HILL CLIMBING

Cavenaghi Emanuele, Zanga Alessio

SCELTE IMPLEMENTATIVE

Python

Strutture implementate

Collegamenti alla libreria BNlearn

MAD

SCELTE IMPLEMENTATIVE

Python

È stato scelto di utilizzare il linguaggio Python per svariati motivi, primo tra tutti il solido supporto e il sempre più vasto impiego.

NetworkX[1]

Altra motivazione è la possibilità di avvalersi delle numerose librerie disponibili tra le quali «**NetworkX**» che rende disponibili strutture per la gestione di grafi, grafi orientati, alberi, ecc. Insieme alle strutture è ovviamente disponibile una vasta collezione di algoritmi su grafi con cui poter confrontare i risultati ottenuti dalle nostre implementazioni.

Numba[2]

Infine, è presente la libreria «**Numba**» che consiste in un Just-In-Time compiler per Python in modo da ottenere codice altamente efficiente. Si mantiene così il vantaggio di scrivere codice di alto livello combinandolo con il vantaggio di velocità dato dalla compilazione di codice a basso livello.

Inoltre, dove possibile, rende più facile il compito di parallelizzare le funzioni sia per CPU che per GPU supportando il linguaggio Nvidia CUDA.

SCELTE IMPLEMENTATIVE

Strutture implementate

Sono state implementate diverse strutture generiche nella libreria creata per poter immagazzinare le informazioni richieste durante la costruzione dell'algoritmo Junction Tree.

Graph

La classe **Graph** rappresenta un generico grafo, implementato tramite matrice di adiacenza binaria, contiene tutti i metodi per poter operare e manipolare il grafo. Estendendo questa classe è stata creata la classe **DirectedGraph** che rappresenta in modo specifico grafi orientati, differenziando, per esempio, tra nodi *parent* e nodi *child*, non presenti nella classe **Graph**.

Tree

La classe **Tree** rappresenta un generico albero costituito da nodi di cui si conosce però solo il nodo *root*. Parte fondamentale della classe è la classe **Node** che contiene al suo interno i riferimenti ai propri nodi *child* e al proprio nodo *parent* oltre agli attributi del nodo stesso.

La classe **JunctionTree** è un'estensione della classe **Tree** e ne condivide tutti gli attributi e metodi oltre ad avere una funzione di plot personalizzata e una funzione che consente, dato il DAG di una Bayesian Network, di costruire il corrispondente Junction Tree.

SCELTE IMPLEMENTATIVE

Collegamenti alla libreria BNlearn

La libreria BNlearn[3] resta comunque un punto di partenza per eseguire comparazioni, test e verifiche vista la grande vastità delle risorse disponibili e gli anni di test e miglioramenti apportati.

- Creazione del DAG partendo dalla struttura in stringa (e.g. [A][C][B|A:C])
- Import dei dataset disponibili
- Import dei file delle Bayesian Networks con le *Probability Conditional Table* nei vari formati

HILL CLIMBING

Score Function (BDs)

Hill Climbing Algorithm

MAD

HILL CLIMBING

Score Function – BDs (I)

La funzione di score implementata per valutare la bontà di un DAG rispetto ai dati raccolti è stata la ***Bayesian Dirichlet Sparse (BDs)*** la cui formula di base è la seguente:

$$BD(\mathcal{G}, \mathcal{D}; \alpha) = \prod_{i=1}^{N} \prod_{j=1}^{q_i} \left[\frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + n_{ij})} \cdot \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ijk} + n_{ijk})}{\Gamma(\alpha_{ijk})} \right]$$

Dove gli indici i, j e k rappresentano:

- $\boldsymbol{i}$ → variabile i-esima del DAG
- $\boldsymbol{j}$ → j-esima configurazione dei parents di una variabile
- $\boldsymbol{k}$ → k-esimo valore che la variabile può assumere (livello)

Inoltre nella funzione di score BDs, a_{ijk} è calcolato come:

$$a_{ijk} = \begin{cases} \frac{a}{r_i \tilde{q}_i} & se\ n_{ij} > 0 \\ 0 & altrimenti \end{cases} \qquad \text{dove} \quad \tilde{q}_i = \{numero\ di\ \Pi_{X_i}\ tali\ che\ n_{ij} > 0\}$$

HILL CLIMBING

Score Function – BDs (II)

È possibile decomporre lo score BDs nella sommatoria degli score dei singoli nodi:

$$BD(\mathcal{G}, \mathcal{D}; \alpha) = \sum_{i=1}^{n} BD(\mathcal{G}, \mathcal{D}; \alpha, X_i)$$

Possiamo quindi scrivere lo score di ogni singolo nodo, sfruttando proprietà dei logaritmi e il logaritmo della gamma function, come:

$$BD(\mathcal{G}, \mathcal{D}; \alpha, X_i) = \sum_{j=1}^{q_i} \left(\log \frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + n_{ij})} + \sum_{k=1}^{r_i} log \frac{\Gamma(\alpha_{ijk} + n_{ijk})}{\Gamma(\alpha_{ijk})} \right) =$$

$$= \sum_{j=1}^{q_i} \left(lgm(a_{ij}) - lgm(\alpha_{ij} + n_{ij}) + \sum_{k=1}^{r_i} \mathrm{lgm}(\alpha_{ijk} + n_{ijk}) - lgm(\alpha_{ijk}) \right)$$

Questo ci permette sia di eseguire il calcolo dello score in parallelo per ogni nodo sia di aggiornare lo score di una rete, quando il DAG subisce modifiche, aggiornando solo i nodi interessati dalle modifiche.

HILL CLIMBING

Score Function – BDs (III)

In particolare notiamo che lo score di un nodo dipende da due «fattori» che vengono ricercati nei dati:

- I valori che il nodo può assumere (livelli)
- Configurazioni dei livelli dei parent

I livelli del nodo non subiscono modifiche mentre le configurazioni dei livelli dei parent sono ovviamente determinati dai parent del nodo di cui si vuole calcolare lo score. È quindi triviale notare che aggiungere, rimuovere o invertire archi possa cambiare questo secondo fattore ed è questo che viene sfruttato per l'aggiornamento dello score del DAG quando vengono apportate delle modifiche.

AGGIUNTA DI UN ARCO

$A \quad B \Rightarrow A \rightarrow B$

Bisogna aggiornare solamente lo score del nodo B poiché viene aggiunto un nodo ai suoi parent

RIMOZIONE DI UN ARCO

$A \rightarrow B \Rightarrow A \quad B$

Bisogna aggiornare solamente lo score del nodo B poiché viene rimosso un nodo dai suoi parent

INVERSIONE DI UN ARCO

$A \rightarrow B \Rightarrow B \rightarrow A$

Bisogna aggiornare lo score del nodo B poiché viene rimosso un nodo dai suoi parent e lo score del nodo A poiché viene aggiunto un nodo ai suoi parent

HILL CLIMBING

Score Function – BDs (IV)

```
configs = [((), size)]
r_i = len(levels)

q_i = 1
if len(parents) > 0:
    configs = dataset.groupby(parents).sum()
    configs = [
        (config, value['count'])
        for config, value in configs.iterrows()
    ]
    q_i = len(configs)

score = 0
for config, n_ij in configs:
    a_ij = iss / q_i
    a_ijk = a_ij / r_i

    score += (lgamma(a_ij) - lgamma(a_ij + n_ij))
    for level in levels:
        n_ijk = _n_ijk(dataset, level, config)
        score += (lgamma(a_ijk + n_ijk) - lgamma(a_ijk))
```

Quella presentata è la parte della funzione che calcola lo score BDs, di un singolo nodo.

Le configurazioni dei parents e le loro occorrenze vengono costruite tramite il pandas dataframe generalizzato dalla classe Dataset.

Le configurazioni dei parents utilizzate per il calcolo sono solamente quelle che hanno almeno una occorrenza nel dataset e vengono accorpate al numero di occorrenze che hanno.

Infine il calcolo dello score è l'applicazione della formula vista precedentemente.

HILL CLIMBING

Hill Climbing Algorithm (I)

Lo pseudocodice dell'algoritmo ***Hill Climbing*** è il seguente:

1. Scegli una rete $\mathcal{G}$ iniziale, solitamente viene scelta vuota ma non è obbligatorio.
2. Calcola lo score di $\mathcal{G}$ (denotato come $Score_{\mathcal{G}} = Score(\mathcal{G})$).
3. Metti $maxscore = Score_{\mathcal{G}}$.
4. Ripeti i seguenti step finché $maxscore$ aumenta:
   I. Per ogni possibile aggiunta, eliminazione o inversione di un arco che non porta a un ciclo
      a) Calcola lo score della rete modificata $\mathcal{G}^*, Score_{\mathcal{G}^*} = Score(\mathcal{G}^*)$.
      b) Se $Score_{\mathcal{G}^*} > Score_{\mathcal{G}}$ allora imposta $\mathcal{G} = \mathcal{G}^*$ e $Score_{\mathcal{G}} = Score_{\mathcal{G}^*}$.
   II. Aggiorna $maxscore$ con il nuovo valore di $Score_{\mathcal{G}}$.
5. Ritorna il DAG ottenuto.

HILL CLIMBING

Hill Climbing Algorithm (II)

Algorithm A.5 Greedy local search algorithm with search operators

```
Procedure Greedy-Local-Search (
  σ_0,     // initial candidate solution
  score,   // Score function
  𝒪,       // Set of search operators
)
1   σ_best ← σ_0
2   do
3     σ ← σ_best
4     Progress ← false
5     for each operator o ∈ 𝒪
6       σ_o ← o(σ)    // Result of applying o on σ
7       if σ_o is legal solution then
8         if score(σ_o) > score(σ_best) then
9           σ_best ← σ_o
10          Progress ← true
11    while Progress
12
13    return σ_best
```

L'implementazione in Python dell'algoritmo ***Hill Climbing*** segue lo pseudo codice presentato dal libro Probabilistic Graphical Models[4] :

Ad ogni modifica della rete non viene calcolato l'intero score della rete ma viene aggiornato lo score solamente dei nodi che subiscono effettivamente una modifica come spiegato precedentemente.

BIBLIOGRAFIA

- [1] NetworkX: https://networkx.github.io/
- [2] Numba: http://numba.pydata.org/
- [3] BNlearn: https://www.bnlearn.com/
- [4] Probabilistic Graphical Models, Principles and Techniques (D. Koller and N. Friedman)

THANK YOU

Cavenaghi Emanuele

Zanga Alessio

MAD

UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA