Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

TITOLO ITALIANO

TITOLO INGLESE

NOME CANDIDATO

Relatore: *Relatore*
Correlatore: *Correlatore*

Anno Accademico 2014-2015

INDICE

ELENCO DELLE FIGURE

"Inserire citazione"
— Inserire autore citazione

FASI PER IL RICONOSCIMENTO DELLA POSIZIONE ALL'INTERNO DEGLI EDIFICI

L'identificazione della posizione all'interno di un edificio si svolge in 2 fasi:

1. Scansione dell'ambiente
2. Ricerca della posizione

SCANSIONE DELL'AMBIENTE

Analisi statica dell'ambiente chiuso, nel quale il software raccogliera' le onde magnetiche per ogni intervallo di tempo, le classifichera' con una semplice label la quale rappresentera' la zona di appartenenza.
La scansione a sua volta composta da diverse sotto-fasi cioè:

1. Raccoglimento dei dati
2. Estrazione del magnitudo
3. Raggruppamento
4. Estrapolazione delle *features*

Raccoglimento dei dati

Innanzitutto esaminiamo la composizione dei dati che stiamo andando ad estrarre: si tratta di onde magnetiche quindi strutturate nel seguente modo:

$$(x, y, z)$$

I tre valori sono espressi in μT (micro Tesla), unita' di misura della densita' di un flusso magnetico. Ad ogni onda magnetica viene assegnata una *label*: una stringa od un numero che identifica univocamente una parte dell'ambiente chiuso

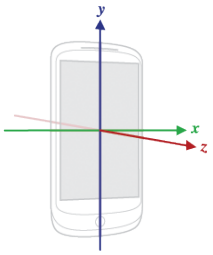

Figura 1: Raffigurazione grafica delle onde raccolte

Estrazione del magnitudo

Per estrarre l'intensità di ogni onda magnetica eseguiamo semplicemente la norma euclidea di un vettore:

$$\sqrt{x^2 + y^2 + z^2}$$

Raggruppamento

Le onde magnetiche con la stessa *label* vengono raggruppate in *fingerprints*, insiemi di dimensione prefissata. A livello logico, ogni *fingerprint* cerca di identificare univocamente un punto all'interno di una zona, identificata con una *label*. L'insieme di *fingerprints* quindi, cerca di distinguere, tramite le caratteristiche dei campi elettromagnetici di ciascun punto, ogni *label* dall'altra.

Estrapolazione delle features

Per ogni *fingerprint*, l'estrazione delle *features* consiste nell'estrazione di variabili statistiche. In questo specifico caso sono:

- Media
- Varianza
- Deviazione standard
- Mediana
- Media troncata
- Coefficiente di variazione
- Massimo
- Minimo
- $1°, 5°, 95°, 99°$ percentile
- $1°, 2°, 3°$ quartile

RICERCA DELLA POSIZIONE

Dopo aver scansionato l'ambiente questa fase viene eseguita dal cliente durante l'utilizzo dell'applicazione. La ricerca consiste nel creare sul momento una *fingerprint* e poi, tramite un algoritmo di apprendimento automatico, cercare di inferire la *label* su cui ci troviamo. Nello specifico, l'algoritmo utilizzato e' il *k nearest neightbours*.

K-NEAREST-NEIGHTBOURS

Uno degli algoritmi piu' semplici di apprendimento automatico, e' il *k-nearest-neightbours* dove l'input consiste in k elementi presi dal *training set* piu' vicini in base ad un criterio scelto da chi utilizza l'algoritmo (per esempio la distanza euclidea o di Mahalanobis).

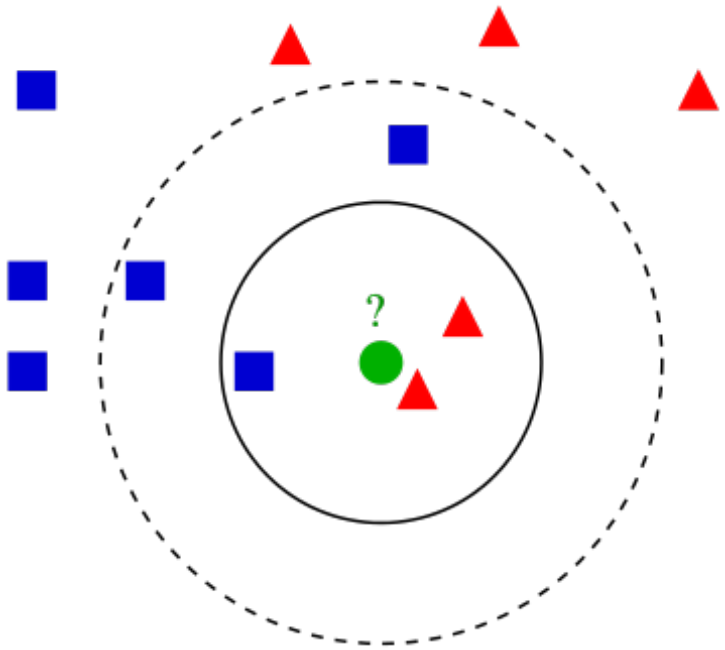

Figura 2: Esempio grafico dell'algoritmo KNN

Scelta del parametro k

La scelta del parametro dipende, ovviamente, dal tipo dei dati che abbiamo e dalla quantita', anche se in generale piu' e' grande k meno rumore viene generato da questo algoritmo. Un buon metodo per trovare il giusto valore e' l'uso di tecniche euristiche, come la *cross validation*. Un altra fonte di rumore di cui bisogna stare attenti e' la presenza di *features* insignificanti nella ricerca del vicino. Per porre rimedio possiamo, ad esempio, usare un algoritmo genetico per selezionare le *features* piu' significative

Cross validation

Tecnica per migliorare le performance di un classificatore, consiste nel suddividere l'intero *dataset* in n parti uguali (di solito 10), ognuna delle quali svolgera' per una volta il ruolo di *validation set* mentre il resto sara' il *training set*. Questa tecnica risolve vari problemi tra i quali l'*overfitting*.

NAIVE BAYES

Teorema di bayes

Un altro tipo di apprendimento usato per classificare le *label* e' Naive bayes: esso si fonda sul famoso teorema di bayes enunciato come segue:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

Prima di spiegare in cosa consiste esattamente *Naive Bayes*, occorre spiegare alcuni preconcetti:

Rete bayesiana

Una rete bayesiana e' un grafo diretto aciclico i cui nodi rappresentano le variabili casuali del sistema analizzato mentre gli archi rappresentato la condizione di dipendenza fra nodi.

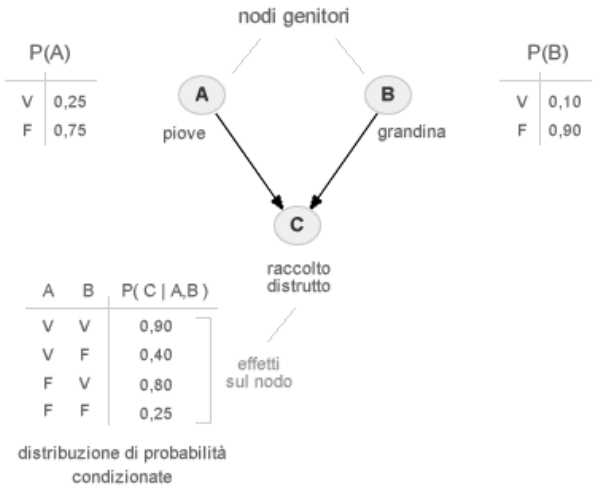

Figura 3: Esempio di rete bayesiana

STRUTTURA DEL SOFTWARE

LINGUAGGI E FRAMEWORK

L'applicazione ha come *target platform* Android percio' il linguaggio usato principalmente e' stato Java. Da notare che su Android e' presente nella versione 7, percio' non sono disponibili alcune funzionalita' come i metodi di default, *stream*, *lambda expression* ecc. Per rimediare soprattutto alla mancanza di quest'ultime, che migliorano la lettura e scrittura di alcune parti di codice, ho affiancato un altro linguaggio a Java: Kotlin. Senza fare un confronto tra i due linguaggi, Kotlin mi ha permesso di scrivere lambda expression compilando un bytecode perfettamente compatibile con Java 6.

INTERFACCIA GRAFICA

L'interfaccia e' stata realizzata ai fini di test pratici delle funzionalita' finali dell'applicazione quindi non ha una grande cura da un punto di vista estetico come vedremo piu' avanti. La parte alta contiene delle label raffiguranti i 3 valori catturati tramite il magnetometro e presi tramite le API di Android.
Nel mezzo ci sono dei pulsanti per:

- Iniziare/Terminare la scansione dell'ambiente.
- Incrementare la *label* che verra' assegnata alla prossima *fingerprint* registrata
- Iniziare/Terminare la ricerca.
- Serializzare tutti i dati registrati finora
- De-serializzare i dati salvati in un JSON.

Nella parte bassa invece c'e' una *textbox* contenente il log che verra' stampato durante l'esecuzione del programma.

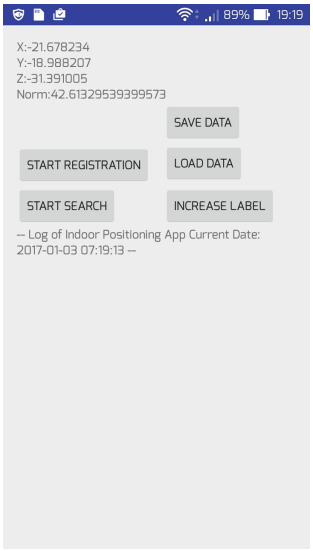

Figura 4: Interfaccia grafica all'avvio dell'applicazione

PERSISTENZA DEI DATI

L'applicazione consente anche di serializzare tutti i dati registrati fino a quel momento nel formato standard JSON tramite la libreria *gson* che fornisce dei metodi semplificati per il linguaggio Java per la serializzazione/deserializzazione di oggetti. Il file viene salvato nella cartella dati dell'applicazione non visibile all'utente.

STRUTTURA DEL CODICE E DESIGN PATTERN

Nello sviluppo del software sono stati applicati vari *design pattern* visti durante i vari corsi e principi di programmazione. Fra questi ultimi abbiamo il *dependency inversion principle*, il *open closed principle*. Riguardo i *design pattern*, ho usato molto l'*observer*, il *template* e *factory*.

BASE DEI TEST

Per testare l'effettivo funzionamento dell'applicazione ho usato alcune stanze di casa mia ed ho assegnato a ciascuna di esse una *label*. Qui di seguito una piccola piantina rappresentante le stanze utilizzate:

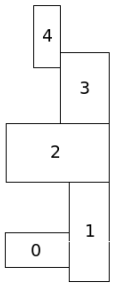

Figura 5: Una piccola raffigurazione delle stanze usate per le prove con sopra scritto la *label* assegnata

RISULTATI DEI TEST