Università degli Studi di Torino
Dipartimento di Informatica

Corso di Laurea in Informatica

Tesi di Laurea in Informatica

Diagnosis by Numbers: Uno strumento basato su logiche descrittive, "tipicalità" e probabilità

Relatore:
Prof. Gian Luca Pozzato

Candidato:
Damiano Gianotti

Sessione Ottobre 2019
a.a 2018/2019

Diagnosis by Numbers

Uno strumento basato sulle logiche descrittive, "tipicalità" e probabilità

Damiano Gianotti

Abstract

Lo stage si pone l'obiettivo di realizzare **DbN**, un tool di supporto per la *diagnosi differenziale*, con applicazione ad un caso di studio in ambito medico, basato su una logica descrittiva con tipicalità e probabilità di avere eccezioni. La logica in questione consente di generare scenari plausibili ma "sorprendenti", che consentiranno di formulare diagnosi non ovvie (iter alternativi potenzialmente *sorprendenti*) e di stimarne la probabilità. Lo strumento potrebbe portare ad ulteriori ricerche, nel caso in cui le spiegazioni più plausibili non siano quelle corrette e mostrare possibili scenari non banali.

Il lavoro iniziale è stato lo studio delle Logiche Descrittive, una famiglia di linguaggi formali utilizzati per esprimere (rappresentare) la conoscenza in un dominio specifico (detto mondo). Sono quindi alla base dei linguaggi impiegati per lo sviluppo di ontologie nel Web Semantico, come il Web Ontology Language (OWL).

In seguito, il progetto è partito come estensione della tesi "Logiche descrittive della tipicalità: sviluppo di uno strumento per il ragionamento sulle probabilità di eccezioni" [11]. Infatti, dopo averne studiato le caratteristiche importanti, si è cercato di costruire sopra un diverso sistema di generazione degli scenari e di rafforzarne le componenti di ragionamento, fortemente orientati all'implementazione dell'esempio sei di "Typicalities and Probabilities of Exceptions in Nonmotonic Description Logics" [8]. Raggiunto questo obiettivo, non banale, si è ottimizzato e pulito il codice e sono state aggiunte funzionalità aggiuntive, come la creazione di grafici interattivi e costi diagnostici.

Struttura della tesi

Di seguito, il piano di quest'opera.

- Il primo capitolo dà una breve infarinatura sulle fondamenta del progetto e descrive schematicamente il lavoro che è stato svolto.
- Il secondo capitolo invece fornisce alcune tra le nozioni teoriche più importanti che servono per comprendere i meccanismi su cui si basa DbN
- Il terzo tratta di quali librerie/linguaggi son state/i utilizzate/i e il perché
- Il quarto descrive, in dettaglio, le singole componenti del *software*
- Il quinto racconta dei principali problemi presenti e delle possibili idee risolutive

Indice

1 Introduzione **6**
- 1.1 Ambiente e metodologie di studio . . . 6
- 1.2 Motivazioni del lavoro . . . 7
- 1.3 Spiegazioni . . . 8

2 Logiche Descrittive **9**
- 2.1 Prefazione . . . 9
- 2.2 Caratteristiche e limiti . . . 11
  - 2.2.1 Tipologia ed Ingredienti . . . 11
  - 2.2.2 Monotonicità . . . 11
- 2.3 Il linguaggio base $\mathcal{AL}$ . . . 12
- 2.4 Operatore T . . . 13
  - 2.4.1 Premessa . . . 13
  - 2.4.2 Definizione di T . . . 14
- 2.5 La logica $\mathcal{ALC} + \mathbf{T}$ . . . 16
- 2.6 La logica non monotona $\mathcal{ALC} + \mathbf{T}_{\mathbf{R}}^{RaCl}$ . . . 18
  - 2.6.1 Traduzione dell'operatore T . . . 19
- 2.7 La logica $\mathcal{ALC} + \mathbf{T}_{\mathbf{R}}^{\mathsf{p}}$ . . . 20
  - 2.7.1 Modifiche alla semantica . . . 20
  - 2.7.2 Estensione dell'*Abox* . . . 21
- 2.8 Aggiunte . . . 23

3 Strumenti utilizzati **24**
- 3.1 OWL2 . . . 24
  - 3.1.1 Semantica . . . 24
  - 3.1.2 Caratteristiche . . . 25
  - 3.1.3 Sintassi e Modellazione di Base . . . 25
  - 3.1.4 Classi complesse e implementazione di $\sqcap, \sqcup, \exists$ e $\forall$ . . . 26
  - 3.1.5 OWl2 *Versus* DB e considerazioni finali . . . 27
- 3.2 Owlready2 . . . 27
  - 3.2.1 Tabella di conversione . . . 28
  - 3.2.2 Che cosa posso fare con OWLReady2? . . . 29
  - 3.2.3 Architettura . . . 30
  - 3.2.4 Paragone con precedenti approcci . . . 30
- 3.3 Plotly . . . 30
  - 3.3.1 Perché usare Plotly.py? . . . 30
  - 3.3.2 Che cosa posso fare con Plotly.py? . . . 31

4 Caso d'uso: Strumento per il supporto diagnostico 32
4.1 PEAR - Sintesi 32
4.2 Visione complessiva 33
4.3 Immissione dei dati 34
4.3.1 I documenti in ingresso 34
4.3.2 La classe dedicata alla traduzione 36
4.4 L' amministrazione dell'ontologia 36
4.5 La creazione dei membri tipici e degli scenari 38
4.6 L'inferenza 40
4.7 Analisi del risultato 41
4.8 Il file principale *Main.py* 43

5 Conclusione e sviluppi futuri 44

A Esempio completo 45

Capitolo 1

Introduzione

1.1 Ambiente e metodologie di studio

La ricerca nel settore della rappresentazione della conoscenza si concentra da sempre sulla possibilità di fornire descrizioni ad alto livello di fatti, gerarchie terminologiche e reti concettuali necessarie ad software 'intelligenti', ossia a quelle applicazioni in grado di ricavare conseguenze implicite (talvolta profonde o nascoste) di conoscenze esplicitamente disponibili o facilmente accessibili.

In tal senso le Logiche Descrittive si pongono come miglior risposta per questo di problema poiché riescono a coniugare in modo sapiente l'espressività ed efficienza: man mano che gli studi e le sperimentazioni relative alle Logiche Descrittive progrediscono, le nostre conoscenze e capacità di classificare in modo sottile i vari frammenti dei linguaggi logici si fanno sempre più profonde ed adeguate alle esigenze dei vari ambiti applicativi.

Nel nostro caso, però, si trovano subito delle difficoltà, infatti si presuppone la congiunzione di due requisiti contrastanti : il bisogno composizioni sintattiche (tipiche dei sistemi logici), e la necessità dell'utilizzo della "tipicalitá". Uno dei limiti di queste logiche è che non sono in grado di rappresentare proprietà tipiche e di ragionare sull'eredità rivedibile [8]. Richiamiamo dunque, in maniera informale, un classico esempio proveniente dalla letteratura: immaginiamo di sapere che gli uccelli volano, ma che i pinguini siano uccelli che non volano. Questa base di conoscenza sarebbe consistente solo se non ci fosse neppure un pinguino. Per affrontare questo problema fin dai primi anni '90 sono state approfondite e studiate numerose estensioni non monotone delle logiche descrittive.

Il programma oggetto delle tesi si basa sulla $\mathcal{ALC}+\mathbf{T}^{\mathsf{P}}_{\mathbf{R}}$. Questa, oltre ad avere complessità ExpTime-completa (come la sottostante $\mathcal{ALC}$), combina diverse componenti importanti:

- in primis la logica $\mathcal{ALC}+\mathbf{T}$ dove le proprietà tipiche possono direttamente essere descritte dall'operatore $\mathbf{T}$ di "tipicalità", grazie a cui è possibile esprimere che, per ogni concetto $C, \mathbf{T}(C)$ indica che le istanze di C sono considerate *tipiche* o *normali*. Così una *TBox* potrà contenere inclusioni della forma $\mathbf{T}(C) \sqsubseteq D$ a rappresentare che "i tipici C sono anche D. A differenza del-

la maggior parte delle altre logiche descrittive questa ci permetterà quindi di esprimere e ragionare sulle eccezioni mantenendo una consistenza della base di conoscenza.[2];

- il secondo ingrediente necessario sarà una semantica distribuita, simile a quella utilizzata per le logiche descrittive probabilistiche, conosciuta come DISPONTE. L'idea è quindi quella di aggiungere un'etichetta alle inclusioni che indichi la probabilità di tale fenomeno, per poter esprimere quanto sia possibile che un evento eccezionale si verifichi.

  Con questa estensione è possibile esprimere fatti del tipo:

  $\mathbf{T}(C) \sqsubseteq_p D$ ("abbiamo una probabilità p che un tipico C sia un D")

  direttamente nella base di conoscenza oppure inferire e/o dedurre fatti del tipo

  $p : \mathbf{T}(C)(m)$ ("il membro m è un tipico C con una certa probabilità p")[8];

- il terzo riguarda il rafforzamento della semantica trattato nell'articolo [3] dove gli autori hanno ristretto la consequenzialità logica ad una classe di modelli minimi. L'idea intuitiva è quella di restringere la consequenzialità logica ai modelli che minimizzano le istanze atipiche di un concetto. La logica risultante è $\mathcal{ALC} + \mathbf{T}_{\mathbf{R}}^{RaCl}$ la cui semantica vedremo meglio in seguito.

Se opportunamente uniti otteniamo proprio $\mathcal{ALC} + \mathbf{T}_{\mathbf{R}}^{\mathsf{P}}$ che è caratterizzata dalla *probabilità di eccezionalità* dalla forma:

$$\mathbf{T}(C) \sqsubseteq_p D$$

dove $p \in (0, 1)$ il cui significato intuitivo è:

"normalmente, gli elementi C sono D
e la probabilità di avere elementi C che non sono D è $1 - p$."

1.2 Motivazioni del lavoro

Sul fatto che un calcolatore possa trattare i dati statistici e fare predizioni probabilistiche meglio di un essere umano non possono esserci dubbi: l'essere umano non è tanto portato per il ragionamento statistico, che in genere appare controintuitivo. In questo senso DbN potrebbe diventare un aiuto diagnostico prezioso, uno strumento di consultazione del medico, più efficace di trattati e riviste, ma con il grande limite di essere disumanizzante. Infatti il dato rilevante alla diagnosi si raccoglie nel contesto del rapporto medico-paziente e il paziente non racconterebbe a DbN la sua anamnesi come la racconterebbe ad un medico di sua fiducia.
Va sottolineato che, a causa della logica non standard utilizzata, DbN non è certo allo stato dell'arte ma, al momento, un semplice ma efficace prototipo.

L'aspetto più interessante è il contesto in cui andrebbe ad inserirsi: in effetti già esistono applicazioni per lo smartphone che dovrebbero aiutare il medico nelle decisioni o, almeno, fornirgli spunti di riflessione e di esercizio. Contrariamente, supportare la diagnostica medica è un problema privo di regole esatte e, in fondo, di ampiezza non

limitata che sembra molto al di là delle possibilità di una macchina, ma la presenza di grandi aziende, come **IMB** con **Watson**, ci fa chiedere per quanto tempo questo rimarrà così.

L'ultimo aspetto da considerare è relativo ai costi: per ora DbN è un embrione, un piccolo prototipo e non è nemmeno immaginabile cosa costerebbe produrne e renderne disponibile un numero sufficiente a soddisfare tutte le possibili richieste di consulenza. D'altra parte la macchina ha delle potenzialità, e anche da sola può dare retta a diverse richieste, ma sicuramente necessita ulteriori sviluppi. Il vantaggio nell'uso di un consulente elettronico sarebbe molto grande, sia per l'accuratezza delle diagnosi e delle terapie, sia per le implicazioni medico-legali come elemento di buona prassi.

1.3 Spiegazioni

detto minimamente come DbN opera: che legame c'è con la logica con tipicalità e probabilità? Ecco un'estensione da fare: aggiunga un paragrafo che descriva per sommi capi cosa fa il sistema, anche solo intuitivamente accennando agli scenari e alla diagnosi. Non è neppure detto cosa si intende per diagnosi

Capitolo 2

Logiche Descrittive

In questo capitolo iniziale forniremo una definizione di **DL** (Description Logics), analizzeremo le sue componenti e verranno descritte brevemente le sue estensioni più importanti, utilizzate negli ultimi anni.

2.1 Prefazione

Le logiche descrittive sono una famiglia di formalismi per la rappresentazione della conoscenza, con la capacità di descrivere ciò che è noto in un dominio di applicazione (detto mondo). Tale rappresentazione si fonda su strutture importanti, come grafi o frames, e ha una difficoltà variabile, che dipende dal linguaggio scelto, poiché l'espressività e la complessità computazionale sono direttamente proporzionali. Tipicamente i nodi rappresentano concetti (oggetti), che possono avere proprietà (semplici o articolate) associate. È piuttosto semplice creare una corrispondenza tra i grafi e le **DL**, perché queste ultime sono dotate di predicati facilmente equiparabili alle strutture dei grafi: *predicati* unari corrispondo agli insiemi di individui, *predicati* binari rappresentano relazioni tra singoli e infine un meccanismo di istruzioni d'inclusione per esprimere proprietà appartenenti ai concetti, come, ad esempio, $Scimmia \sqsubseteq Mammifero$.

Ecco alcuni esempi riguardanti individui:

- *Volpe(foxy)*
- *Scappa(foxy)*
- *Gallina(coco)*
- *Ruba(foxy, coco)*
- *Uomo(jhon)*
- *Insegue(jhon, foxy)*

È anche possibile utilizzare l'intersezione di concetti tramite la sintassi $Cane \sqcap Taglia\ Grossa$ per cercare individui che appartengono ad entrambe le categorie.
Ricordiamo che questa tipologia di logiche ha alla base quella del prim'ordine, da

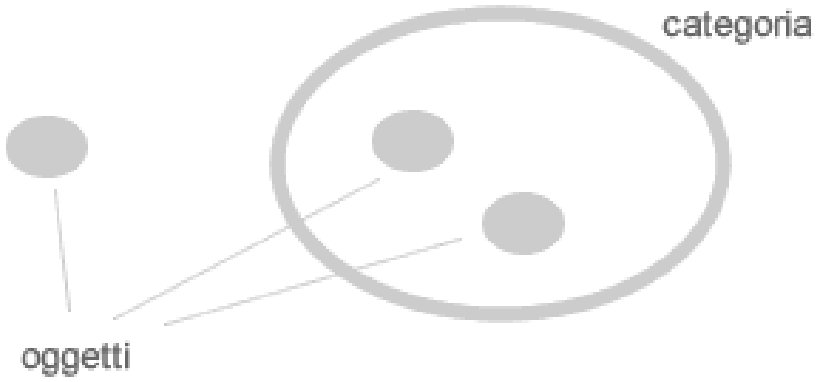


Figura 2.1: Un esempio di grafo

cui eredita la capacità di ragionamento attraverso inferenza, come il *modus ponens*. Infatti se all'insieme precedente aggiungessimo che:

$$Mammifero \sqsubseteq Animale$$

allora da questo potremo inferire che $Volpe \sqsubseteq Animale$.

Per queste e altre caratteristiche peculiari le DL sono ampiamente utilizzate in numerosi sistemi, proprio per il buon compromesso tra capacità di rappresentazione, ragionamento e complessità.
Vediamo, in breve, alcuni dei principali domini applicativi:

Data mining Questo campo, da sempre al centro di molti dibattiti (anche etici), ha come scopo "l'estrazione" di dati grezzi. Successivamente, al fine di classificarli, viene fatto largo utilizzo delle logiche descrittive, per poi, in seguito, attraverso tecniche di ragionamento, fare inferenza su di essi ed ottenere nuova conoscenza.

Medicina Fin dagli anni '80 al centro di molte iniziative è stata la creazione di una grande ontologia delle conoscenza medica, volta ad essere di supporto nelle diagnosi. Per far fronte alla scalabilità della base di conoscenza, si sono spesso utilizzate logiche descrittive basilari. Rappresenta l'ambito di maggiore interesse per questo studio.

Configuration management È uno dei campi di maggior successo: esso include applicazioni che supportano la progettazione di sistemi complessi grazie all'unione di componenti multipli. In particolare attraverso la classificazione di concetti (che possono anche essere basate su modelli object oriented) si può creare una tassonomia che, unita alla possibilità di ragionamento, intrinseca nelle logiche descrittive, permette di trovare con facilità eventuali inconsistenze nel sistema.

Ingegneria del software Uno dei primi domini di applicazione: l'idea alla base era di implementare, attraverso le logiche descrittive, un sistema che permettesse allo sviluppatore di trovare facilmente informazioni rilevanti riguardo ad un sistema particolarmente grande e/o complesso.

2.2 Caratteristiche e limiti

2.2.1 Tipologia ed Ingredienti

Ogni logica descrittiva è caratterizzata da una particolare capacità espressiva che varia a seconda delle possibilità e delle restrizioni imposte nei seguenti elementi:

- $\mathcal{AL}$: indica la logica degli attributi e introduce gli operatori di congiunzione e quantificazione universale ed esistenziale;
- $\mathcal{C}$: descrive la possibilità di usare l'operatore di negazione;
- $\mathcal{S}$: indica la capacità di definire la chiusura transitiva di un ruolo;
- $\mathcal{H}$: fornisce la potenzialità di definire gerarchie tra ruoli;
- $\mathcal{O}$: asserisce la presenza dell'operatore di enumerazione;
- $\mathcal{I}$: permette di riferirsi al ruolo inverso;
- $\mathcal{F}$ $\mathcal{N}$ e $\mathcal{Q}$: caratterizzano le disponibilità di definire, rispettivamente, la cardinalità funzionale, semplice e qualificata (in ordine di espressività crescente);
- $\mathcal{D}$: descrive la possibilità di riferirsi a domini concreti.

2.2.2 Monotonicità

Caratteristica peculiare delle **DL** nella forma di

$$KB \models Q \implies KB \cup \{F\} \models Q$$

ovvero, se da una base di conoscenza **KB** si può concludere un fatto Q, allora lo stesso fatto Q si deduce dalla stessa base di conoscenza arricchita con un nuovo fatto F.

Questa proprietà in sè è molto *importante*, tuttavia non sempre si rivela pratica, anzi, esistono contesti (come il nostro) in cui, più che dare benefici, rende difficile il ragionamento e la deduzione di nuove informazioni. Vediamone il perché con un famoso esempio.
Consideriamo la seguente base di conoscenza o *Knowledge Base* tratta da [11]:

$$Piove \implies Prendo(ombrello)$$

$$Piove$$

si può banalmente concludere che: $Prendo(ombrello)$

Aggiungiamo ora la seguente *formula*

Sono_Squattrinato

La conclusione precedente rimane lecita anche dopo l'arricchimento della **KB**.

Aggiungiamo, invece, questa *espressione*

Ho_Perso(ombrello)

Contro le nostre aspettative/intuizioni la conclusione non cambia, poiché dalle due premesse iniziali si giunge allo stessa conclusione senza che la nuova "conoscenza" influisca sul risultato. Questo semplice modello ci mostra perché è importante, in contesti specifici, avere dei linguaggi e degli automatismi in grado di tenere conto della realtà dei fatti, cioè una logica che possegga una **eredità rivedibile**.

2.3 Il linguaggio base $\mathcal{AL}$

Un sistema di rappresentazione della conoscenza basato sulle **DL** permette di creare, manipolarle e ragionare su diverse *Knowledge Base*.
Ma cos'è formalmente una **KB**? La base di conoscenza è composta principalmente da una coppia *(T,A)* TBox e ABox. La prima introduce i concetti, insiemi di individui, e ruoli, che denotano relazioni binarie tra i concetti.
Invece per quanto concerne la seconda, essa contiene asserzioni su singoli individui, riguardanti concetti della TBox. Definiamo come descrizioni elementari i concetti e ruoli atomici. Invece trattiamo come complesse le descrizioni che possono essere costruite attraverso induzione. In notazione useremo le lettere A e B per indicare concetti atomici, la lettera R per rappresentare i ruoli atomici e le lettere C e D per le descrizioni dei concetti. Qui sotto daremo una definizione di $\mathcal{AL}$ [1], linguaggio minimo di interesse pratico, su cui si basano tutti gli altri linguaggi di questa famiglia.

Le descrizioni di concetti sono definite secondo le seguenti regole sintattiche:

$C, D \rightarrow$

$\top$	(concetto atomico)
$\bot$	(top concept - generico)
$\neg A$	(negazione atomica)
$C \sqcap D$	(intersezione)
$\forall R.C$	(restrizione di valore)
$\exists R.\bot$	(quantificazione esistenziale limitata)

Per definire una semantica formale per i concetti di $\mathcal{AL}$ poniamo alla base l'idea di modello e l'idea di funzione d'interpretazione:

$$\mathcal{M} = \langle \Delta; fx \rangle \tag{2.1}$$

$$fx : A \rightarrow \Delta \tag{2.2}$$

Definito che le interpretazioni $\mathcal{I}$ sono sottoinsiemi non vuoti di Δ (dominio dell'interpretazione), lo scopo di 2.1 è quello di fornire un significato alle formule, mentre, per

quanto riguarda 2.2, quello di permettere di assegnare a ogni concetto atomico A un insieme $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ e per ogni ruolo atomico R una relazione binaria $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. La funzione di interpretazione è estesa alla descrizione di concetti attraverso le seguenti definizioni induttive :

$$\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
(\neg A)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \backslash A^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \sqcap D^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} &= \{x \in \Delta | \forall y.(x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\} \\
(\exists R.\top)^{\mathcal{I}} &= \{x \in \Delta | \exists y.(x,y) \in R^{\mathcal{I}}\}
\end{aligned}$$

Diciamo che due concetti C, D sono equivalenti, e scriviamo $C = D$, se $C^{\mathcal{I}} = D^{\mathcal{I}}$ per ogni interpretazione $\mathcal{I}$.

Aggiunta di $\mathcal{C}$: l'operatore di negazione
A questo linguaggio, possiamo aggiungere l'estensione di negazione, un concetto arbitrario:

$$\neg C \qquad \text{(concept negation)}$$

La cui semantica risulta essere:

$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$$

Grazie a questa aggiunta il linguaggio $\mathcal{AL}$ si arricchisce, passando ad essere la logica descrittiva $\mathcal{ALC}$.

Esempio

Ci starebbe bene un esempio, come quelli che ha fatto all'inizio del capitolo, però con la sintassi di ALC, magari con un quantificatore, la negazione e una congiunzione, oltre a qualche fatto nella ABox

2.4 Operatore T

2.4.1 Premessa

L'obbiettivo della *TBox* è costruire una tassonomia di concetti (*ossia* un albero di classificazione). Come rappresentare le proprietà dei prototipi e ragionare sulla loro perdita a livelli inferiori?
Data una tassonomia T, con A e B tali che A è un concetto "padre" di B, non sempre tutte le proprietà di A possono essere ereditate da B.
L'approccio tradizionale a questo problema è di gestire le perdite di proprietà integrando alcuni meccanismi di ragionamento non monotono, portando allo studio di estensioni delle logiche descrittive in questo ambito e cercando di superare il limite.

2.4.2 Definizione di T

Idealmente un'estensione possibile dovrebbe possedere almeno le seguenti caratteristiche:

1. una chiara semantica, basata sulla stessa della logica sottostante;
2. la possibilità di specificare proprietà dei prototipi in maniera naturale e diretta;
3. mantenere la decidibilità (già ereditata) e dimostrabile attraverso il metodo convenzionale.

Vediamo dunque come nel lavori [2] e [7] si faccia uso dell'operatore **T** di tipicalità per l'inferenze. La nostra **KB** ha, in aggiunta, un insieme di asserzioni della forma $\mathbf{T}(C) \sqsubseteq D$ o $\mathbf{T}(C)(m)$ dove D è un concetto che non menziona C.

Supponiamo che la *Tbox* contenga:

$T(Student) \sqsubseteq \neg IncomeTaxPayer$

$T(Student \sqcap Worker) \sqsubseteq IncomeTaxPayer$

$T(Student \sqcap Worker \sqcap Erasmus) \sqsubseteq \neg IncomeTaxPayer$

Interpretando **T** come "tipico/i", questa corrisponde alle asserzioni:

(I T studenti non sono tassati)

(I T studenti-lavoratori sono tassati)

(I T studenti-lavoratori in erasmus non sono tassati)

Ipotizziamo che la *ABox* contenga, in alternativa, uno dei seguenti fatti:

1. $Student(andrea)$
2. $Student(agnese), Worker(agnese)$
3. $Student(damiano), Worker(damiano), Erasmus(damiano)$

Dunque possiamo inferire le aspettate conclusioni:

1. $\neg IncomeTaxPayer(andrea)$
2. $IncomeTaxPayer(agnese)$
3. $\neg IncomeTaxPayer(damiano)$

Se *ABox* contenesse l'espressione $\exists HasBrother.Student(davide)$ è possibile dedurre proprietà di individui implicitamente introdotte dalla restrizione esistenziale, come:

$$\exists HasBrother.\neg IncomeTaxPayer(davide)$$

Infine, aggiungere informazioni irrilevanti non dovrebbe modificare le conclusioni. Ammettendo, infatti, che la *TBox* precedente abbia la seguente forma:

$T(Student \sqcap Short) \sqsubseteq \neg IncomeTaxPayer$

$T(Student \sqcap Worker \sqcap Short) \sqsubseteq IncomeTaxPayer$

$T(Student \sqcap Worker \sqcap Erasmus \sqcap Short) \sqsubseteq \neg IncomeTaxPayer$

possiamo concludere che *Short* è un'informazione irrilevante rispetto all'essere tassati. Per la stessa ragione, la conclusione che Andrea sia un'istanza di $\neg IncomeTaxPayer(andrea)$ o meno, non è influenzata qualora si aggiunga l'espressione $Tall(andrea)$ alla $ABox$.

2.5 La logica $\mathcal{ALC} + \mathbf{T}$

Dato un alfabeto contenti nomi di Concetti $\mathcal{C}$, di Ruoli $\mathcal{R}$ e costanti di individui $\mathcal{O}$ il linguaggio $\mathcal{L}$ appartenente alla logica $\mathcal{ALC} + \mathbf{T}$ viene definito a partire dalla distinzione chiave tra *Concetti* e *Concetti Estesi* [2]

Concetti	*ConcettiEstesi*
$A \in \mathcal{C}, \top$ e $\bot$ sono *concetti* di $\mathcal{L}$	se C è un *concetto* di $\mathcal{L}$, allora C e $\mathbf{T}(C)$ sono *concetti estesi* di $\mathcal{L}$
se $C, D \in \mathcal{L}$ e $R \in \mathcal{R}$ allora $C \sqcap D, C \sqcup D \neg C, \forall R.C, \exists R.C$ sono concetti di $\mathcal{L}$	combinazioni booleane di concetti estesi sono a loro volta *concetti estesi* di $\mathcal{L}$.

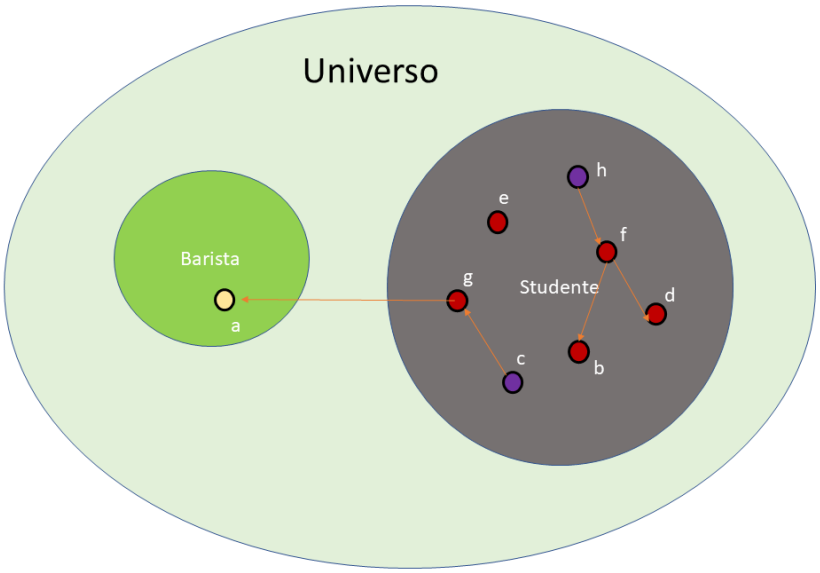
La base di conoscenza **KB** conserva la coppia *(Tbox, Abox)* dove:

- *Tbox*: contiene asserzioni $C \sqsubseteq D$, dove $C \in \mathcal{L}$ è un *concetto esteso* della forma C' oppure $\mathbf{T}(C')$, mentre $D \in \mathcal{L}$ è un *concetto*
- *Abox*: composta da espressioni $C(a)$ e aRb dove $C \in \mathcal{L}$ è un *concetto esteso*, $R \in \mathcal{L}$ e $\mathcal{O}$

Estendiamo la definizione di Modello utilizzato nella terminologia logica sottostante, al fine di fornire una semantica all'operatore **T**

$$\mathcal{M} = \langle \Delta; I; < \rangle$$

Δ è il domino. I è la funzione di estensione. Ecco il nuovo elemento, una relazione d'ordine, non riflessiva, transitiva e modulare, che ha la funzione di determinare, all'interno della *Knowledge Base*, quali individui siano tipici e quali no.



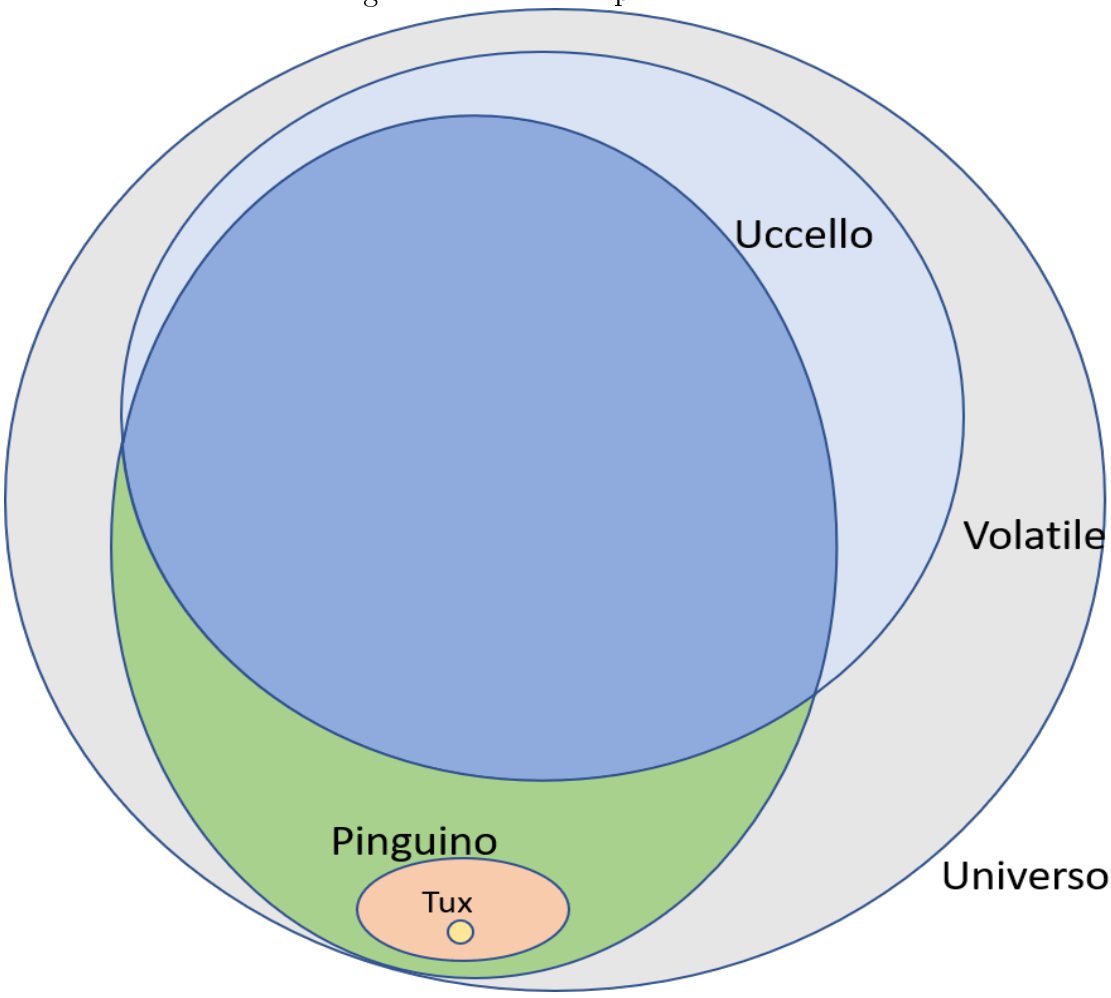
Intuitivamente un individuo m è **tipico** se non esiste alcun elemento più *normale*, viceversa, un n è **atipico** quando esiste almeno un individuo più *normale*. Indichiamo formalmente che l'espressione:

$$x < y \qquad \text{(l'individuo } x \text{ è più } \textit{normale} \text{ di } y\text{)}$$

Consideriamo dunque l'esempio 2.5 della pagina precedente:
gli studenti tipici sono b, c, d, e, f, g, h, in particolare, e è tipico perché non è in alcuna relazione del tipo $x < y$, per quanto riguarda la catenah, f, d, b si evince che d e b siano più *normali* di f che a sua volta è più *normale* di h;
come conseguenza d e b risultano essere tipici poiché questa catena non prosegue bensì termina con loro due. Invece non risulta corretto concludere che g sia uno studente tipico, infatti, nonostante sia più normale di c, è in relazione con a, tuttavia a appartiene ad un insieme diverso (Barista) e questo non influisce sulla sua "tipicità". La relazione di preferenza ha limite di essere parziale, infatti non è sempre possibile stabilire quale elemento sia più tipico degli altri.

In realtà l'estensione di questa sezione viene chiamata $\mathcal{ALC} + \mathbf{T_R}$ dove il pedice R indica il concetto di logica razionale sulle cui proprietà si basa la semantica di T. Tali caratteristiche, come la *specificità*, costituiscono le fondamenta del ragionamento non monotono, permettendo modellare la situazione in questo modo:

Figura 2.2: Un esempio di modello



Nella Figura 2.2 è possibile rappresentare Tux (un *Pinguino*) come è un uccello che non vola; graficamente quest'informazione è racchiusa nella zona verde dell'insieme Uccello, ovvero, quella sezione popolata da tutti gli uccelli atipici, mentre la porzione in blu rappresenta gli elementi tipici.

2.6 La logica non monotona $\mathcal{ALC} + \mathbf{T}_{\mathbf{R}}^{RaCl}$

Nonostante l'aggiunta dell'operatore $\mathbf{T}$ la logica appena vista rimane monotona, nel senso che se il fatto F segue da una certa base di conoscenza $\mathbf{KB}$, allora lo stesso fatto F segue da una qualsiasi $KB' \subseteq KB$. Di conseguenza, a meno che una base di conoscenza contenga delle assunzioni esplicite circa la tipicalità degli individui, non esiste alcun modo per inferire proprietà rivedibili su di loro.
Un'altra limite riguarda l'intrattabilità della *irrilevanza*.

$$LottatoreDiSumo \sqsubseteq Atleta$$

$$\mathbf{T}(Atleta) \sqsubseteq \neg Grasso$$

$$\mathbf{T}(LottatoreDiSumo) \sqsubseteq Grasso$$

per via della monotonia di $\mathcal{ALC} + \mathbf{T_R}$ non si può derivare che:

$$\mathbf{T}(LottatoreDiSumo \sqcap Orientale) \sqsubseteq Grasso$$

malgrado l'etnia sia totalmente irrilevante rispetto all'essere grassi o magri.

Con l'obbiettivo di creare inferenze non monotone utili gli autori in [3] hanno rinforzato la semantica precedente restringendo le assegnazioni ad una classe minimale di modelli. L'idea è quella di restringere l'assegnazione a modelli minimi che *minimizzino l'atipicalità dei concetti* e dove le inclusioni implicate sono quelle che appartengono alla chiusura razionale della base di conoscenza, estensione naturale di [6].

Considerare solo i modelli che massimizzano le istanze tipiche di un concetto quando sono consistenti con la base di conoscenza. Senza entrare troppo nei dettagli la semantica $\mathcal{ALC} + \mathbf{T}_{\mathbf{R}}^{RaCl}$ non monotona si basa su modelli razionali minimi che riducono al minimo il *rank* degli elementi del dominio.

Intuitivamente, dati due modelli $\mathcal{M}_\infty, \mathcal{M}_\in$ di una $\mathbf{KB}$ se è noto che in M_1 un elemento x ha rank 2 (a causa di istanze $z < y < x$) ed in M_2 x ha rank 1 (a causa di $y < x$), noi preferiamo il secondo, perché l'elemento x risulta più tipico che in M_1. I modelli vengono quindi selezionati per il ragionamento scartando quelli grado più elevato poiché in essi gli elementi sono meno "tipici" e quindi verrebbero dedotte meno informazioni (vedi anche Figura a pagina 16).

2.6.1 Traduzione dell'operatore T

In alcuni contesti non è sempre possibile modificare l'intera struttura basata su logiche consolidate. È sensato chiedersi quale sia il significato di questo operatore e se esistano formulazioni equivalenti. Consideriamo la $TBox$:

$$\mathbf{T}(Pesce) \sqsubseteq Oviparo$$

Come sappiamo esprime il fatto che, tipicamente, gli uccelli sono ovipari (depositano le uova).
Ecco la traduzione equipollente, vista in [11] e [7], senza far uso dell'operatore **T** è la seguente:

1. $Pesce \sqcap Pesce1 \sqsubseteq Oviparo$
2. $Pesce1 \sqsubseteq \forall R(\neg Pesce \sqcap Pesce1)$
3. $\neg Pesce1 \sqsubseteq \exists R(Pesce \sqcap Pesce1)$

Questa traduzione implementa la relazione d'ordine $<$ precedentemente introdotta. L'insieme $Pesce1$ rappresenta i pesci tipici mentre $Pesce$ contiene tutti i possibili pesci, compresi quelli atipici, e costituisce infatti un soprainsieme di Pesce1.
La suddivisione in questi due insiemi a 2.3 è fondamentale per poter tenere traccia delle eccezioni, ottenendo così la possibilità di ragionare sia sull'individuo generico sia sul tipico individuo.

Pertanto per inferire che *nemo* sia un tipico $Pesce$ controllerà che la seguente

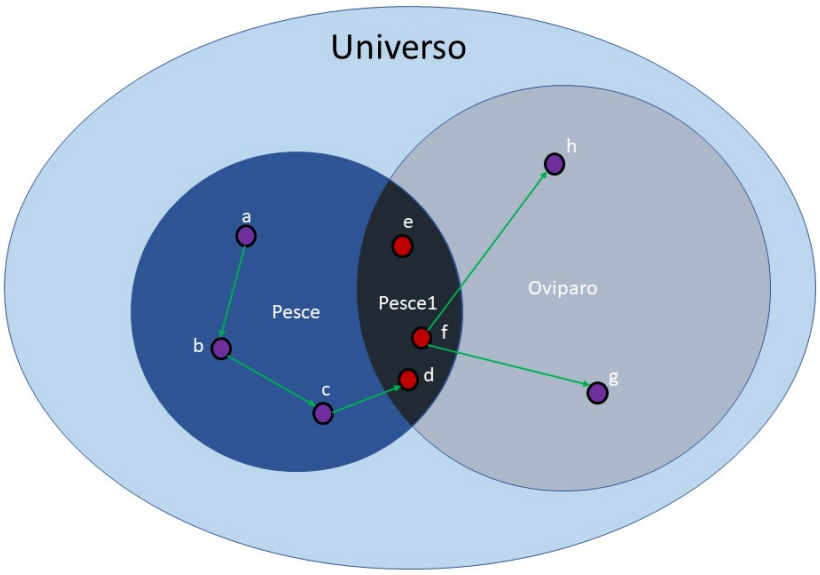


Figura 2.3: Esempio traduzione $\mathbf{T}(Pesce) \sqsubseteq Oviparo$

espressione sia vera:

$$\mathbf{T}(Pesce)(nemo) \iff Pesce(nemo) \wedge Pesce1(nemo)$$

In caso di risposta affermativa sarà verificato che:

$$Pesce(nemo) \sqcap Pesce1(nemo) \sqsubseteq Oviparo(nemo)$$

Ecco quindi verificata la condizione 1 di 2.6.1 Viceversa la definizione 2 e 3 esprimono formalmente cosa significhi essere un individuo a/tipico.

$Pesce1 \sqsubseteq \forall R.(\neg Pesce \sqcap Pesce1)$ $\neg Pesce1 \sqsubseteq \exists R.(Pesce \sqcap Pesce1)$

Nell'esempio 2.6.1, i membri d, e ed f sono pesci tipici. È evidente che l'intersezione $\neg Pesce \sqcap Pesce1$ sia vuota. Dunque deduciamo che, dato l'insieme A, se un elemento $m \in A$ è **tipico** o non è in alcuna relazione del tipo $x < y$ ()e quindi non esiste un individuo più normale di lui) o è più ordinario di un generico x $(m < x)$ con $x \in A$ oppure si trovi nella relazione $x < m$ dove però $x \notin A$.

Per quanto concerne i pesci **atipici**, come le istanze a, b e c, per verificarne la tipicità viene svolto un procedimento molto simile. Prendiamo in esempio b: viene verificato inizialmente lo stato dell'elemento (se si trovi o meno in una o più relazioni) e analizzato. In questo caso scopriamo che è più caratteristico di a ma meno di c che, a sua volta, è più generico di d, che scopriamo essere tipico. Per transitività troviamo la relazione $d < b$ che conferma a b la sua atipicità. In conclusione, un membro è atipico quando esiste **almeno un** individuo più normale di lui

2.7 La logica $\mathcal{ALC} + \mathbf{T}^{\mathsf{P}}_{\mathbf{R}}$

Dopo aver illustrato la logica di base $\mathcal{ALC}$ ed introdotto l'estensione non monotona $\mathcal{ALC} + \mathbf{T}$ con i relativi concetti di **chiusura e logica razionale**, in questa sezione verrà descritta l'estensione "accennata" nel capitolo introduttivo 1.1 che permette di tenere in considerazione la probabilità di individui particolari [8].

2.7.1 Modifiche alla semantica

L'inclusione di tipicalità si evolve e va ad assumere la forma:

$$\mathbf{T}(C) \sqsubseteq_p D$$

con il significato intuitivo aggiuntivo: la probabilità di avere un C eccezionale (cioè **atipico**) che non sia anche un D vale $1 - p$.

La base di conoscenza (**KB**) $(TBox, ABox)$ ha la seguente struttura

$TBox$ in cui $p \in ep \in (0, 1)$

- $C \sqsubseteq_p P$
- $\mathbf{T}(C) \sqsubseteq_p P$

$ABox$ in cui $a, b \in$ ı

- $C(a)$
- $R(a, b)$

È facilmente intuibile che più la probabilità p è alta più l'inclusione è "libera da eccezioni" o, equivalentemente, è meno probabile avere un C speciale che non è anche un D.
A tal proposito è importante sottolineare che la probabilità p con $p = 1$ non è consentita in quanto l'inclusione $\mathbf{T}(C) \sqsubseteq_1 D$ corrisponde all'inclusione stretta $\mathbf{T}(C) \sqsubseteq D$

che esprime invece il fatto che l'elemento C è sicuramente anche un D.
Data una seconda inclusione $\mathbf{T}(C') \sqsubseteq_{p'} D'$, con $p' < p$, si assume che questa inclusione sia meno "restrittiva" rispetto alla prima in quanto la possibilità di avere un eccezionale C' è più alta rispetto alla probabilità di avere un eccezionale C, tenendo rispettivamente conto delle proprietà D' e D.

Considerando, per esempio, la seguente *TBox*

- $\mathbf{T}(Liceale) \sqsubseteq_{0.80} Studioso$
- $\mathbf{T}(Liceale) \sqsubseteq_{0.60} PraticaDelloSport$

si evince che il tipico scolare è studioso e che, normalmente, pratica dello sport; entrambe sono proprietà del prototipo dello studente, tuttavia ci sono più eccezioni di studenti che non fanno sport rispetto a quelli che non studiano.

Una cosa importante da tenere in considerazione è la possibilità di avere *Knowledge Base* contenenti inclusioni con $p \leq 0.5$, che se erroneamente interpretate, potrebbero venir considerate contro-intuitive.

Ad esempio,l'inclusione $\mathbf{T}(Liceale) \sqsubseteq_{0.22} Fumatore$ potrebbe venir erroneamente interpretata come "normalmente, gli studenti non sono fumatori"; anche se la corrispondente probabilità è bassa, la spiegazione corretta è che fare uso di sigarette è in ogni caso una proprietà del prototipo dello studente liceale.
A differenza dell'espressione $\mathbf{T}(Liceale) \sqsubseteq_{0.80} Studioso$, si ha che la probabilità di trovare studenti eccezionali non fumatori è più alta rispetto a quella di trovare studenti eccezionali che siano studiosi.
Ponendo il caso in cui si volesse formalizzare che il tipico studente non è una persona giovane, bisogna semplicemente formulare l'inclusione $\mathbf{T}(Liceale) \sqsubseteq_{0.78} \neg Fumatore$ nella base di conoscenza.

2.7.2 Estensione dell'*Abox*

Data una base di conoscenza **KB** , viene definito l'insieme finito $\mathfrak{Tip}$ dei concetti che occorrono all'interno dell'operatore di tipicalità, formalmente

$$\mathfrak{Tip} = \{C | \mathbf{T}(C) \sqsubseteq_p D \in KB\}.$$

Dato un individuo a esplicitamente dichiarato nell'$ABox$, si definisce l'insieme delle assunzioni di tipicalità $\mathbf{T}(C)(a)$ che possono essere dedotte in maniera minimale dalla **KB** nella logica non monotona $\mathcal{ALC} + \mathbf{T}_{\mathbf{R}}^{RaCl}$, con $C \in \mathfrak{Tip}$.

Quindi si considera un insieme ordinato $\mathfrak{Tip}_{\mathcal{A}}$ (dove $\mathcal{A}$ sta per $ABox$) di coppie (a, C) di tutte le possibili assunzioni $\mathbf{T}(C)(a)$, per tutti i concetti $C \in \mathfrak{Tip}$ e per tutti gli individui a nell'$ABox$.

In aggiunta si definisce il multi-insieme ordinato $\mathcal{P}_{\mathcal{A}}$ tupla della forma $[p_1, p_2, ..., p_n]$ dove p_i è la probabilità dell'assunzione $\mathbf{T}(C)(a)$ tale che $(a, C) \in \mathfrak{Tip}_{\mathcal{A}}$ alla posizione i; inoltre rappresenta il prodotto di tutte le probabilità $p_i j$ delle inclusioni $\mathbf{T}(C) \sqsubseteq_{p_i j} D$ nella $TBox$.

Seguendo le idee di [10], si considerano diverse estensioni $\widetilde{\mathcal{A}}_i$ dell'$ABox$ che vengono equipaggiate con una probabilità p_i. Partendo dagli insiemi $\mathcal{P}_{\mathcal{A}} = [p_1, p_2, ..., p_n]$ e

$\mathfrak{Tip}_{\mathcal{A}}$, il primo passo è quello di definire l'insieme $\mathbb{S}$ di tutte le stringhe di possibili assunzioni, utilizzando lo 0 come p_i per rappresentare che la corrispondente asserzione di tipicalità non viene più assunta.

Successivamente, si definisce l'estensione $\widetilde{\mathcal{A}}_i$ di $\mathcal{A}$ corrispondente ad una stringa $[s_1, s_2, ..., s_n] \in \mathbb{S}$ cosi ottenuta. In questo modo, la probabilità più alta viene assegnata all'estensione dell'$ABox$ corrispondente a $\mathcal{P}_{\mathcal{A}}$, dove tutte le assunzioni di tipicalità vengono considerate. mentre diminuisce nelle altre estensioni, alcune assunzioni di tipicalità vengono scartate, così 0 viene usato al posto della corrispondente p_i.

La probabilità di una estensione quindi $\widetilde{\mathcal{A}}_i$ corrispondente a $\mathcal{P}_{\mathcal{A}i} = [p_{i1}, p_{i2}, ..., p_{in}]$ è definita come il prodotto delle probabilità p_{ij} quando $p_{ij} \neq 0$ (cioè la possibilità della corrispondente assunzione di tipicalità nel momento in cui questa viene selezionata per l'estensione) e $1 - p_j$ quando $p_{ij} = 0$ (cioè la corrispondente assunzione di tipicalità viene scartata, per segnalare che l'estensione contiene un'eccezione all'inclusione).

Si può osservare che, in $\mathcal{ALC} + \mathbf{T}_{\mathbf{R}}^{RaCl}$, l'insieme delle assunzioni di tipicalità che possono essere inferite da una $\mathbf{KB}$ corrispondono all'estensione $\mathcal{A}$ equivalenti alla stringa $\mathcal{P}_{\mathcal{A}}$ (nel caso in nessun elemento sia impostato a 0); vengono considerate tutte le assunzioni di tipicalità, degli individui presenti nell'$ABox$, consistenti con la base di conoscenza.
Al contrario, in $\mathcal{ALC} + \mathbf{T}_{\mathbf{R}}$, nessuna assunzione di tipicalità può esser dedotta da una $\mathbf{KB}$, e questo equivale ad estendere $\mathcal{A}$ con delle asserzioni corrispondenti alla stringa $[0, 0, ..., 0]$, ovvero l'insieme vuoto.

Otteniamo dunque una distribuzione di probabilità sulle estensioni di $\mathcal{A}$) (se presenti). Prendiamo come esempio una $\mathbf{KB}(\mathcal{T}, \mathcal{A})$ in cui le uniche inclusioni di tipicalità in $\mathcal{T}$ siano le seguenti:

1. $\mathbf{T}(C) \sqsubseteq_{0.60} D$
2. $\mathbf{T}(E) \sqsubseteq_{0.85} F$

e a e b siano gli unici individui presenti in $\mathcal{A}$; supponiamo inoltre che $\mathbf{T}(C)(a)$, $\mathbf{T}(C)(b)$ e $\mathbf{T}(E)(b)$ siano dedotte dalla $\mathbf{KB}$ con la logica $\mathcal{ALC} + \mathbf{T}_{\mathbf{R}}^{\mathsf{P}}$.

Il risultato è quindi:

$$\mathfrak{Tip}_{\mathcal{A}} = \{(a, C), (b, C), (b, E)\} \qquad \mathcal{P}_{\mathcal{A}} = [0.6, 0.6, 0, 85]$$

Tutte le possibili stringhe, tutte le corrispondenti estensioni di $\mathcal{A}$ e tutte le probabilità sono illustrate nella seguente tabella 2.4

Stringa	Estensione	Probabilità
[0.6, 0.6, 0.85]	$\widetilde{\mathcal{A}_1} = \{\mathbf{T}(C)(a), \mathbf{T}(C)(b), \mathbf{T}(E)(b)\}$	$\mathbb{P}_{\widetilde{\mathcal{A}_1}} = 0.6 \times 0.6 \times 0.85 = 0.306$
[0, 0, 0.85]	$\widetilde{\mathcal{A}_2} = \{\mathbf{T}(E)(b)\}$	$\mathbb{P}_{\widetilde{\mathcal{A}_2}} = (1 - 0.6) \times (1 - 0.6) \times 0.85 = 0.136$
[0, 0.6, 0]	$\widetilde{\mathcal{A}_3} = \{\mathbf{T}(C)(b)\}$	$\mathbb{P}_{\widetilde{\mathcal{A}_3}} = (1 - 0.6) \times 0.6 \times (1 - 0.85) = 0.036$
[0.6, 0, 0]	$\widetilde{\mathcal{A}_4} = \{\mathbf{T}(C)(a)\}$	$\mathbb{P}_{\widetilde{\mathcal{A}_4}} = 0.6 \times (1 - 0.6) \times (1 - 0.85) = 0.036$
[0, 0.6, 0.85]	$\widetilde{\mathcal{A}_5} = \{\mathbf{T}(C)(b), \mathbf{T}(E)(b)\}$	$\mathbb{P}_{\widetilde{\mathcal{A}_5}} = (1 - 0.6) \times 0.6 \times 0.85 = 0.204$
[0.6, 0, 0.85]	$\widetilde{\mathcal{A}_6} = \{\mathbf{T}(C)(a), \mathbf{T}(E)(b)\}$	$\mathbb{P}_{\widetilde{\mathcal{A}_6}} = 0.6 \times (1 - 0.6) \times 0.85 = 0.204$
[0.6, 0.6, 0]	$\widetilde{\mathcal{A}_7} = \{\mathbf{T}(C)(a), \mathbf{T}(C)(b)\}$	$\mathbb{P}_{\widetilde{\mathcal{A}_7}} = 0.6 \times 0.6 \times (1 - 0.85) = 0.054$
[0, 0, 0]	$\widetilde{\mathcal{A}_8} = \emptyset$	$\mathbb{P}_{\widetilde{\mathcal{A}_8}} = (1-0.6) \times (1-0.6) \times (1-0.85) = 0.024$
	$\mathbb{P}\widetilde{\mathcal{A}_1} + \mathbb{P}_{\widetilde{\mathcal{A}_2}} + \ldots + \mathbb{P}_{\widetilde{\mathcal{A}_8}} =$	1

Figura 2.4: Estensioni plausibili

2.8 Aggiunte

Qui deve assolutamente aggiungere, prima di considerare conclusa la descrizione della logica, la parte relativa alla diagnosi. Allo stato attuale, ciò è limitata alla descrizione dell'implementazione, ma dovrebbe descriverla prima formalmente a partire dalla logica che ha correttamente richiamato. Inoltre, si faccia accompagnare da un esempio, magari il solito di Greg (gli cambi i nomi). Il lettore, prima di leggere di owlready, deve avere idea di dove lo stiamo portando. Se non le piace mescolare questa parte nel capitolo della logica, va benissimo: faccia un capitolo a parte, anche breve, ma sulla DbN

E con questo concludiamo la trattazione dei fondamenti di logica.
Questa capitolo altro non è che un sunto, a tratti informale, della storia di questa branca matematica e non sostituisce certamente gli articoli e studi fatti nel corso degli anni, da figure molto più autorevoli, della mia ma costituisce il corpo fondante di tutta la tesi.

Capitolo 3

Strumenti utilizzati

In questo capitolo descriveremo le componenti che utilizzeremo nella nostra architettura e, per ognuna di queste, introdurremo brevemente le caratteristiche chiave.

3.1 OWL2

L'OWL 2 Web Ontology Language, informalmente OWL 2, è un linguaggio ontologico, basato sulle logiche descrittive o **DL**, costruito per il Semantic Web con un significato formale e definito in [9]. Le ontologie in OWL 2 vengono definite tramite la specifica di classi, proprietà, individui e valori dei dati e sono memorizzate come documenti del Semantic Web. Queste possono essere pubblicate sul web e riferite da altre ontologie, per costruire basi di conoscenza più complesse. Inoltre sono spesso utilizzate assieme insieme a documenti scritti nel formato RDF, ed stesse vengono scambiate principalmente come documenti RDF.

3.1.1 Semantica

La semantica è definita *formalmente*, cioè permette di scrivere *Knowledge Base* sulle quali è possibile applicare inferenze in modo automatico. OWl2 possiede due semantiche:

Semantica diretta, basata sulle Logiche Descrittive (vedi pag.9)

- è applicabile a un frammento del linguaggio detto OWL2 DL;
- è sempre decidibile;
- ha una sintassi più ristretta.

Semantica Indiretta, basata sui grafi RDF

- estende la semantica formale di RDF
- ha la massima espressività
- la decibilità non garantita

la sēmantikós è basata su **iff** (se e solo se) quindi:

$$C \text{ è sottoclasse di D} \iff \text{istanze di } C \subseteq \text{istanze di } D$$

3.1.2 Caratteristiche

Il linguaggio prevede tre componenti principali:

1. **Entità**: elementi atomici usati per riferirsi ad oggetti, categorie e relazioni del mondo reale; costituiscono gli assiomi
   $Lara, donna, Pietro, Sofia, essere\_fidanzati$
2. **Assiomi**: affermazioni (*statement*) di base espressi da un'ontologia OWL
   Lara è una *donna* | *Pietro* e *Sofia* sono *fidanzati*
3. **Espressioni**: combinazioni di entità che costituiscono descrizioni complesse, definite tramite l'utilizzo di costruttori.
   Medico e *Donna* combinate definiscono *DonnaMedico*

Ogni file *.owl* inizia con un preambolo:

```
Prefix(:=<http://example.com/owl/families/>)
Prefix(otherOnt:=<http://example.org/otherOntologies/families/>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
Ontology(<http://example.com/owl/families>
Import( <http://example.org/otherOntologies/families.owl> )
....
```

`Prefix(..)`

permette di fare, sinteticamente, riferimento a elementi definiti in altre ontologie o in altri file; il prefisso più l'etichetta sono composti nell'identificatore dell'elemento di interesse, ad esempio `owl:Thing` diventa `http://www.w3.org/2002/07/owl#Thing`.

`Ontology(..)`

Specifica l'**URI** (Uniform Resource Identifier) del file contenente l'ontologia definita.

3.1.3 Sintassi e Modellazione di Base

È possibile scrivere ontologie OWL utilizzando sintassi differenti:

- Functional-Style `ClassAssertion(:Persona:Damiano)`
- Manchester `Individual: Damiano`
- Turtle `:Damiano rdf:type :Persona`
- RDF/XML `<Persona rdf:about="Damiano">`
- OWL

```
<ClassAssertion>
<Class IRI="Person"/>
<NamedIndividual IRI="Damiano"/>
</ClassAssertion>
```

Ed esiste una chiara equivalenza tra le varie terminologie utilizzate, vediamola:

Ing. della conoscenza	*Description Logic*	*Termini OWL*
• Oggetti	• Costanti	• Individui
• Categorie	• Predicati Unari	• Classi
• Relazioni	• Predicati Binari	• Proprietà

Dopo queste considerazioni generali, entriamo ora nei dettagli della modellazione con OWL2. Nei paragrafi successivi introdurremo le funzionalità essenziali per produrre una base di conoscenza. Queste saranno condite con esempi, semplici dimostrazioni delle varie funzionalità di OWL. Per semplicità useremo il *Functional-Style*.

Ecco come si esprimono gli Assiomi:

- dichiarazioni di individuo: `Declaration(Name Individual(:Dario))`
- dichiarazioni di classe: `Declaration(Class(:Persona))`
- dichiarazioni di proprietà: `Declaration(ObjectProperty(:Uomo))`

E alcune delle relazioni chiave:

`ClassAssertion(:Persona :Dario))`	lega un'istanza ad una classe;
`SubClassOf(:Persona :Uomo))`	relazione di sottoclasse ($\sqsubseteq$);
`EquivalentClasses(:Persona :Umano)`	equivalenza di due classi;
`DisjointClasses(:Donna :Uomo)`	classi disgiunte.

Permette di legare due individui tramite una proprietà.

`ObjectPropertyAssertion(:haMoglie :Donna :Uomo)`

3.1.4 Classi complesse e implementazione di $\sqcap, \sqcup, \exists$ e $\forall$

Tramite opportuni costrutti è possibile specificare classi complicate e relazionarle, anche grazie all'operatore di intersezione $\cap$ e disgiunzione $\cup$.

```
EquivalentClasses(:Padre
        ObjectIntersectionOf(:Uomo :Genitore))
EquivalentClasses(:Genitore
        ObjectIntersectionOf(:Madre :Padre))
```

Nonna è sottoclasse dell'intersezione fra *Donna e Genitore*.

```
SubClassOf(:Nonna
        ObjectIntersectionOf(:Donna :Genitore))
```

L'individuo *Marco* è una *Persona* (e) non *Genitore*.

```
ClassAssertion(
        ObjectIntersectionOf(:Persona
                ObjectComplementOf(:Genitore))
:Marco)
```

Vediamo, infine, come si possa utilizzare i quantificatori $\exists$ e $\forall$ con qualche esempio. La classe *Genitore* è l'insieme di quegli individui che possiedono almeno un'istanza di *Persona* che è loro Figlio; una persona è *Felice* quanto tutti i suoi figli sono felici. Le persone che non hanno figli, vengono inclusi come felici.

Quantificatore esistenziale $\exists$

```
EquivalentClasses(
  :Genitore
  ObjectSomeValuesFrom
  (:haFiglio :Persona))
```

Quantificatore universale $\forall$

```
EquivalentClasses(
  :PersonaFelice
   ObjectAllValuesFrom
    (:haFiglio :PersonaFelice))
```

Questo conclude la nostra breve visione sintetica del linguaggio.

3.1.5 OWl2 *Versus* DB e considerazioni finali

I file *.owl* conservano informazioni ma nonostante ciò OWL2 **non** è un framework per basi di dati; Nonostante parte della terminologia evochi assonanze con i **D**ata**B**ase, la semantica di base ha delle differenze sostanziali.

In primis l'assunzione di mondo: un fatto non contenuto in un **DB** è considerato falso (mondo chiuso) mentre nel mondo logico viene considerato mancante (mondo aperto). In secondo luogo, classi e proprietà possono avere definizioni multiple e OWL non richiede che le uniche proprietà di un individuo siano quelle della classe a cui appartiene. Infine ricordiamo ciò che abbiamo visto a 3.1.2 : le informazioni riguardanti un singolo individuo possono essere distribuite su più documenti diversi, contrariamente ad un classico **DB**.

In conclusione vogliamo ricordare che OWL **non** è un linguaggio di programmazione, bensì un linguaggio *dichiarativo*, in grado di rappresentare della conoscenza. Diversi sono gli strumenti a disposizione per trattare le ontologie: ragionatori automatici, API, validatori, editor e ambienti di sviluppo. Nella sezione successiva tratteremo alcuni di questi tool che fanno parte di DbN.

3.2 Owlready2

Lavorare direttamente con OWL è impegnativo e tedioso, ma, in nostro soccorso, arrivano le API (Application Programming Interface). Tra le più recenti e interessanti spicca Owlready2 [5], pratica libreria *ontology-oriented*, scritta in Python3. Owlready versione 2 permette un accesso trasparente alle ontologie, contrariamente alle API basate su Java. Può caricare ontologie OWL2 come oggetti Python, modificarli, salvarli e, appoggiandosi ad *HermiT* e *Pellet* (reasoner scritti in Java), eseguire veri e propri ragionamenti. Vediamo alcune caratteristiche chiave

3.2.1 Tabella di conversione

Se dovessimo "convertire" le formule tra Description Logics, Owlready2 e/o Protegè, potrebbe essere di interesse la sottostante tabella.

DLs	Protégé	Owlready
$A \sqsubseteq B$	A subclass of B	class A(B): ... (or) A.is_a.append(B)
$A \sqcap B$	A and B	A & B
$A \sqcup B$	A or B	A | B
$\neg A$	not A	Not(A)
$A \sqcap B = \emptyset$	A disjoint with B	AllDisjoint([A, B])
$A \equiv B$	A equivalent to B	A.equivalent_to.append(B)
{i, j,...}	{i, j,...}	OneOf([i, j,...])
$\exists R.B$	R some B	R.some(B)
$\forall R.B$	R only B	R.only(B)
=2R.B	R exactly 2 B	R.exactly(2, B)
$\exists R.\{i\}$	R value i	R.value(i)
$\exists R.\top \sqsubseteq A$	R domain A	R.domain = [A]
$\top \sqsubseteq \forall R.B$	R range B	R.range = [B]
$S \equiv R^-$	S inverse of R	S.inverse = R
A(i)	i type A	i = A() (or) i.is_instance_of.append(A)
R(i, j)	i object property assertion j	i.R = j (R is functional) (or) i.R.append(j) (otherwise)
R(i, n)	i data property assertion j	i.R = n (R is functional) (or) i.R.append(n) (otherwise)
$A \sqsubseteq \exists R.\{i\} \wedge (\exists R^-.A)(i)$	-	A.R = i (R is functional) (or) A.R.append(i) (otherwise)

Molti di questi simboli ci sono familiari e, di conseguenza, il salto rappresentazionale è molto basso. Utilizzare la semantica di questo pacchetto non risulta troppo impegnativo, poiché coerente con le Logiche Descrittive.

3.2.2 Che cosa posso fare con OWLReady2?

```
from owlready2 import *

# Caricare un ontologia da una repostiory locale o da Internet:
onto = get_ontology("http://www.lesfleursdunormal.fr/.. .../pizza_onto.owl")
onto.load()

# Creare nuove classi nell'ontologia
# mischiando costrutti OWL e metodi Python:
class NonVegeterianPizza(onto.Pizza):
    equivalent_to = [onto.Pizza &
    (onto.has_topping.some(onto.MeatTopping) |
    onto.has_topping.some(onto.FishTopping)
    )]

    def eat(self) : print ("Yuuum! So good!")

with onto:
    class Pizza (Thing):
        def eat(self) : print ("I love pizza !")
    pass

# Accedere le classi dell'ontologia e creare nuovi Individui/instanze:
test_pizza = onto.Pizza("test_pizza_owl_identifier")
test_pizza.has_topping = [onto.CheeseTopping(),onto.TomatoTopping()]
print(test_pizza.has_topping)
''' [pizza_onto.cheesetopping1, pizza_onto.tomatotopping1] '''

# In questo pacchetto quasi ogni lista può essere modificata sul posto,
# per esempio aggiungendo/rimuovendo elementi dalla lista.
# Owlready2 aggiornerà in automatico il quadstore RDF.
test_pizza.has_topping.append(onto.MeatTopping())
print(test_pizza.has_topping)
''' [pizza_onto.cheesetopping1, pizza_onto.tomatotopping1,
        pizza_onto.meattopping1]'''
test_pizza.eat() ''' I love pizza ! '''

# Effeturare "reasoning" e classificare le istanze e le classi
print(test_pizza.__class__) ''' pizza_onto.Pizza '''
sync_reasoner()
print(test_pizza.__class__) ''' pizza_onto.NonVegeterianPizza '''
test_pizza.eat()            ''' Yuuum! So good! '''

# Esportare l'ontologia in un file .owl
onto.save("Demo")
```

3.2.3 Architettura

Owlready2 mantiene un quadstore (**DB** di quadruple) RDF in un database ottimizzato *SQLite3*, sia in memoria che, opzionalmente, su disco.
Fornisce, inoltre, un accesso di alto livello alle classi e agli oggetti presenti nell'ontologia. Classi e Individui vengono caricati dinamicamente dal **DB** secondo necessità, salvati in memoria e poi eliminati quando non più necessari.

3.2.4 Paragone con precedenti approcci

	Goldman 2003 [28]	Kalyanpur 2004 [26]	Koide 2005 [24]	Babik 2006 [30]	Stevenson 2011 [32]	**Owlready**
Type	static	static	dynamic	dynamic	semi-dynamic	dynamic
Language	C#	Java	CommonLisp	Python	Java	Python
Classification of individuals	no	no	**yes**	?	**yes**	**yes**
Classification of classes	no	no	?	?	no	**yes**
Methods in OWL classes	no	no	?	no	no	**yes**
Syntax for OWL class expressions	no	no	**yes**	no	?	**yes**
Syntax for "role-fillers"	no	no	no	no	no	**yes**
Local closed world reasoning	no	no	no	no	no	**yes**

Figura 3.1: Paragone di Owlready con altri approci di programmazione.

La tabella 3.1 confronta diverse tecnologie precedenti con Owlready2. Owlready si distingue come uno degli approcci più avanzati. In particolare, è in grado di classificare automaticamente (non solo gli individui ma anche le classi), compiere "reasoning" sul mondo locale chiuso e non e proporre una sintassi ad alto livello per definire i vincoli "role-fillers". Queste ed altre caratteristiche risultano cruciali quando si lavora con ontologie mediche, ma posso essere utili anche in altri domini.

Questo sono le componenti chiave di un sistema capace di adattarsi a differenti casi d'uso in maniera decisamente flessibile

3.3 Plotly

Plotly.py è una delle migliori librerie *Open Source* di plotting: supporta oltre 40 tipi di grafici unici, interattivi e ricchi di funzionalità, andando a coprire una vasta gamma di casi d'uso: statistico, finanziario, geografico, scientifico e tridimensionale. Non mancano i classici grafici a linee, a barre, a bolle e a punti. [4]

3.3.1 Perché usare Plotly.py?

Costruita sopra la più celebre libreria Javascript (Plotly.js, composta da d3.js e stack.gl), Ploty.py permette di creare bellissime realizzazioni interattive basate sul web, che possono essere salvate come file HTML o utilizzate come parte di web-app scritte in Python. Tutti i grafici di Plotly.py sono completamente costumomizzabili. Tutto, dai colori e dalle etichette alle linee della griglia e alle legende, può essere

personalizzato utilizzando una serie di attributi dedicati. I diagrammi, inoltre, sono dotati di funzionalità interessanti come lo zoom, il panning, il ridimensionamento automatico, ecc.
Grazie all'integrazione profonda con Orca, utility per l'esportazione di immagini, Plotly.py fornisce un notevole supporto anche per i contesti al di fuori del web, inclusi gli IDE (PyCharm, Spyder) e la pubblicazione di documenti cartacei, come questo testo.

L'obiettivo era quello di rendere gradevoli ed esplicativi i risultati "diagnostici" prodotti dal tool DbN.

3.3.2 Che cosa posso fare con Plotly.py?

```
# Esistono principalmente due modi per rappresentare le figure:
# usando i dict di python
import plotly.io as pio

fig_dict = {
    "data": [{"type": "bar",
              "x": [1, 2, 3],
              "y": [1, 3, 2]}],
    "layout": {"title": {"text": "A Bar Chart"}}
}
pio.show(fig)

# usando la gerarichia delle classi chiamata "graph objects".
import plotly.graph_objects as go
fig_graph = go.Figure(
    data=[go.Bar(x=[1, 2, 3], y=[1, 3, 2])],
    layout=go.Layout(
        title=go.layout.Title(text="A Bar Chart")
    )
)
fig.show()

# Si possono salvare i diagrammi in singoli file HTML:
fig.write_html('first_figure.html', auto_open=True)

# Esiste anche la possibilità di creare sottografici
from plotly.subplots import make_subplots

# Creiamo un grafico da una riga e due colonne:
fig = make_subplots(rows=1, cols=2)

# Aggiungiamo un grafico a dispersione e un istogramma:
fig.add_scatter(y=[4, 2, 1], mode="lines", row=1, col=1)
fig.add_bar(y=[2, 1, 3], row=1, col=2)
fig.show()
```

Capitolo 4

Caso d'uso: Strumento per il supporto diagnostico

In questo capitolo verrà descritto il programma oggetto della tesi DbN (Diagnosis by Numbers) che altro non è che un'estensione di **PEAR** (Probability of Exceptions and typicAlity Reasoner) [11]. Inizialmente si fornirà un breve riassunto del lavoro di Soriano; successivamente cosa sia stato preso o meno.
Poi obiettivi posti e aggiunte ed infine il progetto completo con annesso esempio. Come linguaggio per lo sviluppo è stato scelto *Python* (in particolare la sua versione *Python3.7*) per diversi motivi, eccone alcuni:

- continuità con il lavoro precedente;
- la presenza di nuove API (vedi 3.2 pag. 27);
- tipicamente richiede meno codice rispetto ad altri;
- è uno dei linguaggi più popolari per l'elaborazione scientifica;

4.1 PEAR - Sintesi

Struttura Questo tool, basato sulla logica $\mathcal{ALC} + \mathbf{T}^{\mathsf{p}}_{\mathbf{R}}$, è strumento che permette di ragionare e dedurre informazioni, di definire precisamente chi siano gli individui tipici, atipici e quali siano le caratteristiche peculiari di una data categoria. Il funzionamento dell'intero strumento è riassumibile in questi passi:

1. dopo aver letto le informazioni costituenti la KB, presenti in un file a parte, viene creata l'ontologia;
2. vengono combinate le probabilità delle assunzioni di tipicalità e generati tutti gli scenari (possibili realtà dei fatti);
3. di ogni scenario viene calcolata la probabilità complessiva;
4. viene verificata o meno la consequenzialità logica del fatto F dato in input per ogni scenario;
5. infine vengono mostrati i risultati dell'interrogazione.

Questo è possibile anche grazie alla seguente architettura: dove i singoli file richia-

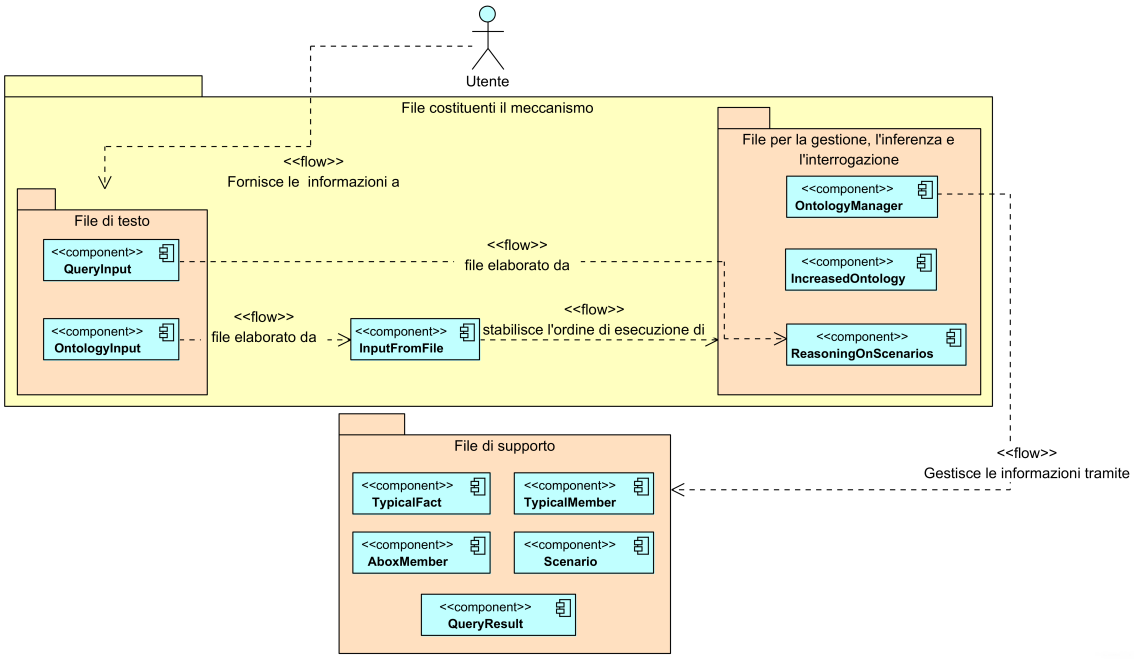


Figura 4.1: Architettura di Pear

mano, spesso, i concetti teorici associati, precedentemente illustrati.

Limiti Le conclusioni così prodotte sono certamente interessanti, ma la procedura di ragionamento in se è vincolata, di volta in volta, dalla query iniziale.
Intuitivamente, ci si chiede se un determinato evento F sia conseguenza logica nella possibili diramazioni (scenari) $B_1, B_2, ..., B_n$ della base di conoscenza B. Questo procedimento non è applicabile in un contesto realistico, poiché chiederebbe all'utente di comprendere la logica anche solo per capire come effettuare un'interrogazione significativa. Inoltre sono presenti dei difetti tecnici, come, ad esempio, la necessità, durante la verifica di conseguenza logica, di effettuare una copia dell'istanza del manager dell'ontologia per ogni scenario.

Alla luce di quanto detto in precedenza ecco la necessità di ottimizzazione ed espansione di PEAR in una naturale evoluzione: *Diagnosis by Numbers*

4.2 Visione complessiva

L'idea di questa tesi, introdotta nel capitolo 1 e ribadita più volte, finalmente inizia a prender forma. Data un'ontologia, più o meno vasta, arricchita da espressioni di tipicalità e dai sintomi/prodromi riguardanti un paziente, l'obiettivo è quello di generare tutte le possibili diagnosi (o spiegazioni), controllarne la veridicità (logicamente parlando) e presentarle in forma grafica, evidenziandone la probabilità e il costo stimato.

Ovviamente il focus principale rimane la correttezza di esecuzione, non tanto la complessità dell'ontologia medica o dei costi diagnostici reali. Questo non toglie che in futuro possa avere delle applicazioni concrete.

Struttura del progetto Il progetto è composto da 12 File; questi, a volte, verranno indicati con un intuitiva abbreviazione, così da rendere la lettura più scorrevole:

- *Main.py* file principale, contenente l'ordine d'esecuzione delle operazioni.
- Due file di testo, di input:
  - *OntologyInput.txt* contiene classi, relazioni tra classi, fatti tipici ed individui;
  - *PatientSetOfSymptoms.txt* composto da coppie individuo-classe.
- Quattro file chiave:
  - *OntologyManager.py* gestisce tutto ciò che riguarda l'ontologia;
  - *InputFromFile.py* si occupa della traduzione del file OntoInput tramite OntoManager;
  - *IncreasedOntology.py* calcola le probabilità di ogni membro tipico e genera tutti gli scenari;
  - *ReasoningOnScenarios.py* calcola le probabilità per ogni scenario ed effettua il ragionamento per ogni scenario arricchito con gli scenari.
- Cinque file di supporto:
  - *AboxMember.py* corrispettivo dell'*ABox*;
  - *TypicalFact.py* assieme alla classe TypicalMember corrispettivo dell'*TBox*;
  - *TypicalMember.py* assieme alla classe TypicalFact corrispettivo dell'*TBox*;
  - *Scenario.py* rappresenta il singolo scenario;
  - *QueryResult.py* memorizza i risultati e genera il grafico associato.

Le successive sezioni del capitolo andranno ad illustrare le funzionalità chiave dello strumento organizzate per file.

4.3 Immissione dei dati

4.3.1 I documenti in ingresso

Partiamo dall'illustrare come vengano memorizzate le informazioni relative all'ontologia dallo strumento e quale sia il corrispettivo del file in termini di logiche descrittive.

```
Classes:
Bipolar Depressed MoodReactivity ProstateCancerPatient Nocturia
Set_as_sub_class:
Bipolar,Depressed
Add_members_to_class:
Greg;Depressed | Luca;ProstateCancerPatient
....
Set_typical_facts:
Typical(ProstateCancerPatient),Nocturia,0.8
Typical(Depressed),Not(MoodReactivity),0.85
......
```

Il file *OntologyInput.txt* è composto da 4 parti:

1. `Classes` altro non è che un elenco di tutte le classi che si andranno ad utilizzare;
2. `Set_as_sub_class` è una lista composta da coppie sottoclasse e classe, vedi il sottoparagrafo 3.1.3;
3. `Add_members_to_class` contiene gli individui e le relative classi di appartenenza;
4. `Set_typical_facts`: contiene l'elenco delle inclusioni di tipicalità.

La stessa ontologia espressa in termini di $\mathbf{KB}(\mathcal{T}, \mathcal{A})$:

- $\mathcal{TB}ox$
  - $Bipolar \sqsubset Depressed$
  - $\mathbf{T}(ProstateCancerPatient) \sqsubseteq_{0.80} Nocturia$
  - $\mathbf{T}(Depressed) \sqsubseteq_{0.85} \neg MoodReactivity$
- $\mathcal{AB}ox$
  - $Depressed(Greg)$
  - $ProstateCancerPatient(Luca)$

In fondo al documento troviamo un nuovo elemento, i costi diagnostici; ogni malattia tipica avrà un costo associato e, alla fine dell'elaborazione, verranno elaborati e combinati per assegnare al singolo scenario un costo indicativo, al fine di dare un ulteriore criterio di preferenza al medico.

```
Set_cost_of:
ProstateCancerPatient: 10000
Depressed: 3000
```

Sotto `Set_cost_of` vi sono le coppie malattia-costo stimato (in €)

Infine il file *PatientSetOfSymptoms.txt* in confronto è più basilare

```
Luca;Not(Bipolar) | Greg;Nocturia
```

Questi fatti verranno manipolati e poi aggiunti all'*Abox* in ogni scenario, solo se consistenti con l'ontologia di base.

4.3.2 La classe dedicata alla traduzione

Il file *InputFromFile.py* contiene due metodi: `build_ontology` e `read_symptoms`. Ogni metodo si occupano del rispettivo file, rispettivamente *OntologyInput.txt* e *PatientSetOfSymptoms.txt*. Infatti, dopo aver letto il contenuto del file di testo, le stringhe vengono elaborate e successivamente data in input ai metodi della classe `OntologyManager` quali `create_class, add_sub_class_, add_member_to_class` ecc. Come ulteriore ottimizzazione, l'esecuzione di `read_symptoms` viene effettuata una sola volta, poiché si è preferito salvare i dati letti in `dict()` (coppie chiave-valore) dedicato tramite il metodo `store_for_reasoning`, che vedremo più in dettaglio nella prossima sezione.

Rispetto alla versione precedente il codice è stato ottimizzato e reso più leggibile anche grazie all'introduzione di due funzioni ausiliare `strip_not` e `strip_typical`. Come si intuisce dal nome lo scopo di queste 1-line-function è di rimuovere le stringhe `"Not"` e `"Typical"` che son superflue e vanno diversamente gestite.

4.4 L' amministrazione dell'ontologia

Nel file *OntologyManager.py*, contenente l'omonima classe, sono presenti numerosi metodi per la creazione e la gestione dei mondi e delle ontologie. Focalizziamo la nostra attenzione sul costruttore:

```
def __init__(self, iri="http://www.example.org/onto.owl"):
        self.typical_facts_list = list()
        self.a_box_members_list = list()
        self.scenarios_list = list()
        self.typical_members_list = list()
        self.cost_dict = dict()
        self.symptoms_dict = dict()
        self.my_world = World()
        self.big_world = World()
        self.onto = self.my_world.get_ontology(iri)
```

notiamo, oltre alla presenza dell'iri 3.1.3, l'utilizzo di una variabile, dal nome esplicativo, per ogni lista necessaria. La novità più importante riguarda l'utilizzo dei mondi. *Owlready2* 3.2 può supportare numerosi mondi isolati, e questo ci aiuta, poiché siamo interessati a caricare diverse versioni della stessa ontologia. Ecco quindi spiegate le due differenti variabili. `my_world` è utilizzata per contenere l'ontologia di base, mentre `big_world` ha lo scopo di racchiudere l'ontoBase arricchita con lo scenario selezionato. Se non si capisce ora quale sia l'utilità di questa scelta, successivamente verrà illustrata in maniera più approfondita.

Passiamo ora ad analizzare i metodi di istanza incaricati della gestione dell'ontologia; la maggior parte sono semplici wrapper delle risorse fornite dalla libreria e

non richiedono ulteriori spiegazioni. Tuttavia, tra questi spiccano i metodi per la gestione dei *fatti tipici* e dei *membri tipici*, operatori non conosciuti dalla nostra **API** di Python. La soluzione migliore, già adotta in PEAR, consiste nell'eliminare l'operatore **T** tramite una particolare traduzione, vista a 2.6.1 e implementata nel metodo `add_typical_fact`. Ricordiamo, brevemente, che un paziente è atipico quando esiste **almeno un** individuo più normale di lui. Per, invece, esprimere il concetto di membro m è un tipico C bisogna dire che m appartiene sia alla classe C (tutti gli elementi) sia alla classe $C1$ (elementi tipici). Questa idea trova la sua realizzazione nel metodo `set_as_typical_member`.

Per ridurre i tempi di lettura e manipolazione da file si è voluto introdurre una mappa dedicata ai sintomi con due funzioni manipolative associate, vediamole:

```
def store_for_reasoning(self, member_name: str, class_id: object):
        self.symptoms_dict.update({class_id: member_name})

def add_symptoms_to_kb(self):
        for class_sy, pname, in self.symptoms_dict.items():
                class_c = self.create_class(class_sy.name)
                not_class_c = self.create_class("Not(" + class_sy.name + ")")
                class_c.equivalent_to = [Not(not_class_c)]
                self.add_member_to_class(pname, not_class_c, symp=True)
                print("Sintomo aggiunto: " + pname + ": " + class_c.name)
```

la prima ha il compito di aggiungere valori al "dizionario" le coppie:
`nome-paziente: classe-sintomo`.
La seconda, invece, per ogni coppia presente nella struttura, crea le classi appropriate e poi aggiunge un membro all'$ABox$ all'ontologia corrente tramite la funzione `add_member_to_class`.

È altrettanto importante focalizzarci sulla gestione dei mondi, effettuata dai metodi:

```
        def save_base_world(self):
         self.onto.save("ontoBase.owl", format="ntriples")

        def create_new_world(self):
         self.onto.destroy()
         self.big_world = World()
         self.onto = self.big_world.get_ontology(
         "file://" + PATH_TO_ONTO + "//ontoBase.owl").load()
```

il primo possiede l'incarico di esportare l'ontologia, creata nell'omonimo mondo nel punto 4.3.2 Il secondo assume la funzione di distruggere l'ontologia contenuta nella variabile *onto*, di creare un nuovo mondo e di caricare l'ontologia salvata precedentemente sul file. Vedremo meglio più avanti quale sia stata la strategia dietro lo sviluppo di questi metodi.
Analogamente a questo concetto citiamo la seguente funzione:

```
def consistency(self, condition: bool = False):
        try:
        with self.onto:
                if condition:
```

```
                        sync_reasoner(self.my_world)
                        classi_incosistenti = list(
                                self.my_world.inconsistent_classes())
                        if not len(classi_incosistenti) == 0:
                                return classi_incosistenti
                else:
                        sync_reasoner(self.big_world)
                  return "The ontology is consistent"
        except OwlReadyInconsistentOntologyError:
                 return "The ontology is inconsistent"
```

I ragionatore *Hermit* è in grado di controllare la consistenza di un ontologia e dedurre nuovi fatti, riclassificando individui e classi in base alle loro relazioni. In caso di inconsistenza viene lanciata l'eccezione appropriata. È possibile avere classi inconsistenti senza rendere l'intera ontologia inconsistente ,a patto che queste non abbiano individui. In nostro aiuto il reasoner inferisce tali classi come equivalenti to **Nothing** (elemento primitivo di OWL) ed è quindi possibile interrompere l'esecuzione del tool prima che l'intera ontologia sia inconsistente! Vediamo un esempio. Se avessi una **KB**:

TBox

- $\neg A \sqsubseteq B$
- $B \sqsubseteq A$
- $\mathbf{T}(C) \sqsubseteq_{0.80} D$

ABox

- $D(Giovanni)$

La classe $\neg A$, nonostante non abbia elementi, è inconsistente poiché in contraddizione con l'assioma $B \sqsubseteq A$. Questo chiude la trattazione del manager.

4.5 La creazione dei membri tipici e degli scenari

Il file *IncreasedOntology.py* è un modulo che fornisce i metodi necessari per la generazione di tutti i membri tipici e gli scenari possibili e per il calcolo delle relative probabilità. I metodi incaricati del valutazione delle "percentuali" sono:

```
def compute_probability_for_typical_members(onto_manager):
        facts_list = onto_manager.typical_facts_list
        abox_members_list = onto_manager.a_box_members_list
        facts_list.sort(key=lambda x: x.t_class_identifier.name)
        abox_members_list.sort(key=lambda x: x.class_identifier.name)
        ......
    __set_probability( prob_to_assign_to_typical_member, onto_manager,
            facts_list[i].t_class_identifier,
            facts_list[i].class_identifier)
        ....

def __set_probability(prob_to_assign, onto_mng, t_class_id, class_id):
```

```
for aboxMember in onto_mng.a_box_members_list:
        if aboxMember.isSymptom is True and
         aboxMember.class_identifier.name == class_id.name:
                onto_mng.typical_members_list.
                append(TypicalMember(
                        t_class_id,
                        aboxMember.member_name,
                        prob_to_assign
                ))
```

Il primo processo è il più complicato e si avvale del secondo come supporto. In sintesi per computare la probabilità da associare ad ogni membro tipico si vanno a moltiplicare tra loro le probabilità di ogni inclusione di tipicalità avente la medesima classe argomento dell'operatore $\mathbf{T}$ (ovvero `t_class_identifier`) e, a quel punto, il metodo di supporto,viene invocato per creare il membro tipico.

Ricordiamo che m è tipico se è nella forma $p : \mathbf{T}(C)(m)$. Tale funzione, quindi, scansionando tutti i membri dell'$ABox$, cerca quello che è un sintomo e ha come classe di appartenenza C. Una volta trovato, passa alla creazione Membro tipico, con la probabilità calcolata precedentemente.

La seconda parte della classe ha la responsabilità degli scenari, realtà possibili dei fatti. Il numero totale cambia al variare dei `TypicalMember` e del numero di Sintomi, in particolare, con n membri tipici è 2^n. Considerando, ad esempio, $mt1$ e $mt2$ l'insieme risultante sarà la combinazione:

$$\{\{\emptyset\}, \{mt1\}, \{mt2\}, \{mt1, mt2\}\}$$

dove $\emptyset$ rappresenta lo scenario vuoto, in cui non si fa alcuna assunzione, $\{mt1\}$ in cui si assume solo $mt1$, $\{mt2\}$ in cui si assume solo $mt2$ e infine l'ultimo dove si assumono entrambi.

Per quanto riguarda la probabilità da associare ad ogni scenario, essa è data dal prodotto di k fattori, con k pari al numero di membri tipici generati (nell'esempio precedente avendo due `TypicalMember`, la probabilità di ognuno dei quattro scenari è data dal prodotto di $k = 2$ fattori) dove ogni fattore è una probabilità che coincide con la p appartenente al membro tipico se assunto, altrimenti vale $1 - p$ se questo, nello scenario, non viene assunto.
Riprendendo l'esempio precedente, supponendo che $mt1$ abbia come probabilità il valore $p1$ e che $mt2$ abbia il valore $p2$, la probabilità dello scenario vuoto vale $(1 - p1)\ (1 - p2)$, quella dello scenario in cui si assume solo $mt1$ vale $p1 \times (1 - p2)$, quella in cui si assume solo $mt2$ vale $(1 - p1) \times p2$ ed infine l'ultimo scenario $p1 \times p2$.

I metodi che si occupano di questa traduzione sono 4, 1 principale e 3 di supporto "privati":

```
def set_probability_for_each_scenario(ontology_manager):
def __generate_scenario(ontology_manager):
def __difference(scenario, typical_members_list):
def __get_typical_member(key, ontology_manager):
```

Per prima cosa il metodo `set_probability_for_each_scenario` genera tutti gli scenari tramite il metodo `__generate_scenario` (determinando l'insieme delle par-

ti), successivamente gli scenari vengono così processati: viene moltiplicata la probabilità dei membri tipici assunti e non assunti (grazie a `__difference`) in modo da calcolare la probabilità per ogni scenario. Questo verrà memorizzato nell'oggetto dedicato `Scenario` ovviamente gestito dall'`OntologyManager`.

In particolare il metodo `__difference` effettua una differenza insiemistica tra i due insiemi: membri tipici: assunti e generati. `__get_typical_member`, invece, calcola la probabilità del membro tipico non assunto nello scenario corrente grazie alla presenza delle chiavi identificative `key`: cerca all'interno dell'`ontology_manager` l'oggetto `TypicalMember` corrispondente a quella chiave.

Abbiamo deciso di ignorare lo scenario $\emptyset$ poiché è il più banale, già scartato in altri contesti. Infatti è corretto logicamente generare tale scenario ma, verificato o meno se sia conseguenza logica della KB, non ha molto senso dare una diagnosi del tipo: con una probabilità X non ha niente.

4.6 L'inferenza

Il file *ReasoningOnScenarios.py* si occupa di verificare se i sintomi in input seguano logicamente dalla base di conoscenza, arricchita con l'aggiunta di un determinato scenario. Questo per ogni oggetto `Scenario` presente. Per verificare l'*implicazione logica*, in simbolo $A \models B$, bisogna verificare che in tutti i modelli in cui A è $True$, anche B è vera. Nel nostro caso dobbiamo provare che $KB \cup S \models V$ dove S è un determinato scenario e V è un set di formule che rappresenta i sintomi del paziente. Possiamo dimostrare ciò grazie alla dimostrazione per **refutazione**, che permette di rispondere $True$ se si dimostra l'equivalenza $(KB \cup S \wedge \neg V) \equiv False$. DbN utilizzerà proprio questa strategia, vediamo come.

```
def __translate_scenario(scenario, ontology_manager):
        for tm in scenario.list_of_typical_members:
                ontology_manager.set_as_typical_member(
                        tm.member_name, tm.t_class_identifier,
                        ontology_manager.
                        onto[tm.t_class_identifier.name + "1"])
def is_logical_consequence(ontology_manager,
        lower_probability_bound=0, higher_probability_bound=1):
```

Il metodo, di supporto, `translate_scenarios` ha il compito di tradurre ogni membro tipico presente all'interno dello scenario. Il metodo `is_logical_consequence` gestisce l'intera fase di verifica, opzionalmente esaminando solo gli scenari in un determinato intervallo di probabilità, tramite gli argomenti facoltativi lower and higher `_bound`. I risultati verranno salvati all'interno di `query_result = QueryResult()`. Una volta selezionati gli scenari, questi vengono esaminati uno per volta nel seguente modo:

1. viene creato un nuovo `mondo`, un'universo isolato, in cui si carica l'ontologia di base creata in precedenza;
2. a questa vengono aggiunti i sintomi **negati** e lo scenario corrente, tramite i metodi `add_symptoms_to_kb()` e `__translate_scenario()`;

3. quindi si verifica la consistenza dell'ontologia:
    - se è consistente, cioè $\equiv True$, allora $KB \cup S \not\models V$ e quindi lo scenario viene ignorato e il ciclo riparte **loop**.
4. In caso contrario, si salva l'oggetto `Scenario` all'interno di `query_result`;
5. e si accumula la probabilità dello scenario, appena verificato, nella var `total_probability`, riparte il **loop**;

Alla fine del ciclo si salva la probabilità totale così all'interno della variabile `query_result`. L' esecuzione quindi termina.

4.7 Analisi del risultato

Alla luce di quanto detto prima, vediamo ora la struttura del file *QueryResult.py* e, in particolare, delle sue operazioni.

```
def show_query_result(self):
        .....
def create_and_show_plot(self, patient_symptoms: str, disease_cost: dict):
        # Crea una figura con un secondo asse y
        self.fig = make_subplots(specs=[[{"secondary_y": True}]])
        # Crea l'istogramma, come asse y la probabilità degli scenari
        trace1 = go.Bar(...)
        # Aggiunge quest'ultimo alla figura
        self.fig.add_trace(trace1)
        # Crea il grafico a dispersione, come asse y il costo degli scenari €
        trace2 = go.Scatter(...)
        # Viene unito alla figura, specificando della presenza del diverso asse y
        self.fig.add_trace(trace2, secondary_y=True)
        self.fig.show()
```

La prima funzione ha il compito di stampare, sullo standar output, gli scenari possibili. Più interessante è `create_and_show_plot`, questa, grazie all'utilizzo di Plotly (vedi 3.3 a pag. 30), permette la realizzazione di un grafico interattivo, contente tutte le informazioni significative e facilmente esplorabile. L'idea è stata quella di utilizzare due grafici distinti per rappresentare la probabilità e i costi dei singoli scenari, vedi la Figura 4.2. Successivamente si è pensato di unire i due grafici, poiché accomunati dagli stessi valori sull'asse delle x, semplicemente il numero dello scenario generato; il risultato è stato, all'incirca, questo è mostrato nella Figura 4.3. Per migliorare il risultato finale sono stati aggiunti ulteriori metadati, così da rendere il grafico il più auto-esplicativo possibile. Vale la pena menzionare la possibilità di esportare e salvare i risultati in due formati distinti. Html per una visione dinamica, mentre pdf per una statica.

```
def save_query_result(self,name=None):
        if name is not None:
                # self.fig.write_html(name + ".html")
                self.fig.write_image(name + ".pdf")
```

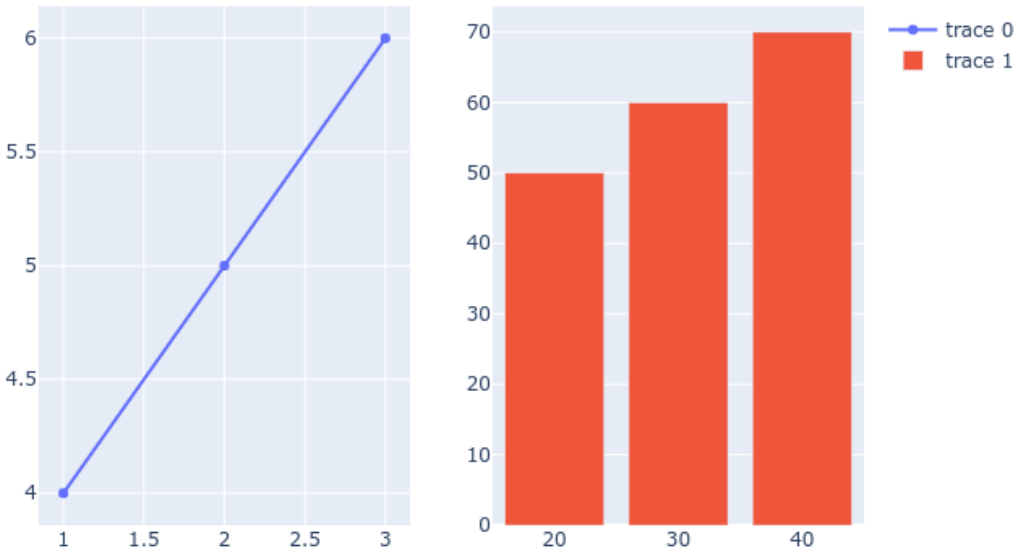


Figura 4.2: Grafico a dispersione e Istogramma separati

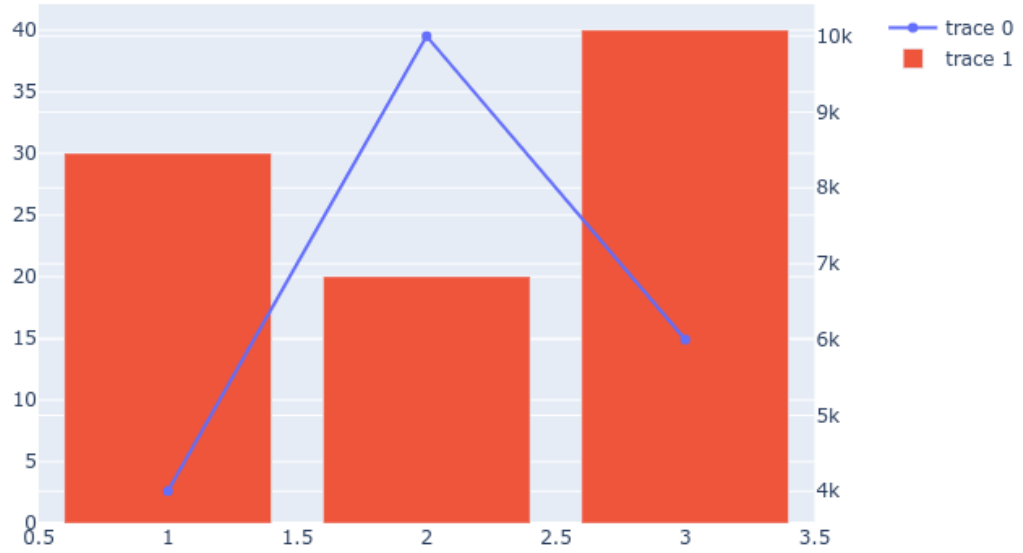


Figura 4.3: Grafico a dispersione e Istogramma uniti

4.8 Il file principale *Main.py*

Composto dal omonimo `__name__ == '__main__'` e dalla nuova funzione di supporto `entailed_knowledge()`.

```
def entailed_knowledge():
        patient_sym = read_symptoms(ontology_manager, result=True)
        result = ontology_manager.consistency(condition=True)
        if not result == "The ontology is consistent":
                # Ontologia inconsistente o classi inconsitenti
                print(result)
                ontology_manager.show_classes_iri_my()
                ontology_manager.show_members_in_classes_my()
                # Termina
                sys.exit(5)
        return patient_sym
```

In quest'ultima effettua la lettura dei sintomi (vedi 4.3.2) e salvato il valore di ritorno si verifica la consistenza della **KB**. In caso negativo si effettuano delle stampe esplicative e si termina l'esecuzione; altrimenti si ritorna i sintomi, sotto forma di stringa.

Il `'__main__'` altro non è che la corretta sequenza di chiamate che permette al tool di funzionare:

```
if __name__ == '__main__':
        ontology_manager = OntologyManager()
        build_ontology(ontology_manager)
        sym: str = entailed_knowledge()
        IncreasedOntology.compute_probability_for_typical_members(ontology_manager)
        IncreasedOntology.set_probability_for_each_scenario(ontology_manager)
        ontology_manager.show_scenarios()
        query_result = is_logical_consequence(ontology_manager)
        query_result.show_query_result()
        query_result.create_and_show_plot(sym, ontology_manager.cost_dict)
```

in primis viene istanziata la classe `OntologyManager`, dopodichè viene invocato il metodo `build_ontology` tramite cui viene costruita l'ontologia ed ecco quindi la chiamata all'attività di supporto. Calcolata le probabilità necessarie per i membri tipici e per gli scenari generati, si passa alla fase di inferenza e successivamente a quella di analisi dei risultati e generazione dei grafici. Questo conclude la simulazione.

Abbiamo trattato lo strumento nel modo più completo possibile, cercando, però, di evitare dettagli o piccolezze trascurabili, focalizzando l'attenzione il più possibile sugli elementi importanti e peculiari.

Capitolo 5

Conclusione e sviluppi futuri

Questo breve capitolo finale serve come resoconto di tutto quello presentato fin'ora e cerca di dare un'idea su come questo tool potrebbe essere impiegato, e sulle possibili future evoluzioni che potrà assumere.

Il focus del progetto è il supporto alle decisioni del medico, l'affiancamento, non la sostituzione di tale figura professionale,

Il considerare i sintomi, normalmente catalogati come atipici, per una certa malattia, sotto una diversa luce potrebbe permettere di scoprire che, in realtà, sono indirettamente collegati ad altre patologie. Per rendere l'idea si pensi ad una persona che soffre di obesità; tendenzialmente un obeso ha problemi di metabolismo, scarsa autostima e alti livelli di colesterolo. L'utilizzo di smartphone durante i pasti è stato riscontrato essere un fattore incidente sull'aumento di peso, poiché i pazienti testati, tendono a prestare meno attenzione a quello che mangiano e, come conseguenza, ingeriscono maggiori quantità di cibo. Questo fatto, indirettamente collegato con l'obesità, potrebbe essere, in alcuni soggetti, un fattore decisivo. Come conseguenza, questa situazione, se ben rappresentata nella KB, potrebbe esser modellata nello strumento come uno scenario poco probabile, ma sempre possibile; la chiave è avere sottomano lo spettro completo delle alternative.

Per quanto riguarda possibili migliorie, le strade percorribili sono numerose; a partire dall'interfaccia utente: la creazione di una GUI e la semplificazione del processo di immissione dati rappresenterebbe un notevole passo avanti per quanto riguarda l'usabilità e abbasserebbe la curva di apprendimento del software. Pensando, invece, all'effettivo test ed utilizzo sul campo, è necessario un lavoro di studio, modifica e creazione di ontologie realistiche o semi-realistiche per dare una parvenza di veridicità alle diagnosi prodotte, soprattutto per quanto riguarda le probabilità delle relazioni e i costi delle diagnosi. Un altro possibile percorso potrebbe essere quello di computare e aggiungere ulteriori metadati ai singoli scenari, come, ad esempio, tempistiche indicative e/o un elenco di esami/visite necessarie alla verifica della "diagnosi". Se dovessimo pensare a delle ottimizzazioni e perfezionamenti, una delle prime idee possibili è l'utilizzodi super-classi **OWL** più significative del generico `Thing`; classi come `Patient`, `MedicalIllness` o `Symptom` scritte in un ontologia di supporto.

Appendice A

Esempio completo

In questa appendice mostreremo un esempio completo di utilizzo di *DbN*, in particolare l'esempio n.6 dell'articolo "Typicalities and Probabilities of Exceptions in Nonmotonic Description Logics" [8] con la seguente $KB = (\mathcal{T}, \mathcal{A})$

$\mathcal{T}Box$:

$Bipolar \sqsupseteq Depressed$

$\mathbf{T}(Depressed) \sqsupseteq_{0.85} \neg\exists hasSymptom.MoodReactivity$

$\mathbf{T}(Bipolar) \sqsupseteq_{0.70} \exists hasSymptom.MoodReactivity$

$\mathbf{T}(ProstateCancerPatient) \sqsupseteq_{0.60} \exists hasSymptom.MoodReactivity$

$\mathbf{T}(ProstateCancerPatient) \sqsupseteq_{0.80} \exists hasSymptom.Nocturia$

$\mathbf{T}(Depressed) \sqsupseteq_{0.60} \exists Smart$

$\mathcal{A}Box$:

$$\{Depressed(Greg), \neg Smart(Greg)\}$$

Set of *Symptoms* $\mathcal{V}$:

$$\{\exists hasSymptom.MoodReactivity(Greg)\}$$

Set of Cost of Disease with *Tipicality*:

$$\{Bipolar : 1000, ProstateCancerPatient : 10000, Depressed : 3000\}$$

Output dello strumento con la seguente ontologia; passo 1, verifica della condizione $KB \cup \mathcal{V} \equiv$ consistente:

```
========== Adding a set of Symptoms to the KB ==========
Sintomo aggiunto: Greg MoodReactivity
========== Checking consistency ==========
========== The ontology is consistent ==========
```

passo 2, generazione scenari possibili:

```
INIZIO SCENARIO 1
Typical(Bipolar),Greg,0.7
Probabilità scenario: 0.364
FINE SCENARIO 1

INIZIO SCENARIO 2
Typical(ProstateCancerPatient),Greg,0.48
Probabilità scenario: 0.144
FINE SCENARIO 2

INIZIO SCENARIO 3
Typical(Bipolar),Greg,0.7
Typical(ProstateCancerPatient),Greg,0.48
Probabilità scenario: 0.3356
FINE SCENARIO 3
```

passo 3, per ogni scenario si effettua l'inferenza, viene mostrato il primo come esempio:

```
ONTOLOGIA PRIMA DELLA LETTURA DELLA QUERY
================================
Greg member_of Depressed
Greg member_of Not(Smart)
Bipolar is_a [ontoBase.Depressed, owl.Thing]
Bipolar1 is_a [owl.Thing, ontoBase.r1.only(Not(ontoBase.Bipolar) & ontoBase.Bipo
IntersectionBipolarBipolar1 is_a [ontoBase.MoodReactivity, owl.Thing]
NotBipolar1 is_a [owl.Thing, ontoBase.r1.some(ontoBase.Bipolar & ontoBase.Bipola
......
================================
FINE ONTOLOGIA PRIMA DELLA LETTURA DELLA QUERY

LETTURA SINTOMI
================================
Sintomo aggiunto: Greg: MoodReactivity
================================
LETTURA SINTOMI TERMINATA

TRADUCENDO LO SCENARIO:
================================
INIZIO SCENARIO
Bipolar,Greg,0.7;
Probabilità scenario: 0.364
FINE SCENARIO
Membro tipico:
Greg is_a Bipolar
Greg is_a Bipolar1
Greg is_a IntersectionBipolarBipolar1
================================
FINE TRADUZIONE SCENARIO
```

```
ONTOLOGIA CON SCENARIO E SINTOMI
================================
Bipolar is_a [ontoBase.Depressed, owl.Thing]
MoodReactivity is_a [owl.Thing]
Not(MoodReactivity) is_a [owl.Thing]
.....
Greg member_of Depressed
Greg member_of MoodReactivity
Greg member_of Not(Smart)
Greg member_of Not(MoodReactivity)
Greg member_of Bipolar1
Greg member_of IntersectionBipolarBipolar1
================================
FINE ONTOLOGIA CON SCENARIO E SINTOMI

Il fatto segue logicamente nel seguente scenario:
INIZIO SCENARIO
Bipolar,Greg,0.7;
Probabilità scenario: 0.364
FINE SCENARIO
```

passo 4, elenco degli scenari verificati e relativo grafico:

```
RISULTATI DELL'INTERROGAZIONE:
SCENARI IN CUI LA QUERY SEGUE LOGICAMENTE
INIZIO SCENARIO 1
Bipolar,Greg,0.7;
Probabilità complessiva dello scenario: 0.364
FINE SCENARIO 1

INIZIO SCENARIO 2
ProstateCancerPatient,Greg,0.48;
Probabilità complessiva dello scenario: 0.14400000000000002
FINE SCENARIO 2

INIZIO SCENARIO 3
Bipolar,Greg,0.7; ProstateCancerPatient,Greg,0.48;
Probabilità complessiva dello scenario: 0.33599999999999997
FINE SCENARIO 3

PROBABILITÀ TOTALE: 0.844

Fine simulazione, tempo totale:  1.984983205795288  s
```

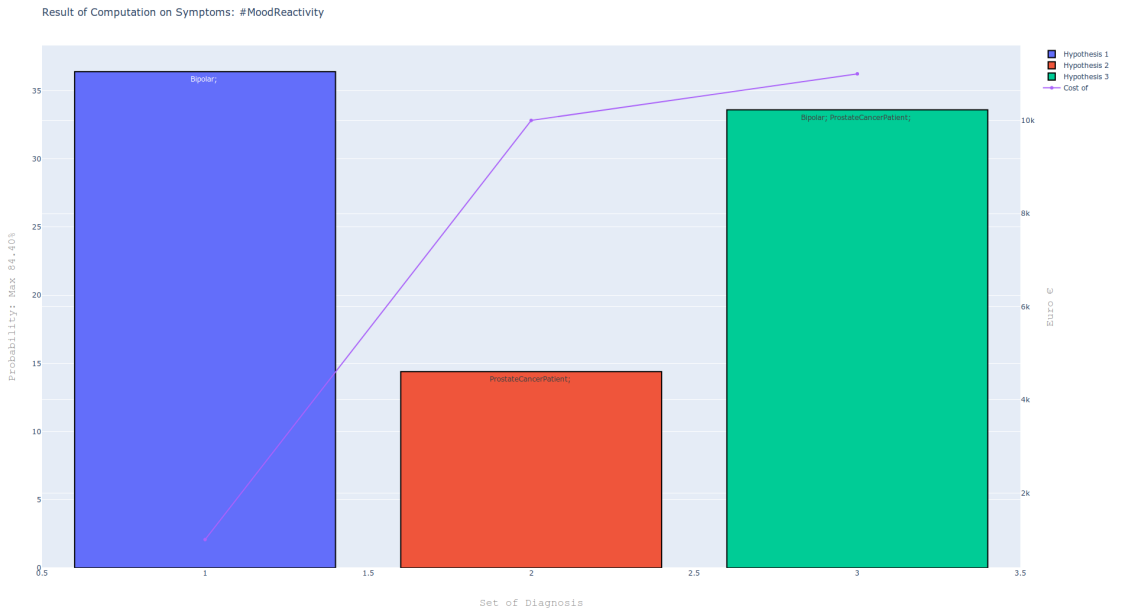


Figura A.1: Versione Web

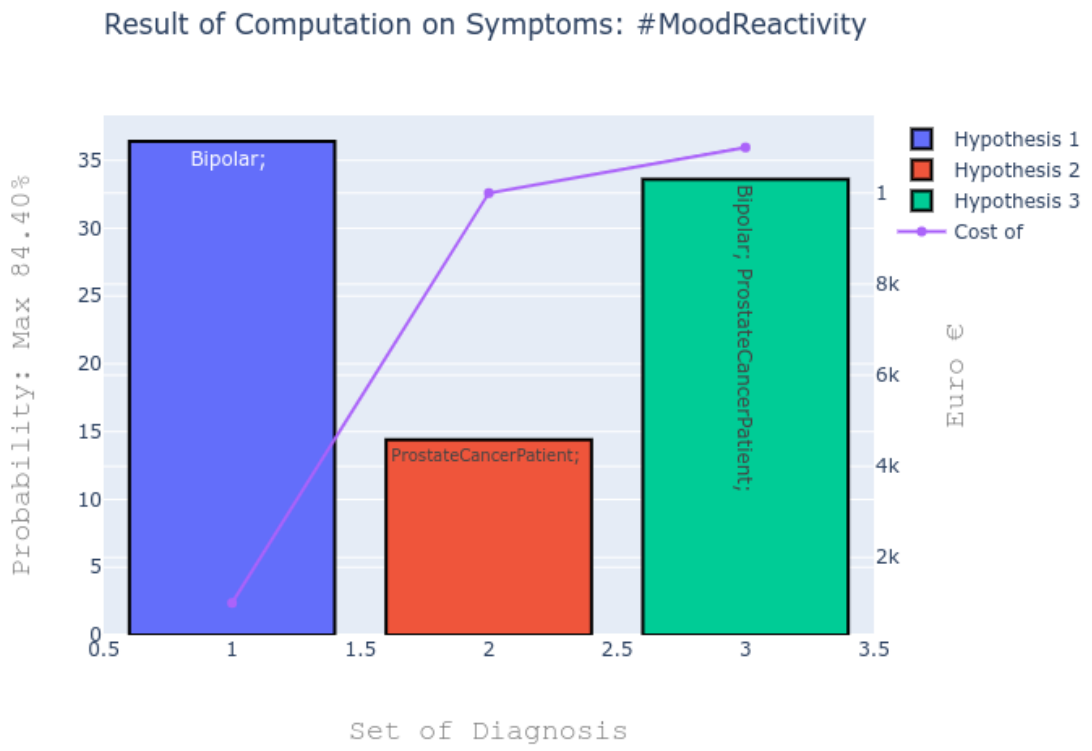


Figura A.2: Versione documento

Bibliografia

[1] Franz Baader et al. *The Description Logic Handbook: Theory, Implementation, and Applications*. Gen. 2007.

[2] Laura Giordano et al. "ALC + T: a Preferential Extension of Description Logics". In: *Fundam. Inform.* 96 (gen. 2009), pp. 341–372. DOI: 10.3233/FI-2009-182.

[3] Laura Giordano et al. "Semantic characterization of rational closure: From propositional logic to description logics". In: *Artificial Intelligence* 226 (mag. 2015), pp. 1–33. DOI: 10.1016/j.artint.2015.05.001.

[4] Plotly Technologies Inc. *Collaborative data science*. [Online; accessed 06-09-2019]. 2015. URL: https://plot.ly/python/.

[5] Jean-Baptiste Lamy. "Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies". In: *Artificial Intelligence in Medicine* 80 (ago. 2017), pp. 1–18. DOI: 10.1016/j.artmed.2017.07.002.

[6] Daniel Lehmann e Menachem Magidor. "What Does a Conditional Knowledge Base Entail?" In: *Artificial Intelligence* 55 (mag. 1992), pp. 1–60. DOI: 10.1016/0004-3702(92)90041-U.

[7] Antonio Lieto, Gian Luca Pozzato e Alberto Valese. "COCOS: a typicality based COncept COmbination System". In: vol. 2214. 33rd Italian Conference on Computational Logic, set. 2018, pp. 55–59.

[8] Gian Luca Pozzato. "Typicalities and Probabilities of Exceptions in Nonmotonic Description Logics". In: *International Journal of Approximate Reasoning* 107 (feb. 2019). DOI: 10.1016/j.ijar.2019.02.03.

[9] Bijan Parsia et al. *OWL 2 Web Ontology Language Primer (Second Edition)*. http://www.w3.org/TR/2012/REC-owl2-primer-20121211/. W3C, dic. 2012.

[10] Gian Luca Pozzato. "Reasoning about surprising scenarios in description logics of typicality". In: Conference of the Italian Association for Artificial Intelligence, 2016, pp. 418–432.

[11] Gabriele Soriano e Gian Luca Pozzato. "Logiche descrittive della tipicalità: sviluppo di uno strumento per il ragionamento sulle probabilità di eccezioni". Laurea in Informatica (triennale, DM270). UniTo, apr. 2019.