*Our goal is to build an interactive, wearable bracelet and GUI that learns a users:*

- *heart rate*
- *general muscle activity*
- *and temperature levels*

*in order to optimize her training routine and provide a vibrational feedback system for spontaneous health-related reminders.* In addition:

- We restrict our main objective to optimizing **distance ran**.
- We need to limit the training time and number of samples taken so that our approach is practical.

# 1 Data structure

The data points fed to our program will be contained as an array of 4-tuples

$$\{(t, h, p, \theta) = (\text{timestep, heart rate, pedometer, temperature})\},$$

where

$$\begin{cases} t \in [0, 8640], \\ h \in [50, 200], \\ p \in [0, \infty), \\ \theta \in [80, 110]. \end{cases}$$

Each running exercise timeseries generates a list $L_1, L_2, ...$, and our program aggregates these arrays into a mother array $\mathcal{L} = \{L_i\}_{i=0}^{N}$. The **distance ran function** $D : \mathbb{R}^4 \to \mathbb{R}_{\geq 0}, D : (t, h, p, \theta) \mapsto d$ is a map from "exercise metrics" to a number, corresponding to the total distance ran at that time.

The Spark IO has an ADC, and we assume that the data read by our program will be in the form of a `.txt` file. Each timeseries (exercise) will correspond to a different `.txt` file of the form

```
t1 h1 p1 t1
t2 h2 p2 t2
...
tn hn pn tn.
```

We store all non-redundant time series in a large `.txt` file, perhaps separated by timestep (but containing global parameters). By "non-redundant", we mean that we can filter out most datapoints that lie between two others.

# 2 Optimization and ML algorithms

Broadly speaking, our optimization goal is

$$\arg\max_{h,p,\theta} D(t = t_{end}, h, p, \theta)$$

with the constraints $h \in H, p \in P, \theta \in \Theta$, where $H, P, \Theta$ are subsets of the relevant intervals corresponding to achievable user values.

**Distance Optimization.** First, there are a few phenomenological aspects we hope to find:

1. if we plot the time series data, ideally we would find clusters of points depending on when users is "in the zone"; our objective is to get runner into the cluster that maximizes $D$.

2. if we don't find discernable clusters, the point cloud should be contained in some convex subset $S \subset \mathbb{R}^4$, and our objective is to get the runner to a point $(t, h, p, \theta) \in \partial S$ maximizing $D$.

The complexity of clustering is **NP**-hard, but should be tractable in our cases with generic run-of-the-mill algorithms (e.g. *Lloyd's algorithm*).

By assuming that the global value of $\arg\max_{h,p,\theta} D(t = t_{end}, h, p, \theta)$ is optimized by optimizing at every timestep $t_0 \in [0, 8640]$, we can also optimize locally; i.e. find

$$\arg\max_{h,p,\theta} D(t = t_0, h, p, \theta), t_0 \in [0, 8640].$$

In practice, however, this assumption may not be entirely valid: optimizing the distance ran for each timestep may not result in the true global optimum if the global optimum is obtained by parameter fluctuations. In these cases, we may wish to take into account the entire timeseries, barring memory and computing power concerns.

Our preliminary algorithm is thus as follows. It is a static algorithm that does not depend on the actual conditions of the current run; it just optimizes from previous datasets at each time point.

**Algorithm 1** MET Energy Optimization Routine (METEOR)

**Require:** objective time $t_{end}$
  load prior timeseries $\mathcal{L} = L_1, ..., L_n$ from `.txt` file
  **while** $t_{now} < t_{end}$ **do**
    run $k$-means clustering on $\mathcal{L}|_{t=t_{now}}$
    **if** $k$-means clustering $= 1$ and returns clusters $C_1, C_2, ..., C_m$ **then**
      $S \longleftarrow \arg\max_{C_i} \mathbb{E}(D(C_i))$
    **else**
      $S \longleftarrow \mathcal{L}|_{t=t_{now}}$
    **end if**
    $S \longleftarrow$ convexHull$(S)$
    $B \longleftarrow$ boundary$(S)$
    $x_{obj} \longleftarrow \arg\max_{(t_{now},h,p,\theta)\in B} D(t_{now}, h, p, \theta)$ (using, for example, a sampling procedure)
    **print** $x_{obj}$
    **update** $t_{now}$
  **end while**

**Remarks.** Note the following:

- If we wish to take some parameters of the current run into consideration, we can feed the algorithm this additional data in the **while** loop, and enforce the requirement that $S$ above is close to these additional parameters.

- If we wish to take into account some global properties of the timeseries, we can extend our definition of $S$ as relevant.

In practice, the algorithm above is useful for training, but may not be applicable without modifications to real-time optimization. The modifications above should another version of the algorithm suitable for real-time coaching.

**Machine learning.** The algorithm above learns the optimal parameters $x_{obj}$ during each timestep via a combination of $k$-means and convex optimization methods. To pursue ML further, we may also consider a model that takes into account an explicit function for $D$, instead of estimating $D$ using past data. For example, our assignment of $x_{obj}$ may be of the form

$$x_{obj} \longleftarrow \text{argmax}_{(t_{now},h,p,\theta)\in B}\ t_{now} + w_1 h + w_2 p + w_3 \theta,$$

where $w_i$ are learned weights. As a simple application of ML, we may also wish to stick a Bayes classifier in down the road.

# 3 Development

Before coding in Objective-C/C++ for the iPhone, we will use Python for a proof-of-concept of our algorithm and assume randomly generated data samples.

# 4 Prototyping

## 4.1 Convex optimization algorithm

## 4.2 Naive Bayes classifier

Try this for fun?