DOCUMENTAZIONE TECNICA DEL BACKEND - GreenRatingAlgorithm

(Versione Gennaio 2026)

1. Architettura Generale

Il backend è un'API REST monolitica basata su Flask. Il flusso logico di una richiesta è il seguente:

1. un client (Angular) invia una richiesta POST /api/greenRatingAlgorithm con un poligono GeoJSON che rappresenta l'area catturata e selezionata dall'utente;
2. il server Flask riceve la richiesta;
3. il server costruisce ed esegue 3 query sequenziali all'API Overpass per scaricare i dati OSM (edifici, alberi aree verdi), girando 3 endpoint diversi di OSM in caso di server sovraccarichi o richieste respinte;
4. i dati JSON grezzi vengono convertiti e "spacchettati" in GeoDataFrame (GDF) ottimizzati tramite GeoPandas e Pandas;
5. i GDF puliti vengono passati al modulo analizzatore_centrale.py, che orchestra l'esecuzione dei 3 algoritmi (regola3, regola30, regola300) in sequenza;
6. gli algoritmi eseguono calcoli geospaziali (buffer, sjoin, clip, line-of-sight) usando GeoPandas;
7. i risultati (sia i dati grezzi che quelli filtrati) vengono serializzati in GeoJSON e restituiti al client.

2. Strumenti principali

2.1 GeoPandas & Shapely

Pandas è una libreria Python per l'analisi e la manipolazione dei dati. Il suo componente principale è il DataFrame, ossia una tabella in-memoria che permette di caricare, pulire, filtrare, aggregare e unire set di dati complessi in modo incredibilmente efficiente. GeoPandas è Pandas per i dati geografici. Estende i DataFrame di Pandas per includere una colonna "geometry", permettendo di trattare poligoni, punti e linee come dati. Shapely invece è la libreria C sottostante che esegue materialmente le operazioni (es. buffer, intersects) sui dati contenuti nella colonna geometry di un GDF. La maggior parte delle operazioni quindi sono ottimizzate e scritte nella libreria in C/C++.

2.2 Overpass API

Overpass API è un'API di sola lettura che permette di interrogare il database completo di OpenStreetMap (OSM). Ci permette di analizzare qualsiasi area del mondo senza dover scaricare e gestire enormi file statici (es. Shapefile).

3. Regole 3-30-300

Tutti gli algoritmi:

1. proiettano tutti i GDF in un CRS metrico (EPSG:32632) per permettere calcoli in metri;
2. spesso i dati provenienti da OSM sono sporchi e quasi sempre non omogenei, non permettendo un'analisi uniforme sugli attributi dei dataset. I GDF vengono quindi puliti accuratamente per rimuovere geometrie non valide, per assicurare che contenga solo i tag rilevanti (es. natural='tree') e gestendo altre tipologie di problemi sintattici (spazi, maiuscole/minuscole);
3. vengono sempre creati indici spaziali (R-tree) una sola volta all'inizio (es. ostacoli_idx = edifici_proj.sindex), per lavorare più velocemente e efficentemente.

3.1. Regola 3 (Regola3.py)

L'obiettivo di questo algoritmo è implementare la "Regola 3", che stabilisce che ogni cittadino dovrebbe poter vedere almeno 3 alberi dalla propria abitazione. Data la natura dei dati OSM (2D, senza altezze), l'algoritmo implementa un'approssimazione 2D di questa regola. Prende in input il GDF degli edifici dell'area di studio e il GDF degli alberi. Restituisce edifici_conformi, un GDF copia dell'originale, ma con una nuova colonna aggiunta, 'visible_trees_count', una variabile integer che contiene il numero di alberi visibili.

L'algoritmo itera su ogni edificio del dataset di input. Per ciascuno:

a. buffer approssimativo: crea un buffer metrico (45 metri) per definire l'area di ricerca;
b. ricerca alberi: per trovare gli alberi nel buffe usa in prima battuta alberi_idx.intersection() per ottenere una lista ridotta di "candidati", per poi eseguire il costoso calcolo geometrico .within() solo sul piccolo sottoinsieme di alberi rimasti;
c. controllo ostruzione: per ogni albero trovato, chiama la funzione is_unobstructed(). Questa funzione controlla diverse caratteristiche di compatibilità edificio-albero:
    i. scompone l'edificio, composto solo dai propri vertici, nei lati, creati con un segmento tracciato due a due sui vertici;
    ii. riduce la geometria dell'edificio di 0.5m verso l'interno. Questo è stato fatto per ridurre drasticamente problemi dovuti al sovrapposizionamento di edifici concatenati (in questo modo vengono separati), che spesso generano falsi negativi. I falsi positivi ottenuti come contraccambio da questa modifica delle geometrie giustifica drasticamente la scelta;
    iii. per ogni lato, traccia una LineString (una linea) 2D tra il centroide dell'albero e il centroide del lato;
    iv. calcola l'angolo compreso tra la LineString creata e il lato stesso; se questo lato è maggiore di +-60° rispetto alla normale, la linea viene scartata. In questo modo si eliminano anche i falsi positivi creati dal punto iii, e si crea una sorta di 'visuale simulata' più realistica (non devi uscire dal balcone per vedere l'albero);

v. usa ostacoli_idx.intersection() per trovare i "probabili" ostacoli (spesso 0 o 1), poi esegue il costoso .intersects() solo su quei pochissimi candidati (dopo aver rimosso l'edificio campione dalla lista). Se la linea non interseca nessun ostacolo, l'albero passa ed è considerato visibile.

Non appena uno di questi controlli fallisce, l'algoritmo prova con gli altri lati dell'edificio. Se nessuno di loro passa tutti i controlli, l'albero è considerato non visibile per quell'edificio, passando così al successivo.

Quando un albero viene etichettato come visibile, viene quindi aggiornata la variabile di conteggio, e salvato l'id dell'albero in un apposito array che andrà ad aggiungersi come nuova colonna degli edifici.

3.2. Regola 30 (Regola30.py)

L'algoritmo calcola questa percentuale per l'intera area di studio definita dall'utente. Il risultato è un singolo valore percentuale (es. 2.86%) che viene poi assegnato a tutti gli edifici all'interno di quell'area per determinarne la conformità.
Prende in input il GDF degli edifici (attualmente non utilizzato nel calcolo, ma passato per coerenza architetturale), il GDF degli alberi e polygon_gdf, il GDF che rappresenta il poligono di studio disegnato dall'utente. Ritorna percentage (float), un singolo valore numerico che rappresenta la percentuale di copertura.

In sostanza, calcola l'area totale, il denominatore della nostra frazione, e l'area degli alberi, il numeratore. Per calcolare il primo, utilizza la funzione .sum() che esegue il calcolo in automatico. Per il numeratore invece, gli alberi di OSM sono solo dei punti, privi di geometria. Viene quindi approssimato un valore ponendo il raggio uguale a 2 metri. L'uso di un raggio fisso per gli alberi puntiformi è un'approssimazione significativa. Questo valore è un numero ambiguo e andrebbe validato o sostituito.

NB: l'algoritmo adotta un'interpretazione "pura" di "copertura arborea", come spiegato nei paper originali della regola. Esclude deliberatamente le "Aree Verdi Ricreative" (Query 2: leisure=park, landuse=grass) dal calcolo. Questa è una scelta metodologica che impatta direttamente e significativamente il risultato finale (spiegando perché è spesso <10%).

3.3. Regola 300 (Regola300.py)

L'obiettivo di questo algoritmo è implementare la "Regola 300", che stabilisce che ogni cittadino dovrebbe vivere entro 300 metri da un'area verde accessibile. L'algoritmo calcola, per ogni singolo edificio, se questa regola è soddisfatta (punteggio 1) o meno (punteggio 0), controllando la vicinanza con le "Aree Verdi Ricreative" (parchi, prati, giardini) di ALMENO un ettaro di area. Le aree ricreative che non rispettano questa soglia non vengono proprio incluse nell'input dell'algoritmo, pertanto non vengono visualizzate dall'utente. Prende in input il GDF degli e delle aree_verdi, mentre restituisce risultato_finale (), una copia del GDF

edifici originale, ma con due nuove colonne aggiunte, ossia score_300 (int), che contiene 1 se l'edificio è conforme, 0 altrimenti, e l'array con gli id delle aree verdi che hanno reso conforme l'edificio alla regola.

Questo algoritmo è il più efficiente del set, poiché è completamente vettorizzato e non usa cicli for in Python.

Proietta tutti i GDF in CRS metrico, crea un buffer metrico di 300 metri attorno a tutti gli edifici in un'unica operazione vettorizzata: edifici_buffer['geometry'] = edifici_proj.geometry.buffer(300), ed esegue un gpd.sjoin (Spatial Join) tra i buffer degli edifici e i poligoni delle aree verdi. sjoin è un'operazione GIS ad alte prestazioni (basata su indici R-tree automatici) che restituisce un nuovo GeoDataFrame contenente solo gli edifici il cui buffer di 300m interseca almeno un'area verde. L'algoritmo infine estrae gli indici unici (.unique()) degli edifici "vincitori" dallo sjoin e li usa per mappare i punteggi 1 (conformi) sul GDF edifici originale. Agli edifici non presenti nel risultato dello sjoin rimane il punteggio di default 0.

L'unico compromesso rispetto ai paper originali è che l'algoritmo calcola la distanza in linea d'aria (buffer di 300m), e non il percorso pedonale reale (che richiederebbe un'analisi di rete su grafo stradale, es. con OSMnx, ed è computazionalmente molto più costoso). È un'approssimazione standard per la pianificazione urbana, ma la distanza reale da percorrere a piedi sarà quasi sempre superiore a quella calcolata.

NB: gestione "Edge Effect" (discorso analogo vale per gli alberi prelevati ai fini della regola 3 e 30): per evitare che un edificio sul bordo dell'area di studio "perda" un parco vicino che è appena fuori dal poligono disegnato, la query Overpass in server.py viene eseguita su un poligono di ricerca "gonfiato" di 300 metri (o 45 nell'analogia della regola 3 e 30).

4. Dati: input, output

4.1. Dati DAL Client (input Angular)

Il backend espone un singolo endpoint (/api/greenRatingAlgorithm) che si aspetta una richiesta POST contenente un oggetto JSON. L'unica chiave richiesta da questo oggetto è polygon.

- Formato: polygon è un Array di Array di coordinate.
- Ordine Coordinate: L'ordine è [latitudine, longitudine], che è il formato nativo usato da Leaflet quando si disegna sulla mappa.
- Requisito: Per essere un poligono valido, l'array deve contenere almeno 4 punti.

Esempio di Payload (Input):

```
{ "polygon": [
   [45.464, 9.188],
   [45.462, 9.188],
   [45.462, 9.190],
   [45.464, 9.190],
   [45.464, 9.188]
 ]}
```

4.2 Dati AL client (output backend):

Il metodo chiave è .to_json(), che converte il GDF in una stringa GeoJSON FeatureCollection. Il codice implementa un controllo di sicurezza per evitare crash su GDF vuoti.

Il dizionario finale che ritorna al client contiene le stringhe GeoJSON e il set di errori (non bloccanti) generati dal backend durante l'elaborazione (non esiste per ora visualizzazione nel frontend di questo):

```
    if errori:
        flag = False
        messaggio = "Analisi completata con errori non bloccanti delle seguenti regole: " + "; ".join(errori)
    else:
        flag = True
        messaggio = "Analisi completata con successo."

    risultato = {
        'EsecuzionePositiva': flag,
        'messaggio': messaggio,
        'alberi': alberi_geojson,
        'aree_verdi': aree_verdi_geojson,
        'risultati': risultati_geojson
    }
```

5. Tabella riassuntiva scelte progettuali prese dal team:

Regola	Parametro Chiave	Valore Attuale	Note
Generale	Dati	OpenStreetMap (OSM)	Sono dati vettoriali ottenuti in live dai server OSM. In parallelo stiamo lavorando con i dati YOLO di Enrico Antonini.
Regola 3	Distanza Visuale	45 metri (buffer)	-
Regola 3	Ostacoli Visivi	Edifici	Muri e altre barriere non sono calcolati per performance e semplicità
Regola 30	Raggio Chioma	2 metri	Usato per simulare l'area poiché l'albero è un punto. La stima è molto a ribasso.
Regola 30	Soglia Calcolo	Area Poligono Utente	La % è calcolata sull'area disegnata dall'utente nel frontend, non sulla città
Regola 300	Distanza	300 metri	Linea d'aria (Euclidea), non pedonale
Regola 300	Filtro Parchi	> 1 Ettaro	Scartiamo aiuole/parchi troppo piccoli (< 10.000 mq)

7. Casi Limite

L'algoritmo, lavorando su dati non omogenei e non controllati, può incontrare configurazioni topologiche/geometriche che generano risultati anomali. Di seguito sono documentati i casi noti non coperti dall'attuale implementazione (potrebbero essercene altri):

- adiacenza edificio-area verde: In rari casi, se la geometria di un edificio condivide un lato o un vertice esattamente con il poligono di un'area verde (es. un palazzo costruito da un utente con geometria sovrapposta ad un parco), le librerie geometriche potrebbero non rilevare correttamente l'intersezione o il buffer a causa di tolleranze di precisione. In questo scenario, la regola 3 potrebbe restituire 0 alberi visibili anche in presenza di alberi adiacenti e altrimenti visibili;
- geometrie bucate: per edifici 'normali' per come è stato implementato l'algoritmo, il verso 'esterno' è individuabile automaticamente. Considerato che gli edifici di OpenStreetMap sono disegnati da utenti comuni, non è garantito il verso con cui l'edificio stesso viene disegnato (i vertici potrebbero essere stati inseriti in modo orario come in modo antiorario). Questo provoca un effetto collaterale per edifici con cortili interni (forma a ciambella), in quanto l'algoritmo non è implementato per "capire" sia il verso esterno che quello interno. Di conseguenza, il calcolo della

normale rispetto ai lati non funziona, e l'edificio termina l'esecuzione con 0 alberi visibili (nonostante potrebbero essercene);

- errori di inserimento: su OpenStreetMap, capita che utenti inesperti posizionino il nodo di un albero dentro la sagoma di un edificio. L'algoritmo attuale di LineOfSight verifica l'intersezione con altri edifici, e l'autostruzione nel caso di linee che partono dal lato opposto rispetto all'albero. Tuttavia, il caso di albero dentro l'edificio stesso, elude questi controlli (ancora da capire come. Il risultato è che l'edificio in questione vedrà l'albero, mentre correttamente gli edifici vicini non lo vedranno (perché la linea di vista intersecherà l'edificio che contiene l'albero). E' ancora da stimare se questo sia effettivamente da considerare errore, in quanto relativo ai dati di input e non all'algoritmo in sé (segnalo inoltre che questo tipo di alberi è presente anche nei file YOLO di Enrico Antonini).

8. Differenze input OSM-YOLO-MANUALE

In riferimento alla richiesta di estrazione dati quantitativi sulle città target (Verona, Genova, Sanremo, Campi Bisenzio), di seguito i conteggi disponibili sui dataset in mio possesso.

Nota metodologica sui dati: Non essendo in possesso dei dataset originali di Enrico Antonini (pre-modifica), ho lasciato quei campi liberi. Inoltre, segnalo che l'accuratezza del poligono d'area di analisi non è del 100% tra dati OSM e dati YOLO, in quanto devo disegnare a mano la zona per le query OSM cercando di prendere l'area coperta da YOLO. Perciò i dati OSM potrebbero variare, seppur di poco, dal reale confronto di zona (potrebbero essere leggermente di più come leggermente di meno).

Città	OSM	YOLO Enrico	YOLO con modifiche di Alessio
Verona	3070	75885	77695 (+1810)
Genova	2913	139944	143566 (+3622)
Sanremo	100	18620	19568 (+948)
Campi Bisenzio	813	11191	11971 (+780)

Come si può notare, la differenza è abissale. Questo è dovuto principalmente al fatto che OpenStreetMap NON tratta gli alberi in proprietà private, mentre le immagini satellitari sono agnostiche da questo punto di vista. Inoltre, luoghi come boschi e foreste, ricchi di alberi, vengono interpretati come elemento unico in OSM, con tag diverso e proprietà differenti. L'integrazione del dataset YOLO ha portato a un incremento medio dei dati disponibili pari a oltre 70 volte il volume originale censito su OSM, con picchi di arricchimento del 19.000% nel caso di Sanremo.

Di seguito invece, lo stesso confronto ma considerando solo le sotto-zone dove sono state eseguite delle modifiche/aggiunte manuali.

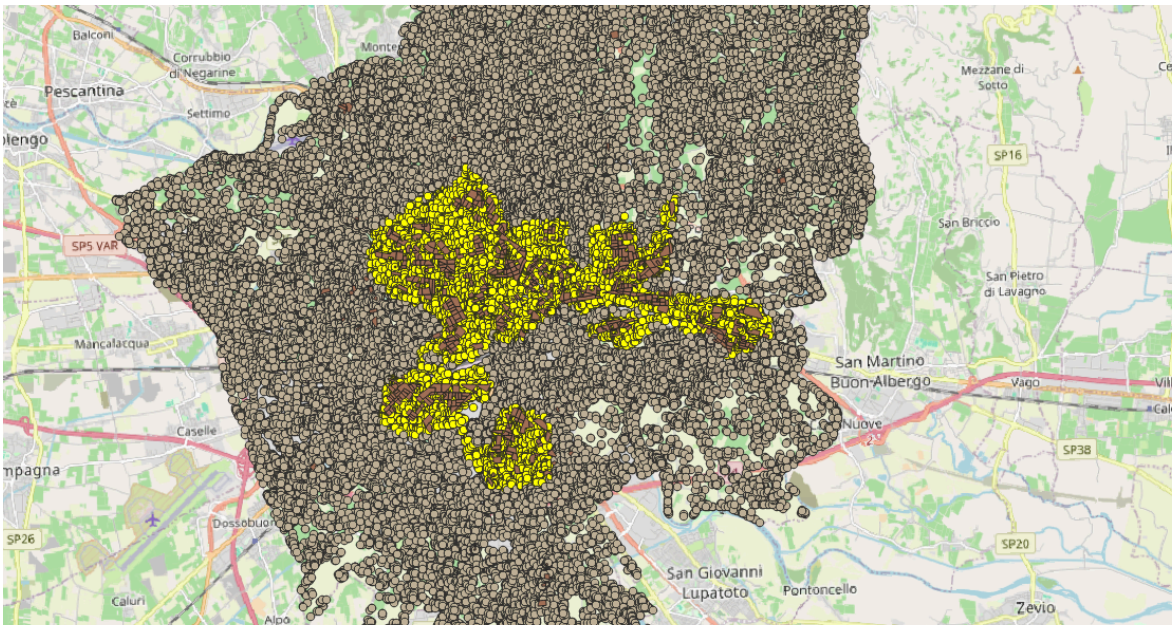
Città (zone ad alta densità)	OSM	YOLO Enrico	YOLO con modifiche di Alessio
Verona	947	17156	18966 (+1810)
Genova	1688	21643	25265 (+3622)
Sanremo	83	843	1791 (+948)
Campi Bisenzio	131	1687	2467 (+780)

Da notare bene come, dalle seguenti immagini prese durante l'analisi, le zone ad alte densità sono diverse e sparpagliate. L'analisi OSM e Yolo post modifica sono state fatte sulle zone di maggior concentrazione e densità, pertanto i valori Yolo pre-modifica potrebbero differire dai valori reali di poco.

Genova (in verde tutti gli alberi Yolo post modifica, in marrone le zone ad alte densità, in giallo gli alberi prelevati per l'ultima tabella):

Sanremo (stessa scala di colori):

Verona (stessa scala di colori):



Campi Bisenzio (rosa tutti gli alberi yolo post modifica, marrone zone ad alta densità, gialli gli alberi selezionati per la tabella):