Privacy-Aware Crowdsensing Architecture Explained

Andrea Gurioli, Giovanni Pietrucci, Mario Sessa

Progetto di Context-Aware Systems, A.A. 2020/21

andrea.gurioli2@studio.unibo.it, giovanni.pietrucci@studio.unibo.it, mario.sessa@studio.unibo.it

0000984711 - 0000985556 - 0000983529

Abstract: The documentation describes private crowdsensing on a context-aware infrastructure for sound detection. During the development, we tried to anonymize sensitive geospatial data associated with a specific user. The activity of crowd-sensing, combined with a context-aware infrastructure, generates a private system of noise surveys which analyzes them in an administration dashboard. Furthermore, data collections are used in a geo-noise prediction that enrich the map with the predicted value itself.

1. Introduction

The proposed system refers to the noise crowd-sensing inside a context-aware platform. The project's aim represents by an application development based on private data management and obfuscation to avoid tracing back to the user. The document consists of a description of how the systems work with particular attention to data management. The document structure in the first step is the design of the architecture pattern and logically specify the different components procedures. Afterwards, we talk about the implementation, which describes which one and how we used APIs. Finally, we added an evaluation chapter to define the prediction accuracy and a conclusion to list some personal considerations.

2. Design

2.1. Mobile Application

The Mobile application concerns the sensing phase of the project, which involves the context acquisition paradigm. Due to the high usage of the localization service, the focus of the application is both the quality of the audio signal and the level of privacy for a given output. As it's said before, the mobile application revolves around the sensing of the audio to obtain the level of noise expressed as:

$$Db = 10 * \log_{10}(\frac{WaveAmplitude}{Reference})$$

Our Reference value defines as 1 Watt, and the mobile device microphone sensor gives the WaveAmplitude value so that it's obtained a measure in Decibel watt, which will be sent to the Server side of the model. Part of the duties of the mobile application is to set the other main part of the project, the privacy level.
The privacy settings decide to fit the Spatial cloaking system designed in the Server phase. Due to this constraint, the manageable values are:

1. **Number of neighbours:** which are the locations to be merged in the spatial point generation
2. **Range in meters:** which is the range in meters used to identify the number of neighbours
3. **Maximum range of time:** time to live in the back-end stack for a specific location sent
4. **On/Off privacy settings:** pre-setting configuration, which makes the spatial cloaking parameters to minimum values (corresponding to the no-privacy level) or taken from the manual configuration.
5. **Automatic privacy settings:** automatic configuration in which the number of neighbours and the range will be set automatically in the back-end based on the alpha values given in input to obtain the best trade-off.
6. **alpha value:** which is the trade off level between privacy and quality of service. It can be settled by the user when the automatic privacy settings is switched to on.

The 'On/Off privacy settings' button minds for disabling the spatial cloaking algorithm; instead, the 'Automatic privacy settings' leaves to the server the computation of the privacy settings given a trade-off ratio between the privacy level and the quality of service.
The mobile application stands on the location services, giving a visual interface of the user's actual position and asking the server the mean Db for the precise location. The user's task is then to send the actual noise sensed. It can either send a single detection or activate a recording phase that periodically sends a perceived value.

2.2. Trusted Server

The trusted server is considered a secure manager that takes input data and returns the obfuscated position to send to the backend server.

2.2.1. Trusted list

The trusted list is one of the main server-side components to manage locations security. It is a typical list structure where we can put the location taken from app requests; it identifies a dynamic collection of input data for the spatial cloaking algorithm. The list uses the maximum range of time values to determine which element in the collection needs to be deleted; this control is event-oriented because the list does it only if some new location arrives at the server. Furthermore, we manage the trusted list with this procedure because the spatial cloaking pre-requisites can be satisfied only with recent locations.

2.2.2. Trusted Server procedures

The trusted server follows the RESTful architecture; it interfaces clients to a single API procedure which includes the following steps:

Data definition Trusted server procedure determines the automatic flag status. If it assumes a 'True' value, we take the alpha parameter and set k and *range* values for the spatial cloaking algorithm. At the end of the procedure, we push the final location in a trusted list.

Spatial cloaking location generation It starts with the updated list locations; using it, we can define if it is possible to aggregate some geospatial points on a filtered list (composed only on locations with no expired time to live). Afterwards, we may be able to carry out a spatial algorithm by selecting a subset of k-nearest geo-points and trying to merge them to define a spatial clock point.

Determinate noise One of the main problem from the spatial cloaking is a well accurate noise computation for the generated point. Using a simple mean between noise values of input locations can make an huge deviation. So, we solved using the Inverse Square Law:

$$r_i = \frac{1}{d_i^2} \text{ for } i \text{ in} = 1,2...k$$
$$L_i = \frac{db_i}{4\pi r_i^2}$$

This formula calculates the noise sensed propagation from a source to a point, we used it on the noise propagation between the k locations and the generated centroid (which identifies the obfuscated node). Finally, we can calculate the mean noise using the logarithmic sum of propagated noises :

$$L_\Theta = 10 * \log_{10}(\sum_{i=1}^{k} 10^{\frac{L_i}{10}})$$

where k is the number of sources which generates the spatial centroid and L_Θ is the spatial noise and L_i is the intensity from the source i.

Define QoS and Privacy After determining noise and location of the centroid of the spatial cloaked area, we added two parameters: Quality of service and Privacy metric.

1. **Quality of service** is calculated using the mean squared error:

$$QoS_\Delta = \frac{\sum_{i=0} k(L_i - L\Theta)^2}{k}$$

where QoS_Δ is the quality of service of the centroid, L_i and L_Δ are the noise sensed of the locations sources and the centroid one (previously calculated).

2. **Privacy metric** is the approximation of the 'haversine' (in Italian 'emisenoverso') formula to calculate the great-circle distance between a spatial location and its own centroid. The distance calculated is the shortest one over the earth's surface and is usually used in navigation app like Google Maps. The formula used to determine the shortest distance between two geodesic points (geodesic) approximates the geode to a sphere with the mean square radius of the Earth, so the distance calculation could have an error of 0.3%, in particular in the polar extremities, and for long distances crossing several parallels.

$$\text{Given two points } P_i(Long_i, Lat_i) \text{ and } P_i(Long_i, Lat_i)$$
$$r : R \to R : r(\lambda) = \frac{\lambda * \pi}{180}$$
$$r(Lat_i) = \frac{\pi * lat_i}{180}$$
$$\gamma = Long_i - Long_j$$
$$dist_{(i,j)} = ((\frac{(\arccos(\sin(r(Lati)) * \sin(r(Lat_j))}{\pi} + \frac{\cos(r(Lat_i)) * \cos(r(Lat_j)) * \cos(r(\gamma))}{\pi}) * 180) * R$$

where R is the mean square radius $r : R \to R$ is the function to convert in radius and Lat_i and $Long_i$ for $i = 1,2$ are the longitude and latitude of the geographical points. After the distance calculation between each location with his own centroid, we can calculate the arithmetic mean:

$$privacy_\Delta = \frac{\sum_i^k dist_{(P_i, P_\Delta)}}{k}$$

P_Δ is the centroid geographical coordinates representation and $privacy_\Delta$ is the floating value which determinate the efficiency of the spatial cloaking.

Integration After calculating every feature of the obfuscate location (which is the output centroid of the spatial cloaking algorithm), we can share the new point with his features to the back-end server using a request forwarding procedure.

2.3. Back-end Server

the back-end server is the direct manager of the database. Using it, we can:

1. **Create a location:** The trusted server passes a spatial location to the back-end server which makes a query on the database to insert the geo-spatial point with its features.
2. **Get locations:** The back-end server can get every location stored in the database and send it to the dashboard for visualization purposes.
3. **Make clusters:** It can group clusters using a k-means clustering algorithm on a location dataset taken from the database layer. There are two variants that can do cluster considering only the locations distances or even the noise sensed.
4. **Point prediction:** It is a back-end API which is able to predict a noise sensed on a point clicked on the dashboard map. We describe the prediction procedure below for more information.

2.4. Database

The storage layer is a relational database with only one table. We use it to store spatial location data. It is impossible to retrieve personal information or the past and current position because accumulated points are already obfuscated. The information inside the relational table is as follows:

1. **Point identity:** It is a simple id to identify each location in the database.
2. **Noise value:** It is the noise sensed in decibel watt that corresponds to the result of the mean of inverse square law between the location and generators (real locations in input to the spatial cloaking algorithm).
3. **Coordinates:** They are the geographical position of a location

4. **Quality of Service:** It is the mean square error on the mean noise sensed differences between the location and generators.

5. **Privacy:** It is the mean geographical distance between the location and his generators.

6. **Alpha:** It is a parameter used to calculate the trade-off between Quality of Service and Privacy. The system determines the trade-off runtime in the dashboard.

Our main goal is to obtain a privacy management system in which only the trusted server contains sensitive data and limit the security to these components while the rest of the architecture manages only obfuscated data.

2.5. Dashboard

The Dashboard is a web mapping platform in which is possible to visualize and analyze data on different views and map styles; it is possible to predict some noise sensed using a specific option, checking different clusters based on a k parameter. The map visualizes also the markers (specific obfuscated location) and his features (displayed on a popup and visible on clicking on the market itself).
Dashboard's homepage is divided in three sections:

- **Map Layer:** Use it to visualize contents on the map.
- **Map Style:** Use it to modify the map style.
- **Filters for data visualization:** Tools to choose different views options.

Since the map is the main part of the web dashboard, it is represented in full screen for a better user experience. In addition, the map supports three types of styles: classic, light and dark topographic.
The data visualization contains three macro groups:

- **Marker Visualization:** Views with only single locations available with a popup to visualize own information.
- **Cluster Visualization:** Views with clusters grouping based on the k parameter.
- **Heatmap Visualization:** Heatmap views, which the intensity is directly proportioned to the noise sensed value.

2.5.1. Marker Visualization

The main purpose of this section is to show on the map all the markers registered in the dataset. Each marker contains the following information: Noise, QoS, Privacy, Trade-off. It is possible to visualize this information by clicking on the marker.
To simplify the visualization of those markers, it is possible to colourize all of them in a red-green scale palette based on noise level (the lower is green, the medium yellow, and the highest is red).
The last option on this section is related to the prediction of a marker in the system. It is possible to activate a prediction method which will define a new point predicted on the selected location, based on the the machine learning algorithms stated below.

2.5.2. Cluster Visualization

This section makes it possible to choose two main cluster methods: spatial clustering and spatial noise clustering.

Spatial Clustering: The spatial clustering is based on the k-means clustering method. Thus it needs a k as input value given by a drop-down list. This method allows the visualization of all the sets of clusters, which is the consequence of homogeneous grouping elements in a collection of data. The second section is related to the settings; each cluster contains a centroid, vertices and a polygon; in order to obtain different analysis visualization, it is possible to modify these options. In the last section, a user can activate the colouring of all the clusters, allowing better and distinct visualization of the clusters presented in the map.

Spatial Noise Clustering: The main purpose of this method is to group the elements according to distance and homogeneous noise; thus, it needs a k as input value given by a drop-down list. To improve the visualization of the resulting clusters, a dynamic filter shows the clusters based on the noise range values on the map.

2.5.3. Heatmap Visualization

This section makes it possible to visualize the heatmap produced by the noise of the points on the map. The main part of this section is related to two range sliders which can change the radius and the amount of blur of each point of the heatmap.

2.6. Predictor

We designed the prediction phase to predict what the noise in a given point would be. It's modelled using two different machine learning techniques trained with the data queried by the Back-end Server and then shown on the dashboard. The prediction results will not be saved in the database and will not be used as other data for the prediction algorithm itself or the clustering/heating map phases.

2.7. Clustering

We designed the clustering phase to group the data queried by the Back-end Server in an unsupervised way. The idea behind the clustering technique is to group values dynamically, giving as input the number of clusters that the user wants to see.
The grouping could be done as only spatial clustering, using an input latitude and longitude, and a spatial-noise clustering, using even the dB noise in the grouping phase. The results of the clustering phase are shown on the dashboard as a composition of colors-shapes computed by the front-end.

3. Implementation

3.1. Mobile Application

The first choice for the implementation is choosing between *Android* and *iOS*; the Mobile app is designed only for Android smartphones due to developing constraints given by the iOS framework. The map used to provide visual feedback to the user is granted by the *GoogleMapsAPI*. The application's core can be seen as a client-server service where the application works as a client. The connection between the application and the server is made by HTTP protocol, using the Volley API. There are two different requests given to the server:

- *getMeanDB*: Given the current location, returns a *JSON* with the mean noise sensed using noise surveys around 3 kilometres.
- *createLocation*: Send the actual location and noise sensed to the trusted server.

Both are POST requests; the *getMeanDB* is a request sent directly to the back-end server where it provides a *GeoJSON* data in the body request. It uses a *GeoJSON* format to manipulate the spatial data homogeneously in the system. So that, for the *getMeanDB*, the location of the application is manipulated as a *GeoJSON* feature and sent to the server to receive and visualize the actual mean noise in a range of 3000 meters. The location sent is *truncated* in order to obtain a higher privacy level due to the fact that the backend is not a trusted server. The *createLocation* request grasps the core of the project sending the location and noise sensed at a given time to the trusted server. We sents data using the GeoJSON format in the post's body. Along with the meaningful data for the back-end server, it's also given a set of properties for the trusted server to set up the spatial clocking within the preferences of the user; the settings provided are:

- *Neighbour* : Integer between 1 and 3 which define the k parameter for the spatial cloaking algorithm
- *Range* : Integer which describes the max radius where catch location to aggregate and form a spatial centroid
- *MaximumTime* : Value for the spatial cloaking
- *Timestamp* : Value for the time evaluation
- *UserID* : UUID associated with a session device and categorized as a string.
- *PrivacyOnOff*: Boolean value establishing the flag value in order to define if the trusted server has to start the privacy obfuscation (spatial cloaking) or not on the point received
- *Auto* : Boolean flag establishing if it's sent a location with automatic configuration or not.
- *Alpha* : Value considered only if the location sent is automatic, ignored in other cases.

3.2. Trusted server

The trusted server is implemented using the *express.js* framework which includes a suite of libraries to build a RESTful server in Javascript; the filename for this component is *serverTrusted.js* located in the *server* folder; we can run it from terminal using *npmruntrust* command and set it in listening on port 3000. Servers APIs passes parameters to one specific procedure located in *routeTrusted.js*. APIs in the *serverTrusted.js* consist only in the */createLocation* : Which invokes the *createLocation* function in *routeTrusted.js*. It is the main function of the whole project. It takes the data json from the body of the *request*, checking if it needs an automatic configuration or not using the *automatic* function (if it is, this procedure assign *k* and *range* features based on the *alpha* value. Afterwards, we invoke the *makeSpatialPoint*, which defines if it could make a spatial location or return the current geoJSON depending on the *privacyOnOff* value given in the app front-end. Finally, we forward the point made by *makeSpatialPoint* to the back-end using the *forwardBackend* function, which makes a request using *makeRequest* procedure and send it to the back-end function (which listen on port 3000).

3.2.1. Spatial Cloaking

We implemented the spatial cloaking algorithm in *spatialCloaking.js* a javascript file located in the static folder of the *server*. The main function is *makePoint*, which filters the *stack* of location, checking if their time-to-live is expired or not, using the function *time*. It takes a subset of stack locations where it is possible to aggregate them in order to make a spatial point using a *filter* procedure. Finally, the collection of locations given by *filter* function will go in input to the *spatialCloaking* procedure where we make a geoJSON location which has the following features:

1. **db:** It is the noise assigned to the centroid; we calculate it using the *defineDbMean* function, which use the *inverseSquareLaw* values on each geoJSON generator to make a propagated noise value. The collection of the propagated noises values will go in arithmetic mean assigned to *db* feature.
2. **QoS:** It is the Quality of Service metric in floating format calculated using the *makeQoS* function as well as we described in the design section.
3. **Privacy:** It is a floating value that defines the mean geographical distance between the centroid and each generator (within the *geographicalDistance* function). We give it as a return value from the *makePrivacy* function.
4. **Alpha:** It is a floating value used to calculate the runtime trade-off, and we give it by the mean of *alpha* values of location generators taken by the return value of the *makeAlpha* function.

We remember that we call the spatial cloaking procedure only if the *privacyOnOff* is true. In other cases, we call the *makeDirectPoint*, which returns the API's effective location given in input.

3.3. Back-end server

The back-end server focuses on the same implementation as the trusted server. We implemented it in *express.js* and used the *routeBackend.js* script to invoke business functions for APIs. The Back-end filename is *serverBackend.js*, which is located in the *Server* folder, while the route procedures filename is *routeBackend.js* situated in the *static* folder of the *Server*. If we want to run this server, we need to use *npmrunbackend* on the command line. Invocable APIs for the back-end server are the following:

1. **/getLocation**: It calls the *getLocations* procedure on *routeBackend.js*. It includes a routine to make a *GET* query for the whole locations of the database, transform *row* data in a *GeoJSON* format and gives in *response* this collection.
2. **/getMeanDb**: It calls the *getMeanDb* function on *routeBackend.js*. It makes a *POSTGIS* query for the locations near a defined point settled by the mobile application, then it gives in *response* a *JSON* element with the mean db near the point itself.
3. **/showClusters**: It calls the *showClusters* procedure on *routeBackend.js*. It includes a routine to make a *GET* query for the whole locations of the database, transforming *row* data in a *GeoJSON* format and giving in *response* a *JSON* collection which includes:
   (a) **Locations**: It's a list of *GeoJSON* locations from the databases
   (b) **Clusters:** It's a *GeoJSON* collections of *Polygons*
   (c) **Centroids:** It's the *GeoJSON* centroids list of *Polygons*.

Clusters are made using the $k-means$ of the *sklearn* library. The *GeoJSON* formatting is done using the *convertClusters* function defined in $utility.js$.

4. **/showClustersOnDb**: It use a query to takes every location from the database, convert them in *GeoJSON* format within the *convertLocations* procedure and, using the *bridgingClustering* calls in a child process a *Python* k-means algorithm to define a classification of locations based on coordinates and *noise* weights.

5. **/createBackendLocation**: It'll call a *createBackendLocation* function on an *INSERT* query to add a new location in the database corresponding to the obfuscated (or the real in case of the no-privacy flag) location.

6. **/prd**: It is the procedure which calls the *prdCall* function of the $routeBackend.js$ script. *prdCall* invoke the *bridgingPredictor* function, which makes a child process to execute a *Python* prediction script described in the prediction phase's paragraph.

For query management, we used the *pg.Pool* object, which is a specific and easy way to invoke queries on a *Postgres* database. This library works on an object instance that we need to specify: *user* identification, *host* name, *database* name, *password* and *port*; it works locally and, in case of deployment, we must change the previous list of data with the remote one.

3.4. Database

The database includes only one relational table called loc_ref_points, where we can find:

1. **Id**: It is an *integer* with $auto-increment$ option that represents the first field of the primary key of the table.

2. **db**: It is a *doubleprecision* value *notnull* where put the location noise value.

3. **coordinates:** It represents a *geometry* attributes available in *PostGIS* extension and mantains the geographical coordinates of the location's $STPoint$.

4. **QoS:** It is a *doubleprecision* value *null* where put the value of the Quality of Service metrics is calculated in the Trusted Server. Value is 0 for no-privacy locations and $n! = 0$ otherwise.

5. **Privacy:** It is a $double-precision$ value calculated in the Trusted server and put in the database on creation. Value is 0 for no-privacy locations and $n! = 0$ otherwise.

6. **Alpha:** It is a $double-precision$ value calculated in the Trusted server and used in order to computate the actual tradeoff between the privacy and the QoS.

Only the *routeBackend* can use *pg.Pool* queries to interface with the database.

3.5. Dashboard

The front-end web application is implemented in *JavaScript* and *HTML*; data manipulation focuses on the application of *AJAX* callbacks (particular callbacks procedures implemented in *JQuery* framework). Using *AJAX*, we can call $express.js$ APIs and give a data collection in *GeoJSON* format in response. We also implemented the web application with Leaflet, a JavaScript library for interactive maps.
Since the system is focus on the graphic visualization of the data, there are presented five types of methods based on ajax calls, all of them in POST request:

- **showMarkers:** It shows locations as marker and link each of them to a popup for information showing.
- **showPredictedMarker:** It shows a predicted location by clicking inside the map views.
- **showHeatmap:** It shows points as heatmap with different intensity based on the noise level.
- **showClusters:** It shows polygons in output to the k-means clustering algorithm basing on only geo-spatial features.
- **showSpatialNoiseCluster:** It shows a different version of clustering basing on geo-spatial features combining with the noise level.

3.6. ShowMarker

In this section, the system receives via *URL* "/*getLocations*" within callback management in case of success or failure response type. Since features are the GeoJson result of the callback, we divided it in two main parts:

- **Geometry**: It represents the coordinates (latitude and longitude) used for the creation of a marker.
- **Properties**: It represents all the properties of a point which include the following data types: $Alpha$, *Db*, *Privacy*, *Privacy* and *QoS* (these data have the same format described above).

There is another data that has been inserted inside each marker, that is the trade off whose calculation is as follows:

$$TradeOff_{\Delta} = (privacy_{\Delta} * alpha_{\Delta} + (QoS_{\Delta} * (1 - alpha_{\Delta})$$

The marker has a circular shape to obtain a clear visualization. It also has colour options: one related to the default colour and the other related to a colour based on a green-red scale palette.

3.7. ShowPredictedMarker

The main core of this section is given by a POST request "/*prd*" with additional data sent to the server with the request; this data contains the coordinates of a point by clicking on the map. The result of the request will be a predicted point with coordinates and properties related to the noise. The point has a circular shape to obtain a clear visualization. It also has colour options: one related to the default colour and the other related to a colour based on a green-red scale palette.

3.8. ShowHeatmap

This part follows the same procedure of ShowMarker since the system needs the geographical information and the noise registered on each point. We implemented this section with $Leaflet.heat$; a plugin given by Leaflet plugins.

3.9. showCluster

We base this section on a POST request given via URL "/*showCluster*" with an additional k sent to the server with the request; this data contains the number of clusters needed for k-means clustering. As a result, we obtain a $FeaturesCollection$ structured as described above on the back-end server section.

3.10. showSpatialNoiseCluster

This section is based on a POST request is given via URL "/*showClustersOnDb*" with an additional k sent to the server with the request; this data contains the number of clusters needed for k-means clustering. As a result, we obtain the same data of $FeaturesCollection$ from the "*showClusters*" call given by a different clustering algorithm that considers noise features.

3.11. Prediction phase

The predictor mechanism is stated to predict the noise for a given location using only latitude and longitude data. The prediction mechanism is given by a python script that runs every time a client calls it in the dashboard platform. The model is settled in two different parts:

- **SQL query:** which takes training dataset from the postgres database
- **Machine learning predictors:** which applies the main algorithm to predict target information

At the beginning of the model, the script's data is queried, creating a connection between the script itself and the *PostgreSQL* database. It is used as a spatial query in which, given a point it is sought all the point in the distance of a given range in meters. All the points are queried by using the 4326 geographical projection. The queried points are then used to train a machine learning model; the algorithm used is selected in behalf of the number of points queried, if there are less than three points in the dataframe, it is instantiated a neural network model (trained with scaled data whithin domain of [0,1]), after an implementation of another query, but this time without range restrictions; if instead there are more than three points, it is used a knn model with 3 neighbour as an algorithm parameter. The sklearn APIs give all the machine learning's algorithms. The value predicted by the model in use is then printed out as a stream to the server, and it's received as a string with a JSON format including only the property 'db'.

3.12. Clustering phase

A clustering algorithm for spatial clustering and a spacial/noise clustering has been used to obtain a natural and representative split of the data in the dashboard. The algorithm used is the Kmeans, in which, as hyperparameters, it's been dynamically settled the number of centroids, calibrating it with a dashboard input. The metric of the distance between the points is the classical euclidean distance between two points.
For the spatial clustering, the result is given using only each sample's latitude and longitude, placing them respectively in the x-axis and y-axis. Instead, the third 'db' field is settled in the z-axis generating a three-dimensional space for the clustering based on spatial and noise data. Given the three-dimensional representation of the space, an issue of magnitude arises; The noise value impacts the algorithm tenfold higher in respect to the latitude and longitude. The noise's value has been normalized in a range within the maximum difference between the latitude and longitude to avoid this problem. The algorithm, given by the sklearn APIs, is, like the predictor phase, settled as a script and ran by the server back-end; and it returns a JSON element as an output for the dashboard.

4. Evaluation

4.1. Trade-off evaluation

Core of the project is the trade-off evaluation between the level of privacy which is given by the distance between the real points and the centroid between them, and the Quality of service, which is given by the mean squared error between the actual noise values and the resulting one. The trade-off is managed by an $alpha$ value which can be settled by the user. In this evaluation phase we have the results obtained with different $alpha$ values:

$alpha$	Privacy	QoS	Trade-Off
0	0	0	0
0.25	4.48	25.14	19.98
0.5	639.28	9.45	324.37
0.75	693.75	64.32	536.39
1	1280.97	123.35	1280.97

As it can be seen, different values of $alpha$ lead to different Privacy and QoS values, the optimal value is merely subjective in regards of the user. The surveys results are biased by the distance used. Given that, the table can be seen as an indicative trade-off evaluation, but the values for further evaluations could be extremely different from the ones obtained.

4.2. Prediction

For the prediction algorithm, apart from the KNN regressor which has been used as a natural implementation for a noise prediction and works with fewer values than the other ones; when the number of close points to the actual 'to be predicted' point are less than the hyper parameter selected, the evaluation metric used is the MSE. It has been evaluated two different type of regression models.

- Linear Regression (LR)
- Neural Network Regressor (NNR)

Regressor	LR	NN
MSE	298.7($\pm$0.1)	287.2($\pm$0.1)

As it can be seen, the linear regressor tends to perform sensibly better with the MSE evaluation metrics, otherwise, this type of regression is stated as an high distance regression from the actual values; given that, the final regression's model chosen is the NN one because of the performance on isolated locations. Indeed, in a testing phase, the linear regression tend to predict values as outliers on a noise distribution and becomes unfeasible.

5. Conclusions

The Proposed architecture can be used for other context-aware projects within the security management, which is delivered to a trusted server; meanwhile the back-end server works as a database manager. Use of *GeoJSON* and *POSTGis* can easily manipulate data for their visualization in front-end or for data extraction in back-end size. The privacy techniques used for Geo-spatial data like *SpatialCloaking* can obfuscate easily personal locations and maintain a good accuracy on features perturbations. Finally, the prediction of noise values using the geographical properties in a spatial environment has an enormous set of use cases for academic or professional purposes.

References

1. *Aaron Larson, Basics of Sound and Noise Propagation - Powermag.com, May 1, 2018*
2. *K. Saputra, N. Nazaruddin, D. H. Yunardi and R. Andriyani, "Implementation of Haversine Formula on Location Based Mobile Application in Syiah Kuala University," 2019 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*
3. *Chow, CY., Mokbel, M.F. & Liu, X. Spatial cloaking for anonymous location-based services in mobile peer-to-peer environments. Geoinformatica 15, 351–380 (2011).*
4. *Celik, Cahit & Guremen, Lale. (2008). Analytical noise contour contruction using inverse square law of sound propagation. Environmental Engineering and Management Journal.*
5. *The Basics: KNN for classification and regression, Max Miller, 2019*