UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO | DIPARTIMENTO DI INFORMATICA

Journey into the Heart of Rome: Unmissable Itineraries

Gruppo di lavoro

Daniele Lassandro, 778113, d.lassandro5@studenti.uniba.it

Repository GitHub

https://github.com/dlassandro70/progetto_icon_lassandro

AA 2024-2025

Introduzione

Il software opera nel campo degli itinerari turistici sulla città di Roma.
Offre all'utente una visione completa di ciò che la città ha da offrire, garantendo al contempo la possibilità di personalizzare l'esperienza in base alle proprie preferenze ed esigenze.
L'obiettivo principale è identificare il percorso di visita ottimale, privilegiando la minimizzazione della distanza percorsa, pur tenendo conto di eventuali vincoli di tempo e budget. Il tutto, assicurando al contempo un alto livello di qualità dei punti di interesse selezionati.

Sommario

Strutturazione del progetto

- **Cartella Knowledge:** raccoglie tutti i file Prolog e Python dedicati alla rappresentazione della conoscenza nella Knowledge Base utilizzata nel progetto. I file principali sono:
    - Facts.pl
    - KbManager.py
    - Landmark.py
    - Preprocessor.py
    - Rules.pl
    - RuntimeFacts.pl
    - Utility.py
- **Cartella Learning:** contiene i file Python relativi alla fase di apprendimento supervisionato. Tra questi:
    - FeedbackGenerator.py
    - Kfold.py
    - Knn.py
    - ModelInitializer.py
    - PreProcessorLearning.py
    - RegressionTree.py
- **Cartella Logs:** comprende tutti i file di log testuali utilizzati per monitorare l'esecuzione degli script presenti nelle altre cartelle.
- **Cartella Search:** include i file Python utilizzati per risolvere il problema di ricerca su grafo. Al suo interno:
    - **Cartella Libs:** contiene gli algoritmi di base impiegati per lo sviluppo del problema di ricerca.
    - ItinerarySearchProblem.py
    - MainSearch.py: il file principale dell'applicativo.
    - NodeGraph.py
- **Cartella Storage:** raccoglie tutti i file serializzati in formato pickle.

Elenco argomenti di interesse

Di seguito vengono elencati i macro-temi affrontati durante lo sviluppo dell'applicativo:

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

Nella fase di definizione dell'idea progettuale e della sua realizzazione, ho cercato di coinvolgere la più ampia gamma di argomenti trattati nel corso di Ingegneria della conoscenza.

Rappresentazione della conoscenza

Sommario

Il primo passo nello sviluppo dell'applicativo turistico è stato raccogliere i dati relativi ai punti di interesse della città di Roma. Questi dati sono stati utilizzati per creare le istanze della classe di riferimento **Landmark**.

Gli attributi definiti in questa classe includono:

- **placeId, name, address, type, properties, lat, lon, age:**
  rappresentano le caratteristiche "anagrafiche" del punto di interesse.
- **rating, ratingCount:**
  indicano la popolarità e la qualità del luogo sulla base delle valutazioni degli utenti.
- **centreDistance:**
  misura la distanza rispetto a un punto centrale della città, definito arbitrariamente come fulcro turistico.
- **tourismRate, price:**
  descrivono le caratteristiche "turistiche" che influenzano la visitabilità del luogo.
- **handicapAccessibility, surface, height:**
  rappresentano gli attributi strutturali del punto di interesse.

Tutti gli altri campi presenti nella classe vengono popolati successivamente, in una fase successiva di elaborazione.

Le istanze create sono poi state memorizzate in un dizionario, e tutte le feature sono state salvate in una **base di conoscenza**[1] sotto forma di fatti. La maggior parte di essi segue il formato standard:

feature(landmark_name, feature_value).

Di seguito sono riportati 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* differisce dal formato Prolog delle altre: *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 riveste un ruolo centrale nel progetto, in quanto costituisce la base del modulo di ricerca su grafo, cuore del caso di studio. Per questo motivo, è stato necessario conservare tutti i fatti ad essa correlati nell'apposito file Prolog, cosa che invece non è stata fatta per tutte le feature generate successivamente.

La Knowledge Base, una volta arricchita con diverse regole, è stata interrogata per produrre nuova conoscenza. Tra le regole implementate, alcune risultano particolarmente rilevanti:

- **calculateDensity:** consente di determinare la densità turistica di un luogo, calcolata come la media delle distanze rispetto a tutti gli altri punti registrati, recuperati precedentemente tramite il predicato Prolog findall.

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

- **calculateTourismPriority:** questa regola permette di determinare la priorità di visita di un punto di interesse, basandosi sulle diverse feature che lo caratterizzano.
Ogni feature contribuisce in maniera differente al calcolo finale, il quale viene poi memorizzato nel dizionario e utilizzato nel modulo di ricerca su grafo. Questo garantisce che l'itinerario turistico rispetti una soglia minima di qualità dei luoghi selezionati. Inoltre, la priorità influisce sul valore dell'euristica utilizzata nell'algoritmo di ricerca.

```
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:** questa regola determina il tempo necessario per visitare un luogo. Il calcolo tiene conto sia degli aspetti strutturali, come le dimensioni, sia di quelli turistici: infatti, all'aumentare del tasso di turismo, il luogo risulta più affollato e il tempo di visita aumenta di conseguenza.

```
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 viene mostrato un frammento di codice Python utilizzato per interrogare la base di conoscenza per ciascun punto di interesse registrato (fase di generazione delle nuove feature). I risultati ottenuti vengono raccolti in un contenitore (result) e, successivamente, estratti e assegnati all'attributo corrispondente dell'istanza di **Landmark** presente 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 producono risultati di tipo diverso, come ad esempio valori booleani, seguono lo stesso approccio logico.

```
# 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, che raccoglie tutte le istanze di Landmark, è stato aggiornato con i valori delle nuove feature e serializzato, così da poter essere utilizzato nei moduli successivi del progetto.

Strumenti utilizzati

La raccolta dei dati utili è stata realizzata tramite l'uso di una API a pagamento fornita da Google Places (la piattaforma di Google Maps): nearbysearch[2], che permette di ottenere una lista di luoghi situati nelle vicinanze di una determinata posizione geografica.

Per il suo utilizzo è stato necessario generare (gratuitamente) una API-Key ed è stata impiegata la libreria Requests[3] fornita da 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 risultati corretti; di seguito sono indicati quelli utilizzati nel presente caso di studio:

- API-Key
- Le coordinate di latitudine e longitudine del centro di ricerca, da cui vengono individuati i risultati.
- Il raggio massimo entro cui effettuare la ricerca dei luoghi.
- Il tipo di luoghi, espresso tramite una parola chiave, sulla base della quale avviene il filtraggio per ottenere risultati specifici.

La risposta fornita dalla chiamata è in formato JSON e contiene numerosi attributi. Tra questi, sono stati selezionati ed estratti quelli più utili all'applicativo:

- **place_id:** identificativo assegnato a ciascun elemento registrato su Google.
- **user_ratings_total:** intero che rappresenta il numero di recensioni degli utenti relative all'elemento.
- **name:** nome del luogo.
- **geometry.location.lat** e **geometry.location.lon:** coordinate spaziali del luogo in formato decimale.
- **rating:** media delle valutazioni (da 1 a 5) contenute nelle recensioni.

Le altre caratteristiche dei punti di interesse, difficili da reperire, sono state generate casualmente utilizzando la libreria **random** di Python. La generazione casuale è stata effettuata rispettando intervalli realistici per ciascuna feature.

Per la rappresentazione della conoscenza e la sua successiva interrogazione, è stato scelto il linguaggio Prolog[4], un linguaggio logico basato su predicati che rappresentano fatti o relazioni tra oggetti. Un programma Prolog è costituito da predicati e regole: le regole servono per derivare nuovi fatti o per rispondere a interrogazioni sulla knowledge base.

Decisioni di progetto

Molteplicità delle chiamate

Poiché le API di Google restituivano venti risultati per ciascuna chiamata, è stato necessario ripetere le chiamate più volte fino a ottenere l'intero set di risultati.
Per ottenere un numero più consistente di luoghi, ho deciso di effettuare quattro chiamate API, utilizzando quattro diverse coppie di coordinate. La scelta di quattro punti differenti è stata fatta per massimizzare i risultati ottenibili, spostandosi di 1 km a nord, sud, est e ovest rispetto al centro turistico selezionato.
La coppia di coordinate iniziale è stata scelta arbitrariamente come punto centrale che contenesse un numero sufficiente di luoghi, così da poter essere definito come centro turistico di Roma.

Assiomatizzazione del dominio nella base di conoscenza

L'assiomatizzazione del dominio nella knowledge base è stata realizzata scrivendo su file i fatti relativi ai vari luoghi. Ho scelto di utilizzare il metodo **write** di Python per scrivere sul file Prolog, invece del metodo **assertz**, perché le informazioni della knowledge base dovevano essere consultabili anche negli altri moduli del progetto. Il metodo **assertz** sarebbe stato più indicato se la knowledge base avesse richiesto l'inserimento dinamico dei fatti durante una singola esecuzione del programma. Nel caso di studio, ogni modulo corrisponde a un'esecuzione separata.

Problema di ricerca su grafo

Sommario

L'obiettivo del caso di studio è fornire ai turisti un punto di riferimento per visitare i luoghi principali di Roma, proponendo il percorso migliore per ottimizzare le risorse disponibili (tempo e budget).
Per individuare il percorso ottimale, è stato utilizzato un algoritmo di ricerca su grafo[5] in grado di considerare le esigenze dell'utente e restituire un itinerario contenente i luoghi più significativi da visitare.

Strumenti utilizzati

L'algoritmo impiegato per la ricerca su grafo è stato **A***[6], un algoritmo euristico che calcola il percorso ottimale tra due nodi sommando il costo accumulato fino al nodo corrente con una funzione euristica che stima il costo residuo fino al nodo goal:

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

Per l'implementazione di A* è stata utilizzata la libreria AIPython[7], che contiene un insieme di algoritmi di ricerca già predefiniti.

Decisioni di progetto

Struttura dei nodi

Per l'utilizzo dell'algoritmo di ricerca, è stata definita una classe che rappresenta i nodi del grafo. I nodi non corrispondono direttamente ai luoghi di interesse, ma rappresentano le "situazioni" durante il percorso.
Ogni nodo contiene le seguenti informazioni:

- **name:** nome del luogo.
- **coveredDistance:** distanza percorsa fino al nodo corrente.
- **remainingBudget:** budget residuo disponibile.
- **remainingTime:** tempo residuo disponibile.
- **visitedNodes:** luoghi già visitati.
- **sumVisitedPriority:** somma delle priorità dei luoghi visitati.

Gli ultimi quattro attributi sono utilizzati durante la fase di ricerca per effettuare controlli specifici sul percorso.

Adattamento del problema di ricerca

Nella definizione del grafo, i nodi goal non sono noti a priori, poiché la loro generazione avviene in modo dinamico a partire dalla posizione corrente dell'utente (che nel caso di studio è stata simulata). Una generazione statica del grafo avrebbe impattato negativamente sulle prestazioni dell'applicativo.

Durante la fase di ricerca è indispensabile disporre di una funzione per individuare i nodi 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** interroga innanzitutto la knowledge base per ottenere la lista dei luoghi adiacenti a quello presente 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 ciascun nodo vicino, verifica che non sia stato già visitato; se non lo è, recupera dalla knowledge base le informazioni necessarie a calcolare:

- la nuova distanza percorsa,
- il budget residuo aggiornato,
- il tempo rimanente aggiornato.

Questi valori vengono poi utilizzati per creare l'istanza **NodeGraph** corrispondente al nodo vicino. L'istanza viene definita solo se i limiti di budget e tempo non vengono superati.
Infine, il nodo e l'arco corrispondente vengono aggiunti al grafo tramite la funzione **arcs.append**.

Poiché il grafo viene generato dinamicamente, è stato necessario implementare una funzione in grado di verificare progressivamente se il nodo in esame rappresenta un nodo obiettivo:

```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 a quanto visto per la funzione precedente, interroga la knowledge base per ottenere la lista dei luoghi vicini al nodo corrente. Successivamente verifica che il budget e il tempo rimanenti non siano negativi; se questa condizione è soddisfatta, tutti i punti di interesse vicini rendono il nodo corrente un candidato obiettivo.
Il nodo viene confermato come goal solo se la media aritmetica delle priorità dei luoghi visitati lungo il percorso non scende al di sotto di una soglia prestabilita (arbitrariamente scelta pari a 3.7).

La caratteristica principale dell'algoritmo **A*** è la presenza di una funzione euristica, definita nel caso di studio 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 il compito di assegnare un valore stimato di costo per raggiungere un nodo obiettivo. Per prima cosa verifica se il nodo corrente coincide con un goal: in tal caso il valore restituito è pari a 0. In caso contrario, attraverso interrogazioni Prolog, recupera:

- la distanza minima possibile tra due luoghi registrati,
- la priorità del luogo associato al nodo in input,
- il tempo massimo di visita e il costo massimo richiesti da un punto di interesse (nel calcolo viene considerato il minore tra i due valori).

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 sì che l'euristica sia ammissibile e che quindi restituisca valori che siano sempre sottostima del costo reale, 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

Durante lo sviluppo è emerso che il problema di ricerca avrebbe potuto essere sostituito con un CSP[8] (Costraint Satisfaction Problem). Tuttavia, tale osservazione è stata formulata in una fase avanzata della realizzazione del caso di studio e si è preferito proseguire con l'approccio basato sulla ricerca su grafo.

Le motivazioni a supporto dell'ipotesi secondo cui un CSP sarebbe stato più opportuno sono le seguenti:

- I CSP forniscono un modello strutturato per la rappresentazione di **variabili**, **domini** e **vincoli**. In questo contesto, le variabili avrebbero potuto rappresentare i luoghi da visitare, i domini i relativi costi, e i vincoli sarebbero stati legati a tempo e budget disponibili.
- I CSP sono noti per la loro efficacia nella risoluzione di problemi complessi e dispongono di algoritmi efficienti (ad esempio *simulated annealing* o algoritmi genetici con *crossover*) che permettono un'esplorazione sistematica dello spazio delle soluzioni alla ricerca di assegnamenti validi.
- I CSP consentono la gestione di **vincoli soft**, ossia vincoli violabili entro certi limiti. In questo caso, tempo e budget avrebbero potuto essere trattati come vincoli soft, introducendo flessibilità nella scelta dei luoghi da visitare.
- I CSP offrono una maggiore adattabilità ai cambiamenti dei requisiti: l'aggiunta, la modifica o la rimozione dei vincoli può essere effettuata senza dover riprogettare completamente l'algoritmo. Questo aspetto risulta utile nel caso si vogliano introdurre nuovi criteri di selezione o adattare quelli esistenti in base al feedback degli utenti.

In definitiva, l'utilizzo di un CSP avrebbe garantito un approccio **più robusto, flessibile 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 in scenari con nuove città o destinazioni turistiche poco conosciute, dove l'assenza di dati preesistenti rende difficile stabilire quali siano i monumenti di maggiore rilievo.

Questa realizzazione rappresenta al momento una predisposizione a futuri sviluppi dell'applicativo, con l'obiettivo di estenderne le funzionalità ad un insieme più ampio di città.

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

Per raccogliere tali dati, è stato predisposto un **sistema di richiesta feedback** al termine della ricerca dell'itinerario turistico: in particolare, viene chiesto all'utente di esprimere una valutazione (da 1 a 5) di alcuni luoghi, selezionati casualmente tra quelli inclusi 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, ho scelto di utilizzare la libreria Pandas[13] che mette a disposizione l'apposita classe *DataFrame*.

Valutazioni

Scelta dei parametri

Per la definizione degli iperparametri di **K-Nearest Neighbors (KNN)** e **Decision Tree** è stata utilizzata la tecnica del **Grid Search**.
Questo approccio ha permesso di esplorare sistematicamente una griglia predefinita di combinazioni di iperparametri al fine di individuare quelli in grado di garantire le migliori prestazioni in riferimento ad una metrica prestabilita.

- Per il **KNN**, gli iperparametri presi in esame sono stati:
    - il numero di vicini (*K*),
    - il tipo di peso da attribuire a ciascun vicino,
    - la norma utilizzata per il calcolo delle distanze.
- Per il **Decision Tree**, invece, sono stati considerati:
    - la profondità massima dell'albero,
    - il criterio di divisione dei nodi,
    - il numero minimo di campioni richiesti in un nodo per consentire ulteriori suddivisioni.
    - Il numero minimo di campioni richiesti in una foglia

In conclusione, grazie al Grid Search è stato possibile selezionare gli iperparametri più adeguati per KNN e Decision Tree, ottimizzando così 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, ho 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	0.760868719551273
MAE	0.1468280368994595
MSE	0.04413782649618613
Max Error	0.5512820512820512

- **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 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, mi sono chiesto 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 certo ho 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 ipotizzo 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 risultavano ridondanti non sono state considerate. L'obiettivo della regressione è la priorità, di conseguenza ho 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, ho 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 sia del tempo sia del budget disponibili.

L'applicazione presenta un significativo 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 ciascuna località, comprendenti informazioni sui monumenti, sulle distanze tra essi e sulle loro caratteristiche individuali.

Un ulteriore sviluppo potrebbe riguardare l'ampliamento delle funzionalità dell'applicazione, includendo fattori aggiuntivi nella pianificazione del percorso, come le preferenze culturali, i gusti culinari o gli interessi specifici degli utenti. Questo obiettivo potrebbe essere perseguito arricchendo il dataset di addestramento con ulteriori attributi e affinando i modelli di apprendimento impiegati.

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