DP4coRUna Android App Implementation Progress Report and Current Documentation

### **Current Design in Mind**

As of now we have a design considered for the features discussed in the CollabLoc paper. We have components setup for each of the three components: Local Learning, network, and machine learning. However, these components still need more refining and more importantly need to be connected together in a more cohesive manner.

Our current design uses services to carry out the three broad tasks mentioned above. Services refers to the Service class found in Android libraries. Services can run in the background which make them a good candidate for carrying out the tasks for local learning and detecting when it is appropriate to begin local learning.

The design would have a motion detecting running service in the background and several other bound services that would be called by the motion detector to carry out local learning logic. The running service would be triggered when the application is started and would continue even after the app is closed for local learning purposes. To keep this service from draining too much power, this service would only be used to detect transition from motion to non-motion. Once the transition is detected a service to calculate the wifi rrsi values and then a local features learning service would be called, to check if the device is in the same room. This is based on the procedure outlined in CollabLoc.

The services to calculate wifi rssi values and obtain local features are bound services which will only be called when necessary. Once these services run and carry out their tasks the network services (also bound services) will be called.

The network services will be called when necessary, based on standards defined by CoolabLoc Paper. The network services implemented in the Android App are documented in another document, please refer to it for more information. The network service will ultimately be responsible for transferring data from one device to another. The device that receives the information will run the Machine Learning algorithm set up and return the output data.

The Machine Learning section will use the DeepLearning4J library to carry out softmax (/multinomial logistic) regression. It will be pre-trained in the device with the data available in the device database. Once trained the model will be saved into the device. Upon receiving the transmission through the network the Machine Learning service will load the model in device and run it with the given input. The model is saved into the device so it does not have to be trained every time it is required and only when new data is inputted into the device.

As of now, the tools for these tasks exist but the services to connect them do not. We only recently discovered the idea to implement these tasks through services. The key task is to correctly connect each of the components.

### **Components That Currently Exist:**

The following description will be organized based on the package structure and the tasks each component is part of.

The tasks are Local Learning, Networking, and Machine Learning.

The directory structure is as follows:

| MainActivity.java

| reportPositiveTestActivity.java

| TempResultsActivity.java

|

+---dataManagement

| | AppDatabase.java

| | DataBaseTestActivity.java

| |

| \---acess

| QueryLocation.java

|

+---localLearning

| | SubmitLocationLabel.java

| |

| +---learningService

| | LearningHandler.java

| | LocalLearningService.java

| |

| +---location

| | | LocationObject.java

| | |

| | +---dataHolders

| | | CellData.java

| | | LocationObjectData.java

| | | WiFiAccessPoint.java

| | |

| | \---learner

| | LocationGrabber.java

| | SensorReader.java

| |

| \---movementTracker

| AccelerationSensor.java

| MovementSensor.java

| TrackMovement.java

|

+---mapmanagement

| enterDestinationActivity.java

| MapsActivity.java

| showRoutesActivity.java

|

+---ml

| MLData.java

| MLModel.java

|

\---network

AES.java

Gateway.java

NetworkConnector.java

NetworkReceiveActivity.java

NetworkRelayActivity.java

NetworkTransmitActivity.java

NetworkUpdater.java

RelayConnection.java

RelayServer.java

RelayService.java

RSA.java

Transmitter.java

TransmitterService.java

WelcomeActivity.java

#### **The main components are as follows:**

**Within the Main Directory the following classes exist:**

MainActivity - The opening screen, it contains buttons for user input based activities and functions. Right now it contains buttons to activities used for testing purposes, eventually most of them will/should be replaced by services.

TempResultsActivity - A screen that is used to test other classes and functions, contains a few buttons and a text view. Buttons trigger the methods while text view shows output from methods. Right now there is random code in the class meant for testing.

reportPositiveTestActivity - Page where user can input their status, currently is not connected to rest of the code or design.

**localLearning.location folder:**

\*\*-- *Notes regarding location features learning:*

* Read google location and fusedlocationclient documents for info on api used.
* New google maps api needs to be obtained upon resumption of project.

--\*\*

localLearning directory contains most of the sub-folders and classes that deal with the Local Learning task. This directory contains services, function classes, and objects that hold location feature data and or obtain said data.

**SubmitLocationLabel** - An activity that takes in user inputted label for a location. See CollabLoc paper for more information on this.

**LocationObject, LocationGrabber, SensorReader:**

The primary class of the folder is LocationObject, which contains all data about a particular location (room/building). This includes data found in Google location api and sensor data.

**LocationObject** - extends the classes LocationGrabber and SensorReader, which are contained within the *learner* folder in the *location* folder. These two classes obtain the location data and make them available to the LocationObject instance. Since LocationObject extends these two classes it has access to their fields.

**LocationGrabber** - class calls google location APIs to obtain location information for the current location. The information obtained here include the building address as well as the building latitude, longitude, and altitude.

**SensorReader** - class initiates connections to device sensors to obtain several different types of sensor data. Currently the types of data obtained are light, sound, geo-magnetic field strength, cell tower information, and wifi access point data. Once the LocationObject class’s updateLocationData() method is called the sensor’s sens() method is called, due to extension. When sense() is called it calls all of the methods that start sensors and wait for their data.

It should be noted that the sound value requires further processing. As it is now it is not reliable due to the sound sampling method returning different data for different devices, details on this can be found in Android MediaRecorder library documentation.

Also, currently both a euclidean magnitude and a vector for geo-magnetic field strength is stored within SensorReader. Geo-magnetic field strength is reported along the x,y,z axes by Android. The values that are required should be used.

Furthermore, there are other sensors that can be implemented based on their availability in the device. These include temperature and humidity sensors as described in the CollabLoc paper.

*Notes on LocationObject, LocationGrabber, and SensorReader*

Both LocationGrabber and SensorReader have asynchronous tasks within them, since google location api is asynchronous and sensors report data in asynchronous manner. Since carrying out the logic, as per our current designs, requires the data be available before other methods can run, we utilize threads and thread synchronization. Primarily we use Threads, HandlerThreads, and Loopers to run sensor and location api functions in other threads. Then join() statements and BlockingQueue are used to wait for data to become available. These methods block the main thread while waiting for the other threads to finish. This causes a hang in the main application UI that needs to be addressed in the future.

**LocationObjectData** is an intermediary object primarily used to convert a LocationObject to JSON string format and then turn it back. A JSON string is required to transfer data across the network. This class is not explicitly used in any other part of the app aside from within LocationObject.

The classes in **localLearning.location.dataHolders** contain intermediary data. They are used for organization purposes and not explicitly used in any other functions or classes, all except for **WiFiAccessPoint** which is used by wifi scanning services.

**localLearnig.learningService and localLearning.movementTracker** :

These two folders currently contain experimental code, since the services functionalities have not been implemented yet.

The movementTracker folder contains the TrackMovement class which is a service that calls upon either the MovementSensor or AccelerationSensor classes to detect transition between motion and non-motion. The TrackMovement service will be a running service which will run continuously even if the app is closed. This service will call the other services to obtain wifi data and then send network transmissions if necessary.

Currently, the TrackMovement class calls the MovementSensor, which uses Android’s built in Significant Motion Sensor. This method is not recommended for the future, since (at least in the device that code was tested on) the motion detection requires more than a room level transition. The amount of movement required to trigger the sensor is greater than what we would want. The best course of action would be to devise an algorithm with raw accelerometer data, this is the purpose of the AccelerationSensor class (which is currently nearly empty).

**dataManagement:**

Contains the code for creating and updating database code. The **AppDatabase** class contains all of the code used to interact with the in device SQLite database. This class extends the SQLiteOpenHelper class that is used to construct and access the database.

Most methods in this class either enter data into the database or retrieve data from the database. Since this is a SQLite database implementation a combination of Java and SQL is used for query/insert purposes. This is done through the Android database libraries that are available. Android’s database library contains methods that can execute string SQL statements. We write the statements as string values and pass them onto the appropriate command. We use “.execSQL()” to create table, “.insert()” to add data, and “.rawquery()” to retrieve data.

The AppDatabase class contains methods that access data from the database for the purpose of training the machine learning model. The getMLDataFromDatabase() method and all of the sub-methods it calls retrieve and format the data necessary for DeepLearning4J machine learning model. While doing so, an intermediary data class MLData is used to hold all of the data necessary for the machine learning model. These methods are already set and have been tested to produce the proper formatted data necessary for the model. The actual data itself is not changed here, this should be done in the machine learning model code if necessary.

**QueryLocation** - class within **dataManagement.access**. This class is an abstraction for retrieving data from databases and or running the machine learning model. It is used by a receiving device to check its own database for locations features of a provided location label or the possible location labels for a set of provided location features. This class calls methods found in AppDatabase and MLModel to carry out the logic for receiver devices in the network, this logic is explained in the CollabLoc Paper. We have not fully implemented the logic explained in the paper but this is the infrastructure upon which that logic will be built.

**ml folder:**

**MLModel :**

Contains the Machine Learning model and logic to train/run it. Currently our machine learning implementation is done through DeepLearning4J (DL4J), an open source machine learning library for Java that is usable in Android devices. This library provides both data structures and trainable machine learning models. We have used DL4J to implement a multinomial logistic regression (softmax) algorithm.

The actual machine learning model has been set up in the createMlModel() method in MLModel class. It was simply an implementation of the DL4J multi-layer network class. Since we only require a softmax algorithm, for now, only one layer was used. The multi-layer network also contains seed, optimizer, updater, and loss function parameters. These parameters are currently set based on online tutorials found, but should be checked more thoroughly and updated if necessary in the future. The createMlModel() method also contains a for loop which calls the “fit()” method on the model to train it. The number of times it should be trained, or epochs, is set to 3 for now. In the future this value should be tested and changed if necessary. The createMlModel() method is called within the class and set up when an MLModel instance is created, based on the request type specified.

The main entry points for this class’s functionalities are the trainAndSaveModel() and the loadModel() methods. These methods abstract away the rest of the machine learning functionalities. They are called by the constructor when a MLModel is initialized in another class. Which method is called is based on the request parameter of the constructor. The two requests, as defined in the class, are “LOAD\_MODEL\_FROM\_DEVICE” and “TRAIN\_MODEL\_AND\_SAVE\_IN\_DEVICE”. These are static variables in the class and should be called when using the constructor for the class, like this: new MLModel(context, MLModel.TRAIN\_MODEL\_AND\_SAVE\_IN\_DEVICE).

The trainAndSaveModel() method calls the createMlModel() method and trains the machine learning algorithm. Before doing so, it calls the getDataAndCompileDataset() method. This method uses the AppDatabase class to obtain all available location data in the device. Then it creates a Dataset object, found in DP4J library, from the data and sets it to the global Dataset declaration. Then createMlModel() generates the machine learning algorithm using the global Dataset variable. Once the algorithm is created, trainAndSaveModel() saves the model to the device using the DP4J library functions. The dataset created for the model training is also saved so its data can be referenced later. The model is saved so it does not have to be retrained every time it is required, only when new data is added to the device database.

When the machine learning model is required it is loaded from the device through the loadModel() method. This method loads a pre-trained model and dataset saved through the trainAndSaveModel into the app so the algorithm can be run.

After a model is loaded, it can be run using the getOutput() method provided by the MLModel class. The getOutput() method takes a LocationObject as its parameter and formats the features within it into a feature vector that can be inserted into the machine learning model to obtain results. The results are returned as a string. In the softmax implementation the results are an array of probabilities adding to 1. The number of elements in the array is equal to the number of locations available in the device and each element corresponds to each of those locations. The value of the element is the probability of that element being the location that the features belong to, see the CollabLoc paper for more details of this idea. The getOutput() method is an instance method, so a MLModel instance has to be created (with the load request) and then used to call getOutput().

**MLData** :

This class is an intermediary data holder used to transfer formatted data from the database to the machine learning class. It contains features, labels, and encoded labels that will be used to construct the Dataset object. Instance of this class are created by the AppDatabase class and returned to be used by the MLModel class.

*Notes Regarding DP4J* :

* DP4J is a massive library that takes up a lot of space in computers using it and apps implementing it. It also may increase the gradle build time for Android. Currently we are using version 1.0.0-beta6 which is not the most recent but fits our size constraints (using default settings). There is a beta7 version but its size is too large for default settings android studios, due to an issue with multi-dexing. As of now, beta6 contains all of the functionalities we need for our app, but if upgrade is required then the multi-dexing issue has to be investigated and addressed.
* DP4J provides several data structures along with machine learning models. One example is the INDArray interface and its implementations which behave like numpy arrays from python. They can be used for non-DP4J models and machine learning algorithms in a way a numpy array might.
* DP4J was picked over TensorFlowLite because it allows custom models to be trained in the device.

**mapManagement**:

This folder contains experimental stuff that hasn’t been fully investigated yet. As of now we do not have any major component implementations here.

**network** :

Discussed in the network doc.