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:

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...
(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 programming 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.

¹http://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.

![Development Lifecycle Diagram]

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.

![Diagram of WDK architecture](image)

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. Signals are two-dimensional arrays of floating-point values. Events represent a specific sample in a Signal and store an integer timestamp and a floating-point value. Segments 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. FeaturesTables are two-dimensional arrays of floating-point features and an 8-bit integer label column. ClassificationResults are arrays of 8-bit integer labels predicted by a machine learning classifier.

<table>
<thead>
<tr>
<th>Component Type</th>
<th>Input</th>
<th>Output</th>
<th>Used to…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessing</td>
<td>Signal</td>
<td>Signal</td>
<td>transform a Signal and prepare it for further processing</td>
</tr>
<tr>
<td>Event Detection</td>
<td>Signal</td>
<td>Events</td>
<td>detect the occurrence of specific events (e.g., peaks) in a Signal</td>
</tr>
<tr>
<td>Segmentation</td>
<td>Signal</td>
<td>Segments</td>
<td>divide a Signal into regions of interest</td>
</tr>
<tr>
<td>Feature Extraction</td>
<td>Segments</td>
<td>FeaturesTable</td>
<td>compute time or frequency-domain features of a Signal</td>
</tr>
<tr>
<td>Classification</td>
<td>FeaturesTable</td>
<td>ClassificationResult</td>
<td>predict a label for each feature vector in a FeaturesTable</td>
</tr>
<tr>
<td>Postprocessing</td>
<td>ClassificationResult</td>
<td>ClassificationResult</td>
<td>add, remove or alter labels in a sequence of predicted labels</td>
</tr>
<tr>
<td>Utilities</td>
<td>multiple</td>
<td>multiple</td>
<td>split, merge or transform data (e.g., extract values from a Signal)</td>
</tr>
</tbody>
</table>

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 activity recognition.

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.

![Graph](image)

**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.

![Diagram](image)

**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 costs are calculated at runtime by executing a recognition algorithm. Each reusable component computes its execution and memory costs 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 costs 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 costs 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.
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 Events of the detected peaks. The EventSegmentation generates Segments around the detected peaks by extracting the 200 samples to the left of the detected peak and 30 samples to its right. The Segments 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 Segments of 512 samples with 50% overlapping using the SlidingWindowSegmentation. The SlidingWindowSegmentation passes Segments 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. FeaturesTables output by the FeatureExtractor are passed to the TreeClassifier, which returns an array of labels in a ClassificationResult.

```matlab
%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 FeaturesTables 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

The post-processing component replaces every label at index $\text{labelIndex}$ in the array of predicted labels with the most frequent label in the range $[\text{labelIndex} - 3, \text{labelIndex} + 3]$, 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.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Gender</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Male</td>
<td>GoalieGlove</td>
</tr>
<tr>
<td>P2</td>
<td>Male</td>
<td>Recognition of basketball defensive training exercises</td>
</tr>
<tr>
<td>P3</td>
<td>Female</td>
<td>HipRApp</td>
</tr>
</tbody>
</table>

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”. 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.

---

4DaVinci Resolve is a video editing and annotation tool: https://www.blackmagicdesign.com/products/davinciresolve/
The Wearables Development Toolkit: An Integrated Development Environment for Activity...

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 annotations 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.

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

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\textsuperscript{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 costs 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

\textsuperscript{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

The Wearables Development Toolkit: An Integrated Development Environment for Activity...


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 costs 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 \textit{HighPassFilter} and \textit{LowPassFilter} refer to these components’ order property. The algorithms with a (*) in the memory field require \( O(1) \) memory when their \textit{computationInPlace} property is set to \textit{true} or \( O(n) \) additional memory otherwise.

<table>
<thead>
<tr>
<th>Preprocessing Components</th>
<th>Runtime</th>
<th>Exec</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>HighPassFilter</td>
<td></td>
<td>13 * o * n</td>
<td>*</td>
</tr>
<tr>
<td>LowPassFilter</td>
<td></td>
<td>31 * o * n</td>
<td>*</td>
</tr>
<tr>
<td>Magnitude</td>
<td></td>
<td>4 * n</td>
<td>*</td>
</tr>
<tr>
<td>SquaredMagnitude</td>
<td></td>
<td>2 * n</td>
<td>*</td>
</tr>
<tr>
<td>Norm</td>
<td></td>
<td>2 * n</td>
<td>*</td>
</tr>
<tr>
<td>Derivative</td>
<td></td>
<td>40 * n</td>
<td>*</td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td>40 * k * n</td>
<td>n</td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td>203 * k * n</td>
<td>n</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Event Detection Components</th>
<th>Runtime</th>
<th>Exec</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimplePeakDetector</td>
<td></td>
<td>11 * n</td>
<td>1</td>
</tr>
<tr>
<td>MatlabPeakDetector</td>
<td></td>
<td>1787 * n</td>
<td>n</td>
</tr>
</tbody>
</table>

Table 6. The segmentation components produce \( s \) or \( l + r \) values. The \( s, l, it \) and \( r \) variables in the \textit{Exec} and \textit{Mem} columns refer to these components’ \textit{segmentSize}, \textit{iterationSize}, \textit{segmentSizeLeft} and \textit{segmentSizeRight} properties, respectively.

<table>
<thead>
<tr>
<th>Segmentation Components</th>
<th>Runtime</th>
<th>Exec</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>SlidingWindow</td>
<td></td>
<td>(n - s)/it</td>
<td>s</td>
</tr>
<tr>
<td>EventSegmentation</td>
<td></td>
<td>11 * n</td>
<td>l + r</td>
</tr>
<tr>
<td>ManualSegmentation</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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 $x_1, x_2, x_3 < 0$.

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

<table>
<thead>
<tr>
<th>Frequency-domain Feature Extraction Components</th>
<th>Runtime</th>
<th>Exec</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td></td>
<td>( n \times \log(n) )</td>
<td>( n )</td>
</tr>
<tr>
<td>FFTDC</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MaxFrequency</td>
<td></td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>PowerSpectrum</td>
<td></td>
<td>( 4 \times n )</td>
<td>( n )</td>
</tr>
<tr>
<td>SpectralCentroid</td>
<td></td>
<td>( 10 \times n )</td>
<td>1</td>
</tr>
<tr>
<td>SpectralEnergy</td>
<td></td>
<td>( 2 \times n )</td>
<td>1</td>
</tr>
<tr>
<td>SpectralEntropy</td>
<td></td>
<td>( 21 \times n )</td>
<td>1</td>
</tr>
<tr>
<td>SpectralFlatness</td>
<td></td>
<td>( 68 \times n )</td>
<td>1</td>
</tr>
<tr>
<td>SpectralSpread</td>
<td></td>
<td>( 11 \times n )</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Classification Components</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDClassifier</td>
<td>Linear Discriminant classifier</td>
</tr>
<tr>
<td>TreeClassifier</td>
<td>Decision tree classifier with properties: maxNumSplits</td>
</tr>
<tr>
<td>KNNClassifier</td>
<td>K-NN classifier with properties: nNeighbors, distanceMetric</td>
</tr>
<tr>
<td>EnsembleClassifier</td>
<td>Ensemble classifier with properties: nLearners</td>
</tr>
<tr>
<td>SVMClassifier</td>
<td>Support Vector Machine classifier with properties: order, boxConstraint</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Postprocessing Components</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>LabelMapper</td>
<td>Transforms the array of labels in a ClassificationResult by mapping labels in the sourceLabeling property to labels in the targetLabeling property</td>
</tr>
<tr>
<td>LabelSlidingWindowMaxSelector</td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 11. Utility components available in the runtime components layer.

<table>
<thead>
<tr>
<th>Runtime Utility Components</th>
<th>Exec</th>
<th>Mem</th>
<th>Comm</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureNormalizer</td>
<td>$2 \times n$</td>
<td>$2 \times n$</td>
<td>$n$</td>
</tr>
<tr>
<td>ConstantMultiplier</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>Subtraction</td>
<td>$2 \times n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>AxisMerger</td>
<td>$3 \times n$</td>
<td>$m \times n$</td>
<td>$m \times n$</td>
</tr>
<tr>
<td>AxisSelector</td>
<td>$- m \times n$</td>
<td>$m \times n$</td>
<td>$m \times n$</td>
</tr>
<tr>
<td>RangeSelector</td>
<td>$2 \times n$</td>
<td>$re - rs$</td>
<td>$rs - rs$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>File Management Components</th>
<th>Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilesLoader</td>
<td>Loads and parses a data file (.csv or .mat) formats.</td>
</tr>
<tr>
<td>AnnotationsLoader</td>
<td>Loads and parses an annotations file (.txt format)</td>
</tr>
</tbody>
</table>

Table 13. The labeling components are methods to label the Events and Segments produced by a recognition algorithm.

<table>
<thead>
<tr>
<th>Labeling Components</th>
<th>Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventsLabeler</td>
<td>Labels Events as the closest event annotation under a specified tolerance</td>
</tr>
<tr>
<td>EventSegmentsLabeler</td>
<td>Labels Segments extracted around a detected Event</td>
</tr>
<tr>
<td>RangeSegmentsLabeler</td>
<td>Labels Segments based on range annotations</td>
</tr>
</tbody>
</table>

Table 14. The validation components receive a set of FeaturesTables as input and produce a ClassificationResult.

<table>
<thead>
<tr>
<th>Validation Components</th>
<th>Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>HoldoutValidator</td>
<td>Trains a classifier using the trainData and tests it with the testData</td>
</tr>
<tr>
<td>LeaveOneOutCrossValidator</td>
<td>Applies the leave-one-subject-out cross-validation technique</td>
</tr>
</tbody>
</table>

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

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