La mia prossima meta

Documentazione Progetto

- Salvatore Terlizzi, 622803, s.terlizzi1@studenti.uniba.it

Repository: https://github.com/ster0708/Icon_ST-22-23

AA 2022/2023

Introduzione

Il progetto “La mia prossima meta” nasce con l’intenzione di ottenere un piccolo esempio di un’applicazione in grado di suggerire la meta più adatta a un utente in base ai suoi gusti e a quello che cerca dal suo prossimo viaggio. Il dominio scelto riguarda principalmente i viaggi.

In fase di analisi sono state scelte sei città italiane: Milano, Torino, Roma, Napoli, Bari e Palermo. Ogni città ha i suoi luoghi più importanti, prelevati grazie ai dati disponibili su *datiopen.it* e ad altri dataset contenenti luoghi inseriti manualmente.

In partenza, il sistema ragiona su una serie di domande mirate a individuare i gusti dell’utente per poi restituire in output la città a lui più affine. L’utente, inoltre, può interrogare la KB per scoprire ulteriori informazioni sul luogo consigliato.

Una volta individuata la città, l’applicativo simula anche un piccolo sistema di prenotazione camere per soggiornare nella settimana successiva.

Sommario

Un Knowledge-based system è un’applicazione che ragiona su una base di conoscenza per risolvere problemi complessi. Un KBS, quindi, è principalmente composto da una base di conoscenza che contiene fatti e regole riguardanti il mondo che si vuole modellare e da un motore di inferenza che opera sulle informazioni contenute nella base di conoscenza.

Il progetto è stato sviluppato a partire da un nucleo centrale caratterizzato da una rete bayesiana composta da venti nodi. Tre di essi, nel livello superiore, rappresentano i nodi genitori. Il livello intermedio è rappresentato dai nodi che dipendono direttamente da uno o più genitori. Al livello più basso troviamo i nodi che rappresentano le singole città (l’illustrazione della rete è presente più in basso).

L’intero progetto è stato sviluppato in Python usando l’IDE PyCharm e le seguenti librerie:

- *Bnlearn*: *https://pypi.org/project/bnlearn/*
- *Pgmpy*: *https://pgmpy.org/*
- *Swiplserver*: *https://pypi.org/project/swiplserver/*
- *Constraint*: *https://pypi.org/project/python-constraint/*
- *Pandas*: *https://pandas.pydata.org/*

Elenco argomenti di interesse

- Base di conoscenza Prolog interrogabile da applicativo;
- Apprendimento Bayesiano (Teorema di Bayes, costruzione di una rete bayesiana e inferenza) per determinare la città più affine agli interessi dell'utente;
- Constraint Satisfaction Problem: prenotazione di una stanza nella città più affine.

Rete Bayesiana

Sommario

Il modello adottato è la rete bayesiana. Sostanzialmente essa è composta da un grafo aciclico direzionato (DAG) dove ogni feature rappresenta un nodo della rete e ogni arco direzionato rappresenta una dipendenza tra nodi.

Ogni nodo ha una probabilità condizionata determinata da *P(node|parents(node)).* Con *parents(node)* indichiamo la funzione che elenca tutti i nodi genitore del nodo oggetto. Per calcolare la probabilità di ogni nodo si fa riferimento alla formula generale:

$$P(X_1,X_2 \dots X_n) = \prod_{i=1}^{n} P(X_1|X_1 \dots X_{n-1})$$

Dove gli X_i sono le feature della rete bayesiana.

A questo punto è stata definita la rete bayesiana costruendo il DAG e calcolando le CPD per ogni nodo, in base alla frequenza di un evento per ogni città esaminata.

Quando all'utente viene chiesto cosa vorrebbe dal suo prossimo viaggio, l'algoritmo raccoglie le evidenze, costruisce la rete e inferisce. Il risultato sarà una lista di n città alle quali è affine in base ai suoi gusti.

Strumenti utilizzati

La rete bayesiana è stata prima progettata in BayANet e poi implementata in Python attraverso la libreria Bnlearn. Nel modulo bayesian_network.py è possibile trovare tutta l'implementazione relativa a questa parte del progetto.

Inizialmente si predispone il sistema per accogliere le osservazioni ottenute tramite l'interazione con l'utente, poi attraverso il metodo ***init_edges_and_CPD()*** si specificano gli archi esistenti fra i nodi, definendo così la struttura della rete, e si specificano le tabelle di probabilità condizionata per ogni nodo a partire dai nodi genitori.

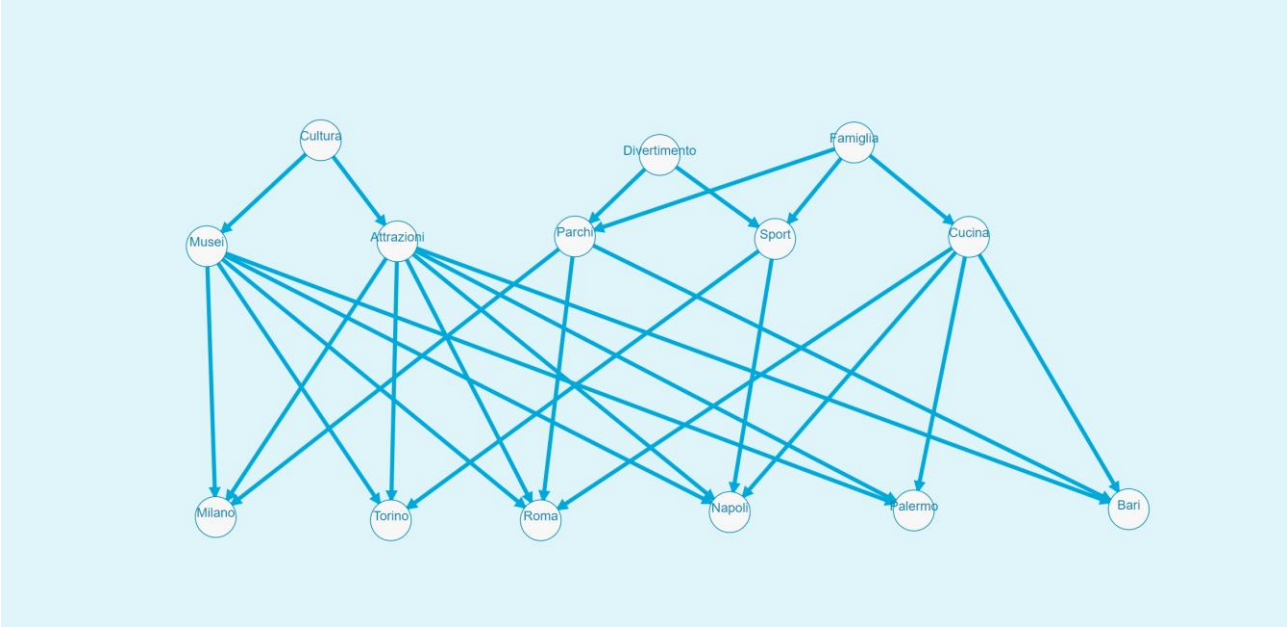
La funzione ***build_network()*** richiama ***init_edges_and_CPD()*** per farsi restituire archi e CPD e attraverso il metodo ***make_DAG()*** costruisce la rete e ritorna un grafo aciclico orientato (DAG). La funzione ***predict_bayesian()*** prende in input il DAG appena costruito insieme alle risposte dell'utente e poi calcola attraverso il metodo ***inference()*** di Bnlearn la probabilità per ogni città inclusa nel progetto.

Nel modulo main.py, infine, si acquisiscono le risposte da parte dell'utente e si richiamano le funzioni sopracitate per poi mostrare a schermo a quale delle città si è più affini.

Decisioni di progetto

Per implementare la rete in Python è stata usata Bnlearn (documentazione: https://erdogant.github.io/bnlearn), una libreria che mette a disposizione diversi strumenti per una molteplicità di esigenze e di situazioni.

In questo caso, come brevemente anticipato, la rete è stata resa nota a priori senza la necessità di utilizzare algoritmi specifici per l'apprendimento della struttura. Di seguito i dettagli della rete:



La rete è strutturata, quindi, su un primo livello costituito dai nodi Cultura, Divertimento e Famiglia; un secondo, quello intermedio, formato da musei, attrazioni, parchi, sport e cucina e un terzo livello rappresentato dalle variabili che indicano le città scelte per il progetto. Ogni livello influenza quello precedente.

Le CPD sono state costruite in base alla frequenza di ogni categoria per ogni città, rispettando i collegamenti presenti.

Funzionamento dell'applicazione

All'avvio dell'applicativo vengono sottoposte quattro domande all'utente, che una volta processate secondo criteri specifici, costituiscono le evidenze. Successivamente viene costruita la rete utilizzando i metodi indicati nel paragrafo precedente e vengono mostrate a video le tre città più affini all'utente, in ordine decrescente.

```
Vediamo cosa cerchi nel tuo prossimo viaggio!

Vuoi visitare musei e luoghi di rilevanza artistica/culturale? [y/n]
y
Ti interessano eventi sportivi e sei un appassionato di calcio? [y/n]
n
Vuoi viaggiare con la famiglia e cerchi divertimento per grandi e piccoli? [y/n]
n
Hai interesse nel degustare cibo tipico della cucina locale? [y/n]
y


Calcoliamo la tua prossima meta...

Le città più compatibili con te sono:
[1] - Palermo, con una probabilità di 53.39
[2] - Napoli, con una probabilità di 52.26
[3] - Roma, con una probabilità di 49.07

Quale città scegli?
2

Hai scelto Napoli
```

CSP – Constrain Satisfaction Problem

Sommario

In informatica molti problemi possono essere rappresentati e risolti come Constraint Satisfaction Problem, ovvero problemi di soddisfacimento di vincoli. Un CSP finito è composto da variabili, ognuna con un suo dominio, e da vincoli di due tipi: hard constraint (vincoli rigidi) e soft constraint (vincoli flessibili).

Un "ragionamento" su un CSP sarà quello di trovare la soluzione al problema rispettando i vincoli rigidi e soddisfando i vincoli flessibili.

Formalmente un vincolo è il prodotto cartesiano fra l'ambito delle variabili (nello specifico le variabili coinvolte nel vincolo) e la relazione sull'ambito. Un mondo possibile W (o un modello W) soddisfa un insieme di vincoli se ogni vincolo è soddisfatto dai valori assegnati alle variabili nel proprio ambito.

In questo caso, dopo che il programma ha individuato la città più affine all'utente, viene avviato un programma per la prenotazione di una stanza che simula quattro strutture ricettive e diverse disponibilità in base al numero di persone.

Come strumento per l'implementazione del Constraint Satisfaction Problem è stata utilizzata la libreria Constraint di Python.

Decisioni di progetto

Tutta la parte relativa all'implementazione del CSP si trova nel modulo **CSP.py** del progetto. La classe room_csp è il nucleo su cui ruota il modulo. In questa classe sono contenute le variabili "base" comuni a tutti i CSP e il metodo **verifica_disponibilità()** che si occupa di verificare se, in base alle persone scelte, esista effettivamente un'assegnazione che soddisfi i CSP.

```
class room_csp(Problem):
    Salvatore *
    def __init__(self, room: str, solver=None):
        super().__init__(solver=solver)
        self.room = room
        self.letto = self.addVariable("letto", ["singolo", "matrimoniale", "king size"])
        self.giorni = self.addVariable("giorni", [1, 2, 3, 4, 5, 6, 7])
        self.disponibilità = None

    Salvatore
    def get_disponibilità(self):
        self.disponibilità = sorted(self.getSolutions(), key=lambda g: ['giorni'])

        if len(self.disponibilità) > 0:
            print("Ecco le stanze disponibili nell'Hotel scelto: ")
            i = 0
            while i < len(self.disponibilità):
                data_iniziale = datetime.strftime(datetime.today(), "%d/%m/%Y")
                data_finale = datetime.today() + timedelta(days=self.disponibilità[i]['giorni'])
                data_finale = datetime.strftime(data_finale, "%d/%m/%Y")
                print("Stanza[%d]: Da %d persone con Letto %s - Disponibile dal %s al %s " % (i, self.disponibilità[i]['persone'], self.disponibilità[i]['letto'], data_iniziale, data_finale))
                i = i+1
        else:
            print("Non ci sono stanze disponibili")

        return self.disponibilità
```

La funzione **genera_csp()** istanzia quattro oggetti room_csp, uno per ogni struttura (H1, H2, H3 e H4) e definisce la variabile “persone” in base al numero di persone passato in input dall’utente. È inoltre il punto in cui si aggiungono a ogni oggetto di tipo room_csp dei vincoli specifici.

```
def genera_csp(num_persone):

    hotel_1 = room_csp(H1)
    hotel_1.addVariable("persone", [num_persone])
    hotel_1.addConstraint(lambda letto, persone, giorni: persone == 2 if letto == "matrimoniale" and giorni == 7 else persone == 1 if letto == "singolo" and giorni == 3 else None, ["letto", "persone", "giorni"])

    hotel_2 = room_csp(H2)
    hotel_2.addVariable("persone", [num_persone])
    hotel_2.addConstraint(lambda letto, persone, giorni: persone == 3 if letto == "matrimoniale + singolo" and giorni == 3 else persone == 4 if letto == "quattro singoli" and giorni == 5 else None,  ["letto", "persone", "giorni"])

    hotel_3 = room_csp(H3)
    hotel_3.addVariable("persone", [num_persone])
    hotel_3.addConstraint(lambda letto, persone,giorni: persone == 5 if letto == "matrimoniale" and giorni == 4 else persone == 1 if letto == "singolo" and giorni == 3 else None, ["letto", "persone", "giorni"])

    hotel_4 = room_csp(H4)
    hotel_4.addVariable("persone", [num_persone])
    hotel_4.addConstraint(lambda letto, persone, giorni: persone == 2 if letto == "king size" and giorni == 3 else persone == 1 if letto == "singolo" and giorni == 3 else None, ["letto", "persone", "giorni"])


    return [hotel_1, hotel_2, hotel_3, hotel_4]
```

Nel metodo verifica_disponibilità() della classe room_csp() viene invocato il metodo getSolutions() della classe Problem (propria della libreria Constraint). GetSolutions() risolve i CSP utilizzando il metodo di risoluzione specificato: Backtracking, Backtracking ricorsivo e Conflitti minimi.

Come mostrato nella prima schermata, inizialmente il nostro solver è impostato sul valore “None”, che indica all’algoritmo di procedere con la tecnica di risoluzione di default che è il **Backtracking Search**. Nel caso specifico si è deciso di lasciare questa impostazione di default, visto il problema poco complesso.

Un’alternativa agli algoritmi di ricerca e assegnazione è quella di costruire uno spazio di ricerca, utilizzando gli stessi criteri visti per gli spazi di stati. Trattando il CSP con un albero, i nodi corrispondono ad assegnazioni di valori di un determinato sottoinsieme di variabili. Il nodo di partenza è l’assegnazione vuota, mentre l’obiettivo è quello in cui ogni variabile ha un valore assegnato senza che questo violi alcun vincolo. Il backtracking search è sostanzialmente una ricerca in profondità che, a differenza del generate and test, ha un controllo sui vincoli a monte che permette di escludere totalmente determinate regioni di ricerca nell’albero.

Funzionamento dell'applicazione

Dopo aver mostrato all'utente le città a lui più affini e aver registrato la sua scelta, si procede con la scelta di una stanza per un determinato numero di persone. Se ci sono soluzioni del CSP per quel determinato numero di persone in quella determinata struttura, allora il programma permetterà la prenotazione e restituirà esito positivo.

```
Vuoi prenotare una stanza per la prossima settimana? [y/n]
y
Per quante persone vuoi prenotare?
2
Scegli una struttura fra quelle proposte:
[1] Hotel Excelsior
[2] B&B Il Belvedere
[3] Hotel La Rotonda
[4] Hotel Principe Savoia

1
Ecco le stanze disponibili nell'Hotel scelto:
Stanza[0]: Da 2 persone con Letto matrimoniale - Disponibile dal 11/02/2023 al 18/02/2023

Scegli una stanza fra quelle disponibili:
0
Hai prenotato la stanza 0 della struttura Hotel Excelsior
```

In caso contrario, invece, il programma restituirà esito negativo:

```
Vuoi prenotare una stanza per la prossima settimana? [y/n]
y
Per quante persone vuoi prenotare?
5
Scegli una struttura fra quelle proposte:
[1] Hotel Excelsior
[2] B&B Il Belvedere
[3] Hotel La Rotonda
[4] Hotel Principe Savoia

1
Non ci sono stanze disponibili
```

Knowledge Base

Sommario

Una knowledge base è una raccolta di dati che rappresenta i fatti sul mondo che si sta rappresentando, regole per ragionare sugli assiomi e, eventualmente, strumenti logici per dedurre nuovi fatti. Il tutto è funzionale alla raccolta, all'organizzazione e alla distribuzione della conoscenza.

Strumenti utilizzati

Per rappresentare la KB si è utilizzato principalmente SWI-Prolog, mentre lato codice, in Python, è stata utilizzata la libreria Swiplserver. Quest'ultima si appoggia all'installazione locale di SWI-Prolog per interrogare la KB e farsi dare le risposte.

Decisioni di progetto

Tutto quello che riguarda l'implementazione della KB in Prolog è reperibile nella cartella "Prolog" del progetto. All'interno di essa si trova sia il file kb_all.pl che il file prolog.py. Il primo contiene la KB vera e propria con tutti i fatti e le regole; il secondo contiene tutta la parte relativa all'implementazione, con funzioni che si occupano di scrivere automaticamente il file .pl in base alle informazioni contenute nei dataset.

Cosa contiene la KB?

Principalmente **fatti**, ovvero gli assiomi della KB su cui si basano le regole. In particolare:

- *posto(nome, città)* associa a ogni città un posto che può essere un'attrazione o un museo;
- *sport(nome, città)* associa a ogni città un luogo dove si tengono eventi sportivi (es. uno stadio);
- *parcodivertimenti(nome, città)* associa a ogni città un parco divertimenti (se presente);
- *cucina(nome, città)* associa a una città una o più specialità culinarie;

```
posto(pietro_doderlein,palermo).
posto(esposizione_strumenti_medici_ex_ospedale_psichia,palermo).
posto(albergo_delle_povere,palermo).

sport(stadio_giuseppe_meazza,milano).
sport(stadio_olimpico,roma).
sport(stadio_diego_armando_maradona,napoli).
sport(stadio_renzo_barbera,palermo).
sport(stadio_san_nicola,bari).
sport(allianz_stadium,torino).

parcodivertimenti(gardaland,milano).
parcodivertimenti(leolandia,milano).
parcodivertimenti(acquatica_park,milano).
parcodivertimenti(acquaworld_,milano).
parcodivertimenti(acquapark_lodi,milano).
parcodivertimenti(le_cornelle,milano).
parcodivertimenti(ondaland_,torino).
parcodivertimenti(acquajoy,torino).
parcodivertimenti(magicland,roma).
parcodivertimenti(zoomarine,roma).
parcodivertimenti(cinecittaworld,roma).
parcodivertimenti(acquafelix,roma).
```

Per ogni categoria è presente una funzione nel modulo Python che interroga la KB e restituisce tutti i dati relativi a una determinata città.

```python
def prolog_getSport(city):

    lista_sport = list()
    with PrologMQI() as mqi:
        with mqi.create_thread() as prolog_thread:
            prolog_thread.query("set_prolog_flag(encoding,utf8).")
            prolog_thread.query("consult(\"iCoN_PyProject/prolog/kb_all.pl\").")

            result = prolog_thread.query("sport(Nome, "+city+")")

            for sport in result:
                lista_sport.append(sport["Nome"])

    return lista_sport
```

E' possibile estendere la KB con **regole**, che possono rispondere a delle domande specifiche che l'utente pone alla base di conoscenza utilizzando i fatti. In questo caso è stata implementata la regola:

- *hamare(X) :- haspiagge(X).*

Che permette di sapere se una delle città proposte è una città di mare o meno. Per ragioni di tempo non è stato possibile implementarne altre, ma la possibilità sono praticamente infinite.

Funzionamento dell'applicazione

Dopo la scelta della stanza, che richiama il CSP, l'utente può interrogare la KB dall'applicazione stessa. Come detto in precedenza, questo avviene mediante una libreria che si interfaccia direttamente con l'installazione di SWI-Prolog presente sulla macchina. Questa scelta si è resa necessaria a causa di problemi di compatibilità con le versioni recenti di Python di altre librerie più immediate e meno vincolanti.

```
Vuoi saperne di più su Roma? Scegli una delle seguenti opzioni:
[1] - E' una città di mare?
[2] - Quali sono le specialità culinarie?
[3] - Quali sono gli eventi sportivi?
[4] - Quali sono i parchi divertimento nei dintorni?
[0] - Esci
```

Vengono messe a disposizione una serie di domande di default, in base ai fatti e alle regole presenti attualmente nella KB, ma le estensioni possibili sono moltissime.

```
Vuoi saperne di più su Roma? Scegli una delle seguenti opzioni:
[1] - E' una città di mare?
[2] - Quali sono le specialità culinarie?
[3] - Quali sono gli eventi sportivi?
[4] - Quali sono i parchi divertimento nei dintorni?
[0] - Esci

2
Le specialità culinarie di Roma sono:
1 - pasta_allamatriciana
2 - pasta_alla_carbonara
3 - pasta_cacio_e_pepe
4 - coda_alla_vaccinara
5 - saltimbocca_alla_romana
6 - carciofi_alla_giudia

Vuoi saperne di più su Roma? Scegli una delle seguenti opzioni:
[1] - E' una città di mare?
[2] - Quali sono le specialità culinarie?
[3] - Quali sono gli eventi sportivi?
[4] - Quali sono i parchi divertimento nei dintorni?
[0] - Esci

3
Gli stadi famosi vicino Roma sono:
1 - stadio_olimpico
```

Conclusione

La mia prossima meta è un progetto che nasce come semplice contenitore di tre argomenti trattati nel corso di Ingegneria della Conoscenza. Le estensioni e i miglioramenti applicabili sono molteplici, partendo dalla sezione di apprendimento che può essere estesa. Fra i tanti sviluppi possibili c'è evidentemente quello di sfruttare ancora di più le informazioni presenti nei dataset.

Inoltre, non tutti i luoghi possono avere la stessa rilevanza per determinate categorie; ad esempio, se si è interessati al divertimento, sappiamo che Gardaland è uno dei parchi più quotati in Italia e questo dovrà avere per forza di cose un peso durante l'apprendimento.

Un'altra possibile estensione potrebbe riguardare il CSP per la prenotazione delle stanze, estendendolo con un algoritmo di apprendimento non supervisionato (KNN ad esempio) che operi su un dataset e che possa individuare una serie di proposte con determinate caratteristiche simili da consigliare all'utente.

Infine, un'app di questo tipo dovrebbe avere un'interfaccia grafica semplice e intuitiva. Sempre per ragioni di tempo, non è stato possibile svilupparne una. Inoltre, rimanendo nel dominio di questo corso, si potrebbe pensare di reperire informazioni sparse sul web per fornire una breve descrizione di ogni città con le sue peculiarità e i suoi difetti in base a ciò che cerca l'utente dal suo viaggio.