So you want to build a ML system

Jan Macháček

May 26, 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.

It concludes with a ML readiness test, and recommendations on how to roll-out machine learning to an organisation (including tips on the process to follow, the people to hire). Mature ML organisation should enable a new person to make an impact in a ML system on the first day!

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. Geo-location, acceleration, and heart rate are fairly comprehensive source of data for outdoors sports: think running or cycling. For cycling, geo-location 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.

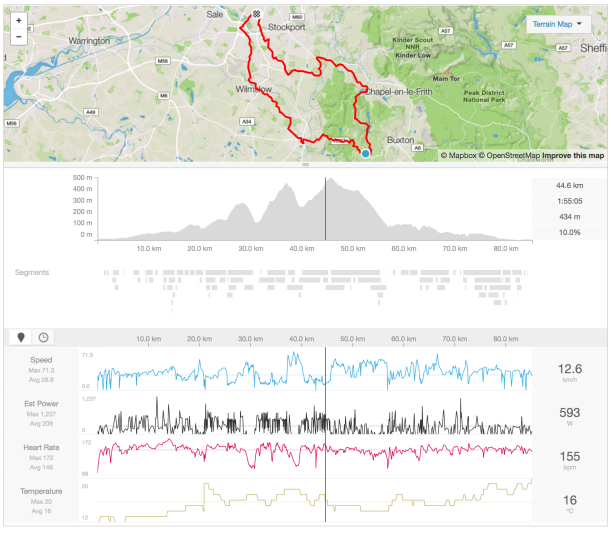

Figure 1: An afternoon ride

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 smart-watch can provide acceleration, rotation, and heart rate[1]; these three sensor readings can be fused with sensor readings from a smart-phone 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, geo-location (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.

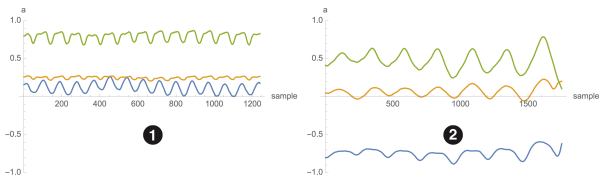

Figure 3: Two exercises

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.

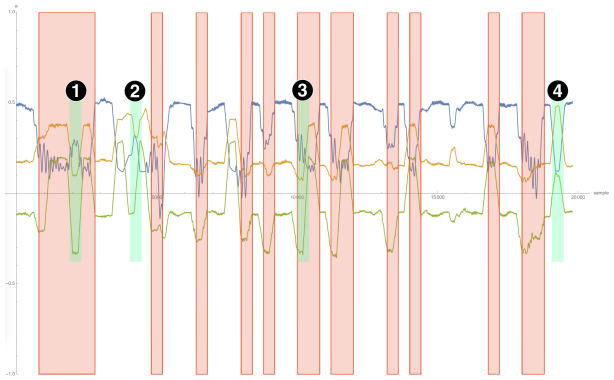

Figure 4: Exercise vs. slacking

The figure shows some of the typical challenges of noisy and mislabelled data. For example in ①, the user has done two exercise sets, but forgot to label the no 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[6], [7] 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 perceived effort.

[1] Though most smart-watches 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.

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.

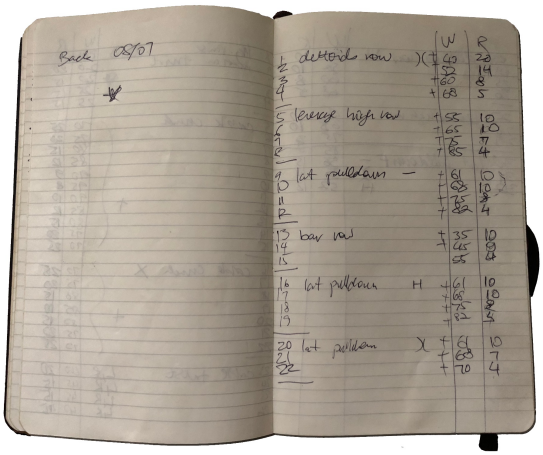

Figure 5: Exercise diary

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. The interaction the users need is for the application to show *what to do next*, with all relevant details of weights, repetitions, time, and intensity. This is what a user would be able to trivially look up in the paper diary. The paper diary also easily deals with the scenario when there the exercise space or machine is not available: it is trivial to scan the diary for the alternative exercise. So, the mobile application's interactions need to match that. (See Figure 6.)

Figure 6: Electronic version of exercise diary

The model used to predict sequence of exercise sessions and exercises within a session is a Markov chain. This allows the application to deal with the reality of exercise in a public gym, where the station for the next suggested exercise might not be available. The Markov chain, together with a bio-mechanical model of the main muscle groups, gives the users the flexibility to achieve their workout targets even in crowded gyms. The sequence of exercise predictions for one particular exercise sessions are illustrated on Figure 7.

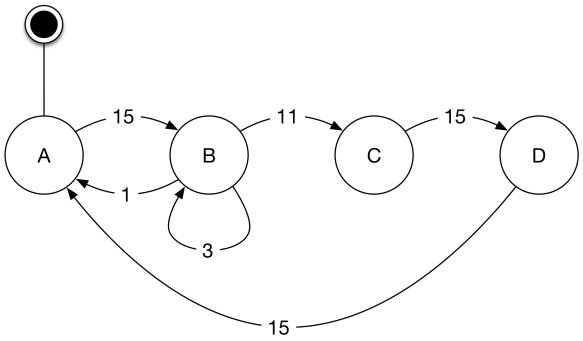

Figure 7: Exercise sequence model with no context

The numbers in Figure 7 represent the count of transitions taken; hence it is possible to calculate the probability of transition from any given state. The state names are the exercise labels, in real application, they are the real exercise names. The mobile application can either receive the chain when the user selects one of the pre-defined exercise programmes, or it can construct the chain from empty if the user chooses to start his or her custom workout. This gives the application an intuitive feel; its suggestions are what the users usually do. Finally, the information in the chain allows the system to identify the most popular sequences of exercises, to identify exercises that the users do not like; more interestingly, the system can use the information in the chain to identify sequence of exercises that leads to the best improvement. (There is no definition of *improvement*: in one case, it may be greatest perceived intensity; in other case, it may be greatest mobility range improvement; etc.)

To make the next-exercise prediction more accurate, the mobile application uses fined-grained location services. The location services are implemented using bluetooth beacons. The beacons operate in the iBeacon mode [2]; each beacon in this mode transmits its identifier, a major, and minor values. The mobile application sets up continuous scanning of a major value which identifies exercise equipment, receiving notifications of beacons and their minor values as they come into range. The mobile application then filters the exercise states keeping only those that are associated with a particular area.

With the fine-grained location data available, the application is *expecting* to see a movement that precedes exercises A or C. We call this movement the *set-up movement*. This now refines the classification into the *setup movement* classifier and the *in-exercise* classifier. Both classifiers follow the general *classify (sensor data)* → *result* signature. The set-up movement classifier takes 1 s of sensor input and produces probabilities classes that represent the set-up movement for groups of exercises. The set-up classes are not the

exercises themselves, but the movements that happen just before the start of an exercise (think picking up the dumbbells and moving them into place for shoulder press); one set-up movement can map to multiple exercises. To provide accurate prediction of the exercise about to be started, the mobile application takes into account the expected exercise (given the user's typical behaviour), and the fine-grained location data. Figure 8 illustrates the sequence of predictors and classifiers.

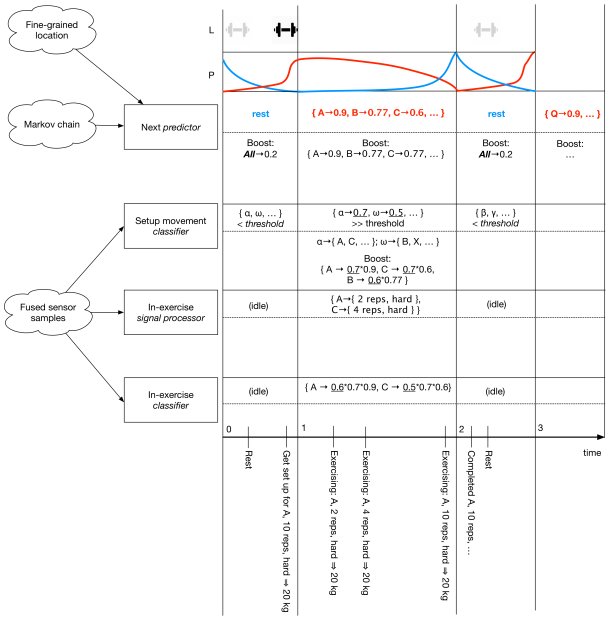

Figure 8: Predictors and classifiers

Note that the predictors' outputs change over time: at time 0 the probability of *rest* is 0.8, and the probability of any exercise is 0; as time increases towards 1, the probability of *rest* decreases, and the probability of exercises A, B, C increases, especially when taking into account that the fine-grained location sensor reports being in proximity of the weights rack. In the time interval $(1; 2)$, the probability of exercises A, B, C steadily decreases, and the probability of *rest* increases. If (particularly at the start) of the time interval $(1; 2)$, the *set-up classifier* classifies movement that is consistent with the set-up movement for muscle groups α, ω (where α contains exercises $A, C, ...$ and ω contains exercises $B, X, ...$), the system can deduce that the user is likely to be getting ready to perform exercise A or C. The details of the likely exercises are passed to the *in-exercise signal processor*, which is responsible for counting repetitions and evaluating the variation in the acceleration that is indicative of effort. The same information (set-up movement α, ω, expected exercises A, C, B) is fed to the *in-exercise* classifier, which seeks to confirm that the exercise being done is indeed A. In gym-speak, the muscle group α can be *shoulders* and ω can be *upper back*; the exercise A can be *dumbbell shoulder press*, and C *barbell shoulder press*. The final task for the application is to associate effort (measured by the *in-exercise signal processor*) with weight. It uses a map of *(exercise, sequence number, effort)* → *weight* that it maintains for each user. (It is important to keep track of the *sequence number*, especially for high-intensity work.)

The performance of the entire prediction and classification flow exercise prediction for accelerometer and gyroscope on the user's wrist and accelerometer and gyroscope in the user's mobile in the left pocket is shown in TODO: Chart here.

2 Bootstrapping

I knew what I was getting into when I was building the application and when I was using it in the gym. It was clear that at the start, there are no transitions from one exercise to another, no maps of *effort* → *weight*, that the classifiers and signal processors' parameters are not tuned or trained. Using the application's labelling interface and the old-school diary, it was possible to collect enough data for all components to perform as well as the tables above show.

The data used to train the *set-up movement* and *in-exercise* classifiers is re-usable[2], and a lot of training data can be generated by using the raw readings as inputs to a bio-mechanical model, which then generate the actual training and validation data for the classifiers. However, the exercise sequences in the Markov chains have proven to be very different between the users. The application required utmost patience during the first several exercise sessions. This was acceptable in the friendly test group, but would turn out to be a significant problem for a commercial application. An acceptable solution–from the user experience as well as accuracy point–is to run the application in *observer* mode for the first 10 sessions. In the observer mode, it does not suggest or classify anything; it submits the recorded data to the server-side for deeper classification. The server-side classifiers reconstruct the exercise sequences and then provide the parameters for the predictors. After the initial observer mode, the users[3] report that the application "does the right thing."

3 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

[2]The test users' athletic abilities were similar to the author's

[3]Embarrassingly, $N = 10$.

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)
    F, timestamp ← fuse(A, R)
    emit F
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 9 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.

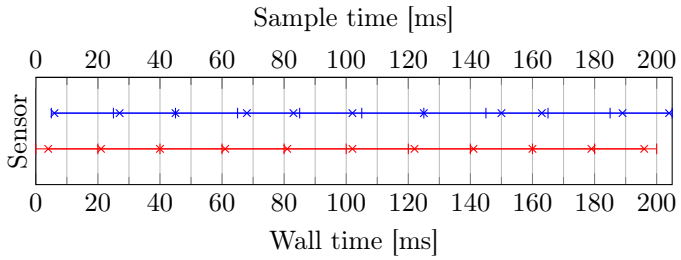

Figure 9: Timing errors

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[4]; see Figure 10.

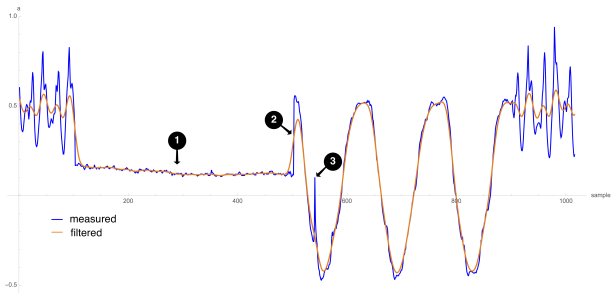

Figure 10: Sensor readings raw & filtered

The area ① in Figure 10 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 $5\,\mathrm{ms}^{-2}$ in the 20 ms period of one sample; ③ is another example of faulty sensor reading. A 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 11 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.

3.1 Low-power sensors

Low-powered sensors bring additional complexity of restricted memory and computational resources, as well as

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

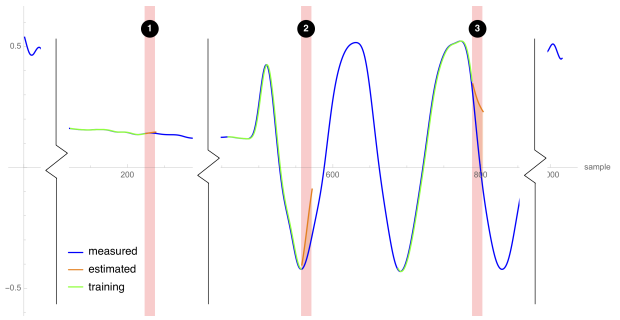

Figure 11: Extrapolating missing values

restricted transfer mechanisms. A good example of such low-powered sensor is the Pebble smartwatch. The first model contains the Cortex-M3 CPU at 64 MHz with 24 KiB of memory (heap and code) for applications. 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 a theoretical maximum of 10 640 B s^{-1}. The Pebble implementation dropped the reliable data transfer rate to only 280 B s^{-1}. To minimise the amount of data being transferred, the system takes into account maximum reasonable acceleration a human can achieve in exercise: $\pm 40\,\mathrm{ms}^{-2}$, and represents that range as 13 bit integer. This means one sample is 5 B: 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[5].

4 Mobile application

Even though modern smart-watches are powerful enough to perform most of the computation, their screen sizes are a bit too small to display all the necessary information; most importantly, though, a smart-watch provides only one set of sensors. This means that exercises where the wrist where the user wears the watch does not move would not be possible to recognise. This is where the mobile application comes in: if the user places the mobile in his or her pocket, we have another sensor source.

Crucially, all models that the mobile application needs are local to the device–the models are downloaded at application start-up (using the details embedded in the authentication JWT). The models are the nodes and transitions for the Markov chain, the neural net parameters for the set-up movement and movement classifiers. The mobile application's core functionality is packed into three components: *Predictor* , *Classifier* , and *SignalProcessor* . Perhaps counter-intuitively, the *Predictor* is the most important element of the user experience. If it makes the wrong predictions, *the paper diary will be immensely more convenient.* The *Predictor* is a fairly naïve implementation of the Markov chain with *Predictor.Node* as its nodes (see Listing 1).

```
class Predictor {
  func predict() → ScoredSet<Node> { ... }
  func update(taken: Node) { ... }

  enum Node {
    case exercise(label: String, ...)
    case rest(duration: TimeInterval?)
  }

}
```

Listing 1: Predictor

If the *Predictor* predicts exercise (with sufficient probability), it will then use the *Classifier* initialised with the set-up movement model to try to find movement that is consistent with getting ready for the exercise. Its public interface is shown in Listing 2.

```
class Classifier {

  func classify(fsd: FusedSensorData) →
    ScoredSet<String> {
    ...
  }

}
```

Listing 2: Classifier

Before Core ML[1], the application included own hand-rolled forward-propagator configured with the appropriate model. It explicitly modelled the activation functions, convolution, and fully-connected layers as runtime structures. (See Listing 3.)

```
enum ActivationFunction {
  case identity
  case sigmoid
  case tanh
  ...
}

enum LayerConfiguration {
  case fullyConnected(size: Int, weights: [Float],
                      function: ActivationFunction)
  case convolution(...)
  ...
}

class ForwardPropagator {
  State such as featureVectorSize : Int, predictionVectorSize : Int,
  layers : [LayerConfiguration], weights, and others.

  init(data: NSData) throws {
    ...
  }
}
```

Listing 3: Runtime structures for NN

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

Using the SIMD functions for the computational steps (see Listing 4) gave the application good power-efficiency compared a pure CPU-bound implementation, but larger networks still incurred significant memory impact.

```
extension ActivationFunction {

  func apply(inout input: [Float],
             ofs: Int, len: Int) {
    switch self {
    case .identity:
      x
      return
    case .sigmoid:
      1/(1+e^-x)
      vDSP_vneg; vvexpf; vDSP_vsadd; vvpowsf
    case .softmax:
      e^xk / sum_{i=1}^{len} e^xi
      vDSP_maxv; vDSP_vsadd; vDSP_sve; vDSP_vsdiv
    ...
    }

}
```

Listing 4: SIMD

With Core ML's availability, the application's code has become even more energy and memory efficient, though at the small expense of pre-compiling the newly downloaded models from the server. Finally, Listing 5 shows that the interface for the signal processor–the *SignalProcessor* –is just as simple; however, the implementation again makes most of the SIMD processing capabilities in iOS.

```
class SignalProcessor {

  func process(fsd: FusedSensorData) → [Element] {
    var conv = fsd.newBuffer(padded: false)
    var corr = fsd.newBuffer(padded: false)

    conv = conv + signal
    vDSP_vadd

    Compute autocorrelation of the padded signal with itself
    var paddedSignal = fsd.newBuffer(padded: true)
    vDSP_conv(&paddedSignal, ..., &corr, ...)

    Normalize the correlation values between 1 and -1
    vDSP_vsmsa(&corr, vDSP_Stride(1), ...)

    Count peaks; measure jitter, etc
    return ...
  }

  enum Element {
    case repetitions(repetitions: Int)
    case intensity(intensity: Double)
    ...
  }
}
```

Listing 5: Signal processor

Even though this code required significant effort, it meant that all processing could be kept on the device. No sensor–biometric!–data has to be sent for further processing. Even if privacy were not a concern, processing the sensor data on the server would bring unacceptable latency; it would significantly increase the application's energy impact (the radios would have to be kept on); it would also significantly increase the server-side processing bills!

5 Server

Even though the mobile application can handle all the required computation for a single user, a server-side code is still required to be able to create and then improve the models that the mobile code runs. Figure 12 shows the main components and the relevant technologies.

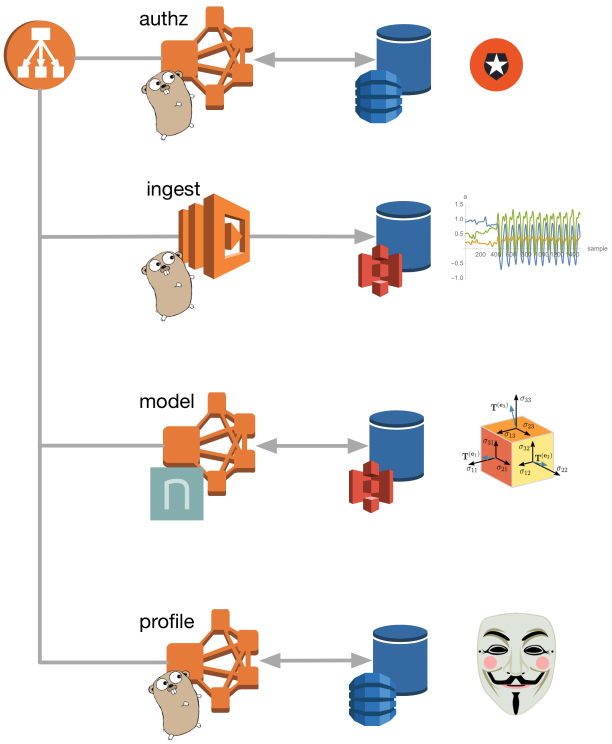

Figure 12: Overall architecture

In any machine learning system, the most valuable component is the data. The first consideration is the data size: after compression, 1 h of sensor data (accelerometer and gyroscope from the wrist; accelerometer and gyroscope from the waist; heart rate) and metadata (coarse and fine location, Markov chains, labels) takes up approximately 2 MiB. This 1 h represents a typical exercise session; with just 1000 users (with 50 % training load) the system ingests 1 GiB every day. At the experiment phase, it is sufficient to implement all services in simple CRUD-style in Go. If the usage grows significantly, some of the services may require the CRUD services to be entirely re-architected using CQRS/ES architecture and reactive toolkits such as Akka[4]. Nevertheless, with a low number of users and low peak loads, the system's services are sufficient to be simple stateless services that rely on the underlying infrastructure to maintain their state[6].

[6] In practice, this means that the services are essentially CRUD-style.

5.1 Ingestion & storage

The ingestion accepts the sensor data, the Markov chain data, and any labels from one or more exercise sessions from the mobile. The submission is a gzipped Protocol Buffers[10] binary representation of a message conforming to Listing 6.

```
All session contain the user_token and array of Session objects
message Sessions { ... }

message Session {
  Metadata about the session such as timestamp, location,
  weather, targetted muscle groups, etc.

  Sensor readings
  repeated Sensor sensors = 10;
  repeated float sensor_values = 11;
  Labelling behaviour
  repeated Label labels = 20;
  Behavioural and user-specific exercise values
  ExerciseSequence exercise_sequence = 30;
  repeated ExerciseProperties exercise_properties =
    31;
  Classification models used
  repeated string model_references = 40;
}

message Label {
  uint32 timestamp = 1;
  Source source = 2;
  map<string, double> predicted_exercise = 3;
  string actual_exercise = 4;
  ExerciseProperties predicted_properties = 5;
  ExerciseProperties actual_properties = 6;
}

message Sensor {
  SensorLocation location = 1;
  repeated SensorKind kinds = 2;
}

L/R wrist, L/R arm, L/R waist, whole-body
enum SensorLocation

acceleration, rotation, heart rate, etc
enum SensorKind
```

Listing 6: Session protocol

Typical ingested *Sessions* object contains a single session of 45 min. This typical session contains acceleration + rotation from samples wrist and waist sensors in *sensor_values*. The layout and stride of the *sensor_values* array is defined by the elements of the *sensors* array (acceleration & rotation yield 3 values in each sample; heart rate yields one value). The *labels* array contains all predictions made during the session; there are multiple predictions in one exercise block: the multiple values represent refinement of the exercise being performed and updated *exercise_properties* with number of repetitions, intensity, weight, etc. If the user decides to correct the predictions, the *Label* instance includes the *actual_properties* and *actual_exercise* with the correct values. With additional metadata, user tokens, etc; a typical ingestion payload is 3.2 MiB.

The ingestion service is a plain REST API that accepts POST requests whose bodies conform to the *Sessions* protocol. The ingestion service verifies the structure of the payload, then persists the entire payload in an Amazon S3 bucket. The subsection 5.2 take the saved *Sessions*, decompress them and then operate on the elements they need. There are no intermediate forms for the data–once the Protocol Buffers is known not to contain malicious data, its memory layout makes it easy to load the entire message into memory without additional transformation steps.

5.2 Model computation

The mobile application uses series of models to predict exercises that are coming up, and to classify the movements. Different [clusters of] users behave and exercise diffently, so the system needs to build specific models for each cluster. The models take blocks of sensor samples as input, which means that the different combinations of sensors need different models. A typical example is a user with a smart-watch and a phone: two sensors capable of providing the acceleration and rotation samples, and a heart rate sensor. The mobile phone runs $classify\ (model,\ samples) \rightarrow result$. The server needs to compute *model* such that it predicts any input sample vector as well as possible. In terms of bits and bytes, the model is a file that contains the model's hyper-parameters and parameters. Hyper-parameters describe the "shape" of the model's elements, the parameters describe the constant parameters that the model's elements need in order to perform their computation.

The process to compute and evaluate the models needs to be stable and repeatable; the model's parameters are often initialized using some *seed* values, and then the computation proceeds by adjusting the seed values in order to minimise the classification error. This means that the model computation is $train\,(data,\ hyper{-}parameters\ ,\ initial{-}values) \rightarrow model$. If the model's classification error measurement uses the same data as the data used for training, the model can be over-fitted: it is very accurate on the training data, but it fails to classify data not included in the training set. So the model computation needs to be $train\,(training{-}data,\ verification{-}data,\ hyper{-}parametrs,\ initial{-}values) \rightarrow model$. The *model* is the combination of hyper-parameters and parameters; so the model computation can be $train\,(training{-}data,\ verification{-}data,\ model) \rightarrow model'$. The constraints of computing environments mean that the *train* function mutates the *model*, and the *training−data* and *verification−data* are iterators that yield small batches of data.

Every time the *train* function runs, it produces a model that best fits the input data. The newly trained model may be the best model for the verification data set; if the verification data set did not change between this model and all previous models, then the newly trained model becomes the best model. However, if the verification data set has changed, the latest model might not be the best, even if the evaluation results for all previously trained models are worse. It is not feasible to keep a long list of previous models because their performance data becomes stale as the system collects more data. The system keeps only the best 3 models. If the verification data set changes, the system re-evaluates the three models and then discards the worst-performing model. The system does not maintain just a single model.

During the implementation, the first iteration of the ML code used manual feature-extraction, encouraged by "eye-

balling" several graphs of acceleration in Figure 13.

Figure 13: Visualised exercises

Taking one of the exercises, the features $f_{1,2}$, $f_{3,4}$, and $f_{5,6}$ are the extrema of the acceleration over some window of time; $f_{7,8,9}$ are the most powerful harmonic component of x, y, z of the sample window. See Figure 14.

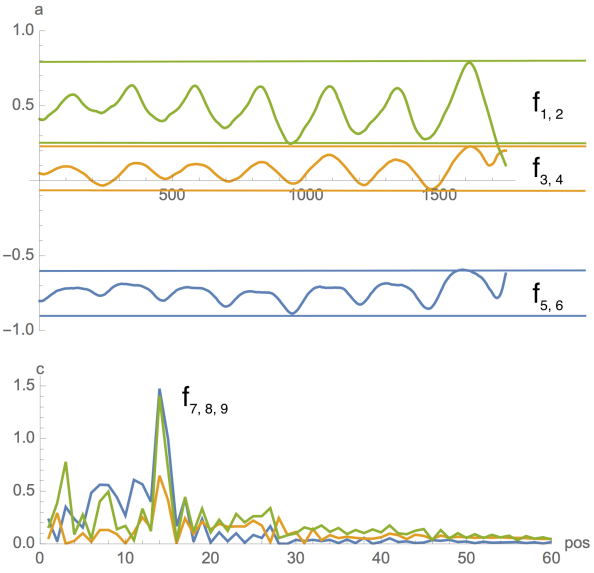

Figure 14: Manual features

A multi-class SVM was trained using the extracted features; Table 1 shows the classifier's performance after training, but without taking into account the *expected* next exercise from the behavioural model.

With the behavioural model, the classifier's performance improves significantly, with only roughly 20 misclassifications in each class.

Another approach is to not attempt to extract the features manually, but to use a neural network with the smoothed vector of sensor readings. The simplest set-up movement

Predicted	Actual		
	biceps curl	triceps ext.	lateral raise
biceps curl	452	112	87
triceps ext.	37	358	54
lateral raise	46	131	387

Table 1. Manual features and SVM (no boost)

model uses only single sensor reading: for example, the *accelerometer on the left wrist* yields $\vec{I} = [\vec{x}, \vec{y}, \vec{z}]$, 1 s window ($|\vec{I}| = 150$) is passed as the input to the model.

Each new sensor adds dimensions to the input vector; the sensor reading vectors are always appended in a well-known order (acceleration then rotation; wrist then arm then waist; then single heart rate). An acceptable simplification is to assume that the sensor on the arm or near waist (think a pocket) is a mobile, and so it produces acceleration and rotation. This leaves us with the combinations of left or right wrist sensor (which can record acceleration only, or acceleration and rotation) *and* left or right arm sensor *or* left or right waist sensor; a total of $2 \times (4 \times 2)$ set-up models, and the same number of in-exercise models. The mobile application selects the right model when it knows the precise sensor setup that the user decided to wear. The simplest model's parameters (and hyper-parameters) need only 1.3 MiB (compression reduces this by another 200 KiB); the most complex model's (acceleration + rotation; acceleration + rotation; heart rate) parameters take up 18 MiB. In the worst-case scenario, the mobile application needs to download roughly 100 MiB of model parameters[7]

Table 2 shows the trained model's performance: as you can see, it is significanly better than the manual-features-with-SVM approach. (With behavioural model, the number of misclassifications drops to 1 in each class.)

Predicted	Actual		
	biceps curl	triceps ext.	lateral raise
biceps curl	529	6	3
triceps ext.	3	591	3
lateral raise	3	4	522

Table 2. Neural network

The trained network performs just as well with larger number of exercises targetting the same muscle group (arms); even with exercises that are very similar. This network relies on fused sensors from the wrist and waist: otherwise, it would struggle to clearly identify *triceps dips*, where the wrists are fairly stationary, but the body moves up and down. Table 3 shows the network's performance.

[7] I only complain about 100 MiB when the mobile application used up a lot of my roaming data allowance while I was on a business trip in the US.

	Classified						
Actual	i	ii	iii	iv	v	vi	vii
i	2153	1	3	1	0	0	1
ii	21	5687	7	3	2	11	0
iii	1	0	1515	3	0	2	0
iv	2	3	1	2742	4	12	0
v	1	0	0	0	2544	0	0
vi	3	3	4	8	4	3371	0
vii	1	0	1	0	75	55	812

Where

i. Triceps dips
ii. Alternating dumbbell biceps curl
iii. Barbell biceps curl
iv. Straight bar biceps curl
v. Rope biceps curl
vi. Rope triceps extension
vii. Straight bar triceps extension

Table 3. Neural network (arms)

In addition to the neural network's better performance, it significantly cuts the development time: beyond low-pass filtering on the raw sensor readings, the sensor data does not need any processing. The neural network's output are the classes of exercise. (Or, if trained so, classes of set-up movements.) The networks have been trained on data gathered from a small number of users exercising over nearly 12 months; this dataset was then used to generate further training and test using a bio-mechanical model.

6 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[9] directive coming into force. Security and privacy extends all the way to the mobile applications and the sensors. ([3] explores viable attacks on sensors.) 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 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 deserialisation from the OWASP[8] list) does not lead to *A3:2017*: sensitive data exposure. The system uses JWT[5] 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 15 illustrates the entire flow.

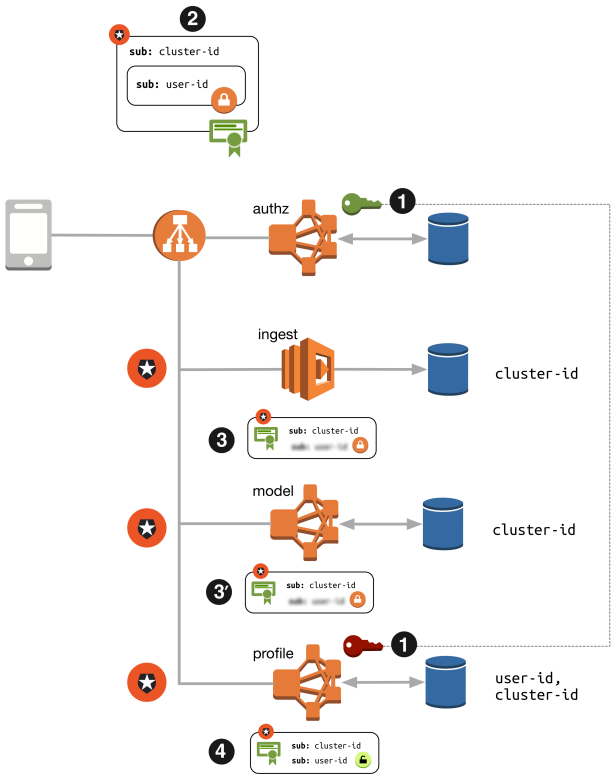

Figure 15: Authorization and data access

Initially, in ① and ⑪ 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. They can use the token to authenticate access, and they use and persist the cluster identifier in their data stores. The *profile* ⑤ 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[8].

7 Further work

There is a lot of further areas of research. I think that exercise should firstly bring enjoyment and fun: this encourages further exercise, and only long-term sustainable exercise brings significant health benefits. The system collects all the information about the completed exercises (names, intensities, timings, sequence); it will then be useful to *ask* the users to indicate their enjoyment after each exercise session. This can be as simple as 🙂 , 😐 , and ☹ ; and then use the server-side processing to identify the exercises, their sequences, and intensities that the particular user (and other users in the same cluster) find enjoyable. Similarly, the concept of improvement needs further exploration: for some users, improvement means being able to lift more weight; but for users recovering from an injury, improvement means increasing their range of movement.

8 Sounds great; where do I begin?

... TODO: ML Joel test TODO: pipeline

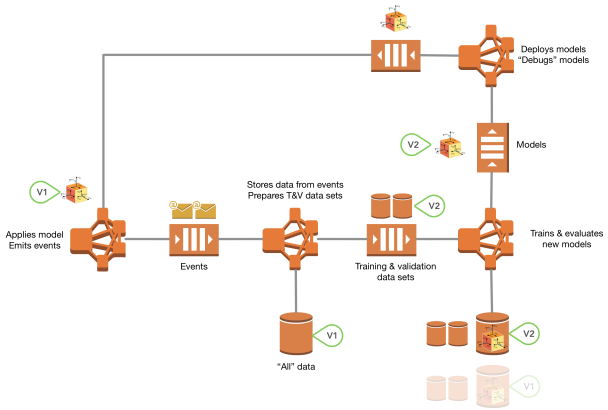

Figure 16: ML pipeline

[8] The system does 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.

And with code...

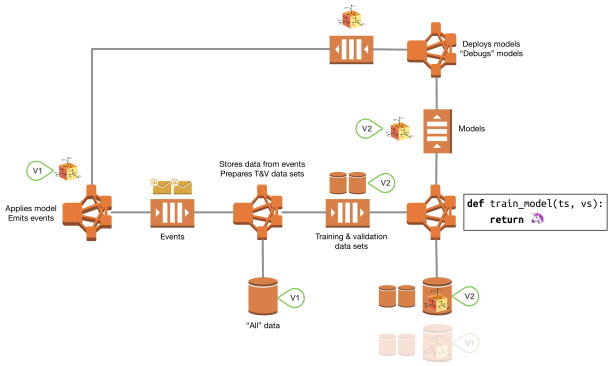

Figure 17: ML pipeline with code

The ML team maintains the tooling for the pipeline, consults on the best models, researches, ...; but the product teams (that ultimately work on the service that *uses* the model) have the first dibs on implementing the model. Successful implementation of this strategy means that anyone can implement a new model (even if only to just see what will happen!), train it, debug it, and deploy it all within a single day. All the mechanics of data ingestion, storage, versioning; runtime of training a model, evaluation, storage, versioning; debugging and deploying; and the usage is all implemented.

In this sense, the machine learning code is just like any other ordinary code; it is subject to all the high engineering standards and safeguards.

References

[1] Apple. *Core ML*. `https://developer.apple.com/documentation/coreml`.

[2] Apple. *iBeacon*. `https://developer.apple.com/ibeacon/`.

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

[4] Lightbend Inc. *Akka*. `https://akka.io`.

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

[6] James F Knight et al. "Uses of accelerometer data collected from a wearable system". In: *Personal and Ubiquitous Computing* 11.2 (May 2006), pp. 117–132.

[7] Dan Morris et al. "RecoFit". In: *the 32nd annual ACM conference*. New York, New York, USA: ACM Press, 2014, pp. 3225–3234.

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

[9] The European Parliament and the Council of the 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`.

[10] *Protocol Buffers*. `https://developers.google.com/protocol-buffers/`.