Spark Reference Architecture
Sensor batch processing

Jan Macháček

***Abstract*—TODO**

I. Introduction

TODO

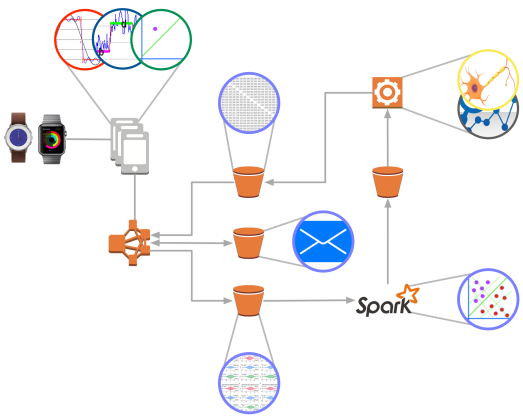
II. Reference implementation

This architecture was used in a connected fitness application. The application processes inputs from one or more sensors (smartwatch, HR sensor, smart clothes) to track fitness regimes and to deliver targeted health and fitness advice to the users.

The application on the user's smartphone connects data from the available sensors, combines it with a statistical model of the user's behaviour, and—where available—fine-grained location services. These three inputs allow the application to make the first distinction: exercise vs. no-exercise. From the user's perspective, the system is an automated fitness trainer; from a data scientist's perspective, the biometric data the system collects allows for detailed analysis of exercises, exercise regimes, impact of exercise on the users' well-being, automated physiotherapy, and many other applications.

A. Main components

Fig. 1. Components



The sensors shown here as consumer-grade wearables perform only the basic hardware interaction: there is no preprocessing of the recorded data. The system accepts accelerometer, gyroscope, heart rate and—where available—data from strain gauges in smart clothes. The mobile application performs the real-time processing of the inputs, displays the next exercise the user should perform (allowing the user to change the suggestion by simply walking to a station for a different exercise, or by beginning a different exercise). The mobile application also prompts the user to confirm correct labels for the recorded data. *This is the key component in the entire system: the frictionless user experience gives the system very accurate labels on very clean data.*

The mobile submits the entire session data (the sensor data, matching labels and latest user behaviour models) in a single request to the Akka [2] cluster—a CQRS/ES [3] microservice implementation. The Akka cluster stores its events and snapshots in a journal, implemented by the Apache Cassandra [4] database. Alongside the events and snapshots, which are opaque to non-Akka systems, the *sensor data* microservice saves the sensor data and the matching labels in a properly formed tabular structure. This structure allows the Apache Spark [5] cluster to be used for typical big data tasks. An important consideration is privacy and security of the data: the system does not use stable identifiers for the biometric data. This is similar to the unstable advertising identifier used in, for example, the iOS devices [7]. The unstable *biometric identifier* can be refreshed at arbitrary points in time; for very sensitive applications (e.g. clinical physiotherapy), we refresh the biometric identifier after each session; for typical consumer scenarios, we give the users the option to refresh the biometric identifier.

The Apache Spark cluster reads the fitness profiles (associated with the unstable biometric identifier) and computes clusters of users by examining the values in the fitness profile. Even though the profile information holds self-reported data, the data we store include age bracket ($(18; 25)$, $(25; 35)$, $(35; 45)$, ..., $(75; \mathrm{inf})$), sex, self-reported fitness level (beginner, intermediate, enthusiast, athlete), self-reported weight and self-reported height.

Once we have the biometric identifiers for each cluster, we read the sensor data, together with the single-session and all-sessions metadata, and find:

- The Markov chain of exercise sessions that results in greatest improvement (where improvement may be muscle mass gain, fat loss, and *hapiness*.)
- The Markov chain of exercises in a session that results in greatest improvement (where improvement may be muscle mass gain, fat loss, and *hapiness*.)
- The most and least popular exercises

The output of these *big data* tasks is written back to the Apache Cassandra database; the Akka cluster reads the exercise-related output to provide advice back to the users; the machine learning training and evaluation programs reads the

sensor-data related output to compute new models to recognise the exercises.

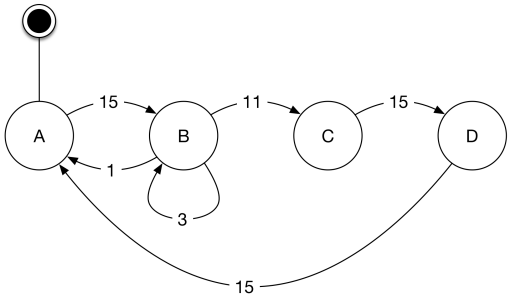
The models to predict exercise from sensor data are implemented in a cluster of keras [8] training and evaluation programs. These programs read the sensor data for each cluster, expand the sensor data to each combination of sensors, then train the most successful convolutional neural networks identified through random evolution of previous training executions. (The first execution is human-defined, starting with a three-layer network.) For every sensor data split and every cluster, the training program selects n most successful CNN hyper-parameters from the previous runs, then divides the data into training and test datasets. It then randomly mutates m ($m << n$) CNNs before fitting each CNN to the testing data. The evaluation program then evaluates all new n CNNs, keeping only the most successful ones (defined by the evaluation's F1 score).

The models' hyper-parameters, parameters and evaluation scores are written back to the database. The Akka cluster then selects the best model (hyper-parameters and parameters) and uses the Apple content delivery network to push it to the users with the matching biometric identifiers. (This enables the Akka cluster to concentrate on its primary task: sensor data ingestion, leaving all other data manipulation and transfer tasks to external services while maintaining frictionless user experience.)

The sensors send the data in $1\,\mathrm{s}$ batches; the application on the mobile resamples the sampling rate to $50\,\mathrm{Hz}$. (Accelerometer and gyroscope typically sample at this rate, heart rate and the strain gauges in smart clothes can be upsampled to $50\,\mathrm{Hz}$ easily.) The mobile application ingests the data from the sensors and, together with a statistical model of the user's short-term behaviour, attempts to identify the movement. The models in the mobile application can make successful predictions of sequence of exercises in a session and the properties of each exercise (i.e. weight, number of repetitions, duration, intensity). The model used to predict sequence of exercise sessions and exercises within a session is a Markov chain [1]. This approach 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 2.

The numbers in Figure 2 represent the count of transitions taken; hence it is possible to calculate the probability of transition from any given state. The state names represent 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 choses 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

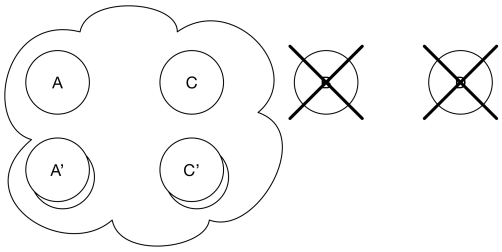
Fig. 2. Exercise sequence model with no context



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. (At this point, we do not define what the improvement is: in some applications, it may be weight loss; in other applications, it may be greatest mobility range improvement; and many others.)

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 [6]; 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. (Viz Figure 3.)

Fig. 3. Exercise sequence model with location context



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 *setup movement*. The setup movement classifier is a multi-layer perceptron, which takes $1\,\mathrm{s}$ of sensor input and produces probabilities classes that represent the setup movement for groups of exercises. The hyper-parameters of the MLP is driven by the sensor data as its inputs; for example, accelerometer-only MLP has 150 inputs (50 samples of the acceleration vector) and as many outputs as the number of recognised setup movements. It is important to measure and optimise the power requirements for

the computation; on iOS, we took advantage of the vDSP and veclib frameworks, which offer optimised vector operations. The MLP classes are not the exercises themselves, but the setup movements; one setup 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. The performance of the exercise prediction for accelerometer on the user's wrist is shown in Table I; the performance of the exercise prediction rises significantly in fully-wired human (viz Table II).

TABLE I
EXERCISE PREDICTION PERFORMANCE (ACCELEROMETER)

	Accuracy	Precision	Recall	F1
No context	!!	!!	!!	!!
Behaviour	!!	!!	!!	!!
Behaviour, location	!!	!!	!!	!!

TABLE II
EXERCISE PREDICTION PERFORMANCE (ALL SENSORS)

	Accuracy	Precision	Recall	F1
No context	!!	!!	!!	!!
Behaviour	!!	!!	!!	!!
Behaviour, location	!!	!!	!!	!!

Finally, the mobile application asks the user to confirm the label for the collected sensor data. Given the good performance of the exercise recognition (particularly when using fine-grained location services), this is typically just a *confirm* step rather than complex data entry exercise. The mobile application builds a local store of the time regions of labelled sensor data. This store, together with the Markov chain representing the exercise sequence and the final feedback (happy, indifferent, unhappy) is submitted to the server. The server decompresses & decodes the submitted data and stores it in tabular form in Apache Cassandra, referencing the user's biometric ID.

B. Mobile and wearables

The mobile application is an iOS only app at the moment; the consumer-grade wearable companions are a watchOS app and a Pebble (aplite, basalt, and chalk) flavours. The wearables send the data to the mobile application over BLE. The protocol as well as the hardware architecture of the devices places restrictions on the sample width and sampling rate. BLE protocol specifies maximum $80\,\mathrm{B}$ per message, and minimum duration of $7.5\,\mathrm{ms}$ 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 $20\,\mathrm{B}$ for header and $5\,\mathrm{B}$ 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, $40\,\mathrm{ms^{-2}}$, and represented sample as $3{\times}13\,\mathrm{bit}$ for x, y, and z axes (with $1\,\mathrm{bit}$ 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 $50\,\mathrm{Hz}$ one sample represents $20\,\mathrm{ms}$ of real-time, but the difference of the timestamps may not be divisible by 20. We found that 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.

Because the consumer-grade sensors often do not have enough processing and communication power to collect & transmit the data without errors, the mobile application also has to pad missing values from the sensors in order to remain responsive and to be able to provide a full set of sensor values for subsequent processing. We found that for gaps less than $200\,\mathrm{ms}$, it is sufficient to extrapolate the samples linearly between the sides of the gap regardless of sensor type. Longer gaps require special treatment in case of accelerometer, gyroscope: in these cases, the application uses the Kalman filter assuming constant acceleration of $10\,\mathrm{ms^{-2}}$ to extrapolate the values between the sides of the gap. The heart rate sensor does not suffer significant loss of accuracy for gaps up to $15\,\mathrm{s}$. Once the sensor data from all sensors has been fused with respect to the sample time, the mobile application performs very simple signal processing: it smooths the signal by convolving it with a 3×3 kernel (Equation 1).

$$\begin{pmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{pmatrix} \tag{1}$$

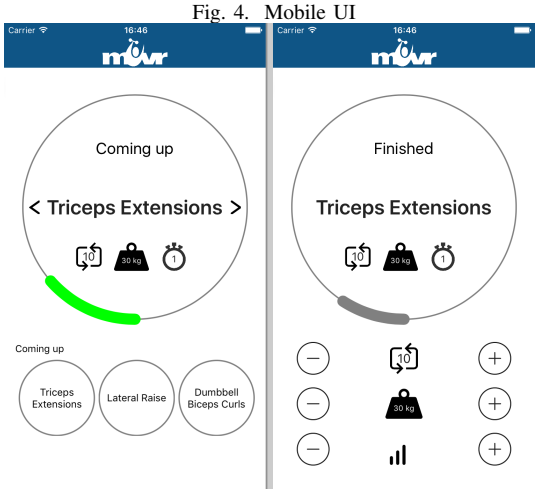
Finally, the sensor data in the mobile application is a set of vectors containing 1 second of sensor input data, normalised to a range of $(-1; 1)$ by using a maximum reasonable value for acceleration, rotation, strain and heart rate. The maximum values are determined by the limitations of the sensor hardware together with reasonable limitations of human movement and together with the ranges of the protocol. For the three-value protocol used for the accelerometer and gyroscope data, and for the strain sensors, the limit of each element is the range of signed $13\,\mathrm{bit}$ integer: $(-4096; 4095)$. The unit of the measurement depends on the underlying sensor type: $0.01\ ms^{-2}$ for accelerometer, $\pi/1800\,\mathrm{rad\,s^{-1}}$ for gyroscope, and so on.

The fused and normalised sensor data is then passed as a single vector to the inputs of the CNN that matches the types of sensor inputs, and its outputs are the classes of setup movements. It is important to measure and optimise the power requirements for the computation; on iOS, we took advantage of the vDSP and veclib frameworks, which offer optimised vector operations. As the classes are not the

exercises themselves, but the setup movements; one setup 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.

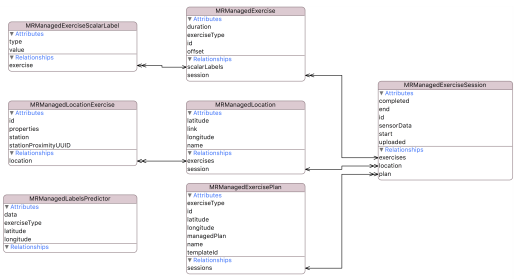
The result on Figure 4 is—what we believe—an application which successfully guides users through their exercise programmes, adapts to changes, encourages improvements; all with *minimal interaction during the exercise session*.

Fig. 4. Mobile UI



The mobile application records all received sensor data locally; however, it only transmits the blocks labelled as exercise to the server. The local storage allows us to handle the device being offline (or simply a situation where the user does not want to use his or her data allowance). Figure 5 shows the data model; notice the *sensorData* attribute in *MRManagedExerciseSession*, which holds all data recieved from all sensors; the *MRManagedExercise* with at least one *MRManagedExerciseScalarLabel* then represent labelled time slices of the *sensorData*. A compact representation (Protobuf [14]) of the *MRManagedExerciseSession* is submitted to the server for further processing.

Fig. 5. Mobile data model

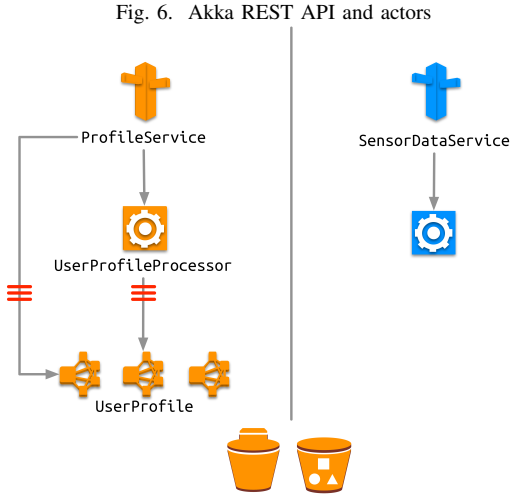


C. Server components

The Akka cluster is a CQRS/ES system, which exposes REST APIs for the mobile application. The REST requests it receives are treated as commands; once a command is validated, the code in the cluster turns it into an *event* and persists it in the journal. Later on, another component may represent the system's state as the sum of all the events in the journal. This *command / query responsibility separation* approach makes it possible to efficiently distribute the processing required by the system's components. (For example, if the users often request a view of the state of the system, then the *query* components receive more resources.) The Akka toolkit uses Actor-based concurrency: within an actor, all computation is synchronous; the toolkit handles asynchronous communication between actors. This allows us to maintain clear boundary on any mutable state: if is kept within an actor, it is safe.

Internally, there are two main kinds of microservice: one that does not use or need clustering or cluster sharding, and one that does. The example of clustered microservice is the *user profile* microservice; its state is all user profiles, which would not fit in a single node. And so, it needs to be clustered over multiple nodes, with all the complexity of routing the requests to the right actors, cluster rebalancing & error recovery. The *sensor data* microservice is not clustered: the nodes that comprise it do not need to know about each other; each node is completely stateless, therefore requests arriving at the microservice can be routed to any node. Figure 6 shows the two microservices.

Fig. 6. Akka REST API and actors



In the Lightbend stack parlance, the *service* is the name for the interface (in this case, the REST API; the *ProfileService* and *SensorDataService*). It processes the HTTP requests, turning them into messages, which are then delivered to the underlying actors. In case of the *sensor data* service, the actor decodes the sensor data payload and persists it in the sesor data database. In case of the *user profile* service—a CQRS/ES microservice—the HTTP request is processed in the same way, turning it into a *command* message. This command is sent to the *UserProfileProcessor*, which validates the command and turns it into an *event*. In the CQRS/ES world, we say that an event event is never lost and it will be processed at some point

by the *UserProfile* view. The processor is typically stateless with respect to the commands; the views typically hold state, which is the result of processing all events.

The separation of command and query processing code allows the system to scale selectively: profile views are more frequent than views; the combination of supervision in Akka and persistence in actors allows the system to scale up and down, and to handle errors gracefully.[1]

D. Apache Spark batch analytics

Vivamus urna velit, volutpat ut tincidunt sit amet, pulvinar vitae est. Nam tortor sapien, sagittis non luctus sit amet, porttitor a purus. Donec mattis blandit bibendum. Morbi ipsum lacus, gravida ut nibh semper, porta tincidunt ex. Sed tellus lectus, posuere in facilisis quis, suscipit eu tortor. Maecenas sagittis diam non orci scelerisque, vehicula rutrum ipsum elementum. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin bibendum justo gravida sem accumsan consequat. Nunc tristique aliquam magna, id convallis tellus gravida et. Sed orci est, tempus eu ligula sed, posuere feugiat nunc. Sed est mauris, dignissim dignissim pretium in, elementum a neque. Phasellus mollis sollicitudin justo, ut commodo turpis volutpat in. Fusce egestas nunc dui, vel mollis tellus interdum vel. Donec porttitor lorem non mauris porttitor, euismod vestibulum nisi tincidunt. Nunc fringilla risus nulla, id vulputate sem blandit ut. Proin est mauris, viverra in interdum quis, fringilla at augue.

Vivamus urna velit, volutpat ut tincidunt sit amet, pulvinar vitae est. Nam tortor sapien, sagittis non luctus sit amet, porttitor a purus. Donec mattis blandit bibendum. Morbi ipsum lacus, gravida ut nibh semper, porta tincidunt ex. Sed tellus lectus, posuere in facilisis quis, suscipit eu tortor. Maecenas sagittis diam non orci scelerisque, vehicula rutrum ipsum elementum. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin bibendum justo gravida sem accumsan consequat. Nunc tristique aliquam magna, id convallis tellus gravida et. Sed orci est, tempus eu ligula sed, posuere feugiat nunc. Sed est mauris, dignissim dignissim pretium in, elementum a neque. Phasellus mollis sollicitudin justo, ut commodo turpis volutpat in. Fusce egestas nunc dui, vel mollis tellus interdum vel. Donec porttitor lorem non mauris porttitor, euismod vestibulum nisi tincidunt. Nunc fringilla risus nulla, id vulputate sem blandit ut. Proin est mauris, viverra in interdum quis, fringilla at augue.

Vivamus urna velit, volutpat ut tincidunt sit amet, pulvinar vitae est. Nam tortor sapien, sagittis non luctus sit amet, porttitor a purus. Donec mattis blandit bibendum. Morbi ipsum lacus, gravida ut nibh semper, porta tincidunt ex. Sed tellus lectus, posuere in facilisis quis, suscipit eu tortor. Maecenas sagittis diam non orci scelerisque, vehicula rutrum ipsum elementum. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin bibendum justo gravida sem accumsan consequat. Nunc tristique aliquam magna, id convallis tellus gravida et. Sed orci est, tempus eu ligula sed, posuere feugiat nunc. Sed est mauris, dignissim dignissim pretium in, elementum a neque. Phasellus mollis sollicitudin justo, ut commodo turpis volutpat in. Fusce egestas nunc dui, vel mollis tellus interdum vel. Donec porttitor lorem non mauris porttitor, euismod vestibulum nisi tincidunt. Nunc fringilla risus nulla, id vulputate sem blandit ut. Proin est mauris, viverra in interdum quis, fringilla at augue.

[1]There are many topics that we have not covered, especially around message delivery semantics in distributed systems.

E. Machine learning

The ML training and evaluation programs use Python and run in Docker containers. To support fast development cycle, we build two flavours of Docker images: a *light* image that can run on nearly any underlying hardware, and a CUDA [19] image that runs on the EC2 *g2.2xlarge* instance. The latest development work uses Deeplearning4j [16] in order to standardise the infrastructure; the improvements in the underlying numerical library implementation (Nd4j) for both CPUs and GPUs should allow us to move from the Python / Theanu implementation to a JVM-based one. This will simplify our infrastructure, even though the model computation can execute on a single node; the paralelisation is achieved through running multiple nodes. Regardless of the underlying implementation, the machine learning microservice performs the following loop:

- Generate new model hyperparameters by appending randomly mutated hyperparameters to the current set of hyperparameters,
- For each model hyperparameters, construct a model and fit the training dataset,
- Measure the fitted model's performance using the test dataset,
- Save the measured performance and hyperparameters

The hyperparameter mutating code is fairly basic at the moment; it can only

- Mutate dense layer's activation functions
- Mutate dropout layer's rate
- Mutate convolution layer's kernel
- Remove or add layer
- Mutate learning rate, momentum and and decay of the SGD optimiser

Notice also that our evolution code is purely random; there is no optimiser. This code does produce well-performing models, though *its energy or cost efficiency is extremely poor*. Figure 7 illustrates this process for 2 models hyperparameters.

The final consideration is the sensor data explosion: this is something that we actually need in our system to be able to make as much of the collected data as possible. While most users wear only a single smartwatch (which gives us accelerometer data from one wrist), there are some users who wear many more sensors. What we call a *fully-wired human* wears accelerometer and gyroscope on the wirst, together heart rate monitor and sports clothes that contain strain gauges along major muscle groups. This gives us (at the moment) 2 data points from the wrist, 1 from the heart rate sensor (we drop

Fig. 7. Model evolution

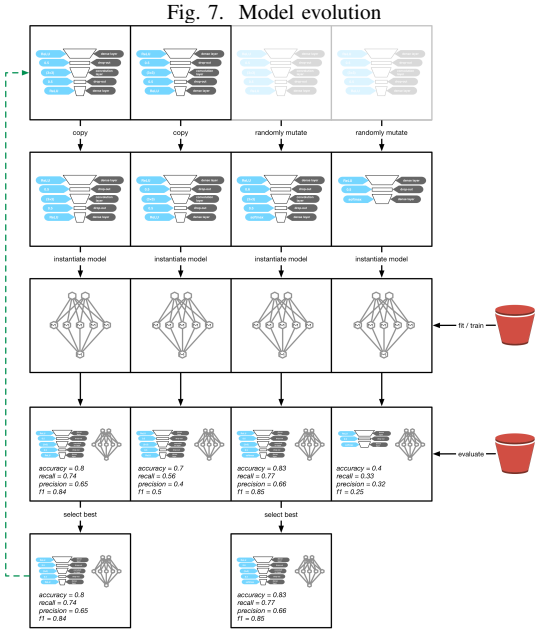
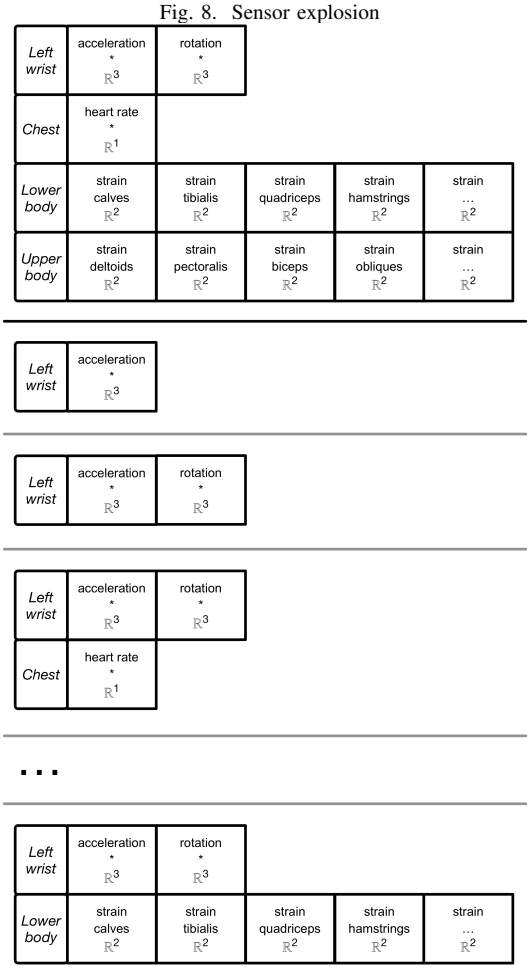


Fig. 8. Sensor explosion



down to simple heart rate, we do not measure the full ECG traces), and *20 data points from the muscle groups*. Just under $50\,\%$ of our users fall somewhere between the smartwatch-only and fully-wired categories. To ensure that we make the most of every user, the training program needs to expand the recorded data into all known sensor combinations. So, for a dataset with n sensors, we expand it into $\binom{n}{n} + \binom{n}{n-1} + \binom{n}{n-2} + \ldots + \binom{n}{1}$ datasets, as illustrated on Figure 8.

The processing code maintains a predetermined ordering of the sensor samples to column vectors to the dataset[2]. The ML microservice persists the model hyperparameters, parameters, evaluation results, the required sensor inputs, and a mapping to the cluster of biometric IDs into the database cluster.

The mobile application checks for updates to the best model (hyperparameters, parameters and the required sensor inputs); if a new model is available, it prompts the user to download it. (Future versions may support paid-for models through the non-consumable in-app purchase model; in this scenario, the Akka cluster is responsible for delivering the updated non-consumable content to Apple in order to make it available to the users who have in-app purchased it.)

III. Server infrastructure

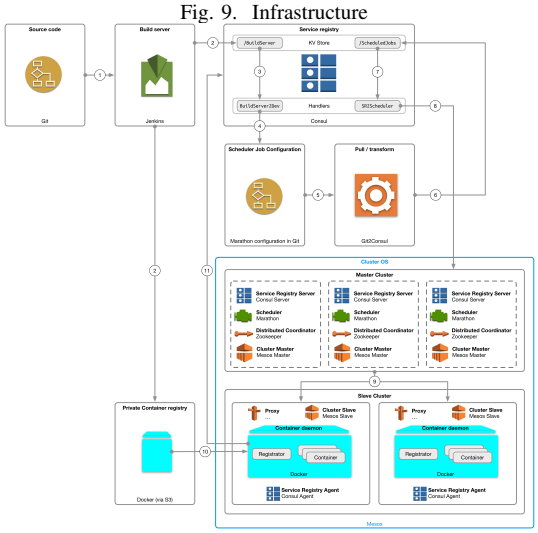
To run the system's server components, a modern cluster management & distributed init system abstracting over a fault-tolerant, self-healing infrastructure is needed. In our case, a highly available Mesos [9] cluster provides abstraction at the datacentre level and acts as a datacentre-wide resource manager. To achieve faster deployments, the microservices that comprise the system are packaged as Docker [12] images.

[2]This ordering code is shared between the ML module and the mobile application. The back- as well as the forward-propagation constructs the input vector from the expanded (or available, in case of the mobile code) sensor data.

The build system (Jenkins [10]) starts by building the Docker images. These images are deployed and managed using a scheduler (Marathon [11]) which acts as the distributed init system and the process manager. Marathon is used for long-running jobs such as the microservices, and the database cluster; Chronos [13] is used to manage the short-lived jobs (e.g. the batching Apache Spark jobs). Having Marathon and Chronos job configurations in git allows for any changes to infrastructure to be versioned, traceable and auditable. Job configuration changes are synchronised to a distributed key-value store (Consul [15]). When a job configuration changes, the Consul handler uses the scheduler API (Marathon REST API) to trigger the infrastructure update through the cluster OS (Mesos). Finally, the cluster OS metrics and the list of running services from the service registry provide the information for the dynamic scaling code. This infrastructure allowed us to implement and maintain a robust and reliable production system. Figure 9 illustrates the outline of our cloud-based infrastructure and highlights the workflow of putting code to production.

There are many more topics that we could not describe here: rolling upgrades without message loss, zero downtime deployment, message versioning, API versioning, message

Fig. 9. Infrastructure



integrity verification, detailed privacy and security concerns, wire formats, cluster consistencies, reliable multi-region and global implementations, test and training data management, model evolution, centralized logging, back pressure reporting and handling, and many more.

IV. Further research

The current system is a great starting point for future work. We are actively pursuing three avenues; we will publish our results as soon as we are ready.

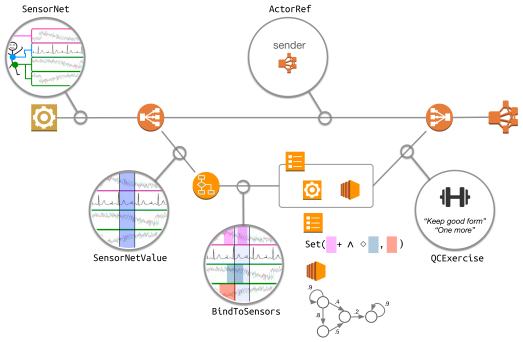
A. Removing self-reporting from activity and food tracking

We are also very excited to use this work as a basis for a Horizon 2020 [17] project proposal; the aim of the Horizon 2020 project is to help as many people as possible to help become more active, while at the same time preventing injury and to assist with weight management. The problems with most weight management programmes center around self-reporting, where the users (sometimes grossly) over-estimate the amount and intensity of exercise performed and (sometimes grossly) under-estimate the amount of food consumed. [**?**]. We can tackle the exercise over-reporting using this system. Understanding what users eat is much more difficult; one of the avenues we are exploring is tracking the amount of food *purchased*. We understand that this is not equal to the amount of food consumed, but we believe we can draw sensible correlations, particularly when comparing large volumes of users, their exercise habits and the details of purchased food. (We are using Mondo [18] for food purchase tracking PoC.)

B. Quality of exercise

One of the approaches we explored for measuring quality of exercise *in laboratory conditions* was to use LDLf [21], where we use a CNN to identify short-duration events in the stream of fused sensor data; we then run LDLf queries over the stream of events, satisfiable queries produce events about the exercise being performed. However, at the moment of writing, we have only implemented this approach in the Akka cluster; the network latency in delivering the sensor data from the sensors through the mobile application to the Akka cluster, evaluating the stream and sending the results back to the mobile is far too great (in the order of units of seconds) to be usable. Figure 10 illustrates the processing pipeline we used.

Fig. 10. LDLf stream processing



The input from the sensors (*SensorNet*) is split into windows (*SensorNetValue*); the sensor data in the windows is classified using CNNs similar to the one use in the current implementation of the mobile application (*BindToSensors*). The *BindToSensors* and the LDLf formulæ then evaluate the stream of events, and together with a state machine which represents particular exercises or exercise regimes, the system can identify sequences of inputs from sensors, which together not only identify, but also express *quality* of exercise being performed. Unfortunately, we were not yet able to implement user-friendly learning mechanism; the classifiers and formulæ are currently hand-crafted. This proves the implementation in principle, but leaves a lot of work for the final product.

C. Innovative UX

Further research will focus on the system's human interaction. Interacting with a screen while exercising greatly reduces the effort the users put in [**?**]; however, listening to music often has the opposite effect [**?**]. Therefore, our future research will explore conversational interfaces delivered through voice synthethiser.

V. Summary

The focus of our work is on guiding the user during his or her exercise or physiotherapy session; to do that, it is necessary to indicate to the user *what to do next* and to recognise that the exercise is actually being performed, and to give feedback about the exercise being performed, as early as possible. In our testing, we found that delays of more than 1 s confuse the users. It was not possible to use the work in [20]; instead, we needed to build our own probabilistic model of next exercise and then eager sensor data recognition layer. It is equally important to accurately collect labelled data, this is achieved through frictionless user interface. The labelled

sensor data, the exercise sequences, and the feedback data is written—together with the matching unstable biometric ID and associated profile—to a Apache Cassandra database in normal tabular form. The Apache Spark jobs then identify clusters of biometric IDs based on the information in the profiles, together with (per cluster) the best sequence of exercises and the best sequence of exercise sessions, the top n user-contributed exercises.

REFERENCES

[1] F. Oo, Q. Uux. Bar. 2016.
[2] Akka.
[3] CQRS/ES.
[4] Apache Cassandra.
[5] Apache Spark.
[6] iBeacon.
[7] iOS advertising identifier.
[8] keras.
[9] Mesos.
[10] Jenkins.
[11] Marathon.
[12] Docker.
[13] Chronos.
[14] Protobuf.
[15] Consul.
[16] DL4J.
[17] European Commission. Horizon 2020. https://ec.europa.eu/programmes/horizon2020/. 28 May 2016.
[18] Mondo.
[19] CUDA.
[20] Morris, D., Saponas, T. S., Guillory, A., & Kelner, I. (2014). RecoFit (pp. 3225–3234). Presented at the the 32nd annual ACM conference, New York, New York, USA: ACM Press. http://doi.org/10.1145/2556288.2557116
[21] LTLf and LDLf Monitoring. Giuseppe De Giacomo, Riccardo De Masellis, Marco Grasso, Fabrizio Maria Maggi and Marco Montali, 2014