SAPIENZA
UNIVERSITÀ DI ROMA

Classificazione dei tipi di parcheggio su app mobile

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Tirocinio Formativo Attivo

Classe Corso di laurea in Informatica

Candidato

Alessio Luciani
Matricola 1797637

Relatore

Emanuele Panizzi

Tutor del Tirocinante

Tutor Coordinatore

Anno Accademico 2019/2020

Tirocinio svolto presso:

Università degli Studi di Roma "La Sapienza"
Piazzale Aldo Moro 5, 00185 Roma
https://www.sapienzaapps.it/
Dirigente scolastico:

Tesi non ancora discussa

Classificazione dei tipi di parcheggio su app mobile
TFA. Relazione di tirocinio. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: alessio99.luciani@gmail.com

Alla mia famiglia

Indice

Capitolo 1

Introduzione

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Capitolo 2

Raccolta dati dei parcheggi

2.1 Procedura generale

2.1.1 Raccolta

Al fine di poter creare un classificatore per i tipi di parcheggio, la prima cosa che è risultata necessaria è stata una raccolta di dati di natura time-series. In particolare, questi dati dovevano essere di buona qualità, generati in maniera controllata e avere una chiara classificazione che gli permetta di essere utilizzabili per un processo di training. Per questo motivo è stato importante che la raccolta fosse portata avanti da poche persone fidate che fossero in grado di eseguire una serie di azioni commettendo il minor numero di errori possibile.
Dal momento che il modello classificatore in questione è destinato ad essere utilizzato su applicativi mobile, e più in particolare sull'app GeneroCity, i dati che quest'ultimo riceve come input devono provenire da sensori che si trovano direttamente sullo smartphone. Così facendo, la modalità più ovvia di raccolta dei dati risulta essere esattamente quella di sfruttare gli stessi sensori dello smartphone.
Coloro che hanno avuto il compito di raccogliere i dati, erano disposti di un sistema di raccoglimento installato all'interno dell'app GeneroCity. Inizialmente, questo era presente solo nella versione iOS dell'app, ma successivamente, è stato fatto il porting anche sulla versione Android. Questa operazione ha reso possibile che il numero di persone disponibili per la raccolta di dati aumentasse significativamente.
Sfruttando alcuni eventi generati automaticamente dall'app, come ad esempio l'ingresso e l'uscita dall'auto, è stato possibile avviare e interrompere la raccolta, senza troppa pressione o attenzione dell'utente. Quindi l'interazione da parte dell'utente, durante il percorso in auto, è stata minima, se non inesistente. Il processo prevedeva soltanto che l'utente entrasse in macchina, facesse il suo viaggio e infine uscisse. Nel frattempo l'app si occupava di raccogliere i dati e caricarli su un database in automatico. In questo modo, non solo le persone addette alla raccolta non hanno dovuto avere troppe accortezze, ma inoltre hanno potuto integrare la raccolta con le loro abitudini quotidiane, senza dover dedicare tempo e sforzi extra solemente per questo scopo. Infatti, qualvolta

essi hanno utilizzato l'auto nella loro vita quotidiana, hanno potuto aggiungere una, o più registrazioni al database.
Anche per quanto riguarda la classificazione del tipo di parcheggio, si è cercato un approccio che semplificasse l'operazione a chi la stava eseguendo. La modalità che è sembrata meno invasiva è stata quella di inviare una notifica a colui che avesse appena effettuato un parcheggio, chiedendo di cliccare su un pulsante che classificasse il tipo di parcheggio, distinguendo qualche opzione. Questa informazione veniva poi salvata insieme alla registrazione dei sensori, all'interno del database. Si può notare che, anche in questo caso, l'interazione dell'utente è stata minima. Infatti, la notifica veniva mostrata ad esso in maniera automatica, dopo qualche secondo dall'uscita dalla macchina e chiaramente l'utente stesso aveva la possibilità di effettuare la scelta del tipo di parcheggio in un secondo momento.
L'importanza di affidare questa responsabilità ad utenti fidati è dovuta principalmente al fatto che selezionare il corretto tipo di parcheggio risulta un operazione cruciale al fine di ottenere un modello accurato, che sia in grado di effettuare una distinzione chiara tra le diverse tipologie di parcheggio. Un'utente qualsiasi, invece, potrebbe selezionare un'etichetta errata per svariati motivi, come un'idea confusa riguardo le diverse tipologie di parcheggi, oppure una scarsa volontà di collaborazione che potrebbe indurlo a selezionare un tipo randomico. Si può ben intuire che la selezione di un tipo randomico, tra le varie opzioni proposte, da' un contributo deleterio e quindi peggiore al caso in cui l'utente non rispondesse proprio alla notifica inviata e quindi non selezionasse alcun tipo per uno specifico parcheggio. Tuttavia, anche quando ad effettuare l'operazione vi è una persona che ha ben chiaro come comportarsi, è possibile che degli errori vengano commessi. Infatti, in alcune situazioni possono sollevarsi diversi dubbi o indecisioni. Potrebbe accadere che un parcheggio abbia una disposizione inusuale, diversa dalle più comuni e quindi particolarmente complicata da individuare. Oppure, è possibile che un parcheggio venga effettuato con una manovra molto diversa dalle più frequenti, per motivi che possono essere dovuti alla condizione del traffico, alla disposizione di altri veicoli circostanti, allo stile di guida o all'urgenza del guidatore, ecc. A causa di questi motivi, il dataset ottenuto non può essere considerato privo di difetti, ma si è cercato, attraverso queste accortezze, di ottenere una qualità dei dati migliore possibile.

2.1.2 Pulitura e processamento

Benchè i dati raccolti si trovassero in buono stato e strutturati in maniera adeguata per essere utilizzati con lo scopo di effettuare il training per un modello ML che classificasse i tipi di parcheggio, essi non potevano essere considerati "puliti" e pronti all'utilizzo. Tra le diverse cose, essi contenevano informazioni aggiuntive e fuorvianti che avrebbero peggiorato le performance del classificatore. Un esempio è composto da tutti i dati raccolti dal momento in cui il parcheggio viene terminato, fino a quando l'app non termina la raccolta, dopo che l'utente è uscito dall'auto.

Addizionalmente alla pulitura iniziale, i dati vanno incontro anche ad un intensa sequenza di operazioni che cercano di esaltare e isolare le feature più significative che possono essere analizzate dal classificatore. Una delle operazioni che può essere presa come esempio consiste nelle rotazioni 3D che vengono applicate ai dati degli accelerometri e dei giroscopi per fare in modo che questi ultimi risultino come se fossero stati raccolti con lo smartphone orientato sempre allo stesso modo rispetto all'auto.
Dunque, è stato realizzato uno script in grado di scaricare tutti le registrazioni di dati presenti nel database e applicarvi queste operazioni per ottenere il risultato finale. Al termine di ciò i dati sono stati organizzati in un formato accettato in input dal modello classificatore.

2.2 Sensori utilizzati

Come già anticipato, la procedura di raccolta dei dati è avvenuta all'interno dell' app GeneroCity, su entrambi i sistemi operativi iOS e Android. L'idea di base è stata quella di utilizzare un oggetto raccoglitore che venisse richiamato in maniera asincrona, all'interno di un thread dedicato. Il fatto di utilizzare un thread a parte ha reso possibile un campionamento con cadenza fissa che non creasse interruzioni al resto dell'app.
Sono stati scelti alcuni sensori, i quali dati sarebbero stati utili per l'estrazione di feature da inviare come input al classificatore. Questi sensori sono principalmente:

- bussola
- accelerometri
- giroscopi
- GPS per la velocità

Con una certa cadenza, il raccoglitore registra i vari valori e li indicizza attraverso il timestamp dello specifico istante. Questa indicizzazione fa in modo che i dati possano avere l'informazione sulla sequenza temporale delle varie registrazioni e quindi possono essere trattati come dati di tipo time-series.

2.2.1 Implementazione su iOS

La prima implementazione del raccoglitore è stata fatta per l'ambiente iOS.
É stato definita una classe *ParkTypeSampleCollector*, da cui poter creare istanze di raccoglitori. All'interno dell'app, un raccoglitore viene attribuito ad un'auto *Car*, in quanto ha lo scopo di raccogliere i dati durante i tragitti percorsi da quella determinata macchina.
Al *ParkTypeSampleCollector* è stata impostata una cadenza costante di 0.1 secondi, in modo da avere un tasso di campionamento abbastanza elevato, in grado di distinguere piccole variazioni in campioni consecutivi.

Per i singoli campioni è stata definita una struttura *Sample*, contenente una serie di valori:

- *heading*: consiste nell'angolo tra la punta dello smartphone è il nord geografico, rappresentato in gradi.

- *acceleration*: contiene i tre componenti dell'accelerazione ottenuti attraverso gli accelerometri e processati direttamente dalla libreria *CoreMotion*. Questo significa che i tre componenti relativi ai tre assi non contengono più l'informazione sulla gravità, questa è stata sottratta automaticamente.

- *rotationRate*: similmente, contiene i tre componenti estratti dai giroscopi e adeguatamente processati.

- *speed*: consiste nell'attuale velocità calcolata dal dispositivo ed ottenuta grazie alla libreria *CoreLocation*.

Il *ParkTypeSampleCollector* è dotato di un buffer di *Sample* di dimensione fissa. Questo buffer è utilizzato per salvare la sequenza di campioni che vengono raccolti durante il tragitto. In quanto l'obiettivo finale è quello di utilizzare i dati per classificare il tipo di un parcheggio, è necessario salvare soltanto i campioni che sono stati raccolti in momenti temporalmente vicini al termine del parcheggio. Chiaramente non si può sapere a priori quanto tempo impiegherà l'utente a parcheggiare, così la raccolta viene avviata alla partenza dell'auto. Nonostante questo, per evitare di occupare grandi quantità di memoria con dati che alla fine verranno eliminati, si è pensato di aggiungere campioni al buffer, fino al suo riempimento, e successivamente aggiungere un potenziale nuovo campione rimpiazzando il meno recente presente nel buffer. Questa operazione è stata implementata in tempo costante, grazie ad un indice utilizzato in aggiunta. Dal momento che non vengono salvati i timestamp "reali" relativi ai singoli campioni, dei timestamp verranno calcolati dinamicamente al termine utilizzando gli indici del buffer e l'intervallo di campionamento. La dimensione che è stata scelta per il buffer è di 2048 elementi, che si traducono in una registrazione finale che coprirà un intervallo di tempo che dura al massimo qualche minuto.
Il raccoglitore viene azionato ogni volta che passa la quantità di tempo relativo all' intervallo di campionamento. Ovvero, all'avvio di esso, ha inizio un ciclo che termina solamente quando l'auto è stata parcheggiata. Ad ogni iterazione viene aggiunto un nuovo campione al buffer e poi viene effettuata un'attesa, prima che si passi all'iterazione successiva. L'intero ciclo viene eseguito asincronamente utilizzando la chiamata *DispatchQueue.global().async*, fornita da iOS. Ciò che fa questa funzione è accodare una funzione, fornita dallo sviluppatore, alla coda di esecuzione globale del sistema operativo, in maniera asincrona.
L'avvio e la terminazione del raccoglitore vengono eseguiti rispettivamente nei metodi *unpark()* e *park()* dell'istanza di *Car*. Quindi, quando l'auto esce dal parcheggio all'inizio del tragitto e quando parcheggia al termine di esso.

2.3 Caricamento dati nel database

Al termine di ogni registrazione, i dati raccolti devono essere caricati nel database. Come formato di salvataggio delle registrazioni è stato scelto JSON. Quindi, non appena il raccoglitore termina la raccolta, il buffer ottenuto deve essere convertito in un oggetto JSON. In particolare, questo avviene ponendo, come chiavi dell'oggetto, i timestamp calcolati rispetto al primo elemento. Ad ogni chiave viene associato un'array di valori, che contiene tutti i dati ottenuti dal campione di quello specifico istante. Così l'array è composto da:

- *heading*
- I tre componenti singoli ottenuti da *acceleration*
- I tre componenti singoli ottenuti da *rotationRate*
- *speed*

Siccome GeneroCity già effettuava una chiamata all'API del backend per registrare dei dati relativi al parcheggio che è stato appena effettuato, è stato deciso di sfruttare quest'ultima anche per caricare l'oggetto JSON che contiene i campioni. Così è stato creato un nuovo campo nel database, relativo al parcheggio e chiamato *parksamples*. All'interno di questo campo viene salvata la stringa serializzata, ottenuta dall'oggetto JSON.

2.4 Preparazione dei dati

Dal momento che i dati sono stati raccolti quotidianamente, il dataset delle registrazioni si è andato a popolare di giorno in giorno. In questo modo, è stato possibile effettuare il training del modello classificatore di tanto in tanto, al fine di migliorarne le performance sempre di più, avendo a disposizione più registrazioni. Per automatizzare il processo di preparazione dei dati, è stato creato uno script, composto da moduli, in Python. Come precedentemente anticipato, la funzione principale dello script è stata quella di scaricare, pulire e processare i dati ottenuti. Infatti, ad ogni invocazione esso

1. richiede al database e scarica tutte le registrazioni che ancora non sono state salvate in locale;
2. effettua tutte le procedure di pulitura e processamento dei dati;
3. prepara i dati per essere utilizzati come input del classificatore e crea una struttura di file CSV, con directory classificate in base all'etichetta scelta dall'utente per la registrazione;

Inoltre, insieme allo script è presente un modulo che ha lo scopo di effettuare il plotting dei dati. Questo plotter è utilizzato per analizzare i dati in maniera visiva ed è in grado di mostrare i valori uscenti dai vari sensori in diverse modalità e quindi evidenziando diversi aspetti di essi. Ad esempio, è possibile

tracciare valori provenienti da un componente dell'accelerazione nel tempo, sotto forma di una curva 2D, ma anche i punti (formati da tutti i tre componenti) dell'accelerazione stessa, rappresentati in uno spazio 3D.

2.5 Modelli ML utilizzati

Avendo i dati pronti e nel giusto formato, il passo successivo è quello di effettuare il training di un modello classificatore.
Per creare i modelli è stato deciso di utilizzare la libreria *CreateML* di Apple. Questa libreria da modo di realizzare nuovi modelli di machine learning con una prospettiva ad alto livello, permettendo l'utilizzo anche ad utenti che non sono dotati di una conoscenza molto approfondita a riguardo. Essa offre anche un'interfaccia grafica che rende la creazione ancora più intuitiva.
Al fine di ottenere un classificatore per tipi di parcheggio, sono stati utilizzati due approcci diversi, a partire da due template differenti di *CreateML*:

- una rete neurale convoluzionale, utilizzando il *Motion Activity Classifier*;
- un tree ensemble, utilizzando il *Tabular Classifier*

Il motivo per cui sono stati creati due modelli differenti è che il dataset che è stato possibile ottenere con un numero molto limitato di utenti disponibili per la raccolta di dati, non è composto da un numero di registrazioni sufficentemente grande. Per questa motivazione, alcuni tipi di modelli hanno ottenuto scarsi risultati nella classificazione e si è preferito tentare in diversi modi.

Capitolo 3

Contributo dell'utente

Come è stato già discusso, nell'ambito della raccolta dei dati il contributo dell'utente è essenziale. Infatti, è bene che quest'ultimo sia veritiero e che esso venga fornito per il maggior numero di registrazioni possibile. Per agevolare l'utente nel dare il contributo è stato necessario creare un'interazione semplice e che non richiedesse procedure troppo sofisticate.
Dato che l'informazione che deve essere fornita dall'utente è una etichetta che classifichi il tipo di parcheggio appena effettuato, è stato deciso di mostrare una notifica di sistema che proponesse questa scelta. Questa notifica viene chiaramente manifestata dal sistema sotto forma di notifica locale di GeneroCity e da modo di selezionare il tipo di parcheggio effettuato.

3.1 Notifica mostrata

Su piattaforma iOS, è stata scelta la notifica di tipo *Actionable Notification*. Le notifiche di questa tipologia permettono di aggiungere, oltre a del testo, un gruppo di pulsanti, corrispondenti a delle azioni. Questa è risultata una soluzione ottima per il nostro scopo. É stato potuto aggiungere del testo che chiedesse "In che modo hai parcheggiato?" e allegare dei pulsanti con le opzioni "A spina", "Parallelo", "A pettine". In questo modo, all'utente resta solo da rispondere alla notifica selezionando una opzione tra quelle disponibili.

3.1.1 Registrazione della notifica

Per creare la nuova notifica, la prima cosa che è stato necessario fare è stata registrarla nel sistema, attraverso l'app. Dato che si tratta di una *Actionable Notification*, è stata registrata una *UNNotificationCategory* e delle *UNNotificationAction* legate ad essa. Queste classi provengono tutte dalla libreria *UserNotifications*. La categoria è necessaria per distinguere una *Actionable Notification* di un tipo da una di un altro. Quindi, è stata creata una nuova categoria chiamata *PARKTYPE*, utilizzata esclusivamente per questa notifica. A questa categoria sono state legate le varie azioni, che corrispondono ai tipi di parcheggio. Anche esse sono dotate di un identificatore che è utilizzato

al momento della ricezione, per distinguerle tra tutte le azioni che l'app può ricevere.

3.1.2 Esposizione della notifica

Ogni volta che una nuova registrazione termina, viene creata e mostrata una nuova notifica con categoria *PARKTYPE*. Per fare ciò, viene creato un *UNTimeIntervalNotificationTrigger* con un ritardo di 5 secondi. Questo significa che la notifica verrà effettivamente mostrata all'utente 5 secondi dopo che GeneroCity abbia rilevato il parcheggio effettuato dall'utente. In questo modo, esso può rispondere alla notifica quando effettivamente è a conoscenza del tipo di parcheggio appena fatto e inoltre si è sicuri che non sia più alla guida e quindi non corra rischi perdendo l' attenzione.

3.2 Etichetta selezionata dall'utente

Nel momento in cui l'utente riceve la notifica, esso può rispondere scegliendo un'opzione, oppure ignorarla completamente. Nel caso in cui la notifica viene ignorata, semplicemente non accade nulla, ovvero, l'etichetta del tipo di parcheggio non viene caricata sul database. Invece, in caso contrario, è necessario aggiornare il valore nel database per legare l'etichetta appena ottenuta attraverso l'utente, al relativo parcheggio.

3.2.1 Ricezione della risposta

Attraverso il *UNUserNotificationCenterDelegate*, si possono registrare callback da eseguire in seguito ad alcuni eventi generati dal sistema, che riguardano le notifiche. Uno di questi eventi è la ricezione di una risposta. Questa callback viene invocata quando l'utente clicca su un pulsante azione di una qualsiasi notifica legata all'app. Questo significa che è compito dello sviluppatore individuare quale azione è stata eseguita dall'utente, all'interno della callback. Ciò viene fatto attraverso gli identificatori delle *UNNotificationAction*. Una volta che è stata identificata l'azione scelta, è possibile capire qual'è il tipo di parcheggio effettuato dall'utente e salvarlo all'interno dell'oggetto *Car*. Per terminare, viene effettuata una chiamata all' API del backend, che aggiorni i dati del parcheggio relativo all'auto corrente nel database.

Capitolo 4

Deploy dei modelli ML

Una volta che è stato effettuato correttamente il training e la creazione di un modello machine learning, arriva il momento di distribuirlo e di utilizzarlo effettivamente per il suo scopo, ovvero quello di classificatore di tipi di parcheggio.
L'idea generale del funzionamento consiste nel raccogliere dati durante il tragitto dell'auto e in seguito fornirli come input al modello. Chiaramente, i dati che il modello prodotto riceve in input devono essere della stessa forma di quelli utilizzati per fare il training. Ciò implica che tutte le procedure di pulizia dei dati e di miglioramento delle feature debbano essere ripetute ogni volta che nuovi dati vengono raccolti per essere sottoposti al modello ed essere classificati.
Garantire che un modello riceva in input dati di forma e qualità identiche a quelle del dataset di training può talvolta risultare laborioso. Ciò è causato dal fatto che gli ambienti di raccolta dati e quelli di produzione possono differire sotto svariati aspetti. Infatti, non è scontato che in queste due situazioni si utilizzino lo stesso linguaggio di programmazione, lo stesso sistema operativo, le stesse librerie, gli stessi approcci, ecc. Al contrario, è molto probabile che un modello venga istruito una volta, per poi essere distribuito ed utilizzato su piattaforme diverse. In queste situazioni si può andare incontro a numerose complicazioni, come dover ri-progettare alcuni algoritmi a causa di un cambiamento di linguaggio di programmazione, che renderebbe l'algoritmo stesso meno efficente o addirittura non più funzionante. In questo caso, è fondamentale avere una conoscenza approfondita del funzionamento dell'algoritmo originale e della buona documentazione da poter consultare. Qualsiasi minimo errore di trascrizione o di comprensione potrebbe generare modifiche sostanziali ai dati che verranno processati, rendendoli così differenti da quelli provenienti dal training set e non più validi per una potenziale classificazione. Spesso si può anche andare incontro ad un cambiamento di librerie, soprattutto se si sta cambiando linguaggio di programmazione. La questione che riguarda le librerie è ancora più delicata rispetto a quella dei linguaggi. Nella maggior parte dei casi, esse contengono delle logiche interne che risultano molto complesse da riprodurre o imitare in un ambiente differente. Per questo motivo, è bene ridur-

re al minimo l'utilizzo di librerie esterne quando ci si trova in situazioni come questa, ovvero scrivendo codice che dovrà essere portato su altre piattaforme. Tuttavia, a volte è necessario utilizzare alcune librerie, come nel nostro caso, in cui si sono dovute utilizzare libreria per l'algebra lineare. Quindi, anche per questo discorso occorre avere massima prudenza nella scelta di un sostituto valido e che non generi inconsistenze nei dati.

4.1 Necessità del calcolo in locale

Per quanto riguarda il deploy di un modello, un'altra scelta da prendere è se eseguire il modello in un applicativo lato client, oppure su un server remoto. Questa decisione dipende da molti fattori, tra i quali vi è la potenza di calcolo del dispositivo che esegue l'applicativo client, potenziali requisiti di esecuzione real-time (con bassa latenza), disponibilità del dispositivo di una connessione ad internet, sensibilità dei dati, ecc.
Entrambi gli approcci hanno vantaggi e svantaggi. Tra i vantaggi di una esecuzione lato server possiamo notare:

- la possibilità di sostituire il modello con una versione più aggiornata senza dover per forza rilasciare un nuovo aggiornamento dell'applicativo client
- poter effettuare l'esecuzione di un modello al posto di un dispositivo con scarsa potenza di calcolo (es. un dispositivo embedded)

Invece tra quelli di una esecuzione lato client:

- non dover dipendere da un server remoto, quindi da una connessione stabile
- evitare di dover attendere una risposta dal server, che può impiegare molto più tempo che una esecuzione in locale (nel caso in cui la potenza di calcolo del dispositivo sia sufficente)
- mantenere privati i dati che sono utilizzati come input del modello (nel caso in cui i dati siano sensibili)

Nel caso del nostro modello, è stato deciso di eseguire la predizione del modello in locale, nel lato client dell'app GeneroCity. Questa scelta è stata presa principalmente per due motivi:

- in prospettiva di un utilizzo dell'app da parte di numerosi utenti, evitare che il server backend venga sottoposto a un carico troppo oneroso, causato dalla potenziale esecuzione di un numero molto elevato di classificazioni in parallelo;
- sfruttare il *Neural Engine* dei dispositivi Apple, attraverso la libreria *CoreML* (solamente per la versione iOS di GeneroCity);

Dal momento che il processo di classificazione, attraverso un modello, è abbastanza intensivo e inoltre richiede la pulitura dei dati preliminare, permettere che venga eseguito su un server potrebbe generare rallentamenti. Questo si verifica soprattutto se il programma backend non è ditribuito. Oltre a ciò, vi è il fatto che l'esecuzione del modello avviene in seguito ad un'azione dell'utente (ovvero al parcheggio dell'auto) e nel caso in cui ci fossero degli utenti malintenzionati, si correrebbe il rischio di ricevere troppe richieste allo stesso tempo.

4.1.1 Predizione in locale con CoreML

Dovendo effettuare la classificazione in locale, su piattaforma iOS, la libreria che è risultata più adatta ai nostri scopi è stata *CoreML*. Questa libreria da la possibilità di integrare modelli machine learning direttamente all'interno delle app. Oltre a fornire dei modelli già addestrati e pronti all'uso (tra cui uno per il riconoscimento di immagini, un altro per la trascrizione audio-testo, ecc.), essa offre dei modi per utilizzare modelli creati direttamente dallo sviluppatore. Innanzitutto, *CoreML* richiede che i modelli proposti dallo sviluppatore siano del formato proprietario ".mlmodel". Questo formato, infatti, è stato creato proprio da Apple. Ci sono principalmente due modi per ottenere un modello di questa tipologia:

- crearlo direttamente attraverso la libreria *CreateML*;
- convertirne uno di un altro formato (es. TensorFlow) attraverso un apposito strumento di conversione;

Siccome per iniziare è stato integrato il modello soltanto nella versione iOS dell'app, è stata scelta la prima strada, ovvero generarlo con *CreateML*. In questo modo non è stato necessario utilizzare strumenti più personalizzabili, ma molto più complessi e a basso livello, come TensorFlow.
La scelta di *CoreML* è stata ovvia, in quanto su iOS è l'unica libreria che permette di sfruttare al meglio l'hardware del dispositivo. Infatti, essa è stata progettata proprio per essere utilizzata in accoppiata con l'hardware GPU presente negli stessi dispositivi Apple. Inoltre, nei dispositivi delle generazioni più recenti è stato inserito un componente dedicato esclusivamente all'esecuzione di algoritmi di intelligenza artificiale e modelli ML, chiamato *Neural Engine*. Esso si tratta di una sezione della GPU che è provvista di una architettura ad hoc e che viene sfruttata quando vengono invocate delle predizioni attraverso *CoreML*. Questa sezione è stata aggiunta a causa di una presenza del machine learning all'interno delle app che è aumentata drasticamente negli ultimi anni e che con ogni probabilità continuerà a diffondersi. Grazie a questo componente, l'esecuzione di una predizione di machine learning (es. la classificazione del tipo di parcheggio) non va ad influire sulle performance del sistema (sul resto dell'hardware) e quindi non crea rallentamenti o problemi visibili dall'utente.
Avendo deciso di eseguire il modello in locale, diventa obbligatorio effettuare anche le fasi preliminari di preparazione dei dati in locale. Quindi, l'intera

procedura, che va dal parcheggio dell'auto fino alla predizione del tipo di parcheggio, viene eseguita all'interno dell'app iOS, senza alcun sostegno di servizi esterni.

4.2 Porting del processore di dati

Come già anticipato, la scelta di eseguire la classificazione del tipo di parcheggio in locale nell'applicazione implica delle conseguenze. Una delle più importanti tra queste è certamente il fatto di dover pulire e preparare i dati direttamente in ambiente iOS. Ciò ha reso obbligato un porting dello script dedicato a processare i dati delle registrazioni, che è scritto in Python. Il linguaggio di destinazione del porting è Swift, ovvero il linguaggio utilizzato per sviluppo di GeneroCity iOS.

4.2.1 Da Python a Swift

Python e Swift differiscono sotto molti aspetti: oltre ad avere una sintassi significativamente diversa, sono stati progettati per scopi abbastanza differenti e i loro ambienti di esecuzione non hanno molti fattori in comune. Mentre Python è pensato per essere utilizzato su una vasta varietà di sistemi operativi e architetture, offrendo un interprete con ampia compatibilità, Swift ha una target più ristretto, che mira principalmente ai dispositivi con sistemi operativi Apple. Tuttavia, la differenza maggiore sta nel fatto che Python si tratta di un linguaggio interpretato, mentre Swift è un linguaggio compilato. Il secondo, quindi, ha bisogno di un compilatore apposito che generi i binari che infine vengono eseguiti.
Essendo due linguaggi molto utilizzati e con un background notevole, entrambi sono dotati di un solida fornitura di librerie. Infatti, considerando sia librerie native, che librerie di terze parti, Python e Swift possono essere sfruttati in moltissimi ambiti. Per quanto riguarda il nostro obiettivo, è stato necessario utilizzare operazioni di algebra lineare e fortunatamente entrambi i linguaggi sono muniti di librerie mature abbastanza da permettere di svolgere tali operazioni in maniera sufficentemente intuitiva.
Ad ogni modo, la trascrizione del codice stesso non è stata affatto priva di complicazioni. Sono state notate numerose differenze tra i linguaggi, che hanno reso il porting una operazione tutt'altro che lineare:

- **strutture sintattiche**: i due linguaggi differiscono alla base, nel modo di gestire gli scope dei blocchi di codice (python utilizza l'indentazione, mentre Swift utilizza uno stile a parentesi graffe C-like) e in molte strutture, come le dichiarazioni di cicli, esecuzioni condizionali, operatori booleani, ecc.

- **presenza di variabili e costanti**: Swift necessita che venga specificata l'entità di una variabile (se si tratta di una variabile o di una costante) al momento della sua dichiarazione, mentre in Python sono già tutte

semplici variabili di default. Inoltre Swift permette di aggiungere molte altre keyword (es. modificatori di accesso) alle dichiarazioni di variabili, in modo modificare le loro proprietà.

- **tipizzazione**: mentre Python appartiene al gruppo di linguaggi tipizzati dinamicamente (ovvero, che permettono ad una stessa variabile di assumere tipi diversi all'interno dello stesso scope di esecuzione), Swift è tipizzato staticamente (cioè, non permette che una variabile cambi il proprio tipo, nello stesso scope, durante una esecuzione). Questo implica anche che in molti contesti, in Swift, bisogni esplicitamente definire il tipo richiesto (ad esempio nella dichiarazione dei parametri e del valore di ritorno di una funzione), a differenza di Python, in cui il tipo non viene mai definito.

- **strutture dati**: non è immediato riportare molte strutture ad alto livello che si trovano su Python (es. la *List*) su un linguaggio come Swift. Spesso è necessario avere delle accortezze maggiori per gestire dei dati ad un livello leggermente più basso rispetto a come venivano gestiti su Python. Inoltre, il formato JSON che nel nostro script è stato ampiamente utilizzato, viene mappato direttamente a strutture dati Python, mentre richiede delle conversioni più mirate all'interno di Swift.

- **passaggi di argomenti a funzioni per riferimento e per valore**: i due linguaggi differiscono in alcuni casi nel passaggio di parametri a funzioni. Infatti, per strutture come gli array, Python non crea copie al momento del passaggio di un'istanza come argomento ad una funzione. Quindi, all'interno dello scope della funzione si ha accesso al riferimento della struttura iniziale e ogni modifica applicata ad essa rimarrà persistente anche quando l'esecuzione della funzione sarà terminata. Al contrario, questo tipo di strutture sono passate in Swift come valore, ovvero, vengono create delle copie al momento della chiamata di funzione. Questo significa che tutte le azioni effettuate su una determinata struttura di questo tipo, dopo essere stata copiata, non andranno a coinvolgere la copia originale. Inoltre, di default, Swift istanzia le variabili passate come argomento di funzione come "let", ovvero costanti, e quindi non permette modifiche.

- **gestione delle variabili opzionali**: mentre Python utilizza un approccio più classico a riguardo, Swift fornisce un sistema abbastanza sofisticato per la gestione delle variabili opzionali, al fine di tutelare il programmatore nel ridurre il numero degli errori dovuti alla presenza di valori nulli all'interno delle variabili. Di fatto, a causa della natura di linguaggio interpretato e della tipizzazione dinamica, Python non segnala potenziali problemi scaturiti da chiamate di funzioni su una variabile nulla e quindi semplicemente genera una eccezione a tempo di esecuzione. A sua differenza, Swift introduce il concetto di tipo opzionale. Un tipo opzionale (sintatticamente con lo stesso nome del corrispettivo tipo

normale, ma seguito da "?") consiste in un tipo le cui variabili possono contenere un valore nullo. Questo implica che i tipi non opzionali di Swift non possono assolutamente trattare valori nulli. Dunque, esistono alcuni operatori che vengono utilizzati per assicurare che una variabile opzionale risulti effettivamente in un valore non nullo. Alcuni di questi sono: "!" che consiste in un'asserzione (ovvero, garantisce che il contenuto della variabile sia non nullo, a costo di generare un'eccezione se questo non è vero), "?? default" che permette di fornire un valore di default nel caso in cui la variabile contenga un valore nullo, ecc. Utilizzare "!" è stato definito come "unwrap", ovvero l'azione di "scartare" metaforicamente una variabile, al fine di ottenere il vero valore che è "incartato" al suo interno. Potenzialmente, si potrebbe applicare l'unwrap in ogni caso, per ottenere un comportamento simile a quello di Python, ma ciò andrebbe contro le linee guida del linguaggio.
I tipi opzionali si possono incontrare spesso all'interno di Swift, come quando si utilizzano dizionari, che potrebbero avere o non avere un valore corrispondente ad una chiave. Per di più, è importante saper applicarli nel modo corretto. Ad esempio, la segnatura di una funzione deve necessariamente specificare se un suo argomento o il suo valore di ritorno abbia un tipo classico o opzionale.
Il controllo del rispetto di queste regole, all'interno di un programma, viene fatto a tempo di compilazione.

4.2.2 Corrispondenza tra le due versioni

Durante il processo di porting, è stato tenuto un occhio di riguardo per quanto riguarda la corrispondenza tra la versione originale dello script (in Python) e quella in Swift. Ovvero, si è cercato di mantenere i due script il più simili possibile, senza stravolgere completamente la struttura del programma. Oltre a ridurre il rischio che i comportamenti di questi due subiscano delle differenze, l'importanza di questa accortezza proviene dal fatto che in futuro potrebbero venire applicate delle modifiche alla copia originale e quindi queste si dovranno rispecchiare nella versione scritta in Swift. Nel caso in cui il programma venisse totalmente riprogettato, si perderebbe la corrispondenza tra le due versioni e quindi una modifica sostanziale nella versione originale, richiederebbe un grande lavoro di adattamento nella versione scritta in Swift.

4.3 Adattamento all'ambiente mobile (iOS)

Per portare a termine il porting dello script di preparazione dei dati, è stato necessario compiere diversi passi. Sono state applicate una serie di modifiche, tenendo conto di tutte le differenze tra i due linguaggi e ambienti di programmazione che sono state descritte in precedenza. In situazioni particolarmente delicate, in cui il dislivello tra i due linguaggi impediva la trascrizione diretta,

non si ha potuto far altro che riprogettare qualche piccola funzione, mantenendo l'opportuna cautela.

4.3.1 Modifica della sintassi

Per prima cosa, è stato utile ricostruire il corpo dello script, ovvero tutte le funzioni principali, modificando la sintassi di base. Sostanzialmente, sono stati modificati gli scope di Python basati su indentazione, in scope C-like, formati da parentesi graffe. Inoltre, sono state modificate tutte le condizioni presenti nelle intestazioni dei cicli e di altri costrutti. Esempi sono l'operatore "and" che diventa "&&" e "for i in range(n)" che diventa "for i in 0..<n".

4.3.2 Dichiarazione delle variabili

Dato che nello script di partenza non appariva alcuna dichiarazione di varibile, ma soltanto assegnamenti, è stato necessario aggiungerle da zero, in modo da adattarsi alla sintassi di Swift. Come procedimento generale, è stato deciso di rendere costanti (utilizzando la keyword "let") tutte le variabili che che contenevano i parametri immutabili, inseriti manualmente nello script. Invece, è stata utilizzata la semplice keyword "var" per tutte le variabili rimanenti, ad eccezione di quelle che rimanevano immutate, con le quali è convenuto quindi utilizzare "let". Inoltre, dato che lo script era composto da più funzioni modulari che venivano invocate da una singola funzione di interfaccia, è stata aggiunta la keyword "private" a tutte queste funzioni, al fine di rendere l'interazione con il nuovo modulo meno incline ad errori.

4.3.3 Aggiunta di tipi espliciti

Sempre al fine di mantenere il codice simile tra le due versioni dello script, si è fatto un forte utilizzo dell'inferenza di tipo fornita da Swift. Quindi, ove possibile non si è specificato il tipo nella dichiarazione di variabili. Tuttavia, è stato obbligatorio specificare il tipo in molti contesti, come nella segnatura delle funzioni e nella dichiarazione di variabili, senza assegnazione diretta. A causa del fatto che Python non rende esplicito il tipo delle variabili, nel porting si è dovuto gestire caso per caso, facendo controlli accurati sull'entità di ogni variabile utilizzata.

4.3.4 Mappatura delle strutture dati

La questione delle strutture dati è stata abbastanza delicata, non tanto per le strutture semplici, ma per quelle più complesse, che sono organizzate diversamente all'interno dei due linguaggi.
In Python si può sfruttare una libreria nativa per convertire dati in formato JSON direttamente in strutture del linguaggio (dizionari, liste, ecc.). In Swift esiste una libreria di terze parti, chiamata *SwiftyJSON*, che ha un comportamento simile a quella di Python, ma con differenze dovute alla presenza dei tipi

opzionali nel linguaggio. Per mantenere il codice Swift, simile a quello Python, è stata utilizzata *SwiftyJSON* per creare una struttura a dizionario "[String : [Double]]" che potesse essere utilizzata per manipolare i dati JSON di una registrazione. La chiave di tipo "String" contiene il timestamp, mentre valore come array di "Double" è la lista dei valori ottenuti dai sensori in quell'istante.

4.3.5 Adattamento del passaggio di argomenti

Per quanto riguarda tutti i parametri delle funzioni che appartenevano a tipi primitivi, la trascrizione è stata diretta e pressocché priva di complicazioni. Invece, non è stato altrettanto semplice gestire parametri di tipi più complessi, come i dizionari. Infatti, in questo caso è stato riscontrato il problema della differenza di modalità in cui vengono passati gli argomenti alle funzioni. Mentre Python tratta i dizionari come una sorta di oggetto e quindi li passa con un riferimento, Swift li tratta come delle struct (simili a quelle esistenti in C) e al momento del passaggio genera una vera e propria copia.
All'interno dello script originale, sono state apportate modifiche al dizionario contenente i dati della registrazione, proveniente dal formato JSON. Tutta la pulizia e la preparazione dei dati sono state fatte principalmente in questo modo. Dato che le modifiche effettuate sui dati hanno raggiunto un numero abbastanza elevato, lo script Python è stato suddiviso in funzioni modulari, in modo da far applicare una sola modifica ad ogni funzione. Per la maggior parte dei casi, queste funzioni possono essere definite come procedure, in quanto operano sui dati come side effect, senza restituire alcun risultato. La maniera più semplice e intuitiva di fare ciò è stata quella di passare il dizionario dei dati a tutte le funzioni. Facendo così, ogni funzione ha potuto lavorare direttamente sull'oggetto e applicare modifiche allo stesso.
Trasferendo questa logica su Swift direttamente, non si sarebbe ottenuto il risultato desiderato. Oltre al fatto che all'interno di ogni funzione sarebbe esistita solamente una copia locale del dizionario e quindi nessuna modifica sarebbe stata portata a termine alla fine del processo, la differenza più proibitiva sarebbe stata che di default Swift istanzia le variabili provenienti dagli argomenti di una funzione come costanti, impedendo così la loro modifica. Quindi, una potenziale opzione sarebbe stata quella di restituire il dizionario modificato come valore di ritorno della funzione e aggiornare la variabile nello scope chiamante, ma per mantenere consistenza con lo script originale, si è scelta un'altra strada. Swift offre un modo di passare argomenti ad una funzione attraverso un riferimento modificabile. Questo meccanismo richiama un po' il passaggio di puntatori che esiste in C. In questa maniera, è possibile modificare direttamente il dizionario all'interno delle funzioni, mantenendo il side effect come si faceva nello script in Python. Per rendere questo possibile, si deve specificare nella segnatura, che i parametri della funzione siano di tipo "in-out" e precedere la variabile passata come argomento con il simbolo "&", per forzare il passaggio per riferimento (similmente a come si farebbe in C).

4.3.6 Gestione dei tipi opzionali

Come già anticipato, Swift incentiva l'utilizzo di tipi opzionali qualora una certa variabile potesse avere valore nullo. Questo avviene sempre con i dizionari, in quanto potrebbe accadere che ad una specifica chiave non sia associato alcun valore. Dato che nello script si è fatto un frequente utilizzo del dizionario contenente i dati della registrazione, questa questione si è presentata spesso. Nel caso di questo dizionario, le chiavi in questione corrispondevano ai timestamp del campionamento, quindi si aveva una certezza riguardo l'esistenza delle stesse e dunque è sempre stato possibile eseguire l'unwrap forzato dei valori, senza dover forinire un valore di default. In qualche altra situazione un po' più delicata, è convenuto invece selezionare un valore di default per garantire un funzionamento corretto nei casi estremi di presenza di valori nulli.

4.3.7 Sostituzione delle librerie

Trovandosi con Swift e su un sistema operativo mobile, l'intero ecosistema di librerie e funzionalità che sfruttava lo script originale, scritto in Python, non può essere acceduto. Questo implica una necessità di adattamento verso il nuovo ambiente. Alcune librerie che venivano utilizzate dallo script originale non sono più necessarie, grazie a delle differenze nel comportamento dei due programmi. Altre, invece, sono rimaste necessarie per il corretto funzionamento dello script e quindi sono state rimpiazzate da opportune librerie Swift.
Dato che il programma originale doveva lavorare su tutte le registrazioni di dati effettuate dagli utenti, faceva largo utilizzo del file system per leggere e scrivere tutti i dati in una struttura di directory dedicata. Per questo venivano utilizzate delle librerie di Python per l'interazione con il file system stesso. Nel caso di Swift, queste non sono state necessarie, in quanto il processo avviene soltanto sui dati di una singola registrazione, e non serve quindi salvare questi su memoria di archiviazione. Infatti, essi vengono caricati nel database e processati in locale immediatamente dopo che la raccolta abbia terminato.
Tra le vere e proprie azioni applicate sui dati, è capitato spesso di dover utilizzare operazioni di algebra lineare, o più in generale, operazioni sofisticate sui numeri. Nello script Python è stata scelta la libreria esterna *Numpy*, in quanto dotata di tutte le funzionalità da noi richieste. Degli esempi di usi che ne sono stati fatti ci sono le rotazioni 3D utilizzando matrici, calcoli di prodotti tra matrici, calcoli di deviazioni standard, ecc. Chiaramente queste tipologie di operazioni sono rimaste presenti nel programma Swift e quindi è stata fatta una ricerca di librerie sostitutive che potessero rimpiazzare *Numpy*.
Per quanto riguarda le operazioni su vettori e matrici, è stata trovata ed utilizzata la libreria *SIMD*, sviluppata direttamente da Apple. Questa libreria offre una vasta varietà di funzionalità riguardanti l'algebra lineare, infatti è stato possibile utilizzarla per definire matrici, effettuare dot e cross product, applicare normalizzazioni, ecc. Per funzioni matematiche più semplici, è stato trovato in Swift un supporto diretto, a differenza di Python che ha richiesto l'uso di altre librerie.

4.4 Uso del modello nell'app

Una volta che sono stati preparati i dati, questi sono pronti per essere passati come input al modello, quindi viene qui descritta l'interazione con esso.

4.4.1 Inserimento del modello nell'app

Per poter effettuare una predizione in locale nell'app, è chiaramente necessario che il modello classificatore venga inserito nell'app stessa. Attraverso lo strumento di creazione del modello, si è potuto estrarre il modello addestrato, compattato in un solo file con estensione ".mlmodel". Per effettuare il deploy di questo file nell'app, è bastato soltanto aggiungerlo al progetto Xcode e utilizzare l'API fornita appositamente da *CoreML*. Infatti, una volta che si ottiene il modello in quel formato, esso può essere sfruttato completamente attraverso questa libreria. *CoreML* si occupa della generazione automatica di una serie di classi che permettono allo sviluppatore di interagire con il modello. In particolare, fornendo un modello chiamato "ModelloEsempio.mlmodel", verranno generate le seguenti classi:

- "ModelloEsempio": classe utilizzata per l'istanziazione del modello nel codice, utilizzata per avviare la predizione.
- "ModelloEsempioInput": classe utilizzata per fornire l'input a "ModelloEsempio", contiene un costruttore con dei parametri autogenerati, uno per ogni feature di input del modello (con il relativo tipo).
- "ModelloEsempioOutput": classe utilizzata ricevere l'output di una predizione di "ModelloEsempio". Contiene degli attributi autogenerati (variabili a seconda della tipologia di modello), tra cui la feature di output del modello (con il relativo tipo) e l'accuratezza della predizione.

4.4.2 Esecuzione delle predizioni

Le predizioni effettuate con il modello vengono eseguite subito dopo che l'auto viene parcheggiata. Questo è possibile perchè gli unici dati che sono necessari al modello sono quelli raccolti durante la guida e fino al momento del parcheggio. In questo modo, la procedura può essere intrapresa non appena viene chiamato il metodo *Car.park()*. Dato che all'interno di *Car.park()* veniva effettuata già una chiamata a *ParkTypeSampleCollector* per l'interruzione della raccolta dati, si è pensato di integrare la predizione direttamente nella classe del raccoglitore e lanciarla subito dopo la terminazione della raccolta. A questo fine, è stato definito un metodo privato del raccoglitore, chiamato *inferType()*. Questo metodo ritorna un *ParkType*, ovvero il risultato della predizione.
All'interno di *inferType()*, la prima cosa che avviene è la la preparazione dei dati. Questo significa che viene presa la struttura JSON con i dati appena raccolti e passata all'algoritmo definito in precedenza, che effettua tutte le operazioni necessarie sui dati. Una volta che si hanno i dati processati, è possibile generare

l'input per il modello. Vengono estratte dai dati le feature necessarie per il modello e passate come argomento a *ParkTypeClassifierInput* (la classe di input per il modello *ParkTypeClassifier*). A questo punto viene istanziato un *ParkTypeClassifier* viene eseguita la predizione su *ParkTypeClassifierInput*. Al termine, si ottiene un *ParkTypeClassifierOutput*, dal quale si può estrarre il tipo di parcheggio classificato e restituirlo come valore di ritorno del metodo.

Caricamento etichetta ML nel database

Successivamente alla predizione del tipo di parcheggio, quest'ultimo può essere salvato nel database. Il caricamento avviene in maniera simile a quella in cui viene salvato il tipo ottenuto dalla notifica mostrata all'utente, attraverso la chiamata di una API del backend di GeneroCity. Tuttavia, è stato deciso di non confondere i due tipi (quello selezionato dall'utente e quello ottenuto attraverso la predizione di ML), sovrascrivendo l'uno con l'altro, bensì di creare due campi diversi per distinguere i due concetti all'interno del database. Dopotutto, l'utilità del tipo selezionato dall'utente è proprio quella di essere utilizzato nel dataset di training per addestrare il modello e portarlo al punto di essere in grado di classificare autonomamente il tipo di parcheggio stesso con buona accuratezza. Tra i motivi per cui si è scelto di non confondere i due valori ci sono:

- non avrebbe più senso effettuare il training utilizzando un dataset in cui sono presenti dati classificati dal modello stesso.
- mantenendo i valori separati, si possono fare statistiche sul miglioramento (o peggioramento) dell'accuratezza del modello, all'aumentare della dimensione del dataset. Infatti, si può controllare la percentuale dei tipi classificati correttamente, comparandoli alle etichette selezionate dagli utenti.
- si possono fare studi sul comportamento del modello, capendo in che situazioni commette più errori e quali sono invece i suoi punti più stabili.

Capitolo 5

Rappresentazione dei parcheggi sulla mappa

Tutti i procedimenti descritti in precedenza, come la raccolta dei dati, le operazioni che vengono effettuate su di essi, la predizione del tipo di parcheggio, ecc. portano al risultato di ottenere una serie di istanze di parcheggio, all'interno del database, che possiedono varie informazioni. Ogni parcheggio, tra le altre cose, può essere fornito di:

- **coordinate finali**, che indicano il punto geografico dove l'auto ha parcheggiato, sono sempre presenti (a volte potrebbero essere invalide)
- **tipo di parcheggio selezionato dall'utente**, presente per i parcheggi per cui è stata selezionata manualmente una etichietta di classificazione.
- **tipo di parcheggio predetto dal modello ML**, presente per i parcheggi per cui è stata effettuata la predizione con il modello (a meno di eccezioni, dovrebbe essere sempre presente)
- **valore di heading finale**, presente per i parcheggi per cui è stata effettuata la predizione con il modello (a meno di eccezioni, dovrebbe essere sempre presente)

Queste informazioni possono essere utilizzate per fornire dei benefici all'utente. Di fatto, tutto il lavoro che porta a questo punto ha come scopo finale lo sfruttamento delle informazioni acquisite per arricchire l'esperienza dell'utente in qualche modo.
In particolare, l'obiettivo principale era l'ottenimento del tipo di parcheggio, ma attraverso l'intero procedimento è stato possibile ottenere anche il valore della bussola senza dover fare sforzi aggiuntivi.
I due utilizzi principali di queste informazioni che per il momento vengono fatti sono mostrare visivamente sulla mappa i tipi di parcheggio, alle rispettive coordinate, e migliorare l'algoritmo di creazione dei match tra utenti che lasciano un parcheggio e quelli che ne cercano uno.
Mostrare i parcheggi sulla mappa offre all'utente la possibilità di poter trovare

facilmente delle zone di posteggio in un'area a proprio piacimento. Integrare nelle figure dei parcheggi anche altri dettagli, come il tipo di parcheggio o l'orientamento, può ulteriormente facilitare l'esperienza dell'utente, ad esempio facendogli capire in anticipo com è fatto il parcheggio che sta cercando. Ad ogni modo, la rappresentazione grafica che è stata implementata all'interno dell'app GeneroCity si tratta di una versione non definitiva, quindi poco affinata e poco testata sull' utente finale. Dato che l'app non è ancora stata rilasciata, questa rappresentazione è utilizzata principalmente per lo sviluppo ed è destinata a subire grandi miglioramenti dal punto di vista grafico e dell'interfaccia. Nonostante ciò, la logica che la gestisce e il proprio ciclo di vita, che viene giostrato dagli eventi dell'interfaccia utente, sono sufficentemente maturi e pronti ad un potenziale rilascio.
Invece, per quanto riguarda l'algoritmo di matching, esso viene utilizzato per rendere possibile uno scambio di parcheggio tra un utente che sta lasciando lo stesso e un altro che ne sta cercando uno. Essendo a conoscenza dei tipi dei parcheggi, al momento della ricerca di un parcheggio disponibile, da parte di un utente che possa prendere il posto lasciato, si potranno preferire i posti in cui è più probabile che l'auto di colui che cerca entri e scartare quelli in cui invece probabilmente l'auto non entrerà a causa delle dimensioni.
Tuttavia, in sviluppi futuri, queste informazioni potranno essere sfruttate anche in altri ambiti, o semplicemente per migliorare in altri modi i servizi già esistenti.

5.1 Recupero dei parcheggi nella zona visualizzata

Mostrare una rappresentazione grafica dei parcheggi all'utente richiede, come prima cosa, l'ottenimento delle informazioni necessarie su di essi. Per ogni *Car* presente nel sistema, vengono salvati nel database tutti i parcheggi effettuati, costituiti dalle informazioni descritte in precedenza, ed altre, tra cui il timestamp del caricamento.
Occorre quindi un metodo per scaricare le informazioni riguardanti i parcheggi che devono essere rappresentati sulla mappa. In particolare, quando l'utente osserva una specifica porzione della mappa, ha bisogno di vedere tutti i parcheggi che si trovano in quella determinata zona e nessun altro. Questo ha permesso una progettazione che minimizzasse la quantità di richieste inviate all'API backend e allo stesso tempo mostrasse visivamente tutti i parcheggi in maniera fluida e senza provocare interruzioni all'esperienza utente. Chiaramente, l'area inquadrata nella mappa cambia molto frequentemente e a seconda di come l'utente interagisce con essa. Quindi, l'algoritmo di recupero dei parcheggi deve tenere conto anche di questo fatto ed impedire che vengano invocate un numero eccessivo di chiamate verso il server.
Dunque, è stato necessario introdurre una nuova funzione fornita dall'API del backend di GeneroCity, che permettesse di scaricare le istanze dei parcheggi in maniera intelligente. Successivamente, è stato progettato un algoritmo che potesse sfruttare al meglio la nuova funzione introdotta e quindi scaricare

le informazioni sui parcheggi, tenendo conto di tutti i vincoli presenti e le necessità dell'utente finale.

5.1.1 Chiamata API per la richiesta dei parcheggi in un'area

In GeneroCity iOS sono state utilizzate le mappe di Apple, che vengono distribuite attraverso la libreria *MapKit*. La classe responsabile della rappresentazione grafica di una mappa è *MKMapView*. Quest'ultima possiede un attributo **region** che corrisponde all'informazione sull'area mostrata sullo schermo nell'istante corrente. Infatti, quando l'area visibile sul display cambia (ad esempio in seguito ad uno swipe dell'utente), anche la **region** viene aggiornata. Questo oggetto è formato dalle coordinate del centro dell'area interessata e da due valori che rappresentano lo scostamento di longitudine e quello di latitudine. Gli scostamenti indicano rispettivamente gli angoli, espressi in gradi, tra la longitudine minore e quella maggiore e la latitudine minore e quella maggiore visibili attualmente sullo schermo. In questo modo, si è a conoscenza della porzione esatta di mappa visibile dall'utente in ogni istante.
Si può utilizzare questa informazione per ottenere le istanze dei parcheggi che si trovano nel database e che sono stati registrati all'interno dell'area richiesta. Dati i parametri **lat** (latitudine del centro dell'area), **lon** (longitudine del centro dell'area), **deltaLat** (scostamento di latitudine), **deltaLon** (scostamento di longitudine), l'interrogazione al database consiste semplicemente nella selezione dei parcheggi **p**, relativi a qualsiasi auto, tali che:

$$\mathbf{lat} - \mathbf{deltaLat} \leq \mathbf{p.latitude} \leq \mathbf{lat} + \mathbf{deltaLat}$$
$$\text{e}$$
$$\mathbf{lon} - \mathbf{deltaLon} \leq \mathbf{p.longitude} \leq \mathbf{lon} + \mathbf{deltaLon}$$

Questa interrogazione è stata inserita all'interno di una nuova chiamata API, di tipo "GET". Questa chiamata è stata definita "Get area park list", in quanto restituisce la lista di parcheggi presenti all'interno dell'area indicata.
I parametri accettati dalla chiamata sono tutti e soli quelli appena definiti, tutti di tipo float. Inoltre, tutti i parametri sono obbligatori, in quanto sono tutti essenziali per il corretto calcolo del risultato.
La risposta fornita da questa chiamata consiste in una lista di oggetti in formato JSON, contenenti le informazioni dei singoli parcheggi trovati. Ogni oggetto di un parcheggio contiene: le coordinate, il timestamp, le etichette del tipo di parcheggio (sia quella selezionata dall'utente che quella predetta dal modello classificatore), l'heading, ecc. Qualora nel database alcuni di questi campi non fossero obbligatori, i rispettivi valori restituiti potrebbero essere nulli.
All'interno del modulo API dell'app Generocity iOS è stata quindi aggiunta una una funzione di interfaccia, in grado di effettuare una chiamata alla funzionalità backend appena descritta. Tra i vari parametri della funzione è presente una callback, che viene eseguita nel momento in cui viene ricevuta la risposta dal server. Questa callback contiene un oggetto JSON, che è esattamente la lista

delle istanze dei parcheggi trovati. In questo modo, è possibile ottenere la lista desiderata in maniera asincrona, direttamente con una chiamata a funzione.

5.1.2 Scheduling delle richieste

Come è stato anticipato, i parcheggi che devono essere mostrati sulla mappa devono essere scaricati con un criterio che tenga conto di diversi vincoli. Al centro di tutto vi è l'esperienza dell'utente. Infatti, si è cercato di progettare un algoritmo di recupero dei parcheggi che permettesse di mostrare immediatamente, o comunque con poco ritardo, le istanze sulla mappa. Inoltre, si è tentato di far mantenere questa proprietà anche in seguito ad un potenziale cambiamento della regione di mappa visibile sullo schermo, causato da uno swipe dell'utente stesso, o da qualsiasi evento all'interno dell'app. Questo può essere classificato come un requisito funzionale. Di contro, possono essere individuati diversi requisiti non funzionali che pongono dei vincoli e delle limitazioni che devono essere rispettati per garantire la fattibilità di questa funzionalità:

- **richiesta della sola area visibile**: al crescere del numero di parcheggi salvati nel database, aumenta la densità dei parcheggi in una data zona e quindi aumenta il numero di potenziali istanze da scaricare. Al fine di contenere le dimensioni del payload delle risposte ricevute dal server, conviene ridurre il più possibile la grandezza della zona richiesta, idealmente soltanto l'area circostante la **region** attualmente visibile. Questo meccanismo non sarà comunque sufficiente per quando il numero dei parcheggi presenti nel database diverrà molto grande. Con sviluppi futuri dell'app si potrà risolvere il problema, ad esempio richiedendo solo i parcheggi registrati in un certo intervallo di tempo, oppure ponendo un limite alla lunghezza della lista in risposta.

- **intervallo di tempo tra due richieste all'API**: la regione della mappa visualizzata sullo schermo può cambiare molte volte e molto rapidamente. Questo fatto potrebbe creare problemi se implicasse un invio non controllato di richieste. Infatti, dato che l'evento di cambiamento della **region** della mappa viene lanciato per ogni piccola variazione, l'effettuare una nuova richiesta ogni volta potrebbe generare decine di chiamate al secondo. Questa cosa sarebbe chiaramente inutile e deleteria, perché il server verrebbe inondato di richieste che oltretutto richiederebbero una lista quasi uguale di parcheggi. Anche nel lato client questo sarebbe dannoso, dovendo gestire un traffico troppo elevato di richieste e risposte HTTP. Viene naturale pensare quindi che sia necessario un limite sul numero di richieste effettuabili in un certo intervallo di tempo.

- **richieste di aree molto grandi**: la mappa offre la possibilità di modificare lo zoom della visuale, così da ingrandire e rimpicciolire l'area visibile sullo schermo. Quando viene applicato uno zoom molto basso e quindi l'area visibile si ingrandisce in maniera considerevole, le zone di

posteggio diventano troppo piccole e invisibili all'utente. Questo significa che quando la **region** della mappa è molto grande, non ha senso mostrare la rappresentazione dei parcheggi. Inoltre, richiedere la lista dei parcheggi registrati in una grande zona porterebbe al problema già discusso di una quantità di istanze trovate troppo grande. Questo farebbe sorgere dei problemi relativi anche alla quantità elevata di memoria occupata nel sistema operativo per l'applicazione.

- **richieste della stessa area**: quando la **region** della mappa cambia di un piccolo delta, la maggior parte dei parcheggi richiesti sarebbero gli stessi che erano stati ottenuti alla richiesta precedente. Quindi, si possono evitare richieste duplicate per la stessa zona e occorre un metodo per evitare di inviare richieste inutili.

Con lo scopo di soddisfare tutti questi requisiti, è stato progettato un algoritmo di scaricamento dei parcheggi che effettua diversi controlli prima di procedere con una nuova richiesta. É stata così definita una classe *ParkingOverlaysLoader*, responsabile dell'esecuzione delle richieste all'API. Un'istanza di questa classe viene gestita dalla mappa. In particolare, vengono sfruttati degli eventi generati dalla mappa per chiamare il metodo *load()* di *ParkingOverlaysLoader* e quindi effettuare una nuova richiesta, se questa viene permessa.
Come già annunciato, la mappa genera un evento ogni volta che la **region** è soggetta ad un piccolo cambiamento. Questo evento può essere catturato attraverso il metodo *mapViewDidChangeVisibleRegion()*, offerto dal delegate della mappa stessa. In questo modo, si può effettuare una chiamata al metodo *load()* e lasciar gestire al *ParkingOverlaysLoader* la scelta di compiere o no una nuova richiesta.
Dopo la prima chiamata, che genera sempre una richiesta all'API, entreranno in regime una serie di condizioni che ogni volta controlleranno se sia il caso di procedere con una nuova richiesta. Sono utilizzate delle variabili che mantengono lo stato del *ParkingOverlaysLoader*: **lastQueryTimestamp** in cui viene salvato l'istante dell'ultima richiesta e **lastQueryRegion** in cui viene invece ricordata l'area per cui è stata fatta l'ultima richiesta. Definendo **MIN_TIME_BETWEEN_QUERIES** come il tempo minimo che deve passare tra una richiesta e la successiva e **MAX_REGION_DELTA** come il massimo scostamento (angolo in gradi) tra i due estremi visibili della mappa per cui abbia senso effettuare una richiesta, vengono controllate le seguenti condizioni:

$$\textbf{mapView.region.span.latitudeDelta} > \textbf{MAX\_REGION\_DELTA}$$

controlla che la visuale attuale abbia una **region** più grande della massima permessa. Quindi il numero dei potenziali parcheggi presenti sarebbe troppo elevato.

$$\textbf{Date().timeIntervalSince1970} - \textbf{self.lastQueryTimestamp} < \textbf{MIN\_TIME\_BETWEEN\_QUERIES}$$

controlla che il tempo trascorso dall'ultima richiesta al momento attuale sia minore del limite minimo predisposto. In questo caso si invierebbe una richiesta dopo troppo poco tempo rispetto alla precedente.

$$
\begin{gathered}
abs(\textbf{mapView.region.center.latitude} - \\
\textbf{lastQueryRegion.center.latitude}) < \\
\textbf{lastQueryRegion.span.latitudeDelta}/2 \\
\text{e} \\
abs(\textbf{mapView.region.center.longitude} - \\
\textbf{lastQueryRegion.center.longitude}) < \\
\textbf{lastQueryRegion.span.longitudeDelta}/2 \\
\text{e} \\
\textbf{mapView.region.span.latitudeDelta} < \\
\textbf{lastQueryRegion.span.latitudeDelta} * 1.5)
\end{gathered}
$$

controlla che la zona attualmente visualizzata sulla mappa non sia cambiata abbastanza rispetto a quella dell'ultima richiesta. Come descritto più avanti, la richiesta viene effettuata per una zona leggermente più grande di quella visibile, e così, quando la visuale si muove di un piccolo delta, i parcheggi caricati in precedenza riescono ancora a coprire l'intera area visibile. In questo caso, non è necessario scaricare i parcheggi nuovamente. Da notare che la zona inquadrata sulla mappa può variare anche venendo rimpicciolita o ingrandita, e non solo venendo traslata. Quindi, può accadere che l'utente rimpicciolisca la mappa e quindi esca fuori dalla zona coperta dall'ultimo scaricamento. La terza condizione in "and" logico controlla che questo non accada.

Se almeno una di queste macro-condizioni è soddisfatta, la richiesta all'API viene interrotta, altrimenti vengono aggiornate le variabili con i dati attuali e si prosegue.
Quindi viene effettivamente chiamata la funzione collegata all'API (definita in precedenza) e inviata la richiesta. I parametri **deltaLat** e **deltaLon** vengono presi il doppio dello scostamento che dal centro della mappa va fino al bordo della porzione di area visibile. In questo modo, il server risponderà con una lista di parcheggi che include anche le istanze che si trovano nelle estreme vicinanze dell'area visibile.

5.2 Disegno dei parcheggi sulla mappa

Una volta che è stata definita la dinamica per lo scaricamento delle istanze di parcheggio, c'è bisogno di un modo per salvare queste in locale e successivamente mostrarle visivamente sulla mappa.
Per impedire che lo scaricamento dei parcheggi vada a occupare troppa memoria all'interno dell'app, si è deciso di eliminare le istanze scaricate con una richiesta precedente, non appena arrivi il risultato di una nuova. Inoltre, così facendo, non si va ad applicare caching a particolari zone della mappa e così, nel caso in cui vengano registrati nuovi parcheggi nel database, questi verranno

scaricati non appena l'utente visualizzerà la zona interessata.
La maniera che fornisce *MapKit* di disegnare delle forme sulla mappa è attraverso gli *MKOverlay*. Infatti, attraverso questi oggetti, è possibile rappresentare forme di svariate tipologie, applicando colori, trasparenze e altro.

5.3 Informazione sul tipo di parcheggio

5.4 Informazione sull'orientamento del parcheggio

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

5.5 Approccio community-driven

I parcheggi vengono aggiunti automaticamente quando gli utenti li effettuano. Per evitare parcheggi invalidi, si mostrano per bene quando tanti parcheggi sono stati fatti nello stesso posto, maggiore sicurezza ... In questo modo l'utente ha l'informazione su dove si trova una zona di posteggio, tutto attraverso l'automatismo dell'app, non c'è bisogno che esse vengano impostate dagli sviluppatori

5.6 Beneficio dell'utente

5.6.1 Informazione visiva

5.6.2 Disponibilità del parcheggio per auto di una certa dimensione