UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO | DIPARTIMENTO DI INFORMATICA

Rome’s treasures unveiled: an insider’s journey

Gruppo di lavoro

- Daniele Guerra, 744399, d.guerra6@studenti.uniba.it
- Nicola Lassandro, 735968, n.lassandro4@studenti.uniba.it

Repository GitHub

https://github.com/NicolaLassandro/progetto_icon_guerra_lassandro

AA 2022-2023

Introduzione

Il software lavora nel dominio degli itinerari turistici, in particolare (al momento) è dedicato alla città di Roma.

Esso mette a disposizione dell'utente una panoramica completa di ciò che la città ha da offrire, garantendo una flessibilità che permette di personalizzare la propria esperienza in base alle proprie preferenze ed esigenze.

L'obiettivo del software è quello di individuare il percorso di visita migliore, principalmente sulla base della distanza percorsa, si tiene anche conto dei limiti di tempo e budget impostati, tutto ciò garantendo una qualità di punti di interesse visitati elevata.

Sommario

Strutturazione del progetto

- Cartella Knowledge: contiene tutti i file prolog e python relativi alla rappresentazione della conoscenza nella Kb utilizzata nel progetto.
  - Facts.pl
  - KbManager.py
  - Landmark.py
  - Preprocessor.py
  - Rules.pl
  - RuntimeFacts.pl
  - Utility.py

- Cartella Learning: contiene tutti i file python relativi alla fase di apprendimento supervisionato.
  - FeedbackGenerator.py
  - Kfold.py
  - Knn.py
  - ModelInitializer.py
  - PreProcessorLearning.py
  - RegressionTree.py

- Cartella Logs: contiene tutti i file testuali di log utilizzati per supervisionare l'andamento degli script di tutte le altre cartelle.

- Cartella Search: contiene tutti i file python utilizzati nel problema di ricerca su grafo.
  - Cartella Libs: contiene gli algoritmi predefiniti che sono stati utilizzati come base per il problema di ricerca realizzato dal gruppo.
  - ItinerarySearchProblem.py
  - MainSearch.py: è il main dell'applicativo.
  - NodeGraph.py

- Cartella Storage: contiene tutti i file serializzati in formato pickle.

Elenco argomenti di interesse

Di seguito sono presentati i macro-argomenti trattati durante lo sviluppo dell'applicativo.

- Rappresentazione della conoscenza
- Problema di ricerca su grafo
- Apprendimento supervisionato (regressione)

In fase di scelta dell'idea progettuale e della sua realizzazione, si è cercato di includere la più vasta gamma di argomenti trattati durante il corso di Ingegneria della conoscenza.

Rappresentazione della conoscenza

Sommario

La prima operazione svolta durante lo sviluppo dell'applicativo turistico è stata la raccolta dei dati relativi ai punti di interesse della città di Roma, i quali sono stati utilizzati per l'inizializzazione delle istanze della classe di riferimento: Landmark.
Gli attributi presenti in tale classe sono:

- *placeId, name, address, type, properties, lat, lon, age*.
  Questi rappresentano le caratteristiche "anagrafiche" del luogo di interesse.
- *rating, ratingCount*.
  Questi illustrano la popolarità e la qualità del punto di interesse sulla base dei giudizi preesistenti degli utenti.
- *centreDistance*.
  E' la distanza da quello che è stato definito arbitrariamente il punto centrale del turismo della città.
- *tourismRate*, *price*.
  Questi fanno riferimento alle carattistiche "turistiche", le quali sono fattori influenzanti della visitabilità del luogo.
- *handicapAccessibility*, *surface*, *height*.
  Questi sono gli attributi strutturali del punto di interesse.

Tutti gli altri campi presenti verranno popolati solo in un secondo momento.

Successivamente queste istanze sono state memorizzate all'interno di un dizionario e tutte le feature sono state immagazzinate all'interno di una base di conoscenza[1] sotto forma di fatti. La maggior parte di essi seguono uno stesso formato: *feature(landmark_name, feature_value)*.

Di seguito ne sono presentati alcuni esempi.

```
tourismRate('Pyramid of Caius Cestius',969971).
tourismRate('Turtle Fountain',1338266).
tourismRate('Basilica di San Bartolomeo all Isola',1179645).
tourismRate('Chiesa di Santa Prisca',1377947).
tourismRate('Tempio Maggiore',565647).
tourismRate('Portico of Octavia',1075336).
tourismRate('Basilica Santi Giovanni e Paolo',1310044).
age('Il Tempio dei Dioscuri',1764).
age('Basilica Julia',1295).
age('Fountain of the Bees',1641).
age('Ludus Magnus',233).
```

La feature *distance* fa eccezione dal formato Prolog delle altre, essa è così strutturata: *feature(landmark_1_name, landmark_2_name, feature_value)*.

```
distance('Mattatoio','Portico of Octavia',1832).
distance('Mattatoio','Basilica Santi Giovanni e Paolo',1917).
distance('Pyramid of Caius Cestius','Turtle Fountain',1946).
distance('Pyramid of Caius Cestius','Basilica di San Bartolomeo all Isola',1553).
distance('Pyramid of Caius Cestius','Chiesa di Santa Prisca',765).
distance('Pyramid of Caius Cestius','Tempio Maggiore',1738).
distance('Pyramid of Caius Cestius','Portico of Octavia',1790).
distance('Pyramid of Caius Cestius','Basilica Santi Giovanni e Paolo',1447).
distance('Turtle Fountain','Basilica di San Bartolomeo all Isola',392).
distance('Turtle Fountain','Chiesa di Santa Prisca',1296).
distance('Turtle Fountain','Tempio Maggiore',207).
distance('Turtle Fountain','Portico of Octavia',167).
distance('Turtle Fountain','Basilica Santi Giovanni e Paolo',1457).
distance('Basilica di San Bartolomeo all Isola','Chiesa di Santa Prisca',923).
distance('Basilica di San Bartolomeo all Isola','Tempio Maggiore',185).
distance('Basilica di San Bartolomeo all Isola','Portico of Octavia',243).
distance('Basilica di San Bartolomeo all Isola','Basilica Santi Giovanni e Paolo',1230).
distance('Chiesa di Santa Prisca','Tempio Maggiore',1094).
distance('Chiesa di Santa Prisca','Portico of Octavia',1130).
distance('Chiesa di Santa Prisca','Basilica Santi Giovanni e Paolo',799).
distance('Tempio Maggiore','Portico of Octavia',71).
distance('Tempio Maggiore','Basilica Santi Giovanni e Paolo',1319).
```

Questa feature è di fondamentale importanza all'interno del progetto in quanto si trova alla base del modulo di ricerca su grafo, nucleo del caso di studio.
Per questo motivo è stato necessario conservare tutti i fatti ad essa relativi nell'apposito file Prolog, cosa che non è avvenuta per tutte le feature generate a posteriori.

La KB, dopo essere stata arricchita con diverse regole, è stata interrogata per generare nuova conoscenza.

Tra le regole realizzate ce ne sono alcune particolarmente importanti:

- *calculateDensity*, che permette di definire la densità turistica del luogo, intesa come media delle distanze da tutti gli altri posti registrati (prelevati precedentemente mediante il predicato Prolog *findall*).

```
calculateDensity(PoiName, Density) :-
    findall(Dist, (distance(PoiName, _, Dist); distance(_, PoiName, Dist)), Dists),
    average(Dists, Density).
```

- *calculateTourismPriority*, serve a calcolare la priorità di visità del punto di interesse sulla base delle altre feature che lo caratterizzano.
  Ognuna di esse influenza in modo diverso il risultato finale, che viene successivamente salvato nel dizionario e utilizzato nel modulo di ricerca su grafo per garantire una soglia minima di qualità dei luoghi previsti dall'itinerario turistico. La priorità influenza anche il valore dell'euristica.

```
calculateTourismPriority(PoiName, TourismPriority) :-
    rating(PoiName, Rating),
    (popular(PoiName) ->
        PopularWeight = 0.6;
        PopularWeight = 0),
    (closeToCityCentre(PoiName) ->
        CloseToCityCentreWeight = 0.3;
        CloseToCityCentreWeight = 0),
    calculateTourismRateOutOfTen(PoiName, TourismRateOutOfTen),
    (ancient(PoiName) ->
        AncientWeight = 0.2;
        AncientWeight = 0),
    (impressive(PoiName) ->
        ImpressiveWeight = 0.3;
        ImpressiveWeight = 0),
    calculateDensity(PoiName, Density),
    NormalizedDensity is (Density - 1138) / (2846 - 1138),
    DensityWeight = 0.6 - (0.6 * NormalizedDensity),
    TourismPriority is Rating / 2 + PopularWeight + CloseToCityCentreWeight + TourismRateOutOfTen * 0.05 + AncientWeight + ImpressiveWeight + DensityWeight.
```

- *calculateTimeToVisit*, si occupa di stabilire il tempo necessario alla visita di un luogo, questo è influenzato sia da aspetti strutturali (come le dimensioni) sia da aspetti turistici (all'aumentare del tasso di turismo il luogo risulta più affollato).

```
calculateTimeToVisit(PoiName, TimeToVisit) :-
    tourismRate(PoiName, TourismRate),
    surface(PoiName, Surface),
    height(PoiName, Height),
    normalizeTourismRate(TourismRate, NormTourismRate),
    normalizeSurface(Surface, NormSurface),
    normalizeHeight(Height, NormHeight),
    TimeToVisitNormalized is (NormTourismRate + NormSurface + NormHeight) / 3,
    TimeToVisitFloat is round(TimeToVisitNormalized * (40 - 5) + 5),
    TimeToVisit is min(max(TimeToVisitFloat, 5), 60).
```

Di seguito è presentata una porzione di codice python con la quale viene interrogata la base di conoscenza per ciascuno dei punti di interesse registrati (fase di generazione delle nuove feature), i risultati ricevuti vengono posti in un contenitore (result) e successivemente estratti e assegnati all'omonimo attributo dell'istanza di Landmark corrispondente nel dizionario.

```python
# Creation new feature (density)
for value in poiMap.values():
    result = list(prolog.query(f"calculateDensity('{value.name}', Density)"))
    value.density = int(result[0]["Density"])
log.info("Density feature created correctly.\n")
```

Anche le interrogazioni che restituiscono risultati di diverso tipo (ad esempio booleani) seguono un ragionamento analogo.

```python
# Creation new feature (impressive)
for value in poiMap.values():
    if bool(list(prolog.query(f"impressive('{value.name}')"))):
        value.impressive = True
    else:
        value.impressive = False
log.info("Impressive feature created correctly.\n")
```

Infine, il dizionario, contenente tutte le istanze di Landmark, è stato aggiornato, con tutti i valori delle nuove feature a disposizione, e serializzato per poter essere utilizzato nei successivi moduli del progetto.

Strumenti utilizzati

La raccolta dei dati utili è stata effettuata mediante l'impiego di una API a pagamento messa a disposizione da Google Places (la piattaforma di Google Maps): nearbysearch[2], essa consente di recuperare una lista di luoghi nelle vicinanze di una determinata posizione geografica.

Per il suo utilizzo è stato necessario generare (gratuitamente) una API-Key della durata di pochi mesi ed è stata impiegata la libreria Requests[3] fornita da Python.

```python
# API call to fetch nearby tourist attractions based on latitude and longitude
def apiCall(latitude: float, longitude: float):
    api_key = "[redacted]"
    url = f"https://maps.googleapis.com/maps/api/place/nearbysearch/json?location={latitude},{longitude}&type=tourist_attraction&radius=1500&key={api_key}"
    return rq.get(url)
```

La chiamata richiede diversi parametri per restituire i risultati corretti, di seguito sono citati quelli impiegati nel corrente caso di studio:

- API-Key

- Le coordinate di latitudine e longitudine del centro di ricerca, a partire dal quale sono stati individuati i risultati.
- Il raggio massimo di estensione della ricerca dei luoghi.
- Il tipo dei luoghi, espresso mediante una parola chiave, sulla base del quale viene effettuato il filtraggio, allo scopo di ottenere risultati specifici.

La risposta risultante della chiamata viene restituita in formato json e contiene al suo interno numerosi attributi, tra questi ne sono stati selezionati ed estratti alcuni utili allo scopo dell'applicativo:

- *place_id*, un identificativo che viene assegnato ad ogni elemento registrato sulla piattaforma Google.
- *user_ratings_total*, un intero che rappresenta la quantità di recensioni degli utenti relative all'elemento.
- *name*, il nome del luogo.
- *geometry.location.lat* e *geometry.location.lon*, le coordinate spaziali del luogo in formato decimale.
- *rating*, la media delle votazioni (da 1 a 5) contenute nelle recensioni.

Le restanti caratteristiche dei punti di interesse, come suggerito dal prof. Fanizzi, per via della difficoltà di reperimento, sono state generate casualmente mediante la libreria random messa a disposizione da python. Ovviamente la generazione casuale ha tenuto conto di intervalli verosimili dei valori delle feature.

Per la rappresentazione della conoscenza e la sua successiva interrogazione si è deciso di utilizzare il linguaggio Prolog[4], un linguaggio di programmazione logico basato su predicati, che rappresentano fatti o relazioni tra oggetti. Un programma Prolog è costituito da un insieme di predicati e regole, queste ultime vengono utilizzate per derivare nuovi fatti o per risolvere interrogazioni sulla base di conoscenza.

Decisioni di progetto

Molteplicità delle chiamate

Dal momento che le API di Google restituivano venti risultati per ogni chiamata, è stato necessario ripetere più volte la chiamata API finché non si fossero ottenuti tutti i risultati.

Inoltre, per poter ottenere un numero discretamente elevato di luoghi si è deciso di ripetere la chiamate API quattro volte, fornendo quattro coppie di coordinate diverse a ciascuna chiamata. Il motivo per il quale ne sono state fornite quattro diverse è dipeso dal fatto che si è cercato di massimizzare i risultati ottenibili ripetendo la chiamata API spostandosi di 1 km a nord, sud, est ed ovest del centro turistico scelto.

Per quanto concerne la scelta delle coppie di coordinate da cui far partire la ricerca, si è deciso di scegliere arbitrariamente un punto che potesse contenere un numero consistente di luoghi tale da poter essere definito come centro turistico di Roma.

Assiomatizzazione del dominio nella base di conoscenza

Infine, l'assiomatizzazione del dominio nella base di conoscenza è avvenuta mediante la scrittura su file dei fatti riguardanti i vari luoghi; si è deciso di utilizzare il metodo *write* di Python per scrivere sul file Prolog, invece del metodo *assertz*, poiché le informazioni contenute nella base di conoscenza avrebbero dovuto essere consultate anche negli altri moduli del progetto.

Il metodo *assertz* sarebbe stato più opportuno qualora, nel ciclo di esecuzione del programma, la base di conoscenza avesse richiesto l'aggiunta di fatti in maniera dinamica durante un'unica esecuzione; nel caso di studio in questione ogni modulo equivale ad un'esecuzione separata.

Problema di ricerca su grafo

Sommario

Il fine del caso di studio è quello di rappresentare un punto di riferimento per i turisti che vogliono visitare i luoghi più importanti della città di Roma, fornendo così il miglior itinerario da seguire per sfruttare al meglio le risorse a disposizione (tempo e budget).

Per poter ottenere il miglior percorso, è stato necessario l'impiego di un algoritmo di ricerca su grafo[5], che potesse tenere in considerazione le necessità dell'utente e che fosse in grado di restituire un percorso contenente i migliori luoghi da visitare.

Strumenti utilizzati

Lo strumento impiegato per la ricerca su grafo è l'algoritmo A*[6], un algoritmo di ricerca euristica. Esso cerca il percorso ottimale tra due nodi, sommando la funzione di costo fino al punto corrente con una funzione euristica, che stima il costo dal punto corrente ad un nodo goal:

$$f(p) = cost(p) + h(p)$$

La libreria utilizzata per l'implementazione di A* è AIPython[7], una libreria contenente un insieme di algoritmi di ricerca preimplementati.

Decisioni di progetto

Struttura dei nodi

Per l'impiego dell'algoritmo di ricerca, è stato necessario definire una classe che rappresentasse i nodi da inserire nel grafo. I nodi non rappresentano, come si potrebbe banalmente pensare, i luoghi di interesse ma al contrario delle "situazioni".

Nello specifico, ciascun nodo conterrà le seguenti informazioni:

- *name*, indica il nome del luogo.
- *coveredDistance*, indica la distanza percorsa fino al punto corrente.
- *remainingBudget*, indica la disponibilità corrente di budget.
- *remainingTime*, indica la disponibilità corrente di tempo.
- *visitedNodes*, indica i luoghi precedentemente visitati.

- *sumVisitedPriority*, indica la somma delle priorità dei luoghi visitati.

Gli ultimi quattro attributi del nodo verranno utilizzati in fase di ricerca per effettuare specifici controlli.

Adattamento del problema di ricerca

Nella fase di definizione del grafo non si conoscono a priori i nodi goal, questo perché la sua generazione avviene dinamicamente a partire dalla posizione corrente dell'utente (nel caso di studio è stata simulata). Una generazione statica del grafo avrebbe gravato eccessivamente sulle prestazioni dell'applicativo.

Durante il problema di ricerca, è necessaria una funzione di individuazione dei vicini:

```python
# Finds the neighboring nodes of a given node.
def neighbors(self, node):
    # Query the prolog knowledge base to find the neighbors of the current node
    neighs = list(self.prolog.query(f"findNeighbors('{node.name}', Neighbors)"))[0][
        "Neighbors"
    ]
    # Initialize an empty list to store the arcs
    arcs = []

    # Create a copy of the visitedNodes list and append the name of the current node
    newVisitedNodes = node.visitedNodes[:]
    newVisitedNodes.append(str(node.name))

    # Iterate over each neighbor
    for neigh in neighs:
        # Check if the neighbor is not already visited
        if str(neigh) not in node.visitedNodes:
            # Query the prolog knowledge base to find the distance, cost, time, and tourism priority of the node
            dist = list(
                self.prolog.query(
                    f"findDistance('{node.name}', '{neigh}', Distance)"
                )
            )
            cost = list(self.prolog.query(f"price('{neigh}', Price)"))
            time = list(
                self.prolog.query(f"calculateTimeToVisit('{neigh}', TimeToVisit)")
            )
            visitedPriority = list(
                self.prolog.query(
                    f"calculateTourismPriority('{neigh}', TourismPriority)"
                )
            )
            # Create a new NodeGraph object representing the neighbor node with updated attributes
            nodeGraph = NodeGraph(
                neigh,
                node.coveredDistance + int(dist[0]["Distance"]),
                node.remainingBudget - int(cost[0]["Price"]),
                node.remainingTime - int(time[0]["TimeToVisit"]),
                newVisitedNodes,
                node.sumVisitedPriority
                + int(visitedPriority[0]["TourismPriority"]),
            )
            # Check if the remaining budget and remaining time of the nodeGraph are non-negative
            if nodeGraph.remainingBudget >= 0 and nodeGraph.remainingTime >= 0:
                # Create an Arc object from the current node to the neighbor node and add it to the arcs list
                arcs.append(Arc(node, nodeGraph, int(dist[0]["Distance"])))
    return arcs
```

La funzione *neighbors*, in primo luogo, interroga la base di conoscenza per ottenere la lista dei luoghi vicini a quello indicato nel nodo corrente, sfruttando l'apposita regola Prolog:

```
nextTo(FirstPoiName, SecondPoiName) :-
    (distance(FirstPoiName, SecondPoiName, Distance); distance(SecondPoiName, FirstPoiName, Distance)),
    Distance < 501.

findNeighbors(PoiName, Neighbors) :-
    findall(Neigh, nextTo(PoiName, Neigh), Neighbors).
```

Successivamente, per ognuno di essi, verifica che non sia stato già visitato, in caso affermativo procede recuperando dalla base di conoscenza le informazioni utili a calcolare la nuova distanza percorsa, il nuovo budget rimanente e il nuovo tempo rimanente.

Tutti questi aspetti vengono utilizzati per la definizione dell'istanza *NodeGraph* relativa ad ogni nodo vicino, questo viene definito solo se non vengono sforate le soglie di budget e tempo rimanenti.

Infine, si aggiunge il nodo e l'arco corrispondente al grafo mediante la funzione *arcs.append*.

Dal momento che, la generazione del grafo avviene dinamicamente è stato necessario predisporre una funzione che controllasse man mano se il nodo in analisi fosse un nodo obbiettivo:

```python
# Check if a given node is goal.
def is_goal(self, node):
    # Query the prolog knowledge base to find the neighbors of the current node
    neighs = list(self.prolog.query(f"findNeighbors('{node.name}', Neighbors)"))[0][
        "Neighbors"
    ]

    # Initialize a flag to indicate if the node is a goal node
    isGoal = True

    # Check each neighbor of the current node
    for neigh in neighs:
        if str(neigh) not in node.visitedNodes:
            cost = list(self.prolog.query(f"price('{neigh}', Price)"))
            time = list(
                self.prolog.query(f"calculateTimeToVisit('{neigh}', TimeToVisit)")
            )
            if (
                node.remainingBudget - int(cost[0]["Price"]) >= 0
                and node.remainingTime - int(time[0]["TimeToVisit"]) >= 0
            ):
                isGoal = False
    # print(node.name)
    # print(node.visitedNodes)
    if isGoal and (
        node.name == "Start"
        or not node.sumVisitedPriority / (node.visitedNodes.__len__())
        >= 3.7  # threshold
    ):
        isGoal = False
    return isGoal
```

La funzione *is_goal*, analogamente alla precedente, interroga la base di conoscenza per ottenere una lista dei luoghi vicini a quello corrente, successivamente viene verificato che il nuovo budget e il nuovo tempo rimanenti non siano negativi, se questa condizione è rispettata, per tutti i punti di interesse vicini, il nodo corrente è considerato candidato obbiettivo.

Esso viene confermato solo nel caso in cui la media aritmetica delle priorità dei luoghi visitati, lungo il percorso, non sia inferiore ad una determinata soglia (arbitrariamente è stato scelto 3.7).

La caratteristica principale dell'algoritmo A* è la presenza di una funzione euristica, che, nel caso di studio, è stata definita come segue:

```python
# Function to calculate the heuristic value of input node
def heuristic(self, node):
    # Check if the current node is a goal node
    if self.is_goal(node):
        return 0
    else:
        # Query the prolog knowledge base to find the minimum distance
        minDistance = int(
            list(self.prolog.query(f"findMinDistance(MinDistance)"))[0][
                "MinDistance"
            ]
        )
        # Calculate the node's priority
        if node.name != "Start":
            nodePriority = round(
                list(
                    self.prolog.query(
                        f"calculateTourismPriority('{node.name}', TourismPriority)"
                    )
                )[0]["TourismPriority"],
                1,
            )
        else:
            # If the node is the start node, set its priority to 0
            nodePriority = 0
        if node.remainingTime <= node.remainingBudget:
            maxTime = int(
                list(self.prolog.query(f"findMaxTimeToVisit(MaxTimeToVisit)"))[0][
                    "MaxTimeToVisit"
                ]
            )
            # Calculate the heuristic value using the remaining time, maximum time to visit, minimum distance, and node priority
            heuristicValue = math.ceil(
                node.remainingTime
                / maxTime
                * minDistance
                * (1 - 0.05 * nodePriority)
            )
        else:
            maxCost = int(
                list(self.prolog.query(f"findMaxPrice(MaxPrice)"))[0]["MaxPrice"]
            )
            # Calculate the heuristic value using the remaining budget, maximum time to visit, minimum distance, and node priority
            heuristicValue = math.ceil(
                node.remainingBudget
                / maxCost
                * minDistance
                * (1 - 0.05 * nodePriority)
            )
        return heuristicValue
```

La funzione *heuristic* ha lo scopo di assegnare un valore stimato di costo per raggiungere un nodo obbiettivo, di conseguenza viene subito controllato se il nodo in questione sia obbiettivo o meno, in caso positivo il valore assegnato è 0. In caso contrario, mediante interrogazioni Prolog, ottiene la minima distanza possibile tra due luoghi registrati, la priorità del luogo associato al nodo in input e il tempo massimo di visita e il costo massimo richiesti da un punto di interesse (nel calcolo successivo si terrà conto del minore dei due).

Il valore restituito dall'euristica deriva dal prodotto di tre fattori:

- Il rapporto tra il tempo rimanente del nodo e il massimo tempo di visita oppure tra il budget rimanente e il costo massimo.
  L'aver messo il valore massimo al denominatore minimizza il fattore.
- La distanza minima registrata.

- Un valore via via decrescente all'aumentare della priorità.
  Questo fattore funge da discriminante in caso di valori molto simili del prodotto dei due fattori precedenti (se vicini o uguali la preferenza è diretta verso il nodo associato al luogo con priorità maggiore, così la sua euristica restituirà un valore più basso).

Minimizzare i fattori è un requisito fondamentale per far si che l'euristica restituisca valori che siano sempre sottostima del reale costo, permettendo così il corretto funzionamento dell'algoritmo.

Valutazioni

Prestazioni dell'algoritmo di ricerca

Di seguito, si osserva la differenza tra il percorso individuato da un generico searcher

```
 --> Name: Largo di Torre Argentina
Covered Distance: 2022
Remaining Budget: 21
Remaining Time: 3
Visited Nodes: ['Start', 'Obelisco del Pantheon', 'Chiesa di Sant  Ignazio di Loyola', 'Chiesa del Gesu', 'Basilica di Santa
 Maria in Ara coeli', 'Doria Pamphili Gallery']
Sum Visited Priority: 20
```

e quello individuato dall'algoritmo A* adattato:

```
 --> Name: Sant Agnese in Agone
Covered Distance: 333
Remaining Budget: 29
Remaining Time: 10
Visited Nodes: ['Start', 'Palazzo Madama', 'Piazza Navona', 'Obelisco Agonale', 'Fiumi Fountain']
Sum Visited Priority: 16
```

Si può notare come in entrambi i casi si arrivi ad un punto di terminazione dove il tempo o il budget rimanenti impediscono di proseguire (in entrambi gli esempi la risorsa terminata è il tempo, con un altro passo si andrebbe sotto lo zero). Anche la condizione sulla priorità media è rispettata: nel primo caso risulta 25 / 6 = 4,17 >= 3.7, nel secondo caso risulta 20 / 5 = 4 >= 3.7.

La differenza sostanziale dettata dall'algoritmo utilizzato sta nella distanza percorsa, che nel primo caso risulta essere molto più elevata (2022 m) in relazione alla distanza percorsa nel secondo caso (333 m).

Implementazione di CSP

In fase di realizzazione, ci si è resi conto che il problema di ricerca avrebbe potuto esser sostituito con un CSP[8] (Costraint Satisfaction Problem), tuttavia, questa osservazione è stata colta in una fase avanzata di realizzazione del caso di studio e, pertanto, si è deciso di proseguire con il problema di ricerca.

Di seguito, vengono analizzate le motivazioni a supporto dell'affermazione precedente, secondo cui sarebbe stato più opportuno l'impiego di un CSP:

- I CSP forniscono un modo strutturato per modellare i vincoli e le restrizioni del problema. Si possono definire esplicitamente le variabili, i domini delle variabili e le relazioni tra di esse. In questo caso, le variabili avrebbero potuto rappresentare i luoghi da visitare, i domini avrebbero potuto essere i costi associati a ciascun luogo e le relazioni avrebbero potuto essere i vincoli sul budget e il tempo a disposizione.
- I CSP sono noti per la loro efficacia nella risoluzione di problemi complessi. Essi offrono algoritmi di risoluzione efficienti, come ad esempio il *simulated annealing* o *algoritmi genetici (crossover)*, che consentono di esplorare in modo sistematico lo spazio delle soluzioni alla ricerca di assegnamenti validi delle variabili che soddisfino tutti i vincoli.
- I CSP consentono di gestire i vincoli soft, che sono vincoli che possono essere violati in modo controllato, consentendo una certa flessibilità nella ricerca delle soluzioni ottimali. In questo caso, si sarebbe potuto considerare i vincoli sul budget e il tempo come vincoli soft, consentendo un certo grado di flessibilità nella selezione dei luoghi da visitare in base a quanto budget e tempo rimangono disponibili.
- I CSP offrono una maggiore adattabilità ai cambiamenti dei requisiti. È possibile facilmente aggiungere, rimuovere o modificare i vincoli senza dover riprogettare completamente l'algoritmo di ricerca. Questo può essere utile se si desidera introdurre nuovi vincoli o modificare i criteri di selezione dei luoghi da visitare in base al feedback degli utenti o a esigenze specifiche.

In definitiva, l'utilizzo di un CSP avrebbe fornito un approccio più robusto e adattabile al sistema di selezione degli itinerari.

Apprendimento supervisionato (regressione)

Sommario

Durante la fase di apprendimento supervisionato[9] del progetto, è stato sviluppato un modello avanzato per addestrare un sistema in grado di predire le priorità dei monumenti nelle città per le quali non si è ancora raccolto del feedback. Questo approccio è particolarmente utile quando ci si trova di fronte a nuove città o destinazioni turistiche poco conosciute, in cui l'assenza di dati preesistenti può rendere difficile stabilire quali siano i monumenti di maggior rilievo.

Questa realizzazione, attualmente, rappresenta solo una predisposizione ad eventuali sviluppi futuri dell'applicativo, in cui si prevede di estendere le funzionalità ad un range di città più ampio.

Per affrontare questa sfida, è stato creato un set di dati di addestramento (successivamente "pulito" e scalato mediante MinMax Scaler), contenente dettagli riguardanti la storia, le recensioni degli utenti (valutazione e quantità), le dimensioni e la posizione dei luoghi di interesse della città di Roma. Tutte queste informazioni erano già disponibili, ad eccezione delle recensioni.

Per ottenerle, è stato inserito un sistema di richiesta feedback al termine della ricerca dell'itinerario turistico, in particolare viene chiesto di effettuare una valutazione (da 1 a 5) di alcuni luoghi, scelti casualmente, compresi nel percorso consigliato.

Strumenti utilizzati

Per la realizzazione dell'apprendimento supervisionato, sono stati impiegati due modelli messi a disposizione dalla libreria *Scikit Learn*[10], ossia:

- KNN, utilizzando la classe *KneighborsRegressor*[11].
- Alberi di regressione, utilizzando la classe *DecisionTreeRegressor*[12].

Per quanto riguarda tutti gli aspetti di gestione del dataset, si è scelto di utilizzare la libreria Pandas[13] che mette a disposizione l'apposita classe *DataFrame*.

Valutazioni

Scelta dei parametri

Per definire gli iperparametri di K-Nearest Neighbors (KNN) e Decision Tree, si è utilizzata la tecnica del Grid Search. Il Grid Search ha consentito di esplorare sistematicamente una griglia predefinita di combinazioni di iperparametri per determinare quelle che ottenevano le migliori prestazioni del modello in riferimento ad una metrica prestabilita.

Nel caso del KNN, gli iperparametri che abbiamo considerato includono il numero di vicini (K), il tipo di peso attribuito ad essi e la norma utilizzata.

Per quanto riguarda il Decision Tree, gli iperparametri considerati includono la profondità massima dell'albero, il criterio di divisione dei nodi e il numero minimo di campioni richiesti in un nodo per effettuare ulteriori divisioni.

In conclusione, utilizzando il Grid Search, si è stati in grado di selezionare gli iperparametri migliori per KNN e Decision Tree, ottimizzando le prestazioni dei modelli.

Valutazione delle prestazioni

Durante lo sviluppo del progetto, sono state svolte valutazioni per verificare l'efficacia delle soluzioni adottate. Uno dei metodi di valutazione impiegato è stato la *k-fold cross validation*[14], un approccio comune per valutare le prestazioni di un modello di machine learning.

Nella k-fold cross validation, il dataset viene suddiviso in k sottoinsiemi (fold) di dimensioni simili. La scelta del valore di k non è casuale, in seguito a diverse considerazioni e sulla base del fatto che gli esempi nel dataset non erano numerosi, si è optato per k = 3, dove 3 coincide con le sezioni della suddivisione concettuale del dataset (attrazioni migliori, attrazioni nella norma, attrazioni carenti) sulla base dell'indice di turismo. Successivamente, si itera k volte, selezionando ogni volta una delle fold come set di test e le rimanenti come set di addestramento. Per ciascuna iterazione avviene l'addestramento e la valutazione delle prestazioni sul rispettivo test set.

L'utilizzo della k-fold cross validation ha fornito una stima affidabile delle prestazioni del modello, consentendo di valutare la sua capacità di generalizzazione.

Sono state considerate diverse metriche di valutazione, tra cui:

- R2, è una misura che indica quanto bene il modello di regressione si adatta ai dati. Assume valori compresi tra 0 e 1, dove 1 rappresenta un perfetto

adattamento del modello ai dati. R2 misura la proporzione di variazione della variabile dipendente che può essere spiegata dalle variabili indipendenti.

- Errore assoluto medio (MAE), è una metrica che calcola la media dei valori assoluti delle differenze tra le previsioni del modello e i valori effettivi.
- Errore quadratico medio (MSE), è una metrica che calcola la media dei quadrati delle differenze tra le previsioni del modello e i valori effettivi.
- Errore massimo, rappresenta la differenza massima tra le previsioni del modello e i valori effettivi nel dataset di test.

Queste metriche ci hanno fornito una visione dettagliata delle prestazioni del modello e hanno aiutato a identificare eventuali aree di miglioramento.

Di seguito vengono riportati gli effettivi risultati ottenuti dai modelli:

- Albero di decisione

METRICHE	RISULTATI
R2	0.802389959593262
MAE	0.14019223443032372
MSE	0.0357484489241697
Max Error	0.47211538461538477

- Knn

METRICHE	RISULTATI
R2	0.47548222454791716
MAE	0.2270732089749059
MSE	0.08747204468805758
Max Error	0.732525652429514

Si può notare che l'albero di decisione ha restituito feedback estremamente positivo; infatti, la metrica r2 ha un valore di circa 0.8, il che significa che il modello lavora piuttosto bene, mentre per quanto riguarda gli errori sono tutti estremamente bassi, considerando l'intervallo delle valutazioni dei punti di interesse (da 1 a 5). L'unico che ha superato le nostre previsioni è stato l'errore massimo, ma, guardando gli altri due, ci si rende facilmente conto che si tratta di pochi casi sporadici e di conseguenza lo si può ritenere un valore accettabile.

Situazione molto diversa si verifica per il modello knn, sebbene anche in questo caso i valori di errore non si discostino troppo da quelli dell'albero, la metrica r2 è

estremamente bassa, il che significa che il modello knn non è riuscito a catturare adeguatamente i pattern o le relazioni presenti nei dati e non è riuscito a spiegare la loro variazione.

In seguito a queste osservazioni, ci si è chiesti quale fosse il motivo di tali risultati, indicando il basso numero di esempi nel dataset come possibile causa principale. Effettivamente, in presenza di un numero ridotto di esempi, il modello potrebbe soffrire di overfitting, cioè adattarsi eccessivamente ai dati di addestramento senza riuscire a generalizzare bene sui nuovi. Per esserne certi si è deciso di effettuare delle prove andando ad aggiungere i punti di interesse di altre città (prima dell'addestramento) e i risultati ottenuti in seguito sono stati i seguenti.

METRICHE	RISULTATI
R2	0.7039013082085753
MAE	0.18437611085303784
MSE	0.05959022140882684
Max Error	0.7234432292818823

Si può notare un netto miglioramento delle prestazioni, di conseguenza il knn non verrebbe escluso totalmente per eventuali sviluppi futuri in quanto si ipotizza che al crescere dei monumenti esso possa raggiungere le stesse prestazioni dell'albero, o addirittura superarle.

Decisioni di progetto

Data Cleaning

Per poter garantire un buon apprendimento dei modelli, è stato necessario selezionare solo alcune feature rispetto all'intero insieme, tutte quelle che non erano abbastanza significative o erano ridondanti non sono state considerate. L'obiettivo della regressione è la priorità, di conseguenza si è deciso di mantenere tutte quelle caratteristiche che andavano a influire su di essa all'interno della regola Prolog presentata nel capitolo di rappresentazione della conoscenza.

Un altro motivo di tale riduzione è stato il basso numero di esempi nel dataset. Diminuire la quantità di feature porta a diversi benefici, tra cui mitigare il rischio di overfitting e migliorare la capacità predittiva del modello.

Modelli impiegati

Per la fase di regressione, si è deciso di impiegare l'algoritmo K-Nearest Neighbors e l'algoritmo Decision Tree Regressor, per diversi motivi:

- Il KNN è un algoritmo di machine learning semplice e flessibile, che si basa sul concetto di vicinanza tra i punti. Il KNN utilizza i valori delle variabili indipendenti dei punti di addestramento più vicini per prevedere il valore della variabile dipendente per un nuovo punto. Questo approccio intuitivo e interpretabile permette di comprendere meglio come i valori delle variabili indipendenti influenzano la priorità assegnata ai monumenti. La sua semplicità è dovuta al fatto che richiede solo una fase di addestramento e una fase di predizione.
- Per quanto concerne l'utilizzo degli alberi di regressione, essi sono stati scelti per la loro capacità di suddividere il set di dati in modo ricorsivo in base a criteri di suddivisione ottimali, creando una struttura ad albero che rappresenta le regole di decisione, e per la loro facilità di interpretazione.

Conclusioni e sviluppi futuri

L’applicativo realizzato ha dimostrato di essere efficace nel fornire percorsi ottimali in base alle preferenze degli utenti, tenendo conto del tempo e del budget disponibili.

L'applicazione ha un potenziale di espansione verso altre città. Ciò consentirebbe ai turisti di ottenere itinerari personalizzati e ottimali per esplorare nuovi luoghi di interesse in diverse destinazioni. Per raggiungere questo obiettivo, sarà necessario acquisire dati specifici per esse, comprese le informazioni sui monumenti, le distanze tra loro e le caratteristiche individuali.

Inoltre, si potrebbe considerare di ampliare le funzionalità dell'applicazione per includere altri fattori rilevanti nella pianificazione del percorso, come le preferenze culturali, i gusti culinari o gli interessi specifici degli utenti. Questo potrebbe essere realizzato aggiungendo ulteriori attributi al dataset di addestramento e affinando i modelli di apprendimento.

Riferimenti bibliografici

[1] https://it.wikipedia.org/wiki/Base_di_conoscenza

[2] https://developers.google.com/maps/documentation/places/web-service/search-nearby?hl=it

[3] https://pypi.org/project/requests/

[4] https://it.wikipedia.org/wiki/Prolog

[5] https://artint.info/2e/html/ArtInt2e.Ch3.S1.html

[6] https://artint.info/2e/html/ArtInt2e.Ch3.S6.SS1.html

[7] https://artint.info/AIPython/

[8] https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS3.html

[9] https://artint.info/2e/html/ArtInt2e.Ch7.S2.html

[10] https://scikit-learn.org/stable/

[11] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

[12] https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

[13] https://pandas.pydata.org

[14] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html