

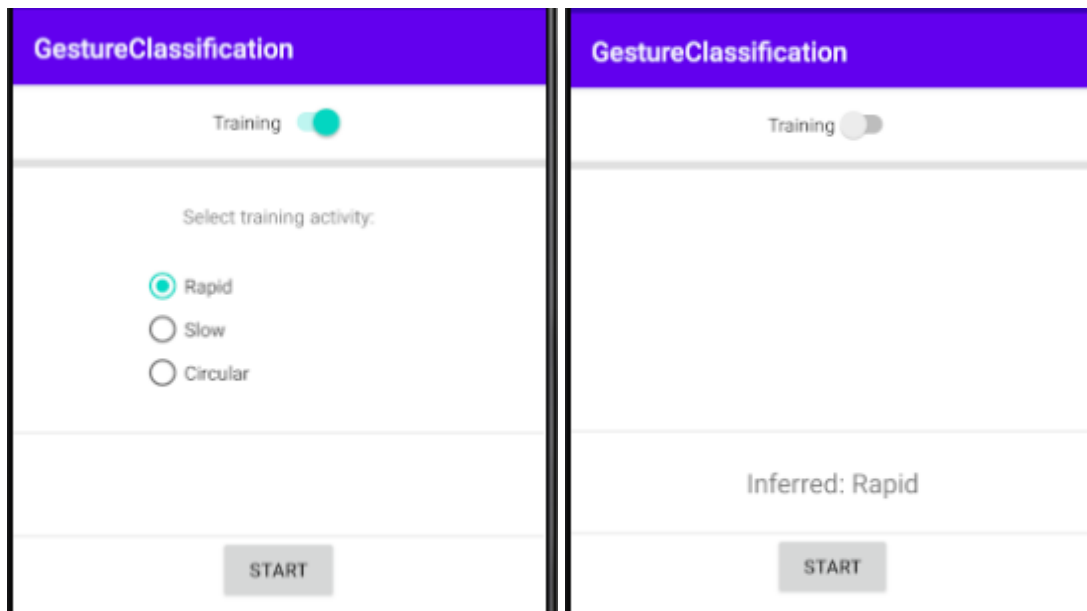
Lab 7 - Machine Learning in Android

Mobile phone sensors provide raw data, e.g. GPS coordinates, and machine learning is used to link the sensed data with higher level concepts, e.g. whether a person is at home or work. In this lab we will use periodic accelerometer sensing and recognise certain phone movements (gestures) using a Naive Bayesian classifier.

Task overview

To shortcut the development process, you can download code from <https://bitbucket.org/pbdfrita/pbd2023-lab-7>

We are going to sample the accelerometer every five seconds. These samples will contain X, Y, and Z axis measurements of the acceleration. We are then going to calculate representative features - the mean, variance, and mean crossing rate (MCR) - of the acceleration. We are going to train a Naive Bayesian classifier to learn how these features change depending on a gesture that a user performs with the phone. Finally, we will use our trained classifier to recognise the gestures in real time.



The user interface consists of a Switch that tells whether the application should be in the training mode (picture on the left) and use the accelerometer samples and the selected training activity to teach the machine learning model what the selected training activity “looks” like, or whether it should be in the testing mode (picture on the right) and use the accelerometer samples to infer the gesture that a user is performing. The button on the bottom instructs the application to start (or stop) the accelerometer sensing.

The core classes of this project are going to be `AccSenseService`, a `JobIntentService` for sensing the accelerometer, and `TrainClassifierService`, a `JobIntentService` for training the classifier. The code for controlling the training and inference will live in `MainActivity`. A `BroadcastReceiver` called `AccBroadcastReceiver` will be used to get the accelerometer sensing results from `AccSenseService` to `MainActivity`.

JobIntentService provides the same functionality as IntentService but uses jobs instead of services when running on Android 8.0 or higher¹. JobIntentService provides two callbacks (OnHandleWork() and OnStopCurrentWork()) and one method (EnqueueWork). We will be using the method and the first callback:

EnqueueWork (Context context, Class cls, int jobId, Intent work) - adds the work to be executed into the queue.

OnHandleWork (Intent intent) - A method that receives dequeued work from the queue. Intents loaded by enqueueWork() are passed to this method when the service is executed. Like IntentService, loaded intents are delivered sequentially.

Periodic accelerometer sensing

To periodically sense the accelerometer in MainActivity we will use a Handler that will every five seconds post a task - to run AccSenseService which will collect accelerometer sensing data.

Find startSensing() function in MainActivity and instruct it to call AccSenseService every five seconds:

```
handler?.postDelayed(object : Runnable {  
  
    override fun run() {  
  
        //call function  
  
        val intent = Intent(applicationContext,  
  
            AccSenseService::class.java)  
  
        AccSenseService.enqueueWork(applicationContext, intent)  
  
        handler?.postDelayed(this, 5000)  
  
    }  
  
}, 5000)
```

Open AccSenseService to understand how the sensing happens. Every time we start this service it registers itself as a listener for accelerometer data. When the data is received (in onSensorChanged) it checks whether it accumulated 50 samples² and if so, it calculates the mean, variance and the mean crossing rate (MCR) of the accelerometer data. These data should be sent via an Intent that is broadcast locally. The data will then be received in MainActivity, via AccBroadcastReceiver. First, implement the sending part in AccSenseService:

```
val localIntent = Intent(MainActivity.ACTION_SENSING_RESULT)  
  
localIntent.putExtra("mean", mean)  
  
localIntent.putExtra("variance", variance)  
  
localIntent.putExtra("MCR", mcr)
```

¹ <https://developer.android.com/about/versions/oreo/background.html>

² It needs roughly ten seconds to get that many samples, because it is subscribed using SENSOR_DELAY_NORMAL.

```
LocalBroadcastManager.getInstance(this).sendBroadcast(localIntent)
```

Next, the data should be received in `MainActivity`. In `onStart` you should register a local broadcast receiver:

```
mBcastRecv?.let { LocalBroadcastManager.getInstance(this).registerReceiver(it,
IntentFilter(ACTION_SENSING_RESULT)) }
```

And then unregister it in `onStop`, the code is trivially similar to the above.

Recognising different movement patterns

Different phone movements, for example, a rapid motion, a slow motion, or a circular motion, result in different accelerometer values' recordings. Our hypothesis is that different movements will result in different values of accelerometer mean, variance and MCR. We will use a third-party machine learning library to manage the movement pattern classifier.

Add the Android machine learning library to your project, by putting the following line in the app's gradle file:

```
implementation 'si.uni_lj.fri.lrss.machinelearningtoolkit:mltoolkit:1.2'
```

The machine learning library lets you create classifiers, train them with labeled data, and then query them to infer high level information from the sensed raw data. The complete code for the library can be found here: <https://github.com/vpejovic/MachineLearningToolkit/>

There are a couple of more things you should take care of to make sure the library can be found and properly used in your project. First, go to `settings.gradle` and add `jcenter()` to the `dependencyResolutionManagement { repositories {} } block`. Second, add `tools:replace="android:theme"` to the `<application>` element in `AndroidManifest.xml`.

Instantiating a classifier

A classifier is instantiated through the `MachineLearningManager`. The manager requires a context:

```
mManager = MachineLearningManager.getMLManager(applicationContext)
```

A classifier is defined by its `Signature`, i.e. the features it connects, and the indicator of the class feature (the one we are trying to predict). Features can be numeric (i.e. expressed through real numbers, can be ordered), or nominal (i.e. representing different categories, cannot be ordered). An example of a numeric feature is the accelerometer mean value. An example of a nominal feature is our class, a particular movement (e.g. "rapid", "slow", "circular", etc.).

The machine learning library supports a few different classifier types, and exposes certain configuration options. We will keep it simple and instantiate a default Naive Bayesian classifier. The following code instantiates a Naive Bayesian with one numeric feature and one nominal class feature with two values.

Using the code below as template, instantiate your own classifier that takes the three numeric accelerometer features (mean, variance, and MCR), and predicts one nominal feature with the movements you want to predict (note: these should be defined in `R.string.txt_gesture_*` variables):

```
val accMean: Feature = FeatureNumeric("accMean")

// TODO: two more features should be defined here
```

```

val classValues = ArrayList<String>()

classValues.add(getString(R.string.txt_gesture_1))

classValues.add(getString(R.string.txt_gesture_2))

// TODO: one more class value should be added here

val movement: Feature = FeatureNominal("movement", classValues)

val features = ArrayList<Feature>()

features.add(accMean)

// TODO: two other features should go here

features.add(movement)

val signature = Signature(features, features.size - 1)

// TODO: a try-catch block is needed here

mManager?.addClassifier(Constants.TYPE_NAIVE_BAYES, signature,

    ClassifierConfig(), "movementClassifier")

```

Where to instantiate your classifier? You can do that in the `MainActivity` directly (we already created the `initClassifier` function as a placeholder). The instantiation itself is not processing-heavy. On the other hand, classifier training and querying might be, and should be done on a separate thread.

Training a classifier

Now, your classifier doesn't know much, so let's train it. First, we need to provide a label (gesture name) for the sensed movement. When a user sets the Switch into the training position, we need to get the label next to the selected radio button. This label, together with the current accelerometer feature values will be used for training the classifier. Note, that we have defined the possible gestures as "rapid", "slow", and "circular" in `R.string.txt_gesture_*` variables. However, you can easily change this to recognise e.g. "forehand" and "backhand" movements for a virtual tennis game, or "Yes" and "No" gestures for a distributed voting app, or any other gesture you fancy. Anyway, you can use:

```

val selectedId = binding.radioGroup.checkedRadioButtonId

val rb = binding.root.findViewById(selectedId) as RadioButton

val label = rb.text

```

To get the label of the selected `RadioButton` (see `recordAccData` function).

Training a classifier can be processing heavy, so it should be performed in a separate thread. That's why we have created the `TrainClassifierService` that extends `JobIntentService`. In its `onHandleWork` we will train the classifier. Here is an example of training a classifier with a class value label (`String`), and a mean accelerometer value `mean` (`double`):

```

val instanceValues: ArrayList<Value> = ArrayList<Value>()

val meanValue = Value(mean.toDouble(), Value.NUMERIC_VALUE)

```

```

instanceValues.add(meanValue)

// TODO: two other features should go here

val classValue = Value(label, Value.NOMINAL_VALUE)
instanceValues.add(classValue)
val instance = Instance(instanceValues)

val instances: ArrayList<Instance> = ArrayList<Instance>()
instances.add(instance)

// TODO: a try-catch block is needed here
val mManager = MachineLearningManager.getMLManager(applicationContext)
val c = mManager.getClassifier("movementClassifier")
c.train(instances)

```

In your implementation, besides mean, add variance and MCR. To train a classifier in a separate service we need to pass the information on the mean, variance, MCR and the label to the service in an `Intent` from the `MainService`. This is not much different from the way we call `AccSenseService` when you do periodic sensing. See `recordAccData` in `MainActivity` and find a suitable place to put:

```

val mIntent = Intent(this@MainActivity, TrainClassifierService::class.java)

mIntent.putExtra("accMean", mean)

mIntent.putExtra("accVar", variance)

mIntent.putExtra("accMCR", MCR)

mIntent.putExtra("label", label)

mIntent.action = ACTION_CLASSIFIER_TRAINING

TrainClassifierService.enqueueWork(applicationContext, mIntent)

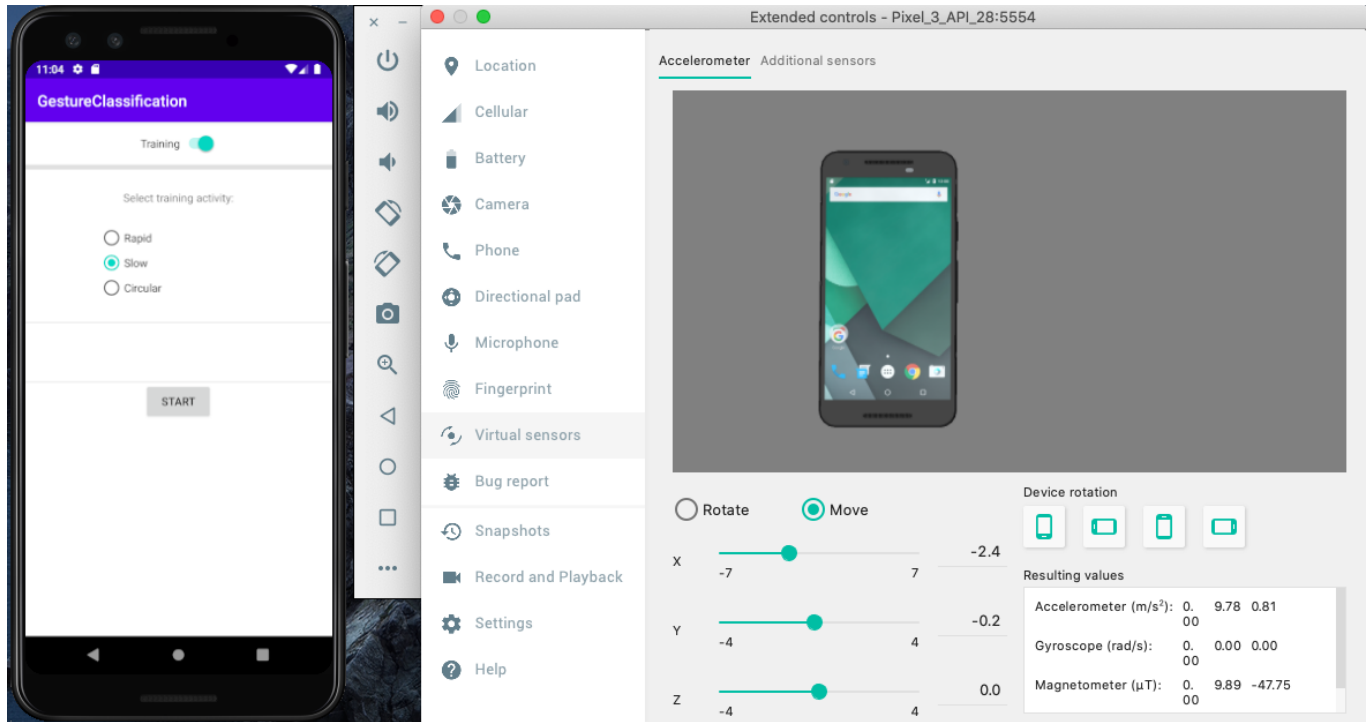
```

Finally, in `TrainClassifierService` after the training you can check the state of your classifier by calling the following (the results will be printed in `LogCat`):

```
c.printClassifierInfo()
```

Movement inference

Turn the Training switch on, start sensing, and ensure that you train your classifier with different phone movements. You can use an emulator with virtual sensors (see below) - moving the virtual phone left-right, up-down, etc. works reasonably well.



Once done with training, switch the training off on the main screen, and let's try to infer the movement - is it rapid, slow, circular (or any category you want to recognise). Since we might be moving the phone rapidly, printing the result in a `TextView` or showing a `Toast` with the inference label might not be readable. Instead, we suggest that you also change the background color of the parent `ConstraintLayout` of the `MainActivity` (the ID of the View is `container`) depending on the movement (e.g. let it be red for one gesture, blue for another, etc.).

The classification is done via `classifier.classify()` method, that requires an Instance consisting of features -- mean, variance, MCR -- which we are trying to find a label for:

```
val c = mManager?.getClassifier("movementClassifier")

val instanceValues = ArrayList<Value>()

val meanValue = Value(mean.toDouble(), Value.NUMERIC_VALUE)

instanceValues.add(meanValue)

// TODO: add two more features here

val instance = Instance(instanceValues)

// TODO: a try-catch block is needed here

val inference = c?.classify(instance)
```

`inference?.value.toString()` gives you the string value of the inference. Check whether this value is equal to `R.string.txt_gesture_1`, `R.string.txt_gesture_2`, or `R.string.txt_gesture_3` and set the value of the `R.id.tv_result` `TextView`. Also, depending on the inference, change the background of the screen:

```
val v = binding.container

v.setBackgroundColor(Color.BLUE) // or any other colour
```

Where to do the inference? In this lab we will do it in `MainActivity` (see `recordAccData` function). However, with more complex machine learning models, inference can be computationally expensive and should be done in a separate `JobIntentService`.

Now you can run the app and try out both training and inference. First, turn the Training switch on, start sensing, and train your classifier with different phone movements. Then, turn the switch off to go to inference mode, and click start. Move the phone using different movements - you should see the classification result shown in the `TextView` and the background color changing accordingly.

If you did not attend this lab session in-person you can commit your code to a private repository named **PBD2023-LAB-7** in your Bitbucket account. User **pbdfrita** (pbdfrita@gmail.com) must be added as a read only member of this repository. The code must be committed by Sunday (April 23th) 23:59.

Happy coding!