua una query. Questo è chiaramente il comportamento standard nella situazione specifica (determinabile anche senza induzione).

Analizzando i risultati ottenuti da questo primo test si nota come siano state riscontrate le regole più banali e frequenti, specialmente in relazione all'*Istanza1*. Per questo si sono ripetuti i test diminuendo le soglie di supporto e confidenza.

La seconda serie di test viene quindi eseguita mantenendo il numero di sessioni considerate invariato (le ultime 60 cronologicamente), ma diminuendo i valori limite per supporto (ora a 0.3) e confidenza (a 0.02). Ci si aspetta la presentazione di un numero maggiore di regole, dato che con i nuovi parametri l'algoritmo di induzione dovrebbe risultare meno restrittivo. In figura 4.2 sono riassunti i risultati ottenuti.

#	***Istanza1***	***Istanza2***
1	*Selection (;a;*;) → Selection (;a;?;)*	*MenuServ&Class (a;;;) → TreeSearch (a;;?;)*
2	*ChangeScale (a;;*;) → ChangeScale (a;;?;)*	*TreeSearch (a;;b;) → TreeSearch (a;?;b;)*
3	TreeSearch (a;;b;) → TreeSearch (a;?;b;)	*TreeSearch (*;a;*;) → Details (?;a;?;)*
4	MenuGlobalView(*;;;)→ Selection(;?;?;)	*MenuGlobalView (*;;;) → Selection (;?;?;)*
5	Zoom(a;b;*;) → Zoom (a;b;?;)	*MenuSearch (*;;;) → Search (?;?;?;?)*
6	Details (a;b;*;) → Details (a;b;?;)	Search(*;*;*;*)→ MenuSearch (?;;;)
7	MenuServ&Class(a;;;) → TreeSearch(a;;?;)	Selection (;*;*;)→ MenuGlobalView(?;;;)

Figura 4.2: Esecuzione del modulo induttivo con confidenza 0.3, supporto 0.02 e 60 sessioni

Sono evidenziate in corsivo le regole presenti anche nella serie di test precedente. Si può notare come esse siano le prime della lista (le più frequenti); questo deriva dall'ordinamento effettuato in base al supporto (vedi figura 3.5).

Per l'esame delle regole non presenti nel primo test si procede come in precedenza, separando le considerazioni sulle due istanze:

- *Istanza1*: sono state individuate le regole relative all'esplorazione dell'albero, sebbene non completamente. La numero 7 e la 3 indicano rispettivamente il passaggio dal menu alla ricerca di primo livello e da questa al secondo livello; manca invece la proposizione dei dettagli più probabili a partire dall'esplorazione delle classi. La regola 4 indica la selezione di un elemento in seguito alla visualizzazione della miniatura della mappa. Gli altri due vincoli (5 e 6) rappresentano altre operazioni tipiche dell'inserimento dati: lo zoom (o i dettagli) relativi ad un elemento di una classe implica lo zoom (la visione dei dettagli) di un altro appartenente alla stessa classe;

- *Istanza2*: in questa situazione sono state riscontrate solo due nuove regole, anche perché in precedenza ne erano già state trovate cinque. La regola 6 individua un comportamento relativo alla ricerca tramite query: dopo l'effettuazione di una ricerca si ritorna al relativo menu principale. La settima indica il passaggio al menu per la miniatura della mappa dopo aver selezionato un elemento.

Comparando tutti i risultati finora ottenuti, si nota che il tipo di utilizzo del software modifica fortemente la teoria culturale ottenuta. Nel caso dell'*Istanza1* sono state riscontrate regole che indicano una forte ripetitività nel comportamento dell'utente, imputabile all'inserimento dati, e poche connesse alla ricerca. Per quanto riguarda l'*Istanza2* i risultati ottenuti rispecchiano fortemente l'utilizzo tipico atteso dell'applicazione; non per questo è mancato l'adattamento alle situazioni particolari: la regola 7 dell'*Istanza2* non è infatti un comportamento predicibile a priori. In seguito ci si concentrerà solamente sull'*Istanza2*, essendo questa un esempio tipico di utilizzo del software, e si tralascerà la prima istanza, che individua una situazione particolare in cui l'utente possiede già una grande esperienza relativa al software ed all'ambito in cui opera.

Prima di passare all'analisi dell'effettivo utilizzo dei suggerimenti forniti dal software, si prova a modificare il parametro del modulo induttivo rimasto per ora invariato, ovvero il numero di sessioni considerate, portandolo a 10 in relazione all'*Istanza2*. I risultati sono in figura 4.3.

#	***Istanza2***
1	MenuServ&Class (a;;;) → TreeSearch (a;;?;)
2	TreeSearch (a;;b;) → TreeSearch (a;?;b;)
3	TreeSearch (*;a;*;) → Details (?;a;?;)
4	MenuGlobalView (*;;;)→ Selection (;?;?;)

Figura 4.3: Esecuzione del modulo induttivo su *Istanza2*: confidenza 0.3, supporto 0.02 e 10 sessioni

Relativamente all'utilizzo del software attraverso la ricerca ad albero sono state riscontrate le stesse regole dei casi precedenti (regole 1-3). La quarta indica il passaggio dal menu della miniatura della mappa alla selezione di un elemento, ed anch'essa era stata individuata precedentemente; mancano regole relative alla ricerca classica tramite query: ciò significa che l'utilizzo di questa funzionalità nel corso delle ultime dieci sessioni non è stato frequente. La scelta di un numero di sessioni così basso può riservare molte sorprese, soprattutto nei casi in cui l'utilizzo del software sia anomalo (bastano poche sessioni per modificare la teoria culturale), precludendo una facilitazione effettiva dell'utilizzo del sistema. Il numero di sessioni "perfetto" è determinabile solo a run-time, a seconda dell'utilizzo del software.

4.3 Test relativo al composer

Per valutare l'efficacia reale del sistema sviluppato nei confronti degli utilizzatori (i quali non gestiscono il modulo induttivo – al contrario degli amministratori di sistema) viene proposta un'analisi del comportamento degli utenti in seguito all'induzione di una teoria culturale.

Il modulo induttivo è stato eseguito come in figura 4.2 in relazione all'*Istanza2*: la teoria culturale ottenuta è dunque composta da sette regole. Per valutare l'efficacia del composer (ed in parte del modulo induttivo – visto che il composer agisce sulla base delle regole individuate da questo) si è provveduto a modificare alcuni aspetti dell'interfaccia intelligente proposta, di modo che venissero memorizzati in un file di testo opportunamente formattato i suggerimenti proposti e le occorrenze in cui l'utente abbia accettato. L'unico metodo individuato per valutare l'efficacia del composer è la determinazione della percentuale con cui i suggerimenti forniti siano stati recepiti, distinguendo fra i vari tipi di facilitazioni.

E' stata eseguita un'analisi su un numero di azioni prossimo alle 2000 unità, relative a circa settanta sessioni. I risultati ottenuti sono riportati in figura 4.4.

#	Regola proposta	Numero Proposte	Numero Accettazioni	Percentuale di accettazione
1	MenuServ&Class (a;;;) → TreeSearch (a;;?;)	275	233	85%
2	TreeSearch (a;;b;) → TreeSearch (a;?;b;)	233	178	76%
3	TreeSearch (*;a;*;) → Details (?;a;?;)	200	67	33%
4	MenuGlobalView (*;;;)→ Selection (;?;?;)	140	38	27%
5	MenuSearch (*;;;)→ Search (?;?;?;?)	58	47	81%
6	Search(*;*;*;*)→ MenuSearch (?;;;)	47	11	23%
7	Selection (;*;*;)→ MenuGlobalView(?;;;)	156	25	16%
	Totale	**1109**	**599**	**54%**

Figura 4.4: Efficacia del composer relativamente all'Istanza2

Una prima considerazione è effettuabile in relazione al numero totale di proposte presentate: esso rappresenta circa il 55% della quantità di azioni considerate (1109/~2000). Questa percentuale è piuttosto elevata, poiché vuol dire che in oltre metà dei casi l'interfaccia intelligente modifica l'ambiente proponendo un suggerimento per l'utente.

Vengono ora esaminate le singole regole, per determinarne l'accettazione da parte dell'utente. Per quanto riguarda la regola 1, la frequenza con cui è stata proposta è la maggiore in assoluto: ciò accade se l'utilizzo del software dopo l'attivazione della teoria culturale e del composer rispecchia quello precedente l'induzione (visto che le regole sono presentate in base al supporto); inoltre anche la percentuale di accettazione è elevata:

nell’85% dei casi in cui è stata proposta l’utente l’ha accettata. Anche per la regola 2 la situazione è simile: viene proposta frequentemente ed accettata molto spesso (76%). Questo risultato è facilitato da due fattori: il primo è che l’interfaccia del composer, per la ricerca ad albero, propone una lista di varianti del conseguente anziché un solo suggerimento (permettendo maggiore scelta); il secondo è la presenza di un numero ridotto di classi e categorie (quasi tutte quelle più frequenti vengono elencate nella lista dei suggerimenti). La regola seguente (3) è il passo successivo all’esplorazione del secondo livello dell’albero: la visualizzazione dei dettagli. La percentuale di accettazione è decisamente inferiore (33%), perché la quantità di oggetti appartenenti ad una stessa classe è elevata e, nonostante qualche volta l’utente voglia agire su elementi già visualizzati o da lui o da altri utenti simili (e quindi suggeribili), in generale questo non è detto (vorrà visualizzare nuovi oggetti). La quarta regola è anch’essa soddisfatta solo in pochi casi (27%); è relativa alla selezione di un elemento dopo la visione della miniatura della mappa: la bassa percentuale è imputabile al possibile utilizzo del menu di vista globale solo per effettuare uno zoom o per spostarsi nella mappa. Per la quinta regola il discorso è simile all’esplorazione ad albero, con una percentuale dell’81%. Anche qui infatti viene proposta una lista di scelte per impostare la propria ricerca; una percentuale così alta può comunque significare un utilizzo di pochi tipi diversi di ricerca. Le ultime due regole (il ritorno al menu dopo una ricerca e l’utilizzo del menu di vista globale dopo la selezione di un elemento) sono accettate piuttosto raramente; uno dei fattori delle basse percentuali può essere il fatto che cliccare su questi suggerimenti o arrivare ai menu in modo tradizionale comporta comunque un solo passo (un click) e quindi non vi sono grandi vantaggi nell’utilizzare il suggerimento fornito.

Considerando i dati generali si nota una buona percentuale di accettazione dei suggerimenti proposti: nel 54% dei casi l’utente ha ritenuto positivo l’aiuto fornito dall’interfaccia intelligente utilizzandolo. D’altra parte è difficile riuscire a comprendere se il suggerimento sia stato accettato perché il nuovo ambiente abbia indotto alla scelta di quell’azione o perché fosse realmente ciò che l’utente volesse fare (difficoltà di valutazione). Il giudizio complessivo relativo ai dati ottenuti è comunque positivo, specialmente in relazione a quei suggerimenti che vengono formattati in maniera particolare dall’interfaccia del composer (l’esplorazione ad albero e la ricerca tramite query).

Conclusioni

Questa tesi ha mostrato come sia possibile realizzare delle interfacce intelligenti basate sulla cultura implicita, applicabili ai software per la gestione dell'overload di informazioni. Il lavoro si è svolto in due fasi: una prima analisi generale del problema e la realizzazione di una soluzione.

Si è iniziato analizzando gli elementi fondamentali del problema: l'overload di informazioni viene risolto efficacemente tramite la creazione di appositi software, che forniscono però spesso un'interfaccia utente insoddisfacente. Per porre rimedio a questa lacuna esistono molte tecniche, fra cui una soluzione valida è rappresentata dalle interfacce intelligenti. Le discipline utilizzate dalle interfacce intelligenti per il proprio funzionamento sono svariate, fra cui hanno rilevanza data mining e collaborative filtering. Ad esse si aggiunge la cultura implicita, che è una generalizzazione del concetto di collaborative filtering. Per la realizzazione di interfacce intelligenti si ritiene la cultura implicita lo strumento più idoneo, essendo più generale del collaborative filtering e permettendo la facilitazione dell'utilizzo in tempo reale, al contrario delle tecniche di data mining che agiscono offline.

Nella seconda fase è stata proposta una soluzione al problema delle interfacce complesse basata sulla cultura implicita. Innanzi tutto è stata progettata e sviluppata un'architettura generale, individuando varie alternative per l'effettiva implementazione di sistemi basati su essa. Si è poi realizzata un'implementazione del framework che si adattasse ad un software specifico (un'interfaccia web per l'accesso a banche dati cartografiche chiamata 3wGIS). Il modulo realizzato ha lo scopo di facilitare le ricerche dell'utente, proponendo lo strumento di ricerca più adatto ed impostandone, qualora possibile, i parametri di funzionamento. Infine il software è stato sperimentato al fine di evidenziarne pregi e difetti.

Dal lavoro svolto è possibile trarre diverse conclusioni relative allo sviluppo di interfacce intelligenti: innanzi tutto si è presentato in più occasioni un rapporto conflittuale fra generalità del software e velocità dello stesso. Esempi di questa situazione sono riscontrabili in relazione al formalismo della teoria culturale ed al numero di osservazioni da conservare in memoria per modulo induttivo e composer. Un secondo tipo di problematica è legato alla difficoltà nella proposizione di un'architettura generale: è necessario permettere estensibilità alla soluzione proposta affinché possa adattarsi con successo a diverse situazioni. Per questo motivo il framework descritto è molto generale, e

sono state fornite diverse alternative per la sua effettiva implementazione (una delle quali è l'applicativo effettivamente sviluppato).

In prospettiva futura, sono molti i campi in cui ci si possa concentrare per estendere la conoscenza dell'argomento esaminato. In primo luogo sarebbe opportuno verificare quale sia il giusto compromesso fra formalismo e rapidità di esecuzione del software, per riuscire ad individuare una soluzione che non penalizzi l'utente né in relazione alla qualità dei suggerimenti proposti né per ciò che riguarda il tempismo con il quale essi gli vengano presentati (è meglio infatti ottenere un aiuto non preciso che uno fornito in ritardo). Un ulteriore approfondimento è effettuabile nei confronti di come si possano proporre i suggerimenti all'utente (la visualizzazione grafica), aspetto certamente non trascurabile specialmente se le interfacce sono applicate ad un software commercializzato. Altro campo (molto vasto) in cui ci si possa concentrare consiste nella scelta o nella progettazione di nuovi algoritmi e strutture dati per la teoria culturale, il modulo induttivo ed il composer. Infine sono sicuramente utili progressi nello studio di metodologie per la valutazione dell'efficacia dei suggerimenti, in modo da poter fornire a future applicazioni simili i mezzi per determinare la qualità delle soluzioni utilizzate.

Bibliografia

1. P. Atzeni, S. Ceri, S. Paraboschi, R. Torlone. *Basi di dati (seconda edizione)*, McGraw-Hill Italia. Pagg. 498-503;
2. E. Blanzieri, P. Giorgini. *From Collaborative Filtering to Implicit Culture: a general agent based framework*. In Proceedings of the Workshop on Agent-Based Recommender Systems (WARS) in Autonomous Agents (Agents2000, ACM). Barcelona, Spain, June 4, 2000;
3. E. Blanzieri and P. Giorgini. *Implicit Culture for Information Agents* . Under scrutiny – 2001;
4. E. Blanzieri, P.Giorgini and F. Giunchiglia. *Implicit Culture and Multiagent Systems*. In Proceedings of the International Conference on advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2000). L'Aquila, Italy, July 31-August 6 , 2000;
5. E.Blanzieri, P.Giorgini, F.Giunchiglia and C. Zanoni. *A Multi-Agent System for Knowledge Management based on the Implicit Culture Framework*. WOA 2002, Università di Milano – Bicocca, 18-19 novembre 2002;
6. E. Blanzieri, P. Giorgini, F. Giunchiglia, C. Zanoni. *Personal agents for implicit culture support, 2002*;
7. E. Blanzieri, P. Giorgini, P.Massa and S. Recla. *Collaborative Filtering via Implicit Culture Support.* Preliminary version submitted to UM2001 (11-11-2000);
8. E. Blanzieri, P. Giorgini, P.Massa and S. Recla. *Implicit Culture for Multi-agent Interaction Support.* Proceedings of Sixth International Conference on Cooperative Information Systems (CoopIS 2001), Trento - Italy, Sept 2001;
9. E. Blanzieri, P. Giorgini, P.Massa and S. Recla. *Information access in Implicit Culture Framework.* . Proceedings of the ACM SIGIR Workshop on Recommender Systems, New Orleans, LA - USA, Sept 13, 2001;
10. E. Blanzieri, P. Giorgini, P.Massa and S. Recla. *Data Mining, Decision support and Meta-Learning: towards an Implicit Culture architecture for KDD*. Proceedings of the Workshop on Positions, Developments and Future Directions in connection with IDDM-2001 , Sept 2001;
11. Bologna Knowledge Discovery/Data Mining Center – CINECA. *Data Mining*, http://open.cineca.it/datamining/dmCineca;

12. G. Brajnik, S. Mizzaro, C. Tasso. *La valutazione di interfacce intelligenti per il reperimento di informazioni*. Dipartimento di Matematica e Informatica Università di Udine, 1996;

13. J. S. Breese, D. Heckerman, C. Kadie*, Empirical Analysis of Predictive Algorithms for Collaborative Filtering*. Microsoft Research – Microsoft Corporation, May 1998, revised October 1998;

14. Alexander Chislenko. *Automated Collaborative Filtering and Semantic Transports*, October 1997;

15. M. Fowler, K. Scott. *UML Distilled: A brief guide to the Standard Object Modeling Language (2nd Edition)*, Addison-Wesley Pub Co, August 1999;

16. K. Gajos and A. Kulkarni. *FIRE: An Information Retrieval Interface for Intelligent Environments*. Artificial Intelligence Laboratory, Massachussets Institute of Technology, 2001*;*

17. N. Good, J.B. Schafer, J.A. Konstan, A. Borchers, B. Sarwar, J. Herlocker, J. Riedl. *Combining Collaborative Filtering with Personal Agents for Better Recommendations*. GroupLens Research Project, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, 1999;

18. J. Han. *Data Mining*, in J.Urban and P. Dasgupta (eds.), *Encyclopedia of Distributed Computing*, Kluwer Academic Publishers, 1999;

19. J. Han and M. Kamber. *Data mining: Concepts and Techniques*, The Morgan Kaufmann Series in Data Management Systems, August 2000;

20. M. Hegland. *Algorithms for Association Rules*, Australian National University Canberra;

21. J.L. Herlocker, J.A. Konstan, A. Borchers, J. Riedl. *An algorithmic framework for performing collaborative filtering*. Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval, Pages 230-237, 1999;

22. K. P. Joshi. *Analysis of Data Mining Algorithms*, 1997;

23. E. Lipp. *Picturing the Data Explosion*, article on Database Trends and Application Site, November 2002;

24. S. Macho, M. Torrens, B. Faltings. *A Multi-Agent Recommender System for Planning Meetings*. Artificial Intelligence Laboratory, Swiss Federal Institute of Technology, Lausanne, 2000.

25. P. Massa. *Dal collaborative filtering alla cultura implicita: analisi e sviluppo di un sistema di supporto*. Tesi di laurea in Informatica, Università degli Studi del Piemonte Orientale "Amedeo Avogadro", A.A. 1999-2000;

26. S. Mizzaro. *Intelligent Interfaces for Information Retrieval: A Review*. Department of Mathematics and Computer Science, University of Udine, 1996;
27. D.W. Oard, J. Kim. *Implicit Feedback for Recommender Systems.* Digital Library Research Group, College of Library and Information Services, University of Maryland, College Park, 1998;
28. Object Management Group. *Unified Modeling Language 1.4 Specifications*, September 2001;
29. R. Rastogi and K. Shim. *Scalable algorithms for mining large databases* (Powerpoint presentation). Lucent Bell laboratories. Presented at CIKM'98, ICDE'99 and SIGKDD 99;
30. M.M. Recker, A. Walker, D.A. Wiley. *An interface for collaborative filtering of educational resources*. Department of Instructional Technology Utah State University - Digital Learning Environments Research Group Department of Instructional Psychology & Technology Brigham Young University, 2001;
31. S. Recla. *Progetto e sviluppo di un Sistema di Supporto all'interazione multiagente basato su Cultura Implicita.* Tesi di laurea in Matematica, Università degli Studi di Trento, A.A. 1999-2000;
32. P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, J. Riedl. *GroupLens: An Open Architecture for Collaborative Filtering of Netnews*. MIT Center for Coordination Science and University of Minnesota – Department of Computer Science. From Proceedings of ACM 1994 Conference on Computer Supported Cooperative Work, Chapel Hill. Pages 175-186;
33. C. Z. Robertson. *Collaborative Techniques in Information Systems*, 2001;
34. Rudjer Boskovic Institute, Data Mining Server. *DM Tutorial*, http://dms.irb.hr/tutorial/tut_main.php, 2001;
35. B. Sarwar, G. Karypis, J. Konstan, J. Riedl. *Item-based Collaborative Filtering Recommendation Algorithms*. GroupLens Research Group/Army HPC Research Center, Department of Computer Science and Engineering, University of Minnesota, Minneapolis. Appears in WWW10, May 1-5, 2001, Hong Kong;
36. J.B. Schafer, J. Konstan, J. Riedl. *Recommender Systems in E-Commerce*. GroupLens Research Project, Department of Computer Science and Engineeirng, University of Minnesota, Minneapolis, 1999;
37. *SearchDatabase,* http://searchdatabase.techtarget.com;
38. P. Simbi. *Princeton Course Recommender*, Computer Science Senior Thesis, Princeton University;
39. D. Stenmark. *Capturing Tacit Knowledge using Recommender Systems*, Volvo Information Technology, 1999;

40. K. Thearling. *Introduction to Data Mining (PowerPoint presentation)*, http://www.thearling.com/dmintro/dmintro.htm;
41. J. Tullio. *Intelligent User Interfaces*. Course slides from Everyday Computing Lab, Spring 2003;
42. Two Crows Corporation. *Introduction to Data Mining and Knowledge Discovery* (3rd Edition), 1999;
43. J. Ullman. *Association-Rule Mining*, Data Mining Lecture Notes, University of Stanford;
44. N. Veerman. *Intelligent Interfaces: overview and application* - Paper for Human Computer Interaction, January 2001;
45. Annika Wærn. *What is Intelligent Interface?* Notes from an introduction seminar, March 1997