

NoiSense

Piattaforma crowdsensing per la misurazione dell'inquinamento acustico nelle varie zone del globo

Alessio Di Dio - 984275

Emanuele Fazzini - 983681

A.A. 2020/2021

Indice

1	Introduzione	3
2	Progettazione	4
2.1	Web	4
2.1.1	Frontend	4
2.1.2	Backend	6
2.1.3	Database	7
2.2	Applicazione	8
3	Implementazione	11
3.1	Web	11
3.1.1	Homepage	11
3.1.2	Previsioni	21
3.1.3	API	24
3.2	Applicazione	34
3.2.1	Algoritmi di privacy	39
3.2.2	Invio di una nuova rilevazione	41
4	Risultati	43
4.1	Quality of Service	43
4.2	Previsioni	44
4.3	Trade-off QoS-Privacy	47

Elenco delle figure

2.1	Main Activity	9
2.2	Settings	9
3.1	Positions	12
3.2	K-Means clustering	12
3.3	Open Layers clustering	12
3.4	Heatmap	12
3.5	Previsione con input text	22
3.6	Previsione con mappa	22
3.7	Swagger API	25
3.8	Settings XML	35
3.9	Main XML	36
4.1	Rilevazioni autogenerate	44

Introduzione

NoiSense è una piattaforma crowdsensing che permette agli utenti di inviare l'inquinamento acustico rilevato nella propria posizione attuale e visualizzare tutte le misurazioni su una mappa presente nel sito web. Permette anche, attraverso l'applicazione, di visualizzare il rumore medio nel raggio di tre chilometri così da essere consapevoli dell'intensità del suono presente in una determinata zona.

Attraverso l'applicazione mobile sviluppata per la sistema Android gli utenti sono in grado di misurare il rumore tramite il microfono presente nel proprio smartphone, che in seguito invieranno, insieme alla loro posizione, alla piattaforma web la quale ha il compito di salvare tale misurazione in un database e mostrarla nel proprio frontend su una mappa, in modo tale da risalire alla posizione della rilevazione e quindi associare il rumore rilevato ad una determinata zona nel mondo.

La posizione prelevata tramite il GPS può essere alterata e inviata scegliendo uno tra due algoritmi:

- GPS Perturbation, che tronca le coordinate in modo tale da spostare la posizione falsificata da quella reale;
- Dummy Updates, che invia altre posizioni false in un intorno di quella vera in modo tale da non sapere quale di esse sia la locazione dell'utente.

È possibile scegliere tale preferenza nelle impostazioni dell'applicazione, in modo da garantire la privacy e rendere impossibile risalire alla posizione reale del soggetto mittente della rilevazione.

In questa relazione viene presentato tutto lo sviluppo che è avvenuto per la creazione di NoiSense, dalla progettazione all'implementazione ed infine verranno presentati i risultati ottenuti sotto forma di qualità del servizio offerto calcolato tramite errore quadratico medio tra misurazioni vere e false, saranno anche descritti gli esiti ottenuti dalle previsioni dei rumori in aree con misurazioni assenti o limitate e il trade-off QoS-Privacy per l'algoritmo GPS Perturbation sulla base di quattro valori α : 0.25, 0.5, 0.75, 1.

Progettazione

2.1 Web

Come strumenti per la creazione della piattaforma web abbiamo usato Django, un framework di Python per la realizzazione di infrastrutture client-server e PostgreSQL come DBMS con estensione PostGIS per la gestione dei dati spaziali. Per la visualizzazione della mappa sul sito web abbiamo optato per Open Layers.

Per l'utilizzo online della piattaforma, non disponendo di un server e quindi doverdolo pagare, abbiamo usufruito di un software chiamato ngrok, disponibile gratis per la sua versione di base, che ci ha permesso di accedere al server in locale tramite un link generato all'avvio del programma. Non abbiamo un URL del sito permanente visto che cambia ogni volta che ngrok viene riavviato.

L'infrastruttura web può essere altresì suddivisa in tre sezioni:

- Frontend;
- Backend;
- Database.

2.1.1 Frontend

Avendo usato Django per la creazione della piattaforma web, le risorse frontend non si trovano in una singola cartella ma in diverse posizioni che di seguito andrò a descrivere.

Per quel che riguarda la parte HTML, vi sono 4 file principali situati nella cartella template della piattaforma:

- base.html: è il file HTML di base nel quale vengono inizializzati i tag <html>, <head> e <body>, dal quale gli altri file HTML estenderanno il loro codice in

modo tale da non dover riscrivere in ogni file le stesse righe, come ad esempio le importazioni delle librerie necessarie¹;

- index.html: la pagina principale del sito dal quale si possono visualizzare le rilevazioni sulla mappa anche sotto forma di clustering o di heatmap;
- previsioni.html: abbiamo scelto di posizionare la pagina delle previsioni del rumore nelle aree con misurazioni assenti o limitate in un altro scheda HTML in modo da suddividere meglio la struttura del sito. Tale pagina è raggiungibile dal link "Forecasts" situato nell'index.html.
- swagger.html: per la visualizzazione delle API sul sito abbiamo installato Swagger, un tool che tramite questa pagina HTML mostra sotto forma di lista le API presenti nella piattaforma. In più da anche la possibilità di testarle.

In merito a tutti gli altri file che non interagiscono con oggetti inviati dal Backend, quali javascript, css, immagini, etc. sono situati nella cartella web -> casprog -> casprog -> static. I file javascript e css da noi creati si trovano dentro la cartella homepage in static. All'interno troviamo altre due cartelle:

- css: contiene un solo file css chiamato style.css, l'unico foglio di style sviluppato da noi per migliorare la grafica del sito;
- js: contenente due file javascript, uno per l'index.html e uno per la pagina previsioni.html.

In static vi sono altre due cartelle: admin e rest_framework. La prima è stata auto-generata dal plugin django-framework installato tramite pip, la seconda è stata creata dal plugin django-rest-framework installato tramite pip che serve alla creazione e visualizzazione delle API del sito.

¹I framework importati sono: jQuery, Bootstrap, Open Layers. In più troviamo l'importazione dei file css da noi creati.

2.1.2 Backend

Django offre la possibilità di gestire una parte del Backend tramite la pagina admin raggiungibile tramite l'aggiunta di /admin/ all'url del sito. Inizialmente abbiamo deciso di impostare la struttura tramite django-admin, ma in seguito abbiamo scelto di gestire tutto tramite codice.

Il Backend va diviso in due sezioni: la sezione del sito posizionata in web -> casprog -> casprog e la sezione delle api situata nella cartella in web -> casprog -> api.

La prima riguarda tutto il materiale per l'utilizzo del sito web, come descritto nella sezione precedente troviamo i file relativi al frontend della piattaforma e, in più, i vari file Python per la gestione del server di cui adesso discuteremo.

Nel file urls.py sono situati i vari url raggiungibili dalla pagina web. Ad ogni url corrisponde una funzione descritta nel file views.py e queste funzioni possono: o dare in risposta la pagina HTML richiesta, oppure eseguire uno script Python e dare come risposta una stringa come risultato dello script².

Il file models.py è necessario nel caso in cui vogliamo definire dei modelli da utilizzare poi nella pagina admin. Tali modelli serviranno poi per interagire con il database, ma nel nostro caso non ne abbiamo fatto uso.

Il file admin.py serve per andare a visualizzare ed interagire con i modelli creati nel file models.py, importandoli.

I file asgi.py e wsgi.py servono per prelevare dal file settings.py le impostazioni necessarie per far sì che Django funzioni.

La seconda sezione riguarda le chiamate API da parte dell'applicazione per richiedere al server i dati necessari da visualizzare e utilizzare così da avere un corretto funzionamento. La piattaforma implementa sei API:

- postAVGNoiseGPSP;
- postAVGNoiseNoPrivacy;

²Questo ultimo meccanismo è stato applicato per la generazione del K-Means clustering al cambiamento del valore dello slider corrispondente al numero di clusters e alla generazione delle previsioni del rumore in aree con misurazioni assenti o limitate.

- postAVGNoiseDummy;
- postInsertRilevazioniGPSP;
- postInsertRilevazioniNoPrivacy;
- postInsertRilevazioniDUMMY.

Come per i file necessari al sito troviamo i file urls.py nel quale sono dichiarati gli url per le chiamate API e views.py per associare ad ogni url lo script Python che restituirà l'informazione richiesta sotto forma di json.

È presente un altro file Python chiamto qos.py nel quale sono implementate le funzioni relative al calcolo della qualità del servizio data la posizione dell'utente e l'algoritmo di privacy in uso.

2.1.3 Database

Il Database Management System da noi utilizzato è PostgreSQL con l'integrazione del framework PostGIS per la gestione dei dati spaziali.

All'interno, oltre alle tabelle autogenerate da Django e PostGIS, troviamo la tabella rilevazioni, dove vengono salvate le misurazioni inviate dagli utenti. La tabella rilevazioni è provvista di cinque colonne:

- posizione: la posizione della misurazione salvata sotto forma di geometry;
- rumore: il rumore in decibel rilevato nella misurazione salvato come double;
- timestamp: la data e l'ora della rilevazione memorizzati come timestamp;
- reale: booleano che contrassegna la rilevazione come falsa, nel caso in cui sia stata generata dall'algoritmo di privacy, o vera, la vera posizione dell'utente³;
- privacy: specifica quale algoritmo di privacy è stato utilizzato dall'utente per inviare la rilevazioni. Può assumere tre valori: 'gpss', se la misurazione è stata inviata tramite l'algoritmo GPS Perturbation; 'dummy', se la misurazione è stata

³Questo parametro viene usato solo per il calcolo della Quality of Service.

inviata tramite l'algoritmo Dummy Updates; 'no', se la posizione è stata inviata senza alcun algoritmo di privacy.

Non è stata impostata nessuna colonna come PRIMARY KEY visto che ognuna di esse avrebbe potuto assumere valori uguali in record differenti. PostgreSQL ha automaticamente aggiunto una colonna con valore AUTOINCREMENT in modo da assegnare ad ogni record un valore che cresce ad ogni inserimento.

2.2 Applicazione

Il sistema su cui abbiamo scelto di sviluppare l'applicazione mobile è Android. Abbiamo optato per Android perché l'utenza è maggiore rispetto alle altre piattaforme e non si necessita di strumenti particolati per la creazione di applicazioni visto che Android Studio è open source e può essere installato su Windows, Mac OS, e Linux.

L'applicazione è stata progettata con una schermata principale dal quale l'utente può richiedere tutte le informazioni di suo interesse, come:

- rumore medio nel raggio di tre chilometri dalla sua posizione;
- la qualità del servizio risultante dalle sue coordinate;
- la metrica di privacy espressa come distanza in metri tra la posizione reale e la modificata;
- il trade-off risultante tra la QoS e la privacy nel caso in cui l'utente stia utilizzando l'algoritmo di perturbazione GPS per nascondere la sua posizione reale.

Tramite lo switch "Contribuisci alle rilevazioni" l'utente sarà in grado di inviare nuove misurazioni ad una frequenza prefissata scelta nelle impostazioni. Figura 2.1

Al primo avvio dell'applicazione viene aperta la schermata "Settings" dove poter andare ad impostare le varie preferenze per quel che riguarda la privacy e l'intervallo di attesa tra l'invio di una rilevazione e l'altra.

Sarà possibile scegliere la propria preferenza di privacy sulla base di tre opzioni:

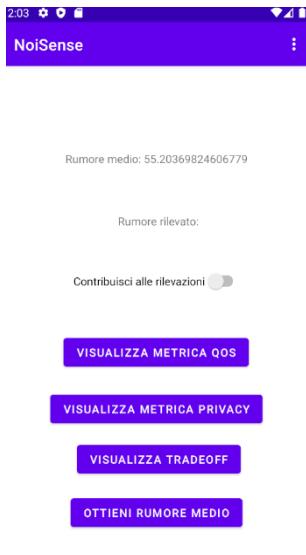


Figura 2.1: Main Activity



Figura 2.2: Settings

- GPS Perturbation: se selezionato apparirà uno slider con il quale l’utente può dare più importanza alla qualità del servizio o alla privacy, più il valore è vicino allo 0 e più le coordinate verranno perturbate;
- Dummy Updates: la propria posizione verrà oscurata inviando altre 5 posizioni con lo stesso rumore nel raggio di 3 km, insieme alla posizione reale;
- No Privacy: l’utente sceglie di non applicare nessuna privacy alla propria posizione, quindi verrà semplicemente inviata la posizione attuale dell’utente.

Tramite il bottone "MODIFICA INTERVALLO" si può scegliere quanti secondi far passare tra l’invio di una rilevazione e l’altra; il valore di default è 20 secondi.

Una volta terminata la decisione, premendo sul pulsante "FATTO" si viene riportati alla Main Activity prima descritta.

L’applicazione è strutturata su 5 file java principali:

- MainActivity: L’activity principale corrispondente anche alla schermata iniziale;

- CreaCanale: classe java usata per la creazione del notification channel necessario per il corretto funzionamento delle notifiche su Android superirie alla versione 8;
- SettingsActivity: file contenente il codice necessario alla creazione della pagina della scelta delle preferenze da parte dell'utente;
- Listener: interfaccia java nel quale sono scritti i metodi che permettono il passaggio dei valori richiesti al server, come QoS o rumore medio, dall'Intent Service Rilevazione alla Main Activity;
- Rilevazione: Intent Service che permette di inviare rilevazioni periodiche al server sulla base delle preferenze dell'utente. Riceve in risposta il rumore medio e la QoS, infine calcola la metrica di privacy.

Nella cartella res si trovano i file XML con utilizzati per dare lo stile all'applicazione.

Implementazione

3.1 Web

L'implementazione del sito web verrà suddivisa in tre sottosezioni principali:

- Homepage;
- Previsioni;
- API.

3.1.1 Homepage

L'Homepage, corrispondente al file index.html, è strutturata così: ci sono quattro radio-button, ognuno di essi ha associata una funzione per rendere visibile o no un layer sulla mappa:

- Positions: permette di visualizzare tutte le misurazioni fatte dagli utenti sulla mappa e, cliccando su una rilevazione, apparirà un popup contenente le informazioni relative a quella misurazione. Se ce ne sono più di una sullo stesso punto, appariranno due pulsanti, "Prev" e "Next", all'interno del popup che permetteranno di muoversi fra le misurazioni in quel punto. Figura 3.1;
- K-Means: permette di visualizzare l'algoritmo di clustering K-Means applicato alle rilevazioni presenti nel database. Tramite lo slider è possibile scegliere il numero di cluster applicabile alle misurazioni che va da 1 a 10. Figura 3.2;
- Clustering: è l'algoritmo di clustering integrato nelle API di Open Layers che mostra le rilevazioni raggruppate sulla base di tre valori: lo zoom che l'utente applica alla mappa, Cluster distance, ovvero la distanza tra i cluster¹ e Minimum distance, ovvero la distanza minima tra i cluster. Figura 3.3;

¹Più è alta e più il numero dei cluster tenderà ad uno, più è bassa e più il numero di cluster sarà alto.

- Heatmap: mostra l'heatmap sulla mappa preso come valore di riferimento il rumore delle rilevazioni. Figura 3.4.

NoiSense

Previsioni

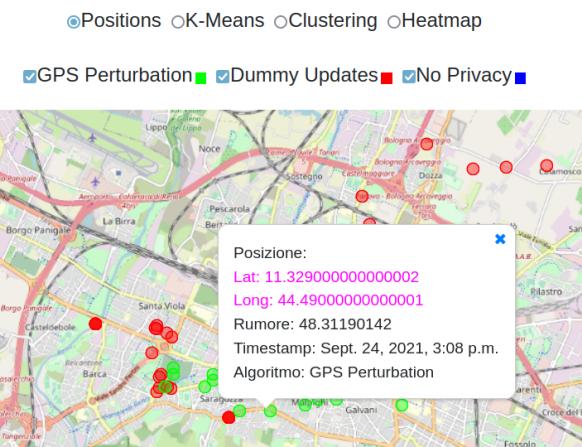


Figura 3.1: Positions

NoiSense

Previsioni

Positions K-Means Clustering Heatmap

N Clusters = 5

The number of the clusters

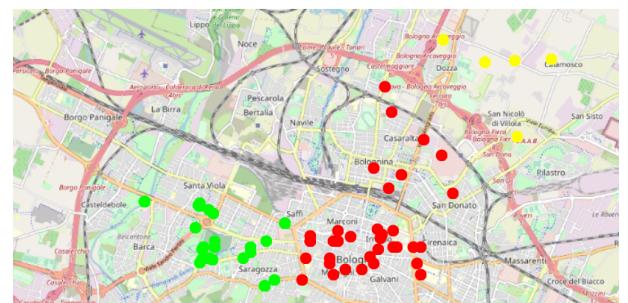


Figura 3.2: K-Means clustering

NoiSense

Previsioni

Positions K-Means Clustering Heatmap

Cluster distance

The distance within which features will be clustered together.

Minimum distance

The minimum distance between clusters. Can't be larger than the configured distance.

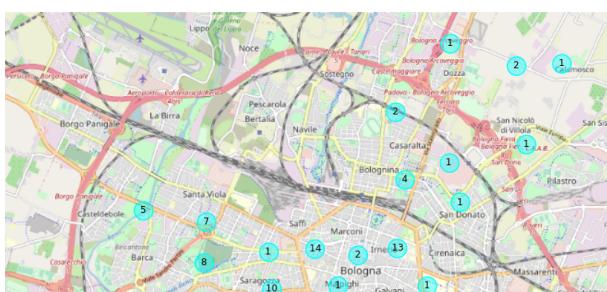


Figura 3.3: Open Layers clustering

NoiSense

Previsioni

Positions K-Means Clustering Heatmap

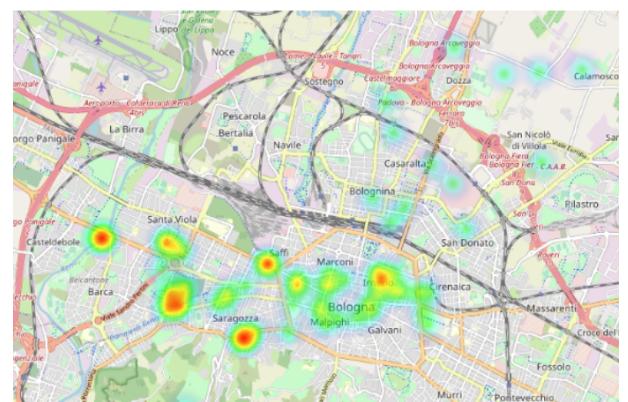


Figura 3.4: Heatmap

Positions

Alla richiesta della Homepage, e quindi all'url di base creato da ngrok, verrà richiamata la funzione index definita all'interno del file views.py che ha il compito di prelevare le misurazioni dal DB e inviarle al frontend così da poterle visualizzare su mappa. Per accedere alla nostra base di dati usiamo la libreria psycopg2 che permette di connettersi ad un database PostgreSQL tramite la funzione connect() al quale vanno passati come parametri gli elementi necessari alla connessione, quali: host, porta, nome del database, username e password dell'utente associato al database.

Dopodiché, sempre all'interno della funzione index, vado a definire il cursore con il quale sarò in grado di eseguire delle query. Fatto ciò prelevo tutti i dati dal database presenti nella tabella rilevazioni e li suddivido in tre array sulla base dell'algoritmo applicato alla rilevazione e li invio al template sotto forma di liste.

```
1 db = psycopg2.connect(host="127.0.0.1", port = 5432, database="progetto",
2                       user="manu", password="7894")
3
4 def index(request):
5     cur = db.cursor()
6
7     cur.execute("SELECT ST_AsGeoJson(posizione), rumore, timestamp, privacy
8                 FROM public.rilevazioni WHERE privacy='dummy' or privacy='no' or
9                 posizione in (SELECT posizione FROM rilevazioni where privacy='gpss'
10                  and reale=false)")
11    rilevazioni = cur.fetchall()
12
13    print(rilevazioni)
14
15    rilevazioni_gpss = []
16    rilevazioni_dummy = []
17    rilevazioni_noprivacy = []
18
19    for el in rilevazioni:
20        if el[3] == 'gpss':
21            rilevazioni_gpss.append(el)
22        elif el[3] == 'dummy':
```

```

19         rilevazioni_dummy.append(el)
20     else:
21         rilevazioni_noprivacy.append(el)
22
23     rilevazioni_gpfp = np.array(rilevazioni_gpfp).tolist()
24     rilevazioni_dummy = np.array(rilevazioni_dummy).tolist()
25     rilevazioni_noprivacy = np.array(rilevazioni_noprivacy).tolist()
26     cur.close()
27
28     return render(request, 'index.html', {'rilevazioni_gpfp':
29         rilevazioni_gpfp, 'rilevazioni_dummy':rilevazioni_dummy,
30         'rilevazioni_noprivacy':rilevazioni_noprivacy})

```

Nel template index.html salvo i dati inviati dal Backend al Frontend in degli array javascript che poi ci serviranno per costruire le features da andare ad inserire nei layer della mappa corrispondenti. Per la precisione abbiamo nove array, tre per ogni tabella, il primo in cui salvo le posizioni sotto forma di geojson, il secondo in cui salvo il rumore rilevato a quella posizione e il terzo in cui salvo il timestamp della misurazione².

```

1  {% for el in rilevazioni_gpfp %}
2
3      var pos = '{{ el.0 }}'.replace(/\"/g, '')
4      var pos = pos.replace("\\", "")
5      var rumore = '{{ el.1 }}'
6      var ts = '{{ el.2 }}'
7
8      json_pos_gpfp.push(pos)
9      rumori_gpfp.push(rumore)
10     timestamp_gpfp.push(ts)
11
12     {% endfor %}

```

²Per questioni di lunghezza del codice andremo a mostrare solo le righe relative ad uno dei tre algoritmi di privacy. Gli altri sono uguali, cambiano il nome delle variabili e i dati che andiamo ad inserirci.

Una volta costruiti gli array mi sposto nel file index.js e inzializzo la mappa sul template. In seguito costruisco i vari oggetti necessari per la visualizzazione dei punti sulla mappa e le features collection³, una per ogni tipo di algoritmo e una contenente tutte le rilevazioni che mi servirà per il clustering e l'heatmap.

```

1 var map = new ol.Map({
2   target: 'map',
3   layers: [],
4   view:
5     new ol.View({
6       center: ol.proj.fromLonLat([0, 0]),
7       zoom: 0
8     })
9 );
10
11 var osmLayer = new ol.layer.Tile({
12   source:
13     new ol.source.OSM({
14       })
15 });

```

```

1 var positions_gpssp = new ol.layer.Vector({
2   title: 'positions Layer',
3   source: urlsource_gpssp,
4   style: positionstyle_gpssp
5 });
6
7 positions_gpssp.setVisible(true)
8 map.addLayer(positions_gpssp);

```

Costruiti tutti i layer, li aggiungiamo alla mappa e li rendiamo visibili tramite i metodi setVisible() e addLayer(), tutti tranne il vettore contenente tutte le misurazioni presenti nel database.

³Negli snippet di codice riportiamo solo il codice relativo alla creazione del layer per le rilevazioni con GPS Perturbation, per gli altri layer il codice è uguale cambiano i nomi delle variabili e il contenuto delle features sulla base della tabella di provenienza della feature.

Dalla Homepage, con il radio-button "Positions" selezionato, sarà inoltre possibile andare a scegliere quali rilevazioni vedere sulla base dell'algoritmo di privacy usato per inviare tali misurazioni. È possibile farlo grazie alle checkbox posizionate sotto i radio-button, dal quale si può scegliere quali rilevazioni mostrare semplicemente scegliendo quali checkbox tenere attive tra GPS Perturbation, Dummy Updates e No Privacy⁴.

Il popup con le informazioni relative alla misurazione è stato implementato tramite la classe Overlay di Open Layers. Nel momento in cui l'utente clicca sulla mappa, viene controllato che ci siano delle features nel punto in cui ha premuto. In seguito controllo se le features sono una o più di una; se ne trovo solo una, prendo i dati relativi a quella misurazione e li inserisco in una stringa HTML che poi andrò ad immettere nel div all'interno nel popup, se ne trovo più di una, creo un array che poi scorrerò premendo i pulsanti "Next" e "Prev" presenti nel popup. Il codice è implementato nella funzione popupString(e, features, len), che prende come parametri l'evento che accade sulla mappa, ovvero il click, le features da mostrare nel popup e la lunghezza dell'array contenente le features in modo tale da costruire poi i diversi popup da scorrere con i bottoni Prev e Next.

```
1 const container = document.getElementById('popup');
2 const content = document.getElementById('popup-content');
3 const closer = document.getElementById('popup-closer');
4
5 var popup = new ol.Overlay({
6   element: container,
7   autoPan: true,
8   autoPanAnimation: {
9     duration: 250,
10   },
11 });
12
13 //popup closer
14 closer.onclick = function () {
```

⁴No Privacy corrisponde alle rilevazioni inviate dagli utenti che hanno scelto di non applicare nessun algoritmo di privacy alla propria posizione.

```
15     popup.setPosition(undefined);
16     closer.blur();
17     return false;
18   };
19 map.addOverlay(popup);
20
21 function popupPositions(e) {
22   const features = map.getFeaturesAtPixel(e.pixel)
23
24   if(features) {
25     if (features.length > 1) {
26       popupString(e, features, features.length)
27     }
28   else {
29     popupString(e, features, 1)
30   }
31 }
32 }
33
34 var singleclick_key = map.on('singleclick', popupPositions)
```

K-Means

Premendo sul bottone K-means nella mappa verranno riportate tutte le rilevazioni presenti nel database ognuna con un colore specifico, ovvero il colore del proprio cluster.

Allo slider creato da HTML viene aggiunto un event listener che al cambio del valore dello slider modificherà il numero di cluster da passare come parametro all'algoritmo nel Backend. Una volta avuto il valore dello slider faccio una chiamata POST all'url /kmeans_clustering/ che nel file urls.py corrisponde al metodo kmeans_c presente in views.py, inviando come dato il numero di cluster scelto dall'utente, che di default è 5. Abbiamo dovuto fare una chiamata POST perché Django non permette di comunicare direttamente con il Backend da javascript. Questo metodo ci restituirà come risposta un array dove ogni elemento corrisponde ad un dizionario contenente come valori: latitudine, longitudine e cluster di appartenenza.

La libreria usata per implementare il K-Means clustering è scikit-learn che tramite i metodi fit() e fit_predict() restituisce come valore di ritorno un array dove ogni elemento corrisponde ad un intero da 0 ad n-1, dove n è il numero di cluster scelto. Di conseguenza l'elemento alla posizione i-esima dell'array delle misurazioni da clusterizzare corrisponderà al cluster con valore nella posizione i-esima dell'array restituito dal metodo fit_predict().

```

1 coordinates = np.array(coordinates)
2 kmeans = KMeans(n_clusters=int(n))
3 kmeans.fit(coordinates)
4 clusters = kmeans.fit_predict(coordinates)
5
6 item_clusterized = []
7 for i in range(0,len(clusters)):
8     dict = {
9         "lat":coordinates[i][0],
10        "long":coordinates[i][1],
11        "cluster": str(clusters[i])
12    }
13    item_clusterized.append(dict)

```

Dopodiché creo una nuova feature collection per ogni cluster dove vado ad inserire le features corrispondenti all'i-esimo cluster, trovate confrontando la latitudine e longitudine dell'array di risposta con le coordinate delle rilevazioni. Aggiungo queste features collection ad un array che poi mi servirà per andare a creare ed eliminare i layer sulla mappa.

```

1 const nClusters = document.getElementById('n-clusters')
2 var colors = ['#FF0000', '#0000FF', '#00FF00', '#FFFF00', '#FF00FF', '#FFFFFF',
3               '#00FFFF', '#990099', '#009999', '#999900', '#999999']
4 var positions_kmeans_array = []
5 var n_clusters_val = $('#n_clusters_val')
6 n_clusters_val.html(nClusters.value)
7
8 nClusters.addEventListener('input', function () {
9     $('#n_clusters_val').html(nClusters.value)

```

```

9   kmeans_layer()
10 } );
11
12 function kmeans_layer() {
13   azzerate_clusters_layer()
14   create_clusters()
15 }
```

La funzione `azzerate_clusters_layer()` rimuove tutti i layer dei cluster creati in precedenza nel caso in cui fossero presenti, mentre la funzione `create_clusters()` ha il compito di invocare la chiamata POST e costruire i layer di ogni cluster sulla base del numero scelto tramite lo slider.

Clustering

Tramite la selezione del pulsante Clustering, sulla mappa verranno mostrate le posizioni delle rilevazioni raggruppate tra loro sulla base di tre valori:

- Zoom della mappa: lo zoom che si applica alla mappa, più saremo lontani e più i cluster tenderanno a diminuire, più saremo vicini e più i cluster aumenteranno;
- Cluster distance: impostata tramite l'apposito slider, rappresenta la distanza tra i cluster. Più aumenta più i cluster tenderanno ad aumentare, più diminuisce più il numero di cluster sarà minore;
- Minimum distance: la distanza minima tra i cluster.

Per l'implementazione ci siamo serviti dell'apposita classe `Cluster` di Open Layers al quale abbiamo passato come fonte la feature collection contenente tutte le rilevazioni ed in seguito abbiamo usato tale fonte per creare un nuovo layer che ha come source le posizioni clusterizzate.

```

1 const distanceInput = document.getElementById('distance');
2 const minDistanceInput = document.getElementById('min-distance');
3
4 const clusterSource = new ol.source.Cluster({
```

```
5   distance: parseInt(distanceInput.value, 10),
6   minDistance: parseInt(minDistanceInput.value, 10),
7   source: urlsource_all
8 } );
9
10 const styleCache = {};
11 const clusters = new ol.layer.Vector({
12   source: clusterSource,
13   style: function (feature) {
14     const size = feature.get('features').length;
15     let style = styleCache[size];
16     if (!style) {
17       style = new ol.style.Style({
18         image: new ol.style.Circle({
19           radius: 10,
20           stroke: new ol.style.Stroke({
21             color: '#00FFFF',
22             width: 1.25
23           }),
24           fill: new ol.style.Fill({
25             color: 'rgba(0,255,255,0.4)'
26           })
27         },
28         text: new ol.style.Text({
29           text: size.toString(),
30           fill: filltext
31         },
32       });
33       styleCache[size] = style;
34     }
35     return style;
36   },
37 }) ;
```

Heatmap

Come per la sottosezione precedente, è possibile costruire un layer sulla mappa rappresentante l'heatmap delle rilevazioni tramite un'apposita classe presente nelle API di Open Layers chiamata Heatmap.

La fonte delle rilevazioni è la feature collection contenente tutte le misurazioni presenti nel database e come peso sul quale andare ad aumentare o diminuire l'intensità del colore gli è stato passato il rumore delle rilevazioni.

```
1 const heatmap = new ol.layer.Heatmap({
2   source: urlsource_all,
3   blur: 15,
4   radius: 7.5,
5   weight: function (feature) {
6
7     const rumore = feature.get("Rumore")
8     return rumore;
9   },
10 }) ;
```

3.1.2 Previsioni

Il template "previsioni.html" è raggiungibile tramite il link "Forecasts" presente sotto il titolo nella Homepage.

Tale pagina riporta solo una mappa in cui sono state caricate delle misurazioni autogenerate tramite uno script python e inserite all'interno del file dataset.csv e due input text nel quale è possibile andare ad immettere rispettivamente latitudine e longitudine di un punto per avere una predizione del rumore cliccando sull'apposito bottone "Predict", la previsione del rumore verrà mostrata di fianco al bottone, figura 3.5.

È anche possibile prevedere il rumore corrispondente ad un punto cliccando sulla mappa la posizione nella quale si vuole prevedere l'inquinamento acustico; verrà aperto un popup che mostra la latitudine e la longitudine del punto cliccato e la previsione del rumore calcolata in quel punto, figura 3.6.

11.349138095522129	44.498468218405264	Predict	55.03670875183297
--------------------	--------------------	---------	-------------------

Figura 3.5: Previsione con input text

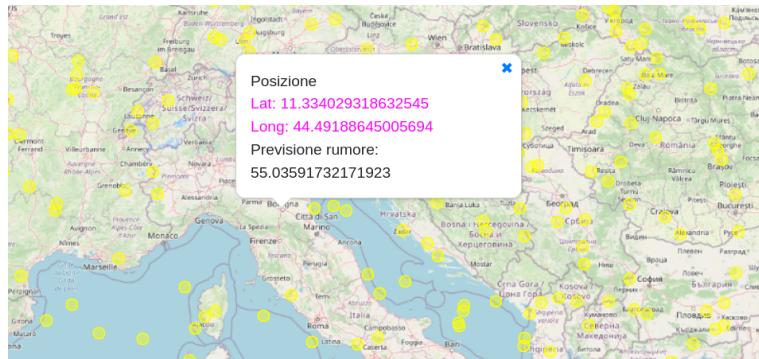


Figura 3.6: Previsione con mappa

Per ottenere questo risultato, come per il K-Means clustering, abbiamo dovuto implementare una richiesta POST all'url /previsione_rm/, dichiarato nel file urls.py, che, o al click sul bottone predict, o al click su una posizione nella mappa, inviasse le coordinate al Backend chiamando la funzione previsione_rm nel file views.py. Tale funzione applica il metodo di regressione lineare importato dalla libreria scikit-learn che restituirà il valore del rumore predetto.

La funzione predict_rm lavora in questo modo: una volta che dal Frontend vengono inviate le coordinate da cui predire il rumore, viene creato un oggetto di tipo LinearRegression() dal quale poi andremo ad usare due funzioni: fit(), al quale passerò le latitudini e longitudini presenti nel file dataset.csv e predict() al quale passerò le coordinate che mi ha inviato l'utente.

Per generare dei random state da passare alla funzione train_test_split(), che si occupa di dividere il dataset in colonne di training e colonne di testing, genero dei random state che vanno da 0 a 1500 con step di un numero random tra 5 e 15.

Il meccanismo di predizione si ripete per ogni elemento generato precedentemente come random seed dal quale ogni volta vado a calcolare l'errore quadratico medio tra i valori predetti e i valori reali sul dataset di testing e viene preso il valore predetto corrispondente alle coordinate inviate dall'utente che ha come risultato l'errore qua-

dratico medio minore tra tutte le possibili predizioni associate ai vari random state. Infine viene inviato il valore al Frontend che lo mostrerà all'utente sulla base di quale tra i due metodi di immissione ha scelto.

```

1 @csrf_exempt
2 def previsione_rm(request):
3     lat = request.POST.get("lat")
4     long = request.POST.get("long")
5     if lat == None or long == None or lat == "" or long == "":
6         return JsonResponse("None")
7
8     rm = calc_pred(float(lat), float(long))
9     return JsonResponse(rm)

```

Per la predizione da input text:

```

1 $("#predict").click( function() {
2     var lat = $('#lat_pred').val()
3     var long = $('#long_pred').val()
4
5     $.post("previsione_rm/", {"lat":lat, "long":long}, function (data) {
6         if (data == "None") {
7             $('#rm_predicted').html("Insert Lat and Long!")
8         } else {
9             $('#rm_predicted').html(data)
10        }
11    });
12 });

```

Per la predizione da mappa:

```

1 $.post("previsione_rm/", {"lat":hdms[0], "long":hdms[1]}, function (data)
2     {
3         console.log(hdms)
4         content.innerHTML = '<div id="popover-content"><span>Posizione</span><br>' +
5             '<span class="popupLatLon">Lat: ' + hdms[0] + '</span><br>' +

```

```
5      '<span class="popupLatLon">Long: ' + hdms[1] + '</span><br>' +
6      '<span>Previsione rumore: ' + data +'</span><br>' +
7      popup.setPosition(coordinate);
8  } );
9 }
```

3.1.3 API

In merito alle possibili richieste dell'applicazione, ci siamo serviti di sei API:

- postAVGNoiseGPSP;
- postAVGNoiseNoPrivacy;
- postAVGNoiseDummy;
- postInsertRilevazioniGPSP;
- postInsertRilevazioniNoPrivacy;
- postInsertRilevazioniDUMMY.

Tutte le api sono contenute nel file views.py all'interno della cartella api.

Per visualizzare e testare le api presenti nel sito abbiamo utilizzato uno strumento chiamato Swagger che, tramite la sua installazione con pip e l'importazione del file HTML nella cartella template, da la possibilità di aggiungere una pagina web raggiungibile inserendo "swagger/" di seguito al dominio del sito web, figura 3.7.

postAVGNoiseGPSP

Questa API si occupa di inviare all'applicazione il rumore medio rilevato nel raggio di tre chilometri nel caso in cui l'utente stia usando l'algoritmo di privacy GPS Perturbation. Oltre al rumore medio si occupa di inviare la Quality of Service relativa alla posizione vera e falsa ricevuta.

Come dati in input prende un geojson inviato tramite richiesta POST, contenente due features sotto forma di features collection: una con la posizione reale e l'altra

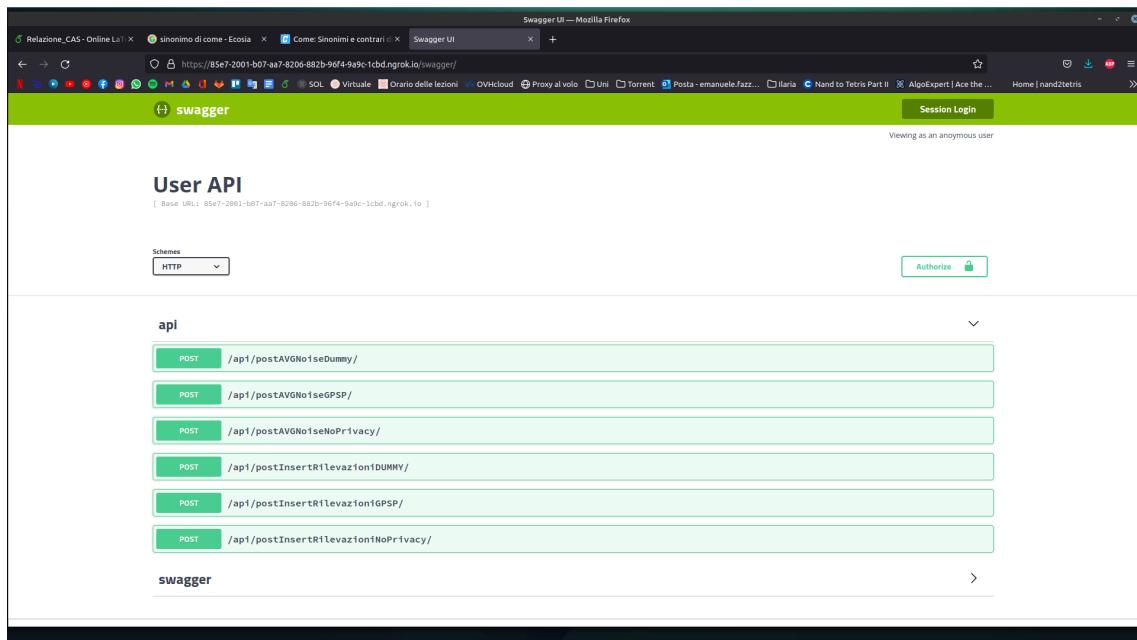


Figura 3.7: Swagger API

con la posizione perturbata. Per riconoscerle abbiamo inserito nelle properties della feature un oggetto chiamato "Reale" al quale assegnamo il valore true, nel caso sia la vera posizione dell'utente, false altrimenti⁵.

Una volta ricavata la geometry del punto con l'oggetto "Reale" uguale a false, richiamo la funzione rumore_medio_gpsp_noprivacy(), sempre dichiarata all'interno del file views.py, che ha come parametro la geometry prima descritta, la quale ha il compito di calcolare il rumore medio su tutte le rilevazioni presenti nel database nel raggio di tre chilometri dalla posizione passata come parametro alla funzione.

Oltre al rumore medio calcolo anche la qualità del servizio sotto forma di errore quadratrico medio che servirà poi all'applicazione per mostrare all'utente le statistiche necessarie.

Per calcolare la QoS mi servo del metodo qosGPSP()⁶ dichiarato nel file qos.py e importato nel file views.py che prende come parametri la posizione vera e la posizio-

⁵Il valore reale ci serve per calcolare la Quality of Service della piattaforma

⁶In seguito non andremo ad inserire il codice relativo al calcolo del qos per questioni di lunghezza, si consiglia di guardare direttamente il file qos.py presente nella cartella api.

ne perturbata dell’utente in formato GeoJSON. Questo metodo semplicemente calcola tramite query il rumore medio delle posizioni nel raggio di tre chilometri dalle coordinate inviate come false prendendo tutte le posizioni con algoritmo Dummy Updates, tutte le posizioni senza algoritmo di privacy e tutte le posizioni con algoritmo GPS Perturbation con il valore reale uguale a false; dopodiché calcola il rumore medio nel raggio di tre chilometri dalle coordinate inviate come reali prendendo però solo le posizioni che hanno valore reale uguale a true nel database. Infine, se nessuno dei due valori risulta null e quindi ci sono delle posizioni su cui calcolare il rumore medio, sottraggo la seconda alla prima ed elevo al quadrato. Il risultato sarà il valore della nostra QoS.

```

1 def rumore_medio_gpssp_noprivacy(geojson):
2
3     cur = db.cursor()
4     rmedio = None
5
6     try:
7         query = "SELECT AVG(rumore) FROM public.rilevazioni WHERE privacy
8             ='dummy' or privacy='no' or posizione in (SELECT posizione FROM
9             rilevazioni where privacy='gpssp' and reale=false) AND ST_DWithin(public
10             .rilevazioni.posizione::geography,ST_GeomFromGeoJSON('"+geojson+"'')::
11             geography,3000);"
12
13         cur.execute(str(query))
14         db.commit()
15         rm = cur.fetchall()[0][0]
16
17         if rm != None :
18             rmedio = rm
19
20     except:
21         db.rollback()
22
23
24     return rmedio

```

```

1 @api_view(['POST'])
2 def postAVGNoiseGPSP(request):
3     gj = request.data.get('geojson')
4     geojson = json.loads(gj)

```

```

5
6     point_t = ""
7     point_f = ""
8
9     for el in geojson["features"]:
10        print("el: ",el)
11        reale = el.get("properties").get("Reale")
12        if reale == "true":
13            point_t = str(el.get("geometry")).replace("/", "\\" )
14        else:
15            point_f = str(el.get("geometry")).replace("/", "\\" )
16
17    rmedio = rumore_medio_gpfp_noprivacy(point_f)
18    qos = qosGPSP(point_f,point_t)
19    #invia anche il qos
20    data = { "rumore_medio": rmedio, "qos":qos }
21
22    return Response(data, status=status.HTTP_200_OK)

```

postAVGNoiseNoPrivacy

Tale API, come la precedente, ritorna il rumore medio e la QoS all’utente nel caso in cui lui non stia usando nessun algoritmo di privacy.

Il meccanismo è molto simile all’API postAVGNoiseGPGP, per il calcolo del rumore medio usiamo di nuovo il metodo rumore_medio_gpfp_noprivacy, passandogli come parametro la posizione reale dell’utente.

Per il calcolo della QoS, invece mi servo del metodo qosNoP() che prende come parametro la posizione inviata dall’app. In seguito esegue due query, una che ritorna il rumore medio considerando tutte le posizioni false nel raggio di tre chilometri e l’altra considerando solo le posizioni vere sempre nel raggio di tre chilometri. Sottraggo il secondo valore al primo e ne calcolo il quadrato se entrambi non risultano nulli. Infine sarà restituito questo risultato come qualità del servizio.

```

1 @api_view(['POST'])
2 def postAVGNoiseNoPrivacy(request):

```

```
3     geojson = request.data.get('geojson')

4

5     rmedio = rumore_medio_gpfp_noprivacy(geojson)
6     #invia anche il qos
7     qos = qosNoP(geojson)
8     data = { "rumore_medio": rmedio, "qos": qos }

9

10    return Response(data, status=status.HTTP_200_OK)
```

postAVGNoiseDummy

Come le altre due API, ritorna il rumore medio nel raggio di tre chilometri, solo che il rumore medio viene calcolato per ogni posizione inviata dall'applicazione in modo tale che solo l'app saprà quali sono le coordinate realmente corrispondenti alla posizione dell'utente e potrà ricostruire in modo esatto il rumore medio associato alla reale posizione. In più ritorna la qualità del servizio calcolata su tutte le posizioni Dummy inviate dall'app in quel momento.

Per quanto riguarda il rumore medio, nel json di risposta vengono costruiti due array, rumore_medio e posizione, dove l'elemento alla i-esima posizione nell'array rumore_medio corrisponderà al rumore medio nella posizione i-esima dell'array posizione. La costruzione del json di risposta è affidata alla funzione rumore_medio_dummy() che prende in input il geojson con all'interno tutte le features corrispondenti alle posizioni Dummy inviate dall'app e ritornerà il json costruito come descritto precedentemente. In più questa funzione aggiunge al json di risposta anche la QoS calcolata dalla funzione qosDummy() sempre nel file qos.py.

Questa funzione calcola il rumore medio delle posizioni reali nel raggio di tre chilometri della posizione contrassegnata con il valore "Reale" uguale a true e lo salva in una variabile. Dopodiché per ogni posizione falsa calcola il rumore medio nel raggio di tre chilometri considerando tutte le posizioni con valore privacy nel database uguale a 'dummy' e 'no' e le posizioni con valore privacy uguale a 'gpfp' ma con valore reale uguale a false. Per ogni posizione falsa inviata dall'app aggiungo il rumore medio calcolato come appena descritto in un array e, infine, per ogni posizione dell'array calcolerò l'errore quadratico medio sottraendo ad ogni rumore medio derivato

dalle posizioni con valore "Reale" uguale a false nel geojson il rumore medio risultato dal calcolo con posizione "Reale" uguale a true. Elevo ogni risultato delle sottrazioni al quadrato e li sommo fra di loro, infine divido per il numero di sottrazioni eseguite. la cifra in output dal calcolo appena descritto sarà la qualità del servizio associata alla posizione dell'utente.

```
1 def rumore_medio_dummy(geojson):
2     qos = None
3     data = { "rumore_medio": [], "posizione": [], "qos":qos}
4
5     cur = db.cursor()
6
7     features = geojson.get("features")
8
9     try:
10         for el in features:
11             query = "SELECT AVG(rumore) FROM public.rilevazioni WHERE
privacy='dummy' or privacy='no' or posizione in (SELECT posizione FROM
rilevazioni where privacy='gpsp' and reale=false) AND ST_DWithin(public.
.rilevazioni.posizione::geography,ST_GeomFromGeoJSON('"+str(el.get("
geometry"))).replace("'", "\")+"')::geography, 3000);"
12             cur.execute(str(query))
13             db.commit()
14             rm = cur.fetchall()[0][0]
15             if rm != None:
16                 data.get("rumore_medio").append(rm)
17                 data.get("posizione").append(el.get("geometry").get("coordinates"))
18             else:
19                 data.get("rumore_medio").append(None)
20                 data.get("posizione").append(el.get("geometry").get("coordinates"))
21     except:
22         db.rollback()
23
24     data["qos"] = qosDummy(geojson)
25     cur.close()
```

```
26
27     return data

1 @api_view(['POST'])
2 def postAVGNoiseDummy(request):
3
4     gj = request.data.get('geojson')
5     geojson = json.loads(gj)
6
7     data = rumore_medio_dummy(geojson)
8
9     return Response(data, status=status.HTTP_200_OK)
```

Le ultime tre API si occupano di inserire la rilevazione inviata dall'utente nel database.

postInsertRilevazioniGPSP

Richiamando questa API, l'applicazione manderà come valore il geojson contenente le rilevazioni da inserire, una reale corrispondente alla vera posizione dell'utente e una falsa corrispondente alla posizione perturbata.

Creo un for per l'array features contenenti le due posizioni prima descritte e controllo se il valore "Reale" nel json sia uguale a false. In quel caso calcoliamo il rumore medio che poi andrà a restituire insieme al QoS calcolato su entrambi i punti e infine lo vado ad inserire nel database tramite una INSERT.

```
1 @api_view(['POST'])
2 def postInsertRilevazioniGPSP(request):
3
4     gj = request.data.get('geojson')
5     geojson = json.loads(gj)
6     rmedio = 0.0
7     cur = db.cursor()
8     #ce ne sono due, una posizione perturbata e una reale
9     features = geojson.get("features")
10    point_t = "
```

```
11 point_f = ""
12
13 for el in features:
14     p = el["geometry"]
15     pt = str(p).replace("'", "\\'")
16     point = str(pt)
17     properties = el["properties"]
18     rumore = properties["Rumore"]
19     bool_reale = properties["Reale"]
20     if bool_reale == "false":
21         rmedio = rumore_medio_gpfp_noprivacy(point)
22         point_f = point
23     else:
24         point_t = point
25         dt = datetime.now() + timedelta(hours=2)
26         date = dt.strftime("%d/%m/%Y")
27         ora = dt.strftime("%H:%M:%S")
28         timestamp = date + " " + ora
29         query = "INSERT INTO rilevazioni (posizione,rumore,timestamp,reale,
30 ,privacy) VALUES (ST_GeomFromGeoJSON('"+point+"'),"+str(rumore)+",
31 TIMESTAMP '" +timestamp+"','"+bool_reale+"','gpfp')"
32         print(query)
33         cur.execute(str(query))
34         db.commit()
35
36         cur.close()
37         qos = qosGPSP(point_f,point_t)
38         data = {
39             "rumore_medio": rmedio,
40             "qos": qos
41         }
42         return Response(data, status=status.HTTP_201_CREATED)
```

postInsertRilevazioniNoPrivacy

Tale API ha lo stesso meccanismo della precedente, solo che non mi viene inviata una feature collection con più punti ma solo la feature del punto corrispondente alla

posizione dell’utente. In questo modo non devo scorrere nessun array e posso andare ad inserire direttamente la nuova rilevazione nel database.

Come nel caso precedente, calcolo il rumore medio e la qualità del servizio e li restituisco in formato json.

```
1 @api_view(['POST'])
2 def postInsertRilevazioniNoPrivacy(request):
3
4     gj = request.data.get('geojson')
5     geojson = json.loads(gj)
6
7     cur = db.cursor()
8
9     p = geojson["geometry"]
10    pt = str(p).replace("'", "\\'")
11    point = str(pt)
12    rmedio = rumore_medio_gpfp_noprivacy(point)
13    properties = geojson["properties"]
14    rumore = properties["Rumore"]
15    dt = datetime.now() + timedelta(hours=2)
16    date = dt.strftime("%d/%m/%Y")
17    ora = dt.strftime("%H:%M:%S")
18    timestamp = date + " " + ora
19    query = "INSERT INTO rilevazioni (posizione,rumore,timestamp,reale,
20    privacy) VALUES (ST_GeomFromGeoJSON('"+point+"'),"+str(rumore)+",
21    TIMESTAMP '"+timestamp+"', true, 'no')"
22    print(query)
23    cur.execute(str(query))
24    db.commit()
25    qos = qosNoP(point)
26    data = {
27        "rumore_medio": rmedio,
28        "qos": qos
29    }
30
31    return Response(data, status=status.HTTP_201_CREATED)
```

postInsertRilevazioniDUMMY

Il procedimento di quest'ultima API è simile alle precedenti. Come valore di ritorno restituisce il rumore medio e il QoS calcolati, però per l'algoritmo Dummy Updates, spiegati nella sezione postAVGNoiseDummy. Una volta ottenuto il geojson dall'applicazione, eseguirà un comando di INSERT per ogni misurazione presente all'interno della feature collection con i relativi dati.

```
1 @api_view(['POST'])
2 def postInsertRilevazioniDUMMY(request):
3
4     gj = request.data.get('geojson')
5     geojson = json.loads(gj)
6     data = rumore_medio_dummy(geojson)
7     cur = db.cursor()
8
9     #ce ne sono due, una posizione perturbata e una reale
10    features = geojson.get("features")
11
12    for el in features:
13        p = el["geometry"]
14        pt = str(p).replace("'", "\\'")
15        point = str(pt)
16        properties = el["properties"]
17        rumore = properties["Rumore"]
18        bool_reale = properties["Reale"]
19        dt = datetime.now() + timedelta(hours=2)
20        date = dt.strftime("%d/%m/%Y")
21        ora = dt.strftime("%H:%M:%S")
22        timestamp = date + " " + ora
23        query = "INSERT INTO rilevazioni (posizione,rumore,timestamp,reale,privacy) VALUES (ST_GeomFromGeoJSON('"+point+"'),"+str(rumore)+",
24        TIMESTAMP '"+timestamp+"','"+bool_reale+"','dummy')"
25        print(query)
26        cur.execute(str(query))
27        db.commit()
```

```
28     return Response(data, status=status.HTTP_201_CREATED)
```

3.2 Applicazione

Al primo avvio dell'applicazione viene chiesto all'utente di scegliere uno tra gli algoritmi di privacy offerti dalla piattaforma, che di default è nessuna privacy, e l'intervallo di attesa tra l'invio di una misurazione e l'altra, che di default è 20 secondi. I valori di default sono salvati nelle Shared Preferences dell'applicazione che, nel momento in cui l'utente andasse a cambiare le proprie preferenze, verranno modificate in modo da poter adattare il comportamento delle richieste al server sulla base della privacy scelta e dell'intervallo di attesa per l'invio di una nuova rilevazione.

Ciò appena descritto è implementato nel file `SettingsActivity.java` nel quale troviamo diverse funzioni:

- `modificaGrafica()`: prende come parametro un oggetto di tipo `RadioButton` e controlla che sia corrispondente all'oggetto `RadioButton` equivalente a `GPS Perturbation` e, in tal caso, verrà visualizzato anche uno slider in cui l'utente può decidere un valore `alpha` che corrisponderà alla priorità da dare alla QoS, altrimenti questa funzione verrà richiamata per nascondere tale slider;
- `seebar()`: funzione per far apparire lo slider sopra citato alla scelta del `RadioButton GPS Perturbation` e, inoltre, visualizzare il valore associato ad esso nella `TextView` sottostante;
- `checkButton()`: prende come parametro una `View` e modifica il valore della privacy impostato nelle Shared Preferences al cambiamento del `RadioButton`;
- `creaAlert()`: si occupa di creare l'`AlertDialog` al push sul bottone "Modifica Intervallo" per andare a cambiare il tempo di attesa tra l'invio di una rilevazione e l'altra;
- `onCreate()`: la funzione nel quale sono inizializzate tutte le `View` e i loro comportamenti all'interno della schermata.

Nella cartella res è presente il file XML associato alla schermata chiamato activity_settings.xml. La struttura della schermata è visibile in figura 3.8.

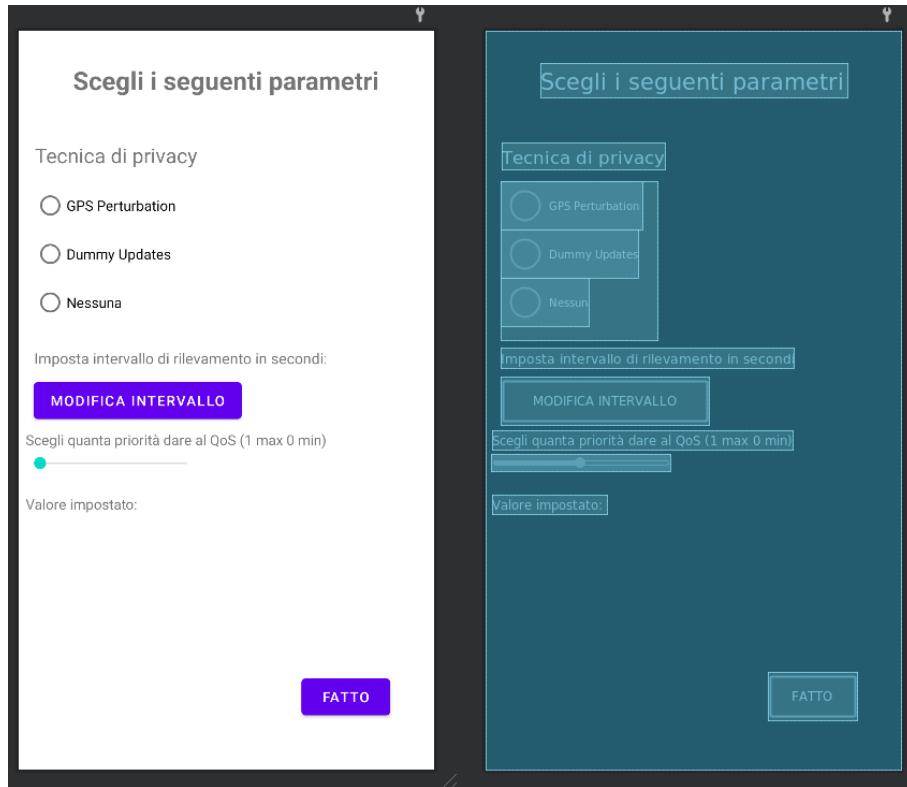


Figura 3.8: Settings XML

Premendo sul pulsante "FATTO" verrà attivato un Intent che ci reindirizzerà alla MainActivity.

Tale schermata è così strutturata: in alto troviamo due TextView, la prima rappresenta il rumore medio rilevato nell'area di tre chilometri dalla posizione dell'utente, la seconda il rumore rilevato dal dispositivo nel momento in cui un utente sceglie di contribuire alle misurazioni, che si aggiorna ad ogni misurazione effettuata. Subito sotto troviamo uno switch che permetterà all'utente, se attivato, di inviare nuove misurazioni, una dopo l'altra con un intervallo deciso sulla base dei secondi riportati nei Settings. Successivamente vi sono 4 pulsanti, i primi tre per richiedere le metriche rispettivamente di: qualità del servizio, privacy e trade-off tra privacy e QoS, mentre l'ultimo per aggiornare il rumore medio ogni volta che lo si desidera.

Nella cartella res troviamo il file activity_main.xml che riporta la struttura della schermata, mostrata in figura 3.9

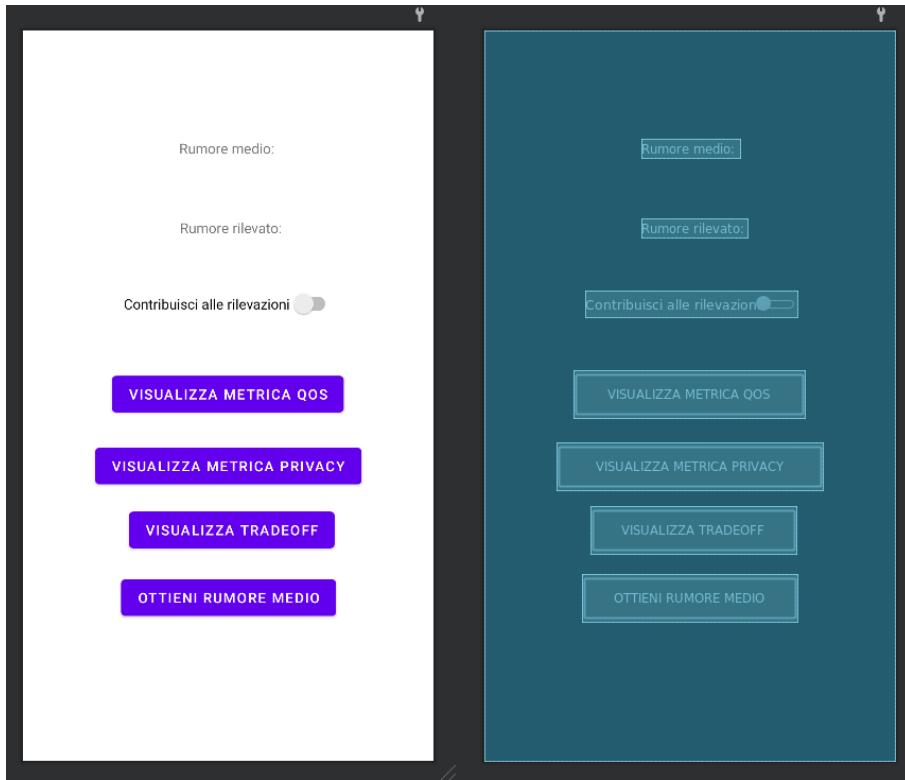


Figura 3.9: Main XML

Il rumore medio può essere aggiornato premendo sul pulsante "OTTIENI RUMORE MEDIO" in fondo alla schermata. È ottenuto tramite una chiamata POST alle API sulla base di quale algoritmo di privacy si sta servendo l'utente. Per controllare questo parametro accediamo alle Shared Preferences e verifichiamo il valore associato alla privacy. In seguito, sulla base di questo valore, uno switch case costruirà il GeoJSON corrispondente all'algoritmo di privacy in uso. Infine verrà richiamato il metodo postRequest() che prende come parametro il GeoJSON creato sulla base delle preferenze di privacy e invierà la richiesta HTTP per ricevere i valori desiderati.

Ottenuti i valori di ritorno li visualizziamo nelle apposite sezioni: il rumore medio nella TextView, tramite il metodo setText(), e la QoS premendo sul bottone "OTTIENI METRICA QOS", mostrando un Toast con il valore.

Per l'invio di richieste HTTP utilizziamo Volley, una libreria che permette di eseguire delle richieste verso degli indirizzi web. Tramite l'oggetto StringRequest creiamo la richiesta del rumore medio all'URL che rappresenta l'API dal quale riceveremo come risposta un altro oggetto di tipo JSON contenente i valori del rumore medio e della QoS. Come tali valori sono calcolati viene descritto nell'implementazione della sezione Web della piattaforma. La risposta è gestita dal metodo onResponse() dell'oggetto StringRequest che si occuperà di far visualizzare all'utente i dati ricevuti dalle richieste⁷. Dopo aver impostato l'inoltro della richiesta, la aggiungiamo alla RequestQueue di Volly così che possa essere inviata.

```
1 RequestQueue requestQueue=Volley.newRequestQueue(MainActivity.this);
2 StringRequest stringRequest=new StringRequest(Request.Method.POST, url,
3         new com.android.volley.Response.Listener<String>() {
4             @Override
5             public void onResponse(String response) {
6                 JSONObject obj = null;
7                 double res = 0;
8                 switch (p) {
9                     case ("Nessuna"):
10                         try {
11                             obj = new JSONObject(response);
12                             res = obj.getDouble("rumore_medio");
13                             qosN=obj.getDouble("qos");
14                         } catch (JSONException e) {
15                             e.printStackTrace();
16                         }
17                         Log.v("R", "VAL R:" + res);
18                         if(res== -1){
19                             RM.setText("Non ci sono abbastanza rilevazioni per
calcolare il rumore medio");
20                         }
21                         else{
22                             RM.setText("Rumore medio: " + res);
23                         }
24                 }
25             }
26         });
27         requestQueue.add(stringRequest);
```

⁷Per motivi di ampiezza del codice riporteremo solo una risposta alla richiesta HTTP. È possibile visualizzare le altre nel file MainActivity.java

```
23         break;  
1 requestQueue.add(stringRequest);
```

In merito alla metrica di privacy, invece, ogni volta che: o viene richiesto il valore medio, o viene inviata una nuova rilevazione, viene calcolata sulla base di quale algoritmo di privacy si sta usando. Per il GPS Perturbation facciamo semplicemente la differenza tra il punto falso e il punto reale, mentre per i Dummy Updates calcoliamo la somma delle distanze tra ogni dummy e la posizione reale e poi dividiamo per il numero di dummy, in modo tale da ottenere una media.

Di questo se ne occupano le due funzioni mPrivacyD e mPrivacyP, implementate qui di seguito.

```
1 public float mPrivacyD(ArrayList<Location> l) {  
2     float distanza=0;  
3     for(int i=0;i<l.size();i++) {  
4         if(i!=reale){  
5             distanza+=l.get(reale).distanceTo(l.get(i));  
6         }  
7     }  
8     distanza/=l.size()-1;  
9     return distanza;  
10 }  
11  
12 public float mPrivacyP(Location lf, Location l) {  
13     float distanza=l.distanceTo(lf);  
14     return distanza;  
15 }
```

La metrica di trade-off tra QoS e privacy è una semplice formula basta sull'alpha scelto dall'utente nelle impostazioni. La formula è la seguente:

$$(\alpha * QoS) + (1 - \alpha) * Privacy$$

L'obiettivo è di massimizzare il risultato perché in questo modo si comunica all'utente che più la privacy sarà elevata e più la qualità del servizio diminuirà.

```
1 m.setOnClickListener(new View.OnClickListener() {
2     @Override
3     public void onClick(View view) {
4         p= sharedpreferences.getString("Privacy", "Nessuna");
5         if(p.equals("GPS Perturbation")) {
6             double tradeoff= ((fgp*qosP)+((1-fgp)*distanzaP));
7             Toast.makeText(getApplicationContext(),"Metrica Tradeoff: "+
tradeoff,Toast.LENGTH_LONG).show();
8         }
9         else{
10             Toast.makeText(getApplicationContext(),"Questa metrica
visualizzabile solo per GPSP",Toast.LENGTH_LONG).show();
11         }
12     }
13 } );
```

3.2.1 Algoritmi di privacy

GPS Perturbation

La perturbazione da applicare alla posizione reale dell'utente viene calcolata sulla base dell'alpha scelto nelle impostazioni, più l'alpha sarà alto più cifre dopo la virgola verranno prese fino ad arrivare all'ultima cifra decimale se $\alpha = 1$, più l'alpha sarà basso meno cifre decimali saranno scelte fino ad arrivare a prendere solo la parte intera della latitudine e longitudine se $\alpha = 0$.

La modifica delle coordinate viene eseguita ogni qual volta l'applicazione deve comunicare con il backend, quindi quando si vuole richiedere il rumore medio o quando si vuole inserire una nuova rilevazione.

Nel codice viene realizzata in uno switch case al momento della creazione del json; viene letta la metodologia di privacy scelta dall'utente dalle Shared Preferences e se corrisponde a "GPS Perturbation" calcoleremo la proporzione per vedere quante cifre decimali tenere nelle coordinate rispettivamente al valore dell' α .

```
1     double latF;
```

```
2  double longif;
3  String lattext = Double.toString(Math.abs(lat));
4  String longitext=Double.toString(Math.abs(longi));
5  int latintegerPlaces = lattext.indexOf('.');
6  int nlat = lattext.length() - latintegerPlaces - 1;
7  int longiintegerPlaces = longitext.indexOf('.');
8  int nlongi = longitext.length() - longiintegerPlaces - 1;
9  fgp=Math.round(sharedPreferences.getFloat("alfa",0)*10);
10 fgp/=10;
11 float gpLat=Math.round(nlat*fgp);
12 float gpLongi=Math.round(nlongi*fgp);
13 latF=lat*Math.pow(10,gpLat);
14 latF=Math.floor(latF);
15 latF=latF/Math.pow(10,gpLat);
16 longiF=longi*Math.pow(10,gpLongi);
17 longiF=Math.floor(longiF);
18 longiF=longiF/Math.pow(10,gpLongi);
19 Location lF = new Location(LocationManager.GPS_PROVIDER);
20 lF.setLatitude(latF);
21 lF.setLongitude(longiF);
```

Dummy Updates

Non avendo un valore alpha scelto dall'utente da applicare all'algoritmo Dummy Updates, da codice si può scegliere il numero di dummy da inviare insieme alla posizione reale, che nel nostro caso è 5.

Per generare queste 5 posizioni dummy abbiamo costruito la funzione randPos() che prende come parametri la latitudine e longitudine della posizione reale, il raggio in metri nel quale generare una posizione dummy, che di default è 3 chilometri, e il numero di posizioni dummy da generare. Infine restituirà un ArrayList contenente tutte le posizioni dummy precedentemente generate. A questo ArrayList andremo ad aggiungere la posizione reale ed infine costruiremo il GeoJSON necessario al richiamo delle API.

```
1 public static ArrayList<Location> randPos(double x0, double y0, int radius
2   ,int rip) {
3   Random random = new Random();
4   ArrayList<Location> l=new ArrayList<>();
5   double radiusInDegrees = radius / 111000f;
6   for(int i=0;i< rip;i++){
7     double u = random.nextDouble();
8     double v = random.nextDouble();
9     double w = radiusInDegrees * Math.sqrt(u);
10    double t = 2 * Math.PI * v;
11    double x = w * Math.cos(t);
12    double y = w * Math.sin(t);
13    double foundLongitude = new_x + x0;
14    double foundLatitude = y + y0;
15    Location lm=new Location(LocationManager.GPS_PROVIDER);
16    lm.setLongitude(foundLongitude);
17    lm.setLatitude(foundLatitude);
18    l.add(lm);
19  }
20  return l;
```

3.2.2 Invio di una nuova rilevazione

Come ultimo elemento da discutere nell'implementazione dell'applicazione è rimasto l'invio di una nuova rilevazione da parte degli utenti.

Questo task è svolto dall'Intent Service Rilvevazione che si occupa sia della registrazione dei rumori tramite il microfono del dispositivo sia della creazione del GeoJSON che sarà inviato alla piattaforma per salvare la nuova rilevazione nel database.

Per quanto riguarda la creazione del GeoJSON da inviare poi al server, il meccanismo è uguale alla realizzazione del GeoJSON per richiedere i dati relativi al rumore medio e alla qualità del servizio; l'unica differenza è che nelle "properties" di ogni feature viene aggiunto un altro oggetto chiave-valore con il rumore misurato dal dispositivo. Le API invocate iniziano con postInsertRilevazione dove cambia la parte finale sulla base delle preferenze di privacy scelte.

Relativamente al rilevamento del rumore, viene usato un oggetto TimerTask chiamato RecorderTask dichiarato all'interno della classe Rilevazione, che ha il compito di attivare il microfono, percepire l'inquinamento acustico e convertirlo in decibel. Il risultato sarà poi assegnato ad una variabile che verrà inserita nel JSON da inviare sotto la chiave "Rumore". Tale TimerTask viene avviato nella funzione start() che sarà poi richiamata dall'onHandleIntent() ogni volta che l'applicazione dovrà creare una nuova rilevazione.

```
1 private class RecorderTask extends TimerTask {
2     private MediaRecorder recorder;
3     public RecorderTask(MediaRecorder recorder) {
4         this.recorder = recorder;
5     }
6     @Override
7     public void run() {
8         if(shouldStop) {
9             return;
10        }
11        valRumore = recorder.getMaxAmplitude();
12        amplitudeDb = 20 * Math.log10((double)Math.abs(valRumore));
13        Log.v("RUMORE F:", "R: "+amplitudeDb);
14    }
15 }
```

Risultati

4.1 Quality of Service

Non disponendo di abbastanza utenza per avere un numero consistente di misurazioni su cui poi andare a fare i test, abbiamo deciso di creare rilevazioni casuali tramite uno script Python. Tale script ha inserito duecento misurazioni per ogni tipo di algoritmo offerto dalla piattaforma: GPS Perturbation, perturbando latitudine e longitudine lasciandole con tre cifre decimali, Dummy Updates, inviando sei misurazioni dummy generate randomicamente nel raggio di 3 km dalla posizione reale e senza privacy ed ha associato un rumore a ciascuna rilevazione tramite una funzione random che genera una cifra decimale compresa tra i 40 e gli 80 decibel. Il risultato è mostrato in figura 4.1.

Dopodiché siamo andati a calcolare la qualità del servizio su 150 rilevazioni prese casualmente, 50 per i record con privacy 'gpsp', 50 per i record con privacy 'dummy' e 50 per i record con privacy 'no'.

Infine per ognuno dei gruppi da 50 misurazioni abbiamo calcolato la media e la mediana:

Algoritmi	Media	Mediana	Differenza Media-Mediana
GPSP	16.397156082473533	14.952077988250059	1.445078094223474
Dummy	18.123851409386006	17.28050977862158	0.8433416307644244
NoP	16.845342758416095	15.44985351030651	1.395489248109584

Come possiamo notare dalla media, il rumore medio derivato dai valori reali e dai valori falsi non discosta di molto, con differenze che vanno tra i 3 e i 5 decibel. Questi dati riportati sono i dati risultanti dall'ultimo test eseguito, ma ne abbiamo fatti degli altri per vedere se ci fossero dei cambiamenti, riportati nel file statistics.csv; ad ogni test sia la media che la mediana erano comprese in un intervallo che va da 12 a 19. Il codice relativo ai test per i risultati si trova nella cartella results_test.

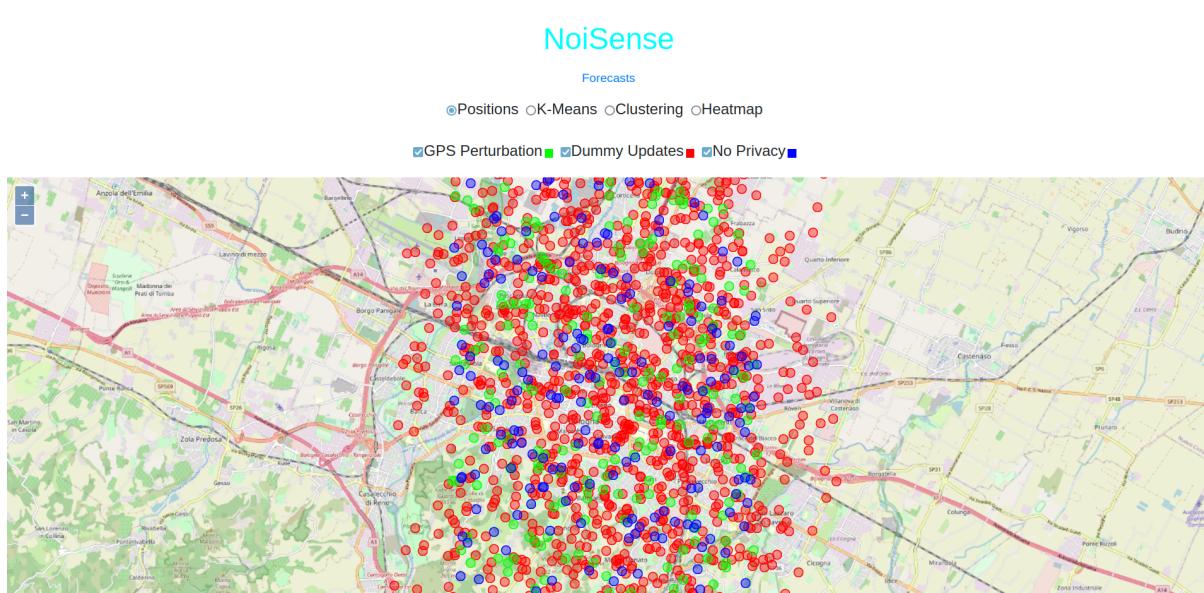


Figura 4.1: Rilevazioni autogenerate

Abbiamo voluto inserire anche la mediana tra le statistiche per dimostrare che non c'è molta differenza tra le due e quindi non ci sono valori outliers che vanno ad intaccare il calcolo della media.

4.2 Previsioni

Abbiamo scelto la regressione lineare dopo aver fatto svariate prove su quale tecnica di predizione ci desse il miglior errore quadratico medio. Queste prove possono essere trovate sotto forma di commento all'interno del file `previsioni.py` situato nella cartella `results_test`.

Il metodo di regressione lineare è stato confrontato con:

- Random Forest Classifier;
- Naive-Bayes Classifier;
- Perceptron Classifier;
- Random Forest Regressor.

Per i primi tre classificatori abbiamo dovuto trasformare i valori da double ad interi perché non gestiscono i valori in virgola mobile.

L'errore quadratico medio risultante da ognuno di essi rientrava circa nello stesso intervallo, tra il 69 e il 72, ma facendoli eseguire per 500 volte il metodo di regressione lineare ha raggiunto per più volte rispetto agli altri l'errore quadratico medio più piccolo. Per questo motivo abbiamo scelto tale metodo per la predizione di rumore nelle aree con misurazioni limitate o assenti.

Nella tabella mostrata qui di seguito mostreremo le coordinate su cui è stata fatta la predizione, il rumore risultato dal metodo di regressione e l'errore quadratico medio calcolato per ognuna delle predizioni.

Lat	Long	Rumore	MSE
53.447142	-100.158375	54.89956275	71.66665702711192
34.233962	-88.982615	54.9419455	70.3458244682766
21.034519	14.541595	55.01871533	71.15342157914263
-17.242522	-114.216142	54.87314612	71.92874374673643
-22.198573	-124.547951	54.90249373	71.50306268746557
33.297436	31.087235	55.01513027	70.96047774324116
-12.035044	71.116167	55.13120716	71.90351676857924
-35.812702	-68.198926	54.86440973	70.3458244682766
27.646261	-66.07144	54.95853953	70.3458244682766
59.382415	46.686352	55.12754161	71.51319821269692
37.07071	-65.9245	54.91931205	72.0154077972724
-51.350355	90.316938	55.08384185	71.45435590642103
44.174691	124.368058	55.17801623	71.51319821269692
3.552894	-52.680304	54.92080925	72.47732198116582
37.72834	-92.958463	54.87967311	71.15342157914263
-24.517686	-37.818717	54.90621462	71.15342157914263
15.572329	-35.449955	54.91162821	72.34890087961415
35.985526	53.839804	55.12940468	71.45435590642103
-3.523045	103.393581	55.13345526	71.50306268746557
2.635188	106.412379	55.12753998	71.65479797189283

Lat	Long	Rumore	MSE
55.811078	-51.116744	54.96016536	71.45435590642103
37.66134	-128.769701	54.89022307	69.66001197310828
-17.595902	-105.193478	54.84592435	72.06432513947078
-55.267682	-73.991572	54.93325336	70.96047774324116
47.352321	-95.290905	54.93998179	70.04846360254686
22.966221	-53.154465	54.9814383	71.51319821269692
-3.96871	-92.044787	54.90602068	71.51319821269692
4.220867	-30.64639	54.96506451	70.3458244682766
15.063512	-67.423583	54.92460019	70.90764469391664
56.532739	31.994277	55.11548594	71.45435590642103
3.241577	-108.229068	54.85479996	71.66665702711192
-48.817716	129.176191	55.14549422	71.87418232468059
46.046914	-25.709588	54.99451804	69.66001197310828
-53.757804	5.394283	54.8756141	70.69872305165876
-52.024704	31.022805	54.97892261	70.04846360254686
-40.681684	126.686102	55.02646888	69.66001197310828
42.938195	22.036066	55.06009805	71.96026372228769
-19.461929	-0.530357	54.94524549	72.29884752604941
8.505723	-85.193095	54.86634829	71.15342157914263
-54.146285	26.177234	54.88601893	70.69872305165876
6.398455	20.050255	55.04895609	71.50306268746557
59.303884	-32.517902	55.04253215	70.3458244682766
1.512023	46.278929	55.07555395	71.50306268746557
33.345414	83.890992	55.0577527	70.69872305165876
17.64726	-16.419994	55.01195876	71.50306268746557
-9.727211	43.43082	55.0293406	70.3458244682766
40.320148	-69.686876	54.95790281	71.50306268746557
14.269363	-63.508733	54.96424466	71.50306268746557
-36.983612	60.339125	55.03812451	70.90764469391664
-22.920393	124.617012	55.14336915	72.17283105704895

Per generare questi dati abbiamo usato lo stesso algoritmo che viene usato per calcolare una previsione nella pagina "Forecast" del sito web, solo che l'abbiamo applicata a 50 coordinate generate casualmente tramite la libreria random.

Come considerazione finale volevamo far notare che l'errore quadratico medio risultante dalla previsione dei dati di training e testing del database non sia elevato considerando il fatto che la grandezza del dataset è di 20,000 record e il dataset di testing è costituito dal 10% del totale. La media calcolata sugli errori quadratici medi della tabella soprastante è di 71.20044237075388, quindi una differenza in media di circa ± 8.4 decibel rispetto al rumore reale.

4.3 Trade-off QoS-Privacy

Per mostrare alcuni risultati del trade-off tra QoS e Privacy, non avendo a disposizione abbastanza utenza per la raccolta dei dati, abbiamo optato per uno script Python che ha scelto casualmente 10 misurazioni nel database con GPS Perturbation e ne ha calcolato la metrica.

I risultati sono riportati nella tabella sottostante.

Lat	Long	$a = 0.25$	$a = 0.5$	$a = 0.75$	$a = 1$
11.370211385	44.46063718	58.671195568257176	10.124806803524237	12.092770637786355	16.116655937048474
11.360379522	44.476083603	512.2811993467424	13.874672665374446	13.790922215418151	18.3655532372242
11.395963574	44.524624829	527.0086440007657	7.8609343979070285	9.734240759360544	12.948218285814058
11.385426573	44.452588283	393.13222061705056	11.355777111275906	9.673628466913861	12.887511442551814
11.346375844	44.461403781	404.90273875206026	10.471879271520827	15.25946636728124	20.317003053041653
11.36808647	44.504622435	622.1478238447197	10.927370234552917	10.970437344838404	14.60688224311787
11.30140798	44.508338024	704.5947609011064	8.96814650513457	10.267193827701853	13.663600610269139
11.335107104	44.482519148	376.0531158758308	11.883189861407105	16.05497980314416	21.40115671085888
11.328258131	44.48498615	652.3979979283808	19.17821028680412	21.565852997093295	28.748854099457727
11.365601961	44.536027205	604.8626650888237	7.928554105112357	9.628322332668535	12.829999650224714

Come possiamo notare con un α molto basso la metrica tende a salire molto perché la distanza in metri tra la posizione reale e la posizione perturbata risulta molto alta.

Man mano che l' α aumenta di valore, la metrica tenderà ad essere uguale al valore della QoS fino a quando non sarà esattamente identica.