So you want to build a ML system

Jan Macháček

March 11, 2018

Abstract

Amongst other things, a machine learning system should be built! It can solve anything, there are so many wonderful frameworks, so many conference talks that show just how easy it is to throw together just a few line of Python, and hey presto!–a machine that can recognise digits in images, recognise hot dog and *not* hot dog... it even runs on a mobile phone. This essay is about the difficulties that are lurking in the execution and running of a robust & mature machine learning system.

1 What can ML do?

The exercise analysis system sets out to be a computerized personal for resistance or strength exercise. Exercise systems collect heart rate data as the baseline indication of physical exertion. Depending on the chosen sport, the systems typically collect GPS and accelerometer data. Geolocation, acceleration, and heart rate are fairly comprehensive source of data for outdoors sports: think running or cycling. For cycling, geolocation sampled at, say, 50 Hz reliably establishes the route ridden, but is also the source of data for speed and acceleration; with underlying elevation data & the weight of the rider and the bicycle, it is possible to estimate power output. With the heart rate, it is possible to get a fairly accurate view of the activity in Figure 1.

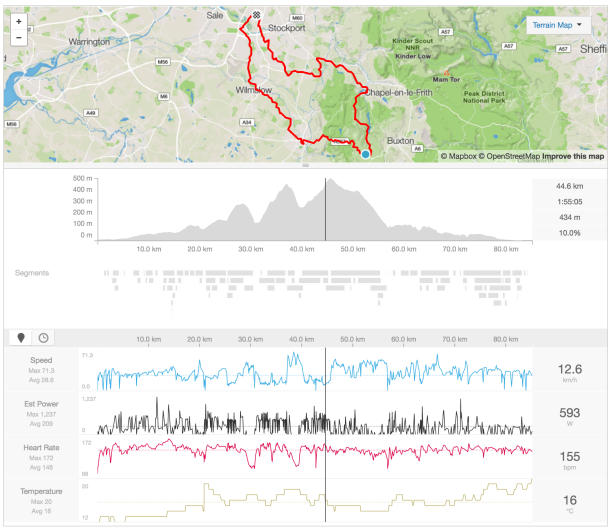

Figure 1: An afternoon ride

In addition to the post-ride analysis, the sensors provide immediate feedback during the ride. A modern cycling computer records and displays speed, elevation, distance, heart rate, pedalling cadence, and power. An athlete can use these values to inform his or her training; for example, one might train for a time trial aiming for a particular power output over a specific distance and use the values a cycling computer displays to guide the training sessions. (See Figure 2.)

Figure 2: Cycling power training

Everything becomes much more difficult indoors; and then even more difficult when not in a swimming pool, on a rowing machine, or on a spinning bike; when the user turns to resistance exercise. Some exercise machines now come with Bluetooth "beacons" that advertise the details of the activity. Any sensors are less likely to be embedded in free weights (and even if there were sensors, the sensors cannot reveal the exercise the user is performing; though identification such as 20 kg dumbbell is useful, and acceleration & rotation is even more useful). There are no machines involved at all in body-weight exercises.

To build a system for resistance exercise analysis that "matches" the features of a cycling system *without requiring super high-tech gym*, it will be necessary for the users to

bring their own sensors. A smartwatch can provide acceleration, rotation, and heart rate[1]; these three sensor readings can be fused with sensor readings from a smartphone that the user might wear in his or her pocket. This gives additional acceleration and rotation. Unfortunately, heart rate lags behind any exertion, so it'll only increase by the time the user is done with a set of a particular exercise (assuming the exercise is done with appropriate intensity); the last easily accessible sensor, geolocation (if at all available due to the location of the exercise floor) will show that the user is not moving much. This leaves only the acceleration and rotation values as useful indications of the movement being performed.

A proof-of-concept application that allows the users to wear a sensor on their wrists and use their mobile phones to *mark the start and end of each exercise*, together with *the name of the exercise* provides the data to build & verify the first models.

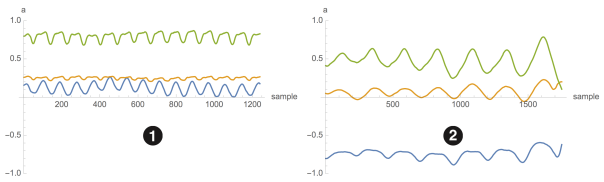

Figure 3: Two exercises

Figure 3 shows the acceleration from a watch worn on the left wrist. The acceleration is along the x, y, and z axes (with the Earth contributing $9.8\,\mathrm{ms}^{-2}$) sampled at 50 Hz; exercise ① is the straight bar biceps curl, and ② is the chest dumbbell flyes. You–the readers, humans–can work out the number of repetitions and you can clearly spot that these are two different exercises, and if you saw enough of these figures, you would be able to learn to identify the two exercises. It should be possible to implement a function *classify (sensor data)* → *exercise* that does the same. Even if this function performs perfectly (i.e. it identifies the movement that corresponds to the exercise for every *sensor data* that matches the exercise, even if the exact *sensor data* was not in the training set), it will struggle to identify areas of no exercise; unfortunately no exercise does not mean no movement, it simply means that the user is walking from one machine to another, setting up for a different exercise, taking a drink, and so on. These no exercise areas are quite often also areas of repetitive motion (walking) or fairly large movements (setting up); see Figure 4 where the red shaded areas represent exercise and the rest is other recorded motion.

The figure shows some of the typical challenges of noisy, mislabelled, and unclensed data. For example in ①, the user has done two exercise sets, but forgot to label the no

[1]Though most smartwatches report "beats per minute", the watch measures changes of the blood flow velocity, which is directly correlated to heart rate. Nevertheless, the watch does not measure heart rate itself.

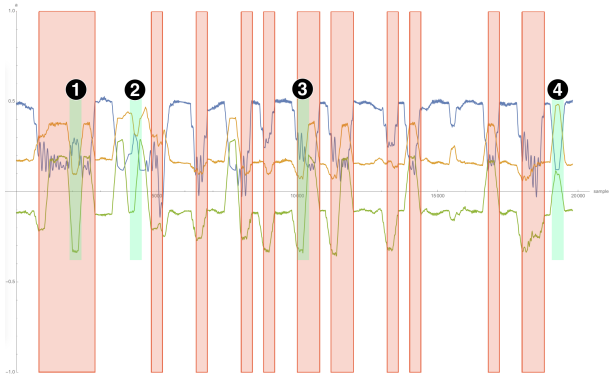

Figure 4: Exercise vs. slacking

exercise area properly. The area in ② shows large movement in the no exercise section–in this particular example, the acceleration actually shows the user dragging an exercise bench. In ③, the labelling is not entirely accurate: the user started the label before actually starting the exercise. Finally, ④ shows further wide movements in the no exercise section. To deal with the complexity of exercise and no exercise, the classification function needs to be *classify (sensor data)* → *result* , where *result* = *exercise* / *no−exercise* .

1.1 Sensors are not enough

Suppose there is indeed a function that takes the sensor readings and returns the details of the exercise being performed: *classify (sensor data)* → *result* . Even if this function performs perfectly, it only allows to build a system that lags behind the users' movement. Even if the system could read the data from the sensors with zero delay, the system would only be able to tell the user what he or she is doing after "seeing" some minimum amount of data, introducing lag. High lag will cause the user experience to be rather frustrating. In our testing, a 100 ms lag is noticeable; 250 ms lag is still tolerable; anything over 750 ms is downright confusing. A good example is the number of repetitions: with an exercise whose single repetition takes around 1 s, a 750 ms lag will mean that the system shows 5^{th} repetition when in fact the user is starting on his or her 7^{th}. Compare this to the sensor readings that the cycling computer reports: speed, altitude and similar can be reported as they are measured (typically a few times per second); cadence and power readings usually lag behind the instantaneous measurements because the computers display average values over a short time window. Just like the repetitions in resistance exercise, the lag is confusing; though if the athlete holds fairly steady power output, the reading usually matches the percieved effort.

The machine learning resistance exercise system is competing with a simple exercise log on paper notebook. Following users that use such an *ancient* technology reveals the flow that the application has to satisfy.

Figure 5: Exercise diary

The athletes use a diary similar to the one in Figure 5 to log the exercises done, to keep track of progress, and to stick to an established sequence of exercises.

2 The complexity of sensors

The system reads values from the sensors; the values are not read continuously, but read at discrete points in time. The range of sensor values can introduce clipping errors, and the sampling rate can introduce aliasing errors. Good choice of sampling rate and sensor types can reduce these errors: both sampling rates and sensor value range must be significantly greater than expected measurements; even in that case, the system must include enough signal processing to eliminate rogue values. For humans in resistance exercise, the sampling rate is sufficient to be 50 Hz, acceleration is expected to be in the range $\pm 40\,\mathrm{ms}^{-2}$, rotation $\pm 4\pi\,\mathrm{rads}^{-1}$. The first stab at implementing the sensor reading code may take the form of a loop in algorithm 1.

```
acc ← sensor(type = accelerometer)
gyro ← sensor(type = gyroscope)
while true do
    sleep(interval = 20 ms)
    a ← acc.currentValue()
    r ← gyro.currentValue()
    emit a, r
end
```

Algorithm 1. Simple sensor sampling

When implemented in real hardware, this algorithm suffers from *timing errors* and *high power consumption*. When it comes to reading the samples, is it acceptable to read acceleration before rotation?; the act of accessing those samples adds time, which means that the 20 ms in the *sleep* is too long, but 19 ms is too short. The infinite loop with the *sleep* syscall prevents the CPU from slowing down into more power-efficient modes. Modern hardware typically provides a way to begin the sensor readings, with the sensors storing the samples (pairs of *timestamp* → *value*) in a memory buffer without any interaction from the user program, and provide a way to access sections of the buffer at arbitrary intervals. (Viz algorithm 2.)

```
acc ← sensor(type = accelerometer)
gyro ← sensor(type = gyroscope)
timestamp ← DistantPast
timerFun ← Function is
    A ← acc.bufferedValuesSince(timestamp)
    R ← gyro.bufferedValuesSince(timestamp)
    S, timestamp ← fuse(A, R)
    emit S
end
timer ← timer(callback = timerFun)
Start: begin
    acc.start(rate = 50 Hz)
    gyro.start(rate = 50 Hz)
    timer.start(interval = 500 ms)
end
```

Algorithm 2. Asynchronous sensor sampling

This approach addresses the high power consumption, as well as some of the aliasing errors in algorithm 1. However, the sample buffers read from the sensors are not always precisely aligned with respect to wall time (i.e. time as measured by a [precise] clock on a wall) and might even contain gaps where the sensor wasn't able to obtain a value. Figure 6 shows sample buffers obtained from two sensors sampled at 50 Hz: the values should be exactly 20 ms apart, and the sample buffers from the two sensors should begin at the same wall time. Unfortunately, the available sensors (and their operating systems / runtimes) usually result in the sample buffers not beginning at the same wall time; similarly, the timestamps of the values in the buffers will not necessarily fall precisely on integer multiples of the sampling period.

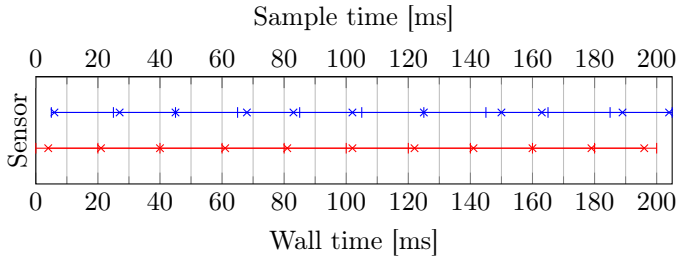

Figure 6: Timing errors

The system has to be able to cope with these misalignments and missing values: its goal is to *fuse* the sensors values so that all samples appear as though they were read from a *single reliable* sensor. It is also important to filter the

input data to reduce the impact of noisy or faulty sensors[2]; see Figure 7.

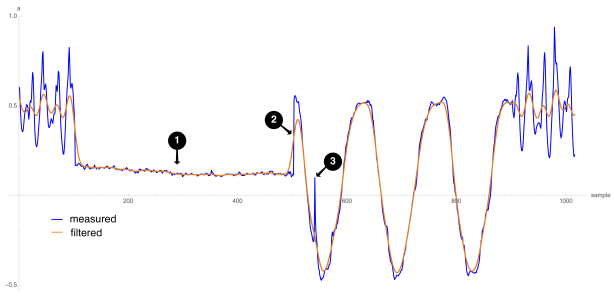

Figure 7: Sensor readings raw & filtered

The area ① in Figure 7 shows area of no movement, but the sensor nevertheless records significant noise; ② is an example of incorrect sensor reading–humans simply cannot accelerate at 5 $\mathrm{m\,s^{-2}}$ in the 20 ms period of one sample; ③ is another example of faulty sensor reading. A simply low-pass filter removes most of the noise, though it does not deal with possible missing values.

A good way to tackle the noise, alignment, and timing errors is to use the raw samples read from the sensors as an input to a model, and then to use the model as the source of the precise samples. Figure 8 shows the performance of a time-series prediction algorithm applied to gaps of 15 samples–the prediction is fairly accurate in the gaps ① and ②, but fails in ③.

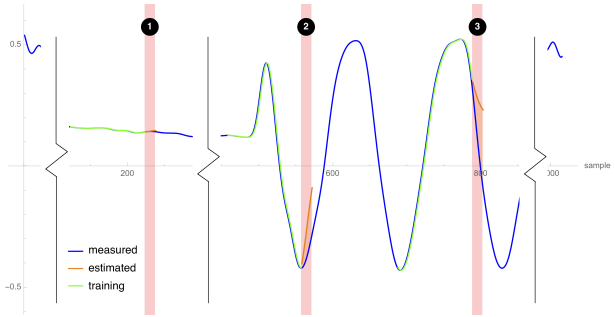

Figure 8: Extrapolating missing values

2.1 Low-power sensors

Low-powered sensors bring additional complexity of restricted memory and computational resources, as well as restricted transfer mechanisms. A good example of such low-powered sensor is the Pebble smartwatch. The most powerful model contains the Cortex-M7 CPU at 144 MHz with 128 KiB of memory for user programs; it uses BLE for data transfers. The protocol as well as the hardware architecture places restrictions on the sample width and sampling rate. BLE protocol specifies maximum 80 B per message, and minimum duration of 7.5 ms between messages; this gives us theoretical maximum of 10 640 $\mathrm{B\,s^{-1}}$. With one particular hardware / firmware implementation, we found that the effective reliable data transfer rate drops to only 280 $\mathrm{B\,s^{-1}}$. That particular device also lacked significant processing power. This gave us baseline protocol for samples with 3 axes (such as acceleration or rotation), with 20 B for header and 5 B per sample. The lack of processing power meant that we could not implement any reasonable compression algorithm. Instead, we took into account maximum reasonable acceleration a human can achieve in exercise, 40 $\mathrm{m\,s^{-2}}$, and represented sample as 3×13 bit for x, y, and z axes (with 1 bit for padding). Along with the samples, the wearable sends its timestamp, which is a monotonously-increasing device time in milliseconds. The mobile application remembers the first seen timestamp from each sensor for each exercise session, and uses this value to properly align the samples from the sensors. The sensor data includes quantisation error: at 50 Hz one sample represents 20 ms of real-time, but the difference of the timestamps may not be divisible by 20. It is sufficient to perform "round-even" correction of the received samples by evenly removing or duplicating the last sample in case of detected quantisation errors[3].

2.2 Security & Attacks

It is possible to attack this system by [2]...

3 Distribution of computation

xxx

3.1 Mobile

xxx

3.2 Server

xxx

4 Privacy

Privacy and security of the data is extremely important. The user details that the *profile* service holds are "bad enough", but when combined with the data held in the *cluster* and *ingest* services, it must be stored and processed with the highest standards–especially with the upcoming GDPR[1] directive coming into force. Naturally, the system applies encryption-at-rest to the data that the *profile* service holds; and the communication between the *profile* service and the mobile application uses transport-layer encryption. Additionally, the system does not leak the raw and

[2] Accelerometers can produce noisy readings when there is no movement

[3] Without the quantisation error correction, together with the unreliable sensors, the time error can add up to 10 s per hour of exercise

stable user identity outside its infrastructure; it isolates its services in a way that programming error (*A2:2017*: broken authentication, *A5:2017*: broken access control, *A6:2017*: security misconfiguration, *A8:2017*: insecure deserialization from the OWASP[4] list) does not lead to *A3:2017*: sensitive data exposure. The system uses JWT[3] as the authorization holder. The token holds the *cluster identifier* in the HMACSHA256-signed, base-64 encoded section, and the *user identifier* in an PKCS8-encrypted section. When the *authz* service issues the token it encrypts the profile-related sub-token using the *profile* service's public key, adds the "public" cluster identifier, signs the entire token and delivers it to the clients. Figure 9 illustrates the entire flow.

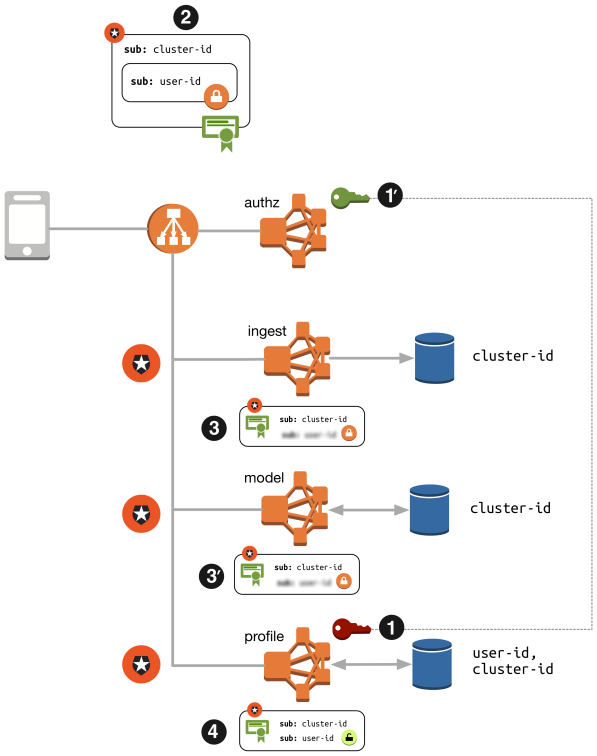

Figure 9: Authorization and data access

Initially, in ① and ①′ the *authz* and *profile* service have access to a key pair; the *authz* service can access the public key, the *profile* service can access the private key. (The implementation does not naturally have access to the actual keys, it relies on a mechanism that issues a public key when it generates and securely stores a private key; it then accepts requests to decrypt a payload using the private key it holds.) The *authz* service then issues a token ②; the token includes encoded & signed-only portion containing only information that cannot identify individual users, and an encrypted portion containing the user identity. The *ingest* ③ and *model* ③′ services do not have access to the private key used to encrypt the JWE token embedded in the JWT token. They can use the token to authenticate access, and they use and persist the cluster identifier in their data stores. The *profile* ④ service is able to authenticate the requests, but it can also decrypt the JWE token to obtain the user identity.

It is important to highlight that in good microservice-based architecture the services do not share data stores. Services that share a data store become tightly coupled: it is difficult for the services to have independent life-cycle, it is difficult to reason about consistency of the data across the services, and it becomes difficult to reason about authorization to access the data in the shared store. This system avoids the sharing data stores anti-pattern; it is very explicit about how the information flows through the system. Ultimately, each service is fully responsible for its data store; each service's data store is never shared amongst other services and there are no writes that subvert the microservice's code. A practical consequence for our system is that it is difficult to associate the session data (which contains the users' biometric data together with their location; very private information indeed) with a particular user's profile.footnote:[We do not add noise to the recorded data to further complicate the association; without the noise, it is easier–though still *extremely* difficult to identify a user based on just the way he or she moves. Nevertheless, it is still impossible to tie this identification with the information held in the profile database.] Let's zoom in on the points of sharing in our architecture in ARCH IMG.

References

[1] EU. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)*. `http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679`.

[2] Kevin Fu and Wenyuan Xu. *Risks of Trusting the Physics of Sensors*. `http://delivery.acm.org/10.1145/3180000/3176402/p20-fu.pdf`.

[3] *JSON Web Tokens*. `https://jwt.io`.

[4] *OWASP Top Ten 2017 Project*. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_2017_Project`.