UNIVERSITATEA DIN BUCUREŞTI
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENŢĂ
Programare Logică Probabilistă

COORDONATOR ŞTIINŢIFIC

Conf. Dr. Denisa Diaconescu

ABSOLVENT

Ana-Cristina Rogoz

BUCUREŞTI, Iunie 2019

Cuprins

Capitolul 1

Introducere

Tema principală pe care o vom aborda în această lucrare este programarea logică probabilistă, axându-ne pe un framework intitulat ProbLog. Bazele sale au fost puse doar în urmă cu câțiva ani, iar din acest considerent există în continuare interes pentru dezvoltarea acestuia și constant îmbunătățirea sa. ProbLog reușește să încorporeze toate părțile bune ale framework-urile ce l-au preces, asimilând totodată și progresul recent în unele direcții. Are la bază limbajul de programare Prolog, dar se diferențiază de acesta prin adăugarea părții probabiliste specifice, iar contribuția de seamă a acestuia este reprezentată prin introducerea unui nou mod concret de rezolvare pentru calculul probabilității de succes a unei întrebări echivalente în Prolog.

Din punctul de vedere al structurii pe care această lucrare o urmează, se debutează prin aducerea în prim plan a unei serii de noțiuni teoretice din mai multe domenii pe care le-am considerat necesare pentru o mai bună înțelegere a temei principale. Astfel, programarea logică probabilistă încununează atât elemente de probabilități, cât și noțiuni ce țin de teoria grafurilor, dar include și aspecte regăsite în învățarea automată. Ulterior acestei părți teoretice, este descris ProbLog atât din punct de vedere semantic, cât și sintactic, se prezintă algoritmul de inferență al acestuia, dar și diverse modalități în care îl putem utiliza.

În final, se pune accentul pe învățarea automată oferită de ProbLog și se trasează o paralelă între aceasta și învățarea automată clasică prin ilustrarea unui exemplu concret. În acest exemplu vom încerca prezicerea rasei unui cățel pe baza

mai multor atribute precum greutate, înălțime sau longevitate. Vom considera pe rând cele doua abordări și vom observa cum putem modela problema în fiecare caz date fiind restricțiile întâlnite.

Capitolul 2

Noțiuni teoretice

În continuare vom acoperi o serie de noțiuni ce țin de domeniul probabilităților, al programării logice, teoriei grafurilor, dar și de cel al învățării automate.

2.1 Teoria elementară a probabilităților

În acest subcapitol vom acoperi concepte și noțiuni de bază din teoria probabilităților precum operații cu evenimente și formule de calcul pentru probabilitățile acestora, conceptul de probabilitate condiționată și evenimente independente.

Următoarele noțiuni de teoria probabilităților pot fi regăsite atât în [1], cât și în [7].

Notații.

Ω - spațiul probelor ce reprezintă mulțimea tuturor rezultatelor posibile într-un context dat

$\emptyset$ - evenimentul imposibil

$\mathcal{P}(\Omega) = \{A \mid A \subseteq \Omega\}$ - mulțimea submulțimilor de evenimente posibile din spațiul probelor Ω

Definiție 1. Fie Ω mulțime. $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ se numește *corp borelian* sau *σ-algebră* dacă și numai dacă următoarele afirmații sunt adevarate:

1. $\mathcal{F} \neq \emptyset$
2. $A \in \mathcal{F} \implies \overline{A} \in \mathcal{F}$, unde $\overline{A}$ complementul mulțimii A

3. $A_1, A_2, ..., A_n \in \mathcal{F} \implies \bigcup_n A_n \in \mathcal{F}$

$(\Omega, \mathcal{F})$ se numește spațiu măsurabil când $\mathcal{F}$ este corp borelian pe Ω.

Definiție 2. Fie $(\Omega, \mathcal{F})$ spațiu măsurabil. Se numește *funcție de probabilitate* orice funcție $\Pr : \mathcal{F} \to \mathbb{R}$ ce satisface următoarele condiții:

1. $\forall A \in \mathcal{F}$, $Pr(A) \geqslant 0$ (nenegativă)

2. $\Pr(\Omega) = 1$

3. pentru orice secvență finită sau numărabilă de evenimente reciproc exclusive $A_1, A_2, ..A_n \in \mathcal{F}$ (i.e. $\forall A_i, A_j \in \mathcal{F}, A_i \cap A_j = \emptyset$), $Pr(\bigcup_{i=1}^{n} A_i) = \sum_{i=1}^{n} Pr(A_i)$

$(\Omega, \mathcal{F}, Pr)$ se numește *câmp de probabilitate* dacă și numai dacă Pr este funcție de probabilitate pe spațiul măsurabil $(\Omega, \mathcal{F})$.
În continuare vom considera $(\Omega, \mathcal{F}, Pr)$.

Lemă 1. Două evenimente $E_1, E_2 \in \mathcal{F}$.

$$Pr(E_1 \cup E_2) = Pr(E_1) + Pr(E_2) - Pr(E_1 \cap E_2)$$

Definiție 3. Două evenimente $E_1, E_2 \in \mathcal{F}$ sunt *independente* dacă și numai dacă:

$$Pr(E_1 \cap E_2) = Pr(E_1) \cdot Pr(E_2)$$

Generalizând, fie $E_1, E_2, ..., E_n \in \mathcal{F}$ evenimente, acestea sunt reciproc independente dacă și numai dacă: $Pr(\bigcap_{i=1}^{n} E_i) = \prod_{i=1}^{n} Pr(E_i)$

Observație 1. Dacă $E_1, E_2 \in \mathcal{F}$ sunt evenimente independente au loc egalitățile:

$$Pr(A \cap \overline{\mathrm{B}}) = Pr(A) \cdot Pr(\overline{\mathrm{B}})$$
$$Pr(\overline{\mathrm{A}} \cap B) = Pr(\overline{\mathrm{A}}) \cdot Pr(B)$$
$$Pr(\overline{\mathrm{A}} \cap \overline{\mathrm{B}}) = Pr(\overline{\mathrm{A}}) \cdot Pr(\overline{\mathrm{B}})$$

Definiție 4. *Probabilitatea condiționată* ca evenimentul E_1 să aibă loc știind că evenimentul E_2 se întâmplă are următoarea reprezentare:

$$Pr(E_1 \mid E_2) = \frac{Pr(E_1 \cap E_2)}{Pr(E_2)}$$

Probabilitatea condiționată este bine definită dacă $Pr(E_2) \neq 0$.

Definiție 5. Fie $B_i \subseteq \Omega, \forall i \in I$. $(\mathrm{B_i})_{i \in I}$ se numește *partiție* a lui Ω dacă și numai dacă $(\mathrm{B_i})_{i \in I}$ sunt disjuncte două câte două (i.e. $\forall i, j \in I, B_i \cap B_j = \emptyset$) și $\bigcup_{i \in I} B_i = \Omega$.

Teoremă 1. (*Teorema probabilității totale*). Fie câmpul de probabilitate $(\Omega, \mathcal{F}, Pr)$ și $(\mathrm{B_i})_{i\in I} \subset \mathcal{F}$ partiție cel mult numărabilă a lui Ω astfel încât $Pr(B_i) > 0$, $\forall i \in I$. Atunci, $\forall A \in \mathcal{F}$:

$$P(A) = \sum_{i\in I} P(A \mid B_i) \cdot P(B_i)$$

Teoremă 2. (*Teorema lui Bayes*) Fie câmpul de probabilitate $(\Omega, \mathcal{F}, Pr)$ și $E_1, E_2 \in \mathcal{F}$ astfel încât $Pr(E_1) \neq 0$ și $Pr(E_2) \neq 0$. Atunci:

$$Pr(E_2 \mid E_1) = \frac{Pr(E_1|E_2)}{Pr(E_1)}$$

2.2 Noțiuni teoretice din programarea logică

2.2.1 Logica de ordinul I

În continuare vor fi prezentate pe scurt noțiuni ce țin de logica de ordinul I, întrucât programarea logica, implicit și Prolog, folosește un fragment al acesteia pentru reprezentare. Majoritatea informației este preluată din [2].

Sintaxa logicii de ordinul I

Un **limbaj de ordinul I** $\mathcal{L}$ conține o mulțime numărabilă de variabile V, conectorii $\wedge, \vee, \rightarrow, \neg$, paranteze, cuantificatorul universal $\forall$ și cel universal $\exists$. De asemenea, adăugând celor anterioare tuplul $\tau = (R, F, C, ari)$ se determină în mod unic un limbaj de ordinul I. Acest tuplu poartă numele de **alfabetul limbajului** $\mathcal{L}$ și este alcătuit dintr-o **mulțime de relații** R, o **mulțime de funcții** F, o **mulțime de constante** C și o **funcție** *ari* ce întoarce pentru orice element din R sau F o valoarea naturală strict pozitivă reprezentând numărul parametrilor necesari.
În continuare putem defini **termenii** lui $\mathcal{L}$ astfel:

- orice variabilă este un termen
- orice constantă este un termen
- fie $f \in F$, $ari(f) = k$ și $t_1, t_2, ..., t_k$ termeni, atunci $f(t_1, t_2, ..., t_k)$ este termen

În sintaxa Prolog termenii compuși sunt predicate, iar operatorii reprezintă simboluri de funcții. După care, definim noțiunea de **formulă atomică** într-un

limbaj dat $\mathcal{L}$. Fie $P \in R, ari(P) = k$ și $t_1, t_2, ..., t_k$ termeni, atunci $P(t_1, t_2, ..., t_k)$ este formulă atomică.

Extinzând definiția de mai sus, putem defini formulele în general pentru un limbaj de ordinul I $\mathcal{L}$ astfel:

- toate formulele atomice sunt formule
- pentru orice formulă φ, negația sa $\neg\varphi$ **este formulă**
- pentru oricare două formule φ și ψ, $\varphi \wedge \psi, \varphi \vee \psi, \varphi \to \psi$ **sunt formule**
- pentru orice formulă φ și orice variabilă x, $\forall x\varphi$ **și** $\exists x\varphi$ **sunt formule**

Semantica logicii de ordinul I

Considerând toate noțiunile de mai sus, lumea respectivului limbaj de ordinul I poate fi modelată prin atribuirea unor interpretări în anumite structuri. Se numește **structură** un tuplu de forma $\mathcal{A} = (A, F^{\mathcal{A}}, R^{\mathcal{A}}, C^{\mathcal{A}})$ unde:

- A este o mulțime nevidă
- $F^{\mathcal{A}}$ este mulțimea de operații pe A ce include toate funcțiile $f \in F, ari(f) = k$ pe care le vom nota drept $f^{\mathcal{A}} : A^k \to A$,
- $R^{\mathcal{A}}$ este mulțimea de operații pe A ce include toate relațiile $P \in R, ari(P) = k$ pe care le vom nota drept $P^{\mathcal{A}} \subseteq A^k$,
- $C^{\mathcal{A}} = \{c^{\mathcal{A}} \in A | c \in C\}$ mulțimea constantelor pe A.

O **interpretare** peste $\mathcal{A}$ se va nota conform:

- $I : V \to A$ pentru variabilele limbajului
- $t_I^{\mathcal{A}}$ pentru termeni, considerând interpretarea variabilelor în I

Cunoscând faptul ca pornind de la o interpretare într-o structură se poate defini prin inducție după modul de formare al formulelor relația de satisfacere $\models$ dintre o structură, o interpretare și o formulă, se vor mai defini și următoarele noțiuni:

O **formulă** φ **este adevarată în** $\mathcal{A}$ **și interpretarea** I dacă $\mathcal{A}, I \models \varphi$. În

cazul acesta $(\mathcal{A}, I)$ este **model** pentru φ.

O **formulă φ este adevarată în structura $\mathcal{A}$** dacă este adevarată sub orice interpretare I și vom nota $\mathcal{A} \models \varphi$, unde $\mathcal{A}$ se va numi **model** al lui φ.

O **formulă φ este adevarată în logica de ordinul I** dacă este adevarată în orice structură.

O **formulă φ este satisfiabilă** dacă există o structură $\mathcal{A}$ și o interpretare I în aceasta astfel încât $\mathcal{A}, I \models \varphi$.

Modele Herbrand

În continuare vom defini ceea ce se numește **universul Herbrand**.

Fie $\mathcal{L}$ un limbaj de ordinul I ce presupunem că are cel puțin un simbol de constantă (iar în caz că nu are îi adăugăm unul), universul Hebrand asociat $\mathcal{T}_\mathcal{L}$ este **mulțimea tuturor termenilor fără variabile**.

Asemănător, o **structură Herbrand** este o structură de forma $\mathcal{H} = (\mathcal{T}_\mathcal{L}, F^\mathcal{H}, R^\mathcal{H}, C^\mathcal{H})$, unde orice constantă $c \in C$ își păstrează interpretarea și în $C^\mathcal{H}$, iar pentru orice $f \in F, ari(f) = k$ și $t_1, t_2, ..., t_k$ termeni, avem $f^\mathcal{H}(t_1, ..., t_n) = f(t_1, ..., t_n)$.

Pentru o structură $\mathcal{H}$, se numește **interpretare Herbrand** o interpretare de forma $H : V \to T_\mathcal{L}$. O altă noțiune ce se poate defini pe baza unei structuri Hebrand $\mathcal{H}$ este cea de **model Herbrand** al unei formule φ. O structură Herbrand $\mathcal{H}$ este model al formulei φ dacă $\mathcal{H} \models \varphi$.

2.2.2 Logica clauzelor definite

Programarea logică folosește doar un fragment al logicii de ordinul I și anume **logica clauzelor definite** ce se mai numește și **logica clauzelor Horn**. Principalele aspecte ale acestui fragment sunt următoarele:

- **se renunță la cuantificatori**, dar se păstrează variabilele
- **se renunță la** $\vee, \neg$, dar se pastrează $\wedge, \to$
- singurele formule admise sunt de forma:
  1. $P(t_1, t_2, ..., t_k)$, formule atomice

2. $\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_k \rightarrow \beta$, unde $\alpha_1, ..., \alpha_k, \beta$ sunt formule atomice, pentru care variabilele din $\alpha_1, ..., \alpha_k$ sunt gândite ca și cum ar fi cuantificate existențial, iar cele din β ca și cum ar fi cuantificate universal

În logica de ordinul I, un **literal** este o formulă atomică sau negația unei formule atomice. Mai departe, o formulă este în **forma normală conjunctivă** dacă este o conjuncție de disjuncții de literali, iar o disjuncție de literali poartă numele de **clauză**.

Clauzele Horn se încadrează într-una dintre următoarele categorii:

1. *clauză program definită*:
   - $Q_1 \wedge ... \wedge Q_n \rightarrow P$, unde $Q_1, ..., Q_n, P$ formule atomice
   - $\top \rightarrow P$, P formulă atomică
2. *clauză scop definit* (întrebare, țintă): $Q_1 \wedge ... \wedge Q_n \rightarrow \bot$

De menționat este faptul că o clauză poate fi exprimată fie ca o mulțime de literali $\{\neg Q_1, ..., \neg Q_n, P_1, ..., P_k\}$, fie ca o formulă $Q_1 \wedge Q_2 \wedge ... \wedge Q_n \rightarrow P_1 \vee ... \vee P_k$.

Astfel, programarea logică modelează totalitatea cunoștințelor sub forma unei mulțimi de clauze definite pentru care se dorește aflarea răspunsului unei întrebări de forma $Q_1 \wedge ... \wedge Q_n \rightarrow P$, unde $Q_1, ..., Q_n, P$ formule atomice. Toate variabilele prezente în mulțimea cunoștintelor sunt cuantificate universal, iar cele din întrebare sunt cuantificate existențial.

Prolog se bazează deopotrivă pe logica clauzelor definite, iar sistemul de deducție folosit pentru aflarea răspunsurilor la întrebările considerate are la bază regula *back-chain*. Această regulă este implementată în programarea logică cu ajutorul rezoluției SLD.

2.2.3 Rezoluția SLD

SLD (Selective, Linear, Definite clause resolution) este regula de bază pentru inferență întâlnită în programarea logică. Fie T o mulțime de clauze definite, regula rezoluției SLD este următoarea:

1. dată fiind clauza țintă: $\neg L_1 \vee ... \vee \neg L_i \vee ... \neg L_n$,
2. având literalul L_i și o clauză definită $L \vee \neg Q_1 \vee ... \vee \neg Q_k$,
3. obținem $(\neg L_1 \vee ... \vee \neg Q_1 \vee ... \vee Q_K \vee ... \vee L_n)\theta$,

unde $L \vee \neg Q_1 \vee ... \vee \neg Q_k \in$ T (în care toate variabilele au fost redenumite) și θ este cel mai general unificator pentru L_i și L [2].

Definiție 6. Fie T o mulțime de clauze definite și $Q_1 \wedge Q_2 \wedge ... \wedge Q_k$ o întrebare, unde $\forall i \in \overline{1,k}, Q_i$ formulă atomică. Atunci:

- O *derivare* din T prin rezoluția SLD este o secvență $G_0 = \neg Q_1 \vee \neg Q_2 \vee ... \vee \neg Q_k$, $G_1, ..., G_k ...$ în care G_{i+1} se obține din G_i prin rezoluția SLD.
- Dacă există un k cu $G_k = \square$, atunci derivarea se numește *SLD-respingere*.

În logica clauzală respingerea SLD se folosește în scopul demonstrării faptului că un set de clauze este nesatisfiabil. Pe de altă parte, în programarea logică aceasta este folosită pentru semnalarea găsirii unei rezolvari.

Teoremă 3. *(Completitudinea rezoluției SLD).* Sunt echivalente următoarele:

- există o SLD-respingere a lui $P_1 \wedge ... \wedge P_m$ din T,
- T $\models P_1 \wedge ... \wedge P_m$.

Exemplu 1.

1. r :- p, q.
2. s :- p, q.
3. v :- t, u.
4. w :- v, s.
5. t.
6. q.
7. u.
8. p.

? - w.

Pasul 1: $G_0 =$ ¬ w

Pasul 2: $G_1 =$ ¬ v ∨¬ s (aplicăm rezoluția SLD pentru G_0 și clauza 4)

Pasul 3: $G_2 =$ ¬ t ∨¬ u ∨¬ s (aplicăm rezoluția SLD pentru G_1 și clauza 3)

Pasul 4: $G_3 = \neg$ u $\vee\neg$ s (din clauza 5)

Pasul 5: $G_4 = \neg$ s (din clauza 7)

Pasul 6: $G_5 = \neg$ p $\vee\neg$ q (aplicând rezoluția SLD pentru G_5 și clauza 2)

Pasul 7: $G_6 = \neg$ p (din clauza 6)

Pasul 8: $G_7 = \square$ (din clauza 8)

2.2.4 Arbore de derivare SLD

Pentru săvârșirea unui arbore SLD presupunem că avem o mulțime de clauze T și o țintă $G_0 = \neg Q_1 \vee \neg Q_2 \vee ... \vee \neg Q_k$, $G_1, ..., G_k$.... Având acestea la dispoziție, arborele se va construi astfel:

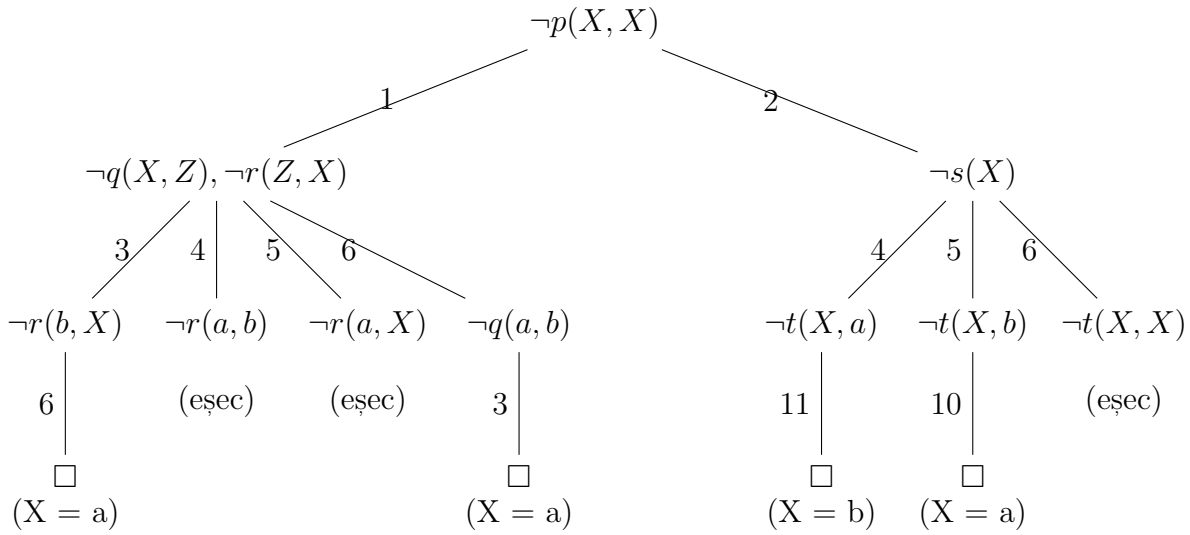
- Fiecare nod al arborelui este o țintă.
- Rădăcina este G_0.
- Dacă arborele are un nod G_i, iar G_{i+1} se obţine din G_i folosind regula SLD și o clauză $C_i \in T$, atunci nodul G_i are copilul G_{i+1}, iar muchia dintre G_i şi G_{i+1} este etichetată cu C_i.

În momentul în care am ajuns să avem o frunză $\square$ (clauza vidă) considerăm că am găsit o SLD-respingere.

Exemplu 2. Alegem următorul exemplu preluat din [2]:

1. p(X, Y) :- q(X, Z), r(Z, Y).
2. p(X, X) :- s(X).
3. q(X, b).
4. q(b, a).
5. q(X, a) :- r(a, X).
6. r(b, a).
7. s(X) :- t(X, a).
8. s(X) :- t(X, b).
9. s(X) :- t(X, X).
10. t(a,b).
11. t(b,a).

? - p(X, X).



2.3 Noțiuni din teoria grafurilor

Partea de teoria grafurilor va fi folositoare în momentul prezentării algoritmului de inferență din spatele ProbLog.
Informațiile din această secțiune sunt preluate din [10] și [8].

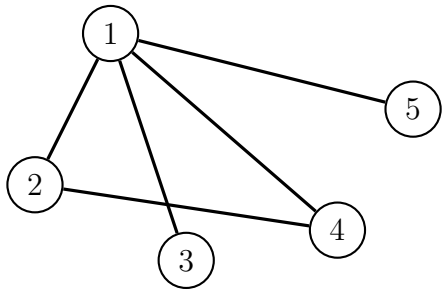
2.3.1 Terminologie

Definiție 7. Numim *graf* tuplul determinat de mulțimea vârfurilor și cea a muchiilor.

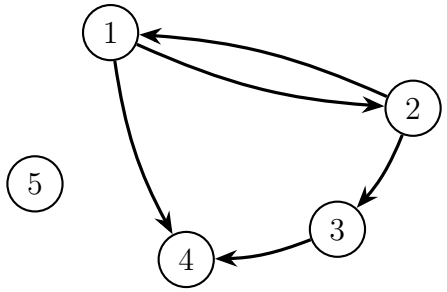
Notații.
G = (V, E) - graful alcătuit din mulțimea vârfurilor V și mulțimea muchiilor E
E = {(u, v) | u, v ∈ V} - mulțimea muchiilor
n = |V| - numărul vârfurilor, m = |E| - numărul muchiilor

În general, putem distinge **două** tipuri de grafuri:

1. **neorientate**: în acest caz perechea (u,v), unde u, v ∈ V este neordonată, iar legătura dintre cele două poartă numele de muchie



2. **orientate**: în acest caz perechea (u,v), u, v ∈ V este ordonată, iar legătura dintre cele două poartă numele de arc



Grafuri neorientate

Pentru acest tip de graf se cunoaște faptul că **gradul unui nod** este egal cu numărul muchiilor incidente cu acel nod. De asemenea, o altă noțiune întâlnită este cea de **lanț** alcătuit dintr-o succesiune de noduri adiacente. Acesta poate fi, în funcție de nodurile parcurse, elementar (când un nod apare o singură dată) sau neelementar (în caz contrar). Mai distingem și noțiunile de lanț simplu (când muchiile ce îl alcătuiesc sunt distincte) și lanț compus (în caz contrar).

Grafuri orientate

Pentru acest tip de graf vom diferenția noțiunea de grad interior (numărul arcelor ce intră) și grad exterior (numărul arcelor ce ies) al fiecărui nod în parte. Corespunzător noțiunii de lanț întâlnită pentru grafurile neorientate, aici vom folosi termenul de drum (elementar sau neelementar).

Ciclu vs. circuit

Fie un lanț simplu în graful neorientat G = (V, E) de forma D = $\{(u_1, u_2), (u_2, u_3), ..., (u_{n-1}), u_1)\}$, acesta se numește **ciclu** deoarece extremitatea inițială coincide cu cea

finală. Echivalent, pentru un graf orientat întâlnim noțiunea de **circuit** atunci când un drum simplu începe și se termină cu același nod.

Conexiune în graf

Numim **graf conex** un graf G=(V, E) în care între oricare două noduri u,v ∈ V există un drum.

Arbore

Fie G=(V, E) un graf neorientat, conex și aciclic, acesta se numește **arbore**.
În particular, se cunosc **arborii binari** ce au proprietate că oricare nod are cel mult 2 fii.

2.3.2 Parcurgeri în graf

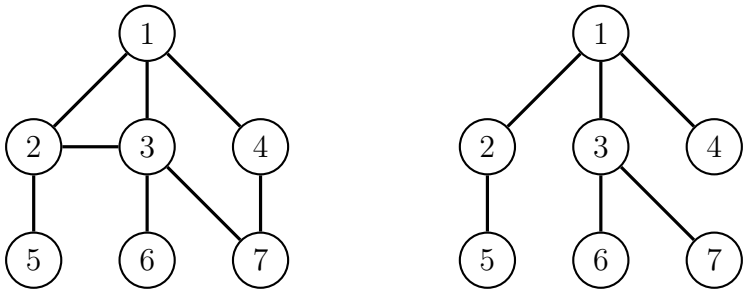
Se cunosc două mari parcurgeri când vine vorba despre grafuri și anume parcurgerea în lățime (Breadth First Search - BFS) și cea în adâncime (Depth-First-Search - DFS). Atât pentru DFS, cât și pentru BFS vom avea nevoie de un vector ce marcheaza dacă un nod a fost vizitat sau nu pentru evitarea ciclării.

Parcurgerea în lățime (BFS)

Structura de date utilizată de către o parcurgere în lățime este o coadă bazată pe principiul primul intrat, primul ieșit (FIFO - first in, first out). Inițial, vectorul pentru vizitarea nodurilor are valoarea *False* pentru toate elementele, iar în stivă se introduce un nod aleator ales. Acesta va reprezenta rădăcina arborelui BFS asociat. După care, în mod recursiv cât timp există elemente în coadă, vom extrage elementul din capătul de ieșire și îi vom căuta toți vecinii nevizitați. Un vecin este un nod pentru care există muchie între el și elementul extras, iar nodul nu apare ca fiind marcat în vectorul pentru vizitare. Fiecare astfel de vecin găsit va fi introdus la capătul cozii. La finalul unui pas nodul inițial extras este cu totul eliminat, iar procedeul se repetă.

Tehnica se numește parcurgere în lățime deoarece la fiecare pas pentru nodul din capătul de ieșire al cozii se epuizează toți vecinii ce se află la distanță 1 înainte de a se trece la cei mai îndepărtați.

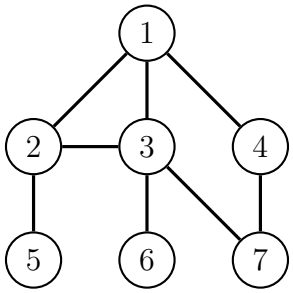
Exemplu 3. Graful inițial și arborele BFS asociat, pornind din nodul 1.



Parcurgerea în adâncime (DFS)

Parcurgerea în adâncime se deosebește de cea în lățime din prisma faptului că pornind dintr-un nod ajungem cât de departe se poate pe o cale inițială, iar de abia în momentul în care aceasta s-a finalizat revenim pas cu pas până când putem începe un nou traseu neexplorat. De asemenea, și în cazul acestei parcurgeri recursivitatea se pornește dintr-un nod considerat rădăcină. Ulterior, pentru nodul curent se caută primul vecin nevizitat și se continuă printr-un apel recursiv pentru acesta. Se va face revenirea la restul vecinilor de abia în momentul în care subarborele primului a fost epuizat.

Exemplu 4. Considerând următorul graf, putem observa diferența dintre ordinea de vizitare a nodurilor pentru parcurgerea BFS și cea DFS pornind în ambele cazuri din nodul 1:



- Parcurgere BFS: 1, 2, 3, 4, 5, 6, 7
- Parcurgere DFS: 1, 2, 5, 3, 6, 7, 4

2.3.3 Diagrame binare de decizie (BDDs)

Pentru o mai ușoară definire a diagramelor binare de decizie vom începe prin a prezenta termenul de arbori binari de decizie. Informația prezentată se bazează pe ceea ce există în [3].

Conceptual, **arborii binari de decizie** păstrează ideea din spatele arborilor binari normali, dar se deosebesc de aceștia întrucât modelează formule boolene. Presupunând că avem variabilele $x_1, x_2, ..., x_n$ ale unei formule boolene, putem construi arborele binar de decizie alegând aleator un nod rădacină, să spunem x_1 ce va avea în descendență doi subarbori, unul pentru $x_1 = 1$ și unul pentru $x_2 = 0$. Continuând în aceeași manieră, vom lua următoarele variabile din formula booleană și vom construi subarbori până ce în final vom ajunge la nodurile frunză ce reprezintă valoarea finală a formulei. Pentru fiecare nod frunză în parte, urmând drumul către rădăcină vom obține de pe muchiile drumului valoarea asignată fiecărei variabile ce a dus la rezultatul frunzei. [3]

Exemplu 5. Considerăm formula: $f : \{0,1\}^2 \to \{0,1\}, f(x_1, x_2) = x_1 \wedge x_2$

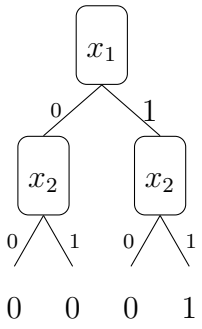


Figura 1. Arborele binar de decizie asociat formulei boolene $x_1 \wedge x_2$

Totuși, arborii binari de decizie întâmpină două mari probleme. Pe de-o parte, pentru un număr dat de n variabile, arborele va ajunge să aibă $2^{(n-1)}$ noduri de decizie și încă 2^n frunze pe ultimul nivel. Pe de altă parte, arborii binari de decizie mențin subarbori nefolositori, precum ramura stângă a nodului rădăcină din exemplul de mai sus, întrucât pentru $x_1 = 0$, rezultatul funcției va fi, indiferent de valoarea de adevăr a lui x_2, 0.

Din aceste motive, ajungem la varianta îmbunătățită a acestor arbori binari de decizie prin intermediul **diagramelor binare de decizie**. Acestea permit omiterea

cazurilor redundante, ce nu influențează în niciun fel rezultatul funcției pe acea ramura, și, de asemenea, acceptă refolosirea nodurilor de catre subarbori identici. În teorie, o diagramă binară de decizie poate avea un număr exponențial de noduri, dar în majoritatea cazurilor din practică s-a observat o îmbunătățire majoră datorită folosirii acestei structuri în defavoarea arborilor binari de decizie.

2.4 Introducere în Învățarea Automată

Informațiile din această secțiune sunt preluate atât din [11], cât și din [4].

2.4.1 Terminologie

Pentru înțelegerea noțiunilor ce vor fi prezentate în capitolele următoare am ales să fac o scurta prezentare a celor mai des întâlnite noțiuni ce țin de învățarea automată. Orice problemă de învățare automată dorește ca pe baza unor **atribute/caracteristici** să poată produce o **etichetă**. Atributele sunt proprietăți observabile, iar cumulul acestora poartă denumirea de vector de atribute. Pe de altă parte, eticheta este ceea ce vrem sa prezicem și poate fi de două feluri: discretă (caz în care algoritmul utilizat va fi unul de clasificare) sau continuă (caz în care algoritmul utilizat va fi unul de regresie).

Considerăm un **set de date** ce conține un număr semnificativ de date pe baza cărora ne vom construi predicțiile. Un **exemplu** (probă) este o instanță a setului de date compusă fie în întregime din vectorul de atribute, fie conține și caracteristici create folosind procedeul de *ingineria caracteristicilor (feature engineering)*.
Pentru stabilirea unei relații între atribute și etichetă se va construi un **model** pe baza setului de date prin procedeul de **antrenare**. Acest procedeu de antrenare se definește concret cu ajutorul unui **algoritm de învățare**, ce are la rândul său o serie de constrângeri pentru modelele admise (unele impuse de hiperparametri). **Hiperparametrii** reprezintă o configurație ce nu ține de model, iar valorile acestora nu pot fi estimate din setul de date.

În final, odată antrenat modelul pentru care am ales algoritmul de învățare și am fixat hiperparametrii, putem trece la pasul final și anume **inferența** sau

deducția. Pentru acest pas modelul antrenat este folosit pentru a clasifica exemple nemaivăzute până atunci.

2.4.2 Tipuri de învățare

1. **Învățare supervizată** - există o etichetă pe care vrem să o prezicem
   - **regresie** - eticheta este continuă (Linear Regression, KNN Regression, Regression Trees)
   - **clasificare** - eticheta este discretă (Logistic Regression, KNN, Decision Trees, SVM)
2. **Învățare nesupervizată** - nu există o etichetă anume, ci vrem să descoperim structura setului de date
   - **clustering** - grupează obiecte similare (KMeans, DBSCAN, Hierarchical Clustering)
   - **reducerea dimensionalității** (Principle component analysis, T-SNE)
3. **Reinforcement learning** - nu există etichete, ci doar rasplată/penalizare pentru acțiunile luate
4. **Învățare semi-supervizată** - majoritatea exemplelor nu au etichete asociate
5. **Învățare prin transfer** - modelul antrenat pentru o sarcină se folosește pentru a învăța mai repede o altă sarcină

2.4.3 Evaluarea performanței

În cazul antrenării unui model există două scenarii nefavorabile: **overfitting** și **underfitting**. În cazul overfitting-ului, modelul se descurcă bine pe setul de antrenare, dar nu este capabil să generalizeze. Pe de altă parte, în cazul underfitting-ului modelul este deplorabil atât pe mulțimea de antrenare, cât și pe cea de testare, nereușind să capteze natura lumii analizate.
Se mai disting de asemenea două tipuri de eroare:

- **eroarea adevărată** - calculată pe toate exemplele teoretic posibile
- **eroarea empirică** - calculată pe un set finit de exemple

2.4.4 Maximum likelihood (probabilitatea maxima)

Adesea numită și funcția de probabilitate multinomiala, aceasta reprezintă reuniunea distribuțiilor de probabilitate pentru o mulțime de variabile aleatoare $X_1, X_2, ..., X_n$, având însemnătatea că acestea se regăsesc fie într-un interval sau îndeplinesc cerințele dorite. În cazul în care avem doar două variabile se numește funcția de probabilitate bivariată.

Având această funcție definită, un procedeu des întâlnit în statistică este maximizarea acestei funcții de probabilitate în ideea în care se dorește găsirea celor mai plauzibili parametri ai funcției pentru setul de date observat.
Facând legătura cu învățarea automată, vom utiliza informație regăsită în [11]. Astfel, putem considera exemplul des întâlnit pentru care se ia în considerare această maximizare a funcției de probabilitate, și anume **regresia logistică**:

- x $\in \mathbb{R}^n$, reprezentând un exemplu alcătuit din n atribute
- y $\in \{0, 1\}$, variabila dependentă (eticheta de prezis)

Modalitatea în care variabila dependentă se prezice de către model este prin aplicarea unei funcții logistice peste un model liniar. Modelul liniar este alcătuit din n parametri $(w_0, w_1, w_2, ..., w_n)$ care modelează funcția ce vrem sa descrie cât mai bine lumea analizată. Astfel, la un moment dat în care avem acei parametri cu anumite valori, eticheta prezisă pentru un exemplu x va fi de forma:

$$\hat{y} = \sigma(w_0 + w_1 * x_1 + ... + w_n * x_n) = \frac{1}{1+e^{-\langle \vec{w}, \vec{x} \rangle}}$$

Eroarea acestui model se va calcula folosind funcția cross-entropy care pentru un exemplu x_i va lua următoarea valoare:

$$\mathcal{L}^{(y_i, \hat{y}_i)} = \begin{cases} -log(\hat{y}_i) & \text{, dacă } y_i = 1 \\ -log(1 - \hat{y}_i) & \text{, dacă } y_i = 0 \end{cases}$$

Observăm că în această variantă ceea ce se dorește este minimizarea acestei funcții,

dar facând paralela cu probabilitatea maximă (aici maximum log-likelihood) se obține echivalent pentru toate exemplele mulțimii de testare:

$$\textbf{max}\ log(\prod_{i=1}^{n} \frac{1}{1+e^{-y_i\langle \vec{w}, \vec{x_i}\rangle}}) \implies \textbf{min} \sum_{i=1}^{n} log(1 + e^{-y_i\langle \vec{w}, \vec{x_i}\rangle})$$

2.4.5 Învățare automată folosind Sklearn

Sklearn (sau varianta sa neabreviată scikit-learn) este o librărie de Python, open-source ce conține suport în pricipal pentru învățarea automată (fie ea supervizată sau nesupervizată). Oferă o multitudine de algoritmi de învățare, dar și elemente pentru preprocesarea datelor și evaluarea modelelor antrenate.

Aceasta va fi principala librărie pe care o vom folosi pentru partea de învățare automată din această lucrare, în particular algoritmii de clasificare: Logistic Regression, Random Forest Classifier și K-Nearest Neighbors.

Capitolul 3

ProbLog

3.1 Aspecte generale

ProbLog, extensia probabilistă a limbajului Prolog, reprezintă o componentă software ce are la bază limbajul de programare Python. Acesta este un limbaj generic cu utilități în multiple arii precum *Dezvoltare Web*, *Învățare Automată*, *Statistică*. Un program în ProbLog este alcătuit din doua componente principale: un set de fapte probabiliste fără variabile și un program logic. O faptă probabilistă este pur și simplu o faptă căreia îi este asignată o probabilitate ca ea să fie adevarată ce se scrie sub forma `p::f.`. De asemenea, se mai acceptă și definiri de forma `p::f(`$V_1, V_2, ..., V_n$`)` atunci când se dorește definirea unui întreg set de probabilități pentru toate faptele non-probabiliste $V_1, V_2, ..., V_n$. Pe de altă parte, cea de-a doua componentă și anume programul logic în sine este compus din toate faptele non-probabiliste și un set de reguli asociat.

Programele în Problog definesc distribuții asupra programelor logice (adică asupra tuturor variantelor posibile ce se pot întâmpla), considerând toate combinațiile existente de asignări ale variabilelor. Mulțimea asignărilor este considerat un univers Herbrand finit.

Astfel, pentru fiecare fapt probabilist putem face o alegere singulară dacă îl vom include sau nu pe baza probabilității asignate. Cumulul tuturor acestor alegeri singulare conduce la obținerea unei alegeri totale, căreia îi este atribuită o probabilitate egală cu produsul tuturor probabilităților faptelor probabiliste în concordanță cu va-

loarea de adevăr care le-a fost atribuită.

Acest aspect reprezintă noutatea adusă limbajului Prolog. Spre exemplu, dacă vrem să aflăm probabilitatea ca o casă să fie dărâmată din cauza unei calamități naturale am putea scrie următorul cod:

```
0.4::cutremur.
0.2::tsunami.
0.1::uragan.

casaDaramata :- cutremur.
casaDaramata :- tsunami.
casaDaramata :- uragan.
```

În codul de mai sus am atribuit fiecărei fapte câte o probabilitate ca aceasta să fie adevarată după care am descris predicatul "casaDaramata". Astfel, dacă ulterior am vrea să răspundem la întrebarea `query(casaDaramata).` am primi răspunsul `casaDaramata: 0.568` . Răspunsul reprezintă probabilitatea ca întrebarea pe care am adresat-o să fie adevarată, luând în considerare cele trei fapte.

ProbLog verifică probabilitatea primei reguli din program `casaDaramata :- cutremur.` care este 0.4, după care trecând mai departe la cea de-a doua regulă caută probabilitatea de a se produce un tsunami, fără a exista un cutremur, iar cea din urmă regulă ia în calcul variantele în care are loc un uragan, fără a exista nici un cutremur și nici un tsunami. În final adună cele trei probabilități reieșite și ajunge la răspunsul final pentru întrebarea adresată în cazul acesta.

3.1.1 Semantica Problog

Un program în ProbLog este un set de clauze definite de forma $p_i :: c_i$, unde p_i reprezintă probabilitatea, iar c_i o clauză definită asemeni unei implicații de forma $u <- p \wedge q \wedge .. \wedge t$. De asemenea, un aspect important este faptul că fiecare clauză este reciproc independentă de oricare alta. Așadar, un program în ProbLog poate fi scris sub forma:

$$T = \{p_1 : c_1, p_2 : c_2, ..., p_n : c_n\}$$

Mulțimea T definește o distribuție de probabilitate peste mulțimea programelor logice L (L $\subseteq L_T$, unde $L_T = \{c_1, c_2, ..., c_n\}$), astfel:

$$Pr(L|T) = \prod_{c_i \in L} p_i \cdot \prod_{c_i \in L_T \setminus L} (1 - p_i)$$

Formula de mai sus poate fi citită în modul următor: valoarea de probabilitate astfel încât clauzele din mulțimea L să fie adevarate, iar restul clauzelor programului logic ce nu fac parte din submulțimea L a lui L_T să fie false.

În continuare, putem extinde estimarea probabilității la cea a unei întrebări adresate unui program logic în Problog. O întrebare în cazul de față este reprezentată de un predicat căruia vrem să-i aflăm probabilitatea de a fi adevarat. Considerând întrebarea q peste programul logic T, putem calcula probabilitatea de succes $Pr(q|T)$ în modul următor: [6]

$$Pr(q|L) = \begin{cases} 1, & \exists\theta : L \models q\theta \\ 0, & altfel \end{cases}.$$

$$Pr(q, L|T) = Pr(q|L) \cdot Pr(L|T)$$

$$Pr(q|T) = \sum_{M \subseteq L_T} Pr(q, M|T),$$

unde L este o submulțime a lui L_T (mulțimea tuturor clauzelor definite ale programului logic T). În limbaj natural, putem spune că probabilitatea de succes a unei întrebări q corespunde probabilității ca aceasta să aibă o demonstrație, dată fiind distribuția de probabilitate peste mulțimea programelor logice ale lui T.

Contribuția de seamă a acestei extensii adusă limbajului Prolog este reprezentată printr-un nou mod concret de rezolvare pentru calculul probabilității de succes a unei întrebări. Pe scurt, ideea din spatele acestui tool se bazează pe rezoluții SLD și metode de calcul a probabilității formulelor booleene, iar algoritmul propus de aproximare a probabilităților este construit ca fiind o combinație între un DFS iterativ (Depth-First-Search) și diagrame binare de decizie, dat fiind recentul progres în această direcție.

Una dintre principalele motivații din spatele acestei extensii a Prolog stă în spatele necesității de a găsi soluții la probleme din viața reală precum cele din dome-

niul biologiei. Acestea necesită deseori un timp îndelungat de căutare a soluțiilor, spre exemplu: determinarea legăturii unei gene cu o anumită afecțiune. ProbLog modelează perfect acest tip de probleme deoarece putem face următoarele asocieri:

unde fiecare nod în parte reprezintă fie o componentă genetică, fie o afecțiune ce poate rezulta în boala țintă (echivalentul unei întrebări în ProbLog), iar muchia orientată $x \rightarrow y$ reprezintă probabilitatea ca, având îndeplinită condiția x, să putem obține afecțiunea y. Graful asociat obținerii unei probabilități pentru o întrebare este orientat, aciclic și conex.

3.1.2 Comparație cu alte framework-uri

În ultimele două decade s-au dezvoltat multe alte framework-uri dedicate programării logice probabiliste precum: PHA[Poole, 1993], PRISM[Sato and Kameya, 2001], SLPs[Muggleton, 1996], probabilistic Datalog (pD) [Fuhr, 2000]. Toate aceste frameworkuri atașează probabilități formulelor logice, de obicei clauzelor definite, dar în majoritatea cazurilor fiecare impune anumite restricții. Spre exemplu, în SLP clauzele ce definesc același predicat nu pot fi considerate simultan adevărate. PRISM și PHA de asemenea nu suportă ca anumite clauze să aibă loc simultan, iar toate aceste restricții simplifică atât calculul probabilității unei întrebări, cât și algoritmul pe baza căruia se fac deducțiile.

Totuși, un framework vag asemănător ProbLog-ului este pD întrucât, spre deosebire de cele anterior menționate nu impune restricții asupra combinațiilor posibile de clauze ce pot fi simultan adevarate. De asemenea, atât ProbLog, cât și pD se deosebesc de restul framework-urilor prin faptul că probabilitatea fiecărei clauze este independentă față de oricare alta. Cu toate acestea, un mare dezavantaj al pD este mecanismul de deducție utilizat. Acesta este unul naiv ce poate evalua cel mult 10 conjuncții. În opoziție, algoritmul de aproximare utilizat de Problog este capabil să furnizeze un rezultat pentru până la 10000 de formule. [6]

3.1.3 Algoritmul de inferență

Framework-ul Problog utilizează în mare parte același mecanism de inferență precum limbajul Prolog, aducând totuși o serie de îmbunătățiri acestuia. Datele inițiale pe care le avem sunt un program logic sub forma unei mulțimi de clauze cu probabilități asociate $T = \{p_1 : c_1, p_2 : c_2, ..., p_n : c_n\}$ și o întrebare de forma $query(q|T)$.

Metoda utilizată este alcătuita din două componente principale. Prima parte calculează pe baza tuturor programelor logice toate demonstrațiile posibile din care reiese o formulă în forma normală disjunctivă. Cea de-a doua componentă este cea ce distinge Problog-ul de Prolog, întrucât presupune calculul probabilității formulei în forma normală disjunctivă obținută cu ajutorul primei părți.

Cunoscând aceste două componente, prima chestiune ce trebuie deslușită este cum se produce o formulă în forma normală disjunctivă pe baza unui program logic în momentul în care este adresată o întrebare. Fie q întrebarea de la care se pornește, se va folosi rezoluția SLD pe baza căreia se va dezvolta arborele de derivare SLD. Ulterior, folosind acest arbore se pot identifica atât traseele ce se sfârșesc cu succes, cât și cele finalizate prin eșec.

ProbLog utilizează modelul clasic de rezoluție SLD prin care se construiește arborele de derivare de sus în jos. Ceea ce stă la rădăcina arborelui este însăși întrebarea q, iar nodurile ce se derivă ulterior se formează aplicând rezoluția SLD între rădăcină și una dintre clauzele programului logic. Acest procedeu se continuă recursiv din fiecare nou nod creat.

Așadar, în final când arborele nu mai poate fi extins, fiecare drum săvârșit de la rădăcină până la una dintre frunzele ce au dus la succes reprezintă una dintre înșiruirile de conjuncții ale formulei în forma normală disjunctivă.

Cea de-a doua problemă ce intervine este calculul probabilității acelei formule reieșite după pasul anterior. Întrucât gasirea tuturor modelelor unei formule în formă normală disjunctivă pentru calcularea respectivei probabilități este o problema NP-tare, metoda concepută în cadrul framework-ului Problog utilizează diagrame binare de decizie. Astfel, formula obținută este modelată sub forma unei astfel de diagrame în care fiecărui nivel îi este asignată variabila x_i din care se continuă doua muchii.

Nodul ce presupune ca variabila x_i să continue luând valoarea de adevăr 1 este numit copilul superior, iar cel ce presupune în mod contrar se numește copilul inferior. Având așadar și această diagramă binară de decizie alcătuită pe baza formulei de la primul pas s-au mai găsit următoarele îmbunătățiri:

- Variabilele cu probabilitate 1 de adevăr poate fi eliminate.
- Putem elimina demonstrațiile care deja au mai fost încheiate. (i.e. dacă avem $b_1 \wedge b_2 \wedge ... \wedge b_n$ și d, formula în FND în care se mențin demonstrațiile obținute, iar $d \models b_1 \wedge b_2 \wedge ... \wedge b_n$, atunci putem elimina calea aceasta).

De asemenea, algoritmul propus parcurge diagrama binară de decizie folosind o căutare în adâncime cu un anumit prag limită. Metoda nu reține demonstrații mai lungi de un anumit număr fixat de pași. Avantajul acestei abordări este faptul că reușește să evite buclele infinite. Pentru claritate, vom prezenta un pseudocod al acestui algoritm preluat din [6]:

- **Calculează Probabilitate:**
  DATE DE INTRARE: Nodul n al diagramei binare de decizie
  Pas 1.1: dacă n este nodul terminal 1, returnează 1
  Pas 1.2: dacă n este nodul terminal 0, returnează 0
  Pas 2: considerăm l (copilul inferior) și h (copilul superior) al lui n
  Pas 3: calculăm prob(h) = Calculează Probabilitate(h) și prob(l) = Calculează Probabilitate(l)
  Pas 4: returnează p_n· prob(h) + (1-p_n)· prob(l)
- **Aproximează:**
  DATE DE INTRARE: întrebarea q, programul T, precizia ϵ
  Pas 1: lungimeDemonstratie = 1, d1 = false
  Pas 2: repetă
    Pas 2.1: d2 = d1
    Pas 2.2: apelează Iterează(q, true, lungimeDemonstratie, d1, d2)
    Pas 2.3: p1 = Calculează Probabilitate(d1)
    Pas 2.4: p2 = Calculează Probabilitate(d2)

Pas 2.5: lungimeDemonstratie ++

cât timp p1 - p2 $\leq \epsilon$

- **Iterează:**

  DATE DE INTRARE: întrebarea q, conjucția c, lungimea demonstrației d, d1, d2 (în FND)

  Pas 1: dacă q este vid și $d1 \not\models$ c atunci d1 = d1 $\vee$ c și d2 = d2 $\vee$ c

  Pas 2: altfel, dacă d < 0 și $d1 \not\models$ c atunci d2 = d2 $\vee$ c

  Pas 3: altfel înseamnă că q = $l, q_1, ..., q_n$

  alegem regula h :- $b_1, b_2, ..., b_m$ astfel încât putem calcula cel mai general unificator dintre h și l, notându-l cu θ d –

  apelăm Iterează$(b_1\theta, ..., b_m\theta, q_1\theta, ..., q_n\theta; c \wedge b; d)$

3.2 Instalare

Pentru a putea folosi toate funcționalitățile pe care ProbLog le pune la dispoziție avem nevoie de o versiune Python 2.7 sau 3 deoarece acesta vine totodata cu întreaga suită de pachete oferite de PyPI (Python Package Index - site-ul oficial afiliat acestui limbaj de programare).

Odată dobândită această cerință minimală putem utiliza ProbLog atât din consolă, cât și dintr-un editor online. Dacă alegem să folosim ProbLog din consolă o putem face prin intermediul comenzii `problog shell` ce va deschide o interfață asemănătoare aceleia pentru Prolog prin intermediul căreia putem da comenzi similare. [9]

```
% Welcome to ProbLog 2.1 (version 2.1.0.34)
% Type 'help.' for more information.
?- consult('test.pl').
?- ordonat([1,2,3]).
True
?-
```

3.3 Moduri de utilizare

Anterior am prezentat cum ne putem folosi de opțiunea `problog shell` pentru a putea utiliza ProbLog într-o manieră asemănătoare Prolog. Însă, ProbLog pune la dispoziție mai multe opțiuni de utilizare precum `sample`, `mpe`, `lfi`, `explain` ce pot fi adăugate cuvântului principal `problog` din linia de comandă [9].

Prima variantă în care putem folosi Problog este cea **default** pe care o putem apela din linia de comandă printr-o sintaxă de felul `problog numeFisier.pl`. Presupunând că fișierul pe care l-am dat conține și o serie de întrebari, comanda rulată va afișa probabilitatea asociată calculată pentru fiecare întrebare în parte. Spre exemplu, dat fiind următorul cod:

```
0.5::heads1.
0.6::heads2.
someHeads :- heads1.
someHeads :- heads2.

query(someHeads).
```

și apelând `problog someHeads.pl`, se va afișa `someHeads: 0.8` având semnificația că predicatul someHeads despre care am intrebat are o probabilitate de 0.8 să fie adevarată. Totuși, dacă dorim să extindem răspunsul putem apela `problog someHeads.pl -v`, unde `-v` provine de la cuvantul `verbose` (i.e. exprimat în multe cuvinte) și vom primi următorul rezultat:

```
[INFO] Output level: INFO
[INFO] Propagating evidence: 0.0000s
[INFO] Grounding: 0.0010s
[INFO] Cycle breaking: 0.0001s
[INFO] Clark‘s completion: 0.0000s
[INFO] DSharp compilation: 0.0061s
[INFO] Total time: 0.0092s
someHeads: 0.8
```

Cu ajutorul acestei comenzi observăm toți pașii din spatele rezolvării unei întrebări. Problog primește un model sub forma programului dat și calculează probabilitățile întrebărilor, având la bază transformarea codului inițial într-o reprezentare a unei baze de cunoaștere. Ulterior, codul transformat în formatul dorit poate fi rezolvat cu ajutorul unui compilator de cunoaștere asociat bazei de cunoaștere pe care am folosit-o. Toate aceste compilatoare de cunoaștere se preocupă de convertirea unor forme generale de "cunoaștere" în forme mai usor de evaluat.

ProbLog pune la dispoziție următoarele reprezentări ale unor baze de cunoaștere propoziționale: SDD(Sentential Decision Diagram) și d-DNNF(Deterministic Decomposable Negation Normal Form). Pentru SDD avem asociat compilatorul de cunoaștere cu același nume, iar pentru d-DNNF avem asociate două posibile compilatoare c2d (sistem ce compileaza forma normală conjunctivă în d-DNNF) și dsharp(de asemenea compilează forma normală conjunctivă în d-DNNF, bazându-se pe Sharp-SAT).

Cea de-a doua variantă în care putem folosi ProbLog este cu ajutorul cuvântului cheie **sample** urmat de numărul de exemple pe care ni-l dorim. Apelând o comandă de felul `problog sample numeFisier.pl -N 10`, rezultatul asociat este o serie de zece exemple, fiecare fiind alcătuit dintr-un subset de fapte cărora li se atribuie o valoare de adevar în concordanță cu probabilitatea pe care o au, și predicatele calculate în concordanță cu faptele din care sunt alcatuite. Continuând cu ajutorul codului prezentat anterior, dacă am apela `problog sample someHeads.pl` vom obține următoarea ieșire:

```
----------------
someHeads.
----------------
someHeads.
----------------
someHeads.
```

După cum se poate observa, ulterior unui apel fără alți parametri, comanda **sample** nu furnizează prea multă informație, ci doar afișeaza întrebările adevarate în urma setului de fapte ales asociat fiecărui exemplu. Totuși, putem îmbunătăți ieșirea acestei comenzi adăugând mai multe opțiuni precum: `--with-probability`, `--with-facts`. Așadar, păstrând același exemplu și executând comanda `problog sample someHeads.pl -N 2 --with-facts --with-probability` obținem următoarea ieșire:

```
someHeads.
\+heads1.
heads2.
% Probability: 0.3
----------------
\+heads1.
\+heads2.
% Probability: 0.2
```

De data aceasta, spre deosebire de prima data, putem observa valoarea de adevăr a faptelor ce au dus la rezultatul predicatului `someHeads` și de asemenea putem vedea probabilitatea ca acel subset de fapte să fie ales. În primul exemplu ales cunoaștem că fapta `heads1` nu este adevarată, `heads2` este adevarată, iar acest lucru duce la îndeplinirea regulii `someHeads :- heads2`. De asemenea, la finalul exemplului este calculată probabilitatea ca valorile de adevar să fie asociate faptelor în modul respectiv (în cazul de față: `\+heads1` are probabilitatea 0.5, `heads2` are probabilitatea 0.6, deci probabilitatea totală a subsetului ales pentru primul exemplu este 0.5 * 0.6 = 0.3).

O altă modalitate în care putem utiliza cuvântul cheie `sample` este cu scopul de a estima probabilitatea întrebărilor pe baza unui set de exemple. Apelând comanda `problog sample someHeads.pl -N 20 --estimate` se vor genera 20 de exemple aleatoare pe baza cărora se va calcula probabilitatea ca întrebarea someHeads să fie adevărată.

```
$ problog sample some_heads.pl -N 20 --estimate
```

```
% Probability estimate after 20 samples (334.2541 samples/second):
someHeads: 0.65
$ problog sample some_heads.pl -N 20 --estimate
% Probability estimate after 20 samples (316.8490 samples/second):
someHeads: 0.9
$ problog sample some_heads.pl -N 200 --estimate
% Probability estimate after 200 samples (332.2214 samples/second):
someHeads: 0.78
$ problog sample some_heads.pl -N 200 --estimate
% Probability estimate after 200 samples (332.9496 samples/second):
someHeads: 0.83
```

În exemplele de mai sus avem două apeluri în care numărul de exemple este 20 și două apeluri în care numărul de exemple este 200. Astfel, putem observa că un numar mai mic de exemple duce la o variație a răspunsului mai mare, pe când un număr ridicat de exemple pe baza caruia se trage o concluzie diferă mult mai puțin de la un apel la altul, iar răspunsul ia valori într-un interval mai restrans.

Alte modalități pentru utilizarea ProbLog este alături de cuvântul cheie `mpe` (Most Probable Explanation) sau alături de cuvântul cheie `lfi` (Learning from interpretations). Prima variantă, `problog mpe numeFisier.pl` va produce un set de fapte care respectă toate regulile date în numeFisier. Anume, pe lângă faptele anotate cu o anumită probabilitate, putem avea și niste fapte a căror valoare de adevăr o știm cu siguranță, iar acestea sunt reprezentate în program sub forma `evidence(fapt1, true)` sau `evidence(fapt1, false)`. În momentul în care apelăm comanda anterioară, ProbLog va încerca să găsească un set de atribuiri ale faptelor ce respectă toate structurile `evidence`. Cea de-a doua variantă anterior menționată este `problog lfi numeFisierDeInvatat.pl numeFisierCuDate.pl numeFisierModelInvatat.pl`. În cazul acesta, numeFisierDeInvatat va conține atât fapte, cât și reguli, dar acum faptele nu vor mai avea probabilitatea cunoscută, ci se dorește să fie învățată pe baza exemplelor existente în numeFisierCuDate.pl. Astfel, pentru a denumi un fapt a cărui probabilitate trebuie învățată putem scrie `t(_) :: fapt.`, iar valoarea inițială va fi aleasă aleator. Totuși, dacă dorim să pornim învățarea de

la o anumită valoare, putem specifica între paranteze în locul caracterului `_`.

În final, o ultimă funcționalitate pe care o putem folosi din linia de comandă este `explain`. Acest cuvânt cheie deservește cu scopul de a oferi mai multe detalii despre modul în care se calculează probabilitățile întrebărilor, arătând în mod explicit probabilitatea fiecărei reguli de-a lungul programului și faptele luate în considerare. Cu toate acestea, în momentul actual, `explain` nu poate fi utilizat pentru programe ce conțin și instrucțiunea `evidence`. Pentru a apela la această funcționalitate putem scrie în linia de comanda `problog explain numeFisier.pl`. Astfel, pentru exemplul folosit până acum, comanda aceasta ar afișa următoarea secvență:

```
Transformed program
-------------------
0.5::heads1.
0.6::heads2.
someHeads :- heads1.
someHeads :- heads2.
query(someHeads).

Proofs
------
someHeads :- heads2. % P=0.6
someHeads :- heads1, \+heads2. % P=0.2



Probabilities
-------------
someHeads: 0.8
```

Aici putem observa toate cele trei părți prin care se trece în momentul în care utilizăm funcționalitatea de `explain`. Astfel, în prima fază programul este rescris, după care pentru fiecare întrebare în parte se va calcula o serie de clauze disjuncte a caror sumă va fi ulterior calculată drept răspuns al întrebării respective.

Capitolul 4

Învățare Automată folosind ProbLog

4.1 ProbLog librărie integrată în Python

Așa cum am anterior menționat, ProbLog reprezintă un framework de Prolog scris în cea mai mare parte în Python. Singura parte a acestuia ce este scrisă într-un alt limbaj, și anume C, este pasul de compilare. Dat fiind acest aspect, ProbLog poate fi utilizat nu numai ca o unealtă de sine stătătoare, ci și direct integrat într-un program scris în Python.

Majoritatea funcționalităților pe care ProbLog le are de oferit se regăsesc în biblioteca cu același nume, `problog`. În caz că acest modul nu este deja descărcat, se va rula comanda `!pip install problog`. Astfel, vom considera în continuare două modalități principale de integrare a unui cod ProbLog într-un program scris în Python, preluate din [5].

4.1.1 Program transmis sub forma unui șir de caractere

Pentru această alternativă vom considera următorul exemplu:

```
from problog.program import PrologString
from problog.core import ProbLog
from problog import get_evaluatable
```

```
program = PrologString("""
                eveniment(poze).
                eveniment(absolvire).
                0.3::sambata; 0.7::duminica.
                0.6::soare(sambata); 0.4::ploaie(sambata).
                0.7::soare(duminica); 0.3::ploaie(duminica).

                activitate:- soare(EV).
                query(activitate).
""")

get_evaluatable().create_from(program).evaluate()
```

Prin intermediul acestui exemplu putem observa structura principală a unui program în Python ce integrează folosind un șir de caractere întocmai o bucată de cod aparținând ProbLog-ului. Componenta principală este funcția `PrologString()` ce procesează intrarea și o modelează în consecință. Ulterior, ultima linie din program este cea ce stabilește automat cel mai potrivit compilator dintre cele oferite de către ProbLog (SDD, d-DNNF) apelând metoda `get_evaluatable()`, modelând apoi codul în consecință prin `create_from(sursa)` și oferind rezultatele corespunzătoare întrebărilor prin intermediul `evaluate()`. Rezultatul este dat sub forma unui dicționar în care cheia este întrebarea, iar valoarea este probabilitatea asociată.

Această abordare în care lăsăm la latitudinea limbajului de programare modul în care se prelucrează codul ProbLog oferit este una destul de superficială. O altă modalitate de evaluare a codului oferit prin intermediul `PrologString()` este de a diseca ultima linie a codului anterior în pași multipli precum:

```
from problog.program import PrologString
from problog.formula import LogicFormula, LogicDAG
from problog.logic import Term
from problog.ddnnf_formula import DDNNF
```

```
from problog.cnf_formula import CNF
...
lf = LogicFormula.create_from(p) # creeaza varianta initiala a programului
dag = LogicDAG.create_from(lf)  # o converteste intr-un graf orientat
    aciclic
cnf = CNF.create_from(dag)      # pe baza grafului se obtine o formula in
    forma normala conjunctiva
ddnnf = DDNNF.create_from(cnf)    # formula este compilata folosind DDNNF

ddnnf.evaluate()
```

Alternativ DDNNF-ului mai putem utiliza fie `NNF.create_from(cnf)`, urmat de `nnf.evaluate()` sau `SDD.create_from(cnf)`, urmat de `sdd.evaluate()`.

Asemănător modului de utilizare din linia de comandă, putem include unele opțiuni direct într-un program scris în Python folosind importuri corespunzătoare. În continuare vom exemplifica pentru `lfi` și pentru `sample`:

LFI (Learning from interpretations)

Încă din numele acestei opțiuni ne putem da seama ca de data aceasta o parte a programului nu va mai fi stabilită, ci va putea fi dedusă pe baza acestor ”interpretări”. Interpretările se referă la diverse scenarii pe baza cărora se pot învăța probabilitățile lipsă ale faptelor probabiliste. Considerăm următorul exemplu:

```
from problog.logic import Term
from problog.program import PrologString
from problog.learning import lfi

model = """
t(_)::cutremur.
t(_)::tsunami.
t(_)::uragan.

casaDaramata :- cutremur.
```

```
casaDaramata :- tsunami.
casaDaramata :- uragan.
"""

cutremur = Term('cutremur')
tsunami = Term('tsunami')
uragan = Term('uragan')
casaDaramata = Term('casaDaramata')

interpretari = [
    [(cutremur, False), (tsunami, False), (casaDaramata, False)],
    [(cutremur, False), (tsunami, False), (casaDaramata, True)],
    [(cutremur, True), (uragan, True), (casaDaramata, True)],
    [(cutremur, False), (tsunami, False), (uragan, True), (casaDaramata,
       True)],
    [(cutremur, False), (tsunami, False), (uragan, True), (casaDaramata,
       True)],
    [(cutremur, False), (tsunami, True), (uragan, True), (casaDaramata,
       True)],
]

score, weights, atoms, iteration, lfi_problem =
   lfi.run_lfi(PrologString(model), interpretari)

print (lfi_problem.get_model())
```

Pornind de la programul inițial, pe primele linii avem trei fapte a căror probabilitate dorim să o învățăm. Aici am ales să folosesc varianta `t(_)`, unde se pleacă de la o valoare aleatoare a probabilității, dar s-ar fi putut opta și pentru `t(x)`, unde x ar fi fost probabilitatea inițială considerată la momentul învațării. Apoi este descris predicatul `casaDaramata`, iar întreaga construcție se consideră ca fiind modelul ce urmează să fie antrenat. Mai jos în cod modelăm fiecare predicat existent în program sub forma unui termen căruia ulterior îi vom putea asigna valori de adevăr.

Partea cea mai importantă a codului de mai sus este reprezentată de variabila `interpretari`. Aceasta este un vector în care fiecare element reprezintă un scenariu ce cuprinde perechi de forma `(predicat, valoare de adevăr)`.
Rulând codul acesta vom obține exact ceea ce căutăm și anume probabilitățile asociate fiecărui predicat:

```
0.166666666666667::cutremur.
0.2::tsunami.
0.833333333333333::uragan.
casaDaramata :- cutremur.
casaDaramata :- tsunami.
casaDaramata :- uragan.
```

Sampling

Cea de-a doua opțiune facil de utilizat în Python este cea de obținere a unor probe bazându-ne pe un program existent. Putem considera că opțiunea `sample` este exact inversul opțiunii `lfi`. Dacă în cazul învățării prin folosirea unor probe ne doream să avem o bază de date ce ilustrează cât mai bine lumea modelată pentru a obține probabilități cât mai precise, opțiunea `sample` va oferi întocmai scenariile respective. Considerând ieșirea programului pentru `lfi` la care vom adăuga `query(casaDaramata).`, putem obține probele astfel:

```
from problog.tasks import sample


model = lfi_problem.get_model()
result = sample.sample(model, n=3, format='dict')
```

Obținând astfel în `result` următoarele:

```
[{casaDaramata: False}, {casaDaramata: False}, {casaDaramata: True}]
```

4.1.2 Program alcătuit folosind structuri de date specifice

Un mod echivalent de transmitere a unui program scris în ProbLog către un script în Python este prin intermediul unor structuri de date specifice. Libraria anterior menționată conține de asemenea structuri menite pentru folosirea variabilelor, a termenilor, a constantelor, dar și pentru formarea disjuncțiilor aferente regulilor. Pe acestea le putem accesa cu ajutorul următoarelor importuri:

```
from problog.program import SimpleProgram
from problog.logic import Constant,Var,Term,AnnotatedDisjunction
```

Pentru a exemplifica voi folosi unul dintre exemplele anterior prezentate:

```
cutremur,tsunami,uragan,casaDaramata,query = Term('cutremur'),
    Term('tsunami'),Term('uragan'),Term('casaDaramata'),Term('query')


p = SimpleProgram()


p += cutremur(p=0.4)
p += tsunami(p=0.2)
p += uragan(p=0.1)
p += (casaDaramata << cutremur)
p += (casaDaramata << tsunami)
p += (casaDaramata << uragan)
p += query(casaDaramata)


get_evaluatable().create_from(p).evaluate()
```

În urma evaluării codului de mai sus obținem aceeași probabilitate pentru predicatul `casaDaramata` ca și în cazul celeilalte abordări, anume 0.568. Așadar, codul în ProbLog poate fi integrat ușor în restul codului Python folosind multiple metode, fără a îi modifica în vreun fel însemnătatea.

4.2 Învățare din interpretări

Pentru aceasta parte ne vom axa pe modalitatea de învățare a probabilităților unor fapte, dat fiind un set foarte mare de scenarii. Se va prezenta mijlocul de învățare a parametrilor, analizând de asemenea cadrul în care are loc învățarea așa cum se regăsește și în [14].

În general, învățarea se bazează pe un set parțial de interpretări, numite de asemenea și dovezi, obținerea unei baze exhaustive fiind în general dificil. Fie un program logic pentru care vrem să calculăm probabilitățile asociate, se va cunoaște universul Herbrand corespunzător. Acesta este o mulțime finită ce reprezintă toate posibilele interpretări ale atomilor, eliminănd existența tuturor variabilelor. De asemenea, se va mai cunoaște și un set de date conținând dovezi, unde fiecare dovadă în parte este alcatuită dintr-o multime de atomi ce au fost unificați cu elemente ale universului Herbrand pentru care se cunosc valori de adevăr.

De obicei, fiecare dovadă în parte este parțială, nereușind să spună cu certitudine valoarea de adevăr a tuturor faptelor regăsite în program.

Astfel, având programul în ProbLog și cunoscând universul finit Herbrand asociat, învățarea pe baza interpretărilor dorește să găsească cea mai plauzibilă combinație de probabilități pe baza mulțimii date de antrenare. Conceptul echivalent din învățarea automată în limba engleză este găsirea probabilităților ce par a fi cele mai plauzibile (*maximum likelihood*).
Formal putem modela problema astfel:

- **Ce cunoaștem**:

  1. un program în ProbLog $\mathcal{P}$ alcătuit dintr-un set de fapte probabiliste $\mathcal{F} = \{p_i : f_i\}$, unde p_i necunoscute $i \in \overline{1,n}$, și un set de reguli $\mathcal{R}$.
  2. setul de date (interpretarile) $E = \{E1, ..., Em\}$, unde pentru $\forall i \in \overline{1,m}, E_i = \{(f_1, e_1), ..., (fk, e_l)\}$ mulțimea perechilor termen-valoare de adevăr, toți termenii regăsindu-se și în faptele existente în program.

- **Ce vrem să aflăm:** combinația de probabilitate maximala $\hat{p} = \langle \hat{p_1}, ..., \hat{p_i} \rangle$ pentru valorile probabilităților lipsă din program. Altfel spus:

$$\hat{p} = arg\ max \prod_{i=1}^{m} P_{T_p}(E_{i,j} = e_j),$$

unde $P_{T_p}(E_{i,j} = e_j)$ reprezintă probabilitatea ca având parametri probabilităților la momentul curent setați conform mulțimii p, în programul $\mathcal{P}$ toți atomii corespunzători dovezii E_i să ia valorile de adevăr menite perechilor din care provin.

4.3 Observabilitatea lumii analizate

Specifică învățării pe baza interpretărilor sau a dovezilor este mulțimea pe care o considerăm drept set de antrenare. Această mulțime trebuie să fie cât mai reprezentativă pentru lumea analizată, iar dovezile disponibile în concordanță.

Pastrând în minte structura unui program scris în ProbLog: mulțimea faptelor probabiliste și mulțimea regulilor și a faptelor non-probabiliste, putem analiza setul de antrenare ca fiind de doua feluri: fie ne oferă o **observabilitate totala** asupra lumii, fie o **observabilitate parțială**.

4.3.1 Observabilitate totală

Observabilitatea parțială presupune ca pentru fiecare dovadă în parte a mulțimii de antrenare să cunoaștem valoarea de adevăr a fiecărei fapte existente în programul dat.

Pentru această variantă, maximul funcției de probabilitate asociată programului pentru aflarea valorilor optime ale faptelor probabiliste se poate calcula direct din mulțimea de antrenare.

Păstrând notațiile de dinainte, avem un set de fapte probabiliste $\mathcal{F} = \{p_i : f_i\}$, p_i necunoscute $i \in \overline{1,n}$, un set de reguli $\mathcal{R}$, dar și setul de date (interpretările/dovezile) $E = \{E1, ..., Em\}$, unde pentru $\forall i \in \overline{1,m}, E_i = \{(f_1, e_1), ..., (fk, e_l)\}$ mulțimea perechilor termen-valoare de adevar. Pentru fiecare faptă probabilistă $p_i :: f_i$ se poate calcula numărul aparițiilor în fiecare dovadă E_j sub orice formă în care variabilele au fost înlocuite cu termeni din universul Herbrand asociat și vom nota acest număr drept $\mathcal{K}_i^j$. Astfel, a fost găsita relația următoare:

$$\hat{p}_i = \frac{1}{Z_n} \sum_{j=1}^{m} \sum_{k=1}^{K_i^j} \delta_{i,k}^j \quad ,$$

unde $\delta_{i,k}^j = \begin{cases} 1 & \text{, dacă } (f_{i,k}, True) \in E_j \\ 0 & \text{, dacă } (f_{i,k}, False) \in E_j \end{cases}$.

Astfel, putem calcula direct pe baza setului de date al dovezilor valori corespunzătoare fiecărei probabilități asociate faptelor din programul dat. Suma este normalizată cu ajutorul termenului Z_n ce reprezintă numărul aparițiilor faptei f_i în întreg setul de date.

4.3.2 Observabilitate parțială

Această variantă este cea ce se întâlnește adesea în cazuri din viața reală. Se poate întâmpla să cunoaștem efectul unei întâmplări, dar cauza nu se poate determina cu exactitate. Acesta este genul de scenariu modelat în cazul observabilității parțiale. Se cunosc valorile de adevăr doar pentru o parte dintre predicatele existente în programul ProbLog respectiv, iar din acest motiv probabilitățile nu mai pot fi direct estimate precum în cazul precedent.

Așadar, pentru estimarea funcției maxime de probabilitate se va folosi un algoritm ce se își pune baza în cunoștințele anterior acumulate până la un moment de timp și dorește să estimeze cele mai probabile valori de adevăr pentru ceea ce nu cunoaștem conform dovezii de la momentul curent.

În continuare se va folosi următoarea notație pentru toate probabilitățile de la momentul de timp `k`: p_i^k, $\forall i \in \overline{1,n}$. Inițial, algoritmul propune o alegere aleatoare a tuturor valorilor de probabilitate p_i^0. După care, la fiecare pas `t` se folosesc probabilitățile de la acel moment pentru estimarea probabilității de adevăr a tuturor predicatelor ce nu au fost observate din fiecare interpretare a mulțimii de antrenare, ce vor fi notate drept $P_{T_{p^t}}(f_{i,k}|E_j)$. Această notație se traduce drept probabilitatea ca în programul T, predicatul f_i interpretat în cel de-al k-lea fel, în interpretarea E_j, având mulțimea probabilităților p_t la momentul de timp t, să fie adevărat.

Cu ajutorul acestor probabilități se vor calcula probabilitățile corespunzătoare pasului $t+1$ după următoarea formulă:

$$p_i^{t+1} = \frac{1}{Z_i} \sum_{j=1}^{m} \sum_{1}^{K_i^j} P_{T_{p^t}}(f_{i,k}|E_j).$$

O variantă de pseudocod ce descrie mai bine toți pașii algoritmului este următoarea:

DATE DE INTRARE: $T = \{p_1 :: f_1, ..., p_n :: f_n\} \cup \mathcal{R}, E = \{E_1, E_2, ..., E_m\}$, t = 0

Pas 1: pentru $i = \overline{1,n}$ execută

$p_i^0 = random(0,1)$ # generează o valoarea aleatoare între 0 și 1

Pas 2: pentru $j = \overline{1,m}$ execută

$dDNNF_j = genereaza\_dDNNF(P_{T_0}(E_j))$

Pas 3: execută cât timp probabilitățile nu converg

incrementează timpul $t = t + 1$

Pas 3.1: pentru $j = \overline{1,m}$ execută

pentru $i = \overline{1,n}$ execută

pentru $k = \overline{1, K_i^j}$ execută

calculează $P_{T_{p^t}}(f_{i,k}|E_j)$ folosind $dDNNF_j$

Pas 3.2: pentru $i = \overline{1,n}$ execută

$p_i^{t+1} = \frac{1}{Z_i} \sum_{j=1}^{m} \sum_{1}^{K_i^j} P_{T_{p^t}}(f_{i,k}|E_j)$

DATE DE IEȘIRE: mulțimea probabilităților $p_i^t, i \in \overline{1,n}$ după pasul t în care valorile probabilităților ce vrem să le estimăm au convers

Capitolul 5

Studiu de caz pentru învățarea automată vs. învățarea din interpretări

În această secțiune ne propunem să abordăm o problemă de clasificare atât prin prisma învățării automate clasice, cât și prin cea a învățării automate regăsite în framework-ul ProbLog ce se bazează pe interpretări. Totuși, cele două metode de rezolvare și răspunsurile obținute vor fi ușor diferite pentru cele doua abordări.

5.1 Setul de date

Setul de date pe care îl vom folosi este alcătuit din 1000 de exemple ce vor fi folosite atât pentru partea de antrenare, cât și pentru cea de testare. Acesta conține nouă atribute pentru fiecare exemplu în parte și anume: *rasa, greutatea (g), înălțimea (cm), longevitatea (ani), nivelul de energie, atenția necesară, lungimea blănii, sexul și numele stăpânului.* Aceste atribute se împart în două categorii: **atribute numerice** și **atribute categoriale**. Atributele numerice sunt coloanele reprezentate de greutate, înălțime, longevitate, iar printre acestea există valori lipsă pentru coloana *înălțime*. Atributele categoriale sunt toate celelalte și pot lua valori potrivit următoarelor mulțimi discrete:

1. **Rasa**: Amstaff: 252 exemple, Pug: 246 exemple, Bloodhound: 250 exemple,

Jack Russel Terrier: 252 exemple

2. **Nivelul de energie**: scăzut (80 exemple), mediu (616 exemple), ridicat (304 exemple)

3. **Atenția necesară**: scăzută (23 exemple), moderata (273 exemple), sporită (704 exemple)

4. **Lungimea blănii**: scurtă (895 exemple), medie (105 exemple)

5. **Sexul**: feminin (509 exemple), masculin (491 exemple)

6. **Numele stăpânului**: 511 valori distincte

5.2 Abordare folosind Sklearn

Problema căreia ne adresăm este prezicerea raselor câinilor bazându-ne pe o serie de atribute. Deoarece rasa este un atribut categorial, vom folosi algoritmi de învățare potriviți acestui tip de problemă și anume Logistic Regression, Random Forests și KNN. Pentru toți acești alogritmi eticheta ce vrem să reiasă pentru fiecare exemplu în parte poate lua valori într-o mulțime discretă. Din aceste considerente, pentru a utiliza setul de date am început printr-o serie de **preprocesări**.

1. Am completat valorile lipsă pentru coloana înălțime conform valorii medii pentru fiecare specie în parte.

2. Am asignat fiecărui atribut categorial o mulțime discretă de numere naturale drept echivalent al mulțimii formate din respectivele șiruri de caractere.

3. Am normalizat setul de date folosind două variante: *StandardScaler* și *MinMaxScaler*, ambele preprocesări oferite de Sklearn, ale căror rezultate le vom compara mai târziu.

4. Am analizat corelația atributelor, iar pe baza acesteia am observat că rasa (Breed Name) se corelează semnificativ cu greutatea (Weight), înălțimea (Height), longevitatea (Longevity), nivelul de energie (Energy Level) și atenția

necesară (Attention Needs). Așadar, vom exclude atributele: lungimea blănii (Coat Length), sexul cățelului și numele stăpânului (Owner Name) în continuare.

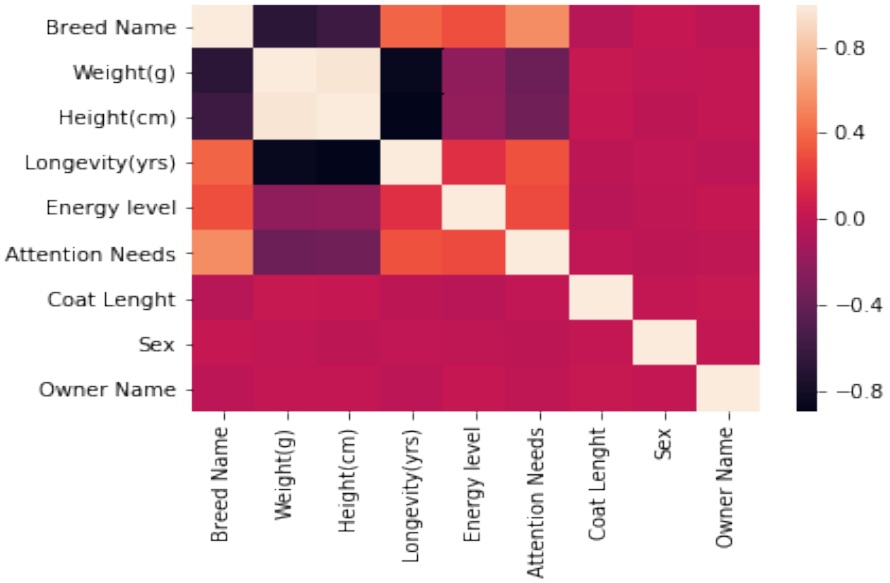


Figura 5.1: Matricea de corelație pentru toate atributele existente în baza de date folosită.

De asemenea, tot pe baza matricei de corelație observăm că cele mai puternice două corelații pe care atributul rasă le are sunt cu greutatea (Weight) și înălțimea (Height). Astfel, în figura de mai jos putem observa că cele patru rase pe care le avem de prezis se disting aproape perfect doar cu ajutorul acestor două atribute.

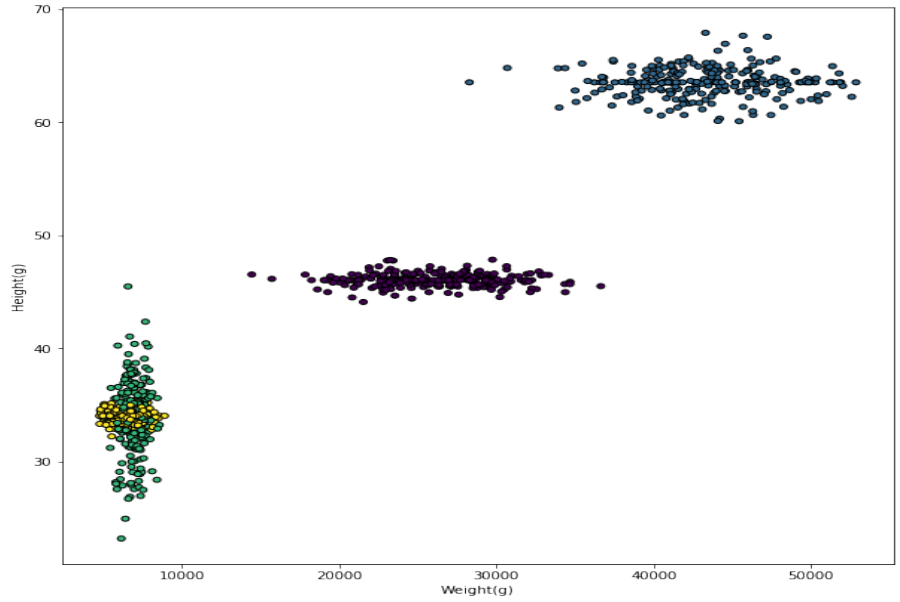


Figura 5.2: Reprezentarea celor patru rase (fiecare cu o culoare distinctă) având pe abscisă greutatea, iar pe ordonată înălțimea.

În continuare, am testat algoritmii Logistic Regression, Random Forest și KNN (toți regăsiți în Sklearn) pentru cele trei seturi de date (setul de date inițial, cel normalizat cu StandardScaler și cel normalizat cu MinMaxScaler). De asemenea, am împărțit seturile în două subseturi: mulțimea de antrenare și mulțimea de testare (folosind `train_test_split()` tot din Sklearn) și am folosit tehnica 10-fold cross-validation pentru a evita overfitting-ul pe mulțimea de antrenare.

5.3 Abordare folosind ProbLog

În ProbLog am păstrat în mare aceeași problemă, și anume cea de prezicere a rasei unui cățel pe baza anumitor atribute. Totuși, de data aceasta nu ne-am mai folosit de toate atributele, ci am ales să folosesc doar înălțimea (Height(cm)) și greutatea (Weight(g)) întrucât a reieșit din analiza anterioara cât de bine corelate sunt acestea două cu rasa cățelului.

Și în cazul acesta am facut unele **preprocesări** înaintea începerii procesului de învățare propriu-zis:

1. Am completat valorile lipsă pentru coloana înălțime conform valorii medii pentru fiecare specie în parte.

2. Am transformat valorile numerice ale celor două atribute (înălțime și greutate) în atribute categoriale, întrucât ProbLog nu poate modela mulțimi continue. Astfel, pentru **atributul înălțime** am transformat în modul următor: toate valorile mai mari de 60 cm în categoria înalt, cele între 40 și 60 sunt în categoria mediu, iar cele sub 40 sunt în categoria scund, iar pentru **atributul greutate** am transformat în modul următor: toate valorile mai mari de 30000 g în categoria greu, cele între 30000 și 20000 sunt în categoria mediu, iar cele sub 20000 sunt în categoria ușor.

După aceste preprocesări am împărțit setul de date asemeni abordării precedente într-un subset de antrenare și unul de testare folosind comanda din Sklearn `train_test_split()`, iar pentru fiecare rasă în parte am format setul de interpretări corespunzător. În continuare, fiecare rasă avea un program ProbLog asignat, în care existau fapte probabiliste pentru greutate și înălțime, fiecare având un parametru, și anume una dintre categoriile posibile. Programul ProbLog de la care s-a pornit pentru fiecare rasă a fost de forma:

```
t(_)::weight(low).
t(_)::weight(medium).
t(_)::weight(high).

t(_)::height(low).
t(_)::height(medium).
t(_)::height(high).
```

Setul de interpretări pentru fiecare rasă a fost alcătuit doar din exemple ale rasei respective, în care fiecare exemplu s-a alcătuit din șase perechi (fapt probabilist, valoare de adevăr). Având programul inițial și setul de interpretari a reieșit câte un model cu probabilități asignate în locul celui inițial pentru fiecare rasă în parte.

În continuare, pentru mulțimea de testare am calculat probabilitățile reieșite din fiecare model pentru fiecare exemplu în parte și am ales de fiecare dată drept rasă prezisă cea pentru care probabilitatea reieșită a fost cea mai mare. Ulterior am comparat cu atributul rasă adevărat.

5.4 Rezultate comparative

Problema abordată atât prin învățarea automată clasică, cât și prin învățarea din interpretări a fost prezicerea etichetei rasă, dar am ales un subset diferit de atribute din setul inițial de date pentru fiecare în parte.

Pentru **prima abordare** s-au folosit o serie de algoritmi clasici de clasificare: Logistic Regression, Random Forests și KNN pentru trei seturi de date: setul de date inițial, setul de date pentru care am utilizat StandardScaler asupra atributelor numerice și setul de date pentru care am utilizat MinMaxScaler asupra acelorași atribute. Așa cum am mai menționat, am împărțit setul mare în set de antrenare și set de testare. Ulterior, am folosit 10-Fold Validation atât pentru evitarea overfitting-ului, dar și pentru a găsi cei mai buni parametri pentru fiecare model în parte, împărțind setul de antrenare în 10 subseturi. Fiecare dintre acestea a fost la un moment dat setul de evaluare pentru modelul, iar celelalte noua s-au folosit pentru antrenare. Cu acești "cei mai buni parametri", am evaluat modelele pe setul de testare și am tras concluzii pe baza scorurile de acuratețe obținute. Rezumând rezultatele de acuratețe avem pentru setul de testare următoarele:

1. Pentru setul de date fără normalizări:
    - Logistic Regression: 0.732
    - Random Forest: 1.0
    - KNN: 0.844
2. Pentru setul cu StandardScaler
    - Logistic Regression: 0.9
    - Random Forest: 0.996
    - KNN: 0.968
3. Pentru setul cu MinMaxScaler
    - Logistic Regression: 0.884
    - Randomte Forest: 0.988

- KNN: 0.96

Astfel, în urma rezultatelor de mai sus, consider că cel mai bun rezultat a fost găsit folosind Random Forests cu StandardScaler deoarece pentru setul de testare a obținut o acuratețe aproape maximă (clasificând greșit doar două exemple), iar acuratețea medie a celor zece folduri este de asemenea cea mai mare dintre toate variantele încercate fapt ce indică stabilitatea acestei metode.

Pentru cea de-a **doua abordare** am calculat pentru fiecare exemplu din mulțimea de test patru valori de probabilități, reprezentând cât este de probabil ca exemplul respectiv pe baza înălțimii și a greutății să facă parte dintr-o anume rasă. Din cele 250 exemple ale acestei mulțimi de test, 207 au fost clasificate corect conform regulii pe care am ales-o, iar cea mai frecventă greșeală a fost de a clasifica drept Jack Russel Terrier un cățel ce în realitate era Pug. Cu toate acestea, acuratețea obținută este de aproximativ 82% variind în funcție de împărțirea setului mare de date în set de antrenare și set de testare.

Concluzionând, atât prima cât și cea de-a doua abordare au atacat același tip de problemă, una de clasificare, în care s-a dorit aflarea rasei unui cățel pe baza unor atribute. Totuși, au existat unele diferențe precum faptul că varianta ce a utilizat ProbLog nu a putut modela o mulțime continuă de valori, pierzând astfel niște informație în legătură cu lumea analizată, iar predicțiile sale s-au bazat doar pe două atribute: greutate și înălțime. Cu toate acestea, ambele variante s-au descurcat remarcabil și au furnizat rezultatele dorite.

Codul pentru întreaga comparație se găsește aici.

Capitolul 6

Concluzii

Trecând prin toate etapele acestei lucrări, de la introducerea noțiunilor teoretice și până la ilustrarea conceptelor ce țin de programarea logică probabilistă prin intermediul ProbLog, consider că am reușit să analizez și să ofer o viziune de ansamblu asupra temei alese. Am acoperit părțile ce țin de utilitatea și domeniile de aplicabilitate ale programării logice probabiliste în general, dar și în particular ale ProbLog, framework ce a apărut în primul rând ca o necesitate de a găsi soluții la probleme din viața reală precum cele din domeniul biologiei. De asemenea, am făcut o trecere în revistă a multiplelor moduri de utilizare ale acestui framework cărora le-am atașat exemple relevante, am prezentat algoritmul de inferență ce stă la baza acestuia și am ales să mă axez în final pe ceea ce se numește învățare din interpretări. Aceasta din urmă a considerat din punctul meu de vedere cea mai interesantă parte dintre tot ceea ce ProbLog are de oferit, atât prin prisma modalității în care se realizează, dar și prin paralela ce a reieșit din comparația cu învățarea automată.

Mai mult de atât, un alt aspect ce m-a surprins plăcut pe tot parcursul elaborării acestei lucrări a fost cât de armonios această tema reușește să îmbine elemente din multiple domenii cum ar fi teoria grafurilor, programarea logică clasică, învățarea automată, dar și teoria probabilităților.

6.1 Planuri viitoare

Așa cum am mai menționat înainte, ProbLog în special este un proiect în continuare activ și în curs de dezvoltare, care își propune să rămână la curent și să încorporeze noutăți pe parcurs ce acestea au loc. Mi-aș dori să țin pasul cu tot ceea va mai urma pe aceasta temă, dar de asemenea aș dori și să experimentez mai mult partea de învățare din interpretări. În această lucrare am debutat printr-o paralelă între învățarea automată și învățarea din interpretări, dar cea din urmă s-a bazat pe mai puține atribute decât cea în cazul clasic. Astfel, o continuare a ceea ce deja există aici este să aduc modelul realizat pe baza învățării din interpretări la un nivel la fel de complex precum cel realizat pe baza învățării automate.

Bibliografie

[1] Cristian Niculescu. *Probabilități și statistică*. (Română) Editura Universității din București, 2014.

[2] Denisa Diaconescu, Ioana Leuștean. *Curs și Seminar- Programare Logică*. `https://cs.unibuc.ro/ ddiaconescu/2018/pl/` (Preluat în iunie 2019)

[3] Frank Pfenning. *Lecture Notes on Binary Decision Diagrams* `https://www.cs.cmu.edu/ fp/courses/15122-f10/lectures/19-bdds.pdf` (Preluat în martie 2019)

[4] Google Machine Learning Crash Course `https://developers.google.com/machine-learning/crash-course/` (Preluat în iunie 2019)

[5] Introduction. - ProbLog: Probabilistic Programming, `https://dtai.cs.kuleuven.be/ProbLog/` (Preluat în iunie 2019)

[6] Luc De Raedt, Angelika Kimmig and Hannu Toivonen. *ProbLog: A probabilistic Prolog and its Application in Link Discovery*. (Engleză) Machine Learning, 100:1, pp. 5 - 47, Springer New York LLC, 2015.

[7] Michael Mitzenmacher, Eli Upfal. *Probability and Computing - Randomized Algorithms and Probabilistic Analysis*. (Engleză) Cambridge University Press, 2005.

[8] Marinescu-Ghemeci Ruxandra. Curs "Algoritmica grafurilor", primavară 2017, Facultatea de Matematică și Informatică.

[9] ProbLog 2.1 documentation, `https://problog.readthedocs.io/en/latest/` (Preluat în iunie 2019)

[10] Richard J. Trudeau. *Introduction to Graph Theory.* (Engleză)
Dover Publications; 2nd edition (February 9, 1994)

[11] Sparktech Software. Curs: "Introduction to Machine Learning using Python", toamnă 2018, Facultatea de Matematică și Informatică.

[12] scikit-learn Machine Learning in Python
`https://scikit-learn.org/stable/` (Preluat în iunie 2019)

[13] Tom M. Mitchell. *Machine Learning.* (Engleză)
McGraw-Hill Science/Engineering/Math; (March 1, 1997) Inference and Learning in Probabilistic Logic

[14] Daan Fierens, Guy Van Den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, Luc De Raedt. *Inference and Learning in Probabilistic Logic Programs using Weighted Boolean Formulas*
Department of Computer Science KU Leuven Celestijnenlaan 200A, 3001 Heverlee, Belgium