Evaluation report on state of the art algorithms for sustainability services

Technical report on multi modal tracking and activity mode recognition

Version 1

Deliverable 3.2

Project title:	SimpliCITY – Marketplace for user-centered sustainability services
Project acronym:	SimpliCITY
Project duration:	10/2018–03/2021
Project number:	870739
Work package/Task:	WP3 / T3.2.
Project website:	www.simplicity-project.eu

Authors:

Thomas Layer-Wagner, Polycular OG
Christoph Wögerbauer, Polycular OG
Michael Kager, Polycular OG
Birgit Schönauer, Polycular OG
Irina Paraschivoiu, Polycular OG

URBAN EUROPE

This project has received funding in the framework of the Joint Programming Initiative Urban Europe.

Document versions:

Version	Date	Changes	Author/s
v0.1	23.10.2018	Technical report on multi modal tracking and activity mode recognition (outline)	Thomas Layer-Wagner, Robert Praxmarer
v0.2	10.12.2018	Activity Recognition APIs and Custom Activity Recognition (draft)	Thomas Layer-Wagner, Birgit Schönauer
v0.3	27.09.2019	Update and Revision, Conclusion and Lessons Learned	Thomas Layer-Wagner, Robert Praxmarer, Birgit Schönauer
v1.0	30.09.2019	Update and Revision	Thomas Layer-Wagner, Birgit Schönauer

List of abbreviations

AGPS	Assisted Global Positioning System
ANN	Artificial Neural Network
API	Application Program Interface
CNN	Convolutional Neural Network
DS	Down Sampling
ELM	Extreme Learning Machine
GPS	Global Positioning System
HAR	Human Activity Recognition
LBP	Locally Binary Pattern
WiFi	Wireless Fidelity
WL	Window Length

Table of contents

1 Executive Summary 4
2 Administrative Information 5
3 Introduction 6
4 Activity Recognition API 8
5 Custom Activity Recognition 25
6 Conclusions 32
7 Lessons learned 35
8 References 37

1 Executive Summary

SimpliCITY will be a service platform, developed by a project consortium and will be implemented and tested in the cities of Salzburg (AT) and Uppsala (SWE). The platform will use incentivisation and nudging to boost green services on a city level. The project focus is on three areas namely bike mobility, local production and consumption and social inclusion. For cities and their city managers it should become easier to promote and boost regional green services through a unified channel and platform. The platform will provide information, incentives and challenges to support services, so that not every single service has to have its own incentivisation and nudging system, but one throughout the city, that ties the different city services together and creates a shared user base.

In the SimpliCITY mobile application “cycling” as green and sustainable mode of transportation constitutes a central topic. Users will be able to start bike tracking and receive not only information on environment and health benefits but also collect points when they finish.

This report discusses the requirements and implications of activity mode recognition of citizens for SimpliCITY. Activity mode recognition allows the deduction of a transportation mode based on collected data from built-in sensors in mobile devices. Basic activity modes are e.g.: “on foot”, “cycling”, “automobile”, “still”. By determining the activity mode users won’t be able to cheat and collect points for a route, they have driven in a car.

In addition, the multi modal detection of activities potentially allows to reward various activity modes differently. It also offers interesting insights on how users travel the city and where they switch to another mode of transportation.

Aim is to research and evaluate existing solutions for activity mode recognition and compare them with state-of-the-art research.

2 Administrative Information

Basic information on the SimpliCITY project and the present deliverable:

Project title	SimpliCITY - Marketplace for user-centered sustainability services
Project coordinator	Salzburg Research Forschungsgesellschaft mbH (SRFG), Salzburg, Austria; project manager: Petra Stabauer BSc MSc
Project partners	Polycular OG, Hallein, Austria Stadt Salzburg (City of Salzburg), Austria Salzburger Institut für Raumordnung und Wohnen – SIR (Salzburg Institute for Regional Planning & Housing), Salzburg, Austria Uppsala Kommun (City of Uppsala), Sweden University of Uppsala, Sweden
Funding	JPI Urban Europe, Innovation Actions (Call: Making Cities Work) Funding is being provided by Vinnova (Sweden) for the Swedish project partners, and the Austrian Research Promotion Agency (FFG) for the Austrian project partners.
Project nr.	870739
Deliverable number	D3.2
Deliverable title	Technical report on multi modal tracking and activity mode recognition
Authors	Thomas Layer-Wagner (Polycular), Robert Praxmarer (Polycular), Birgit Schönauer (Polycular)
Version & status	Version 1
Date	30.09.2019

3 Introduction

In the SimpliCITY mobile application "cycling" as green and sustainable mode of transportation constitutes a central topic. We will reward going distances by bike (or another sustainable way). Users collect points for cycling but will also receive interesting information on environmental and health benefits for finishing a route on bike instead of going by car. Therefore, it is necessary to have some means for detecting the user's mode of transportation.

Activity mode recognition APIs allow the deduction of different transportation modes based on collected data from built-in sensors in mobile devices (Fig. 1).

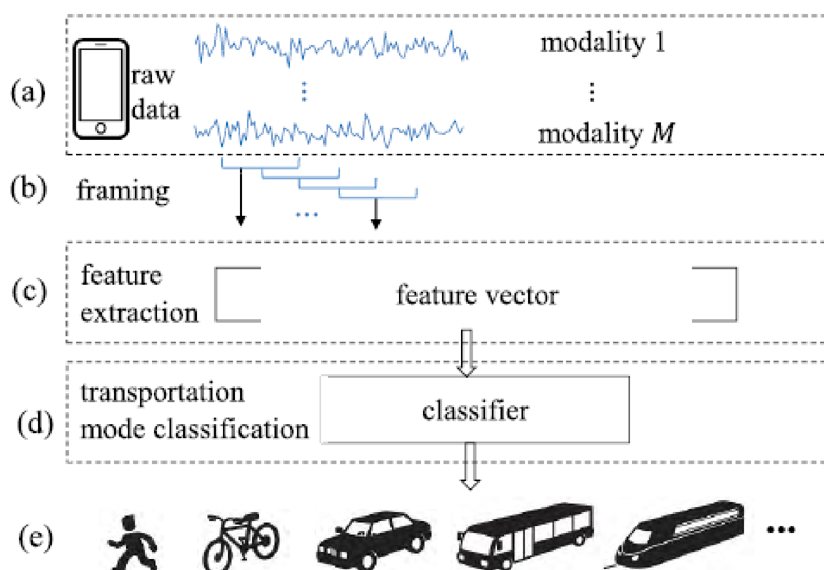

Figure 1 taken from Wang 2019, p.10871
Built-in sensors provide raw data, from which features are derived and a feature vector is calculated. A classifier maps the feature vector to a transportation mode.

Given the near ubiquity of mobile phones and their various built-in sensors, these devices are the perfect means to gather rich sensor data in relation to user Multi Modal Tracking and Activity Recognition. The recognition of user activity modes bases on **location** and **motion systems**.
In general, radio frequency signals are used to determine the location of a mobile phone (mainly GPS, AGPS, WiFi, Bluetooth and cellular networks), built-in sensors (e.g. accelerometer and gyroscope) provide information on motion data for Activity Mode deduction (e.g. walking, cycling, …).

The geographic location and orientation can be determined through different APIs. Most prominent ones are mentioned here:

- Google Maps (cross platform)
- Core Location (iOS -> Apple) & Android.Location (Android -> Google) subscribe to changes in user' location specified by distance /time
- as well as various Geolocator Plugins for Xamarin, React Native

Basic activity modes are e.g.: “on foot”, “cycling”, “automobile”, “still”. By determining the activity mode users won’t be able to cheat and to collect points for a route, they have driven in a car.

In addition, the multi modal detection of activities potentially allows to reward various activity modes differently. It also offers interesting insights on how users travel the city and where they switch to another mode of transportation.

The following activity types are considered in the SimpliCITY mobile app:

Type	Type String	Description
0	Still	The user (device) is not moving STILL in Google Activity Recognition API STATIONARY in Apple Activity Recognition API
1	Walking	The user is on foot, walking. WALKING in Google Activity Recognition API (is subcategory of ON FOOT) WALKING in Apple Activity Recognition API
2	Running	The user is on foot, running. RUNNING in Google Activity Recognition API (is subcategory of ON FOOT) RUNNING in Apple Activity Recognition API
3	Cycling	The user is cycling. ON_BICYCLE in Google Activity Recognition API CYCLING in Apple Activity Recognition API
4	Automotive	The user is a car or other vehicle. IN_VEHICLE in Google Activity Recognition API AUTOMOTIVE in Apple Activity Recognition API
5	Unknown	There was no activity recognized. Often occurring when there is an activity transition. UNKNOWN, TILTING in Google Activity Recognition API UNKNOWN in Apple Activity Recognition API

There are two approaches for recognizing activity modes in an application.
First, Activity Recognition APIs eliminate the need to define fine grained heuristics for a custom Activity Mode Recognition and are available for different platforms. Some solutions target cross platform applications. Available Activity Recognition APIs will be characterized in short, including their range of detectable activities and information on power consumption, if available.

Second, creating a Custom Activity Recognition.A state-of-the-art approach will be described in chapter “Custom Activity Recognition”. In general, much more time and effort must be invested in the creation of a custom-made solution, which should outperform available APIs in some of the following characteristics, to make worth the effort.

Ideally, Activity Recognition considers the following:

- accuracy (also in relation to the location of the mobile phone on the user’s body / in car, …)
- latency should be within an acceptable range for the according application
- low power consumption (when possible, e.g. activity mode still with low update rates)
- complexity and real-time capability (-> Custom Activity Recognition)

Ultimately, there is always a trade-off between accuracy, power consumption and real-time updates.

Activity Recognition API

This section consists of two parts. In the first part, we provide a short overview on main characteristics of various available activity recognition APIs for different platforms. We also address important topics like power consumption and accuracy.

The second part introduces each API and describes implementation as well as usage details.

4.1.1 Detected Activity Modes for Different Platforms in Overview:

ANDROID

- **Google:**
  Activity Modes: still, on_foot (incl. subactivities: walking, running), on_bicycle, in_vehicle, tilting (e.g. device pick-up), unknown
  Functionality: uses machine learning and only on-board smartphone sensors

iOS

- **Apple:**
  Activity Modes: stationary, walking, running, automotive, cycling, unknown
  Functionality: coprocessor processes data from built-in sensors and derives activity modes using a neural network

- **LocoKit – open source (announces Android SDK coming soon)**
  Activity Modes: stationary, walking, running, cycling, automotive (distinguishes car, train, bus, motorcycle, airplane, boat)
  Functionality: machine learning
  allows queries e.g. timeline items for a specific geographic region, for a specific activity mode, timeline items above/below a speed value

CROSS PLATFORM

- **React-native, Xamarin, Flutter or Unity Activity Recognition**
  wrapping Android and iOS functionality

- **PathSense (for Android 2.3+** (Api Level 9) **or iOS) – free**
  https://pathsense.com/awesomeactivity

  Activity Modes: walking, driving, holding, still, shaking, in-vehicle holding
  Functionality: machine learning.
  Models for additional activities (e.g. cycling) are planned
  Includes Location service
  Claims to be 6 x faster, more accurate and ½ battery consumption compared to Google Activity Recognition API.
  This solution was not considered, since cycling detection has not been implemented yet.

- **Tizen – not considered**
  https://developer.tizen.org/dev-guide/tizen-iot-headed/latest/group__CAPI__CONTEXT__ACTIVITY__MODULE.html#gae17e97a1a51a9d5d5d8330f29f4a895d
  Activity Modes: stationary, walk, run, in_vehicle
  Again lack of cycling detection ruled this solution out.

4.1.2 Power Consumption

Google's Activity Recognition for Android

Google's recognition detection allows to **set a detection interval** for the updating frequency of activities (detectionIntervalMillis). Smaller values result in more frequent activity updates and therefore an increased power consumption. Since only **data from built-in sensors is used battery consumption is reduced**.
If the device is still for an extended period, the activity reporting may stop and resume once the device is moving again. This **conserving battery function** is only available for devices which support the Sensor.TYPE_SIGNIFICANT_MOTION hardware.
Beginning in API 21, activities may be received less frequently than the detectionIntervalMillis parameter if the device is in power save mode and the screen is off.

Apple's Activity Recognition for iOS

Devices use a motion coprocessor thus running all sensor processing on this dedicated hardware and reducing the CPU load and reducing energy usage.

LocoKit Activity Recognition for iOS

LocomotionManager dynamically adjusts various device monitoring parameters, balancing current conditions and desired results to achieve the desired accuracy in the most energy efficient manner.

4.1.3 Accuracy

Activities do not only have a type (e.g. “Walking”) but also provide a confidence value of detection. In Android the confidence value ranges from 0 to 100, whereas in iOS confidences of Low, Medium and High are offered. Since machine learning is used for Activity Recognition, it is assumed that used models are updated by Google and Apple at some frequency. Therefore, reviews on different characteristics should be interpreted in relation to the time of testing.

In general, there are surprisingly few evaluations on proprietary Activity Recognition APIs. Apart from improvements from older to current API-versions provided by Google/Apple, there are some few reviews by companies which took part in beta testing. Still, most of these (e.g. Stogaitis 2018) simply mention an overall improvement of accuracy and less power consumption using the latest API. No evaluation comprised of in-depth-data like accuracy percentages for each mode.

Overall, there will always be some latencies in Activity mode detection because the transition from one state to another does necessarily take some time. Some reviews mentioned though, that latencies were reduced when the mobile phone was in a somewhat fixed position (e.g. mounted in car) rather than being pocketed or handheld.

Reviews on Google’s Activity Recognition for Android

2018 reviews: Cross 2018, Prajakt 2018

- **Latency**
  - 1 min. reporting activity transition from actual transition
  - Appears biased towards in_vehicle, as this transition more frequently arrived sooner than walking
  - In_vehicle detected quickly when mounted, having a higher latency when in cup holder

- **Accuracy**
  - Fairly accurate when actual activity transitions occurred
  - Walking was not always detected, esp. when in hand and walking
  - In slow motion traffic, a continuous drive would be reported as a mix of: in_vehicle, on_bicycle, still
  - No false positives

- **Battery results**
  - less battery drains than before
    Both devices were fully charged at test start.
    Test duration: 50min.
    The Activity Transitions API ran in the background 90% of the time, both screens turned off
  - Samsung S8: 67% remaining
  - Google Pixel: 80% remaining

2017 review: Zhong 2017

- **Latency**
  - ranging from 4 - 12 seconds (cycling mode). Here, it was also noted that different interval settings did not have any impact on the accuracy.

- **Accuracy**
  - most problems with stationary (39% - got mixed up with tilting, unknown), cycling 68% (often confused with tilting esp. when traveling on uneven roads) other modes ranging from 73% to 88%.

2013 review: Jackpotek 2013

- **Accuracy**
  - (tested driving, on foot, bike) very accurate for on_foot and in_vehicle
    Riding a bike included many tilt and unknown:

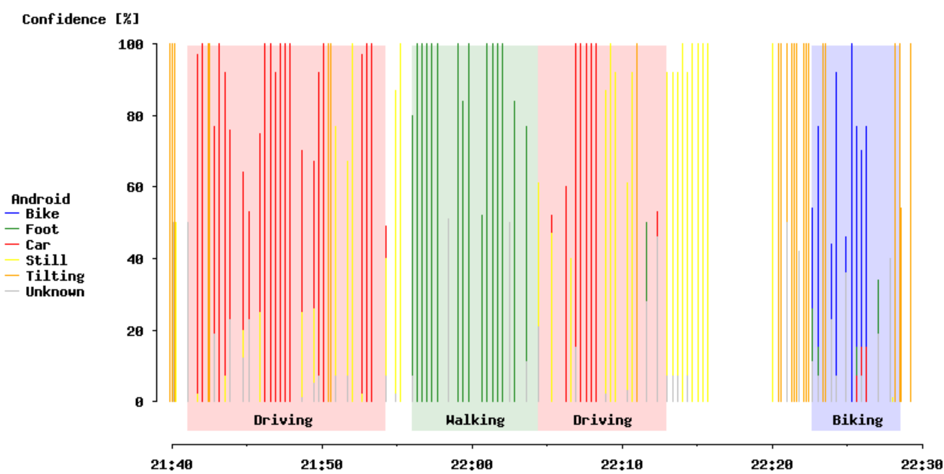

Figure 2 taken from Jackpotek 2013
This figure shows detected activities while traveling in different modes. As the figure shows, walking was accurately detected, whereas various modes were recognized while riding a bike.

Reviews on Apple's Activity Recognition for iOS

Surprisingly, hardly any reports on the accuracy of Apple's CoreMotion package could be found. Similar to the Google's API some problems have been reported with correct cycling mode detection (Veugen 2015). Only advertising articles (Veugen 2015) on new coprocessor releases mention improved processing of motion and location data through gathered data, while reducing battery drain.

2014 review: Apple WWDC 2014

- **Latency**
  - running, walking (5-10sec) - vehicle very fast, when in cup-holder, otherwise about the time of walking, cycling taking the most time to be detected (no nr of seconds provided "a lot longer")

- **Accuracy**
  - overall high

4.2 Cross-Platform Activity Recognition

These solutions wrap proprietary Activity Recognition APIs from Google (Android) and Apple (iOS).

4.2.1 React native activity recognition (Artistic License 2.0)

https://www.npmjs.com/package/react-native-activity-recognition

Updated January 7th and tested with react-native v0.57.5

This chapter includes installation and setup details as well as a short description on using the API on each platform.

Installation

```
npm i -S react-native-activity-recognition
```

or with Yarn:

```
yarn add react-native-activity-recognition
```

Linking (Automatic)

```
react-native link react-native-activity-recognition
```

You also must manually add the according permission in the manifest file in Android (see below: Android, Step 4) and the according key to Info.plist in iOS (see below: iOS, Step 4).

Linking (Manually)

Android

1. Add following lines to `android/settings.gradle`

```
...
include ':react-native-activity-recognition'
project(':react-native-activity-recognition').projectDir =
new File(rootProject.projectDir, '../node_modules/react-native-activity-
recognition/android')
...
```

2. Add the compile line to dependencies in `android/app/build.gradle`

```
...
dependencies {
       ...
       compile project(':react-native-activity-recognition')
       ...
}
```

3. Add import and link the package in `android/app/src/…/MainApplication.java`

```
import com.xebia.activityrecognition.RNActivityRecognitionPackage; // add import

public class MainApplication extends Application implements ReactApplication {
       // …
```

```
        @Override
        protected List<ReactPackage> getPackages() {
            return Arrays.<ReactPackage>asList(
                new MainReactPackage(),
                // …
                new RNActivityRecognitionPackage() // add package
            );
       }
```

4. Add activityrecognition service in `android/app/src/main/AndroidManifest.xml`

```
...
<application ...>
    ...
    <service android:name="com.xebia.activityrecognition.DetectionService"/>
  ...
</application>
...
```

iOS

1. In the XCode's "Project navigator", right click on your project's Libraries folder ➜ `Add Files to <...>`
2. Go to `node_modules` ➜ `react-native-activity-recognition` ➜ `ios` ➜ select `RNActivityRecognition.xcodeproj`
3. Add `RNActivityRecognition.a` to `Build Phases -> Link Binary With Libraries`
4. Add `NSMotionUsageDescription` key to your `Info.plist` with strings describing why your app needs this permission

Usage

```
import ActivityRecognition from 'react-native-activity-recognition'
...
// Subscribe to updates
this.unsubscribe = ActivityRecognition.subscribe(detectedActivities => {
  const mostProbableActivity = detectedActivities.sorted[0]
})
...
// Start activity detection
const detectionIntervalMillis = 1000
ActivityRecognition.start(detectionIntervalMillis)
...
// Stop activity detection and remove the listener
ActivityRecognition.stop()
this.unsubscribe()
```

Android

`detectedActivities` is an object with keys for each detected activity, each of which have an integer percentage (0-100) indicating the likelihood – confidence - that the user is performing this activity. For example:

```
{
  ON_FOOT: 8,
  IN_VEHICLE: 15,
  WALKING: 8,
  STILL: 77
}
```

Additionally, the `detectedActivities.sorted` getter is provided which returns an array of activities, ordered by their confidence value:

```
[
  { type: 'STILL', confidence: 77 },
  { type: 'IN_VEHICLE', confidence: 15 },
  { type: 'ON_FOOT', confidence: 8 },
  { type: 'WALKING', confidence: 8 },
]
```

Because the activities are sorted by confidence level, the first value will be the one with the highest probability. ON_FOOT and WALKING are related but won't always have the same value.

The following activity types are supported:

- IN_VEHICLE
- ON_BICYCLE
- ON_FOOT
- RUNNING
- WALKING
- STILL
- TILTING
- UNKNOWN

iOS

`detectedActivities` is an object with key to the detected activity with a confidence value for that activity given by CMMotionActivityManager. Confidence range in iOS is 0-2 according to confidence enums Low, Medium, High. For example:

```
{
    WALKING: 2
}
```

`detectedActivities.sorted` getter will return it in the form of an array.

```
[
    {type: "WALKING", confidence: 2}
]
```

The following activity types are supported:

- RUNNING
- WALKING
- STATIONARY
- AUTOMOTIVE
- CYCLING
- UNKNOWN

4.2.2 Xamarin.Forms for iOS and Android

http://www.devsdna.com/blog/ArticleID/18/Activity-recognition

Every platform has its own API, objects and definitions. Therefore, it is necessary to create some shared artefacts to translate platform specific code to shared core code.

After defining a model class ActivityRecognized and a service interface, this service needs to be implemented on each platform.

```
public enum ActivityTypes
{
        Stopped = 0,
        Walking = 1,
        Running = 2,
        OnBicycle = 3,
        OnVehicle = 4
}

public class ActivityRecognized
{
    public ActivityTypes ActivityType { get; set; }
    public int Confidence { get; set; }
}

public class ActivityChangedEventArgs : EventArgs
{
    public ActivityChangedEventArgs(ActivityRecognized activity)
    {
        Activity = activity;
    }
    public ActivityRecognized Activity { get; set; }
}

public interface IRecognitionActivityService
    {
        event EventHandler ActivityChanged;
        ActivityRecognized LastActivity { get; }
        void StartService();
        void StopService();
    }
```

Android needs to have installed the `Xamarin.GooglePlayServices.Location` nuget package to access the GooglePlay Services. In addition, in the `manifest` file permission for `ActivityRecognition` must be requested.

To connect to GooglePlay the *Callback* must be set to receive the response.

```
public class RecognitionActivityService : Java.Lang.Object,
GoogleApiClient.IConnectionCallbacks, GoogleApiClient.IOnConnectionFailedListener,
IRecognitionActivityService
```

iOS must create a CMMotionActivityManager which will be responsible for starting and stopping the Recognition service.

Example implementation:
https://github.com/DevsDNA/DevsDNAActivityRecognitionSample

Activity recognition has to be implemented according to Google, iOS specifications. A recognized activity will be passed to the (ActivityChanged). By subscribing to the event, we can act on updated activities.

4.2.3 Flutter

https://flutter.dev/
https://pub.dev/packages/activity_recognition_alt
https://github.com/tonywei92/flutter_activity_recognition

Activity recognition plugin for Android and iOS. Only working while App is running (= not terminated by the user or OS)

Android Integration

- Add permission to manifest

```
<uses-permission android:name="com.google.android.gms.permission.ACTIVITY_RECOGNITION" />
```

- Add plugin

```
<service
android:name="at.resiverbindet.activityrecognition.activity.ActivityRecognizedService" />
```

IOS Integration
An iOS app linked on or after iOS 10.0 must include usage description keys in its *Info.plist* file for the types of data it needs. Failure to include these keys will cause the app to crash. To access motion and fitness data specifically, it must include `NSMotionUsageDescription`.

```
import
'package:activity_recognition/activity_recognition.dart';ActivityRecognition.activityUpdate
s()
```

GitHub providing example project.

4.2.4 Unity Activity Recognition (Price €4.47, no ratings yet)

https://assetstore.unity.com/packages/tools/integration/user-activity-recognition-140756
http://www.kokosoft.pl/user-activity-recognition-docs/

First released on Apr 16, 2019 for Unity versions 2017.3.0 or higher

- Works on both iOS (versions >= 11.0.) and Android (>= 4.0)
- Uses motion detection built in phone devices
- Supports driving, biking, running, stationary and walking
- example scene, doc includes project setup

4.3 Activity Recognition in Android

4.3.1 Google Activity Recognition

https://developers.google.com/location-context/activity-recognition/

The Activity Recognition API is built on top of available device sensors and automatically detects activities by periodically reading sensor data and processing them using machine learning models. If the device has been still for a while the API may stop activity reporting to reduce power consumption and resumes reporting on movement.

The API delivers its results to a callback (IntentService) at specified intervals or the app can use the results requested by other clients without consuming additional power itself. By using a PendingIntent you define how the API delivers results and prevent a constantly running service in the background. Detected activities are sent as list, each activity including a confidence level as well as type properties.

Setting up the API includes several steps:
https://codelabs.developers.google.com/codelabs/activity-recognition-transition/index.html?index=..%2F..index#1

1. Add Google Play Services API
2. Add permissions to the app manifest
3. Register for activity updates
4. Process events
5. De-register updates

1. **Add Google Play Services APIs**
   https://developer.android.com/guide/topics/location/transitions#java

   In the modules `build.gradle` file add a new build rule under `dependencies`: (check https://developers.google.com/android/guides/setup for current play-services-location link, as well as general adding of Google Play Services)

```
apply plugin: 'com.android.application'
…
dependencies {
    implementation ‘com.google.android.gms:play-services-location:17.0.0'
}
```

The projects `build.gradle` file needs a reference to the `google()` repo:

```
repositories {
    google()
    …
}
…
```

```
Allrepositories {
    google()
}
```

Alternatively include a reference to the maven `{ url "https://maven.google.com" }`

2. Add permission for the Activity Recognition API in the manifest

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
         package="com.example.myapp">

   <uses-permission android:name="com.google.android.gms.permission.ACTIVITY_RECOGNITION"/>
  ...
</manifest>
```

Up to Android 9 permission for Activity Recognition is granted automatically if the above snippet is added to the manifest. In Android 10 the user has to grant permission explicitly unless the app is upgraded to this API and permission has already been granted.

Following, steps for receiving updates on Activity transitions are described. It is also possible to register for Activity updates at a user defined time interval.

3. Registering for Updates

To start receiving notifications about activity transitions, you must implement the following:

- An ActivityTransitionRequest object that specifies the type of activity and transition
- A PendingIntent callback where your app receives notifications.

First, create a list of ActivityTransition objects, each having an activity type (DetectedActivity: in_vehicle, on_bicycle, running, walking, still).
Transition types are:

- ACTIVITY_TRANSITION_ENTER
- ACTIVITY_TRANSITION_EXIT

To create the ActivityTransitionRequest object, create a list of objects, which represent the transition that you want to receive notifications about.

The following code shows how to create a list of ActivityTransition objects:

```
List<ActivityTransition> transitions = new ArrayList<>();

transitions.add(
  new ActivityTransition.Builder()
    .setActivityType(DetectedActivity.WALKING)
    .setActivityTransition(ActivityTransition.ACTIVITY_TRANSITION_ENTER)
    .build());

transitions.add(
  new ActivityTransition.Builder()
    .setActivityType(DetectedActivity.WALKING)
    .setActivityTransition(ActivityTransition.ACTIVITY_TRANSITION_EXIT)
    .build());

transitions.add(
  new ActivityTransition.Builder()
```

```
    .setActivityType(DetectedActivity.STILL)
    .setActivityTransition(ActivityTransition.ACTIVITY_TRANSITION_ENTER)
    .build());

transitions.add(
  new ActivityTransition.Builder()
    .setActivityType(DetectedActivity.STILL)
    .setActivityTransition(ActivityTransition.ACTIVITY_TRANSITION_EXIT)
    .build());
```

Then, you can create an ActivityTransitionRequest object by passing the list of ActivityTransitions to the ActivityTransitionRequest class:

```
ActivityTransitionRequest request = new ActivityTransitionRequest(transitions);
```

Register for activity transition updates by passing your instance of ActivityTransitionRequest and your PendingIntent object to the requestActivityTransitionUpdates() method. The requestActivityTransitionUpdates() method returns a Task object that can be checked for success or failure, as shown in the following code example:

```
// myPendingIntent is the instance of PendingIntent where the app receives callbacks
Task<Void> task =
  ActivityRecognition.getClient(context)
                     .requestActivityTransitionUpdates(request, myPendingIntent);

  task.addOnSuccessListener(
    new OnSuccessListener<Void>() {
      @Override
      public void onSuccess(Void result) {
        // Handle success
      }
    }
  );

  task.addOnFailureListener(
    new OnFailureListener() {
      @Override
      public void onFailure(Exception e) {
        // Handle error
      }
    }
  );
```

4. **Processing Events**
   When the requested activity transition occurs, your app receives an Intent callback. An ActivityTransitionResult object can be extracted from the Intent, which includes a list of ActivityTransitionEvent objects.
   The events are ordered in chronological order, for example, if an app requests for the IN_VEHICLE activity type on the ACTIVITY_TRANSITION_ENTER and ACTIVITY_TRANSITION_EXIT transitions, then it receives an ActivityTransitionEvent

object when the user starts driving, and another one when the user transitions to any other activity.
You can implement your callback by creating a subclass of BroadcastReceiver and implementing the onReceive() method to get the list of activity transition events. For more information, see Broadcasts. The following example shows how to implement the onReceive() method:

```
@Override
protected void onReceive(Context context, Intent intent) {
  if (ActivityTransitionResult.hasResult(intent)) {
    ActivityTransitionResult result = ActivityTransitionResult.extractResult(intent);
    for (ActivityTransitionEvent event : result.getTransitionEvents()) {
        // chronological sequence of events....
    }
  }
}
```

5. **De-registering Updates**

You can deregister for activity transition updates by calling the removeActivityTransitionUpdates() method of the ActivityRecognitionClient and passing your PendingIntent object as a parameter, as shown in the following example:

```
// myPendingIntent is the instance of PendingIntent where the app receives callbacks
Task<Void> task =
  ActivityRecognition.getClient(context).removeActivityTransitionUpdates(myPendingIntent);

task.addOnSuccessListener(
  new OnSuccessListener<Void>() {
    @Override
    public void onSuccess(Void result) {
      myPendingIntent.cancel();
    }
  });

task.addOnFailureListener(
  new OnFailureListener() {
    @Override
    public void onFailure(Exception e) {
      Log.e("MYCOMPONENT", e.getMessage());
    }
  });
```

Example projects:
https://github.com/googlecodelabs/activity_transitionapi-codelab
https://github.com/tutsplus/Android-ActivityRecognition

4.4 Activity Recognition in iOS

4.4.1 Apple: CMMotionActivityManager - Core Motion framework

https://developer.apple.com/documentation/coremotion/cmmotionactivitymanager

Core Motion reports motion- and environment-related data from the onboard hardware of iOS devices, including from the accelerometers and gyroscopes, and from the pedometer, magnetometer, and barometer. Core Motion supported devices are equipped with a motion coprocessor (first M-series coprocessor was shipped in the iPhone 5S, Sept 2013), which processes data from accelerometer, gyroscope and compass and deducts activity modes therefrom. By using dedicated hardware, the system can offload all sensor processing from the CPU and minimize energy usage. Even if the device is in power save mode, this functionality is not limited.
Although Activity Recognition has been provided since the introduction of the M7 coprocessor, cycling mode detection was added with the M8 coprocessore (iOS 8).

In order to use the activity manager and receive updates on activities, it is necessary to create an activity manager and use the startActivityUpdate method. Every time the device updates the motion activity, it executes the specified closure, passing a CMMotionActivity object.

```
let manager = CMMotionActivityManager()
manager.startActivityUpdates(to: .main) { (activity) in
    guard let activity = activity else {
        return
    }

    var modes: Set<String> = []
    if activity.walking {
        modes.insert("🚶")
    }

    if activity.running {
        modes.insert("🏃")
    }

    if activity.cycling {
        modes.insert("🚴")
    }

    if activity.automotive {
        modes.insert("🚗")
    }

    print(modes.joined(separator: ", "))
}
```

As the documentation states, motion-related properties are not mutually exclusive. Therefore, more than one motion-related property can have the value true. For example, if the user was driving in a car and the car stopped at a red light, the update event associated with that change in motion would have both the cycling and stationary properties set to true.

Each CMMotionActivity object includes a confidence property (.low, .medium, .high) and a startTime.

Wiki: https://wiki.appcelerator.org/display/guides2/Core+Motion+Module#CoreMotionModule-Activity

4.4.2 LocoKit

https://github.com/sobri909/LocoKit

A Machine Learning based location recording and activity detection framework for iOS. The LocomotionManager monitors raw device location and motion data and applies filtering and smoothing algorithms to procute a stream of high level LocomotionSample objects, a composite representation of the device and user's location and activity state at each point in time.

Activity Type Detection

- Machine Learning based activity type detection
- Improved detection of Core Motion activity types (stationary, walking, running, cycling, automotive)
- Distinguish between specific transport types (car, train, bus, motorcycle, airplane, boat)

Record High Level Visits and Paths (timelineItem)

- Optionally produce high level Path and Visit timeline items, to represent the recording session at human level. Similar to Core Location's CLVisit, but with much higher accuracy, much more detail, and with the addition of Paths (e.g. the trips between Visits).
- Optionally persist your recorded samples and timeline items to a local SQL based store, for retention between sessions.

Each `TimelineItem` is a high-level grouping of samples representing a Visit or a Path. Inside each `TimelineItem` there is a time ordered array of `LocomotionSample` samples (`timelineItem.samples`), first being the start, last being the stop. If data accuracy is high, new samples will be produced about every 6 seconds. The maximum frequency is configurable with `TimelineManager.samplesPerMinute`.

A path timeline item will have an `activityType` of `.walking`. Other samples of this item might have `.stationary`, if a person was stopping for a few seconds in between.

Location and Motion Recording:

- Combined, simplified Core Location and Core Motion recording
- Filtered, smoothed, and simplified location and motion data
- Near real time stationary / moving state detection
- Automatic energy use management, enabling all day recording
- Automatic stopping and restarting of recording, to avoid wasteful battery use

Installation

```
pod 'LocoKit'
pod 'LocoKit/LocalStore' # optional
```

Note: Include the optional LocoKit/LocalStore subspec if you would like to retain your samples and timeline items in the SQL persistent store.

Instruction for High or Low Level Recording: https://github.com/sobri909/LocoKit

Fetching TimelineItems / Samples

If you wanted to get all timeline items between the start of today and now, you might do this:

```
let date = Date() // some specific day
let items = store.items(
        where: "deleted = 0 AND endDate > ? AND startDate < ? ORDER BY endDate",
        arguments: [date.startOfDay, date.endOfDay])
```

Complex Queries

You can also construct more **complex queries**, like for fetching all timeline items that overlap a certain geographic region. Or all samples of a specific activity type (eg all "car" samples). Or all timeline items that contain samples over a certain speed (eg paths containing fast driving).

LocoKit Demo App available

5 Custom Activity Recognition

A first presented custom solution is rather more an enhancement of the existing Google Activity Recognition AR. Although it is not the current state-of-the-art it still constitutes an effective and above all easy to implement solution with provided open source code.

The second presented solution can be rated as state-of-the-art and uses machine learning for Activity Recognition.

5.1.1 ARshell+

https://github.com/myzhong/ARshell

Zhong (Zhong 2017) proposed ARshell+ to improve the accuracy of Google's Activity Recognition. The **accuracy of activity detection using ARshell+ is increased significantly,** resulting in an **average accuracy of 91%** compared to 69,8% for the basic Google Activity Recognition AR. ARshell+ basically uses Android detected Activities and applies certain functions on this data to derive Activities.

The proposed solution was compared in relation to CPU usage, memory occupancy and power consumption. ARshell+ did not have any impact on CPU usage but resulted in an increased power consumption of 0.24 W. For memory occupancy Resident Set Size RSS (memory occupied by a process that is held in the physical memory) and Virtual Set Size VSS (virtual memory occupied by a process in total) were evaluated. Google activity recognition used 7516 K RSS and 4040 K VSS, whereas ARshell+ occupied 10324 K RSS and 5340 K VSS.

ARshell+ provides two mechanisms for delivery of the activity recognition results:

- A pull-based mechanism that replies with the latest activity when requested, supporting request up to every second
- A notification-based mechanism that updates the recognition result per interval; notifications are sent when an activity change is detected or per interval depending on how ARshell+ is configured.

How does it work?
ARshell+ combines the functionality of ARshell (Zhong 2015) and ARsignal.
When the Google Activity Recognition reports a recognition result x_t (list of probable activities, consisting of tuples (probable activity a, confidence value cv)) for time *t*, ARshell applies the Markov model to smooth the result and generates y_t.

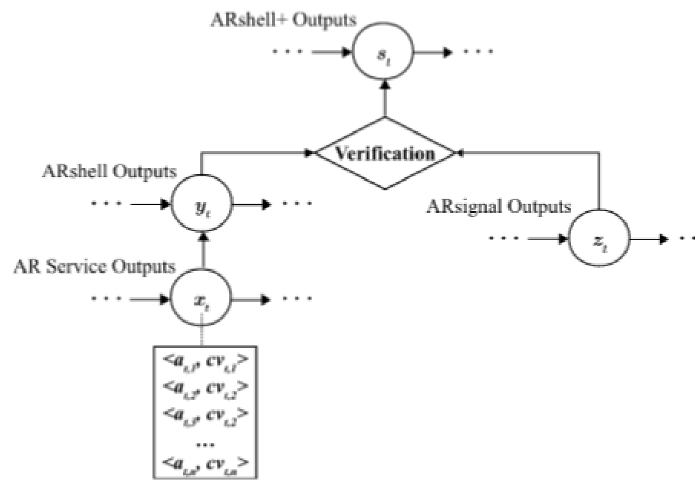

Figure 3 taken from Zhong 2015, p.42
ARshell+ flow diagram

There are 4 transitions that need to be addressed by ARshell:

1) From *unknown* to a *specific activity*
   The most probable activity is proposed for y_t
2) From a *specific activity* to *unknown*
   Assumption: the last historical activity is the current activity $y_t=y_{t-1}$, with a self-transition probability (P= 1−ε, whereεis a sufficiently small number)
3) Self-transition
   A specific activity continues into next timestamp
4) Activity mode switch
   A Markov smoother is applied to the Google AR service output. Experiments showed most misclassifications (or noises) have confidence values < 60. This value is used as a transition threshold. If the confidence value is < threshold, the Markov smoother backtracks to previous AR service outputs and finds all tuple-list values of x_{t-1} and the current tuple for each activity. We compute the $c_{ConfValueSum}=\sum(cv_{t,1}, cv_{t-1,k})$ for each activity. If the result is above threshold a transition occurs, otherwise the activity mode stays the same.

Then ARsignal will maintain 15s of cellular signal data in a moving sampling window and generates result z_t based on this data. If z_t is not *stationary*, the current activity will by y_t – otherwise another threshold value is applied. If the confidence value is below, the current activity y_t = stationary, otherwise the current activity will by y_t.

Implementation

a. Include ARshell.jar to the libs of your projects
b. Add a permission to the AndroidManifest.xml:

c. Add 5 lines of code to get ARshell working:

- Instantiation: ARshell as = new ARshell()
- Initialise ARshell with an interval of recognition: as.init(this, 10)
- Start recognition: as.start(this)
- Get the recognition result wherever you want: Result.getNameFromType(Result.getActivity())

  Stop recognition: as.stop(this)

5.1.2 State of the Art

Human Activity Recognition HAR can be described as classification problem. Therefore, especially machine learning techniques have been used in recent research to tackle the problem of Activity Recognition. Feature extraction and used learning algorithms are core aspects of any machine learning system.

Latest research shows (Suto 2018, p.13) that well-constructed Artificial Neural Networks ANNs are more accurate and therefore a better choice than CNNs. Furthermore, CNNs are not to be recommended for real-time HAR due to their extensive training time. Even faster than classical ANN is Extreme Learning Machine ELM, which is a model of an artificial neural network ANN with input hidden weights and analytically computed random output weights. It is much faster than classical ANN models (according to Kuncan2019, p. 35).

HAR systems comprise of two major steps.
In a first step, features are derived from sensor signals. The accuracy of Activity Recognition directly correlates with feature extraction. Therefore, feature extraction is the core aspect as activities are distinguished and defined by them.
The second step consists in the classification process using a learning algorithm which uses said features from the first step.

A Novel Approach for Activity Recognition with Down-Sampling 1D Local Binary Pattern Features

Kuncan (Kuncan 2019, p.39) propose an Activity Recognition Method consisting of 6 steps

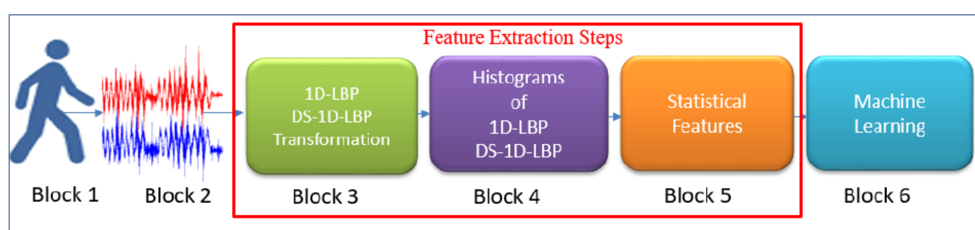

Figure 4 reproduced from Kuncan 2019, p. 39

Block 1 - 2: dataset comprising of data generated by accelerometer and gyroscope sensors

Block 3: application of 1D-LBP (Local Binary Pattern) and DS-1D-LBP to signals

Block 4: histogram generation of newly formed 1D-LBP and DS-1D-LBP signals

Block 5: obtain statistical features from histograms

Block 6: classification process with ELM using statistical features according to a 10-fold cross-validation test

- **Feature Extraction – DS-1D-LBP**

The proposed system uses the Down Sampling-1D-LBP (Locally Binary Pattern) method for feature extraction. The DS-1D-LBP is the application of the 1D-LBP method to different levels of sensor signals. Advantages of this method consist in (Kuncan2019, p.35):

1) Individual values are used in all marks for feature extraction
2) Model implementation is fast and easy
3) Different feature groups can be extracted depending on the window length WL and related sampling parameters

Data was obtained from the Kaggle (Kaggle 2019) repository, which offers an Activity Sense Dataset. This dataset contains time series data generated by accelerometer and gyroscope sensors of an iPhone 6s placed in the right front pocket of different subjects. Data was obtained using the iPhone 6s with Sensing Kit, which collected information from the Core Activity frame on iOS devices placed in the right front pocket of 24 people for 6 specified activities. (Fig. 5)

TABLE III. 12 DIFFERENT TIME SERIES DATA

Signal Code	Signal Name
S1	attitude.roll
S2	attitude.pitch
S3	attitude.yaw
S4	gravity.x
S5	gravity.y
S6	gravity.z
S7	rotationRate.x
S8	rotationRate.y
S9	rotationRate.z
S10	userAcceleration.x
S11	userAcceleration.y
S12	userAcceleration.z

Figure 5 taken from Kuncan 2019, p. 36
Different Time Series Data

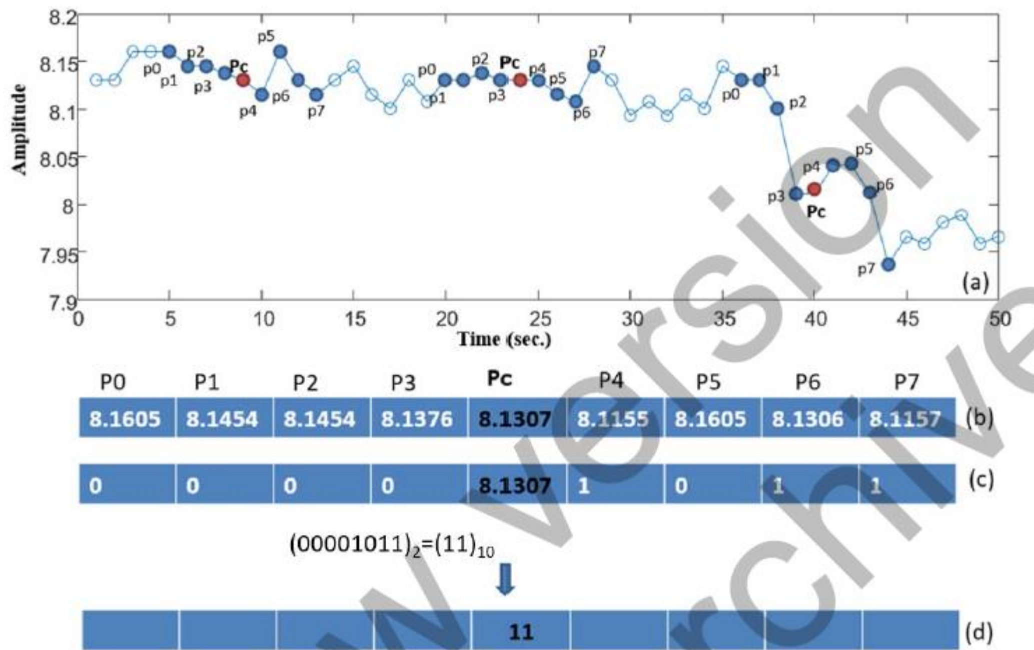

Figure 6 taken from Kuncan 2019, p.37.
Visualizing the process from raw data (a) to signal value (b)
to binary string (c) and decimal value of 8-digit binary number (d)

- **1D-LBP method**

Fig. 6 visualizes this process:

(a) shows the raw sensor signal

(b) the signal value (value range 0 to 255)

(c) the creation of a binary string by comparing the central value with neighbouring values (0 if central value is lower, 1 if higher)

(d) is the decimal representation of the resulting 8-digit binary number = 1D-LBP value.

- **Down Sampling 1D-LBP**

According to a defined window length WL, new signals are created by taking sample values from signals. If WL = 4, the window has four signal values. The average, median, the minimum or maximum value of these four signal values are taken for the signal to be newly generated. Fig. 7 shows this process: (a) the original data, (b) Level 1 DS, (c) Level 2 DS applies the DS-Means method to the signals in (b)

The DS method has three important parameters:

1) Number of levels to be applied by the DS method
2) Window length
3) Which sample values that enter a window / which results are to be moved to a higher level (minimum, maximum, medians or averages of values)

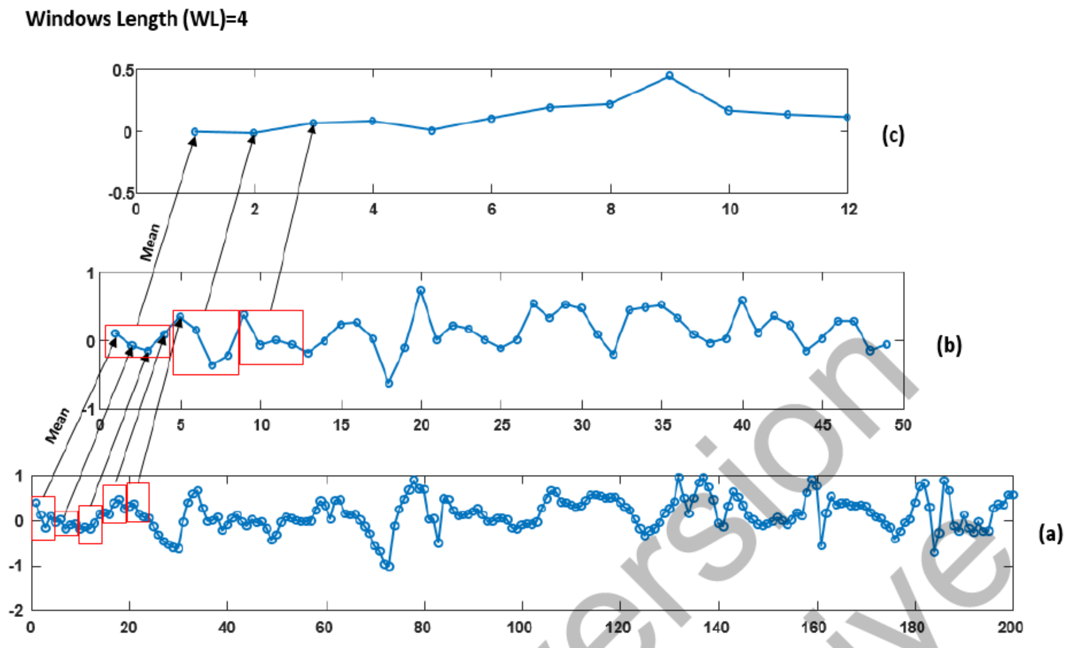

Figure 7 taken from Kuncan 2019, p. 38.
Showing the down-sampling of data within a predefined window length of 4

- **Statistical Features**

In this study, 12 statistical features were obtained from signal histograms. Since the dataset offered 12 values and foreach value 12 statistical features were calculated, 144 features were obtained for each level.

Classification - ELM

In the classification phase the extreme learning machine ELM is used, which randomly generates input weights and threshold values. Output weights are obtained analytically. The ELM algorithm comprises of three steps:

1) Randomly generate: $W_i = (W_{i1}, W_{i2}, \ldots, W_{in})$ –input weights and hidden layer b_i threshold values
2) Hidden layer H output calculation (O_k = output value)
3) ß output weights are calculated according to ($ß = H_+Y$; Y is to decide the property).

Ultimately, the training process wants to find the smallest squares solution in the Hß = Y linear equation in ELM.

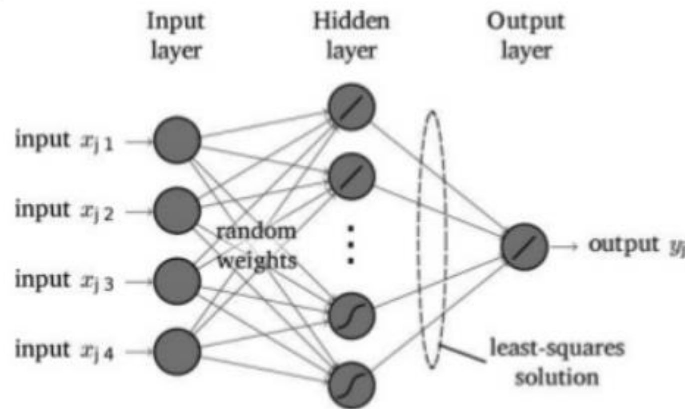

Figure 8 from https://www.slideshare.net/CSAalto/applications-of-machine-learning-59300486, accessed October 10 2019
Extreme Learning Machine ELM model

The classification was performed according to a 10fold cross-validation. Kuncan (2019, p. 40) note best results with a WL parameter of 4. It shall be noted that the sampling parameter (which sample values are used) correlate with the success rate. Hence it is optimal to test different settings for distinct applications.

The success of the ELM model correlates with the activation function used in the neurons and the number of neurons in the hidden layer. In this study, best results were obtained using the sigmoid activation function. In addition, a higher number of neurons (100) resulted in better success rates.

In addition, the study also tested different machine learning methods for classification (Fig. 9), with ELM being the best method with a success rate of 96.87%.

Model	WL=4	WL=5	WL=6	WL=7	WL=8
ELM	**96.8950**	94.5108	95.0123	93.2881	93.0385
RF	95.0000	93.6111	95.5556	93.3889	93.0111
MLP	77.7778	70.2778	74.7222	76.3889	74.3889
Knn	93.8889	92.5000	94.7222	93.8889	92.2222
SVM	78.6111	71.6667	76.1110	70.8333	72.6330

Figure 9 taken from Kuncan 2019, p. 42.
Success Rates Observed with Different Machine Learning Methods

Note

Another interesting approach (orientation independent, real-time) with a success rate of 95% was proposed by Siirtola in 2012 (Also cited by Kuncan2019). As one of the few studies also aspects like CPU usage are considered, which are under 5% for the proposed method.

6 Conclusions

In regard of already available Activity Recognition APIs, it is recommended to implement an API for the targeted (cross-)platform, all of which provide multi-modal Activity Recognition functionality. If gathered data does not suffice for requested information output, additional API implementations might be necessary, e.g. obtain health relevant information. In addition, depending on necessary input information (e.g. gender, age) for additional functionality privacy settings might become an issue.

6.1 Activity Tracking Implementation and Learnings

The SimpliCITY cross-platform mobile app pilot is developed using Visual Studio and Xamarin. Proprietary Activity Recognition APIs are used for Activity Recognition on each platform. Activities consist of a type (Still, Walking, Running, Cycling, Automotive, Unknown) and have a confidence value on correct type recognition. To improve the accuracy of Activity Recognition we applied some adaptions. During optimisations we noted, that in Android, the reduction of the activity update interval didn't have any impact on accuracy of Activity mode detection.

First, we only updated an Activity when the confidence level was above a low threshold. In Android, the confidence value has a range of 0-100, whereas iOS uses enums of Low, Medium and High. Therefore, iOS enums were mapped to values (Low – 33, Medium – 66, High – 99). As a result, Activity updates occurred if the confidence value was above 33.

Second, the Activity type "Unknown" was ignored. Most of the time this type was detected when Activity transitions occurred.

Third, on Android Activity updates were used instead of Activity transition updates because they resulted in a higher accuracy.

Latency

We found that latencies for Activity Recognition differed quite strongly in relation to Activity type. Types like "Walking", "Still" and "Automotive" were detected fairly fast in an average time of about 15 seconds. This time could be reduced when the smartphone was not handheld but in a fixed position.

Latencies were highest (on average about a minute) when cycling was involved. This observation also applies to transitioning from automotive to another state.

In iOS latencies regarding the Activity type "Walking" could be as fast as a second, but also meant that handheld devices would trigger "Walking" on slight position changes, although the person was standing.

Accuracy

The accuracy of Activity type recognition differs in type and speed. Activities like “Still”, “Walking” and “Automotive” were very accurate unless the car would be driving very slowly and had problems to differentiate between “Automotive”, “Cycling” and “Unknown”.

We had most problems with the “Cycling” mode detection. In iOS cycling was only detected when location updates occurred at the same time. On both platforms, (very) slow cycling speeds did not trigger “Cycling” detection at all. Otherwise “Cycling” recognition had an average latency of about a minute.

Since bike tracking always amounts to saving of locations and Activities the correctness of Activity Recognition can be deduced and verified by getting an average speed for a route after bike tracking has been stopped. This enables us to add an Activity check, providing a speed range for cycling. It also enables us to gather data on detected Activity types during bike tracking and evaluate the accurateness.

Only if the accuracy of Activity Recognition does not suffice and workarounds like comparing a route’s average speed with detected Activity types do not output expected results, custom AR systems should come into focus. Custom AR systems ultimately mean a good deal of time and effort for developing them. With this in mind, we would like to point out some problems and challenges in current research and available datasets for training models, which would have to be considered and are relevant for custom solutions.

Although Activity Recognition using sensor-based systems (e.g. smartphones) are a vast field of research, there is a lack of standardized datasets, recognition tasks and evaluation criteria as Wang stress (Wang 2019, p. 10870). This problem is of increased importance, since simple success rates may not be as meaningful as one would assume considering important criteria for real-world applications.

Consumption analysis

Most studies lack a consumption analysis (also in regard of used sensors, number of features) as Bhooshan (Bhooshan 2017, p.487) states. This analysis would be of great interest to decide if a found method was worth implementing. Although research exists on energy efficiency on HAR systems, offered improvements might only be useful for certain feature extraction methods.

Orientation-(In)Dependence

Furthermore, some methods use a smartphone fixed in orientation and position when gathering data, whereas an orientation-independent system would be far more interesting for real-world applications. Ustev (Ustev 2013) propose a solution for this problem.

Activity/Transport Modes

Different number and types of activities between different research also complicates comparisons. Some solutions might not include relevant Activity types for one’s project making it impossible to assess how accurate solutions might be.

DataSet
Basically, many of these problems are directly related to the datasets. Datasets comprise of different amounts of data from x smartphone sensors collected from x people carrying the smartphone in one or more positions. Apart from the number of subjects also their age is of importance, since a test group consisting of elderly will gather quite different data signals for multiple activities than a more heterogenous group.

6.2 Activity Recognition and Data Insights

Activity and location information allow drawing conclusions on different aspects. Some of which might need additional data to be collected but are still mentioned in the following paragraphs.

Travel Patterns – Urban Areas
In general, data on preferred transport modes but also, if according data is provided, for distinct population groups categorized by gender, age, marital status etc. can be collected. Thus, valuable insight of public transport acceptance can be gathered for different groups. If special offers for target groups are announced, feedback and impact can be gathered. In addition, the focus can easily shift to specific regions. Problem and model regions can be compared, and differences recognized and assessed. As a result, necessary improvements can be more easily identified and targeted.

Preferred transport modes are categorizable for distinct time periods (weekdays, weekends, or other time spans) and regarding weather conditions. This might have impact on future ticket offers or even the planning of transport systems. Not only a high demand as such but desirable combinations with e.g. possibility for bike storage / transport / rent can be recognized and corresponding offers developed.

It is possible to deduce street condition (applicable for cycling, vehicle mode) (see Sattar 2018, Zang 2018) which stresses the need for road maintenance and might provide insight why going by bike isn't a likely choice in certain areas.

People's travel patterns allow deductions for home and workplace and show daily travel routines. Relevant information e.g. route information, suggestions on route optimization can be sent when a person is more likely to read a message (e.g. not while riding a bike). If construction work will affect a person's daily route, information on duration and redirection can be provided, thus offering an improved service. This also applies for delays of relevant trains etc. Since travel patterns show the average travel radius of a person, different travel types can be assessed.

Other relevant information includes city activities like bike service checks, new/changed tickets, special offers, and points of interest. The possibility to give feedback on and provide suggestions for distinct transport systems is of importance to identify chances or problems which have not been on the radar yet. At the same time, people can raise their concerns.

Health-awareness is on the rise. Therefore, another relevant topic are health and sport related information. Apart from information like steps per day, minutes riding a bike, calorie consumption, small reminders which motivate people to do some small workouts after long periods of sitting (at work) help to keep fit. If the workday is too long small feedback for work/life-balance can be provided. Furthermore, push notifications regarding weather conditions can notify a person to pick up the umbrella before leaving the house.

Activity and location information give powerful insights into people's travel behaviour and allow to draw conclusions for a multitude of aspects. Above mentioned possible data insights are only some exemplary examples, which can be more refined as the focus shifts and experience is gained.

7 Lessons learned

In the SimpliCITY cross-platform mobile app activity recognition is implemented by using proprietary APIs for each platform. Both offer the detection of most important transportation modes, for which machine learning is used. It can only be assumed that Google and Apple provide updates at some intervals.

It was found that in Android, constant activity updates provided more reliable feedbacks than using updates only on transportation mode transitions (enter and exit). Initially we assumed, that activity mode transitions would be preferable to constant updates because only a change in transportation mode is of interest. But it turned out, that these transitional updates were less reliable. Therefore, we opted for constant activity updates.

Changing the time-interval (in ms) for activity updates did not have any real impact on results, unless the interval was expanded to several seconds, thus increasing the latency but becoming more battery friendly.

In iOS, the biggest surprise was that cycling mode detection was only successful as long as location updates occurred at the same time.

Detection of different activity modes was slightly faster when the device was in a fixed position and not handheld or loosely in a bag. But it cannot be assumed that many users will have bike phone holders.

Although activity mode recognition can be very accurate, especially cycling proved to be problematic. This was all the more true, if speeds were very slow. In that case, there often was no cycling mode detection at all. In addition, latencies for cycling mode detection were quite high – around a minute on average.

As mentioned before, we improved activity mode detection by allowing updates only above a certain activity's confidence value and if the activity type was not of type "UNKNOWN".

In the SimpliCITY app the user must start bike tracking explicitly. First, because bike tracking is resource demanding due to the fact of location updates. Activity updates are less battery demanding. Second, the user is in more control. He or she can opt when to allow the app to access the device's location.

When bike tracking is active, every saved location has a reference to the current activity mode (transportation mode). On finishing bike tracking, the distance and average speed for the just finished route are saved, thus enabling us to assess the data at any time. We can analyse which activity modes have been saved for a route X, e.g. 20% walking, 80% cycling. In a next step we are able to validate if the average speed is consistent with or contradicts the activity type. With this safe check, the user cannot cheat and collect points for a route which was driven in a car. Moreover, we can evaluate how accurate proprietary activity mode recognition APIs are and have information on (a) user's average speed when cycling, walking ...

Starting the bike tracking service explicitly also reduces the latency problem. Of course, the transportation mode is detected with a certain delay, but the starting and finishing locations are accurate (on button press). This also means that the tracked route's distance is accurate in the current setup. If bike tracking would solely rely on cycling mode detection to start and stop the tracking, the system would track a far shorter distance if a route had many stops (e.g. traffic light) due to the long latency and the uncertainty of accurate detection.

The current setup avoids aforementioned problems, improves the accuracy of bike tracking itself and offers ways to cross check results from e.g. a route with some dominant transportation mode.

8 References

Apple WWDC 2014. "Asciiwwdc - Motion Tracking With The Core Motion Framework". *Asciiwwdc*. Accessed October 14 2019. https://asciiwwdc.com/2014/sessions/612.

Bhooshan, Jyotshana. 2017. "Review on Recognition of Human Activity through Smartphone Built-In Sensors". International Journal of Advanced Research in Computer Engineering & Technology (IJARCET), 6(4), 482-489. ISSN:2278–1323.

Cross, Orrin. 2018. "Beta Testing Google'S New Activity Transition API". *Medium*. Accessed October 14 2019. https://medium.com/life360-engineering/beta-testing-googles-new-activity-transition-api-c9c418d4b553.

Jackpotek. 2013. "Experimenting With Google Play Services – Activity Recognition". *FP7 Carmesh: The Blog*. Accessed October 14 2019. https://carmesh.wordpress.com/2013/05/31/experimenting-with-google-play-services-activity-recognition/.

Kaggle Inc. 2019. "Kaggle: Your Home For Data Science". *Kaggle.Com*. Accessed October 14 2019. https://www.kaggle.com/.

Kuncan, Fatma, Yilmaz Kaya and Melih Kuncan. 2019. "A Novel Approach for Activity Recognition with Down-Sampling 1D Local Binary Pattern." *Advances in Electrical and Computer Engineering*, 19(1), 35-45. https://doi.org/10.4316/AECE.2019.01005.

Prajakt. 2018. "Hypertrack Supports The New Transition API For Android Activity Recognition". *Medium*. Accessed October 14 2019. https://medium.com/hypertrack/hypertrack-supports-the-new-transition-api-for-android-activity-recognition-5a5f22930285.

Sattar, Shahram, Songnian Li, and Michael Chapman. 2018. "Road surface monitoring using smartphone sensors: A review." *Sensors,* 18(11), 3845. https://doi.org/10.3390/s18113845.

Siirtola, Pekka and Juha Röning. 2012. "Recognizing Human Activities User-independently on Smartphones Based on Accelerometer Data". International Journal of Interactive Multimedia and Artificial Intelligence, 1(5), 38-45. https://doi.org/10.9781/ijimai.2012.155.

Stogaitis, Marc, Tajinder Gadh and Michael Cai. 2018. "Activity Recognition'S New Transition API Makes Context-Aware Features Accessible To All Developers". *Android Developers Blog*. Accessed October 14 2019. https://android-developers.googleblog.com/2018/03/activity-recognitions-new-transition.html.

Suto, Joszsef, Stefan Oniga, Claudia Lung and Ioan Orha. 2018. Comparison of offline and real-time human activity recognition results using machine learning techniques. *Neural Computing and Applications*, 1-14. https://doi.org/10.1007/s00521-018-3437-x.

Ustev, Yunus Emre, Ozlem Durmaz Incel and Cem Ersoy. 2013. "User, device and orientation independent human activity recognition on mobile phones: Challenges and a proposal." *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous*

computing adjunct publication, ACM, 1427-1436. https://doi.org/10.1145/2494091.2496039.

Veugen, Paul. 2015. "Taking Activity Tracking To The Next Level With The Iphone 6 And 6S". *Medium.* Accessed October 14 2019. https://medium.com/@pveugen/taking-activity-tracking-to-the-next-level-with-the-iphone-6-and-6s-58c0b0d12847.

Wang, L. et al. 2019. Enabling reproducible research in sensor-based transportation mode recognition with the Sussex-Huawei dataset. *IEEE Access*, *7*, 10870-10891. https://doi.org/10.1109/ACCESS.2019.2890793.

Zang, Kaiyue, Jie Shen, Haosheng Huang, Mi Wan, and Jiafeng Shi. 2018. "Assessing and mapping of road surface roughness based on GPS and accelerometer sensors on bicycle-mounted smartphones." *Sensors* 18(3), 914. https://doi.org/10.3390/s18030914.

Zhong, M., Jiahui Wen, Peizhao Hu and Jadwiga Indulska. 2015. “Advancing Android Activity Recognition Service with Markov Smoother“. Accessed October 14 2019. http://scholarworks.rit.edu/other/841.

Zhong, M., J. Wen, P. Hu and J. Indulska, 2017. Advancing Android activity recognition service with Markov smoother: Practical solutions. *Pervasive and Mobile Computing*, *38*, 60-76. https://doi.org/10.1016/j.pmcj.2016.09.003