The Wearables Development Toolkit: An Integrated Development Environment for Activity Recognition Applications

JUAN HALADJIAN, Technische Universität München

Although the last two decades have seen an increasing number of activity recognition applications with wearable devices, there is still a lack of tools specifically designed to support their development. The development of activity recognition algorithms for wearable devices is particularly challenging because of the several requirements that have to be met simultaneously (e.g., low energy consumption, small and lightweight, accurate recognition). Activity recognition applications are usually developed in a series of iterations to annotate sensor data and to analyze, develop and assess the performance of a recognition algorithm. This paper presents the Wearables Development Toolkit, an Integrated Development Environment designed to lower the entrance barrier to the development of activity recognition applications with wearables. It specifically focuses on activity recognition using on-body inertial sensors. The toolkit offers a repository of high-level reusable components and a set of tools with functionality to annotate data, to analyze and develop activity recognition algorithms and to assess their recognition and computational performance. We demonstrate the versatility of the toolkit with three applications and describe how we developed it incrementally based on two user studies.

CCS Concepts: • **Human-centered computing → Ubiquitous and mobile computing systems and tools**;

Additional Key Words and Phrases: Human Activity Recognition, Wearables, Toolkit, Flow-based programming, Development Environment, Machine Learning

ACM Reference Format:
Juan Haladjian. 2019. The Wearables Development Toolkit: An Integrated Development Environment for Activity Recognition Applications. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 3, 4, Article 134 (December 2019), 26 pages. https://doi.org/10.1145/3369813

1 INTRODUCTION

Over the last two decades, a number of activity recognition applications based on wearable sensors have been introduced, mostly by the research community. Applications areas include *sports* (e.g., table tennis [6], soccer [61], cricket [32]), *health* (e.g., gait analysis of patients of Parkinson's Disease [46], rehabilitation after knee injuries [25]), *daily activity monitoring* (e.g., drinking [52], eating [1], fall detection [8]) and *animal welfare* (e.g., lameness detection in dairy cattle [24], horse jump and gait classification [13]). These applications can help assess, keep track of and improve the physical condition of the wearer unobtrusively, often with minimum setup and independently of the wearer's location.

While the existing applications have already highlighted the potential benefits of activity recognition to different end user groups, developing activity recognition systems that are ultimately accepted by end users remains a challenging task, for several reasons. First, in contrast to other recognition applications (e.g., computer vision, speech recognition), activity recognition applications with wearable devices are bound to additional requirements besides a highly accurate recognition. Common requirements include: low energy consumption

Author's address: Juan Haladjian, juan.haladjian@cs.tum.edu, Technische Universität München, Boltzmanstr. 3, Munich, Germany, 85748.

2474-9567/2019/12-ART134 $15.00
https://doi.org/10.1145/3369813

(i.e., long-lasting battery), small and lightweight device and user comfort (e.g., form-factor, does not heat up) [15]. Second, the design space of a wearable device application is large. Design decisions have to be made regarding the device itself (CPU, memory, sensors, communication and storage modules), the computations that will be run on the device (sensor configurations, signal processing and machine learning methods) and the architecture of the wearable system (e.g., hardware-software mapping involving the wearable, mobile devices, the cloud and the communication between devices). As a consequence, it is often not possible to find a design that meets every requirement, in which case a suitable trade-off between design alternatives has to be made. For example, a particular recognition algorithm might deliver a higher accuracy, but might drain the battery faster than another, less accurate recognition algorithm. Hosting a larger battery could make the device remain functional for a longer period of time, but will usually also increase its size and weight, which might affect user comfort. Furthermore, the entrance barrier to wearable device development remains high, as knowledge in multiple disciplines (e.g., computer science, data science, electrical engineering, human-computer interaction) is often necessary to design wearable systems that meet the user needs.

Due to the aforementioned challenges, developers can rarely make every decision regarding the design of a wearable system upfront. Instead, they usually engage in a series of iterations to assess different design alternatives before they can decide for a suitable one. In particular, they collect and annotate data, they study the collected data and devise, implement and assess different recognition methods. Based on the results of the assessment, they decide whether further iterations are needed. Further iterations might include the collection of new data, or the development, assessment and optimization of recognition methods.

While there exist Integrated Development Environments (IDEs) specifically designed to support the development of other physical devices (e.g., mobile devices), there is up to date no IDE for activity recognition applications with wearable devices. As a consequence, most developers of wearable systems still use general-purpose data analysis tools and programing languages such as Matlab, Python, WEKA and C++. However, as these tools were not designed for activity recognition applications with wearables, they do not directly support the aforementioned tasks and have a high entrance barrier.

In this paper, we present the Wearables Development Toolkit (WDK), a development environment for activity recognition applications with wearable devices. To lower the entrance barrier to the development of activity recognition applications, the WDK offers a set of reusable software components that hide the complexity of algorithms commonly used across activity recognition applications such as signal processing procedures and machine learning classifiers. For developers with less programming experience, the same components are made available within a visual flow-based programming environment. The WDK also facilitates the iterative and incremental design of wearable device systems. In particular, it enables developers to assess the suitability of a particular wearable system (recognition algorithm, hardware and architecture) to the requirements of an application. To this end, it offers four tools to support common development tasks including the annotation of sensor data, the analysis of the data produced by different algorithms, the development of a recognition algorithm and the assessment of the computational performance (CPU usage, memory consumption, amount of data transferred) of a particular wearable system design.

The rest of the paper is structured as follows. Section 2 provides an overview of other toolkits and development environments and discusses how the WDK relates to them. In Section 3, we present the WDK and describe its features, including the goals we based its design on, its architecture and the functionality it offers. Section 4 presents a step-by-step walkthrough describing how the WDK is used to create an activity recognition application. We also demonstrate the versatility of the WDK to re-create two further activity recognition applications in Section 5. In Section 6, we present the results of two user studies we conducted to assess and improve the usability of the WDK.

2 RELATED WORK

Several toolkits have been created that support the development of interactive, ubiquitous and wearable devices and their applications. This section first provides an overview of the toolkits developed so far and then discusses different development methods existing toolkits have relied on to lower the entrance barrier and reduce the time needed to develop applications.

2.1 Toolkits

The toolkits developed so far can be grouped by the kind of applications they support. These include:

- Toolkits that facilitate the 3D-scanning, computer-aided design and 3D printing of objects as well as the integration of electronics into them. These toolkits often offer high-level programming semantics to develop applications that interact with the created objects. Toolkits that fall into this category include: Pineal [34], Retrofab [48], Makers Marks [51], Sauron [50], Modkit [40].
- Toolkits that support the development of applications that sense information from a physical environment (e.g., room temperature) and/or users in the environment (e.g., their posture) and enable joint interactions between them. Some of the toolkits under this category are: EagleSense [58], Physikit [29], Sod-toolkit [54].
- Toolkits that enable the creation of applications distributed across multiple wearable, mobile or ubiquitous devices. These toolkits offer programming semantics that span across multiple devices; hence, they save users from having to program each device as well as the communication protocols between them. Toolkits under this category include: Interactex [20], Panelrama [59], XDStudio [42], Weave [9], WatchConnect [30], iStuff Mobile [3], ToyVision [39] and C4 [33].
- Toolkits that lower the entrance barrier to the development of applications that rely on specific hardware technologies such as: smart textiles, [20], printed circuit boards [56], electrical muscle stimulation devices [47], capacitive sensors [19] and paper-based electronics [49]. These toolkits offer a set of reusable hardware and software components with high-level programming semantics that hide low-level implementation details about the particular technology.

Most of these toolkits were not developed for activity recognition applications with wearables; hence, they target different kinds of applications than the WDK. A more related class of toolkits corresponds to those designed to lower the entrance barrier to the development of applications that react to user gestures. Toolkits under this category include: Exemplar [27], MAGIC [2], GART [38] and (GT^2k) [57]. These toolkits are similar to our work in that they facilitate the creation of applications that detect specific patterns in sensor data. However, they focus on gesture recognition and offer predefined recognition methods for this purpose: Exemplar [27] uses Dynamic Time Warping (DTW), the MAGIC toolkit [2] relies on DTW together with a set of predefined features extracted from the input data, the (GT^2k) [57] uses a Hidden Markov Model (HMM) configured with a grammar specified by the user and the GART toolkit [38] relies only on a HMM. The WDK makes a broader set of recognition methods available to enable developers to experiment and ultimately design a recognition algorithm that fulfills the requirements of the particular application.

The CRN Toolbox [4] and the more recent Gesture Recognition Toolkit (GRT) developed by Nick Gillian [16] are perhaps the existing toolkits which share most similarity with the WDK. Both toolkits enable the development of recognition algorithms with a set of reusable software components. While these toolkits ease the implementation (i.e., programming) and assessment of an activity recognition algorithm, they do not support the rest of the development lifecycle of a recognition algorithm. For example, these tools don't facilitate the annotation of data, its analysis and don't provide a detailed assessment of the performance of an algorithm besides an aggregated metric (e.g., F1-Score). This is an issue because developers rarely know upfront what algorithm to implement without annotating and studying the data, developing, assessing and optimizing different recognition algorithms.

The WDK consists of different tools integrated within a development environment to support these tasks and facilitate the iterative development of activity recognition algorithms for wearables.

2.2 Programming Semantics

The existing toolkits lower the entrance barrier to users with high-level programming semantics. We have identified four main programming paradigms used by most toolkits. In *programming by demonstration*, the toolkit learns from demonstrations performed by users. Since this technique saves users from having to write code, it has been used in several toolkits from the human-computer interaction community such as: a CAPpella [11], Exemplar [27], Topiary [37], d.tools [28] and PaperPulse [49]. The ease to program an application using this technique often comes at the cost of a lack of flexibility to define custom behaviors and performance limitations. Since this paradigm relies on predefined recognition methods, users can often not optimize the recognition algorithms to their applications.

In *rule-based programming*, the user defines which predefined behaviors should be executed upon the occurrence of predefined events. Depending on the variety of events and behaviors available in the toolkit, relatively complex programs can be created with this technique by *connecting* events to behaviors that might themselves trigger other events. Toolkits that rely on this method include: Phidgets [17], Calder [36], Intuino [55], Amarino [31].

Flow-based programming is a popular visual programming approach where functionality commonly used in a particular domain is modularized in so-called *nodes*. Nodes are visual representations that encapsulate functionality and can be manipulated over a graphical user interface. A flow-based program looks like a directed graph; users draw arrows between nodes to define the order of execution of the nodes as well as the flow of data between them. Several toolkits have taken advantage of this programming technique in the past, including iStuff Mobile [3], the CRN Toolbox [4] and Interactex [20].

In *block-based programming*, different programming constructs (for loops, if-conditions, variables) are represented visually in the form of blocks that can be otherwise used as in conventional programs. While the visual representations of blocks make it easier to understand the syntax of a program (e.g., to understand the scope of a for loop), users still need to be able to create programs using conventional programming constructs. Toolkits that feature *block-based programming* include: Modkit [40] and AppInventor[1].

Previously developed toolkits have also relied on *text-based programming* approaches, including scripting and domain-specific languages. For example, the Weave toolkit [9] provides high-level APIs in Javascript for rapid prototyping wearable device applications and C4 [33] is a script language with APIs to manipulate and animate media objects such as images and movies in mobile device applications.

Since the *programming by demonstration* approach hides the recognition algorithm from developers, it also takes away the opportunity for developers to optimize it, which we considered important due to the limited computational resources available in a wearable device. Hence, we decided against it. We also thought that a *rule-based programming* paradigm would not provide developers enough freedom to create and optimize activity recognition algorithms. Furthermore, we considered that a *block-based programming* paradigm would make sense for relatively simple programs developed for educational purposes, but not for activity recognition applications. Since activity recognition applications often rely on similar functionality (signal processing and feature extraction algorithms), we opted to encapsulate this functionality under a uniform interface and made it available for reuse within a flow-based programming environment. However, we thought that a visual programming approach alone would be prone to scalability issues when developing complex recognition algorithms with several feature extraction methods. Therefore, we decided to offer the same functionality within a text-based programming language.

[1]http://appinventor.mit.edu

3 THE WEARABLES DEVELOPMENT TOOLKIT

The WDK consists of a library of reusable software components and a set of tools built on top of them. This section first discusses the goals we aimed for in the design of the WDK and then describes the reusable components, tools and main features in the WDK. The WDK is implemented in Matlab and is open source[2] under MIT license.

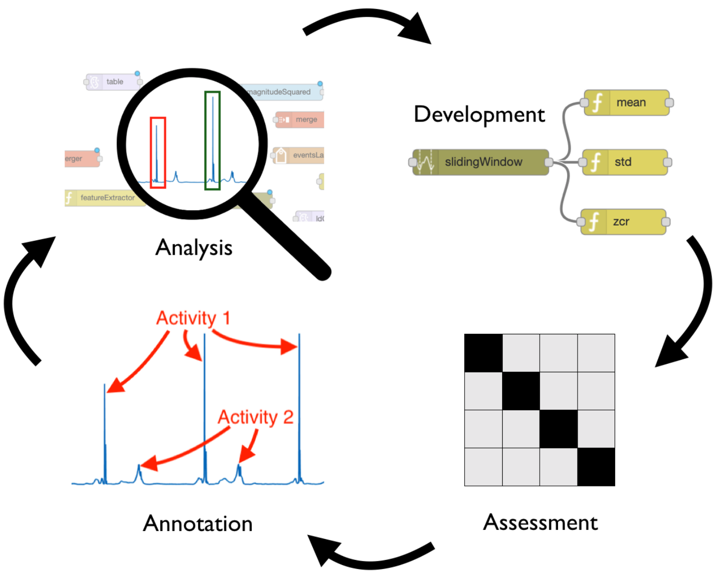

Fig. 1. Typical development lifecycle of an activity recognition algorithm. Developers usually engage in a series of iterations to collect and annotate a data set, study the collected data and then develop one or more recognition algorithms, assess and optimize their performance until the requirements of the application are met.

3.1 Design Goals

We designed the WDK based on the following design goals:

Low entrance barrier. The development of a wearable system requires knowledge from multiple disciplines including data analysis, signal processing, pattern recognition and embedded firmware development. Hence, their entrance barrier is still high. However, many wearable systems rely on similar functionality (e.g., feature extraction methods) and are developed in a similar way, as illustrated by Figure 1. A main goal of the WDK is to provide a simple way for developers to reuse common functionality as well as to ease the development tasks.

Extensibility. Even if the most common functionality used across activity recognition applications was made available for reuse within a toolkit, developers are likely to need new functionality for their particular applications, such as custom feature extraction algorithms. For this reason, one goal in the design of the WDK was to enable its set of available functionality to be extensible by developers with little effort.

Assessment of the computational requirements. Activity recognition applications are usually constrained by the computational capabilities of the wearable device. In order to study the suitability of a recognition algorithm to a particular wearable device, developers need to assess its computational requirements (CPU speed, memory capacity, battery duration). As data analysis tools used to develop wearable device applications not always provide an insight into the computational requirements of an algorithm, these are often estimated once the algorithm is

[2]https://github.com/avenix/WDK

ported to the target device. A goal in the design of the WDK was to aid developers with an early estimation of the computational requirements of a recognition algorithm.

Recognition insight. Most activity recognition applications with wearable sensors extract patterns in a stream of sensor data using machine learning classifiers. While existing tools provide aggregated metrics describing the performance of a classifier (e.g., accuracy, F1-Score) they don't provide further insight to aid developers find issues in the recognition algorithm. A goal we pursued in the design of the WDK was to enable developers to spot issues in a recognition with a frame-by-frame comparison between the ground truth and the recognition results.

Quick assessment. The training and performance assessment of a recognition algorithm is usually a computationally intensive task. As a consequence, the iterative process to develop, optimize and assess the performance of an activity recognition algorithm could be hindered by long algorithm execution times. Therefore, in the design of the WDK we aimed for solutions to quickly execute and assess recognition algorithms.

3.2 Architecture

Figure 2 illustrates the architecture of the WDK. The WDK is based on a repository architectural style. The repository consists of a set of reusable components organized as a layered architecture on top of the Matlab runtime environment. The middle layer of the repository contains the *runtime components*, a set of procedures executed by activity recognition algorithms, whereas the top layer contains functionality to facilitate the development of such algorithms. The different tools in the WDK create, make changes to, simulate and assess the performance of activity recognition algorithms using the abstractions in the repository. Applications running on wearable devices rely on the *runtime components* to execute the activity recognition algorithms created with the WDK.

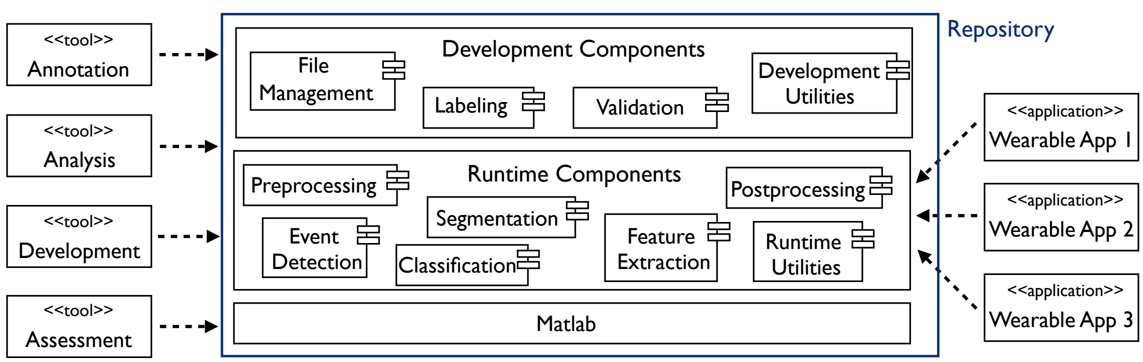

Fig. 2. The WDK is based on a repository architecture. The different tools in the WDK use the repository to create activity recognition algorithms which are executed by applications running on a wearable device.

The main design goal that drove our decision for this architecture was the *extensibility* goal. The repository architecture decouples the reusable components from the tools, enabling the components to be reused independently of the tools and the tools to be extended without changes to the reusable components. It also decouples the different tools from each other, as they interact only indirectly through the repository. This facilitates extending each tool without affecting the other tools. In addition, decoupling the *runtime components* from the rest of the toolkit eases their reuse by the wearable applications.

The decision to base the WDK on Matlab was mainly driven by the *low entrance barrier* goal. Matlab facilitates data analysis tasks with a broad set of functionality and native language semantics to perform arithmetic, statistical and signal processing operations on multi-dimensional arrays of data. This functionality can be used in combination with the set of reusable components in the WDK to manipulate and process data. Another alternative would have been Python in combination with third-party libraries such as *NumPy*, *Matplotlib*, *TensorFlow* and *Keras*.

3.3 Reusable Components

Other toolkits have lowered the entrance barrier to the development of different applications by hiding implementation details behind high-level components. Similarly, the WDK provides a set of high-level reusable components with functionality commonly used across activity recognition applications. To reuse a component, developers don't have to understand its implementation, but only what it does and what data types it requires and produces.

Table 1. Summary of the functionality in the WDK's repository. *Signal*s are two-dimensional arrays of floating-point values. *Event*s represent a specific sample in a *Signal* and store an integer timestamp and a floating-point value. *Segment*s represent a range of samples in a *Signal* and contain a two-dimensional array of floating-point values and the start and end indices in the original *Signal*. *FeaturesTable*s are two-dimensional arrays of floating-point features and an 8-bit integer *label* column. *ClassificationResult*s are arrays of 8-bit integer labels predicted by a machine learning classifier.

	Component Type	Input	Output	Used to...
Runtime	Preprocessing	*Signal*	*Signal*	transform a *Signal* and prepare it for further processing
	Event Detection	*Signal*	*Events*	detect the occurrence of specific events (e.g., peaks) in a *Signal*
	Segmentation	*Signal* / *Events*	*Segments*	divide a *Signal* into regions of interest
	Feature Extraction	*Segments*	*FeaturesTable*	compute time or frequency-domain features of a *Signal*
	Classification	*FeaturesTable*	*ClassificationResult*	predict a label for each feature vector in a *FeaturesTable*
	Postprocessing	*ClassificationResult*	*ClassificationResult*	add, remove or alter labels in a sequence of predicted labels
	Utilities	multiple	multiple	split, merge or transform data (e.g., extract values from a *Signal*)
Development	File Management	N/A	multiple	load and parse a data file or an annotations file
	Labeling	*Segments*	*Segments*	assign *Labels* to *Event*s or *Segment*s using an annotations file
	Validation	*FeaturesTable*	*ClassificationResult*	train and evaluate a machine learning classifier
	Utilities	multiple	multiple	different functions used at development time (e.g., feature selection)

Table 1 provides a summary of the reusable components in the WDK and the data types they take as input and produce as output. The *runtime components* encapsulate methods for each *stage* of the Activity Recognition Chain [7], including: preprocessing, event detection, segmentation, feature extraction, classification and post-processing. The *development components* offer functionality needed to manipulate the data used by the *runtime components*, label the segments produced by a segmentation algorithm and validate machine learning classifiers. Most of the preprocessing, feature extraction, classification and validation algorithms are standard off-the-shelf methods commonly used in activity recognition and offered by Matlab. In contrast, most of the event detection, segmentation, labeling and postprocessing components are our own implementations of less common algorithms described in different scientific papers [6, 7, 10, 14, 45] or derived from our own previous work. A full list of the components offered by the WDK until the date of submission of this article is available in the Appendix.

The set of reusable components is designed as a modular object-oriented architecture based on a pipes and filter architectural style. The reusable components act as filters: they receive data, process it and pass it over to other components. Developers create recognition algorithms by instantiating components and connecting them together. An algorithm is represented as a directed graph and executed with a stack in a depth-first order. To extend the functionality available in the repository, developers only have to subclass the *Computer* class and implement its *compute* method.

3.4 Tools

The WDK offers four tools to support the main tasks in the development lifecycle of an activity recognition application: the *Annotation* tool is used to annotate a time series data set, the *Analysis* tool provides a way to study the data and segments produced by a segmentation algorithm, the *Development* tool enables the creation of activity recognition algorithms with the set of components and the *Assessment* tool is used to evaluate the runtime performance of a recognition algorithm.

The *Annotation* tool is used to add annotations to a multi-dimensional time series signal. The tool supports two kinds of annotations: *event annotations* and *range annotations*. *Event annotations* correspond to events that occur at specific moments in time (i.e., a single timestamp) and *range annotations* correspond to activities that have a duration in time (i.e., two timestamps indicating start and an end of the activity). Both annotation types can be used simultaneously.

Fig. 3. The *Annotation* tool displays the squared magnitude of the accelerometer signal collected by a motion sensor attached to a cow. The individual strides of the cow have been annotated as *event annotations* (red) and the walking and running activities as *range annotations* (black rectangles).

Video is commonly used as a reference to annotate collected wearable sensor data. The *Annotation* tool displays video and data next to each other and automatically updates the current video frame to the current data selection and vice-versa. Users synchronize video and data once by providing two video frames and two data timestamps which correspond to the same event. In addition, external markers can be displayed on top of the data when annotations are performed in real time (i.e., during the data collection) or using external video annotation software.

The *Analysis* tool provides insight into the behavior of an activity recognition algorithm by displaying the segments produced by it. To this end, developers design an activity recognition algorithm either directly over the user interface of the *Analysis* tool or by importing it from the *Development* tool. The segments produced by the algorithm are then labeled, grouped by activity and shown to the user. A visualization strategy can display the segments next to each other, as shown in Figure 4, or on top of each other to help spot the pattern or signature of a particular activity. A particular kind of recognition algorithm generates segments from the annotations, which can be helpful to review the annotations and to gain insight into the patterns the algorithm should recognize.

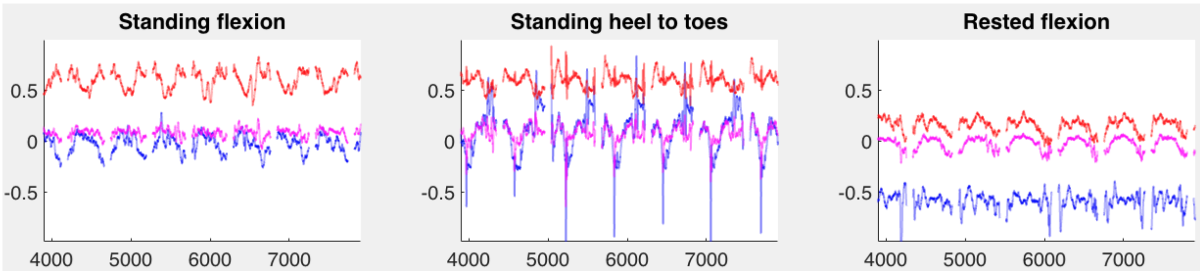

Fig. 4. The *Analysis* tool shows segments produced by a recognition algorithm corresponding to different physical rehabilitation exercises performed by patients after a hip replacement surgery.

The *Development* tool is a visual programming interface to enable less experienced users to create applications by reusing the components in the WDK. To this end, we extended Node-RED, a popular flow-based programming platform with a Javascript implementation of each reusable component. This implementation is available in a separate open source repository[3]. Algorithms created in Node-RED can be imported and executed in the different tools of the WDK. Figure 5 shows a simple activity recognition algorithm developed with the *Development* tool.

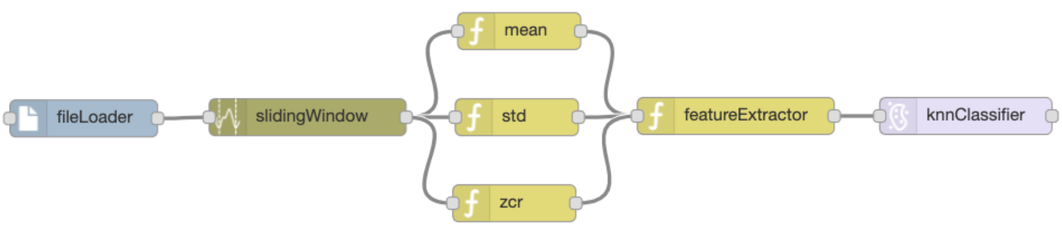

Fig. 5. Activity recognition algorithm developed in the *Development* tool. The algorithm generates consecutive segments of a one-dimensional signal using a *SlidingWindow*. For each segment, it extracts the mean, standard deviation and zero-crossing rate features. The *featureExtractor* groups the three features into a *FeaturesTable*, which is passed as input to a KNN classifier.

The *Assessment* tool enables the assessment of activity recognition algorithms regarding their *recognition* and *computational* performance. To this end, the WDK simulates the execution of an activity recognition algorithm and computes different metrics. The *recognition* performance of an algorithm is quantified by the following metrics: *accuracy*, *precision*, *recall*, *F1-Score* and *confusion matrix*. These metrics are calculated per data file and activity (i.e., class). The *computational* performance is quantified with three cost metrics: *execution*, *memory* and *communication costs*. To enable developers to compare different architectures of their wearable systems, the WDK estimates these metrics for each stage of a recognition algorithm. These metrics are averaged across data files and displayed over the user interface for each algorithm execution. Next subsection describes how the *computational* performance metrics are computed.

[3]https://github.com/avenix/WDK-RED

3.5 Computational Performance Assessment

Every reusable component in the WDK computes three computational performance metrics: *execution*, *memory* and *communication cost*. The *execution cost* is an estimation of the number of floating point operations performed by the recognition algorithm normalized by the amount of data samples provided as input. The *memory cost* is an estimation of the maximum amount of memory required to execute a recognition algorithm. *Execution* and *memory cost*s are calculated at runtime by executing a recognition algorithm. Each reusable component computes its *execution* and *memory cost*s for a provided input based on the values of its properties at runtime. The *execution cost* of an algorithm is then calculated by adding the *execution cost*s returned by each reusable component every time their *compute* method is invoked. The *memory cost* is calculated by adding the *memory cost* returned by each component in a recognition algorithm once. The *communication cost* of an algorithm is computed by adding up the amount of bytes produced by the last component in the algorithm. The *execution*, *memory* and *communication cost*s of each reusable component are listed in Section A in the Appendix.

3.6 Frame-by-frame Analysis

Many activity recognition applications are evaluated with respect to time [7]. To provide further insight into the recognition performance of an algorithm with respect to time, the *Assessment* tool displays a frame-by-frame comparison between the ground truth and the classifier's prediction on top of the raw data and reference video. To this end, the WDK stores the list of labels predicted by a classification algorithm, feature vectors extracted by a feature extraction algorithm, segments generated by a segmentation algorithm and signals produced by a preprocessing algorithm. The start and end index of a segment are used to correlate predicted labels to original annotations in the ground truth.

3.7 Cache

To enable the quick assessment of a recognition algorithm, the WDK stores the execution results of a recognition algorithm in a cache under a hash-key that uniquely identifies the algorithm. This key is generated by concatenating a description of each component used in the algorithm in depth-first order. The description of a component is a string containing its name and the value assigned to each of its properties. Before executing a particular algorithm, the WDK loads its execution results, in case these are available in the cache.

4 WALKTHROUGH: GOALIEGLOVE

This section describes step-by-step how to develop an algorithm to recognize the training exercises performed by soccer goalkeepers including dives, catches and throws with an inertial sensor inserted into goalkeepers' gloves. The goal of this application is to give goalkeepers personalized feedback about their training.

4.1 Data Collection

This application uses a data set collected from 7 goalkeepers during their training sessions using a sensor device based on the ICM20948 9-axis Inertial Measurement Unit. To capture the full range of motion of exercises that might contain high intensity impacts and rotations, the accelerometer, gyroscope and compass were set to their maximum ranges: ±16 g, ±2000 dps and ±4900 μT respectively, as done in similar IMU-based sports applications [6, 18, 53]. The sensor device collected data at 200 Hz and normalized it to the range $[-1, 1]$. Each training session lasted an average of 33 minutes. The training sessions were recorded on video for annotation purposes. On average, each video and data file in binary format had a size of 2.22 GB and 18.25 MB, respectively. To synchronize the data and video, goalkeepers were asked to applaud three times in front of the camera between exercise sets.

4.2 Data Annotation

This application aims at detecting sporadic events that have a high energy of motion. Previous work has detected similar events by finding peaks on the (squared) magnitude of acceleration or gyroscope signals [6, 18, 23]. In order to be able to assess the performance of a peak detection algorithm later on, we add an *event annotation* to each peak in the magnitude of acceleration that corresponds to an exercise repetition. We also annotate other motions with high accelerations performed often by goalkeepers such as ball passes and bouncing the ball on the ground. Annotating these motions will enable us to train a classifier to filter these motions out in case they are detected. The ground truth contains 4153 annotated motions, out of which 916 correspond to relevant exercise repetitions and 3237 are instances of irrelevant motions.

4.3 Analysis

Most relevant exercises have a high intensity of acceleration. We know that high intensity accelerations can be detected using a peak detector. Therefore, we use the *Analysis* tool to study the signal to determine how to create segments of data around the peaks detected by a peak detector. Figure 6 shows the data around the relevant events we annotated. We observe that the characteristic motion of most exercises starts approximately 200 samples before the peak and that most exercises end shortly after it. Furthermore, we observe that the relevant motion previous to the peak might last longer than a second (200 samples) in some exercises such as the dives. However, extending the segments to more than 200 samples before the peak would cause motion to be included in the segment that is not characteristic of most exercises. Furthermore, longer segments increase the amount of memory required by the device. Based on these observations, we decide to segment the signal according to: $[p - 200, p + 30]$.

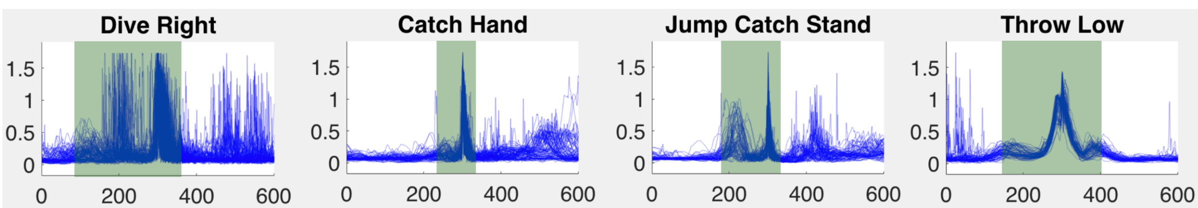

Fig. 6. The *Analysis* tool displays the magnitude of acceleration of segments corresponding to different exercises plotted on top of each other and grouped by their label. We marked the parts of the signal that contain motion characteristic of each exercise with a green overlay. We used this visualization to devise an event detection and segmentation algorithm.

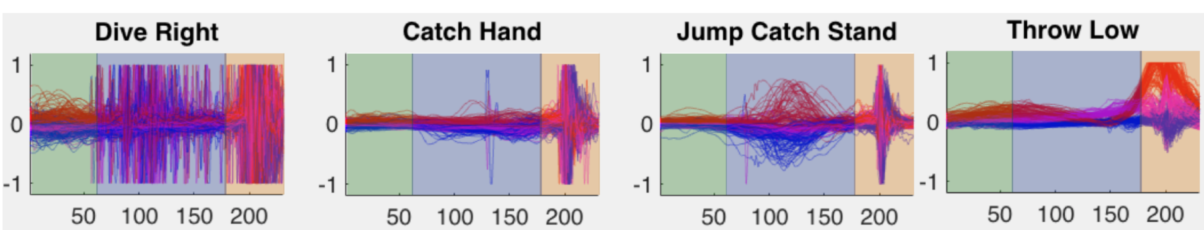

Fig. 7. The exercises performed by soccer goalkeepers can be divided in three sub-segments. The range $[181, 230]$ (orange overlay) corresponds to a ball or ground contact. The range $[61, 180]$ (blue overlay) can provide information to determine whether the exercise is a dive or a jump, as these exercises present more acceleration in this range than the other ones. The range $[1 - 60]$ (green overlay) can be used to recognize dives due to the arm swing goalkeepers perform before a dive.

To decide what features should the algorithm extract for each segment, we study the characteristic motions of the different exercises in the *Analysis* tool, as shown in Figure 7. Most exercises consist of sequences of motions. For example, the *Jump Catch Stand* consists of a jump, a ball catch in the air and a ground contact. Based on this analysis, we decide to divide segments in three sub-segments: $[1, 60]$, $[61, 180]$ and $[181, 230]$ and extract a total of 45 time-domain features including: *Min, Max, Mean, Median, Variance, STD* and *AUC* computed on different axes of the accelerometer and magnetometer signals for each sub-segment.

4.4 Development

Next, we develop the algorithm shown in Listing 1 in a Matlab script. The algorithm first selects all three accelerometer axes in the input *Signal* using the *AxisSelector* and passes the resulting Nx3 *Signal* to the *Magnitude*. The *Magnitude* computes the magnitude of each accelerometer vector in the input *Signal* and passes the computed magnitude in an Nx1 *Signal* to the *SimplePeakDetector*. The *SimplePeakDetector* detects peaks in the magnitude *Signal* and returns the *Event*s of the detected peaks. The *EventSegmentation* generates *Segment*s around the detected peaks by extracting the 200 samples to the left of the detected peak and 30 samples to its right. The *Segment*s are passed to a *FeatureExtractor* (loaded from the *features.mat* file), which extracts the 45 features mentioned in the previous subsection for each *Segment* and outputs a *FeaturesTable*. The *FeatureNormalizer* normalizes the *FeaturesTable* so that each of its feature columns has zero mean and a standard deviation of 1 and passes it to the *SVMClassifier*. The *SVMClassifier* returns the predicted labels in a *ClassificationResult* object.

```
%computes magnitude of acceleration
axisSelector = AxisSelector(1:3);
magnitude = Magnitude();

%minPeakHeight=0.8, minPeakDist=100
peakDetector = SimplePeakDetector(0.8,100);

%segments in the range: [p-200,p+30]
segmentation = EventSegmentation(200,30);

%loads feature extraction algorithm
featureExtractor =
    DataLoader.LoadComputer('features.mat');
```

```
%computes normalization values and normalizes
featureNormalizer = FeatureNormalizer();
featureNormalizer.fit(trainTable);
featureNormalizer.normalize(trainTable);

%order=1, boxConstraint=1.0
classifier = SVMClassifier(1,1);
classifier.train(trainTable);

components = {axisSelector, magnitude, peakDetector,
    segmentation, featureExtractor,
    featureNormalizer, classifier};
algorithm =
    Computer.ComputerWithSequence(components);
```

Listing 1. Algorithm to detect and classify soccer goalkeeper training exercises. The algorithm starts at the left and continues at the right column. The *trainTable* variable in the right column has been generated with a similar sequence of computations, excluding the *featureNormalizer* and *classifier* components and using an *EventSegmentsLabeler* after the segmentation.

4.5 Performance Assessment

After having developed the algorithm, we use the *Assessment* tool to assess and optimize its recognition performance. We test different values for the properties *minPeakHeight* and *minPeakDistance* of the *SimplePeakDetector*. Low values for these parameters might cause more irrelevant motions to be detected whereas high values might cause relevant exercises to be missed. We use the frame-by-frame analysis to understand the effects of different values for these parameters on the data set, as shown in Figure 8. After this analysis, we decide for the values: $minPeakHeight = 0.8$ and $minPeakDistance = 100$. The *SVMClassifier* component configured as: ($order = 1$ and *boxConstraint* = 1.0) achieves the highest performance with an accuracy of 81.8%, a precision of 81.4% and a recall of 79.8%. Adapting the previous script to select subsets of features with the *FeatureSelector* component reveals that up to 5 features can be excluded with a minimal drop in accuracy.

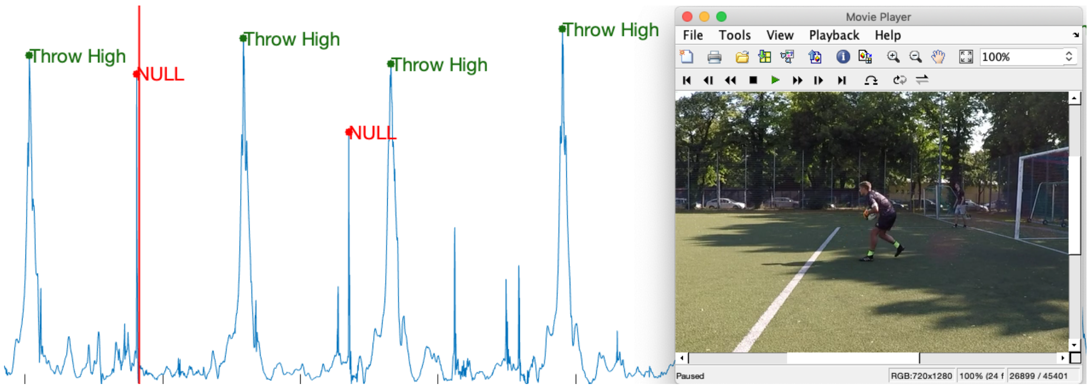

Fig. 8. The frame-by-frame analysis displays the results of a recognition algorithm on top of the magnitude of acceleration. The algorithm detected four exercises (shown in green) and two irrelevant motions (shown in red). After this goalkeeper performs a throw, the ball is passed back at him with high intensity, which is detected (as a false positive) by the algorithm.

The *Assessment* tool provides an overview of the computational performance of different architectures to run this algorithm. If only the segmentation was done on the wearable device, 657.7 KB of data would have to be transferred from the wearable device for an average training session. This is calculated by the WDK as an average of 244 segments per training session with a size of 230x6 values each and using 2 bytes per value. If the feature extraction was also performed on the wearable device, only 38,1 KB of data would be produced on average per training (244 feature vectors with 40 features represented with 4 bytes each). Finally, if the classification was also performed on the wearable device, only 2.1 KB of data would be generated (244 1-byte labels and an 8-byte timestamp). The *Assessment* tool estimates a *memory cost* of 2.7 KB for the event detection, segmentation and feature extraction stages - most of which corresponds to the *EventSegmentation* component which allocates a matrix of 230x6 cells of 2 bytes per value.

5 REFERENCE APPLICATIONS

Ledo et al. [35] proposed four types of ways to evaluate toolkits: demonstration, usage, technical performance and heuristics. Demonstration evaluations show how a toolkit is used to create applications. Usage evaluations investigate the usability of a toolkit, often by means of user studies. Technical performance evaluations assess the non-functional requirements of a toolkit such as the recognition accuracy of a created algorithm. A heuristics evaluation investigates a toolkit's usability with respect to a set of heuristics, such as Nielsen's usability heuristics [43, 44]. The previous section demonstrated the usage of the WDK with a step-by-step walkthrough to create an application. This section demonstrates the WDK's versatility to support different applications.

We created the WDK iteratively by extending and refining its abstractions to replicate different activity recognition applications from the literature and from our previous work. These applications include: an algorithm to classify daily activities presented by Bao and Intille [5], a smart bandage to track the rehabilitation progress of patients after a knee injury [21, 25, 26], a chest belt strap band to recognize basketball defensive training exercises, a lameness detection system for dairy cattle [23, 24], an activity tracker for pigs [22] and a sensor-based horse gait and jump detection system for show jumping applications [12, 13]. Next, we demonstrate how two of these applications are developed using the reusable components in the WDK.

5.1 Daily Activity Monitoring

Listing 2 replicates the activity recognition algorithm presented by Bao and Intille [5]. This algorithm recognizes physical activities (e.g., walking, sitting, eating) using two-axis accelerometers worn on different parts of the body. The algorithm processes a stream of sensor values in *Segment*s of 512 samples with 50% overlapping using the *SlidingWindowSegmentation*. The *SlidingWindowSegmentation* passes *Segment*s of 512x2 samples to a *FeatureExtractor*. The *FeatureExtractor* computes a feature vector for each segment it receives as input and appends it to a *FeaturesTable*. Each feature vector contains the mean, spectral entropy and spectral energy of both accelerometer axes and the correlation between the two axes. *FeaturesTable*s output by the *FeatureExtractor* are passed to the *TreeClassifier*, which returns an array of labels in a *ClassificationResult*.

```
%segmentSize=512, 50% overlapping
slidingWindow =
    SlidingWindowSegmentation(512,256);

%creates feature extraction algorithm
featureExtractor =
    createFeatureExtractor();

%maxNumSplits=30
classifier = TreeClassifier(30);

%creates algorithm
algorithm =
   Computer.ComputerWithSequence({
   slidingWindow, featureExtractor,
   classifier});
```

```
function featureExtractor = createFeatureExtractor()
  fftFeatures = FFT();
  fftFeatures.addNextComputers({SpectralEntropy(),
      SpectralEnergy()});
  featureComputers = {Mean(),fftFeatures};

  %extract features on accelerometer axes x and y
  axis1 = AxisSelector(1);%x-axis
  axis2 = AxisSelector(2);%y-axis
  axis1.addNextComputers(featureComputers);
  axis2.addNextComputers(featureComputers);

  %returns feature extraction algorithm
  featureExtractor = FeatureExtractor({axis1,axis2,
      Correlation()});
end
```

Listing 2. Algorithm to classify daily activities proposed by Bao and Intille [5] reproduced with the WDK's components.

5.2 Hip Rehabilitation App

The Hip Rehabilitation App (HipRApp) is a wearable strap band to track the rehabilitation progress of patients who underwent a hip replacement surgery. It counts the amount of exercise repetitions and walking steps performed by patients during a training session. The algorithm shown in Listing 3 recognizes exercise repetitions in a stream of samples produced by a 6-axis inertial sensor (accelerometer and gyroscope) worn by patients at the ankle. First, the *AxisSelector* extracts the accelerometer axes from the input data into an Nx3 *Signal*. The *LowPassFilter* applies a Butterworth low-pass filter to each of the *Signal*'s columns to eliminate high-frequency noise in the accelerometer signal. The filtered data is processed using a sliding window. For each *Segment* produced by the *SlidingWindowSegmentation*, the *Min*, *Max*, *Mean*, *Median*, *Variance*, *STD*, *AUC*, *AAV*, *MAD*, *IQR*, *RMS*, *Skewness* and *Kurtosis* are computed. These features are extracted on every axis of the accelerometer and gyroscope *Signals* and aggregated by the *FeatureExtractor* into a *FeaturesTable*. The *FeatureNormalizer* normalizes *FeaturesTable*s and passes them to the *KNNClassifier*. The *KNNClassifier* predicts a label for each row in a *FeaturesTable* and returns a *ClassificationResult* containing an array of predicted labels. Finally, the *SlidingWindowMaxLabelSelector* post-processing component replaces every label at index *labelIndex* in the array of predicted labels with the most frequent label in the range $[labelIndex - 3, labelIndex + 3]$, or with the NULL-class if no label occurs at least 4 times in the range. This is done to '*favor*' the most frequent label within a 6-label window and avoid the sporadic misclassification of unrelated exercises or instances of the NULL-class. This increases the recognition accuracy due to the fact that patients usually perform 10 to 20 repetitions of an exercise in a row.

```
%select signals 1,2,3 (accelerometer x,y,z)
axisSelector = AxisSelector(1:3);

%order=1, cutoff=20Hz
lowPassFilter = LowPassFilter(1,20);

%segmentSize=488, 50% overlapping
segmentation =
    SlidingWindowSegmentation(488,244);

%max, min, etc. on signals 1,2,3,4,5 and 6
features =
    FeatureExtractor.DefaultFeatures();
featureExtractor =
    FeatureExtractor(features,1:6);
```

```
%computes normalization values
featureNormalizer = FeatureNormalizer();

%k=10, distanceMetric='euclidean'
classifier = KNNClassifier(10,'euclidean');

%windowSize=6, minimumCount=4
postprocessor = LabelSlidingWindowMaxSelector(6,4);

%creates algorithm
components = {axisSelector, lowPassFilter,
    segmentation, featureExtractor,
    featureNormalizer, classifier, postprocessor};
algorithm =
    Computer.ComputerWithSequence(components);
```

Listing 3. Algorithm to classify rehabilitation exercises performed by patients of hip replacement. The algorithm starts in the left column and continues in the right. In a separate script, the *classifier* is trained and the *featureNormalizer* is fit with normalization values.

5.3 Discussion

The incremental development process we used to create the WDK enabled us to assess its coverage of the functionality present in a variety of applications and to refine it accordingly. The applications presented in this section demonstrate the WDK's versatility to different domains and illustrate that complex activity recognition algorithms can be created with a few components in the WDK.

6 USABILITY EVALUATION

To study the usability of the WDK, we conducted a user study with three participants who used the WDK to create different applications, as summarized in Table 2. The participants were students of computer science at the Technical University of Munich who contacted us to write a bachelor's or master's thesis at our department after they read a project description on our department's website. None of them had previous experience in activity recognition or in Matlab. They were instructed to develop an application using the WDK during a period of two to four months. After the development phase, we conducted an semi-structured interview where the participants described their experiences using the WDK and demonstrated to us how they had used it.

Table 2. Participants of the first user study and applications they developed using the WDK.

Participant	Gender	Application
P1	Male	GoalieGlove
P2	Male	Recognition of basketball defensive training exercises
P3	Female	HipRApp

All three participants found the functionality to annotate data while looking at the video useful. P3 said: *"The Annotation tool is very useful because everything is in the same place. I was using DaVinci Resolve for the annotations in the video but that was a lot of back and forth switching"*[4]. The participants also praised the functionality to compare the segments produced by an algorithm in the *Analysis* tool. In particular, they welcomed the functionality to quickly switch between signals to design feature extraction [P2,P3] and event detection algorithms [P1,P2]. P1 said: *"It's good to compare different players: what segments are too small and which ones are too big"*. Furthermore, every participant reported that they found the frame-by-frame comparison in the *Assessment* tool useful in their projects. Notably, P2 mentioned that he had been using wrongly annotated data for months until he observed a contiguous sequence of misclassified exercises in the frame-by-frame comparison. He described the insights he gained as: *"if we go frame-by-frame, then we can see that longer strides have a longer intensity and that the player is lean forward a bit more. That explains that instance A was detected and not instance B"*. Furthermore, P2 and P3 mentioned that they could save time by reusing functionality available in the WDK. P3 said: *"I had to implement a lot of machine learning algorithms in Python. Here you can reuse a lot of functionality"*.

While using the WDK, the participants also mentioned different issues, bugs and feature requests. Two main issues they mentioned were the difficulty to identify the root of an error in a recognition algorithm they had developed and the difficulty to understand some of the reusable components in the WDK. Errors when executing a recognition algorithm were caused when two reusable components were connected to each other, although the data type produced by the predecessor component was not compatible with the input type required by the successor component. P1 said: *"If something fails, you don't know what went wrong"*. Furthermore, when executing an invalid algorithm, Matlab displays an error message containing the execution stack trace. Although the first line in the stack trace contained the name of the reusable component that caused the failure, the participants did not find this information helpful to identify the root of errors. Based on this feedback, we introduced a major change to the reusable components to prevent developers from connecting two incompatible components to each other. To this end, every reusable component now specifies a meta-data describing the type of its input and output parameters. The output type of a component is used to determine whether it can be connected to another component. At runtime, the reusable components print an error message when they receive an incompatible object as input and return an empty object, which causes the execution of an algorithm to stop. Furthermore, we adapted the user interfaces of every tool in the WDK to dynamically adapt the reusable components developers can choose from at each stage of the recognition pipeline based on the components selected at the previous stages.

Participants P1 and P3 also mentioned that they did not understand some of the reusable components available in the WDK, such as the *ManualSegmentation* and the different components to label events and segments. P1 said: *"The Labeling is not clear what it does"* and also pointed out that he did not know what the *LabelMapper* was for. P3 reported that she did not know how to *"get to a segment from an event"*, which can be done with the *EventSegmentation*. To address this issue, we documented every reusable component in the WDK's GitHub website. For each reusable component, the documentation describes the type of input it requires and output it produces.

[4]DaVinci Resolve is a video editing and annotation tool: https://www.blackmagicdesign.com/products/davinciresolve/

The participants also mentioned several minor issues. When using the *Annotation* tool, the participants mentioned the loss of annotations because of closing the window without saving them beforehand [P2], the lack of information regarding what signals were being produced when a preprocessing algorithm was executed [P3], the lack of a legend to indicate how computed signals mapped to colors in the plot [P3] and the difficulty to recognize a data selection due to the similarity of the colors used to plot data and to select a range of data [P1,P3]. Regarding the *Analysis* tool, the participants pointed out that the tool was too 'laggy' when zooming into a plot with more than 400 segments [P1, P2], the lack of feedback to indicate that a time-intensive computation had finished [P1], that there was no way to know which table was editable and which one was not [P1] and that the labels shown above each list box were not consistent, as some of them were numbered and others were not [P3]. P2 also requested a feature to plot different signals without having to reset the zoom level of the plots. In the *Assessment* tool, the participants had difficulties to create a feature extraction algorithm. This was due to a lack of consistency between the user interfaces to reuse components in the different stages: for the preprocessing, segmentation, classification and validation stages, a single reusable component had to be selected from the user interface, whereas feature extraction algorithms had to be created by selecting multiple components and defining on which signal each of them were to be computed. P2 and P3 also noted a lack of consistency in the user interface to select features, which required developers to have executed a recognition algorithm once before a subset of features could be selected, but provided no indication about this restriction over the user interface. In the detail view of the *Assessment* tool, P2 and P3 criticized that the results of the recognition were not always visible depending on the zoom level of the plot that displays the data. We performed several minor changes to improve the usability of our toolkit based on the issues mentioned by the participants.

To assess the usability of the improved version of the WDK, we conducted a second user study with two engineers from the industry. To this end, we contacted two companies located in Munich that had collaborated with our research lab in the past and asked them to participate in our user study. The first participant (P1) was a senior software engineer (33 years old) working at a startup that offers professional coaching to soccer goalkeepers. The second participant (P2) was a recent graduate of computer science (25 years old) working as a data scientist in a startup specialized in wearable electronics. Both participants had previous experience with activity recognition. P1 had used mostly Matlab and had only passing experience in Python and P2 had two years of experience in Python and was familiar with the Node-RED platform but had no experience in Matlab.

We gave the participants specific tasks to solve with the WDK while thinking out-loud using the data from the GoalieGlove application. The tasks included annotating a data set with *event* and *range* annotations, finding outliers in the annotated data set, comparing the different signals (accelerometer, gyroscope and magnetometer) corresponding to two exercises, discussing possible feature extraction algorithms based on the exercise signatures, developing the algorithm we presented in Section 4 and assessing its recognition performance. After solving these tasks, we conducted an unstructured interview with the participants to inquire about their impression using the WDK. Finally, the participants were given a questionnaire with seven 5-point Likert scale questions. Each session lasted approximately 90 minutes. Table 3 shows the questionnaire we asked and the participant's answers.

The ease to understand the reusable components in the WDK was rated 4 by P1 and 3 by P2. While the participants understood how to instantiate components and connect them together in the *Development* tool and in the code, they acknowledged the need to refer to the documentation to understand the functionality behind the different components. P1 said: *"I am not sure what all of these do, but I am sure you will have some documentation"*. P2 had difficulties to understand how to combine the event detection and segmentation components: *"Obviously you need some user manual to know that SimplePeakDetector works with the EventSegmentation"* but had no difficulty reusing the feature extraction and classification components.

We found that both participants were quickly able to understand what *event* and *range annotation*s are and to annotate a data set using the *Annotation* tool. They welcomed the *Annotation* tool and mentioned that they were

Table 3. Questionnaire and answers of participants of the second study. The scales were: 1 (very difficult) to 5 (very easy) for Q1 and Q2; 1 (useless) to 5 (very useful) for Q3-Q6 and 1 (very unlikely) to 5 (very likely) for Q7.

#	Question	P1	P2
Q1	Do you find the reusable components in the WDK easy to understand?	4	3
Q2	Do you find the tools in the WDK easy to use?	4	4
Q3	Do you find the WDK useful to annotate your data?	5	5
Q4	Do you find the WDK useful to study your data set?	5	4
Q5	Do you find the WDK useful to develop a recognition algorithm?	4	5
Q6	Do you find the WDK useful to assess the performance of a recognition algorithm?	5	4
Q7	How likely are you to use the WDK within your organization?	5	5

not aware of other free annotation tools for time series that display video files next to the data. Both participants rated the WDK's usefulness to annotate data with a 5 (very useful). We also observed that the participants could use the *Analysis* tool without issues to display the annotated data. They quickly found outlier motions in the annotations and discussed possible feature extraction algorithms based on the data. Both participants found the tool useful to make sense of their data sets and design feature extraction algorithms. P1 said: *"The Analysis App is the most useful tool because it helps you see what's going on with the data. It helps you choose the features because you can see patterns in the data and on which axes to calculate the feature"*. The participants rated the WDK's usefulness to study their data sets with a 5 (P1) and a 4 (P2).

Both participants rated the WDK's usefulness to assess the performance of a recognition algorithm with a score of 5 (P1) and 4 (P2). In particular, the functionality to display the recognition results on top of the raw data in the frame-by-frame analysis was identified as the most convenient feature. P2 said: *"the part of the assessment can differentiate [the WDK] from other tools. [...] if you see a confusion matrix you see it misclassifies these exercises but you don't have a clue why [...]. It can help a lot to see the video and see that because of this it was not properly predicted and see that together with the data"*.

Both participants praised the WDK and rated how likely they were to use it within their organizations with a 5 (very likely). P2 said: *"there are no tools that are publicly available to developers so they create their own software [...] or they just do it intuitively by using standard parameters trusting they will work for their specific problem. With this tool I can see the data with different parameters and decide"*. On the other hand, both participants pointed out Matlab license fees as an issue and mentioned that their organizations would not be willing to afford the fees.

6.1 Discussion

Based on what we observed, we feel confident that the WDK can significantly lower the entrance barrier to the development of activity recognition applications. The participants of our studies mentioned that they were not aware of similar tools and found the WDK useful to automatize their development tasks. In particular, they praised the ability to reuse a broad set of existing functionality in their own applications. The features perceived to be the most useful by the participants are the functionality to annotate the data together with the video in the *Annotation* tool, to quickly assess and optimize the parameters of different algorithms and the frame-by-frame analysis to correlate the recognition results to the original data and reference video.

Furthermore, the participants of the user studies mentioned the difficulty to understand some reusable components. While the WDK enables the reuse of high-level components without having to understand their implementation details, developers still need to 1) be familiar with the Activity Recognition Chain and 2) understand the function, inputs and output produced by the components in the WDK. However, we believe that understanding and reusing the components in the WDK is significantly less time consuming than implementing a recognition algorithm without them. To facilitate learning the Activity Recognition Chain as well as the abstractions behind the WDK, we recently created a tutorial on activity recognition that relies on the reusable components in the WDK[5]. We found that most developers with no experience in activity recognition are able to finish the tutorial in a few hours and that they have less questions and are more effective at using the WDK afterwards.

In addition, both engineers from the second user study pointed out Matlab's license fees as a main limitation and suggested Python as a free alternative. In the future, the components the WDK offers could be re-implemented in Python or C++, or a combination of both. A C++ implementation of the *runtime components* would avoid differences between the execution of algorithms in the development environment and target device and is likely to lead to better execution performance. The current design of the WDK can be reused in future implementations.

7 CONCLUSIONS

This paper presented a toolkit to facilitate the development of activity recognition applications with wearables. In contrast to previous work, the WDK supports different tasks in the development lifecycle of an activity recognition application, such as the annotation and analysis of data and the development and performance assessment of an algorithm. Supporting these tasks within a single environment facilitates an iterative development process which is often necessary because developers rarely know upfront how to design activity recognition systems but rather develop them iteratively. To ensure the versatility of the toolkit, we developed it incrementally based on a variety of applications from different domains including sports, health, animal welfare and daily activity monitoring. We also collected feedback from different users with varying levels of experience in activity recognition and adapted the WDK to ensure it meets their needs.

One aspect we haven't studied until now is how well the *execution* and *memory cost*s computed by the WDK correlate to the amount of floating point operations and memory an actual algorithm implementation in the target device would require. As these metrics depend on the target device, its architecture and drivers, estimating them accurately at development time can be challenging. However, the costs estimated by the WDK provide a rough estimate that can be used to compare two or more algorithms to each other and make decisions early in the development lifecycle of an activity recognition application. Furthermore, the *execution* and *memory* costs of each reusable component can be adapted to a specific benchmark by modifying two lines of code in each component.

Furthermore, the current version of the WDK is limited to local computations. If the scale of the data exceeds what is physically possible to compute in a reasonable amount of time on a local device, developers might need to use remote computing power. Future work could extend the WDK to enable the simulation and assessment of activity recognition algorithms in parallel. To this end, every stage until the classification stage could be executed in parallel for the different input data files.

Despite the variety of reusable components and functionality already available in the WDK, the toolkit is far from finished. We are still extending its set of reusable components, refactoring its code, improving its usability and documenting it. A particular feature we are working on is the deployment of recognition algorithms into wearable devices. Our vision is to do so by sending an algorithm configuration wirelessly, without recompiling and flashing a firmware. To this end, we are currently porting the WDK's *runtime components* to C++.

We also haven't studied how to support application developers at creating applications that rely on activity recognition algorithms. We believe that many applications handle recognition results in similar ways. For example, they keep track of a training performance over time and compare the training performances among users. Future work could identify patterns of usage of recognized activities within applications and facilitate their development.

While the WDK eases the effort to develop recognition algorithms, these still need to be crafted manually. Different groups are investigating how to avoid this manual effort by adapting artificial neural networks to activity

[5] https://github.com/avenix/ARC-Tutorial

recognition applications with wearables [41, 60]. We are currently studying how to automatize the development of recognition algorithms by means of *generative design*. In generative design, the assembly of activity recognition algorithms is formulated as an optimization problem where the recognition performance (e.g., F1-Score) is used as an optimization metric and the computational requirements (e.g., memory, energy consumption) derive into constraints to the optimization. The reusable components in the WDK and the functionality to assess the performance of an algorithm represent a first step towards the realization of this idea.

ACKNOWLEDGMENTS

This work would not have been possible without the support from Prof. Bernd Brügge and Prof. Dan Siewiorek over the last two years. The author would also like to thank Prof. Oliver Amft and Prof. Antonio Krügger for allowing this work to be presented at their research labs and providing valuable ideas to improve the WDK.

REFERENCES

[1] Oliver Amft. 2010. A wearable earpad sensor for chewing monitoring. In *SENSORS, 2010 IEEE*. IEEE, 222–227.

[2] Daniel Ashbrook and Thad Starner. 2010. MAGIC: a motion gesture design tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2159–2168.

[3] Rafael Ballagas and Faraz Memon. 2007. iStuff mobile: rapidly prototyping new mobile phone interfaces for ubiquitous computing. *Proceedings of the SIGCHI conference on Human factors in computing systems* (2007), 1107–1116. https://doi.org/10.1145/1240624.1240793

[4] David Bannach, Oliver Amft, and Paul Lukowicz. 2008. Rapid Prototyping of Activity Recognition Applications. *IEEE Pervasive Computing* 7, 2 (apr 2008), 22–31. https://doi.org/10.1109/MPRV.2008.36

[5] Ling Bao and Stephen S Intille. 2004. Activity recognition from user-annotated acceleration data. In *International conference on pervasive computing*. Springer, 1–17.

[6] Peter Blank, Julian Hoßbach, Dominik Schuldhaus, and Bjoern M Eskofier. 2015. Sensor-based stroke detection and stroke type classification in table tennis. In *Proceedings of the 2015 ACM International Symposium on Wearable Computers*. ACM, 93–100.

[7] Andreas Bulling, Ulf Blanke, and Bernt Schiele. 2014. A tutorial on human activity recognition using body-worn inertial sensors. *ACM Computing Surveys (CSUR)* 46, 3 (2014), 33.

[8] Jay Chen, Karric Kwong, Dennis Chang, Jerry Luk, and Ruzena Bajcsy. 2006. Wearable sensors for reliable fall detection. In *2005 IEEE Engineering in Medicine and Biology 27th Annual Conference*. IEEE, 3551–3554.

[9] Pei-Yu Peggy Chi and Yang Li. 2015. Weave: Scripting cross-device wearable interaction. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. ACM, 3923–3932.

[10] Guglielmo Cola, Marco Avvenuti, Alessio Vecchio, Guang-Zhong Yang, and Benny Lo. 2015. An on-node processing approach for anomaly detection in gait. *IEEE Sensors Journal* 15, 11 (2015), 6640–6649.

[11] Anind K. Dey, Raffay Hamid, Chris Beckmann, Ian Li, and Daniel Hsu. 2004. a CAPpella. In *Proceedings of the 2004 conference on Human factors in computing systems - CHI '04*. ACM Press, New York, New York, USA, 33–40. https://doi.org/10.1145/985692.985697

[12] Jessica Echterhoff, Juan Haladjian, and Bernd Brügge. 2018. Gait Analysis in Horse Sports. In *Proceedings of the Fifth International Conference on Animal-Computer Interaction*. ACM, 3.

[13] Jessica Echterhoff, Juan Haladjian, and Bernd Brügge. 2018. Gait and Jump Classification in Modern Equestrian Sports. In *Proceedings of the 2018 ACM International Symposium on Wearable Computers*. ACM, 88–91.

[14] Davide Figo, Pedro C Diniz, Diogo R Ferreira, and João M Cardoso. 2010. Preprocessing techniques for context recognition from accelerometer data. *Personal and Ubiquitous Computing* 14, 7 (2010), 645–662.

[15] Francine Gemperle, Chris Kasabach, John Stivoric, Malcolm Bauer, and Richard Martin. 1998. Design for wearability. In *digest of papers. Second international symposium on wearable computers (cat. No. 98EX215)*. IEEE, 116–122.

[16] Nicholas Gillian and Joseph A Paradiso. 2014. The gesture recognition toolkit. *The Journal of Machine Learning Research* 15, 1 (2014), 3483–3487.

[17] Saul Greenberg and Chester Fitchett. 2001. Phidgets. In *Proceedings of the 14th annual ACM symposium on User interface software and technology - UIST '01*. ACM Press, New York, New York, USA, 209. https://doi.org/10.1145/502348.502388

[18] Benjamin H Groh, Martin Fleckenstein, Thomas Kautz, and Bjoern M Eskofier. 2017. Classification and visualization of skateboard tricks using wearable sensors. *Pervasive and Mobile Computing* 40 (2017), 42–55.

[19] Tobias Grosse-Puppendahl, Yannick Berghoefer, Andreas Braun, Raphael Wimmer, and Arjan Kuijper. 2013. OpenCapSense: A rapid prototyping toolkit for pervasive interaction using capacitive sensing. In *2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 152–159.

[20] Juan Haladjian, Katharina Bredies, and Bernd Bruegge. 2016. Interactex: An integrated development environment for smart textiles. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2016 ACM International Symposium on Wearable Computers*. ACM, 8–15. https://doi.org/10.1145/2971763.2971776

[21] Juan Haladjian, Katharina Bredies, and Bernd Bruegge. 2018. KneeHapp Textile: A Smart Textile System for Rehabilitation of Knee Injuries. In *Proceedings of the 15th International Conference on Wearable and Implantable Body Sensor Networks (BSN)*. IEEE, 9–12.

[22] Juan Haladjian, Ayca Ermis, Zardosht Hodaie, and Bernd Brügge. 2017. iPig: Towards Tracking the Behavior of Free-roaming Pigs. In *Proceedings of the Fourth International Conference on Animal-Computer Interaction (ACI2017)*. ACM, New York, NY, USA, 10:1—-10:5. https://doi.org/10.1145/3152130.3152145

[23] Juan Haladjian, Johannes Haug, Stefan Nüske, and Bernd Bruegge. 2018. A Wearable Sensor System for Lameness Detection in Dairy Cattle. *Multimodal Technologies and Interaction* 2, 2 (2018), 27.

[24] Juan Haladjian, Zardosht Hodaie, Stefan Nüske, and Bernd Brügge. 2017. Gait Anomaly Detection in Dairy Cattle. In *Proceedings of the Fourth International Conference on Animal-Computer Interaction (ACI2017)*. ACM, New York, NY, USA, 8:1—-8:8. https://doi.org/10.1145/3152130.3152135

[25] Juan Haladjian, Zardosht Hodaie, Han Xu, Mertcan Yigin, Bernd Bruegge, Markus Fink, and Juergen Hoeher. 2015. KneeHapp: A Bandage for Rehabilitation of Knee Injuries. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*. ACM, 181–184.

[26] Juan Haladjian, Constantin Scheuermann, Katharina Bredies, and Bernd Bruegge. 2017. A Smart Textile Sleeve for Rehabilitation of Knee Injuries. In *Proceedings of the 2017 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2017 ACM International Symposium on Wearable Computers (UbiComp '17)*. ACM, New York, NY, USA, 49–52. https://doi.org/10.1145/3123024.3123151

[27] Björn Hartmann, Leith Abdulla, Manas Mittal, and Scott R. Klemmer. 2007. Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '07)* (2007), 145–154. https://doi.org/10.1145/1240624.1240646

[28] Björn Hartmann, Scott R Klemmer, Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher, and Jennifer Gee. 2006. Reflective physical prototyping through integrated design, test, and analysis. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*. ACM, 299–308.

[29] Steven Houben, Connie Golsteijn, Sarah Gallacher, Rose Johnson, Saskia Bakker, Nicolai Marquardt, Licia Capra, and Yvonne Rogers. 2016. Physikit: Data engagement through physical ambient visualizations in the home. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 1608–1619.

[30] Steven Houben and Nicolai Marquardt. 2015. Watchconnect: A toolkit for prototyping smartwatch-centric cross-device applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 1247–1256.

[31] Bonifaz Kaufmann and Leah Buechley. 2010. Amarino: A Toolkit for the Rapid Prototyping of Mobile Ubiquitous Computing. In *Proceedings of the 12th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI '10)*. ACM, New York, NY, USA, 291–298. https://doi.org/10.1145/1851600.1851652

[32] Aftab Khan, James Nicholson, and Thomas Plötz. 2017. Activity Recognition for Quality Assessment of Batting Shots in Cricket using a Hierarchical Representation. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 3 (2017), 62.

[33] Travis Kirton, Sebastien Boring, Dominikus Baur, Lindsay MacDonald, and Sheelagh Carpendale. 2013. C4: a creative-coding API for media, interaction and animation. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction*. ACM, 279–286.

[34] David Ledo, Fraser Anderson, Ryan Schmidt, Lora Oehlberg, Saul Greenberg, and Tovi Grossman. 2017. Pineal: Bringing Passive Objects to Life with Embedded Mobile Devices. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 2583–2593.

[35] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. Evaluation strategies for HCI toolkit research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 36.

[36] Johnny C. Lee, Daniel Avrahami, Scott E. Hudson, Jodi Forlizzi, Paul H. Dietz, and Darren Leigh. 2004. The calder toolkit. In *Proceedings of the 2004 conference on Designing interactive systems processes, practices, methods, and techniques - DIS '04*. ACM Press, New York, New York, USA, 167–175. https://doi.org/10.1145/1013115.1013139

[37] Yang Li, Jason I Hong, and James A Landay. 2004. Topiary: a tool for prototyping location-enhanced applications. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*. ACM, 217–226.

[38] Kent Lyons, Helene Brashear, Tracy Westeyn, Jung Soo Kim, and Thad Starner. 2007. Gart: The gesture and activity recognition toolkit. In *International Conference on Human-Computer Interaction*. Springer, 718–727.

[39] Javier Marco, Eva Cerezo, and Sandra Baldassarri. 2012. ToyVision: a toolkit for prototyping tabletop tangible games. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM, 71–80.

[40] Amon Millner and Edward Baafi. 2011. Modkit: blending and extending approachable platforms for creating computer programs and interactive objects. In *Proceedings of the 10th International Conference on Interaction Design and Children*. ACM, 250–253.

[41] Vishvak S Murahari and Thomas Plötz. 2018. On attention models for human activity recognition. In *Proceedings of the 2018 ACM International Symposium on Wearable Computers*. ACM, 100–103.

[42] Michael Nebeling, Theano Mintsi, Maria Husmann, and Moira Norrie. 2014. Interactive development of cross-device user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2793–2802.

[43] J Nielsen. 1994. *Usability Engineering*. Academic Press Inc.

[44] Jakob Nielsen and Rolf Molich. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 249–256.

[45] Girish Palshikar and Others. 2009. Simple algorithms for peak detection in time-series. In *Proc. 1st Int. Conf. Advanced Data Analysis, Business Analytics and Intelligence*, Vol. 122.

[46] Shyamal Patel, Delsey Sherrill, Richard Hughes, Todd Hester, Theresa Lie-Nemeth, Paolo Bonato, David Standaert, and Nancy Huggins. 2006. Analysis of the Severity of Dyskinesia in Patients with Parkinson's Disease via Wearable Sensors. In *International Workshop on Wearable and Implantable Body Sensor Networks (BSN'06)*. IEEE, 123–126. https://doi.org/10.1109/BSN.2006.10

[47] Max Pfeiffer, Tim Duente, and Michael Rohs. 2016. Let your body move: a prototyping toolkit for wearable force feedback with electrical muscle stimulation. In *Proceedings of the 18th International Conference on Human-Computer Interaction with Mobile Devices and Services*. ACM, 418–427.

[48] Raf Ramakers, Fraser Anderson, Tovi Grossman, and George Fitzmaurice. 2016. Retrofab: A design tool for retrofitting physical interfaces using actuators, sensors and 3d printing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 409–419.

[49] Raf Ramakers, Kashyap Todi, and Kris Luyten. 2015. PaperPulse: An Integrated Approach for Embedding Electronics in Paper Designs. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15* (2015), 2457–2466. https://doi.org/10.1145/2702123.2702487

[50] Valkyrie Savage, Colin Chang, and Björn Hartmann. 2013. Sauron: embedded single-camera sensing of printed physical user interfaces. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 447–456.

[51] Valkyrie Savage, Sean Follmer, Jingyi Li, and Björn Hartmann. 2015. Makers' Marks: Physical markup for designing and fabricating functional objects. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 103–108.

[52] Giovanni Schiboni and Oliver Amft. 2018. Automatic dietary monitoring using wearable accessories. In *Seamless Healthcare Monitoring*. Springer, 369–412.

[53] Dominik Schuldhaus, Carolin Jakob, Constantin Zwick, Harald Koerger, and Bjoern M Eskofier. 2016. Your personal movie producer: generating highlight videos in soccer using wearables. In *Proceedings of the 2016 ACM International Symposium on Wearable Computers*. ACM, 80–83.

[54] Teddy Seyed, Alaa Azazi, Edwin Chan, Yuxi Wang, and Frank Maurer. 2015. Sod-toolkit: A toolkit for interactively prototyping and developing multi-sensor, multi-device environments. In *Proceedings of the 2015 International Conference on Interactive Tabletops & Surfaces*. ACM, 171–180.

[55] Akira Wakita and Yuki Anezaki. 2010. Intuino: an authoring tool for supporting the prototyping of organic interfaces. In *Proceedings of the 8th ACM Conference on Designing Interactive Systems*. ACM, 179–188.

[56] Chiuan Wang, Hsuan-Ming Yeh, Bryan Wang, Te-Yen Wu, Hsin-Ruey Tsai, Rong-Hao Liang, Yi-Ping Hung, and Mike Y Chen. 2016. CircuitStack: supporting rapid prototyping and evolution of electronic circuits. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 687–695.

[57] Tracy Westeyn, Helene Brashear, Amin Atrash, and Thad Starner. 2003. Georgia tech gesture toolkit: supporting experiments in gesture recognition. In *Proceedings of the 5th international conference on Multimodal interfaces*. ACM, 85–92.

[58] Chi-Jui Wu, Steven Houben, and Nicolai Marquardt. 2017. Eaglesense: Tracking people and devices in interactive spaces using real-time top-view depth-sensing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 3929–3942.

[59] Jishuo Yang and Daniel Wigdor. 2014. Panelrama: enabling easy specification of cross-device web applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2783–2792.

[60] Ming Zeng, Haoxiang Gao, Tong Yu, Ole J Mengshoel, Helge Langseth, Ian Lane, and Xiaobing Liu. 2018. Understanding and improving recurrent networks for human activity recognition by continuous attention. In *Proceedings of the 2018 ACM International Symposium on Wearable Computers*. ACM, 56–63.

[61] Bo Zhou, Harald Koerger, Markus Wirth, Constantin Zwick, Christine Martindale, Heber Cruz, Bjoern Eskofier, and Paul Lukowicz. 2016. Smart soccer shoe: monitoring foot-ball interaction with shoe integrated textile pressure sensor matrix. In *Proceedings of the 2016 ACM International Symposium on Wearable Computers*. ACM, 64–71.

A APPENDIX

This section lists the reusable components in the WDK until the date of submission of this article. The first and second columns of the tables provide the name and a description of each component. The *execution*, *memory* and *communication cost*s are abbreviated as *Exec*, *Mem* and *Comm* and described with respect to an input of size n.

Table 4. The preprocessing components produce n 32-bit floating-point values. The o variables in the *HighPassFilter* and *LowPassFilter* refer to these components' *order* property. The algorithms with a (*) in the memory field require O(1) memory when their *computationInPlace* property is set to *true* or $O(n)$ additional memory otherwise.

Preprocessing Components	Runtime	Exec	Mem
HighPassFilter	Butterworth High-pass filter	$13 * o * n$	*
LowPassFilter	Butterworth Low-pass filter	$31 * o * n$	*
Magnitude	$\sqrt{a_x(x_i)^2 + a_y(x_i)^2 + a_z(x_i)^2}$	$4 * n$	*
SquaredMagnitude	$a_x(x_i)^2 + a_y(x_i)^2 + a_z(x_i)^2)$	$2 * n$	*
Norm	$\|a_x(x_i)\| + \|a_y(x_i)\| + \|a_z(x_i)\|)$	$2 * n$	*
Derivative	$D'_i(x) = (x_i - x_{i+1})/\delta$ and $D''_i(x) = (x_{i-1} - x_i + x_{i+1})/\delta^2$	$40 * n$	*
S1	$\frac{max(x_i - x_{i-1}, ..., x_i - x_{i-k}) + max(x_i - x_{i+1}, ..., x_i - x_{i+k})}{2}$	$40 * k * n$	n
S2	$\frac{max(x_i - x_{i-1}, ..., x_i - x_{i-k}) + max(x_i - x_{i+1}, ..., x_i - x_{i+k})}{2k}$	$203 * k * n$	n

Table 5. The event detection components produce either none or one 32-bit floating-point value.

Event Detection Components	Runtime	Exec	Mem
SimplePeakDetector	Threshold-based peak detector	$11 * n$	1
MatlabPeakDetector	Matlab's peak detector	$1787 * n$	n

Table 6. The segmentation components produce s or $l + r$ values. The s, l, it and r variables in the *Exec* and *Mem* columns refer to these components' *segmentSize*, *iterationSize*, *segmentSizeLeft* and *segmentSizeRight* properties, respectively.

Segmentation Components	Runtime	Exec	Mem
SlidingWindow	Extracts *Segment*s of *windowSize* from the input *Signal*	$(n - s)/it$	s
EventSegmentation	Creates *Segment*s around the input *Event*s	$11 * n$	$l + r$
ManualSegmentation	Converts *event* and *range annotation*s to *Segment*s	-	-

Table 7. The time-domain feature extraction algorithms produce a single value except for the *Quantile* component, which produces *numQuantileParts* values. The octant is defined as: $Octant = 1$ if $x_1, x_2, x_3 > 0$ and $Octant = 7$ if $x_1, x_2, x_3 < 0$.

Time-domain Feature Extraction Components	Runtime	Exec	Mem
Min	Minimum value in the input *Signal*	n	1
Max	Maximum value in the input *Signal*	n	1
Mean	Average of every value in the input *Signal*	n	1
Median	Median of the values in the input *Signal*	$15 * n$	1
Variance	Variance of the input *Signal*	$2 * n$	1
STD	Standard Deviation of the values in the input *Signal*	$2 * n$	1
ZCR	Zero Crossing Rate of the input *Signal*	$5 * n$	1
Skewness	Skewness of the input *Signal*: $\sum_{i=1}^{n} \left(\frac{x_i - \bar{x})}{\sigma} \right)^3$	$6 * n$	1
Kurtosis	kurtosis of the input *Signal*: $\sum_{i=1}^{n} \left(\frac{x_i - \bar{x})}{\sigma} \right)^4$	$6 * n$	1
IQR	Interquartile Range of the values in the input *Signal*	$57 * n$	n
AUC	Area under the curve (trapezoid rule) of the input *Signal*: $\sum_{i=1}^{n-1} \frac{x_i + x_{i+1}}{n}$	$8 * n$	1
AAV	Average Absolute Variation of the input *Signal*: $\sum_{i=1}^{n-1} \frac{\lvert x_i - x_{i+1} \rvert}{n}$	$5 * n$	1
Correlation	Pearson correlation coefficient of the two input *Signal*s	$3 * n$	n
Energy	Sum of squared values of the input *Signal*	$2 * n$	1
Entropy	Entropy of the input signal: $\sum_{i=1}^{n} p_i \log(p_i)$ where p_i are the probability distribution values of the input *Signal*	n^2	n
MAD	Mean Absolute Deviation of the input *Signal*: $\sum_{i=1}^{n} \frac{\lvert x_i - \bar{x} \rvert}{n}$	$5 * n$	1
MaxCrossCorr	Maximum of the cross correlation coefficients of two input *Signals*	$161 * n$	n
Octants	Octant of each sample in the three input *Signal*s	$7 * n$	1
P2P	Difference between max. and min. values of the input *Signal*	$3 * n$	1
Quantile	q cutpoints that separate the distribution of values in the input *Signal*	$3 * n * log(n)$	q
RMS	Root Mean Squared of the input *Signal*: $\sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n}}$	$2 * n$	1
SMV	Signal Vector Magnitude of a two-dimensional input *Signal*: $\frac{1}{n} \sum_{i=1}^{n} \sqrt{x_i^2 + y_i^2)}$	$4 * n$	1
SMA	Sum of absolute values of a one or two-dimensional input *Signal*: $\sum_{i=1}^{n} \sum_{j=1}^{n} \lvert x_{ij} \rvert$	$m * n$	1

Table 8. The frequency-domain feature extraction components output a single value except for the *FFT* and *PowerSpectrum* which produce $n/2$ and n values respectively. Every frequency-domain feature extraction component receives the *Signal* with FFT coefficients produced by the *FFT* component as input.

Frequency-domain Feature Extraction Components Runtime		Exec	Mem
FFT	FFT of the input *Signal*	$n * \log(n)$	n
FFTDC	DC component of FFT coefficients	1	1
MaxFrequency	Largest Fourier coefficient	n	1
PowerSpectrum	Power spectrum of FFT coefficients	$4 * n$	n
SpectralCentroid	Centroid of FFT coefficients: $\frac{\sum_{i=1}^{n-1} \bar{y}_i y_i}{\sum_{i=1}^{n-1} y_i}$	$10 * n$	1
SpectralEnergy	Squared sum of FFT coefficients: $\sum_{i=1}^{n} \bar{y_i}^2$	$2 * n$	1
SpectralEntropy	Entropy of the FFT coefficients: $-\sum_{i=1}^{n} y_i \log 2(y_i)$	$21 * n$	1
SpectralFlatness	Flatness of the distribution of FFT coefficients: $\frac{\sqrt[n]{\prod_{i=1}^{n} x_i}}{\frac{1}{n}\sum_{i=1}^{n} x(n)}$	$68 * n$	1
SpectralSpread	Variance of the distribution of FFT coefficients	$11 * n$	1

Table 9. The classification components produce 9 bytes (a 1-byte label and an 8-byte timestamp). Their computational performance depend strongly on their implementation.

Classification Components Runtime	
LDClassifier	Linear Discriminant classifier
TreeClassifier	Decision tree classifier with properties: *maxNumSplits*
KNNClassifier	K-NN classifier with properties: *nNeighbors*, *distanceMetric*
EnsembleClassifier	Ensemble classifier with properties: *nLearners*
SVMClassifier	Support Vector Machine classifier with properties: *order*, *boxConstraint*

Table 10. The postprocessing components produce 9 bytes (a 1-byte label and an 8-byte timestamp).

Postprocessing Components Runtime		Exec	Mem
LabelMapper	Transforms the array of labels in a *ClassificationResult* by mapping labels in the *sourceLabeling* property to labels in the *targetLabeling* property	n	n
LabelSlidingWindowMaxSelector	Replaces every label at index *labelIndex* in a *ClassificationResult* with the most frequent label in the range [*labelIndex* − *windowSize*, *labelIndex* + *windowSize*], or with the NULL-class if no label occurs at least *minimumCount* times in the range	1	1

Table 11. Utility components available in the *runtime components* layer.

Runtime Utility Components Runtime		Exec	Mem	Comm
FeatureNormalizer	Normalizes a *FeaturesTable* by subtracting each row from the *means* property and dividing it by the *stds* property	$2 * n$	$2 * n$	n
ConstantMultiplier	Multiplies an input *Signal* by the *constant* property	n	n	n
Substraction	Subtracts the second column from the first column of a two-dimensional input *Signal*	$2 * n$	n	n
AxisMerger	Merges m *Signals* of size n into an nxm *Signal*	$3 * n$	$m * n$	$m * n$
AxisSelector	Selects the *axes* columns of the provided input *Signal*	-	$m * n$	$m * n$
RangeSelector	Outputs a new *Signal* with the values in the range $[rs...re]$ of the input *Signal*	$2 * n$	$re - rs$	$re - rs$

Table 12. The *FilesLoader* and *AnnotationsLoader* are convenience components used during development.

File Management Components Development	
FilesLoader	Loads and parses a data file (.csv or .mat) formats.
AnnotationsLoader	Loads ans parses an annotations file (.txt format)

Table 13. The labeling components are methods to label the *Event*s and *Segment*s produced by a recognition algorithm.

Labeling Components Development	
EventsLabeler	Labels *Events* as the closest *event annotation* under a specified *tolerance*
EventSegmentsLabeler	Labels *Segments* extracted around a detected *Event*
RangeSegmentsLabeler	Labels *Segments* based on *range annotations*

Table 14. The validation components receive a set of *FeaturesTable*s as input and produce a *ClassificationResult*.

Validation Components Development	
HoldoutValidator	Trains a classifier using the *trainData* and tests its with the *testData*
LeaveOneOutCrossValidator	Applies the leave-one-subject-out cross-validation technique

Table 15. Utility components available in development *components layer* of the repository.

Development Utility Components Development	
FeatureExtractor	Generates a *FeaturesTable* from an array of *Segments*
FeatureSelector	Identifies the *nFeatures* most relevant features of a *FeaturesTable*
NoOp	Outputs the input object without modification
PropertyGetter	Outputs the *property* property of the input object
PropertySetter	Sets the *property* property of the object in the *node* property to the input value
SegmentsGrouper	Outputs the input *Segments* grouped by their class
TableRowSelector	Removes every row of the input *FeaturesTable* with a label column not contained in the *selectedLabels* property.