L’extrasensory concatena tutte le feature dei vari sensori e poi crea un modello per ogni label (figura C). [versione 1 2017]
ogni modello di logistic regression restituisce una probabilità per quel label
l’evoluzione di questo è il MLP, modello unico che restituisce un array di risultati, una prob per ogni label

capire come combinare i risultati dei vari modelli/api

-
- l’output finale come deve essere
- i modelli devono usare tutti le stesse label? gli stessi dell’extrasensory? però lo yamnet ha delle classi diverse rispetto all’extrasensory

abbiamo più modelli con output differenti (differenti label) ma parzialmente sovrapposti
vogliamo combinarli, ma non possiamo fare un ensemble classico perchè non condividono le label (ne il dataset)
possiamo creare un modello a partire dai risultati dei singoli modelli? non sapremmo su cosa trainarlo, non avendo un dataset

ha senso creare un albero decisionale, il cui input è l'output degli altri modelli, che si basa solo su euristiche? (esempio se l’api di google mi dice che l’utente sta camminando escludo a priori i label non coerenti con questo tipo di attività)

oppure questa operazione è fattibile con altre tecniche di machine learning?

ipotesi arricchire il dataset dell'extrasensory

il dataset extrasensory è formato dai dati raccolti dai sensori e dai label assegnati dagli utenti, l'insieme di questi è usato per il training del il modello
possiamo dividere i label assegnati dagli utenti in due tipi, attività primaria e attività secondaria.

l'attività primaria è del tipo "camminando" "seduto" "in bicicletta" "dormendo", l'attività secondaria è un dettaglio, quindi ad esempio "mangiando" che può essere eseguita se l'attività primaria è "camminando" o "seduto", ma realisticamente non se è "dormendo"

nel nostro sistema abbiamo come fonti aggiuntive di dati

- le api di sistema di activity recognition, che hanno come output proprio l'attività principale che l'extrasensory ha come label
- le api di google per il sonno, che hanno come output un livello di confidenza per cui l'utente sta dormendo

l'output di queste due fonti può essere usato per arricchire il dataset?
noi abbiamo gli stessi dati dei sensori dell'extrasensory ma in più abbiamo le api che ci danno l'attività principale e lo stato del sonno, quindi possiamo arricchire il dataset inserendo come feature l'attività principale che gli utenti hanno segnato poichè possiamo usare questo dato nella prediction
ha senso?

extrasensory
dati raccolti -> sensori
label -> attività principale e secondaria (ground truth definita dall'utente)

TRAINING						
sensore 1	sensore 2	sensore 3	walking	cycling	eating	talking
123	123	123	0	1	0	1
123	123	123	1	0	0	1
123	123	123	0	1	0	1
123	123	123	1	0	1	0
PREDICTION						
sensore 1	sensore 2	sensore 3	walking	cycling	eating	talking
123	123	123				
123	123	123				
123	123	123				
123	123	123				

proposta
dati raccolti -> sensori, api sonno, attività principale da api OS
label -> dall'extrasensory teniamo solo le label delle attività secondarie

per la prediction abbiamo più dati da utilizzare grazie alle API, e dobbiamo predire solo l'attività secondaria

TRAINING						
sensore 1	sensore 2	sensore 3	walking	cycling	eating	talking
123	123	123	0	1	0	1
123	123	123	1	0	0	1
123	123	123	0	1	0	1
123	123	123	1	0	1	0
PREDICTION						
sensore 1	sensore 2	sensore 3	walking	cycling	eating	talking
123	123	123	0	1		
123	123	123	1	0		
123	123	123	0	1		
123	123	123	1	0		

Dataset preparation

Flutter activity recognition

The package recognizes the following activity types (with the related confidence):

`IN_VEHICLE` The device is in a vehicle, such as a car.

`ON_BICYCLE` The device is on a bicycle.

`RUNNING`	The device is on a user who is running. This is a sub-activity of ON_FOOT.
`STILL`	The device is still (not moving).
`WALKING`	The device is on a user who is walking. This is a sub-activity of ON_FOOT.
`UNKNOWN`	Unable to detect the current activity.

In the Extrasensory Dataset we plan to replace the following column features (discrete) as follows:

label:FIX_walking -> activityType:walking
label:FIX_running -> activityType:running
label:BICYCLING -> activityType:on_bicycle

It's worth to notice that the activity type IN_VEHICLE could be associated to several Extrasensory dataset labels:

- label:IN_A_CAR
- label:ON_A_BUS
- label:DRIVE_-_I_M_THE_DRIVER
- label:DRIVE_-_I_M_A_PASSENGER

So, for the new feature column activity:in_vehicle, we're going to set 1 in the dataset for rows having a 1 value in at least one of the four columns above. Since the user is necessarily in a vehicle for any of the 4 labels above.

Android Sleep API

The official Android Sleep API gives us a "yes/no" status with a confidence value. So, we're going to replace only the feature column "label:SLEEPING" with the new activityType:sleeping one.

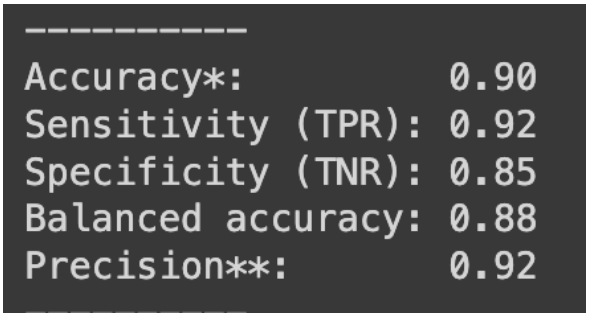
Question: how should we use the confidence from the activity recognition and sleep api? Could the model understand that it is a confidence? Is it a problem if we put only 0,1 in the dataset and then we pass a 0.8 for evaluation?

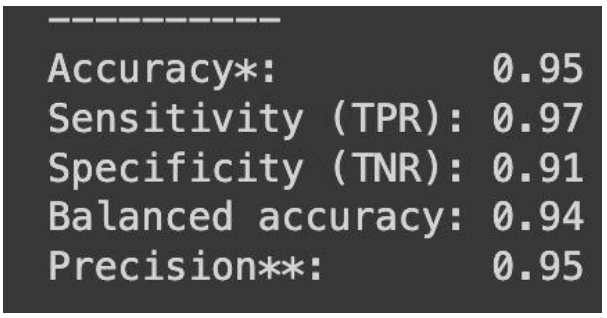
next steps
app with output of activity recognition and sleep api
script to modify the dataset according to the modifications described, check also the notebook to understand how to design the new dataset

26/11

We trained the same model of the original paper using the same dataset but edited according to the idea above
We tested the model against the label “sitting” which is highly correlated to the activity recognized by the apis.

Below the scores, first without including the new activityType features, then with them





So far the improvement looks promising

Next steps

1. run the dataset adjusting script on all the CSVs
2. train the models on all the data (one model for label for now)
3. test multiple label
4. update the app to collect the data from all the sensors required for extrasensory (currently only collecting activity type and sleep)
5. make a backend api to query the model
6. test and evaluate the model on the field

Step 4 - update the app to collect the data from all the sensors required for extrasensory (currently only collecting activity type and sleep)

What they did in the papers

From the Vaizman 2017 paper: “The system is based on measurements from five sensors in a smartphone: accelerometer (Acc), gyroscope (Gyro), location (Loc), audio (Aud), and phone state (PS), as well as accelerometer measurements from a smartwatch (WAcc). For a given minute, the system samples measurements from these six sensors and the task is to detect the combination of relevant context labels (Figure 1 (A)), i.e. declare for each label l a binary decision: yl = 1 (the label is relevant to this minute) or yl = 0 (not relevant).”

In the 2018 version, they increased the sensors coverage and lowered the sampling time as follows: “When ExtraSensory is running (in either the foreground or background of the smartphone), it records a 20-second window of sensor measurements every minute and sends the measurements to a dedicated server. The measurements include 40Hz 3-axial motion sensors (accelerometer, gyroscope, and **magnetometer**), location coordinates, audio (the app processes the raw audio on the phone to produce 13 Mel Frequency Cepstral Coefficients [17]), and phone-state indicators (app-state, WiFi availability, time-of-day, etc.). During the 20-second window, the app also collects measurements from the optional watch, if it is available and used by the participant (25Hz 3-axial accelerometer and compass heading updates).”

How we implemented in Flutter

We implemented measuring from the same five sensors they used in the paper:

- accelerometer (Acc)
- gyroscope (Gyro)
- magnetometer (Magnet)
- location (Loc)
- audio (Aud)
- phone state (PS)

We ignored the smartwatch at this moment.
The app collects data during a (configurable) 10-seconds time window, then it computes all the feature values needed from the model and it finally sends these features to the backend.cr
We decided to move the feature computation phase from the backend to the app in order to decrease the amount of data exchange between the app and the backend with respect to the app developed by the paper authors which sends all the raw values collected during the 20-seconds window to the backend every minute.

Data collection

For the accelerometer, gyroscope and magnetometer data collection we used this package, while for the location data retrieval we used this one. We ignored the following fields:

- ignoring fields
- watch sensors
- value entropy
- time entropy
- spectral entropy

Audio data (Android only)

For the audio recording and processing, we followed their Android-only implementation using MFCC and FFT (Fast Fourier Transform) algorithms.

App state

For the app state discrete stats, it's not clear from the app code how they handle these fields (probably they actually do not handle these fields at all), so we used the following approach:

- app is considered **active** when the internal app state is 'resumed' (i.e. this state indicates that the application is in the default running mode for a running application that has input focus and is visible)
- app is considered **in background** when the internal app state is either 'detached', 'hidden' or 'paused' (i.e. when the app is still hosted by a Flutter engine but is detached from any host views or is not currently visible to the user and not responding to user input or it is about to be paused)
- app is considered **inactive** when the internal app state is 'inactive' (i.e. at least one view of the application is visible, but none have input focus, so the application is otherwise running normally)

Battery info data

For the battery discrete stats, we used this package to retrieve battery information. We were not able to distinguish between ac, usb and wireless charging because we couldn't find a package that gave us this information. It's also not clear what the "battery_state:is_unplugged" field means and looking at the paper Android app code we couldn't find any implementation for this field, so we treated it in the same way as the "battery_state:is_discharging" field. For the "battery_state:is_not_charging" field we mapped it as the state when the cable is connected but the device is still not charging, considering that some devices (mostly tablets) consume more power than what can be provided via USB port. We also put the field "battery_state:missing" always to false since the value from the battery package is always present.

On the phone data

For the on the phone discrete stats, we used this package to retrieve phone incoming calls information. We put the field “on_the_phone:missing” always to false since the status value from the phone package is always present.

Ringer mode data (Android only)

For the ringer mode discrete stats, we used the Android-native AudioManager through a platform-native implementation in Flutter where:

- ringer_mode:is_normal is mapped to AudioManager state 2
- ringer_mode:is_silent_with_vibrate is mapped toAudioManager state 1
- ringer_mode:is_silent_no_vibrate is mapped toAudioManager state 0

Wifi status data

For the wifi status discrete data, we used this package to retrieve the device connectivity status. We put the field “wifi_status:missing” always to false since the value from the connectivity package is always present, also in case of no connection available.

Screen brightness data

For the screen brightness data, we used this package to retrieve the device current level of brightness.

Battery level data

For the battery level, we used this package to retrieve the device current level of battery.

Low frequency (one-time) environment sensors data (Android only)

For the low frequency environmental data (like pressure, relative humidity etc.), we used this package to retrieve all the environment sensor values needed except for “lf_measurements:proximity_cm” and “lf_measurements:proximity” for which we used this other package. These measurements are taken one-time only just before sending processed data to the backend. These packages support only Android devices.

Step 5 - data processing on the app side

The app collects data through async callbacks, so whenever each single sensor/source of data sends an event, the app stores it in a collection.
Every 1 second, it processes data from each source and then it computes the aggregates (according to what the paper does) and then it stores the object containing the aggregated data in another collection using insertion-order policy.
The only additional fields we’re using with respect to the paper are:

- activityType: the type of the recognized activity, it can be:

        - IN_VEHICLE
        - ON_BICYCLE
        - RUNNING
        - STILL
        - WALKING
        - UNKNOWN
- activityConfidence: the confidence of the recognized activity, it can be:
    - HIGH [80-100]
    - MEDIUM [50-80]
    - LOW [0, 50]

The paper implementation has some potentially missing values, like for example “discreteAppStateMissing” or “discreteBatteryPluggedMissing”, which we always set to false since we always have a value for this kind of information from the respective source of data.

Step 6 - update app and backend to communicate and request prediction based on a timeseries (?)

Every 10 seconds, the app sends aggregated data objects to the backend as a time series in order to get a prediction of the resulting status.