

Progetto di Sistemi Context Aware:
Smart Parking System
A.A. 2021-2022

Luca Genova 1038843
Marco Benito Tomasone 1038815
Simone Boldrini 1038792

2022-2023

Indice

1	Introduzione	2
2	Progettazione	4
2.1	Feature Aggiuntive	5
3	Implementazione	6
3.1	App Mobile	6
3.2	Splash Screen	6
3.2.1	Activity Recognition	7
3.3	MapWidget	8
3.3.1	Feature Aggiuntiva	10
3.4	Backend	11
3.5	Frontend	14
4	Algoritmi di classificazione e Risultati	16
4.1	Dataset e tecnologie utilizzate	16
4.1.1	Data mining	17
4.2	Gli algoritmi	18
4.2.1	KNN	18
4.2.2	Random Forest	20
4.2.3	Gaussian Naive Bayes	22
4.3	Confronto dei risultati	23
4.4	Accuratezza sistema Har integrato	24
4.5	Implementazione del Random Forest nell'app	25
5	Conclusioni	27

Capitolo 1

Introduzione

Il parcheggio è una delle sfide più comuni che i conducenti affrontano quotidianamente nelle città. Trovare un posto auto libero in una zona affollata può richiedere molto tempo e aumentare il traffico, causando stress e frustrazione per i conducenti. Per risolvere questi problemi, è stato proposto un sistema di crowdsensing dei parcheggi che monitora la disponibilità dei posti auto in una zona specifica della città di Bologna. Il sistema fornisce informazioni sul numero di posti auto liberi, aiutando i conducenti a risparmiare una quantità notevole di tempo.

Il sistema si basa sul riconoscimento dell'attività dell'utente e della posizione per determinare lo stato di guida o cammino dell'utente e fornire informazioni sullo stato dei parcheggi disponibili. Questo è il cuore del progetto e consente di tenere conto degli eventi di ingresso (DRIVING → WALKING) e uscita (WALKING → DRIVING) dal parcheggio.

L'esecuzione di questo progetto è stato suddiviso in due fasi:

- La prima fase consiste nel realizzare il sistema di parcheggio intelligente che monitora la disponibilità dei posti auto in una zona specifica della città di Bologna. Esso è diviso in tre parti:
 - **Un'app mobile:** l'applicazione mobile che permette ai conducenti di visualizzare le informazioni sullo stato dei posti auto in una zona specifica. La funzionalità dell'app è basata sul riconoscimento della posizione e dell'attività dell'utente.
 - **Un frontend:** il sito web che permette agli amministratori di visualizzare le informazioni sulle richieste di parcheggio in una zona specifica, di utilizzare l'algoritmo K-means per visualizzare gli eventi di parcheggio in cluster e anche di visualizzare la heatmap di essi.
 - **Un backend:** il server che gestisce le richieste dell'applicazione e fornisce le informazioni sullo stato dei posti auto interrogando un database PostgreSQL (con estensione PostGIS) tramite query spaziali.

- La seconda fase consiste nell’analizzare le prestazioni di tre diversi algoritmi di classificazione: K-Nearest Neighbors, Random Forest e Gaussian Naive Bayes. Questi algoritmi vengono, poi, confrontati con le prestazioni del sistema integrato all’interno dell’applicazione, per determinare quale sia il più adatto per essere implementato all’interno di essa. In questo modo, è possibile ottenere una soluzione più precisa e affidabile per fornire informazioni sullo stato dei posti auto disponibili.

Il presente report, quindi, fornisce una descrizione dettagliata del funzionamento del sistema di parcheggio intelligente e analizza le sue prestazioni e la sua fattibilità. Verrà presentato il design del sistema, le tecnologie utilizzate e i risultati delle prove effettuate. Inoltre, verrà fornita una valutazione complessiva del sistema e verranno proposte possibili soluzioni per eventuali problemi riscontrati.

Capitolo 2

Progettazione

Nella prima fase di questo progetto, abbiamo eseguito un’analisi approfondita dei requisiti specifici. Abbiamo identificato le seguenti esigenze per l’app mobile, il server e il frontend:

- **Applicazione mobile:** L’applicazione mobile deve essere in grado di riconoscere l’attività dell’utente, la variazione dell’attività di un utente (WALKING → DRIVING e viceversa), visualizzare una mappa con le varie aree poligonali di Bologna, chiedere al backend la disponibilità di parcheggi all’interno di una certa area poligonale, riconoscere la posizione dell’utente e inviare al server gli eventi di entrata e uscita di un parcheggio.
- **Server:** Il server deve essere in grado di ricevere e rispondere alle richieste di disponibilità di parcheggi da parte dell’app, ricevere le richieste di entrata e uscita di parcheggi da parte dell’app, inviare le aree poligonali di Bologna all’app e, in caso di zone prive di dati (ad esempio per mancanza di utilizzo dell’app da parte di utenti in quella zona) deve prevedere un meccanismo di interpolazione dei dati, per stimare il numero di parcheggi disponibili.
- **Frontend:** Il frontend deve essere in grado di visualizzare, per ogni zona della città, il numero totale di richieste di parcheggio provenienti da quella zona, utilizzare il clustering K-Means per visualizzare gli eventi di parcheggio e rappresentare gli eventi di parcheggio mediante una heatmap.

Per quanto riguarda la seconda fase, è stato richiesto di valutare l’accuratezza di tre classificatori di attività binari (solo WALKING e DRIVING). In particolare si è lavorato su un dataset fornito per allenare il modello di Human Activity Recognition (HAR). Sono stati testati tre algoritmi di classificazione: Random Forest, KNN, Guassian Bayes. Inoltre, è stata valutata l’accuratezza del sistema di HAR della libreria nell’app mobile mediante un piccolo testbed (esperimenti con pochi dati reali) e confrontata con l’accuratezza degli algoritmi testati, tenendo presente che il confronto ha una valenza scientifica

limitata in quanto si basa su dataset differenti. Infine, è stato integrato il Random Forest direttamente nell'app mobile prendendo i dati dai sensori e sono state sviluppate due feature aggiuntive a discrezione del gruppo.

2.1 Feature Aggiuntive

In aggiunta alle feature precedenti, la consegna prevedeva l'inclusione di due feature aggiuntive a discrezione del gruppo. Data la natura del progetto, abbiamo pensato a delle feature che potessero rendere più completo l'utilizzo del sistema da parte dell'utente e che rispecchiassero i contenuti del corso di studi. Abbiamo quindi deciso di implementare come prima feature aggiuntiva un secondo algoritmo di clustering già visto a lezione: l'algoritmo **DBSCAN**. L'algoritmo DBSCAN è un algoritmo di clustering basato su densità, che permette di individuare cluster di dati che sono vicini tra loro e che sono separati da dati che non fanno parte del cluster. Come seconda feature aggiuntiva abbiamo trovato un dataset offerto dagli Open-Data del comune di Bologna che presenta la posizione di tutte le colonnine per la ricarica elettrica della città di Bologna. Abbiamo quindi integrato all'interno dell'app un sistema di crowdsensing che permette agli utenti di verificare lo stato di disponibilità di una cerca colonnina. Quando un utente parcheggerà il proprio veicolo nei pressi di una colonnina per la ricarica, apparirà un pop-up nell'applicazione che chiederà all'utente se sta utilizzando la colonnina. In caso di risposta affermativa il backend aggiornerà lo stato di disponibilità della colonnina che sarà quindi visibile agli altri utenti. La colonnina verrà liberata quando l'utente che l'ha occupata avrà lasciato il parcheggio. L'implementazione di queste feature aggiuntive ci ha permesso di estendere le funzionalità del nostro progetto con funzioni utili agli utenti pur rimanendo all'interno di implementazioni legate allo scopo del corso.

Capitolo 3

Implementazione

In questa parte del report verranno descritte le tecnologie utilizzate per la realizzazione del sistema e le scelte fatte per la sua implementazione.

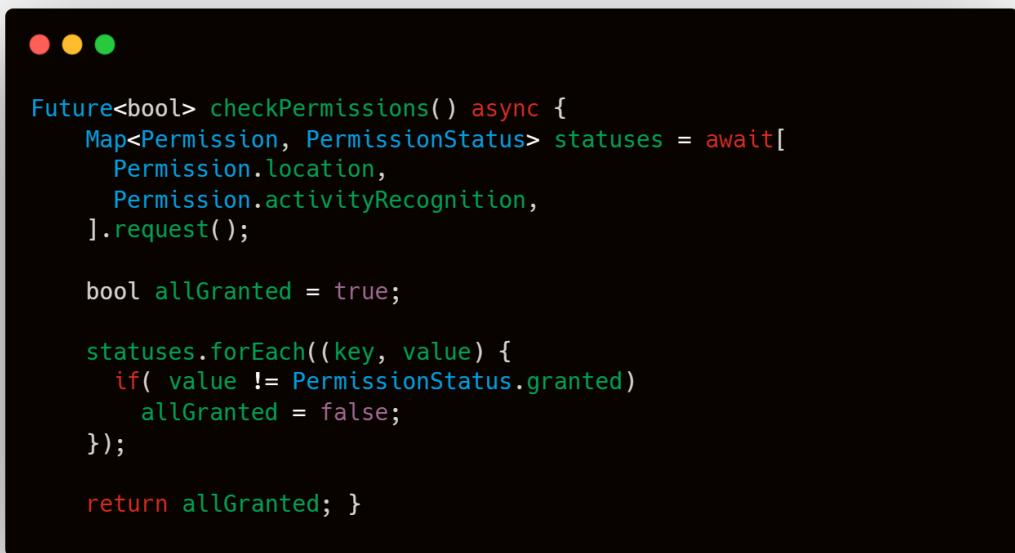
3.1 App Mobile

Per offrire un'app mobile che possa essere utilizzata sia su sistema operativo Android che iOS abbiamo deciso di utilizzare il framework *Flutter*, un framework open-source sviluppato da Google che si poggia su *Dart*. Il progetto è stato però testato solo su android poichè nessun componente del gruppo possedeva un dispositivo con MacOS. Per la gestione della mappa abbiamo utilizzato il pacchetto *flutter_map* che permette di integrare comodamente delle mappe Leaflet all'interno delle propria app, mentre per quanto riguarda la libreria per fare Activity Recognition abbiamo utilizzato un pacchetto chiamato *flutter_Activity_Recognition*, un wrapper alle librerie di sistema Android e iOs per fare riconoscimento dell'attività dell'utente.

3.2 Splash Screen

Lo splash screen è la prima schermata che viene mostrata all'utente quando avvia l'app. Questa schermata è molto semplice e mostra solo il logo dell'app, una semplice icona di un'auto con il nome dell'applicazione. Questa schermata ha lo scopo di intrattenere l'utente durante il caricamento dell'app e di chiedergli i permessi per la localizzazione e per il riconoscimento dell'attività.

Mostriamo il codice per la richiesta dei permessi:



```
● ● ●

Future<bool> checkPermissions() async {
  Map<Permission, PermissionStatus> statuses = await [
    Permission.location,
    Permission.activityRecognition,
  ].request();

  bool allGranted = true;

  statuses.forEach((key, value) {
    if( value != PermissionStatus.granted)
      allGranted = false;
  });

  return allGranted; }
```

Figura 3.1: Codice per la richiesta dei permessi

Qualora l'utente non conceda i permessi, invece che caricare Map Widget verrà caricato un altro Widget con un messaggio di errore.

3.2.1 Activity Recognition

Per la parte di Activity Recognition, come precedentemente accenato abbiamo utilizzato il pacchetto *flutter_Activity_Recognition* che permette di fare riconoscimento dell'attività dell'utente. Il pacchetto riconosce varie attività come: *WALKING*, *RUNNING*, *IN_BICYCLE*, *IN_VEHICLE*, *STILL*, *UNKNOWN*. L'implementazione del sistema di activity recognition è stata fatta tramite la classe *ActivityRecognitionClass*. Questa classe viene inizializzata all'interno del MapWidget (la schermata principale dell'app). Al momento della sua inizializzazione questa classe farà partire un listener che farà partire una callback ogni qualvolta viene riconosciuta una nuova attività. Importante notare come verranno eseguite operazioni (aggiornamento dell'attività corrente ed eventuali transizioni) solo dal momento in cui l'attività riconosciuta viene riconosciuta con un grado di confidenza alto. In più, tutto funziona se e solo se l'utente ha dato i permessi all'app di far riconoscere la propria attività.

Mostriamo ora il costruttore della classe e la callback che viene eseguita ogni qualvolta

viene riconosciuta una nuova attività.



The screenshot shows a mobile application window with three colored status bar dots (red, yellow, green). The main screen displays a portion of the ActivityRecognition class code in a monospaced font. The code includes a constructor and a callback method:

```
ActivityRecognition(Function updateCurrentActivity){  
    this.updateCurrentActivity = updateCurrentActivity;  
    isPermissionGrants();  
    _activityStreamSubscription = activityRecognition.activityStream  
        .handleError(_handleError)  
        .listen(_onActivityReceive);  
}  
  
void _onActivityReceive(Activity activity) {  
    //Took only the activities that interest us  
    if(activity.confidence == ActivityConfidence.HIGH)  
        this.updateCurrentActivity(activity.type);  
}
```

Figura 3.2: Costruttore e callback di ActivityRecognitionClass

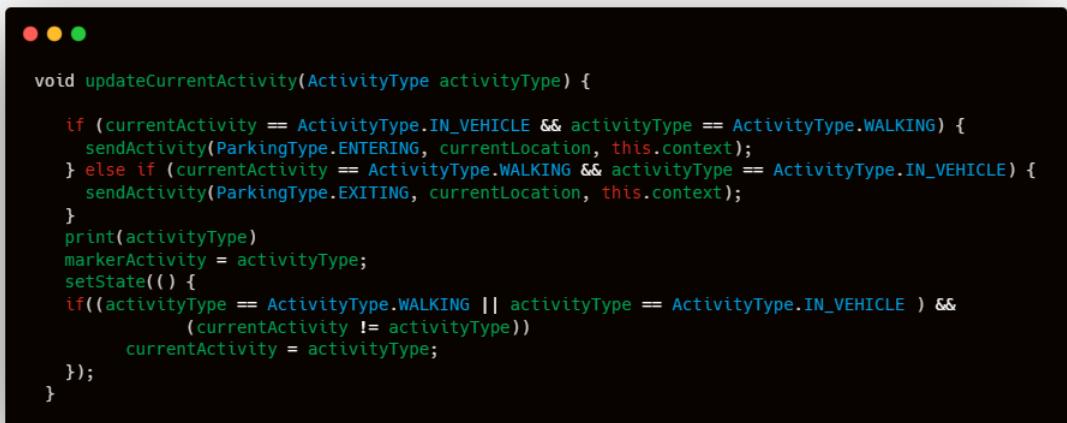
3.3 MapWidget

MapWidget rappresenta la schermata principale dell'applicazione. In questa schermata l'utente può vedere la sua posizione sulla mappa che mostrerà i poligoni. Cliccando su un poligono verrà visualizzato un Toast che indica il numero di parcheggi disponibili per quella zona. Sono inoltre presenti (solo in questa versione Beta dell'app a scopo dimostrativo) dei pulsanti che permettono di simulare l'arrivo e la partenza dall'area di parcheggio, pulsanti che non sarebbero presenti in un'ipotetica versione commerciale dell'app.

Per la gestione della mappa abbiamo utilizzato il pacchetto *flutter_map* che permette di integrare comodamente delle mappe Leaflet all'interno delle propria app. Per la visualizzazione dei poligoni ci siamo affidati al Layer Polygon di Leaflet, quindi non sono altro che un layer aggiuntivo sulla mappa.

Il riconoscimento della posizione è stato fatto tramite il pacchetto Geolocator che permette di ottenere la posizione dell'utente in tempo reale.

Lato implementativo, questo widget è il widget più importante di tutta l'applicazione. Difatti nell'initState di questo widget esso si occuperà di inizializzare anche altre classi usate all'interno del progetto, come la classe per fare il riconoscimento dell'attività dell'utente. Lo snippet di codice più importante di questa classe è probabilmente la funzione che viene chiamata nella callback del listener dell'activity Recognition: updateCurrentActivity. Questa funzione si occupa di aggiornare l'attività corrente dell'utente e di inviare un messaggio al server per indicare che l'utente è entrato o uscito dall'area di parcheggio.



```

void updateCurrentActivity(ActivityType activityType) {
    if (currentActivity == ActivityType.IN_VEHICLE && activityType == ActivityType.WALKING) {
        sendActivity(ParkingType.ENTERING, currentLocation, this.context);
    } else if (currentActivity == ActivityType.WALKING && activityType == ActivityType.IN_VEHICLE) {
        sendActivity(ParkingType.EXITING, currentLocation, this.context);
    }
    print(activityType)
    markerActivity = activityType;
    setState(() {
        if((activityType == ActivityType.WALKING || activityType == ActivityType.IN_VEHICLE ) &&
           (currentActivity != activityType))
            currentActivity = activityType;
    });
}

```

Figura 3.3: Funzione updateCurrentActivity

Lo state currentActivity verrà aggiornato solo se l'attività riconosciuta è *WALKING* o *IN_VEHICLE* poichè spesso è molto facile che guidando o camminando ci si fermi, portando il sistema a riconoscere l'attività come *STILL*. Questo causerebbe un'impossibilità da parte dell'app di riconoscere un cambiamento netto di attività da *WALKING* a *DRI-VING* e viceversa. Tutte le altre attività vengono comunque riconsociute e salvate nella variabile markerActivity che, tramite uno switch permette di cambiare la visualizzazione a schermo, permettendo all'utente di capire quale attività viene riconosciuta in quel momento dal sistema.

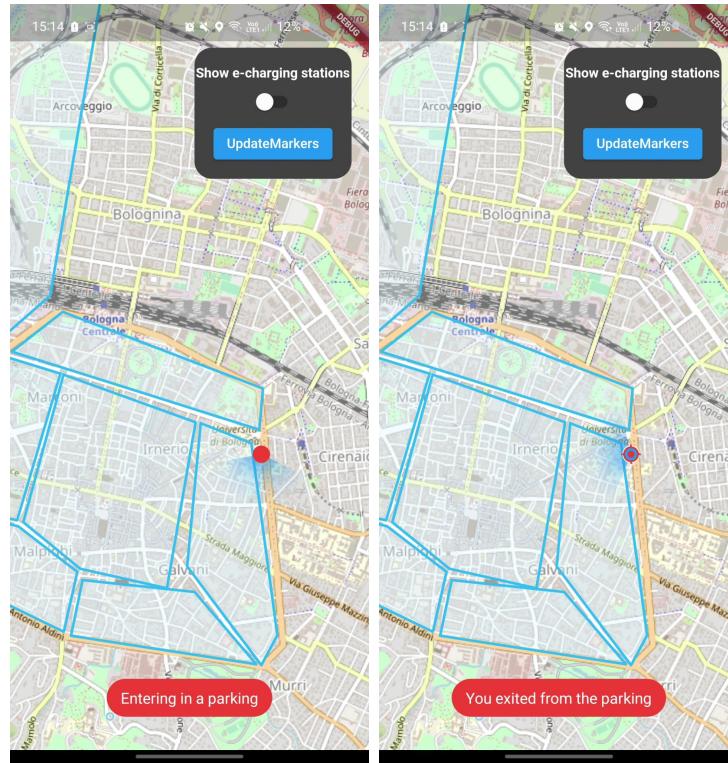


Figura 3.4: Riconoscimento di un’entrata e di un’uscita da un parcheggio

3.3.1 Feature Aggiuntiva

Lato app mobile, per l’implementazione della feature aggiuntiva ci limitiamo a chiedere al backend i dati relativi alle colonnine elettriche, che visualizzeremo all’interno della mappa.

Quando un utente parcheggerà il suo veicolo, se si trova in una posizione vicina ad una colonnina elettrica, l’app visualizzerà una dialog che chiederà all’utente se sta utilizzando o meno quella colonnina elettrica. Se l’utente risponde di sì, il backend si occuperà di aggiornare lo stato della colonnina nel database.

Mostriamo ora un esempio di questa funzione, tramite una chiamata ad una posizione fissata vicino ad una colonnina elettrica posta in via S.Giacomo:

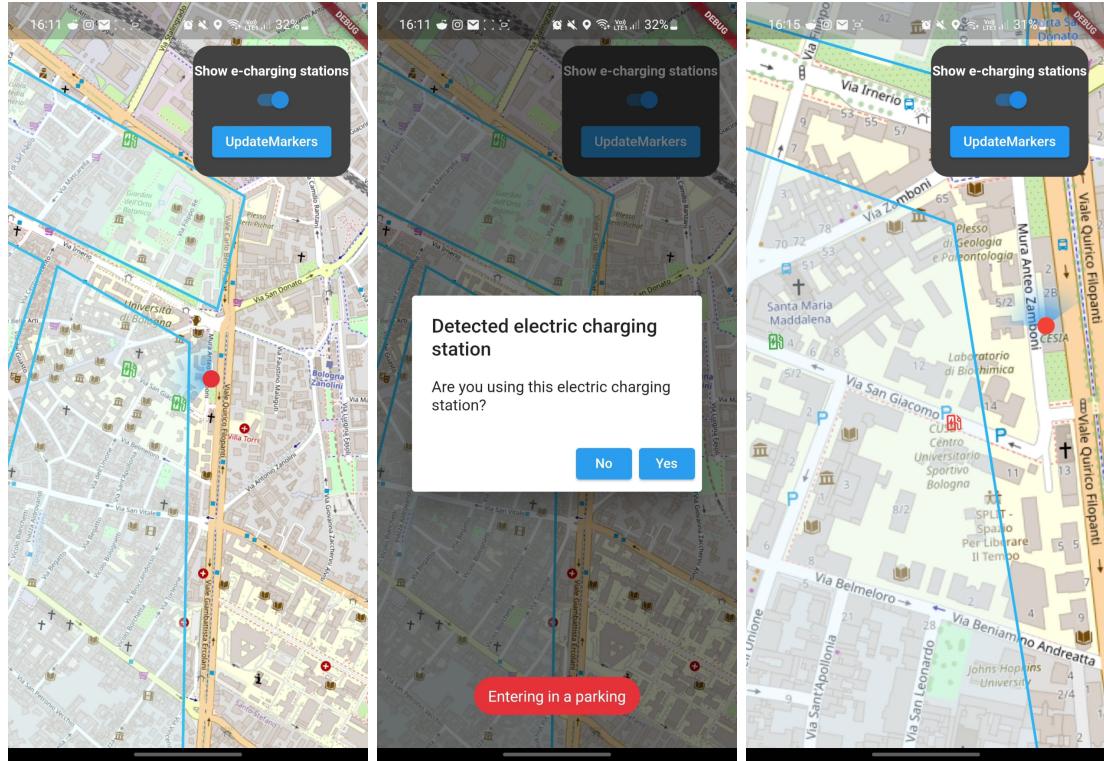


Figura 3.5: Riconoscimento di un’entrata e di un’uscita da un parcheggio nei pressi di una colonnina

Dalle immagini si può notare come l’utente sia abbastanza distante dalle colonnine, questo poichè per motivi di test abbiamo deciso di passare una posizione fissata vicina alla colonnina.

3.4 Backend

Per il backend abbiamo deciso di utilizzare *Node.js*, un runtime Javascript che permette di creare applicazioni web veloci e scalabili, il Database invece, è stato creato utilizzando *PostgreSQL* con estensione *PostGIS* per la gestione dei dati spaziali.

Lato implementativo abbiamo deciso di dividere il nostro database in moduli, in modo da avere una struttura più pulita e ordinata. Avremo quindi un modulo per la creazione del database, uno per le query e rispettivamente uno per le funzioni esposte al frontend e all’applicazione mobile.

Il nostro database è composto da 5 tabelle e ha questo schema:

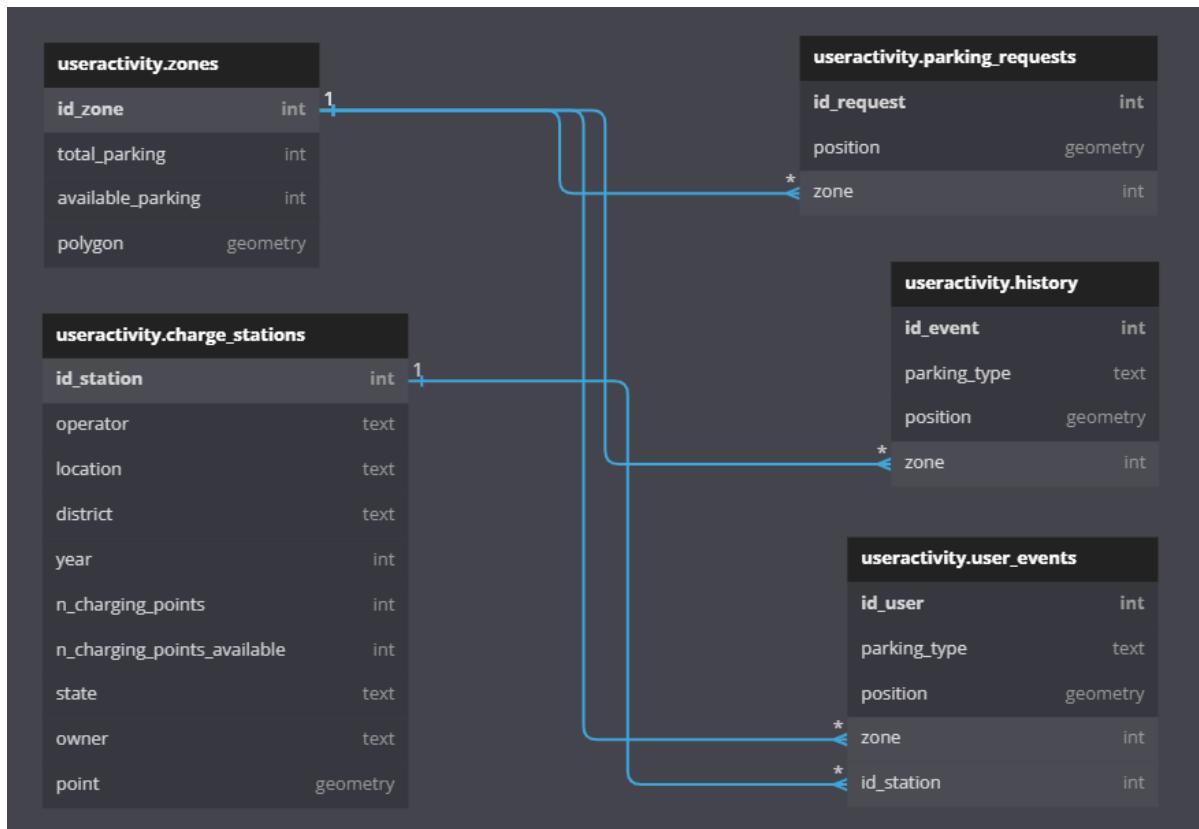


Figura 3.6: Schema del database

Il database è stato gestito tramite pgAdmin, un software open-source che permette di gestire database PostgreSQL, al quale abbiamo utilizzato anche il plugin PostGIS per la gestione dei dati spaziali.

Entrando nello specifico delle query, mostriamo la query che viene chiamata quando un utente chiede il numero di parcheggi disponibili in una zona:

```

getParkingsInterpolation: async (position) => {
    const client = new Client(QUERY_CONFIGURATION);
    await client.connect();
    try {
        const zone = await module.exports.find_zone(position);
        let n_parkings;
        if (zone instanceof Error)
            throw new Error(zone.message);
        const parkingInZone = await get_nParkingEvents_for_zone(zone);
        console.log("Parking Events in zone: " + zone)
        console.log(parkingInZone);
        if(parkingInZone < 5){
            console.log("I'm interpolating the result from other zones")
            n_parkings = await parkingIDWInterpolation(zone);
            console.log("Used parking in zone : " + zone)
            console.log(n_parkings);
            const total_parking = await client.query(`SELECT total_parking FROM zones WHERE id_zone = ${zone}`);
            n_parkings = total_parking.rows[0]['total_parking'] - parseInt(n_parkings) - parkingInZone;
        } else {
            n_parkings = await module.exports.getParkingsFromZone(zone);
        }
        console.log("Free Parking in zone: " + zone)
        console.log(n_parkings);
        return n_parkings;
    } catch (e) {
        console.error(e);
        return e;
    }
    finally {
        await client.end();
    }
},

```

Figura 3.7: Query per il numero di parcheggi disponibili

Questa query ci permette di discutere di un'altra importante feature presente nel progetto, l'interpolazione dei parcheggi. Come si può notare dapprima calcoleremo il numero di posti occupati in zona, se questo è minore di 5, un numero molto basso per una città come Bologna dove ci sono molti parcheggi, allora procederemo con l'interpolazione dei dati, altrimenti restituiamo semplicemente il numero di posti liberi in zona.

L'interpolazione dei dati è stata fatta tramite l'algoritmo IDW (Inverse Distance Weighting), un algoritmo che permette di interpolare i dati in base alla distanza tra i punti. Per calcolare la distanza tra due zone, abbiamo dapprima calcolato i centroidi delle zone, poi abbiamo calcolato la distanza tra i centroidi delle zone e infine abbiamo interpolato i dati in base alla distanza tra i centroidi.

Dato un insieme di punti: $\{\mathbf{x}_i, u_i \mid \text{per } \mathbf{x}_i \in \mathbb{R}^n, u_i \in \mathbb{R}\}_{i=1}^N$, la funzione di interpolazione $u(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$ è definita come:

$$u(\mathbf{x}) = \begin{cases} \frac{\sum_{i=1}^N w_i(\mathbf{z}) u_i}{\sum_{i=1}^N w_i(\mathbf{x})}, & \text{if } d(\mathbf{x}, \mathbf{x}_i) \neq 0 \text{ per ogni } i \\ u_i, & \text{if } d(\mathbf{x}, \mathbf{x}_i) = 0 \text{ per alcune } i \end{cases}$$

Dove

$$w_i(\mathbf{x}) = \frac{1}{d(\mathbf{x}, \mathbf{x}_i)}$$

Guardando l'implementazione lato codice:

```
SELECT ST_AsText( ST_Centroid( polygon ) )FROM zones WHERE id_zone=${zone}
```

Questa query ci permette di calcolare il centroide di una zona, dato il suo id.



```
for(row in nParkEventsForZone.rows) {
    nParkEventsForZone.rows[row].distance = computeDistance(nParkEventsForZone.rows[row].centroid,
    centroidOfZoneOfInterest);
}
interpolatedValue = 0;
//Compute result
for(row in nParkEventsForZone.rows) {
    if(nParkEventsForZone.rows[row].distance != 0)
        interpolatedValue += nParkEventsForZone.rows[row].count * (1/nParkEventsForZone.rows[row].distance);
    else
        interpolatedValue += nParkEventsForZone.rows[row].count;
}

sumOfDistances = 0;
for(row in nParkEventsForZone.rows)
    if(nParkEventsForZone.rows[row].distance != 0)
        sumOfDistances += (1/nParkEventsForZone.rows[row].distance);

interpolatedValue = interpolatedValue / sumOfDistances;
return interpolatedValue;
```

Figura 3.8: Calcolo dell'interpolazione dei dati

Questo codice rappresenta invece la computazione effettiva del valore interpolato.

3.5 Frontend

Per motivi di riusabilità del codice e dei componenti abbiamo deciso di utilizzare il framework React per il frontend, un framework *Javascript* open-source sviluppato da Facebook che permette di creare componenti riutilizzabili. Per la gestione delle mappe nel frontend abbiamo utilizzato la libreria *Leaflet*, un framework Javascript open-source che permette di creare mappe interattive, dal quale abbiamo preso anche due pacchetti per fare Clustering e per visualizzare la Heatmap dei dati. Infine per la parte grafica è stato utilizzato il framework *Material-UI*, un framework Javascript open-source che

permette di creare componenti grafici in stile *Material Design*.

Il frontend è composto da tre sezioni, accessibili tramite delle tab in una navbar:

- La sezione delle richieste: in cui è possibile vedere le zone poligonali di Bologna, selezionarne una e visualizzare il numero di posti liberi in quella zone ed il numero di richieste di parcheggio, cioè quante volte gli utenti hanno chiesto la disponibilità di parcheggi in quella zona.
- La sezione della Heatmap: in cui è possibile visualizzare la mappa di Bologna con la heatmap dei dati, cioè il numero di richieste di parcheggio per ogni zona poligonale. La Heatmap è visibile sia complessiva che suddivisa per poligono.
- La sezione del Clustering: in cui è possibile visualizzare una visualizzazione dei dati in forma di cluster, cioè dei ragguagliamenti dei dati in base alla loro vicinanza. Sono presenti due visualizzazioni possibili, il clustering K-means e il clusterings DBSCAN.

Di seguito mostriamo delle schermate della sezione della Heatmap, sia complessiva che suddivisa per poligono:



Figura 3.9: Heatmap complessiva e suddivisa per poligono

Mostriamo ora la schermata della sezione del Clustering, in cui è possibile visualizzare i cluster K-means e DBSCAN:



Figura 3.10: Schermata della sezione del Clustering

Capitolo 4

Algoritmi di classificazione e Risultati

Nella presente fase del progetto, è stata effettuata la valutazione di tre algoritmi di classificazione: **KNN**, **Random Forest** e **Gaussian Bayes**. L'obiettivo era quello di eseguire un confronto tra i tre algoritmi e di determinare quale di essi fosse più preciso nel classificare i dati utilizzando il dataset HAR fornito.

La valutazione è stata effettuata attraverso la misurazione dell'accuratezza degli algoritmi e il confronto dei risultati ottenuti. Questo ha permesso di determinare quale algoritmo fosse il più adatto per il problema di riconoscimento delle attività svolte dall'utente.

4.1 Dataset e tecnologie utilizzate

Il dataset utilizzato per l'analisi ci è stato fornito dal prof. Marco Di Felice. Il dataset è composto da 62.584 osservazioni e 13 colonne riguardanti il valore dei sensori di uno smartphone android e l'attività dell'utente, come seguente:

accelerometer#mean	la media delle osservazioni dell'accelerometro.
accelerometer#min	l'osservazione minima dell'accelerometro.
accelerometer#max	l'osservazione massima dell'accelerometro.
accelerometer#std	la deviazione standard delle osservazioni dell'accelerometro.
gyroscope#mean	la media delle osservazioni del giroscopio.
gyroscope#min	l'osservazione minima del giroscopio.
gyroscope#max	l'osservazione massima del giroscopio.
gyroscope#std	la deviazione standard delle osservazioni del giroscopio.
gyroscopeuncalibrated#mean	la media delle osservazioni del giroscopio non calibrato.
gyroscopeuncalibrated#min	l'osservazione minima del giroscopio non calibrato.
gyroscopeuncalibrated#max	l'osservazione massima del giroscopio non calibrato.
gyroscopeuncalibrated#std	la deviazione standard delle osservazioni del giroscopio non calibrato.
target	l'attività svolta dall'utente.

Le etichette di questo dataset assumono 5 valori differenti:

- **STILL**: l'utente non si muove.
- **WALKING**: l'utente cammina.
- **CAR**: l'utente è in auto.
- **BUS**: l'utente è in bus.
- **TRAIN**: l'utente è in treno.

4.1.1 Data mining

Per il data mining è stato utilizzato il linguaggio di programmazione *Python* e per il training e prediction dei modelli la libreria *Scikit-learn*. Inoltre, durante la fase di pre-processing e per la gestione dei dati è stata utilizzata la libreria di Python *Pandas*.

Non è stato fatto un grande preprocessing dei dati, in quanto il dataset fornito era già di ottima qualità. Per la rimozione dei valori mancanti, abbiamo rimosso solo le righe per le quali mancavano almeno due attributi per ogni classe di sensori. Questo ci ha portato a perdere circa 5000 tuple. Per quelle righe in cui era presente solamente un valore mancante questo è stato sostituito con la media. I dati sono stati lavorati al fine di eliminare i valori NaN e di normalizzare i dati, scalandoli utilizzando la funzione *Min-MaxScaling*. Inoltre sono state eliminate tutte le tuple che non riguardavano le attività di interesse (WALKING e DRIVING).

Per tutti e tre gli algoritmi il dataset è stato suddiviso in due parti:

- **Training**: è stato utilizzato per il training del modello.

- **Testing:** è stato utilizzato per il testing del modello.

Per il training è stato utilizzato l'80% dei dati e per il testing il 20%.

4.2 Gli algoritmi

Come già detto precedentemente, in questa fase di valutazione degli algoritmi di classificazione, sono stati considerati tre differenti algoritmi: il **KNN**, il **Random Forest** e il **Gaussian Naive Bayes**.

Questi algoritmi rappresentano tre approcci distinti alla risoluzione del problema di classificazione, ciascuno con i suoi punti di forza e limiti. Il **KNN** utilizza una logica basata sulla vicinanza dei dati, il **Random Forest** è un algoritmo di ensemble che utilizza molteplici alberi di decisione e il **Gaussian Naive Bayes** si basa sull'assunzione che le feature sono distribuite normalmente.

In questa sezione, verranno descritti in dettaglio ciascuno di questi algoritmi e verrà presentato un confronto tra i loro risultati e le loro prestazioni.

4.2.1 KNN

Per quanto riguarda il KNN è stato deciso di effettuare un tuning degli iperparametri per il KNN, con lo scopo di ottenere una prestazione ottimale del modello.

I parametri che sono stati modificati sono stati:

- **n_neighbors:** rappresenta il numero di vicini considerati nella classificazione
- **weights:** determina il peso dei vicini nella classificazione
- **leaf-size:** rappresenta la dimensione massima di una foglia dell'albero di ricerca
- **p:** determina la potenza della metrica utilizzata nella calcoli della distanza tra i vicini

Il tuning degli iperparametri è stato effettuato utilizzando la funzione *GridSearchCV* di *Scikit-learn*. Questa funzione permette di eseguire una ricerca su una griglia di valori per gli iperparametri, testando tutte le possibili combinazioni e selezionando quella che produce la migliore prestazione in termini di accuratezza.

L'obiettivo di questa ottimizzazione era quello di migliorare la precisione del modello KNN e di fornire una soluzione ottimale per la classificazione dei dati. Il risultato ottenuto ha dimostrato che l'ottimizzazione degli iperparametri è stata efficace e ha contribuito a migliorare la performance del modello.

Risultati

I risultati del tuning degli iperparametri utilizzando il dataset precedentemente descritto sono riportati nella tabella seguente:

	Best value
n_neighbors	14
weights	distance
leaf_size	1
p	2

Con questi parametri siamo riusciti ad ottenere un'accuratezza finale dell'89%.

Facendo training di questo modello con il dataset su cui abbiamo applicato il MinMax-Scaling, abbiamo ottenuto un'accuratezza del 90%.

I parametri risultanti dal tuning sono invece:

	Best value
n_neighbors	12
weights	distance
leaf_size	1
p	2

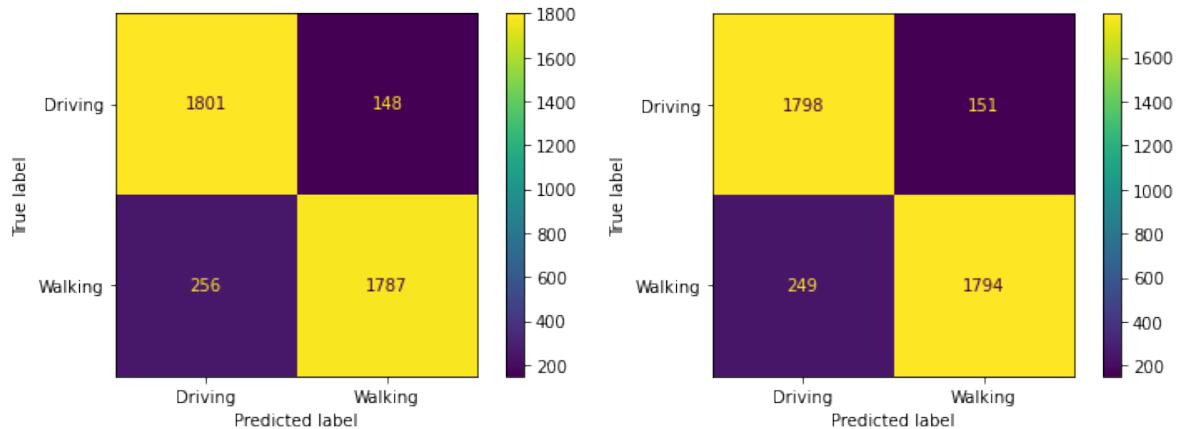


Figura 4.1: Matrice di confusione del modello senza scaling (sx) e con scaling (dx)

Si può notare come questo modello commetta più facilmente errori nel riconoscere la classe DRIVING.

4.2.2 Random Forest

Come per il KNN, anche per il Random Forest è stato effettuato un tuning degli iperparametri per ottenere una prestazione ottimale del modello. I parametri che sono stati modificati sono stati:

- **n_estimators**: che rappresenta il numero di alberi decisionali nella foresta
- **max_features**: che rappresenta il numero di features da considerare quando si cerca la migliore divisione
- **max_depth**: che rappresenta la profondità massima degli alberi decisionali
- **min_samples_split**: che rappresenta il numero minimo di campioni richiesti per dividere un nodo interno
- **min_samples_leaf**: che rappresenta il numero minimo di campioni richiesti per costruire una foglia dell'albero
- **bootstrap**: che rappresenta la modalità di campionamento dei dati

Il tuning degli iperparametri è stato effettuato utilizzando la funzione *RandomizedSearchCV* di *Scikit-learn*.

L'obiettivo di questa ottimizzazione era quello di migliorare la precisione del modello Random Forest e di fornire una soluzione ottimale per la classificazione dei dati. Il risultato ottenuto ha dimostrato che l'ottimizzazione degli iperparametri è stata efficace e ha contribuito a migliorare la performance del modello. La RandomizedSearchCV è stata eseguita per un numero di 100 iterazioni, usando una 5-fold cross validation. Per un totale di 500 fit. In questo caso è stata preferita una RandomizedSearchCV rispetto ad una GridSearchCV poiché il numero di combinazioni di iperparametri da testare è molto elevato e testarli tutti avrebbe richiesto un tempo di esecuzione molto elevato rispetto alla potenza di calcolo disponibile.

Risultati

I risultati del tuning degli iperparametri utilizzando il dataset precedentemente descritto sono riportati nella tabella seguente:

	Best value
n_estimators	800
max_depth	90
min_samples_split	5
min_samples_leaf	1
max_features	sqrt
bootstrap	False

Con questi parametri siamo riusciti ad ottenere un'accuratezza finale del 95.24% utilizzando il dataset su cui non è stato applicato il MinMaxScaling.

Facendo tuning sul modello Random Forest utilizzando il modello con il MinMaxScaling, si è ottenuto un'accuratezza del 89,95%.

I parametri ottenuti dal tuning sono:

	Best value
n_estimators	1400
max_depth	80
min_samples_split	10
min_samples_leaf	2
max_features	sqrt
bootstrap	True

Mostriamo adesso i grafici delle Matrici di confusione ottenute per i due modelli Random Forest, uno con il dataset su cui non è stato applicato il MinMaxScaling e l'altro con il dataset su cui è stato applicato il MinMaxScaling.

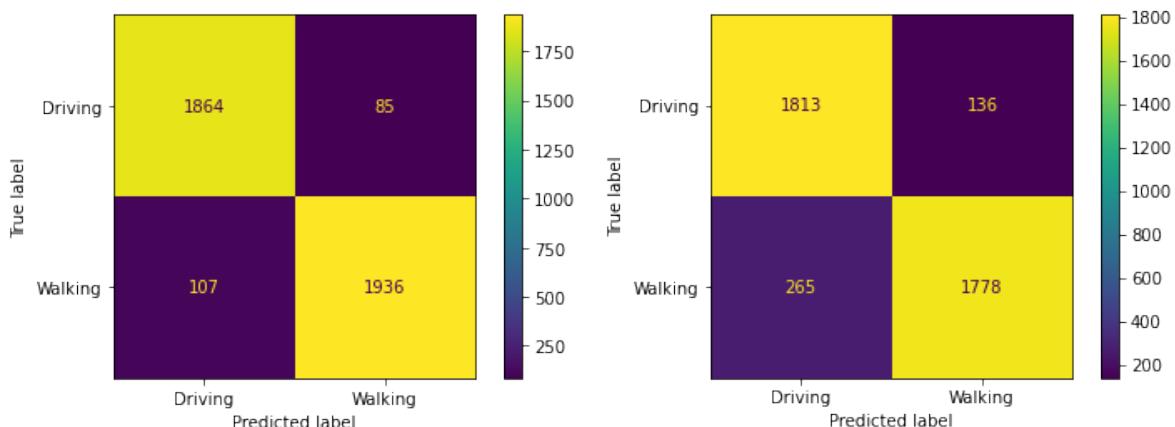


Figura 4.2: Matrice di confusione del modello senza scaling (sx) e con scaling (dx)

Queste matrici di confusione mostrano come il modello Random Forest sia ottimo per la classificazione di questo tipo dati. Sia il modello senza scaling che quello con scaling sono in grado di classificare correttamente la maggior parte dei dati, ma è possibile notare che riescano a riconoscere più facilmente l'etichetta WALKING, rispetto a quella DRIVING, nella quale è presente un errore maggiore.

4.2.3 Gaussian Naive Bayes

Per quanto riguarda il Gaussian Naive Bayes, abbiamo deciso di utilizzare questo algoritmo per la classificazione dei dati in quanto si tratta di un metodo semplice e veloce per la classificazione basato sull'assunzione che le feature siano distribuite secondo una distribuzione normale.

Per questo modello il tuning dei parametri è stato effettuato utilizzando la funzione *GridSearchCV* di *Scikit-learn* solo su un'unico parametro *var_smoothing*, che rappresenta la varianza della distribuzione normale.

	Unscaled	Scaled
<code>var_smoothing</code>	0.0012328467394420659	0.0003511191734215131

Nonostante ciò le accuracy sono risultate essere identiche per entrambi i modelli. Mostriamo quindi solo una delle due matrici di confusione ottenute, essendo esse identiche.



Figura 4.3: Matrice di confusione

Risultati

Il modello Gaussian Naive Bayes ha fornito un'accuratezza del 81.82%, che risulta essere un risultato inferiore rispetto ai modelli KNN e Random Forest.

4.3 Confronto dei risultati

Illustriamo di seguito i risultati completi dei tre modelli precedenti, ottenute utilizzando il dataset precedentemente descritto.

Evaluating algorithms

	KNN			RF			GNB			Support
	P	R	F1	P	R	F1	P	R	F1	
0	0.88	0.92	0.90	0.95	0.95	0.95	0.76	0.98	0.86	1949
1	0.92	0.87	0.90	0.95	0.95	0.95	0.98	0.71	0.82	2043
Accuracy	0.8987975951903807			0.9524048096192385			0.843436873747495			3992
Macro AVG	0.90	0.90	0.90	0.95	0.95	0.95	0.87	0.85	0.84	3992
Weighted AVG	0.90	0.90	0.90	0.95	0.95	0.95	0.87	0.84	0.84	3992

Tabella 4.1: P = Precision, R = Recall e F1 = F1-score

Utilizzando il dataset su cui è stato applicato il MinMaxScaling i risultati sono:

Evaluating algorithms

	KNN			RF			GNB			Support
	P	R	F1	P	R	F1	P	R	F1	
0	0.88	0.88	0.92	0.87	0.93	0.90	0.76	0.98	0.86	1949
1	0.92	0.88	0.90	0.93	0.87	0.90	0.98	0.71	0.82	2043
Accuracy	0.8997995991983968			0.8995490981963928			0.843436873747495			3992
Macro AVG	0.90	0.90	0.90	0.90	0.90	0.90	0.87	0.85	0.84	3992
Weighted AVG	0.90	0.90	0.90	0.90	0.90	0.90	0.87	0.84	0.84	3992

Tabella 4.2: P = Precision, R = Recall e F1 = F1-score

Prima di confrontare i tre modelli tra loro, confrontiamo le differenze nell'applicazione per ogni modello dello scaling dei valori. Per quanto riguarda il Gaussian Naive Bayes i risultati sono praticamente identici, mentre per il modello KNN c'è un miglioramento di circa un 0.1% dell'accuratezza. Per il modello Random Forest invece c'è un peggioramento dell'accuratezza di circa il 6%, un risultato molto significativo. Dati i risultati ottenuti abbiamo tralasciato quindi questa tecnica. Riguardo invece al confronto tra i

tre modelli, con ben un 6% di accuratezza in più il modello Random Forest è il modello migliore.

4.4 Accuratezza sistema Har integrato

Dati i precedenti risultati, abbiamo valutato l'accuratezza del sistema di HAR integrato nell'applicazione e offerto dalla libreria *flutter_activity_recognition*. Per farlo, data l'impossibilità di testare la libreria con il dataset utilizzato per testare i modelli precedentemente presentati, abbiamo raccolto un piccolo testbed tramite esperimenti reali. Abbiamo raccolto 1147 record, formati dai dati dei sensori e due colonne target: il target riconosciuto dall'utente (la ground truth) e il target riconosciuto dalla libreria. Proprio il confronto tra questi target ci ha permesso di valutare l'accuratezza del sistema. Il risultato ottenuto è un'accuratezza del **91,02%**. Dato che il dataset è stato raccolto da noi stessi e che abbiamo previsto l'apparizione di un toast sull'app mobile quando la libreria riconosceva un'attività, abbiamo potuto verificare sul campo quali fossero le situazioni che portavano in errore la libreria. Dobbiamo dire che sul riconoscimento di un'attività di WALKING o STILL la libreria si presenta solida. Qualche errore è stato commesso durante il riconoscimento di attività IN_VEHICLE, in caso di grandi frenate o curve particolari portavano la libreria per qualche secondo a riconoscere UNKNOWN. Nel grafico mostriamo un istogramma che mette a confronto le tuple per ogni classe target e il numero di predizioni corrette fatte. Come si può notare la maggior parte degli errori è stata commessa nella classe STILL, questo però è dipeso anche dal nostro metodo di raccolta dei dati, in cui raccoglievamo tuple ad intervalli regolari di tempo e non solo quando la libreria riconosceva una nuova attività.

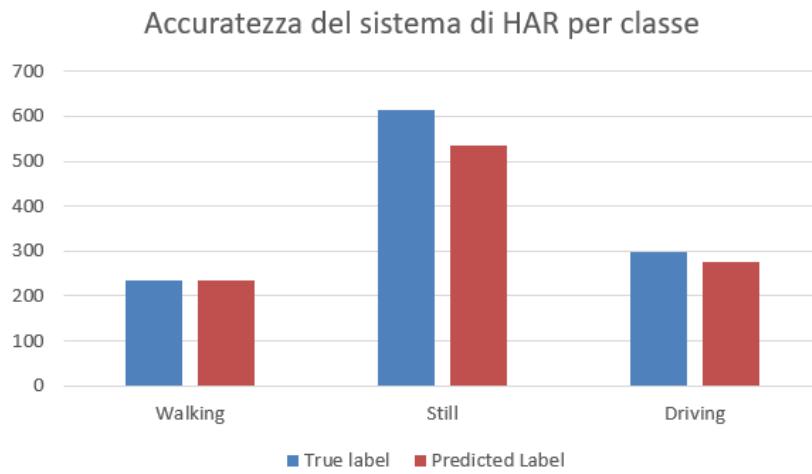


Figura 4.4: Istogramma che mette a confronto le tuple per ogni classe target e il numero di predizioni corrette fatte

4.5 Implementazione del Random Forest nell'app

Dati i risultati ottenuti nella sezione 4.2.2, comparati con i risultati ottenuti dagli altri due modelli, abbiamo scelto il Random Forest come algoritmo di classificazione da implementare direttamente nell'applicazione. Nonostante nella fase precedente è stato allenato un classificatore binario, implementando direttamente nell'applicazione questo tipo di classificatore fa sì che l'applicazione non sia in grado di riconoscere quando l'utente è fermo, riconoscendo comunque un'attività. Per motivi di completezza quindi, abbiamo deciso di trasformare il classificatore da binario a ternario riconoscendo WALKING, DRIVING e STILL. Data la poca disponibilità di dati raccolti da noi stessi, questo è stato allenato su una combinazione dei dati raccolti direttamente tramite l'applicazione e dei dati del dataset HAR, discusso nelle sezioni precedenti. Anche il successivo testing è stato fatto su una combinazione dei dati provenienti da queste due diverse fonti. Data la variazione del dataset, abbiamo deciso di effettuare nuovamente il tuning degli iperparametri del modello.

	Best value
n_estimators	1600
max_depth	None
min_samples_split	2
min_samples_leaf	1
max_features	log2
bootstrap	False

Nella tabella precedente possiamo vedere i migliori valori trovati per ogni iperparametro, mentre nella successiva mostriamo i risultati ottenuti dalla fase di testing del nostro modello. La tabella presenterà i risultati ottenuti sul test set del dataset HAR ottenuto dal prof e sul test set dei dati reali raccolti direttamente da noi tramite l'applicazione. Ovviamente questo test set si presenta molto piccolo data la nostra impossibilità a raccogliere una grande quantità di dati, quindi la valenza di questi risultati è molto limitata.

Random Forest								
	Dataset HAR				Dataset Real			
	P	R	F1	Support	P	R	F1	Support
0	0.90	0.91	0.90	982	0.70	0.81	0.75	26
1	0.93	0.93	0.93	1004	0.81	0.96	0.88	23
2	0.94	0.93	0.94	1125	0.93	0.82	0.87	66
Accuracy	0.9247830279652844			3111	0.8434782608695652			115
Macro AVG	0.92	0.92	0.92	3111	0.82	0.86	0.83	115
Weighted AVG	0.92	0.92	0.92	3111	0.86	0.84	0.85	115

Tabella 4.3: P = Precision, R = Recall e F1 = F1-score

Mostriamo ora per entrambi questi test set le matrici di confusione:

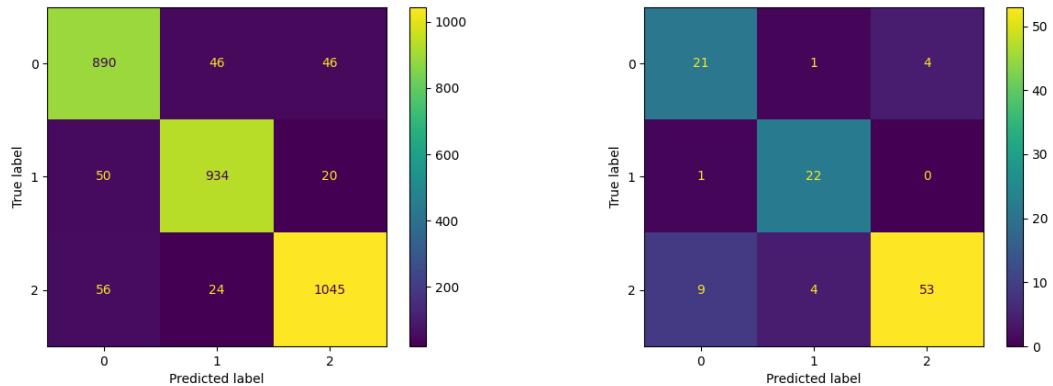


Figura 4.5: Matrice di confusione per il test set del dataset HAR e per il dataset dei dati raccolti da noi

Capitolo 5

Conclusioni

Lo scopo di questo progetto era quello di costruire un'applicazione di crowdsensing dei parcheggi per la città di Bologna. Il progetto prevedeva l'implementazione di un'applicazione mobile, un backend ed un frontend. Lo scopo principale del progetto era quello di sperimentare con delle tecniche location-aware e activity-aware andando a lavorare con la posizione dell'utente (per il riconoscimento della zona di parcheggio) e con le attività svolte dall'utente (per la gestione del riconoscimento di entrate e uscite dal parcheggio). Per rispettare queste proprietà oltre alle specifiche abbiamo deciso di estendere il progetto implementando un sistema di riconoscimento dell'utilizzo delle colonnine di ricarica elettrica e di implementare come seconda feature aggiuntiva l'algoritmo di clustering DBSCAN, oltre a quello K-Means. Il progetto è stato molto interessante e ci ha dato la possibilità di sperimentare nuove tecnologie e di approfondire le conoscenze acquisite durante il corso.

Bibliografia

- [1] flutter activity recognition package. https://pub.dev/packages/flutter_activity_recognition.
- [2] flutter map package. https://pub.dev/packages/flutter_map.
- [3] Flutter official page. <https://flutter.dev/>.
- [4] React leaflet official page. <https://react-leaflet.js.org/>.
- [5] Node.js official page. <https://nodejs.org/en/>.
- [6] React official page. <https://it.reactjs.org/>.