Chapman & Hall/CRC
Computer & Information Science Series

Human Activity Recognition

Using Wearable Sensors and Smartphones

Miguel A. Labrador
Oscar D. Lara Yejas

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Human Activity Recognition

Using Wearable Sensors and Smartphones

CHAPMAN & HALL/CRC
COMPUTER and INFORMATION SCIENCE SERIES

Series Editor: Sartaj Sahni

PUBLISHED TITLES

ADVERSARIAL REASONING: COMPUTATIONAL APPROACHES TO READING THE OPPONENT'S MIND
Alexander Kott and William M. McEneaney

DELAUNAY MESH GENERATION
Siu-Wing Cheng, Tamal Krishna Dey, and Jonathan Richard Shewchuk

DISTRIBUTED SENSOR NETWORKS, SECOND EDITION
S. Sitharama Iyengar and Richard R. Brooks

DISTRIBUTED SYSTEMS: AN ALGORITHMIC APPROACH
Sukumar Ghosh

ENERGY-AWARE MEMORY MANAGEMENT FOR EMBEDDED MULTIMEDIA SYSTEMS: A COMPUTER-AIDED DESIGN APPROACH
Florin Balasa and Dhiraj K. Pradhan

ENERGY EFFICIENT HARDWARE-SOFTWARE CO-SYNTHESIS USING RECONFIGURABLE HARDWARE
Jingzhao Ou and Viktor K. Prasanna

FUNDAMENTALS OF NATURAL COMPUTING: BASIC CONCEPTS, ALGORITHMS, AND APPLICATIONS
Leandro Nunes de Castro

HANDBOOK OF ALGORITHMS FOR WIRELESS NETWORKING AND MOBILE COMPUTING
Azzedine Boukerche

HANDBOOK OF APPROXIMATION ALGORITHMS AND METAHEURISTICS
Teofilo F. Gonzalez

HANDBOOK OF BIOINSPIRED ALGORITHMS AND APPLICATIONS
Stephan Olariu and Albert Y. Zomaya

HANDBOOK OF COMPUTATIONAL MOLECULAR BIOLOGY
Srinivas Aluru

HANDBOOK OF DATA STRUCTURES AND APPLICATIONS
Dinesh P. Mehta and Sartaj Sahni

HANDBOOK OF DYNAMIC SYSTEM MODELING
Paul A. Fishwick

HANDBOOK OF ENERGY-AWARE AND GREEN COMPUTING
Ishfaq Ahmad and Sanjay Ranka

HANDBOOK OF PARALLEL COMPUTING: MODELS, ALGORITHMS AND APPLICATIONS
Sanguthevar Rajasekaran and John Reif

HANDBOOK OF REAL-TIME AND EMBEDDED SYSTEMS
Insup Lee, Joseph Y-T. Leung, and Sang H. Son

HANDBOOK OF SCHEDULING: ALGORITHMS, MODELS, AND PERFORMANCE ANALYSIS
Joseph Y.-T. Leung

HIGH PERFORMANCE COMPUTING IN REMOTE SENSING
Antonio J. Plaza and Chein-I Chang

HUMAN ACTIVITY RECOGNITION: USING WEARABLE SENSORS AND SMARTPHONES
Miguel A. Labrador and Oscar D. Lara Yejas

INTRODUCTION TO NETWORK SECURITY
Douglas Jacobson

LOCATION-BASED INFORMATION SYSTEMS: DEVELOPING REAL-TIME TRACKING APPLICATIONS
Miguel A. Labrador, Alfredo J. Pérez, and Pedro M. Wightman

METHODS IN ALGORITHMIC ANALYSIS
Vladimir A. Dobrushkin

MULTICORE COMPUTING: ALGORITHMS, ARCHITECTURES, AND APPLICATIONS
Sanguthevar Rajasekaran, Lance Fiondella, Mohamed Ahmed, and Reda A. Ammar

PERFORMANCE ANALYSIS OF QUEUING AND COMPUTER NETWORKS
G. R. Dattatreya

THE PRACTICAL HANDBOOK OF INTERNET COMPUTING
Munindar P. Singh

SCALABLE AND SECURE INTERNET SERVICES AND ARCHITECTURE
Cheng-Zhong Xu

SOFTWARE APPLICATION DEVELOPMENT: A VISUAL C++®, MFC, AND STL TUTORIAL
Bud Fox, Zhang Wenzu, and Tan May Ling

SPECULATIVE EXECUTION IN HIGH PERFORMANCE COMPUTER ARCHITECTURES
David Kaeli and Pen-Chung Yew

VEHICULAR NETWORKS: FROM THEORY TO PRACTICE
Stephan Olariu and Michele C. Weigle

Human Activity Recognition

Using Wearable Sensors and Smartphones

Miguel A. Labrador
Oscar D. Lara Yejas

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20131014

International Standard Book Number-13: 978-1-4665-8828-8 (eBook - PDF)

Dedication

Dedicado a mi esposa Mariela, y a mis hijos Miguel Andrés y Daniel Ignacio.

Miguel A. Labrador

A mis padres Melit y Roberto, mi hermana Melissa y a Anamaría, con mi más inmenso cariño.

Oscar D. Lara Yejas

Contents

List of Figures

List of Tables

Preface

The pervasiveness of today's mobile devices and their increasing range of capabilities - especially in terms of wireless communication interfaces and sensors - has enabled a wide spectrum of mobile applications that are transforming our daily lives. Smartphones equipped with GPSs, accelerometers, compasses, cameras, microphones, among others, are now able to not only track the user, but also gather contextual information such as audio, video, and motion patterns. Furthermore, additional sensors can be easily integrated into mobile applications to acquire physiological data, e.g., body temperature, heart rate, respiration rate, respiration depth, oxygen level, or blood pressure, just to mention a few, allowing for real-time health monitoring. These measurements may even be compared to reference values from individuals with similar physical conditions in terms of age, weight, gender, and height, for the diagnosis and prognosis of diseases or medical disorders.

This book addresses the automatic recognition of human activities from pervasive wearable sensors, which is a crucial component for health monitoring. In addition to simply collecting measurements from variables of interest, we elaborate on the additional inference layers required to determine the user's activity during a certain period of time. For that purpose, machine learning and pattern recognition tools play a key role. Although *Human Activity Recognition* (HAR) is not a new research topic, it is gaining more and more attention due to its endless potential applications. For instance, a mobile application could generate a daily report summarizing the total amount of time the user has *walked*, *run*, *slept*, or *eaten*. This information —in real time as well as historically— might help detecting potential anomalies and could serve as motivation for users to improve their daily exercising, eating, and sleeping behaviors. As we will see throughout the book, HAR also has potential in other application domains such as entertainment, elderly assistance, and tactical operations, among others.

Performing real-time activity recognition locally in a smartphone (as opposed to requiring a server for processing) is beneficial in terms of reliability, scalability, and energy consumption. Yet, it also becomes challenging since mobile devices are still constrained in terms of storage, communication, and processing capabilities. Traditionally, HAR has been carried out in desktop computers or servers, capable of storing large amounts of data collected from sensors and executing specialized and computationally expensive machine learning algorithms. The book studies these two approaches by pre-

senting Centinela, an offline server-oriented HAR system, and later Vigilante, a completely mobile real-time activity recognition system. Finally, the reader is provided with a practical guide to the Android framework toward the development of activity recognition applications.

Book Origin and Overview

This book is the result of nearly four years of research in human activity recognition using wearable sensors and smartphones. It involved the investigation of new architectures, middleware, sensors, algorithms, protocols, and mechanisms to address particular problems related to the automatic identification of human activities.

The book has two parts and fourteen chapters. Part I encompasses the design of HAR systems from the research point of view. Chapter 1 introduces the topic of human activity recognition using wearable sensors. Chapter 2 explains the most important design issues to consider when developing HAR systems. In addition, it introduces the reader to the fundamentals of feature extraction and machine learning tools. Chapter 3 covers the state of the art in HAR systems, including online (i.e., real-time) and offline HAR systems. Chapter 4 shows how the incorporation of physiological signals improves the accuracy in the recognition of activities. Chapter 5 shows how to perform real-time human activity recognition, i.e., embedding all tasks in the smartphone. Chapter 6 expands machine learning boundaries by presenting new fusion and selection strategies for multiple classifier systems. Finally, Chapter 7 concludes Part I with a summary of findings, results, and future research considerations.

Part II includes a practical guide to developing mobile HAR applications in an Android smartphone. Chapter 8 presents a brief introduction to the Android platform. Chapter 9 guides the reader through the software development environment setup. In addition, it shows how to test an Android application in a software device (i.e., emulator) and in an actual smartphone. Chapter 10 illustrates how to connect and obtain data from some of the most common sensors available in current smartphones. Chapter 11 shows how to connect an Android device with external sensors using the Bluetooth interface. Chapter 12 elaborates on the different alternatives available in Android to permanently store and retrieve data. Chapters 13 and 14 conclude the book by showing how to implement feature extraction and evaluation of classification algorithms in the smartphone.

Intended Audience

The book is intended for undergraduate students in their junior or senior years, professors, researchers, and industry professionals interested in the design and implementation of human activity recognition systems using smartphones and wearable sensors. The book can also be used as reference material in a graduate class on the same topic.

Acknowledgments

There are several people and organizations that deserve our appreciation. First, we would like to recognize the work of other members of our research team: special thanks to Idalides Vergara, Alfredo Pérez, Diego Mendez, José Posada, and Sean Barbeau who made many useful contributions through technical discussions in the lab. Also, special thanks to Juan José Marrón and Adrián Menéndez, who contributed with the code to explain how to use the sensors of the smartphone (Chapter 10). Second, we would like to give thanks to the National Science Foundation for the financial support to the Research Experience for Undergraduates (REU) program. Many thanks to the REU students who participated in the development of Vigilante over the last three years, such as Elizabeth Rodríguez, Aaron Young, José Cadena, Andrés Pérez, Joshua Sheehy, and Christopher Eggert. Third, we would also like to give special thanks to Elsevier and the IEEE for giving us permission to use the research material that we published in their journals and conferences, and the staff of Taylor & Francis, particularly to Randi Cohen, for their support during all the phases of the book. Last but not least, we want to acknowledge our own families for their patience, support, and understanding during all these years of continuous, hard work. Many thanks to Anamaría Bejarano for reviewing the book's manuscript.

About the Authors

Miguel A. Labrador earned his M.Sc. in telecommunications and the Ph.D. degree in information science with concentration in telecommunications from the University of Pittsburgh, in 1994 and 2000, respectively. Since 2001, he has been with the University of South Florida, Tampa, where he is currently a full professor in the department of computer science and engineering, the director of the graduate programs, and the director of the research experiences for undergraduates program. His research interests are in ubiquitous sensing, location-based services, energy-efficient mechanisms for wireless sensor networks, and design and performance evaluation of computer networks and communication protocols. He has published more than 100 technical and educational papers in journals and conferences devoted to these topics. Dr. Labrador has served as technical program committee member of many IEEE conferences and is currently area editor of *Computer Communications* and editorial board member of the *Journal of Network and Computer Applications*, both Elsevier Science journals. Dr. Labrador is the lead author of *Location-Based Information Systems: Developing Real-Time Tracking Applications*, Taylor & Francis, and *Topology Control in Wireless Sensor Networks*, Springer. Dr. Labrador is senior member of the IEEE and a member of ACM, ASEE and Beta Phi Mu.

Oscar D. Lara Yejas received his B.Sc. in systems engineering from Universidad del Norte, Barranquilla, Colombia, in 2007. He received his M.Sc. in

computer science in 2010 and his Ph.D. in computer science and engineering in 2012, both from the University of South Florida. Dr. Lara Yejas has significant industry experience in the public utilities sector, leading projects related to mobile visualization of geographic and cartographic information, real-time tracking applications, and telemetry. He has also worked on intelligent transportation systems with the Center for Urban Transportation Research (CUTR) at the University of South Florida. He was part of the development team of the Travel Assistance Device (TAD), a mobile application for aiding cognitively disabled people to use public transportation. Dr. Lara Yejas' dissertation on human activity recognition with wearable sensors —under the advising of Dr. Labrador— has given birth to this book. In 2012, Dr. Lara Yejas joined International Business Machines Corporation (IBM) within the InfoSphere BigInsights group. His current work focuses on large-scale analytics in distributed computing environments. Further research interests of his encompass but are not limited to machine learning, big data analytics, location-based systems, as well as multiobjective optimization using swarm intelligence methods.

Miguel A. Labrador
Oscar D. Lara Yejas

Part I

Human Activity Recognition: Theory Fundamentals

Chapter 1

Introduction

During the past decade, there was an exceptional development of microelectronics and computer systems, enabling sensors and mobile devices with unprecedented characteristics. Their high computational power, small size, and low cost allowed people to interact with these devices as part of their daily living. That was the genesis of *Ubiquitous Sensing*, an active research area with the main purpose of extracting knowledge from the data acquired by pervasive sensors [91]. Particularly, the recognition of human activities has become a task of high interest within the field, especially for medical, military, and security applications. For instance, patients with diabetes, obesity, or heart disease are often required to follow a well-defined exercise routine as part of their treatment [67]. Therefore, recognizing activities such as *walking*, *running*, or *resting* becomes quite useful to provide feedback to the caregiver about the patient's behavior. Likewise, patients with dementia and other mental pathologies might be monitored to detect abnormal activities and thereby prevent undesirable consequences [114]. An interactive game or simulator might also require information about which activity the user is performing in order to respond accordingly. Finally, in tactical scenarios, precise information about the soldiers' activities along with their locations and health conditions is highly beneficial for their performance and safety. Such information is also helpful to support decision making in both combat and training environments. Given the relevance of HAR and its applications, an overview of the most noticeable approaches is presented next.

1.1 Human activity recognition approaches

The recognition of human activities has been approached in two different ways, namely using *external* and *wearable* sensors. In the former, the sensors are fixed in predetermined points of interest, so the inference of activities entirely depends on the voluntary interaction of the users with the sensors. In the latter, the sensors are attached to the user.

Intelligent homes [108, 106, 113, 98] are a typical example of external sensing. These systems are able to recognize fairly complex activities (e.g., eating, taking a shower, washing dishes, etc.) because they rely on data from a num-

ber of sensors placed in target objects which people are supposed to interact with (e.g., stove, faucet, washing machine, etc.). Nonetheless, nothing can be done if the user is out of the reach of the sensors or they perform activities that do not require interaction with them. Additionally, the installation and maintenance of the sensors usually entail high costs.

Cameras have also been employed as external sensors for HAR. In fact, the recognition of activities and gestures from video sequences has been the focus of extensive research [107, 33, 68, 20]. This is especially suitable for security (e.g., intrusion detection) and interactive applications. A remarkable example, and also commercially available, is the *Kinect* game console [100] developed by Microsoft, which allows the user to interact with the game by means of gestures, without any controller device. Nevertheless, video sequences certainly have some issues in HAR. The first one is *privacy*, as not everyone is willing to be permanently monitored and recorded by cameras. The second one is *pervasiveness* because video recording devices are difficult to attach to target individuals in order to obtain images of their entire body during daily living activities. The monitored individuals should then stay within a perimeter defined by the position and the capabilities of the camera(s). The last issue would be *complexity*, since video processing techniques are computationally expensive, hindering a real time HAR system from being scalable.

The aforementioned limitations motivate the use of wearable sensors in HAR, in which most of the measured attributes are related to the user's movement (e.g., using accelerometers or GPS), environmental variables (e.g., temperature and humidity), or physiological signals (e.g., heart rate or electrocardiogram).

1.2 Human activity recognition with wearable sensors

Similar to other machine learning applications, activity recognition requires two stages, i.e., *training* and *testing* — also called *evaluation*. Figure 1.1 illustrates the common phases involved in these two processes. The training stage initially requires a time series dataset of measured attributes from individuals performing each activity. The time series are divided into time windows to apply *feature extraction* in order to filter relevant information in the raw signals and define metrics to compare them. Later, *learning* methods are used to generate an activity recognition model from the dataset of extracted features. Likewise, for testing, data are collected during a time window and a feature vector — also called feature set — is calculated. Such a feature set is evaluated in the a priori trained learning model, generating a predicted activity label.

A generic data acquisition architecture for HAR systems was also identified, as shown in Figure 1.2. In the first place, *wearable sensors* are attached to

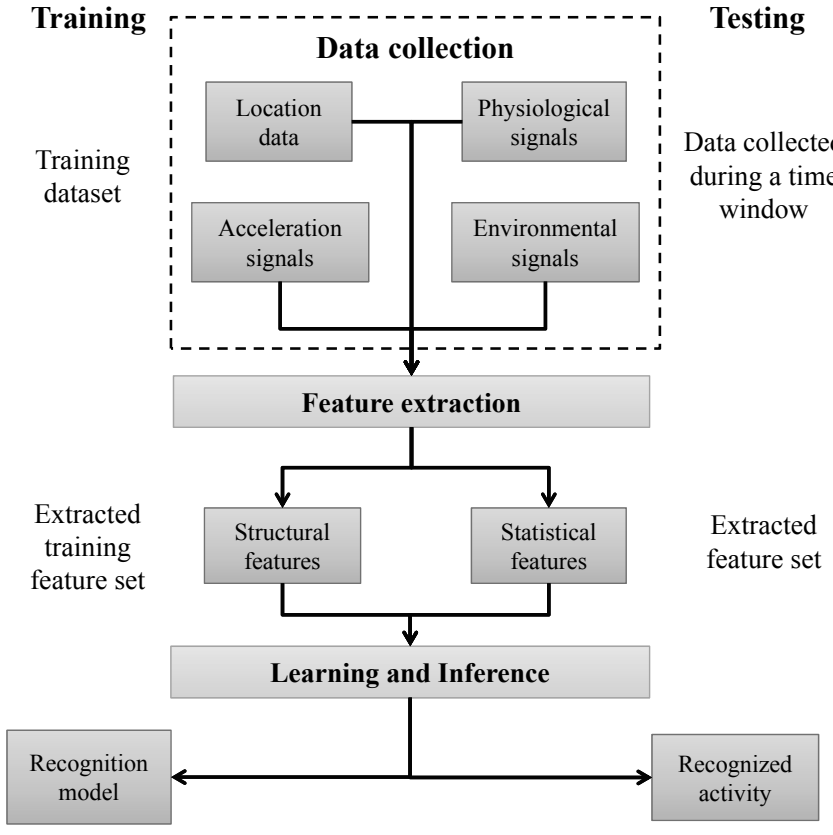

FIGURE 1.1: General data flow for training and testing HAR systems based on wearable sensors © 2012 IEEE. Reprinted, with permission, from [78].

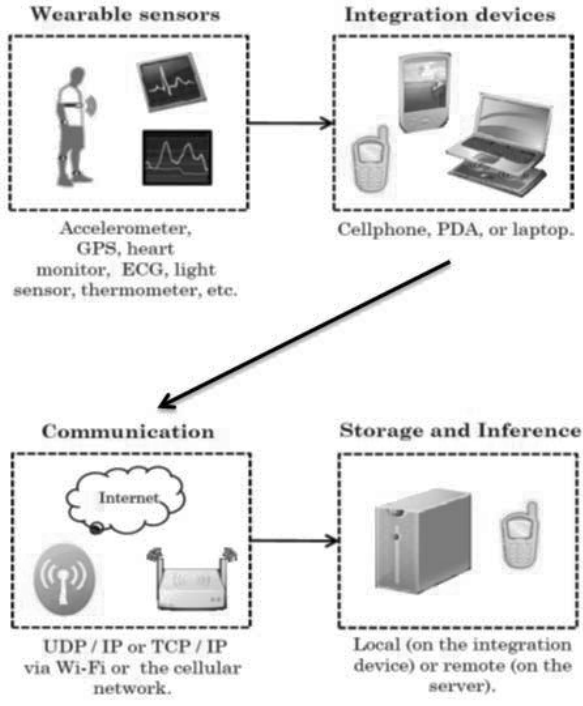

FIGURE 1.2: Generic data acquisition architecture for human activity recognition © 2012 IEEE. Reprinted, with permission, from [78].

the person's body to measure attributes of interest such as motion [65], location [38], temperature [88], and ECG [66], among others. These sensors communicate with an *integration device* (ID), which can be a cellphone [31, 86], a PDA [88], a laptop [104, 66], or a customized embedded system [69, 81]. The main purpose of the ID is to preprocess the data received from the sensors and, in some cases, send them to an application server for real time monitoring, visualization, analysis, inference, and storage. The *communication* protocol might be UDP/IP or TCP/IP, according to the desired level of reliability.

Notice that not all of these components are necessarily implemented in every HAR system. In [65, 57, 60], the data are collected offline, so there is neither communication nor server processing. Other systems incorporate sensors within the ID [28, 96, 32], or carry out the inference process directly on it [26, 96].

1.3 Human activity recognition problem

The data collected from wearable sensors are naturally indexed over the time dimension, which allows to define the human activity recognition problem as follows:

Definition 1.1 ***Human Activity Recognition Problem (HARP):*** *Given a set $S = \{S_0, ..., S_{k-1}\}$ of k time series, each one from a particular measured attribute, and all defined within time interval $I = [t_\alpha, t_\omega]$, the goal is to find a temporal partition $\langle I_0, ..., I_{r-1} \rangle$ of I, based on the data in S, and a set of labels representing the activity performed during each interval I_j (e.g., sitting, walking, etc.). This implies that time intervals I_j are consecutive, non-empty, non-overlapping, and such that $\bigcup_{j=0}^{r-1} I_j = I$.*

This definition is valid assuming that activities are not simultaneous, i.e., a person does not walk and run at the same time. Note that the HARP is not feasible to be solved deterministically. The number of combinations of attribute values and activities can be very large — or even infinite — and finding transition points becomes hard as the duration of each activity is generally unknown. Therefore, machine learning tools are widely used to recognize activities. A relaxed version of the problem is then introduced, dividing the time series into fixed length time windows, as follows:

Definition 1.2 ***Relaxed HAR problem:*** *Given a set $W = \{W_0, ..., W_{m-1}\}$ of m equally sized time windows, totally or partially labeled, and such that each W_i contains a set of time series $S_i = \{S_{i,0}, ..., S_{i,k-1}\}$ from each of the k measured attributes, and a set $A = \{a_0, ..., a_{n-1}\}$ of activity labels, the goal is to find a mapping function $f : S_i \rightarrow A$ that can be evaluated for all possible values of S_i, such that $f(S_i)$ is as similar as possible to the actual activity performed during W_i.*

Notice that this relaxation introduces some error to the model during *transition windows*, since a person might perform more than one activity within a single time window. However, the number of transitions is expected to be much smaller than the total number of time windows, which makes the relaxation error not significant for most of the applications.

Definition 1.2 specifies the very first problem addressed in this book: building a learning model to recognize activities offline given a set of measured attributes. Yet, a number of real-world applications are compelled to also deliver immediate feedback on the user's activities, especially in health care and military operations. Such tasks require coupling the sensors with an integration device (e.g., a smartphone), bringing about additional challenges as mobile devices are constrained in terms of computational power, memory, and energy.

These limitations are particularly critical in activity recognition applications which entail high demands due to communication, feature extraction, classification, and transmission of large amounts of raw data. Moreover, current open source machine learning API's such as WEKA [18] and JDM [11] are neither designed nor optimized to run on mobile platforms. Thus, the second problem to be addressed in the present book is the mobile implementation of a human activity recognition system meeting response time and energy consumption requirements.

A noticeable approach in activity recognition is to rely not only on a single model, but combine the output of several learners in order to produce more accurate and diverse predictions. That is the main philosophy behind *multiclassifier systems*, which are shown to be effective, though they entail additional computational complexity. In such direction, a third focus of study in this book attempts to formulate new strategies to effectively combine predictions from several learners. The problem is formally defined as follows:

Definition 1.3 *Given a classification problem with a feature space* $\mathcal{X} \in \mathcal{R}^n$ *and a set of classes* $\Omega = \{\omega_0, ..., \omega_{n-1}\}$*, an instance* $x \in \mathcal{X}$ *to be classified, and a set of predictions* $S = \{s_0, ..., s_{k-1}\}$ *for* x*, from* k *classifiers, the goal of a multiclassifier system is to return the correct label* ω^* *iff* $\exists s_i \in S \mid \omega^* = s_i$*.*

1.4 Structure of the book

The book has two parts and fourteen chapters. Part I is about the description of the HAR systems. Chapter 1 introduces the topic of human activity recognition using wearable sensors. Chapter 2 explains the most important design issues to consider when developing HAR systems. In addition, it introduces the reader to the fundamentals of feature extraction and learning mechanisms and tools. Chapter 3 includes a comprehensive literature review on human activity recognition with wearable sensors, with a two-level taxonomy based on the learning approach and the response time of the system. Both, offline and on-line (real-time) HAR systems are considered. A qualitative analysis of twenty-eight HAR systems is presented, providing the reader with guidelines to select the most appropriate approach to use in a specific case study. Finally, the chapter also features introductory material to pattern recognition and machine learning techniques. These three chapters are based on an article published in IEEE Surveys and Tutorials [78]. Chapter 4, based on a paper published in Pervasive and Mobile Computing [79], introduces Centinela as a HAR system that considers physiological and acceleration signals. In that paper it was shown that ambulation activities can be recognized more accurately by incorporating physiological signals (e.g., heart rate, respiration rate, and skin temperature, among others) besides the tradition-

ally used acceleration signals. Only a few works have been reported in this matter [104] yet they were not successful because of the use of time-domain statistical features, which do not describe the morphological interrelationship among vital sign data. Instead, to take advantage of physiological signals, this chapter shows the application of structural feature extraction methods, such as polynomial regression and transient features, in conjunction with classifier ensembles. The system design, evaluation, and main results are also part of this chapter. Chapter 5 is devoted to Vigilante, a mobile framework to enable real-time activity recognition. Vigilante is presented as a new real-time human activity recognition system under the Android mobile platform. Implementing activity recognition on a mobile device is advantageous in terms of response time (as some applications need to provide feedback to the user in real time), energy consumption (as raw data would not have to be sent to a server), robustness (because it would not depend on unreliable wireless communication links), and scalability (since the most complex computations would not have to be executed in the server). The evaluation demonstrates that Vigilante allows for significant energy savings while maintaining high accuracy and low response time. The chapter also describes the design of MECLA, the underlying library for mobile evaluation of classification algorithms. This chapter is based on a paper published in the IEEE CCNC Conference [77]. Chapter 6 presents the topic of multiple classifier systems as a way to improve the accuracy of the classification task. Here, the book presents two new strategies for decision selection and fusion in multiple classifier systems. The evaluation of seven base classifiers and eleven datasets demonstrates that the proposed methods significantly improve the classification accuracy, not only for HAR but also for other machine learning problems. Moreover, a new algorithm is proposed to select base classifiers in order to maximize the benefits of such strategies. Finally, Chapter 7 concludes Part I with a summary of findings and results and future research considerations.

Part II includes a practical guide to developing HAR applications in an Android-based smartphone. Chapter 8 presents a brief introduction to the Android platform. Chapter 9 guides the reader through the procedure to download and install all the necessary tools to set up the software development environment. In addition, it presents how to build the typical Hello World application and run it in a software device (emulator) and in a real smartphone. Chapter 10 shows how to connect and obtain data from some of the most common sensors available in your smartphone. Chapter 11 shows how to connect your Android device with external sensors using the Bluetooth interface. Chapter 12 explains the different alternatives available in Android to store and retrieve the data. Chapters 13 and 14 conclude the book, showing how to perform the feature extraction and evaluate classification algorithms in an Android smartphone.

Chapter 2

Human Activity Recognition

This chapter explains the fundamentals of human activity recognition in three main sections. Section 2.1 describes the most important aspects to consider when designing a HAR system. Section 2.2 follows with the description of the different human activity recognition methods. Finally, Section 2.3 covers the topic of HAR system evaluation.

2.1 Design issues

Any system attempting to recognize human activities is compelled to address at least eight main design issues: (1) *definition of the activity set*, (2) *selection of attributes and sensors*, (3) *obtrusiveness*, (4) *data collection protocol*, (5) *recognition performance*, (6) *energy consumption*, (7) *processing*, and (8) *user flexibility*. The main aspects and solutions related to each one of them are analyzed next. These design issues will be later referred to in Section 3.1 to qualitatively analyze the state-of-the-art HAR systems.

2.1.1 Definition of the activity set

The design of any HAR system depends on the activities to be recognized. In fact, changing the activity set A immediately turns a given HARP into a completely different problem. From the literature, seven groups of activities can be distinguished. These groups and the individual activities that belong to each group are summarized in Table 2.1.

The nature of the activities has a direct relation with the required sensors. For example, ambulation activities such as *walking* and *running* could be determined using accelerometers. Others such as *talking* might need a microphone whereas *swallowing* would require specialized sensors on the individual's throat [36].

TABLE 2.1: Types of activities recognized by state-of-the-art HAR systems © 2012 IEEE. Reprinted, with permission, from [78].

Group	Activities
Ambulation	Walking, running, sitting, standing still, lying, climbing stairs, descending stairs, riding escalator, and riding elevator.
Transportation	Riding a bus, cycling, and driving.
Phone usage	Text messaging, making a call.
Daily activities	Eating, drinking, working at the PC, watching TV, reading, brushing teeth, stretching, scrubbing, and vacuuming
Exercise/fitness	Rowing, lifting weights, spinning, Nordic walking, and doing push ups.
Military	Crawling, kneeling, situation assessment, and opening a door.
Upper body	Chewing, speaking, swallowing, sighing, and moving the head.

2.1.2 Selection of attributes and sensors

Four groups of attributes are measured using wearable sensors in a HAR context: *environmental attributes*, *acceleration*, *location*, and *physiological signals*.

2.1.2.1 Environmental attributes

These attributes, such as temperature, humidity, audio level, etc., are intended to provide context information describing the individual's surroundings. If the *audio level* and *light intensity* are *fairly low*, for instance, the subject may be *sleeping*. Various existing systems have utilized microphones, light sensors, humidity sensors, and thermometers, among others [81, 88]. Those sensors alone, though, might not provide sufficient information, as individuals can perform each activity under diverse contextual conditions in terms of weather, audio loudness, or illumination. Therefore, environmental sensors are generally accompanied by accelerometers and other sensors [114].

2.1.2.2 Acceleration

Triaxial accelerometers are perhaps the most broadly used sensors to recognize ambulation activities (e.g., *walking*, *running*, *lying*, etc.) [60, 59, 57, 58, 35, 25]. Accelerometers are inexpensive, require relatively low power [95], and are embedded in most of today's smartphones. Several papers have reported high recognition accuracy 92% [60], 95% [72], 97% [58], and up to 98% [71], under different evaluation methodologies. However, other daily activities such as *eating*, *working at a computer*, or *brushing teeth*, are confusing from the ac-

celeration point of view. For instance, eating might be confused with brushing teeth due to arm motion [81].

The impact of the sensor specifications has also been analyzed. In fact, Maurer et al. [81] studied the behavior of the recognition accuracy as a function of the accelerometer sampling rate (which lies between 10 Hz [114] and 100 Hz [69]). Interestingly, they found that no significant gain in accuracy is achieved above 20 Hz for ambulation activities. In addition, the amplitude of the accelerometers varies from ±2g [81], up to ±6g [71], yet ±2g was shown to be sufficient to recognize ambulation activities [81]. The placement of the accelerometer is another important point of discussion: He et al. [60] found that the best place to wear the accelerometer is inside the trousers pocket. Instead, other studies suggest that the accelerometer should be placed in a bag carried by the user [81], on the belt [50], or on the dominant wrist [104]. At the end, the optimal position to place the accelerometer depends on the application and the type of activities to be recognized.

2.1.2.3 Location

The Global Positioning System (GPS) enables all sort of *location based services*. Current smartphones are equipped with GPS devices, making this sensor very convenient for context-aware applications, including the recognition of the user's transportation mode [95]. The place where the user is can also be helpful to infer their activity using ontological reasoning [96]. As an example, if a person is at a *park*, they are probably not *brushing their teeth* but might be *running* or *walking*. And, information about places can be easily obtained by means of the Google Places Web Service [6], among other tools. However, GPS devices do not work well indoors and they are relatively expensive in terms of energy consumption, especially for real-time tracking applications [95]. Consequently, this sensor is usually employed along with accelerometers [96]. Finally, location data have privacy issues because users are not always willing to be tracked. Encryption, obfuscation, and anonymization are some of the techniques available to ensure privacy in location data [63, 39, 109].

2.1.2.4 Physiological signals

Vital signs data (e.g., heart rate, respiration rate, skin temperature, skin conductivity, ECG, etc.) have also been considered in a few works [114]. Tapia et al. [104] proposed an activity recognition system that combines data from five triaxial accelerometers and a heart rate monitor. However, they concluded that the heart rate is not useful in a HAR context because after performing physically demanding activities (e.g., running) the heart rate remains at a high level for a while, even if the individual is lying or sitting. In this book, it was shown that, by means of structural feature extraction, vital signs can be exploited to improve recognition accuracy. Now, in order to measure physiological signals, additional sensors would be required, thereby increasing the system cost and increasing the level of obtrusiveness [88].

2.1.3 Obtrusiveness

To be successful in practice, HAR systems should not require the user to wear many sensors nor interact too often with the application. Nevertheless, the more sources of data available, the richer the information that can be extracted from the measured attributes. There are systems which require the user to wear four or more accelerometers [25, 48, 104], or carry a heavy rucksack with recording devices [88]. These configurations may be uncomfortable, invasive, expensive, and hence not suitable for activity recognition. Other systems are able to work with rather unobtrusive hardware. For instance, a sensing platform that can be worn as a sport watch is presented in [81]. Finally, the systems introduced in [95, 26] recognize activities with a cellular phone only.

2.1.4 Data collection protocol

The procedure followed by the individuals while collecting data is critical in any HAR. In 1999, Foerster et al. [49] demonstrated 95.6% of accuracy for ambulation activities in a controlled data collection experiment, but in a naturalistic environment (i.e., outside of the laboratory), the accuracy dropped to 66%! The number of individuals and their physical characteristics are also crucial factors in any HAR study. A comprehensive study should consider a large number of individuals with diverse characteristics in terms of gender, age, height, weight, and health conditions. This is with the purpose of ensuring flexibility to support new users without the need of collecting additional training data.

2.1.5 Recognition performance

The performance of a HAR system depends on several aspects, such as (1) the activity set, (2) the quality of the training data, (3) the feature extraction method, and (4) the learning algorithm. In the first place, each set of activities brings a completely different pattern recognition problem. For example, discriminating among *walking*, *running*, and *standing still* [80] turns out to be much easier than incorporating more complex activities such as *watching TV*, *eating*, *walking upstairs*, and *walking downstairs* [25]. Secondly, there should be a sufficient amount of training data, similar to the expected testing data. Finally, a comparative evaluation of several learning methods is desirable as each dataset exhibits distinct characteristics that can be either beneficial or detrimental for a particular method. Such interrelationship among datasets and learning methods can be very hard to analyze theoretically, which accentuates the need of an experimental study. In order to quantitatively understand the recognition performance, some standard metrics are used, e.g., *accuracy, recall, precision, F-measure, Kappa statistic,* etc. These metrics will be discussed later in Section 2.2.

2.1.6 Energy consumption

Context-aware applications rely on mobile devices — such as sensors and smartphones — which are generally energy constrained. In most scenarios, extending the battery life is a desirable feature, especially for medical and military applications that are compelled to deliver critical information. Surprisingly, most HAR schemes do not formally analyze energy expenditures, which are mainly due to processing, communication, and visualization tasks. Communication is often the most expensive operation, so the designer should minimize the amount of transmitted data. In most cases, short range wireless networks (e.g., Bluetooth or Wi-Fi) should be preferred over long range networks (e.g., cellular network or WiMAX) as the former require lower power. Some typical energy saving mechanisms are *data aggregation* and *compression* yet they involve additional computations that may affect the application performance. Another approach is to carry out feature extraction and classification in the integration device, so that raw signals would not have to be continuously sent to the server [77, 96]. This will be discussed in Section 2.1.7. Finally, since all sensors may not be necessary simultaneously, turning some of them off or reducing their sampling/transmission rate is very convenient to save energy. For example, if the user's activity is *sitting* or *standing still*, the GPS sensor may be turned off [95].

2.1.7 Processing

Another important point of discussion is where the recognition task should be done, whether in the server or in the integration device. On one hand, a server is expected to have huge processing, storage, and energy capabilities, allowing to incorporate more complex methods and models. On the other hand, a HAR system running on a mobile device should substantially reduce energy expenditures, as raw data would not have to be continuously sent to a server for processing. The system would also become more robust and responsive because it would not depend on unreliable wireless communication links, which may be unavailable or error prone; this is particularly important for medical or military applications that require real-time decision making. Finally, a mobile HAR system would be more scalable since the server load would be alleviated by the locally performed feature extraction and classification computations. However, implementing activity recognition in mobile devices becomes challenging because they are still constrained in terms of processing, storage, and energy. Hence, feature extraction and learning methods should be carefully chosen to guarantee a reasonable response time and battery life. For instance, classification algorithms such as Instance Based Learning [32] and Bagging [111] are very expensive in their evaluation phase, which makes them not convenient for HAR.

2.1.8 User flexibility

There is an open debate on the design of any activity recognition model. Some authors claim that, as people perform activities in a different manner (due to age, gender, weight, and so on), a *specific* recognition model should be built for each individual [28]. This implies that the system should be re-trained for each new user. Other studies rather emphasize the need of a *monolithic* recognition model, flexible enough to work with different users. Consequently, two types of analyses have been proposed to evaluate activity recognition systems: *subject-dependent* and *subject-independent* evaluations [104]. In the first one, a classifier is trained and tested for each individual with his/her own data and the average accuracy for all subjects is computed. In the second one, only one classifier is built for all individuals and the evaluation is carried out by cross validation or leave-one-individual-out analysis. It is worth highlighting that, in some cases, it would not be convenient to train the system for each new user, especially when (1) there are too many activities, (2) some activities are not desirable for the subject to carry out (e.g., falling downstairs), or (3) the subject would not cooperate with the data collection process (e.g., patients with dementia and other mental pathologies). On the other hand, an elderly lady would surely walk quite differently than a ten-year-old boy, thereby challenging a single model to recognize activities regardless of the subject's characteristics. A solution to the dichotomy of the monolithic vs. particular recognition model is to create groups of users with similar characteristics. Additional design considerations related to this matter will be discussed in Section 7.2.

2.2 Activity recognition methods

Section 1.2 showed that, to enable the recognition of human activities, raw data have to first pass through the process of feature extraction. Then, the recognition model is built from the set of feature instances by means of machine learning techniques. Once the model is trained, unseen instances (i.e., time windows) can be evaluated in the recognition model, yielding a prediction on the performed activity. Next, the most noticeable approaches in *feature extraction* and *learning* are covered.

2.2.1 Feature extraction

Human activities are performed during relatively long periods of time (in the order of seconds or minutes) compared to the sensors' sampling rate (which can be up to 250 Hz). Besides, a single sample on a specific time instant (e.g., the Y-axis acceleration is 2.5g, or the heart rate is 130 bpm) does not provide

sufficient information to describe the performed activity. Thus, activities need to be recognized in a time window basis rather than in a sample basis. Now, the question is: how can two given time windows be compared? It would be nearly impossible for the signals to be exactly identical, even if they come from the same subject performing the same activity. This is the main motivation for applying *feature extraction* (FE) methodologies to each time window: filtering relevant information and obtaining quantitative measures that allow signals to be compared.

In general, two approaches have been proposed to extract features from time series data: *statistical* and *structural* [87]. Statistical methods, such as the Fourier transform and the Wavelet transform, use quantitative characteristics of the data to extract features, whereas structural approaches take into account the morphological interrelationship among data. The criterion to choose either of these methods is certainly subject to the nature of the given signal.

Figure 2.1 displays the process to transform the raw time series dataset — which can be from acceleration, environmental variables, or vital signs — into a set of *feature vectors*. w is the window consecutive; s_i is the sampling rate for the group of sensors i, where sensors in the same group have the same sampling rate; and f_i is each of the extracted features. Each instance in the processed dataset corresponds to the set of features computed from an entire window in the raw dataset.

The next sections will cover the most common FE techniques for each of the measured attributes, i.e., acceleration, environmental signals, and vital signs. GPS data are not considered in this section since they are mostly used to compute the speed [88, 95] or include some knowledge about the place where the activity was performed [96].

2.2.1.1 Acceleration

Acceleration signals (see Figure 2.2) are highly fluctuating and oscillatory, which makes it difficult to recognize the underlying patterns using their raw values. Existing HAR systems based on accelerometer data employ statistical feature extraction and, in most of the cases, either time- or frequency-domain features. Discrete Cosine Transform (DCT) and Principal Component Analysis (PCA) have also been applied with promising results [58], as well as autoregressive model coefficients [60]. All these techniques are conceived to handle the high variability inherent to acceleration signals. Table 2.2 summarizes the feature extraction methods for acceleration signals. The definition of some of the most widely used features [35] are listed below for a given signal $Y = \{y_1, ..., y_n\}$.

- Central tendency measures such as the *arithmetic mean* $\bar{y}$ and the *root mean square* (RMS) (Equations 2.1 and 2.2).
- Dispersion metrics such as the *standard deviation* σ_y, the *variance* σ_y^2, and the *mean absolute deviation* (MAD) (Equations 2.3, 2.4, and 2.5).

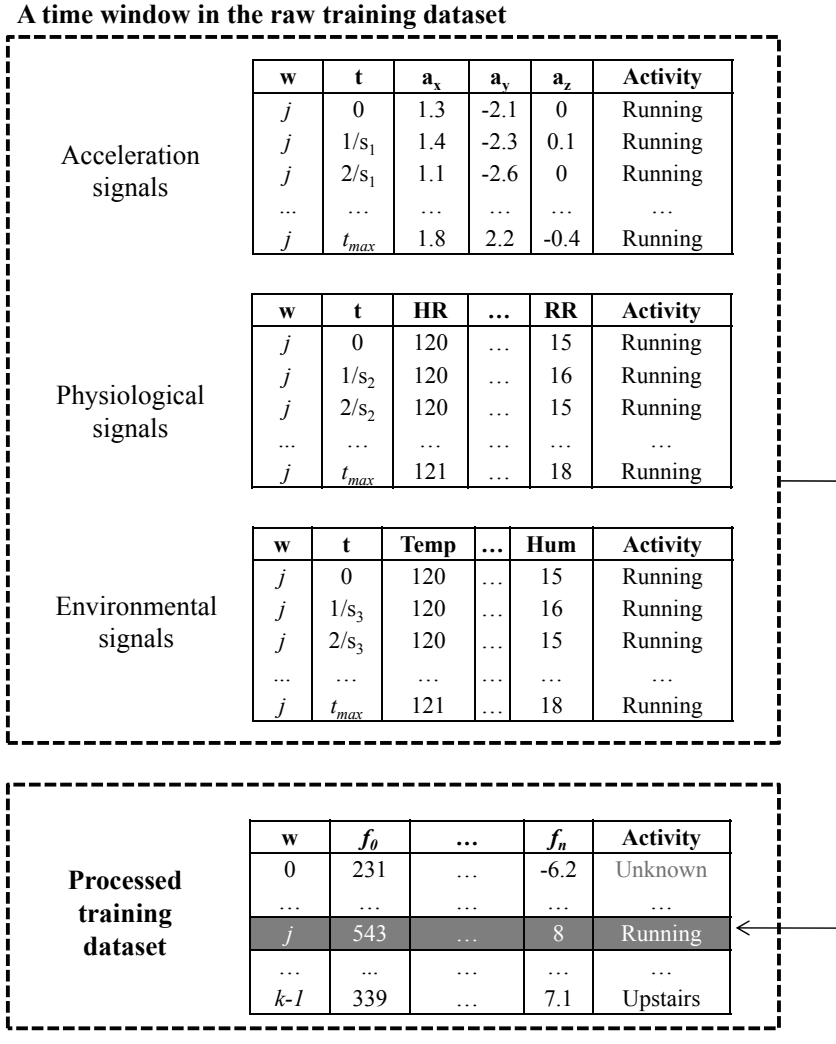

FIGURE 2.1: An example of the mapping from the raw dataset to the feature dataset © 2012 IEEE. Reprinted, with permission, from [78].

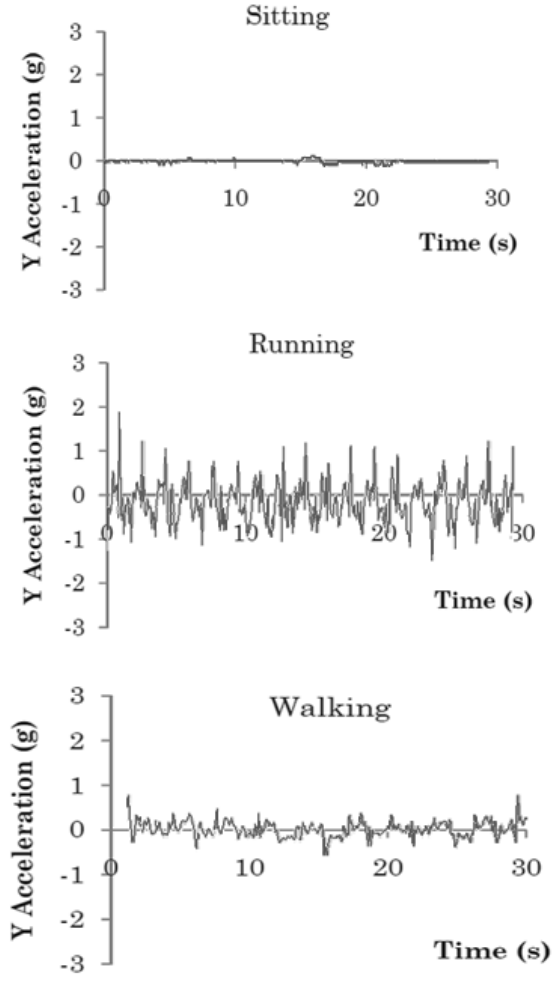

FIGURE 2.2: Acceleration signals for different activities © 2012 IEEE. Reprinted, with permission, from [78].

TABLE 2.2: Summary of feature extraction methods for acceleration signals © 2012 IEEE. Reprinted, with permission, from [78].

Group	Methods
Time domain	Mean, standard deviation, variance, interquartile range (IQR), mean absolute deviation (MAD), correlation between axes, entropy, and kurtosis [35, 25, 81, 88, 104, 48, 69].
Frequency domain	Fourier Transform (FT) [25, 35] and Discrete Cosine Transform (DCT) [22].
Others	Principal Component Analysis (PCA) [58, 22], Linear Discriminant Analysis (LDA) [35], Autoregressive Model (AR), and HAAR filters[57].

- Domain transform measures such as the *energy*, where F_i is the i-th component of the Fourier Transform of Y (Equation 2.6).

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{2.1}$$

$$RMS(Y) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} {y_i}^2} \tag{2.2}$$

$$\sigma_y = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} (y_i - \bar{y})^2} \tag{2.3}$$

$$\sigma_y^2 = \frac{1}{n-1}\sum_{i=1}^{n} (y_i - \bar{y})^2 \tag{2.4}$$

$$MAD(Y) = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} |y_i - \bar{y}|} \tag{2.5}$$

$$Energy(Y) = \frac{\sum_{i=1}^{n} {F_i}^2}{n} \tag{2.6}$$

2.2.1.2 Environmental variables

Environmental attributes, along with acceleration signals, have been used to enrich context awareness. For instance, the values from air pressure and light intensity are helpful to determine whether the individual is outdoors or indoors [37]. Also, audio signals are useful to conclude that the user is having a conversation rather than listening to music [88]. Table 2.3 summarizes the feature extraction methods for environmental attributes.

TABLE 2.3: Summary of feature extraction methods for environmental variables © 2012 IEEE. Reprinted, with permission, from [78].

Attribute	**Features**
Altitude	Time-domain [88]
Audio	Speech recognizer [90]
Barometric pressure	Time-domain and frequency-domain [37]
Humidity	Time-domain [88]
Light	Time-domain [82] and frequency-domain [88]
Temperature	Time-domain [88]

2.2.1.3 Physiological signals

Since the incorporation of physiological signals is one of the main aspects of this book, a detailed description of feature extraction from vital signs is provided in Chapter 4.

2.2.1.4 Selection of the window length

In accordance with Definition 1.2, dividing the measured time series in time windows is a convenient solution to relax the HAR problem. Therefore, a key factor is the selection of the window length because the computational complexity of any FE method depends on the number of samples. Having rather short windows may enhance the FE performance, but would entail higher overhead due to the recognition algorithm being triggered more frequently. Besides, short time windows may not provide sufficient information to fully describe the performed activity. Conversely, if the windows are too long, there might be more than one activity within a single time window [95]. Different window lengths have been used in the literature: 0.08s [28], 1s [95], 1.5s [36], 3s [84], 5s [22], 7s [83], 12s [79], or up to 30s [104]. Of course, this decision is conditioned to the activities to be recognized and the measured attributes. The heart rate signal, for instance, required 30s time windows in [104]. Instead, for activities such as swallowing, 1.5s time windows were employed.

Time windows can also be either overlapping [25, 104, 92] or disjoint [95], [36], [84], [22]. Overlapping time windows are intended to handle transitions more accurately, although using small non-overlapping time windows, misclassifications due to transitions are negligible.

2.2.1.5 Feature selection

Some features in the processed dataset might contain redundant or irrelevant information that can negatively affect the recognition accuracy. Then, implementing techniques for selecting the most appropriate features is a suggested practice to reduce computations and simplify the learning models. The *Bayesian Information Criterion* (BIC) and the *Minimum Description Length* (MDL) have been widely used for general machine learning problems.

In HAR, a common method is the *Minimum Redundancy and Maximum Relevance* (MRMR) [89], utilized in [66]. In that work, the *minimum mutual information between features* is used as criteria for minimum redundancy and the *maximal mutual information between the classes and features* is used as criteria for maximum relevance. In contrast, Maurer et al. [82] applied a *Correlation-based Feature Selection (CFS) approach* [56], taking advantage of the fact that this method is built in WEKA [18]. CFS works under the assumption that features should be highly correlated with the given class but uncorrelated with each other. Iterative approaches have also been evaluated to select features. Since the number of feature subsets is $O(2^n)$, for n features, evaluating all possible subsets is usually not computationally feasible. Hence, metaheuristic methods such as multiobjective evolutionary algorithms have been employed to explore the space of possible feature subsets [40].

2.2.2 Learning

In a machine learning context, patterns are to be discovered from a set of given examples or observations denominated *instances*. Such an input set is called a *training set*. In our specific case, each instance is a feature vector extracted from signals within a time window. The examples in the training set may or may not be labeled, i.e., associated to a known class (e.g., walking, running, etc.). In some cases, labeling data is not feasible because it may require an expert to manually examine the examples and assign a label based upon their experience. This process is usually tedious, expensive, and time consuming in many data mining applications.

There exist two learning approaches, namely *supervised* and *unsupervised* learning, which deal with labeled and unlabeled data, respectively. Since a human activity recognition system should return a label such as *walking, sitting, running*, etc., most HAR systems work in a supervised fashion. Indeed, it might be very hard to discriminate activities in a completely unsupervised context. Some other systems work in a semi-supervised fashion allowing part of the data to be unlabeled.

2.2.2.1 Supervised learning

Labeling sensed data from individuals performing different activities is a relatively easy task. Some systems [104, 82] store sensor data in a non-volatile medium while a person from the research team supervises the collection process and manually registers activity labels and time stamps. Other systems feature a mobile application that allows the user to select the activity to be performed from a list [79]. In this way, each sample is matched to an activity label, and then stored in the server.

Supervised learning — referred to as *classification* for discrete-class problems — has been a very productive field, bringing about a great number of

TABLE 2.4: Classification algorithms used by state-of-the-art human activity recognition systems © 2012 IEEE. Reprinted, with permission, from [78].

Type	**Classifiers**
Decision tree	C4.5, ID3 ([25, 48, 66, 81])
Bayesian	Naïve Bayes and Bayesian Networks ([104, 25, 66, 77])
Instance Based	k-nearest neighbors ([81, 66])
Neural Networks	Multilayer Perceptron ([94])
Domain transform	Support Vector Machines ([58, 59, 60])
Fuzzy Logic	Fuzzy Basis Function, Fuzzy Inference System ([69, 35, 27])
Regression methods	MLR, ALR ([96, 79])
Markov models	Hidden Markov Models, Conditional Random Fields ([116, 110])
Classifier ensembles	Boosting and Bagging ([84, 79])

algorithms. Table 2.4 summarizes the most important classifiers in Human Activity Recognition and their description is included below.

- *Decision Trees* (DT) build a hierarchical model in which attributes are mapped to nodes and edges represent the possible attribute values. Each branch from the root to a leaf node is a classification rule. C4.5 is perhaps the most widely used decision tree classifier and is based on the concept of information gain to select which attributes should be placed in the top nodes [93]. Decision trees can be evaluated in $O(\log n)$ for n attributes, and usually generate models that are easy to understand by humans.

- *Bayesian* methods calculate posterior probabilities for each class using estimated conditional probabilities from the training set. The *Bayesian Network* (BN) [23] classifier and *Naïve Bayes* (NB) [115] classifier — which is a specific case of BN — are the principal exponents of this family of classifiers. A key issue in Bayesian Networks is the topology construction, as it is necessary to make assumptions on the independence among features. For instance, the NB classifier assumes that all features are conditionally independent given a class value, yet such assumption does not hold in many cases. As a matter of fact, acceleration signals are highly correlated, as well as physiological signals such as heart rate, respiration rate, and ECG amplitude.

- *Instance-Based Learning* (IBL) [111] methods classify an instance based upon the most *similar* instance(s) in the training set. For that purpose, they define a distance function to measure similarity between each pair of instances. This makes IBL classifiers quite expensive in their evaluation phase as each new instance to be classified needs to be compared to the

entire training set. Such high cost in terms of computation and storage, usually makes IBL models not convenient to be implemented in a mobile device.

- *Support Vector Machines* (SVM) [42] and *Artificial Neural Networks* (ANN) [52] have also been broadly used in HAR although they do not provide a set of rules understandable to humans. Instead, knowledge is hidden within the model, which may hinder the analysis and incorporation of additional reasoning. SVMs rely on kernel functions that project all instances to a higher dimensional space with the aim of finding a linear decision boundary (i.e., a hyperplane) to partition the data. Neural networks replicate the behavior of biological neurons in the human brain, propagating activation signals and encoding knowledge in the network links. Besides, ANNs have been shown to be universal function approximators. The high computational cost and the need for large amount of training data are two common drawbacks of neural networks.

- *Classifier Ensembles* (CE) combine the output of several classifiers to improve classification accuracy. Some examples are *bagging*, *boosting*, and *stacking* [111]. Classifier ensembles are clearly more expensive, computationally speaking, as they require several models to be trained and evaluated. Chapter 6 elaborates on these methods.

2.2.2.2 Semi-supervised learning

Relatively few approaches have implemented activity recognition in a semi-supervised fashion, i.e., having part of the data without labels [103, 102, 21, 55, 64]. In practice, annotating data might be difficult in some scenarios, particularly when the granularity of the activities is very high or the user is not willing to cooperate with the data collection process. Since semi-supervised learning is a minority in HAR, there are no standard algorithms or methods, but each system implements its own approach. Section 3.4 provides more details on the state of the art of semi-supervised activity recognition approaches.

2.3 Evaluating HAR systems

2.3.1 Evaluation methodologies

When evaluating a machine learning algorithm, the training and testing datasets should be disjoint. This is with the aim of assessing how effective the algorithm is to model unseen data. A very intuitive approach is called *random split* and it simply divides the entire dataset in two partitions: one for training and the other one for testing — usually, two thirds of the data are for training

and the remaining one third is for testing. However, a random split is highly biased by the dataset partition. If instances in any of the training or testing sets are concentrated in a particular feature space subregion, the evaluation metrics would not reflect the actual performance of the classifier. Therefore, a more robust approach is the *cross validation*. In a k-fold cross validation, the dataset is divided into k equally-sized folds. In the first iteration, the very first fold is used as the testing set while the remaining $k-1$ folds constitute the training set. The process is repeated k times, using each fold as a testing set and the remaining ones for training. In the end, the evaluation metrics (e.g., accuracy, precision, recall, etc.) are averaged out over all iterations. In practice, a 10-fold cross validation is the most widely accepted methodology to calculate the accuracy of a certain classifier (see Figure 2.3). Yet, if the goal is to compare two classifiers in order to choose the most accurate one, a 5×2-fold cross validation with a paired t-test is recommended [45]. This is nothing but repeating a 2-fold cross validation five times with different dataset partitions, which is often achieved by using different seeds for the random number generator. The result of a 5×2-fold cross validation is shown in Table 2.5, where the accuracies a_i and b_i are for classifiers A and B, respectively, throughout the five repetitions. The next step is to apply a statistical paired t-test to find the most accurate classifier if there is significant statistical difference among their accuracies. The null hypothesis is that both classifiers A and B have the same accuracy — or conversely, the same error rate. The $\tilde{t}$ statistic is defined in Equation 2.7 as follows:

$$\tilde{t} = \frac{p_1^{(1)}}{\sqrt{\frac{1}{5}\sum_{i=1}^{5} s_i^2}} \tag{2.7}$$

where $p_i^{(j)}$ is the difference between the accuracies of both classifiers in the j-th iteration for $1 \leq j \leq 2$ and the i-th replication; $s_i^2 = (p_i^{(1)} - \bar{p})^2 + (p_i^{(2)} - \bar{p})^2$ is the estimated variance from the i-th replication for $1 \leq i \leq 5$; and $\bar{p}$ is the average of $p_i^{(1)}$ and $p_i^{(2)}$. Under the null hypothesis, $\tilde{t}$ follows the Student's t distribution with five degrees of freedom. It has been shown that the 5×2-fold cross validation with the paired t-test is more powerful than the non-parametric McNemar's test and provides a better measure of the variations due to the choice of the training set [45].

2.3.2 Evaluation metrics in machine learning

In general, the selection of a classification algorithm for HAR has been merely supported by empirical evidence. The vast majority of the studies use cross validation with statistical tests to compare classifiers' performance for a particular dataset. The classification results for a particular method can be organized in a *confusion matrix* $M_{n\times n}$ for a classification problem with n classes. This is a matrix such that the element M_{ij} is the number of instances

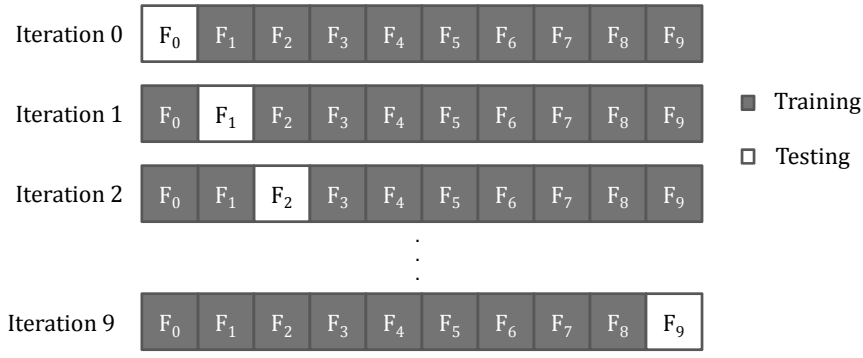

FIGURE 2.3: The iterations of a 10-fold cross validation.

TABLE 2.5: 5×2-fold cross validation.

Seed	A	B
s_1	a_1	b_1
s_2	a_2	b_2
s_3	a_3	b_3
s_4	a_4	b_4
s_5	a_5	b_5

from class i that were classified as class j. The following values can be obtained from the confusion matrix in a binary classification problem:

- *True Positives* (TP): The number of positive instances that were classified as positive.
- *True Negatives* (TN): The number of negative instances that were classified as negative.
- *False Positives* (FP): The number of negative instances that were classified as positive.
- *False Negatives* (FN): The number of positive instances that were classified as negative.

The *accuracy* is the most standard metric to summarize the overall classification performance for all classes and it is defined as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.8}$$

The *precision* — often referred to as *positive predictive value* — is the ratio of correctly classified positive instances to the total number of instances

classified as positive:

$$Precision = \frac{TP}{TP + FP} \tag{2.9}$$

The *recall*, also called *true positive rate*, is the ratio of correctly classified positive instances to the total number of positive instances:

$$Recall = \frac{TP}{TP + FN} \tag{2.10}$$

The *F-measure* combines precision and recall in a single value:

$$F - measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.11}$$

Finally, the *false positive rate* (FPR) and the *false negative rate* (FNR) are defined as follows:

$$FPR = \frac{FP}{TN + FP} \tag{2.12}$$

$$FNR = \frac{FN}{TP + FN} \tag{2.13}$$

Although defined for binary classification, these metrics can be generalized for a problem with n classes. In such case, an instance could be positive or negative according to a particular class, e.g., positives might be all instances of *running* while negatives would be all instances other than *running*.

2.3.3 Machine learning tools

The Waikato Environment for Knowledge Analysis (WEKA) [18] is certainly the best known tool in the machine learning research community. It contains implementations of a number of learning algorithms and it allows to easily evaluate them for a particular dataset using cross validation and random split, among others. WEKA also offers a Java API that facilitates the incorporation of new learning algorithms and evaluation methodologies on top of the pre-existing framework. One of the limitations of WEKA [18] and other machine learning platforms such as the Java Data Mining (JDM) platform [11] is that they are not optimized to work on current mobile platforms. The book introduces MECLA [77], a mobile platform for the evaluation of classification algorithms under the Android platform.

Another widely-used machine learning tool in both industry and academy is R [13]. R not only compiles machine learning algorithms but also provides statistical modeling tools, feature extraction, time series analysis, data manipulation, and visualization in a fully programmable environment.

Chapter 3

State of the Art in HAR Systems

The first works on *human activity recognition* (HAR) date back to the late '90s [99, 49]. However, there are still a number of opportunities for the development of new techniques to improve accuracy, efficiency, and pervasiveness. This chapter surveys the state-of-the-art in human activity recognition systems using wearable sensors. In addition, it also includes a qualitative evaluation of these systems according to several aspects, such as the type of activities that they recognize, the sensors utilized, their energy consumption and computational complexity, the accuracy, and others.

3.1 Evaluation of HAR systems

In this book, the two-level taxonomy proposed in [78] to categorize HAR systems (see Figure 3.1) has been adopted. The first level relates to the learning approach, which can be either *supervised* or *semi-supervised*. In the second level, according to the response time, supervised approaches can work either *online* or *offline*. The former provide immediate feedback on the performed activities. The latter either need more time to recognize activities due to high computational demands, or are intended for applications that do not require real-time feedback. To the best of our knowledge, current semi-supervised systems have been implemented and evaluated offline. This taxonomy has been adopted because the systems within each class have very different purposes and associated challenges and should be evaluated separately. For instance, a very accurate fully supervised approach might not work well in a semi-supervised scenario, whereas an effective offline system may not be able to run online due to processing constraints. Furthermore, there is a significant number of systems that fall into each group, which also favors the comparison and analysis by means of the proposed taxonomy.

This chapter also presents a qualitative evaluation of the different HAR systems. The evaluation considers the activities included in Table 2.1 and the design issues presented in Section 2.1. It, therefore, encompasses the following aspects:

- Recognized activities.

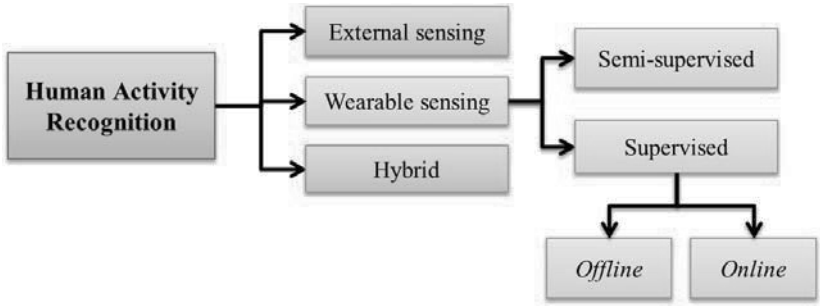

FIGURE 3.1: Taxonomy of human activity recognition systems © 2012 IEEE. Reprinted, with permission, from [78].

- Type of sensors and the measured attributes.
- Integration device.
- Level of obtrusiveness, which could be *low*, *medium*, or *high*.
- Type of data collection protocol, which could be either a *controlled* or a *naturalistic* experiment.
- Level of energy consumption, which could be *low*, *medium*, or *high*.
- User flexibility level, which could be either *user-specific* or *monolithic*.
- Feature extraction method(s).
- Learning algorithm(s).
- Overall accuracy for all activities.

3.2 Online HAR systems

Applications of online activity recognition systems can be easily visualized. In healthcare, continuously monitoring patients who have physical or mental pathologies becomes crucial for their protection, safety, and recovery. Likewise, interactive games or simulators may enhance the user's experience by considering activities and gestures. In the following subsections, the most relevant online HAR systems found in the literature are reviewed.

3.2.1 eWatch

Maurer et al. [81] introduced *eWatch* as an online activity recognition system which embeds sensors and a microcontroller within a device that can be worn as a sports watch. Four sensors are included, namely an accelerometer, a light sensor, a thermometer, and a microphone. These are passive sensors and, as they are embedded in the device, no wireless communication is needed; thus, eWatch is very energy efficient. Using a C4.5 decision tree and time-domain feature extraction, the overall accuracy was up to 92.5% for six ambulation activities, although they achieved less than 70% for activities such as *descending* and *ascending*. The execution time for feature extraction and classification is less than 0.3 ms, which makes the system very responsive. However, in eWatch, data were collected under controlled conditions, i.e., a lead experimenter supervised and gave specific guidelines to the subjects on how to perform the activities [81]. Section 2.1.4 describes the disadvantages of this approach.

3.2.2 Vigilante

Vigilante [77] is also presented in this book in Chapter 5, as a mobile application for real-time human activity recognition using the Android platform. The Zephyr's BioHarness BT [19] chest sensor strap was used to measure acceleration and physiological signals such as heart rate, respiration rate, breath waveform amplitude, and skin temperature, among others. Statistical time- and frequency-domain features were extracted from acceleration signals while transient features and linear regression were applied to physiological signals. The C4.5 and the Additive Logistic Regression classifiers were employed to recognize five activities with an overall accuracy of up to 96.8%. The application can run for up to 12.5 continuous hours with a response time of no more than 8% of the window length. Unlike other approaches, Vigilante was evaluated completely online to provide more realistic results. Vigilante is moderately energy efficient because it requires permanent Bluetooth communication between the sensor strap and the smartphone.

3.2.3 Tapia

This system [104] recognizes 17 ambulation and gymnasium activities such as *lifting weights*, *rowing*, *doing push-ups*, etc., with different intensities (a total of 30 activities). A comprehensive study was carried out, including 21 participants and both subject-dependent and subject-independent studies. The average classification accuracy was reported to be 94.6% for subject-dependent analysis whereas a 56% of accuracy was reached in the subject-independent evaluation. If intensities are not considered, the overall subject-independent accuracy is 80.6%. This system works with very obtrusive hardware, i.e., five accelerometers were placed on the user's dominant arm and wrist, hip, thigh,

and ankle, as well as a heart rate monitor on the chest. Besides, all these sensors require wireless communication, involving high energy consumption. Finally, the integration device is a laptop, which allows for better processing capabilities, but prevents portability and pervasiveness.

3.2.4 ActiServ

In 2010, Berchtold et al. introduced *ActiServ* as an activity recognition service for mobile phones [26, 28]. The system was implemented on the Neo FreeRunner phone. They make use of a fuzzy inference system to classify ambulation and phone activities based on the signals given by the phone's accelerometer only. This makes ActiServ a very energy efficient and portable system. The overall accuracy varies between 71% and 97%. However, in order to reach the top accuracy level, the system requires a runtime duration in the order of days! When the algorithms are executed to meet a real-time response time, the accuracy drops to 71%. ActiServ can also reach up to 90% after personalization, in other words, a subject-dependent analysis. From the reported confusion matrices, the activity labeled as *walking* was often confused with *cycling*, while *standing* and *sitting* could not be differentiated when the cellphone's orientation was changed.

3.2.5 COSAR

Riboni et al. [96] presented COSAR, a framework for context-aware activity recognition using statistical and ontological reasoning under the Android platform. The system recognizes ambulation activities as well as *brushing teeth*, *strolling*, and *writing on a blackboard*. COSAR gathers data from two accelerometers, one in the phone and another on the individual's wrist, as well as from the cellphone's GPS. Since COSAR makes use of the GPS sensor, it was cataloged as a moderately energy efficient system. COSAR uses an interesting concept of *potential activity matrix* to filter activities based upon the user's location. For instance, if the individual is in the *kitchen*, he or she is probably not *cycling*. Another contribution is the statistical classification of activities with a historical variant. For example, if the predictions for the last five time windows were {*jogging*, *jogging*, *walking*, *jogging*, *jogging*}, the third window was likely a misclassification (e.g., due to the user performing some atypical movement) and the algorithm should automatically correct it. However, this introduces an additional delay to the system's response, according to the number of windows analyzed for the historical variant. The overall accuracy was roughly 93% though, in some cases, *standing still* was confused with *writing on a blackboard*, as well as *hiking up* with *hiking down*.

3.2.6 Kao

Kao et al. [69] presented a portable device for online activity detection. A triaxal accelerometer is placed on the user's dominant wrist, sampling at 100 Hz. They apply time domain features and the *Linear Discriminant Analysis (LDA)* to reduce the dimension of the feature space. Then, a *Fuzzy Basis Function* learner — which uses fuzzy *If-Then rules* — classifies the activities. An overall accuracy of 94.71% was reached for seven activities: *brushing teeth*, *hitting*, *knocking*, *working at a PC*, *running*, *walking*, and *swinging*. The system reports an average response time of less than 10 ms, which supports its feasibility. All the computations are done in an embedded system that should be carried by the user as an additional device. This has some disadvantages with respect to a mobile phone in terms of portability, comfort, and cost. Moreover, the size of the time window was chosen to be 160 ms. Given the nature of the recognized activities, this excessive granularity causes accidental movements when *swinging* or *knocking* may be confused with *running*, for instance. Such a small window length also induces more overhead due to the classification algorithm being triggered very often, and it is not beneficial for the feature extraction performance, as time domain features require $O(n)$ computations.

3.2.7 Other approaches

The system proposed by Brezmes et al. [32] features a mobile application for HAR under the Nokia platform. They used the k-nearest neighbors classifier, which is computationally expensive and not scalable for mobile phones, as it needs the entire training set — that can be fairly large — to be stored in the device. Besides, their system requires each new user to collect additional training data in order to obtain accurate results. Ermes et al. [48] developed an online system that reached 94% overall average accuracy but they only applied a subject-dependent evaluation. Besides, their data were collected from only three subjects, which inhibits flexibility to support new users.

3.2.8 Discussion

As it can be seen, each online HAR system has its own benefits and drawbacks. Thus, the selection of a particular approach for a real case study depends on the application requirements. If portability and obtrusiveness are the key issues, eWatch would be an appropriate option for ambulation activities. But if a broader set of activities needs to be recognized, COSAR should be considered, although it entails higher energy expenditures due to Bluetooth communication and the GPS sensor. The system proposed by Tapia et al. would be a better choice to monitor exercise habits yet it may be too obtrusive.

Overall, most systems exhibit similar accuracy levels (more than 92%),

but since each one works with a specific dataset and activity set, there is no significant evidence to argue that a system is more accurate than the others. Vigilante is the only approach that collects vital sign information, which opens a broader spectrum of applications for healthcare purposes. In addition, COSAR and Vigilante work under the Android platform — reported as the best-selling smartphone platform in 2010 by Canalys [8] — facilitating the deployment in current smartphones. The system cost is also an important aspect, especially when the application's aim is being scaled to hundreds or thousands of users. Vigilante, COSAR, eWatch, and the work of Kao et al. require specialized hardware such as sensors and embedded computers whereas ActiServ and the system proposed by Brezmes et al. only need a conventional cellular phone.

Next, Table 3.1 includes the abbreviations and acronyms used in the qualitative evaluation of the human activity recognition systems presented in this chapter and Table 3.2 presents the qualitative evaluation of the online HAR systems presented in this section.

3.3 Supervised offline systems

There are systems in which the user does not need to receive immediate feedback. For example, applications that analyze exercise and diet habits in patients with heart disease, diabetes, or obesity, as well as applications that estimate the number of calories burned after an exercise routine [24, 105] can work on an offline basis. Another example is the discovery of user commercial patterns for advertisement. For instance, if an individual performs exercise activities very frequently, the systems could be advertised on sports wear items. In all these cases, gathered data can be analyzed on a daily — or even weekly — basis to draw conclusions on the person's behaviors. In the following subsections, the most relevant offline HAR systems found in the literature are reviewed.

3.3.1 Parkka

The work of Parkka et al. [88] considers seven activities: *lying, rowing, riding a bike, standing still, running, walking*, and *Nordic walking*. Twenty-two signals were measured, including acceleration, vital signs, and environmental variables. This requires a number of sensors on the individual's chest, wrist, finger, forehead, shoulder, upper back, and armpit. The integration device is a compact computer placed in a 5 kg rucksack. Therefore, this system was cataloged as highly obtrusive. Time- and frequency-domain features were extracted from most signals while a speech recognizer [90] was applied to the audio signal. This entails not only high processing demands but also privacy

TABLE 3.1: List of abbreviations and acronyms © 2012 IEEE. Reprinted, with permission, from [78].

3DD	3D Deviation of the acceleration signals
ACC	Accelerometers
AMB	Ambulation activities (see Table 2.1)
ANN	Artificial Neural Network
ALR	Additive Logistic Regression classifier
AR	Autoregressive Model Coefficients
AV	Angular velocity
BN	Bayesian Network Classifier
CART	Classification And Regression Tree
DA	Daily activities (see Table 2.1)
DCT	Discrete Cosine Transform
DT	Decision Tree-Based Classifier
DTW	Dynamic Time Warping
ENV	Environmental sensors
FBF	Fuzzy Basis Function
FD	Frequency-domain features
GYR	Gyroscope
HMM	Hidden Markov Models
HRB	Heart Rate Beats above the resting heart rate
HRM	Heart Rate Monitor
HW	Housework activities (see Table 2.1)
KNN	k-Nearest Neighbors classifier
LAB	Laboratory controlled experiment
LDA	Linear Discriminant Analysis
LS	Least Squares algorithm
MIL	Military activities
MNL	Monolithic classifier (subject independent)
NAT	Naturalistic experiment
NB	The Naïve Bayes classifier
NDDF	Normal Density Discriminant Function
N/S	Not Specified
PCA	Principal Component Analysis
PHO	Activities related to phone usage (see Table 2.1)
PR	Polynomial Regression
RFIS	Recurrent Fuzzy Inference System
SD	Subject Dependent evaluation
SFFS	Sequential Forward Feature Selection
SI	Subject Independent evaluation
SMA	Signal Magnitude Area
SMCRF	Semi-Markovian Conditional Random Field
SPC	User-specific classifier (subject dependent)
SPI	Spiroergometry
TA	Tilt Angle
TD	Time-domain features
TF	Transient Features [79]
TR	Transitions between activities
UB	Upper body activities (see Table 2.1)
VS	Vital sign sensors

TABLE 3.2: State-of-the-art in online human activity recognition systems © 2012 IEEE. Reprinted, with permission, from [78].

Ref.	Activities (#)	Sensors	ID	Obtrusiveness	Type of Experim.	Energy	Flexibility	Processing	Features	Learning	Accuracy
Ermes [48]	AMB (5)	ACC (wrist, ankle, chest)	PDA	High	N/S	High	SPC	High	TD, FD	DT	94%
eWatch [81]	AMB (6)	ACC, ENV (wrist)	Custom	Low	LAB	Low	MNL	Low	TD, FD	C4.5, NB	94%
Tapia [104]	EXR (30)	HRM, ACC (5 places)	Laptop	High	LAB	High	Both	High	TD, FD, HB	C4.5, NB	86% (SD) 56% (SI) 56% (SI)
Vigilante [77]	AMB (5)	ACC and VS (chest)	Phone	Medium	NAT	Medium	MNL	Low	TD, FD, TF	C4.5	96.8% (SD) 92.6% (SI)
Kao [69]	AMB DA (7)	ACC (wrist)	Custom	Low	N/S	Medium	MNL	Low	TD, LDA	FBF	94.71%
Brezmes [32]	AMB (5)	ACC (phone)	Phone	Low	N/S	Low	SPC	High	TD, FD	KNN	80%
COSAR [96]	AMB DA (10) DA (10)	GPS ACC (watch, phone)	Phone	Low	NAT	Medium	MNL	Medium	TD	COSAR	93%
ActiServ [26, 28]	AMB, PHO (11)	ACC (phone)	Phone	Low	N/S	Low	SPC	High	$\bar{y}, \sigma_y^2$	RFIS	71%-98%

issues due to continuous recording of the user's speech. Three classification methods were evaluated, namely an *automatically generated decision tree*, a *custom decision tree* which introduces domain knowledge and visual inspection of the signals, as well as an *artificial neural network*. The results indicate that the highest accuracy was 86%, given by the first method, though activities such as as *rowing*, *walking*, and *Nordic walking* were not accurately discriminated. Parkka et al. mentioned that one of the causes of such misclassification is the lack of synchronization between the activity performances and annotations.

3.3.2 Bao

With more than 700 citations [7], the work of Bao and Intelle in 2004 [25] brought significant contributions to the field of activity recognition. The system recognizes 20 activities, including ambulation and daily activities such as *scrubbing, vacuuming, watching TV*, and *working at the PC*. All the data were labeled by the user in a naturalistic environment. Five bi-axial accelerometers were initially placed on the user's knee, ankle, arm, and hip, yet they concluded that with only two accelerometers — on the hip and wrist — the recognition accuracy is not significantly diminished (in about a 5%). Using time- and frequency-domain features along with the C4.5 decision tree classifier, the overall accuracy was 84%. Ambulation activities were recognized very accurately (with up to 95% of accuracy) but activities such as *stretching*, *scrubbing*, *riding escalator* and *riding elevator* were often confused. The inclusion of location information is suggested to overcome this issues. Such an idea was later adopted and implemented by other systems [96].

3.3.3 Khan

The system proposed by Khan et al. [71] not only recognizes ambulation activities, but also transitions among them, e.g., *sitting to walking*, *sitting to lying*, and so forth. An accelerometer was placed on the individual's chest, sampling at 20Hz, and sending data to a computer via Bluetooth for storage. Three groups of features were extracted from the acceleration signals: (1) *autoregressive model coefficients*; (2) the *Tilt Angle* (TA), defined as the angle between the positive Z-axis and the gravitational vector g, as well as the (3) *Signal Magnitude Area* (SMA), which is the summation of the absolute values of all three signals. Linear Discriminant Analysis was used to reduce the dimensionality of the feature vector and an Artificial Neural Network classified activities and transitions with a 97.76% subject independent accuracy. The results indicate that the TA plays a key role in the improvement of the recognition accuracy. This is expected because the sensor inclination values are clearly different for lying and standing, in view of the sensor being placed on the chest. One of the drawbacks of this system is its high computational complexity, as it requires noise reduction, state recognition, time

and frequency-domain features, LDA, and a neural network to recognize the activities.

3.3.4 Zhu

The system proposed by Zhu and Sheng [116] uses *Hidden Markov Models* (HMM) to recognize ambulation activities. Two accelerometers, placed on the subject's wrist and waist, are connected to a PDA via a serial port. The PDA sends the raw data via Bluetooth to a computer which processes the data. This configuration is obtrusive and uncomfortable because the user has to wear wired links that may interfere with the normal course of activities. The extracted features are the angular velocity and the 3D deviation of the acceleration signals. The classification of activities operates in two stages. In the first place, an Artificial Neural Network discriminates among stationary (e.g, sitting and standing) and non-stationary activities (e.g., walking and running). Then, a HMM receives the ANN's output and generates a specific activity prediction. An important issue related to this system is that all the data were collected from one single individual, which does not permit to draw strong conclusions on the system flexibility.

3.3.5 Centinela

This book also presents Centinela [79], a system that combines acceleration data with vital signs to achieve accurate activity recognition. Centinela recognizes five ambulation activities and includes a portable and unobtrusive real-time data collection platform, which only requires a single sensing device and a mobile phone. Time- and frequency-domain features are extracted from acceleration signals while polynomial regression and transient features [79] are applied to physiological signals. After evaluating eight different classifiers and three different time window sizes, and six feature subsets, Centinela achieves an overall accuracy of over 95%. The results also indicate that incorporating physiological signals allows for a significant improvement of the classification accuracy. As a tradeoff, Centinela relies on classifier ensembles accounting higher computational cost, and it requires wireless communication with an external sensor, increasing energy expenditures.

3.3.6 Other approaches

In 2002, Randel et al. [94] introduced a system to recognize ambulation activities which calculates the *Root Mean Square* (RMS) from acceleration signals and makes use of a *Backpropagation Neural Network* for classification. The overall accuracy was 95% using user-specific training data but no details were provided regarding the characteristics of the subjects, the data collection protocol, and the confusion matrix. The system proposed in [57] used HAAR filters to extract features and the C4.5 algorithm for classification purposes.

HAAR filters are intended to reduce the feature extraction computations, compared to traditional TD and FD features. However, the study only collected data from four individuals with unknown physical characteristics, which might be insufficient to provide flexible recognition of activities on new users. He et al. [59, 60, 58] achieved up to 97% of accuracy but only considered four activities: *running*, *being still*, *jumping*, and *walking*. These activities are quite different in nature, which considerably reduces the level of uncertainty thereby enabling higher accuracy. Chen et al. [35] introduced an interesting Dynamic LDA approach to add or remove activity classes and training data online, i.e., the classifier did not have to be re-trained from scratch. With a Fuzzy Basis Function classifier, they reached 93% of accuracy for eight ambulation and daily activities. Nonetheless, all the data were collected inside the laboratory, under controlled conditions. Finally, Vinh et al. [110] used semi-Markovian conditional random fields to recognize not only activities but routines such as *dinner*, *commuting*, *lunch*, and *office*. These routines are composed of sequences of subsets of activities from a total set of 20 activities. Their results indicate 88.38% of accuracy (calculated by the authors from the reported recall tables).

3.3.7 Discussion

Unlike online systems, offline HAR systems are not dramatically affected by processing and storage issues because the required computations are performed in a server with large computational and storage capabilities. Additionally, energy expenditures are not analyzed in detail as a number of systems require neither integration devices nor wireless communication so the application lifetime would only depend on the sensor specifications.

Ambulation activities are recognized very accurately by [70, 71, 79]. These systems place an accelerometer on the subject's chest, which is helpful to avoid ambiguities due to abrupt corporal movements that arise when the sensor is on the wrist or hip [82]. Other daily activities such as *dressing*, *preparing food*, *using the bathroom*, *using the PC*, and *using a phone* are considered in [83]. This introduces additional challenges given that, in reality, an individual could use the phone while walking, sitting, or lying, thereby exhibiting different acceleration patterns. Similarly, in [25], activities such as *eating*, *reading*, *walking*, and *climbing stairs* could happen concurrently yet no analysis is presented to address that matter. Chapter 7 provides insights to the problem of recognizing concurrent activities.

Unobtrusiveness is a desirable feature of any HAR system but having more sensors enables the recognition of a broader set of activities. The scheme presented by Cheng et al. [36] recognizes head movements and activities such as *swallowing*, *chewing*, and *speaking* but requires obtrusive sensors on the throat, chest, and wrist, connected via wired links. In tactical scenarios, this should not be a problem considering that a soldier is accustomed to carrying all sorts of equipment (e.g., sensors, cameras, weapons, and so forth). Yet, in

healthcare applications involving elderly people or patients with heart disease, obtrusive sensors are not convenient.

The studies presented in [35, 36, 66] are based on data collected under controlled conditions while the works in [110, 92, 83, 59, 60, 58] do not specify the data collection procedure. This is a critical issue, since a laboratory environment affects the normal development of human activities [25, 49]. The number of subjects also plays a significant role in the validity of any HAR study. In [116, 84] only one individual collected data while in [57], data were collected from four individuals. Collecting data from a small number of people might be insufficient to provide flexible recognition of activities on new users.

Table 3.3 summarizes state of the art works in supervised offline human activity recognition based on wearable sensors.

3.4 Semi-supervised approaches

The systems studied so far rely on large amounts of labeled training data. Nonetheless, in some cases, labeling all instances may not be feasible. For instance, to ensure a naturalistic data collection procedure, it is recommended for users to perform activities without the participation of researchers. If the user cannot be trusted or the activities change very often, some labels could be missed. These unlabeled data can still be useful to train a recognition model by means of semi-supervised learning. Some of the most important works in this field are described next.

3.4.1 Multi-graphs

Stikic et al. [103, 102] developed a multi-graph-based semi-supervised learning technique which propagates labels through a graph that contains both labeled and unlabeled data. Each node of the graph corresponds to an instance while every edge encodes the similarities between a pair of nodes as a probability value. The topology of the graph is given by the k-nearest neighbors in the feature space. A probability matrix Z is estimated using both Euclidean distance in the feature space and temporal similarity [103]. Once the labels have been propagated throughout the graph (i.e., all instances are labeled), classification is carried out with a Support Vector Machine classifier that relies on a Gaussian radial basis function kernel. The classifier also uses the probability matrix Z to introduce knowledge on the level of confidence of each label. The overall accuracy was up to 89.1% and 96.5% after evaluating two public datasets and having labels for only 2.5% of the training data.

TABLE 3.3: State-of-the-art in offline human activity recognition systems © 2012 IEEE. Reprinted, with permission, from [78].

Ref.	Activities (#)	Sensors	Obtrusiveness	ID	Type of Experim.	Flexibility	Features	Learning	Accuracy
Bao [25]	AMB DA (20)	ACC (wrist, ankle, thigh, elbow, hip)	High	None	NAT	MNL	TD, FD	KNN, C4.5, NB	84%
Hanai [57]	AMB (5)	ACC (chest)	Low	Laptop	N/S	MNL	HAAR filters	C4.5	93.91%
Parkka [88]	AMB DA (9)	ACC, ENV, VS (22 signals)	High	PC	NAT	MNL	TD, FD	DR, KNN	86%
He [60]	AMB (4)	ACC	Low	PC	N/S	MNL	AR	SVM	92.25%
He [58]	AMB (4)	ACC (trousers pocket)	Low	PC	N/S	MNL	DCT, PCA	SVM	97.51%
Zhu [116]	AMB TR (12)	ACC (wrist, waist)	High	PC	N/S	SPC	AV, 3DD	HMM	90%
Altun [22]	AMB (19)	ACC, GYR (chest, arms, legs)	High	None	NAT	MNL	PCA, SFFS	BN, LS, KNN DTW, ANN	87%-99%
Cheng [36]	UB (11)	Electrodes (neck, chest, leg, wrist)	High	PC	LAB	MNL	TD	LDA	77%
McGlynn [83]	DA (5)	ACC (thigh, hip, wrist)	Low	None	N/S	SPC	DTW	DTW ensemble	84.3%
Pham [92]	AMB DA (4)	ACC (jacket)	Medium	N/S	N/S	Both	Relative Energy	NB, HMM	97% (SD) 95% (SI)
Vinh [110]	AMB DA (21)	ACC (wrist, hip)	Medium	N/S	N/S	N/S	TD	SMCRF	88.38%
Centinela [79]	AMB (5) AMB (5)	ACC and VS (chest)	Medium	Smartphone	NAT	MNL	TD, FD, PR, TF	ALR, Bagging, C4.5, NB, BN	95.7%
Khan [71]	AMB TR (15)	ACC (chest)	Medium	Computer	NAT	MNL	AR, SMA, TA, LDA	ANN	97.9%
Jatoba [66]	AMB (6)	ACC SPI	High	Tablet	LAB	Both	TD, FD	CART KNN	86% (SI) 95% (SD)
Chen [35]	AMB, DA HW (8)	ACC (2 wrists)	Medium	N/S	LAB	MNL	TD, FD	FBF	93%
Minnen [84]	AMB MIL (14)	ACC (6 places)	High	Laptop	Both	SPC	TD, FD	Boosting	90%

3.4.2 En-co-training

A well-known method in semi-supervised learning is *co-training*, proposed by Blum and Mitchel in 1998 [30]. This approach requires the training set to have two sufficient and redundant attribute subsets, a condition that does not always hold in a HAR context. Guan et al. [55] proposed *en-co-training*, an extension of co-training which does not have the limitations of its predecessor. The system was tested with ten ambulation activities and compared to three other fully supervised classifiers (the k-nearest neighbors, naïve Bayes and a decision tree). The maximum error rate improvement reached by en-co-training was from 17% to 14% when 90% of the training data were not labeled. If 20% or more of the training data are labeled, the error rate difference between en-co-training and the best fully supervised classifier does not exceed 1.3%.

3.4.3 Ali

Ali et al. [21] implemented a Multiple Eigenspaces (MES) technique based on the Principal Component Analysis combined with Hidden Markov Models. The system is designed to recognize finger gestures with a laparoscopic gripper tool. The individuals wore a sensor glove with two bi-axial accelerometers sampling at 50Hz. Five different rotation and translation movements from the individual's hand were recognized with up to 80% of accuracy. This system becomes hard to analyze since no details are provided on the amount of labeled data nor on the evaluation procedure.

3.4.4 Huynh

Huynh et al. [64] combined Multiple Eigenspaces with Support Vector Machines to recognize eight ambulation and daily activities. Eleven accelerometers were placed on individuals' ankles, knees, elbows, shoulders, wrists, and hip. The amount of labeled training data varied from 5% to 80% and the overall accuracy was between 88% to 64%, respectively. Their approach also outperformed the fully supervised naïve Bayes algorithm, which was used as a baseline. Still, activities such as *shaking hands*, *ascending stairs* and *descending stairs* were often confused.

3.4.5 Discussion

The next step in semi-supervised learning HAR would be the implementation online, opening the possibility to use the data collected in the production stage — which are unlabeled — to improve the recognition performance. Nevertheless, implementing this approach becomes challenging in terms of computational complexity. This is because most semi-supervised HAR approaches first estimate the labels of all instances in the training set and then apply a

conventional supervised learning algorithm. And, the label estimation process is often computationally expensive; for instance, in [103, 102], a graph with one node per instance has to be built. In their experiments, the resulting graphs consisted of up to 16875 nodes, causing the computation of the probability matrix to be highly demanding with regard to processing and storage. Other approaches do not seem to be ready for real scenarios: En-co-training [55] did not report substantial improvement in the classification accuracy. The system proposed by Ali et al. [21] was intended for a very specific purpose but is not suitable for recognizing daily activities thereby limiting its applicability to context-aware applications. Finally, the system proposed in [64] required eleven sensors, which introduces high obtrusiveness. Overall, the field of semi-supervised activity recognition has not reached maturity and needs additional contributions to overcome the aforementioned issues.

Chapter 4

Incorporating Physiological Signals to Improve Activity Recognition Accuracy

As it was mentioned in Chapter 3, most of the previously proposed activity recognition systems collect data from either triaxial accelerometers, video sequences [107], or environmental variables. However, little work has been reported considering vital sign data. It is not difficult to show that there is a noticeable relationship between the behavior of physiological signals and the user's activity. When an individual begins running, for instance, it is expected that their heart rate and breath amplitude would increase. Consequently, the hypothesis of this chapter is that higher human activity recognition accuracy can be achieved using both acceleration and vital sign data. To illustrate this, consider the situation in Figure 4.1. Data from triaxial acceleration and vital signs were recorded while a subject was *ascending* (i.e., walking upstairs) after *walking*. Note that the acceleration signals within most time intervals are very similar for both activities. Instead, the heart rate time series exhibits a very clear pattern, as a person requires more physical effort to climb stairs than to walk. This might allow to classify said activities more accurately.

This chapter presents Centinela, a human activity recognition system that considers acceleration and physiological signals. The proposed methodology encompasses (1) collecting vital sign and acceleration data from human subjects; (2) extracting features from the measured attributes; (3) building supervised machine learning models for activity classification; and (4) evaluating the accuracy of the models under different parameter configurations. The main features of Centinela are listed below:

- Centinela combines acceleration data with vital signs data to achieve highly accurate activity recognition. In fact, it provides higher accuracy than using acceleration signals solely.
- Five activities are recognized as a proof of concept: *walking*, *running*, *being still* (sitting or standing), *ascending* (i.e., walking upstairs), and *descending* (i.e., walking downstairs).
- Since vital signs are not expected to change abruptly, Centinela applies structure detectors [87], i.e., linear and non-linear functions, to extract features.

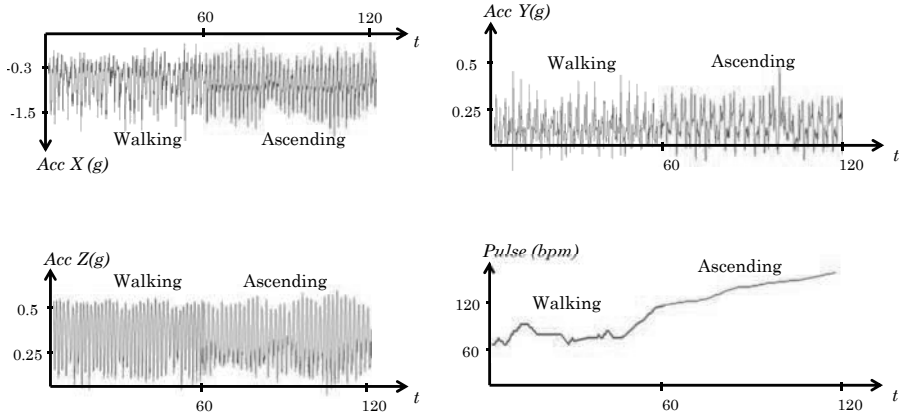

FIGURE 4.1: Acceleration signals and heart rate for the activities *walking* and *ascending* © 2012 Elsevier. Reprinted, with permission, from [79].

- Three new features are proposed for physiological signals: *trend*, *magnitude of change*, and *signed magnitude of change*, intended to discriminate among activities during periods of vital sign stabilization.
- Centinela relies on a portable and unobtrusive real-time data collection platform, which allows not only for activity recognition but also for monitoring health conditions of target individuals.
- Several classifiers are analyzed in this study, allowing other researchers and application developers to use the most appropriate classifiers for specific activities.

The rest of the chapter is organized as follows: Section 4.1 introduces the global structure of Centinela. Section 4.1.1 describes the data acquisition architecture as well as the data collection protocol. Section 4.1.2 covers the methods applied for feature extraction, i.e., statistical, structural, and transient features. Then, Section 4.2 presents the methodology of the experiments and main results. Finally, Section 4.3 summarizes the most important conclusions and findings.

4.1 Description of the system

Figure 4.2 illustrates the process fulfilled for activity recognition. First, labeled data are collected from the accelerometer and vital sign sensors, as described in Section 4.1.1. Then, time- and frequency-domain statistical feature

extraction are applied to the acceleration signals (Section 4.1.2.1), and structural and transient features are extracted from vital signs (Sections 4.1.2.2 and 4.1.2.3). Next, the dataset with the extracted features is passed as input to various classification algorithms in order to select the most appropriate model (Section 4.2).

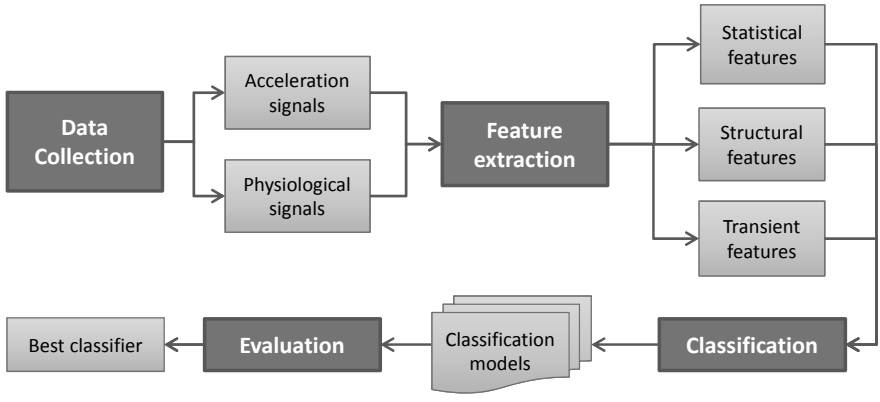

FIGURE 4.2: Centinela's data flow © 2012 Elsevier. Reprinted, with permission, from [79].

4.1.1 Data collection

Figure 4.3 shows the system architecture for the data collection phase. The sensing device (see Section 4.1.1.1 for more details) communicates via Bluetooth with an Internet-enabled cellphone. There is a mobile application which decodes the packets and sends labeled data to the application server via the Internet. The server then receives these data and stores them in a relational database.

4.1.1.1 Sensing device

Centinela uses the BioHarness BT^{TM} chest sensor strap [19] manufactured by Zephyr Technology Corporation (see Figure 4.3). This device features a triaxial accelerometer and allows for measuring vital signs as well. The strap is unobtrusive, lightweight, and can be easily worn by any person. The measured attributes are: *heart rate* (i.e., pulse), *respiration rate*, *breath waveform amplitude*, *skin temperature*, *posture* (i.e., inclination of the sensor), *ECG amplitude*, and *3D acceleration*, among others. The accelerometer records measurements at 50 Hz, each one between $-3g$ to $+3g$, where g stands for the acceleration of gravity. Acceleration samples are aggregated in packets sent every 400 ms, so every packet contains twenty acceleration measurements in all three dimen-

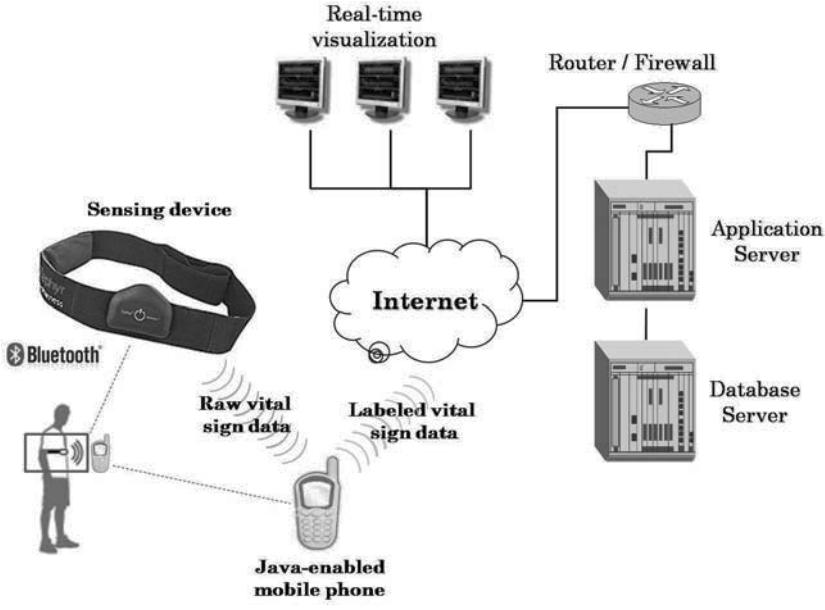

FIGURE 4.3: Data collection architecture © 2012 Elsevier. Reprinted, with permission, from [79].

sions. On the other hand, the vital signs are sampled at 1 Hz., since they are not expected to change considerably in short periods of time.

In the literature, accelerometers are commonly placed on the wrist [25, 48, 88], ankle [25, 48], or in the trousers pocket [60, 58, 59], yet a person might be, for instance, moving his/her arms or legs while seated. This fact may introduce noise to the data, thereby causing misclassification. Therefore, placing the accelerometer on the chest makes the system more noise tolerant, and the results presented in Section 4.2.2 support this hypothesis.

4.1.1.2 Mobile application

A mobile software application was built to collect training data under the Java ME platform. This allows Centinela to run in any mobile phone that supports Java, thereby avoiding the inconvenience of requiring the user to carry additional recording devices. The mobile application receives and decodes the raw data sent from the sensor via Bluetooth, visualizes the measurements (see Figure 4.4.A), and labels each measurement according to the option selected by the user, either: *running*, *walking*, *sitting*, *ascending,* or *descending* (see Figure 4.4.B). The samples are sent in real time, via UDP, to the application server, which stores the labeled data in a relational PostgreSQL database. As Java ME is falling into disuse, the mobile component of Centinela was

(A) (B)

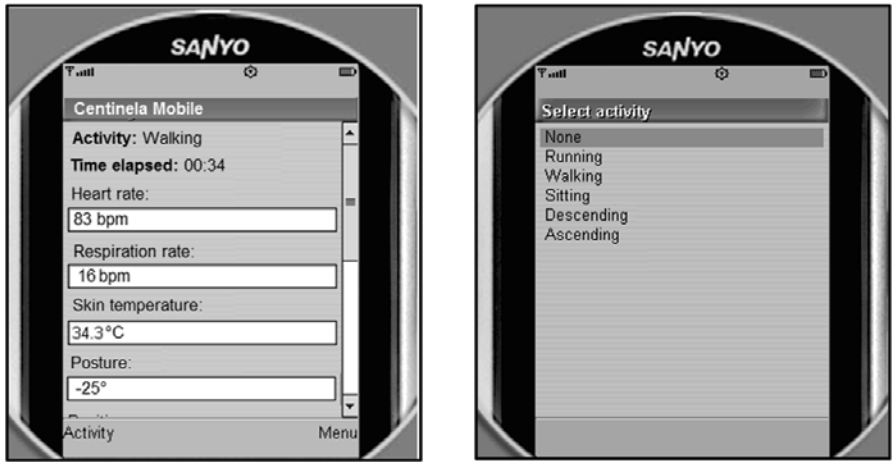

FIGURE 4.4: Mobile application user interface © 2012 Elsevier. Reprinted, with permission, from [79].

TABLE 4.1: Physical characteristics of the participants © 2012 Elsevier. Reprinted, with permission, from [79].

	Avg.	Min	Max
Age (years)	24	9	34
Weight (kg)	76.5	27	95
Height (m)	1.74	1.35	1.88
BMI (kg/m^2)	24.23	20.96	29

migrated to the Android platform, including additional features and functionalities. In Chapter 5, the reader may find more details about this matter.

4.1.1.3 Data collection protocol

The data were collected in a naturalistic fashion, thus, no specific instructions were given to the participants on how to perform the activities. The speed, intensity, gait, and other environmental conditions were arbitrarily chosen by the subjects. Eight individuals, 7 males and 1 female, participated in this study. Their physical characteristics, namely age, weight, height, and body mass index are summarized in Table 4.1.

Unlike accelerometer signals, vital signs do not abruptly vary after the person changes activities. On the contrary, the values of vital signs during time interval I_j depend of the activity during I_{j-1}. If the individuals were at rest before recording each session, the system would not be trained to

TABLE 4.2: List of features extracted © 2012 Elsevier. Reprinted, with permission, from [79].

Measured Signals	Extracted Features			
	Statistical	**Structural**	**Transient**	**Total**
AccX (g)	×			8
AccY (g)	×			8
AccZ (g)	×			8
Heart rate		×	×	10
Respiration rate		×	×	10
Breath amplitude		×	×	10
Skin temperature		×	×	10
Posture		×	×	10
ECG amplitude		×	×	10
Total	24	54	6	84

recognize interleaving activities! Consequently, the data were collected from subjects while performing successive pairs of activities, e.g., *running* before *sitting, walking* before *descending*, and so on. This was carried out for all twenty possible combinations of pairs of consecutive activities.

4.1.2 Feature extraction

In general, two approaches have been proposed to extract features in time series data: *statistical* and *structural* [87]. The former, such as the Fourier transform and the Wavelet transform, uses quantitative characteristics of the data to extract features. The latter takes into account the morphological interrelationship among data. Hence, they have been widely used for image processing and time series analysis. Due to both acceleration and physiological signals being distinct in nature, Centinela applies methods from both statistical and structural feature extraction.

In order to overcome the problem of detecting transitions between activities, all measured signals were divided into fixed size 50% overlap time windows, as suggested by [35, 25]. For every time window, 84 features were extracted as follows: eight statistical features for each of the acceleration signals (i.e., 24 features), nine structural features for each of the physiological signals (i.e., 54 features), and one *transient* feature for each of the physiological signals (i.e., 6 features). Transient features are proposed in this book (see Section 4.1.2.3) to address activity recognition during periods of vital sign stabilization — they could be considered structural as they also provide information on the signal shape. Table 4.2 summarizes the feature set computed from raw signals. The definitions of these features are presented in the following subsections.

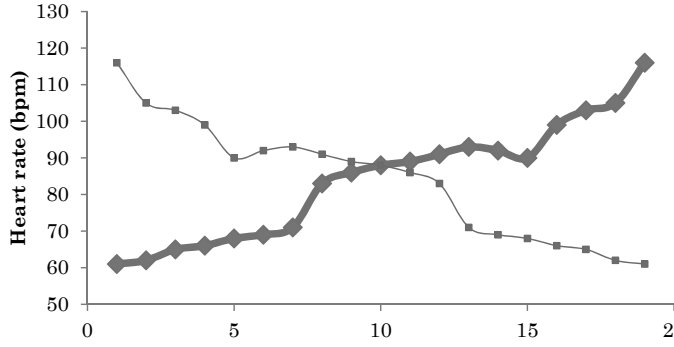

FIGURE 4.5: Heart rate signal for *walking* (bold) and flipped signal (thin).

4.1.2.1 Statistical features from acceleration signals

Time-domain and frequency-domain features have been extensively used to filter relevant information within acceleration signals [35, 25, 81, 88, 104, 48, 69]. In this work, eight features were calculated for all three acceleration signals (a total of 24 features). These are: *mean*, *variance*, *standard deviation*, *correlation between axes, interquartile range, mean absolute deviation*, and *root mean square,* from the time domain; and, *energy* from the frequency domain. These features were defined in Section 2.2.1.1.

4.1.2.2 Structural features from physiological signals

Previous works that explored vital sign data with the aim of recognizing human activities applied statistical feature extraction. In [104], the authors computed the number of heartbeats above the resting heart rate value as the only feature. Instead, Parkka et al. [88] calculated time domain features for heart rate, respiration effort, SaO_2, ECG, and skin temperature. Nevertheless, the signal's shape is not described by these features. Consider the situation shown in Figure 4.5. A heart rate signal $S(t)$ for an individual that was *walking* is shown with a bold line and the same signal in reverse temporal order, $S'(t)$, is displayed with a thin line. Notice that most time domain and frequency domain features (e.g., mean, variance, and energy) are identical for both signals while they may represent different activities. This is the main motivation for applying structural feature extraction: describing the morphological interrelationship among data.

Given a time series $Y(t)$, a *structure detector* implements a function $f(Y(t)) = \hat{Y}(t)$ such that $\hat{Y}(t)$ represents the structure of $Y(t)$ as an approximation [87]. The extracted features are the parameters of $\hat{Y}(t)$, which depend on the nature of the function. In order to measure the goodness of fit of $\hat{Y}(t)$ to $Y(t)$, the sum of squared errors (SSE) is calculated as follows:

TABLE 4.3: Common functions implemented by structure detectors © 2012 Elsevier. Reprinted, with permission, from [79].

Function	Equation	Parameters
Linear	$\hat{Y}(t) = mt + b$	$\{m, b\}$
Polynomial	$\hat{Y}(t) = a_0 + a_1 t + ... + a_{n-1} t^{n-1}$	$\{a_0, ..., a_{n-1}\}$
Exponential	$\hat{Y}(t) = a\|b\|^t + c$	$\{a, b, c\}$
Sinusoidal	$\hat{Y}(t) = a * sin(t + b) + c$	$\{a, b, c\}$

$$SSE = \sum_{t} \left(Y(t) - \hat{Y}(t) \right)^2 \tag{4.1}$$

Then, for each measured attribute, the goal is to find the function $\hat{Y}^*(t)$ with the smallest SSE. Table 4.3 summarizes the different types of functions that have been evaluated in this work. The median of the SSE was calculated for all time windows from all six physiological signals and all four structure detectors. The median was preferred over the mean to prevent noisy samples to bias the goodness of fit of the feature detectors. From the evaluation, polynomial functions of third degree had the lowest SSE for all six vital signs. Polynomials of a degree higher than three were not considered to avoid overfitting due to Runge's phenomenon [43].

A total of nine structural features were extracted from each vital sign time window, i.e., the coefficients of the polynomials of degree one, two, and three that best fitted the points in the time window.

4.1.2.3 Transient features from physiological signals

Consider, for instance, that someone is running for one minute and then sits down for two minutes. Even though the individual is seated, their vital signs (e.g., heart rate, respiration rate, etc.) remain as if she were running for an interval of time called the *transient period*. To overcome this issue, three new features are proposed in this work — besides structural features —, namely the *trend* τ, the *magnitude of change* κ, and the *signed magnitude of change* η, intended to describe the behavior of the vital signs during transient periods. The trend indicates whether the signal is increasing, decreasing, or constant. Notice that, due to the nature of the human activities considered in this work, it is expected that vital signs are either strictly increasing, strictly decreasing, or remain constant while an individual is performing one single activity.

Definition 4.1 ***Trend:*** *Let m be the slope of the line that best fits the series*

S. Then, the trend $\tau(S, r)$ of S is defined as follows:

$$\tau(S,r) = \begin{cases} 1\ (increasing) & if\ (m \geq r) \\ -1\ (decreasing) & if\ (m \leq -r) \\ 0\ (constant) & if\ (|m| < |r|) \end{cases} \tag{4.2}$$

where r is a positive real number that stands for the slope threshold.

The value of r was set to $\tan(15°)$ after doing an experimental analysis over the entire the dataset. The *trend* can be computed in $O(1)$ given that the slope of the line that best fits the data points was calculated beforehand as one of the structural features.

Now, it is important not only to detect whether the vital signs increased or decreased in a time window, but also to measure how much they varied. For this purpose the magnitude of change feature is presented as follows:

Definition 4.2 ***Magnitude of change:*** *Let S be a given time series defined from t_{min} to t_{max}. Let S_p^- be a subset of S which contains all measurements between t_{min} and $t_{min} + (t_{max} - t_{min})p$, where $0 < p < 1$ is a percentage of the series. Let S_p^+ be a subset of S which contains all samples between $t_{min} + (t_{max} - t_{min})(1-p)$ and t_{max}. Then, the magnitude of change $\kappa(S, p)$ is defined as*

$$\begin{aligned} \kappa(S,p) \quad = \quad & \max\{\ \left|\max\left(S_p^+\right) - \min(S_p^-)\right|, \\ & \left|\max\left(S_p^-\right) - \min(S_p^+)\right|\ \} \end{aligned} \tag{4.3}$$

The value of p was set to 0.2 after doing an experimental analysis over the entire dataset. This implies that S^- is the first 20% of the series and S^+ would be the last 20% of the series. The purpose of the magnitude of change is to estimate the maximum deviation between the beginning and the end of the series, and it can be calculated in linear time. Figure 4.6 illustrates the process of calculating this feature.

Both magnitude of change and trend are combined in a single feature, the *signed magnitude of change* η as follows:

Definition 4.3 ***Signed magnitude of change:*** *Given a time series S, the signed magnitude of change η is defined as follows:*

$$\eta(S,p,r) = \kappa(S,p)\tau(S,r) \tag{4.4}$$

where p and r are the parameters of κ and τ, respectively. Even though the transient features are strongly related to the parameters of a linear regression, they are different measures of the data shape. Section 4.2.2.4 analyzes the effectiveness of these proposed features.

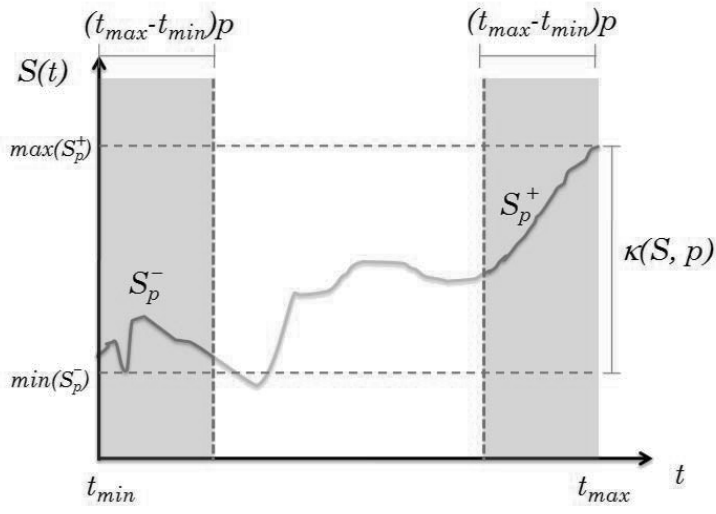

FIGURE 4.6: Calculation of the *magnitude of change* feature © 2012 Elsevier. Reprinted, with permission, from [79].

4.2 Evaluation

This section describes the methodology used to evaluate the system and provides further analysis and discussion of the main results and findings.

4.2.1 Design of the experiments

The recognition of activities was fulfilled by assessing two different datasets: the first one, D_{acc}, solely contains the features extracted from acceleration data; the second, D_{vs}, includes all features (i.e., statistical, structural, and transient). The comparison of these two datasets is with the purpose of measuring the impact of vital sign features in the classification accuracy. Seven classification algorithms were evaluated:

1. Naïve Bayes (NB) [115].
2. Bayesian Network (BN) using the K2 search algorithm [23].
3. J48 decision tree, which is an implementation of the C4.5 algorithm [111].
4. Multilayer Perceptron (MLP), which relies on a Backpropagation Neural Network [111].

5. Additive Logistic Regression (ALR) [51], performing Boosting with an ensemble of ten Decision Stump classifiers.

6. Bagging using an ensemble of ten Naïve Bayes classifiers (BNB), each bag having the same size as the training set.

7. Bagging using an ensemble of ten J48 classifiers (BJ48), each bag having the same size as the training set.

The interested reader may refer to [111, 23, 115, 51] for a complete description of these classification methods.

The evaluation encompasses two parts: the selection of the best classifier(s), and the calculation of their accuracy. In order to determine whether a classifier is better than another, a 5×2-fold cross validation along with a paired t-test were performed, as suggested in [45]. In general, a two-fold cross validation is preferred, to reduce the probability of concluding that one classifier is better than another when that is not the case. For all the statistical tests, the significance level was fixed to $\alpha = 0.05$.

All the classification algorithms were tested in the Waikato Environment for Knowledge Analysis (WEKA) [18]. This is a well-known software tool developed by the University of Waikato, New Zealand, which facilitates the evaluation and analysis of machine learning algorithms.

4.2.2 Results

An interesting fact in machine learning is that the performance of a classification algorithm depends on the dataset it is applied to. As the goal of this study is to prove that vital signs account for more accurate recognition, each classification algorithm was evaluated in both datasets.

The data repository used for this study is more complete than the one presented in [79]. In fact, the former has over 35 minutes of labeled data (i.e., up to 630 instances) while the present one has only 19 minutes (i.e., up to 342 instances). Evidently, a larger dataset allows one to draw more robust conclusions on the system performance. Two time window sizes were evaluated, namely 5s and 12s. In the previous study [79], longer time windows were shown to diminish the overall classification accuracy and could contain more than one activity.

In total, fourteen classifiers were evaluated — seven for each window size — with five different random seeds $s_i \in \{1, 128, 255, 1023, 4095\}$. As a notation, the name of the classifiers is accompanied by the window size written as a superscript and the dataset as a subscript. For example, ALR_{vs}^{12s} stands for the Additive Logistic Regression algorithm over the D_{vs} dataset using 12s time windows.

TABLE 4.4: Percentage classification accuracy given by the 5×2-fold cross validation on D_{vs}.

	s_1	s_2	s_3	s_4	s_5	Avg.	p-value
MLP_{vs}^{5s}	88.87	90.78	89.35	88.71	88.08	89.16	0.001
NB_{vs}^{5s}	74.09	76.95	80.76	79.33	73.77	76.98	0.000
BN_{vs}^{5s}	90.62	90.14	89.67	89.83	88.08	89.67	0.008
ALR_{vs}^{5s}	**91.57**	**93.64**	**93.64**	**93.48**	**93.32**	**93.13**	-
$J48_{vs}^{5s}$	85.85	87.28	87.44	90.78	86.96	87.66	0.001
BNB_{vs}^{5s}	74.09	75.99	78.54	79.81	73.77	76.44	< 0.001
$BJ48_{vs}^{5s}$	87.28	92.53	88.39	92.21	90.14	90.11	0.021
MLP_{vs}^{12s}	80.14	83.39	82.67	85.56	80.51	82.45	< 0.001
NB_{vs}^{12s}	74.01	80.87	71.12	74.73	73.29	74.80	< 0.001
BN_{vs}^{12s}	85.20	89.53	85.56	88.09	88.81	87.44	0.009
ALR_{vs}^{12s}	90.25	92.06	91.70	90.97	91.34	91.26	0.001
$J48_{vs}^{12s}$	85.92	85.92	84.84	85.56	80.87	84.62	0.001
BNB_{vs}^{12s}	75.45	79.42	71.48	74.73	74.37	75.09	0.002
$BJ48_{vs}^{12s}$	84.48	89.53	88.45	87.00	85.20	86.93	0.001

4.2.2.1 Dataset with features from vital signs and acceleration

Table 4.4 shows the results of the 5×2-fold cross validation for the D_{vs} dataset. Note that the highest overall accuracy was achieved by the ALR_{vs}^{5s} classifier. In order to find the best classifier for this dataset, a paired two-tailed t-test was performed between the ALR_{vs}^{5s} and all other classifiers with a significance level $\alpha = 0.05$. The null hypothesis is that each pair of classifiers achieved the same mean accuracy. As a result of the tests, the p-values are included in the last column of Table 4.4. Since all the p-values are below the significance level, there is strong statistical evidence that ALR_{vs}^{5s} is more accurate than all the other classifiers in the D_{vs} dataset.

4.2.2.2 Dataset with features from acceleration only

The same procedure was carried out over D_{acc} (i.e., the dataset that only contains features from accelerometer data). Table 4.5 summarizes the results of the 5×2-fold cross validation for this dataset; here, the $BJ48_{acc}^{5s}$ classifier achieved the highest average accuracy (i.e., 88.7%). However, ALR_{vs}^{5s} still outperforms all classifiers in D_{acc} since the p-values are below α. This is a very remarkable result as it provides strong statistical support for one of the hypotheses presented in this book: physiological signals being beneficial to improve the recognition accuracy of human activities.

4.2.2.3 Measuring the impact of vital signs

It is important to highlight that a 5×2-fold cross validation is not intended to measure the classification accuracy but rather to find differences in

TABLE 4.5: Percentage classification accuracy given by the 5×2-fold cross validation on D_{acc}.

	s_1	s_2	s_3	s_4	s_5	Avg.	p-value
MLP_{acc}^{5s}	86.65	85.37	87.92	87.60	85.21	86.55	< 0.001
NB_{acc}^{5s}	83.62	74.09	79.97	80.29	79.65	79.52	< 0.001
BN_{acc}^{5s}	83.31	82.51	83.31	84.90	80.76	82.96	0.001
ALR_{acc}^{5s}	88.24	87.60	87.60	86.65	86.49	87.31	0.002
$J48_{acc}^{5s}$	84.42	86.17	85.69	86.33	86.33	85.79	< 0.001
BNB_{acc}^{5s}	83.78	74.88	81.88	79.01	80.45	80.00	0.002
$BJ48_{acc}^{5s}$	87.28	**89.83**	**88.39**	**90.78**	**88.08**	**88.87**	0.001
MLP_{acc}^{12s}	**90.61**	80.14	87.36	87.73	85.92	86.35	0.028
NB_{acc}^{12s}	78.34	80.14	76.90	78.34	78.34	78.41	< 0.001
BN_{acc}^{12s}	77.26	83.03	79.42	83.39	81.95	81.01	< 0.001
ALR_{acc}^{12s}	87.00	85.92	83.03	87.00	84.84	85.56	0.002
$J48_{acc}^{12s}$	84.48	84.48	78.70	84.84	81.23	82.74	0.002
BNB_{acc}^{12s}	79.42	80.87	79.42	80.51	79.42	79.93	< 0.001
$BJ48_{acc}^{12s}$	84.84	85.92	85.56	84.84	87.00	85.63	< 0.001

the overall accuracy of the classifiers. This is because a two-fold cross validation only uses half of the dataset for training. Now, the actual classification accuracy was measured by a 5×10-fold cross validation. After evaluating the best classifiers in each dataset for all five random seeds, the overall accuracy for ALR_{vs}^{5s} was 95.37% whereas $BJ48_{acc}^{5s}$ only reached 90.02%. This is a reduction of 53% in the error rate. A more detailed analysis was carried out for each class (i.e., activity) by calculating a number of performance metrics: *precision, recall, F-measure, false positive rate (FPR), and false negative rate (FNR)* — their definitions were presented in Section 2.3.2. Figures 4.7 and 4.8 compile the results of the evaluation metrics. Observe that all of them confirm that physiological signals are useful to enhance the recognition of all activities. The most noticeable improvements are for *ascending*, *descending*, and *walking*. Indeed, the FPR was reduced in 52% for *walking*, 67% for *ascending*, and 72% for *descending*. At the same time, the FNR was reduced 69% for *walking* and nearly 50% for *ascending* and *descending*. This was expected as acceleration signals tend to be similar for *ascending*, *walking*, and *descending* whereas vital signs provide clearer patterns to distinguish among these activities (see Figure 4.1).

4.2.2.4 Analyzing the impact of transient features

To measure the effectiveness of transient features, two new datasets were evaluated: $D_{acc+str}$ and $D_{acc+tra}$. The former incorporates statistical features from acceleration signals along with structural features from physiological signals. The latter includes statistical features from acceleration signals, transient features, and the parameter b of the line $y(t) = mt + b$ that best fits the points

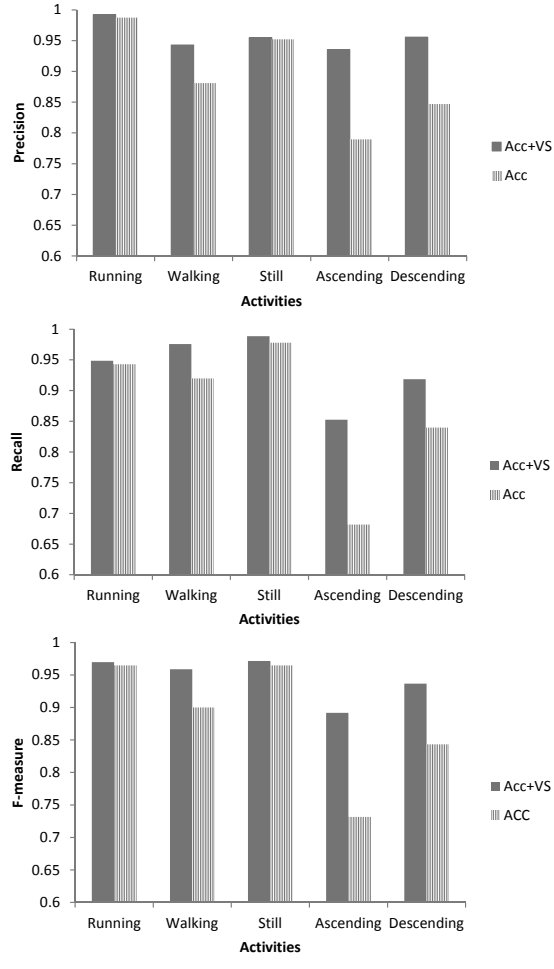

FIGURE 4.7: Evaluation metrics for the best classifiers in each dataset: precision, recall, and F-measure.

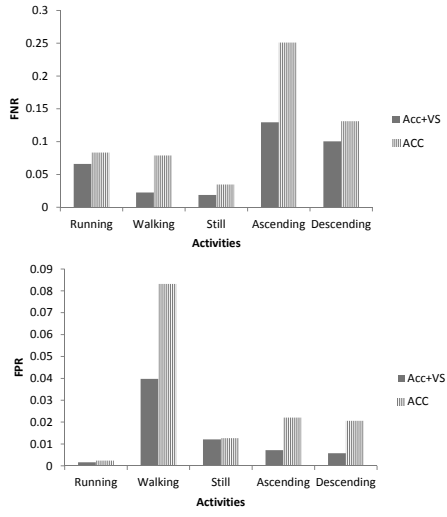

FIGURE 4.8: Evaluation metrics for the best classifiers in each dataset: false positive rate (FPR) and false negative rate (FNR).

TABLE 4.6: Evaluation metrics for the best classifiers in each dataset: precision, recall, false positive rate (FPR), false negative rate (FNR), and F-measure for ALR_{vs}^{5s} (A) and $BJ48_{acc}^{5s}$ (B).

(A)

Overall accuracy: 95.37%					
ALR_{vs}^{5s}	**Running**	**Walking**	**Still**	**Ascending**	**Descending**
Precision	0.997	0.939	0.958	0.931	0.96
Recall	0.95	0.973	0.985	0.854	0.930
F-measure	0.973	0.956	0.971	0.891	0.945
FPR	0.001	0.042	0.011	0.08	0.005
FNR	0.05	0.027	0.015	0.146	0.070

(B)

Overall accuracy: 90.03%					
$BJ48_{acc}^{5s}$	**Running**	**Walking**	**Still**	**Ascending**	**Descending**
Precision	0.987	0.881	0.952	0.789	0.847
Recall	0.943	0.920	0.978	0.682	0.840
F-measure	0.965	0.9	0.965	0.732	0.843
FPR	0.002	0.083	0.013	0.022	0.21
FNR	0.057	0.08	0.022	0.318	0.160

of the signal. This last feature is included in $D_{acc+tra}$ to describe not only the shape, but also the values in the signal. In this manner, a heart rate signal that linearly varies from 80 bpm to 70 bpm, for instance, could be differentiated from another one which changes from 110 bpm to 100 bpm, given that they might represent different activities.

Tables 4.7-(A) and 4.7-(B) display the overall accuracy for $D_{acc+str}$ and $D_{acc+tra}$, respectively. Notice that the accuracies for D_{vs}, $D_{acc+str}$, and $D_{acc+tra}$ are very similar, being slightly higher for $D_{acc+tra}$. This implies that transient features could substitute the coefficients of the polynomials that best fit the physiological signals, thereby simplifying the model and reducing the computational complexity of the feature extraction process. This is because there are only two transient features per attribute (i.e., the signed magnitude of change and the intersection with the Y axis) versus nine polynomial coefficients.

The last experiment compares transient features to linear regression features. Therefore, the last dataset evaluated was D_{acc+lr}, which contains statistical features from acceleration signals and linear coefficients (i.e., slope and intersection with the Y-axis) from physiological signals. The results for D_{acc+lr} are included in Table 4.7 (C). Notice that the computational complexity for calculating the features in D_{acc+lr} and $D_{acc+tra}$ is the same, but the accuracy for the latter is slightly higher.

4.2.3 Confusion matrix

The confusion matrix for the best classifier, $ALR^{5s}_{acc+tra}$, is shown in Table 4.8 after five iterations using five different random seeds. Confusions are, on average, less than 5%, and mostly among three activities: *walking*, *ascending*, and *descending*. This was expected since these three activities exhibit similar patterns depending on the intensity at which they are performed by the individual.

4.3 Concluding remarks

In this chapter, physiological signals have been shown to be an important source of information to improve the recognition of human activities. As a matter of fact, incorporating vital signs besides the traditionally used acceleration signals accounts for a reduction of 53% in the error rate. Such improvement is especially significant for *ascending*, *descending*, and *walking* activities that could be confusing from the acceleration point of view. The overall accuracy was up to 95.65%.

Of course, in order to measure physiological signals, additional sensors and wireless communication are required, introducing higher energy expendi-

TABLE 4.7: Evaluation metrics for the best classifiers in each dataset: precision, recall, false positive rate (FPR), false negative rate (FNR), and F-measure.

(A)

Statistical and structural features ($D_{acc+str}$)					
Overall accuracy: 95.48%					
	Running	**Walking**	**Still**	**Ascending**	**Descending**
Precision	0.995	0.944	0.960	0.925	0.956
Recall	0.951	0.974	0.986	0.860	0.927
F-measure	0.973	0.959	0.973	0.891	0.941
FPR	0.001	0.039	0.011	0.008	0.006
FNR	0.049	0.026	0.014	0.140	0.073

(B)

Statistical and transient features ($D_{acc+tra}$)					
Overall accuracy: 95.654%					
	Running	**Walking**	**Still**	**Ascending**	**Descending**
Precision	0.978	0.942	0.967	0.950	0.967
Recall	0.952	0.982	0.988	0.884	0.889
F-measure	0.965	0.962	0.978	0.916	0.926
FPR	0.004	0.041	0.009	0.006	0.004
FNR	0.048	0.018	0.012	0.116	0.111

(C)

Statistical and linear regression (D_{acc+lr})					
Overall accuracy: 95.27%					
	Running	**Walking**	**Still**	**Ascending**	**Descending**
Precision	0.985	0.949	0.957	0.906	0.956
Recall	0.947	0.975	0.984	0.887	0.893
F-measure	0.965	0.962	0.970	0.896	0.924
FPR	0.003	0.035	0.011	0.011	0.006
FNR	0.053	0.025	0.016	0.113	0.107

TABLE 4.8: Confusion matrix for the best classifier $ALR^{5s}_{acc+tra}$ after five iterations with different random seeds.

	Running	**Walking**	**Still**	**Ascending**	**Descending**
Running	**499**	12	12	2	0
Walking	5	**1236**	7	4	8
Still	4	4	**636**	1	0
Ascending	2	33	4	**298**	3
Descending	0	27	0	12	**336**

tures and obtrusiveness. All these aspects should be carefuly evaluated before extrapolating a HAR system to a real world application. Finally, transient features were introduced as an efficient alternative to describe underlying patterns within physiological signals. They can be computed in linear time and are an important piece to achieve the highest recognition accuracy.

Chapter 5

Enabling Real-Time Activity Recognition

Chapter 4 shows that human activities can be effectively recognized with acceleration and physiological signals. However, such recognition and the performance analyses were carried out offline, in a server. This chapter focuses on the next step in HAR: the online implementation (i.e., providing real-time feedback) on smartphones. Such a task is not trivial in view of the energy, memory, and computational constraints present in these devices. In activity recognition applications, those limitations are particularly critical since they require data preprocessing, feature extraction, classification, and transmission of large amounts of raw data. Furthermore, to the best of our knowledge, available machine learning API's such as WEKA [18] and JDM [11] are neither optimized nor fully functional under current mobile platforms. This fact accentuates the necessity for an efficient mobile library to evaluate machine learning algorithms and implement HAR systems in mobile devices.

A mobile HAR system will bring important advantages and benefits. In the first place, energy expenditures are expected to be substantially reduced, as raw data would not have to be continuously sent to a server for processing. The system would also become more robust and responsive because it would not depend on unreliable wireless communication links, which may be unavailable or error prone. Finally, a mobile HAR system would be more scalable since the server load would be alleviated by the locally performed feature extraction and classification computations.

This chapter introduces Vigilante, a mobile framework in support of real-time human activity recognition under the Android platform — reported as the best-selling smartphone platform in 2010 by Canalys [8].

5.1 Existing mobile real-time HAR systems

Human activity recognition is a well-studied field, yet very few works have successfully been deployed in mobile phones. In 2010, Berchtold et al. introduced *ActiServ* as an activity recognition service for mobile phones [26]. They make use of a fuzzy inference system to classify daily activities, achieving up

to 97% of accuracy. Nevertheless, it requires a runtime duration in the order of days! When their algorithms are executed to meet a feasible response time, the accuracy drops to 71%. ActiServ can also reach up to 90% after personalization, in other words, a subject-dependent analysis (i.e., the system needs to be re-trained for new users). Brezmes et al. [32] proposed a mobile application for HAR under the Nokia platform; but they used the k-nearest neighbors classifier, which is not scalable for mobile phones as it would need the entire training set, which can be fairly large, to be stored in the device. Finally, Riboni et al. [96] presented COSAR, a framework for context-aware activity recognition using statistical and ontological reasoning under the Android platform. The system recognizes a number of activities very accurately yet it depends on the GPS sensor and an additional accelerometer on the user's wrist. This introduces higher energy expenditures and privacy concerns. Furthermore, their proposed historical variant increases the response time of the system.

5.2 Proposed system

Figure 5.1 illustrates the main components of Vigilante and their interrelationships. Three *wearable sensors* have been integrated: the phone accelerometer, the GPS, and the Bioharness BT sensor strap, manufactured by Zephyr technology [19]. The measured attributes are collected by the mobile application, which executes the data preprocessing, feature extraction, activity recognition, and visualization modules. The *application server* performs several tasks. In the first place, it delivers previously trained classification models that can be downloaded by the mobile application. In this fashion, improved classification algorithms and new feature extraction methods can be easily available on the phone. In the second place, the application server provides authentication and session management, and records additional training data according to the users' preferences. Finally, the *database server* runs a PostgreSQL database to store sessions, user information, training data, and classification models, among other miscellaneous data.

The rest of this chapter is devoted to the mobile application. Figure 5.2 shows its design: first, the *communication* and *sensing* modules, which allow for the collection of raw data from the sensing devices. These data are decoded and organized in time windows by the *data preprocessing* component. Then, the *feature extraction* module extracts statistical and structural features from each time window, producing a feature set instance which is later evaluated by the *classification* module. The output of the classifier is indeed the recognized activity, displayed by the *visualization* module and sent to the server for further analysis and historical querying. When the system is working on

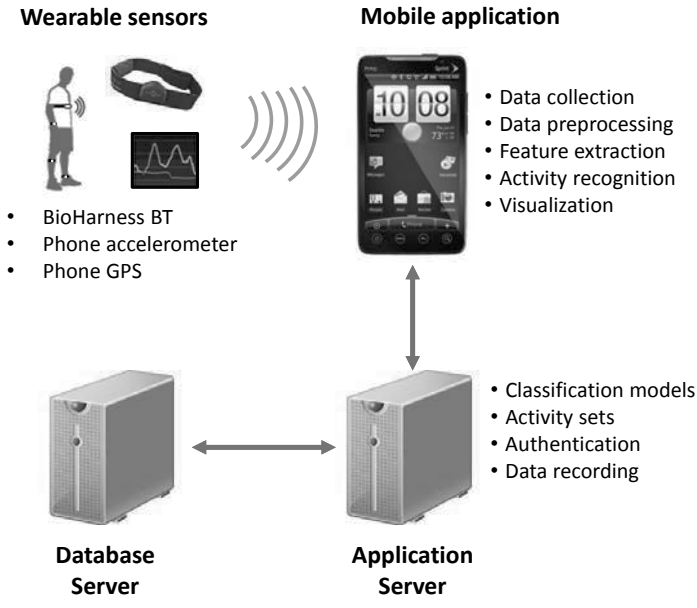

FIGURE 5.1: System architecture.

training mode, all raw data are also sent to the server. All these modules are explained in more detail next.

5.2.1 Sensing devices

Vigilante currently supports three sensing devices, namely the smartphone GPS, smartphone accelerometer, and the BioHarness^{TM} BT chest sensor strap, manufactured by Zephyr. The strap is unobtrusive, lightweight, and can be easily worn by any person. More information on this sensor can be found in Section 4.1.1.1 and in [19]. Although the smartphone's accelerometer and GPS sensors are available in the Vigilante platform, they were not used for activity recognition purposes. Instead, the accelerometers embedded in the chest strap were used.

5.2.2 Communication

The communication module encompasses three levels: (1) receiving raw data from the sensors (via Bluetooth); (2) sending raw data or activity results to the server (via TCP/IP); and (3) querying the database (via HTTP servlets). In the first level, three different types of packets are received from the sensing device: acceleration, vital signs, and electrocardiogram. In the second level, these packets are aggregated and sent to the application server, which

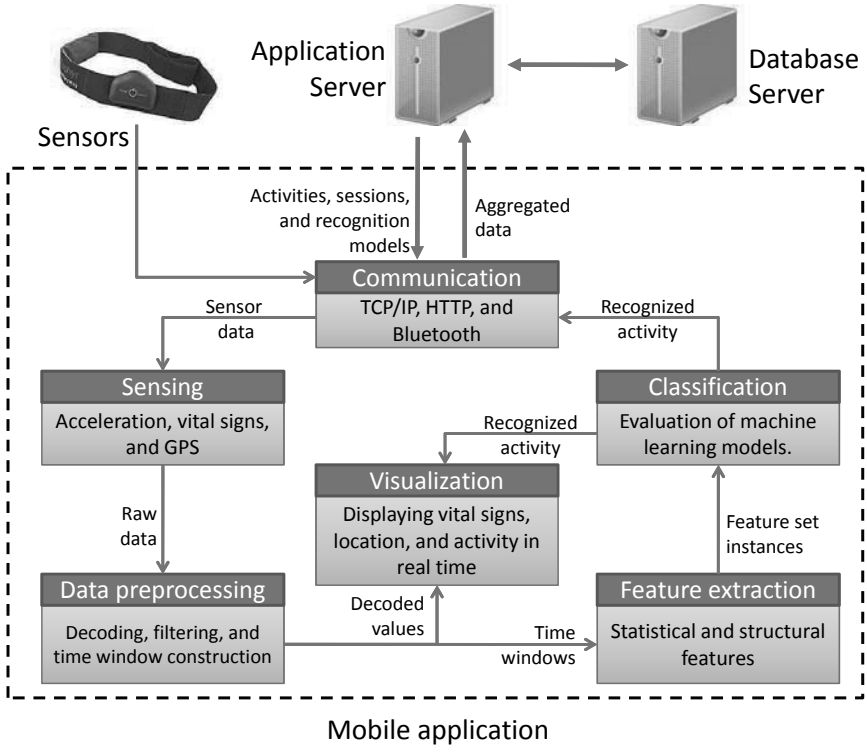

FIGURE 5.2: Mobile application design © 2012 IEEE. Reprinted, with permission, from [77].

decodes and stores them into the database. Finally, in the third level of communication, HTTP servlets allow for validating user credentials and querying the list of activities to be recognized, the list of features to be extracted, as well as the classification models to be used.

5.2.3 Sensing and data preprocessing

The sensing component manages and synchronizes the flow of data from all sensing devices, i.e., accelerometer, GPS, and the Bioharness BT strap. Figure 5.3 illustrates the interrelationship among the classes in this module. Sensors are represented as entities that extend from the abstract class *Sensor* and implement methods to *connect*, *read data*, and *finalize*. Additionally, they periodically report measurements to the *SensorManager* class through the Observer-Observable pattern [12]. With this model, new sensors can be easily incorporated in the system and they are able to work concurrently — in a different thread — as the class *Sensor* also implements the Java's *Runnable* interface.

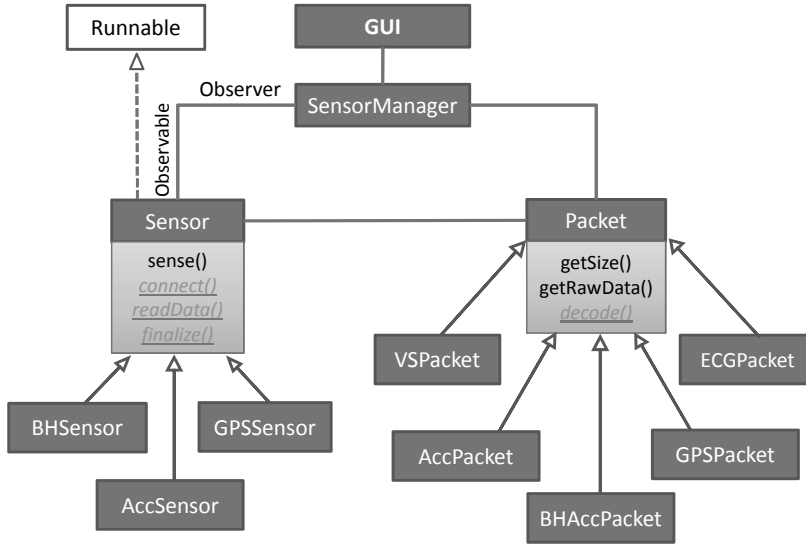

FIGURE 5.3: Simplified UML class diagram for the sensing component © 2012 IEEE. Reprinted, with permission, from [77].

Every piece of raw data is represented as a packet, which might contain one or several samples, according to the sensor specifications. Each specific type of packet extends from the abstract *Packet* class, and implements a particular *decode* method. The *SensorManager* class receives all packets and controls the data flow to achieve the recognition of activities.

5.2.4 Feature extraction and selection

This component provides an efficient implementation of statistical, structural, and transient feature extraction methods for nine attributes: heart rate, respiration rate, breath amplitude, skin temperature, posture, ECG amplitude, and tri-axial acceleration. The methods were presented in Section 4.1.2, accounting for a total of 84 features. Given the mobile devices' computational constraints, only the most relevant features have been chosen for real-time activity recognition. This process, denominated *feature selection*, is very useful in any machine learning context to simplify the model, eliminating redundant features which could even diminish the classification performance. The *correlation based feature selection* (CFS) algorithm [56] was employed in this work using the WEKA implementation. The selected features are listed in Table 5.1. Further methods such as Principal Component Analysis and Genetic Search Feature Selection [18] were also evaluated but they considerably affected the overall classification accuracy.

TABLE 5.1: Selected features for mobile activity recognition.

Feature	Description
$corr_{x,y}$	Correlation among acceleration signals in X and Y axes
$corr_{x,z}$	Correlation among acceleration signals in X and Z axes
MAD_x	Mean absolute deviation of the acceleration signal in the X axis
MAD_y	Mean absolute deviation of the acceleration signal in the Y axis
RMS_x	Root Mean Square of the acceleration signal in the X axis
RMS_y	Root Mean Square of the acceleration signal in the Y axis
σ_x	Standard deviation of the acceleration signal in the X axis
σ_x^2	Variance of the acceleration signal in the X axis
b_{HR}	Parameter b of the line $y = mt + b$ that best fits the heart rate signal
b_{ECG}	Parameter b of the line $y = mt + b$ that best fits the ECG amplitude signal
η_{RR}	Signed Magnitude of Change of the respiration rate signal

5.2.5 Classification

This module handles the evaluation of previously trained classification algorithms. Two different libraries, WEKA and MECLA, were integrated into Vigilante.

5.2.5.1 Toward mobile WEKA

The Waikato Environment for Knowledge Analysis (WEKA) is one of the most widely used research tools for machine learning. And, one of its most noticeable features is its Java API, which enables the integration of new classification and evaluation methodologies using the underlying core libraries. As part of this book, the WEKA API has been partially integrated to the Android platform. However, it was found that some functionalities are still not available on the smartphone.

Given that the training phase is generally more expensive than the evaluation, the classifiers used in this work were previously trained on the server. Now, to make them available on the smartphone, Vigilante uses the Java's object serialization API, which permits to convert a given class instance (i.e., an object) to a stream of bytes and vice versa. The object that is intended to be *serialized* should implement the *java.io.Serializable* interface. This is not an issue since the WEKA classifiers extend from the superclass *weka.classifiers.Classifier* which, in turn, implements the *java.io.Serializable* interface. The class *ObjectOutputStream* was used to export each trained classifier object as a binary file. Such a file is copied to the smartphone's persistent memory and then, by means of the class *ObjectInputStream*, the trained clas-

sifier instance is reconstructed from the stream of bytes. Chapter 14 explains this process in more detail and includes the Java code to implement it.

Nevertheless, running WEKA in the smartphone presents a couple of important disadvantages. In the first place, not all the classification methods are supported by the mobile platform. Secondly, the file *weka.jar* should be part of the Android application, increasing the size of the executable APK file in more than 6 MB. This is because WEKA libraries comprehend training algorithms and other functionalities that are not necessary for HAR purposes. This fact may also hinder a final application from being massively deployed.

5.2.6 MECLA

Given that WEKA is neither designed nor optimized for existing mobile platforms, a new library, MECLA, is proposed to evaluate classification models on the smartphone. The current implementation supports decision trees — as a proof of concept, thereby enabling the *C4.5*, *ID3*, *Decision Stump*, *Random Tree*, and *M5*, among other classification algorithms.

The design of the classification module is shown in Figure 5.4. Nodes correspond to features (e.g., mean acceleration in X) whereas edges define *relations* of an attribute with a numeric or nominal *value* (e.g., mean acceleration in X is greater than $2.382g$). The method *evaluate* of the class *Edge* returns whether or not a given value fulfills the edge's relational condition. This method is utilized by class *Node*'s *evaluate* method, which returns the next child node to be visited given a particular attribute value. In this way, the evaluation of a feature vector (i.e., an instance of the class *FeatureSet*) starts at the root node and follows the corresponding branches according to the output of the node evaluation methods. The process stops when the current node does not have any children and the associated activity class (e.g., running, walking, etc.) is returned.

The classification model was trained a priori in WEKA using the C4.5 algorithm. The alphanumeric representations of all decision trees, as given by the WEKA output, were stored in the database. An HTTP servlet allows the mobile application to query a decision tree model in accordance to the parameter values set by the user. An iterative algorithm was implemented to build a *DecisionTree* object (i.e., nodes and edges) based upon the alphanumeric representation retrieved by the servlet, in order to generate predictions on the user's activity.

5.2.7 User interface

Vigilante has three different user profiles: *trainer*, *tester*, and *administrator*. A tester is a regular user whose activities are to be monitored and recognized. A trainer is, as indicated by its name, intended to collect training data that can be used to build additional classification models. In this profile, the user is also required to enter the real performed activity as a ground truth. Finally,

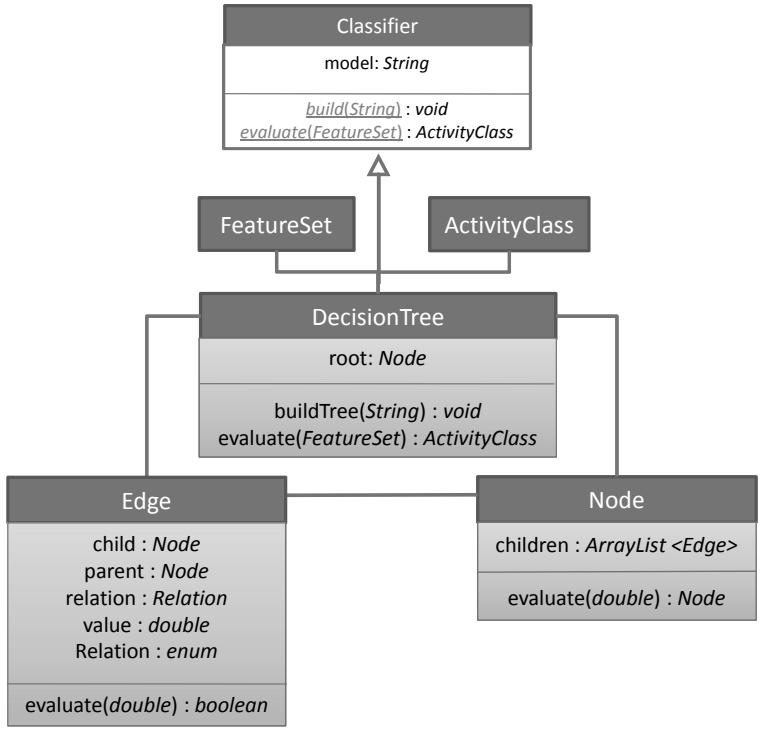

FIGURE 5.4: Simplified UML class diagram for the classification component © 2012 IEEE. Reprinted, with permission, from [77].

(A) (B)

FIGURE 5.5: Mobile application user interface © 2012 IEEE. Reprinted, with permission, from [77].

the administrator is able to do training, testing, and modifying the application settings.

Figure 5.5 displays the two main screens of Vigilante, namely the *monitor* (A) and the *configuration* (B). The former displays the sensed data (e.g., heart rate, respiration rate, skin temperature, etc.), the current user's activity, and the data collection time in real time. The latter allows the administrator to set parameters such as the window size, number of aggregated packets, maximum number of Bluetooth reconnections, and time between Bluetooth reconnections. It also permits turning on and off each individual sensor, as well as enabling or disabling data transmission to the server. A video of Vigilante and its capabilities can be watched in [17].

TABLE 5.2: Physical characteristics of the new participants.

	Avg.
Age (years)	20.3
Weight (kg)	65
Height (m)	1.68
BMI (kg/m^2)	26.4

5.3 Evaluation

5.3.1 Experiment design

Vigilante was tested on an HTC Evo 4G smartphone, being compatible with Android 2.1 or higher. Four individuals: I_1 through I_4 — three males and one female — performed each activity in a sequential fashion during two to five minutes. I_1 was part of the training data collection phase, while individuals I_2 through I_4 were new to the system. The physical characteristics of the new individuals are summarized in Table 5.2. A naturalistic experiment was carried out, i.e., no instructions were given to the participants on how to perform the activities.

Table 5.3 includes the most important parameters for all the experiments. The following classification methods were attempted on the smartphone using the WEKA libraries:

- *J48*: the C4.5 decision tree.
- *BayesNet*: a Bayesian Network classifier using the K2 search algorithm.
- *SMO*: the Sequential Minimal Optimization algorithm.
- *IBK*: the k-nearest neighbors classifier.
- *MultilayerPerceptron*: a neural network using backpropagation.
- *Logitboost*: the Additive Logistic Regression classifier.
- *Bagging*: an ensemble of ten *J48* base learners using the bootstrap aggregation technique.

It was found, nonetheless, that some of the WEKA's classifier implementations were not suitable for smartphones. Indeed, the *J48* (i.e., the C4.5 decision tree), *Bagging*, and the *MultilayerPerceptron* generated stack overflow exceptions in the process of de-serialization.

The reader may recall that Chapter 4 suggests ALR as the most appropriate classifier for the HAR problem presented in this book. Since that algorithm was supported by the Android platform, it was used for the experiments with

TABLE 5.3: Parameters to evaluate WEKA classifiers in smartphones.

Hardware	HTC Evo 4G
Operating system	Android 2.X
Dataset	D_{vs}
Features	12
Instances	629
Overlapping	50%

the WEKA API. On the other hand, the C4.5 algorithm was evaluated using the MECLA libraries. This is with the aim of comparing the performance of WEKA and MECLA. The evaluation encompasses three main aspects: accuracy, response time, and energy consumption.

5.3.2 Accuracy

It is worth highlighting that the evaluation presented in this book was accomplished online. Instead, previous studies which implemented real-time activity recognition calculate the confusion matrices and other performance metrics offline [32, 96], applying pre-processing and filtering the data. These two stages are helpful to remove noise but require human intervention, which makes them unavailable offline. Thus, the results presented in this section are more realistic.

To avoid potential mistakes when labeling the data, the assessment of the classification accuracy was done programatically. In training mode, each user set the *real activity* as the ground truth and performed said activity for a certain amount of time. Meanwhile, the application computes the *classified activity* for each time window using the ALR classifier powered by WEKA, as well as the MECLA's C4.5 classifier implementation. Given both the predicted and the real activity, the confusion matrices are directly calculated and displayed on the phone. The results for individual I_1 are shown in Table 5.4, where columns correspond to the real activity and rows represent the classified activity. The overall accuracy was very similar for both classifiers, i.e., 96.39% for ALR (A) and 96.84% for C4.5 (B), demonstrating the effectiveness of WEKA and MECLA. This seems to contradict the results of Chapter 4, which suggest that ALR is significantly more accurate than J48. Nevertheless, that study uses cross validation among all the users' data to measure the overall accuracy. In this case a user-specific analysis was done because individual I_1 also participated in the training phase, which favours higher accuracy levels for activity recognition [104].

A second experiment considers individuals I_2 and I_3, which did not participate in the training collection phase. Tables 5.5 (A) and (B) display the confusion matrices for both classifiers. Notice that here the overall accuracy is quite lower than for I_1 (between 64% and 69%). This was expected as the gait and the intensity of the activities are individual-specific, specially for *ascend-*

TABLE 5.4: Confusion matrix for individual 1 (5 activities).

(A)

ALR (WEKA). Overall accuracy: 96.39%					
	Running	**Walking**	**Still**	**Ascending**	**Descending**
Running	40	0	0	0	0
Walking	0	45	0	6	0
Still	0	0	46	0	0
Ascending	0	2	0	44	0
Descending	0	0	0	0	39

(B)

C4.5 (MECLA). Overall accuracy: 96.84%					
	Running	**Walking**	**Still**	**Ascending**	**Descending**
Running	40	0	0	0	0
Walking	0	51	0	0	0
Still	0	0	46	0	0
Ascending	0	5	0	41	0
Descending	1	1	0	0	37

ing and *descending*. Another important factor is that I_2 and I_3 have different physical characteristics than the individuals who collected training data for *ascending* and *descending*. In Chapter 7, the hypothesis of a group-specific data collection methodology is presented to overcome this issue.

The last experiment was accomplished with individuals I_2 and I_4 but only considering three activities: *walking, running*, and *still*. The classification model generated by the C4.5 algorithm is shown in Figure 5.6. Four features are part of the tree: (1) *MAD(AccX)*, i.e., the maximum absolute deviation of the acceleration in the X axis; (2) *RMS(Accx)*, i.e., the root mean square of the acceleration in the X axis; (3) *MoC(HR)*, i.e., the magnitude of change of the heart rate signal; and (4) *Slope(Temp)*, i.e., the slope of the line that best fits the skin temperature signal. In this case, even though these two individuals did not participate in the training phase, the system's accuracy was more than acceptable (92.25%). This brings up a very interesting point of discussion: some activities are user-specific, while others can be generalized for individuals with different physical characteristics. Therefore, to improve the recognition accuracy in this particular case study, the system would only need to be retrained for two activities: *ascending* and *descending*.

5.3.3 Response time

The response time was measured for each time window as the total time spent by preprocessing, feature extraction, and classification. The total average response time after issuing 100 time windows was about 171 ms. The most

TABLE 5.5: Confusion matrix for individuals 2 and 3 (5 activities).

(A)

ALR (WEKA). Overall accuracy: 69.38%					
	Running	**Walking**	**Still**	**Ascending**	**Descending**
Running	53	1	2	0	1
Walking	0	23	0	20	72
Still	0	0	124	0	0
Ascending	1	9	1	48	18
Descending	2	0	1	7	58

(B)

C4.5 (MECLA). Overall accuracy: 64.05%					
	Running	**Walking**	**Still**	**Ascending**	**Descending**
Running	70	1	2	0	2
Walking	0	35	0	1	79
Still	0	0	124	0	0
Ascending	1	8	1	11	56
Descending	10	0	1	3	54

TABLE 5.6: Confusion matrix for individuals 2 and 4 (3 activities).

C4.5 (MECLA). Overall accuracy: 92.25%			
	Running	**Walking**	**Still**
Running	123	0	0
Walking	5	112	21
Still	0	1	105

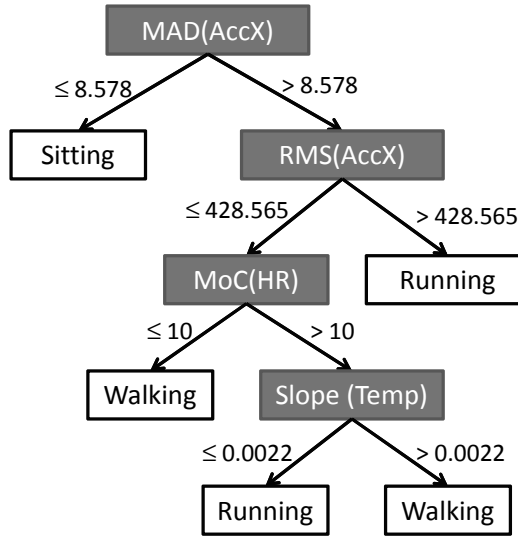

FIGURE 5.6: Classification model generated by the C4.5 algorithm with three activities © 2012 IEEE. Reprinted, with permission, from [77].

demanding stage was the preprocessing phase, which, on average, consumed 53% of the total response time. Feature extraction consumed about 45% of the time and classification required only roughly 2% of the total computation time. This was expected as the feature extraction algorithms run in $O(n)$, for 250 samples. On the other hand, the classification runs in $O(\log m)$ for the decision tree —for m nodes— while it requires constant time for the ALR classifier —since it evaluates ten one-level decision trees and then uses voting. Overall, the response time is less than 4% of the window length, confirming that Vigilante can be successfully deployed in current smartphones.

5.3.4 Energy consumption

One of the hypotheses of this chapter is that executing activity recognition locally in the phone, rather than sending all raw data to the server, is effective to save energy. In order to prove it, three cases were considered: (1) recognizing activities locally in the phone without sending raw data to the server; (2) sending all raw data to the server to recognize the activities remotely; and (3) not running the application to measure the energy consumed by the operating system and other phone services alone. The classes *Intent* and *BatteryManager* of the Android API were used to measure the battery charge difference after three continuous hours of use. Then, energy consumption was estimated given that the smartphone's battery works at 3.7 V and has a total charge of 1500 mAh. The results, shown in Table 5.7, indicate that performing local ac-

tivity recognition allows for increasing the application lifetime by 25%. More precisely, the system can run for up to 12.5 hours in case 1, versus 9.38 hours in case 2. Now, excluding the energy consumed by the operating system, the total energy savings in case 1 with respect to case 2 were roughly 26.7%. This result becomes remarkable as case 2 was already using data aggregation as an energy saving mechanism.

TABLE 5.7: Estimated energy consumption after executing the application for three hours © 2012 IEEE. Reprinted, with permission, from [77].

Case	Charge diff.	Current	Power	Energy
1	0.36 Ah (24%)	0.12 A	0.444 W	4795.2 J
2	0.48 Ah (32%)	0.16 A	0.592 W	6396.6 J
3	0.03 Ah (2%)	0.01 A	0.037 W	399.6 J

5.4 Concluding remarks

This chapter presents Vigilante as a mobile framework for real-time human activity recognition under the Android platform. The system features a library for mobile evaluation of classifiers (MECLA), which can be utilized in further machine learning applications as an alternative to WEKA. MECLA becomes useful since it enables decision-tree based algorithms, which are unavailable using WEKA libraries. The implementation and evaluation of Vigilante present the following main results and contributions:

- A library for the *mobile evaluation of classification algorithms* (MECLA) was designed and successfully implemented.
- The WEKA API was partially integrated in the Android platform to enable the evaluation of a number of classification algorithms for activity recognition.
- An application for real-time HAR was implemented in an Android smartphone. It uses MECLA as well as WEKA, and it supports multiple sensing devices integrated in a Body Area Network (BAN).
- The evaluation shows that the system can be effectively deployed on current smartphones in three main regards:
  - *Accuracy*: Human activities are recognized with an overall accuracy of up to 96.8%.

- *Response time*: The total computational time required for preprocessing, feature extraction, and classification accounts for less than 4% of the window length.
- *Energy consumption*: The application is able to run for up to 12.5 continuous hours and it enables energy savings of up to 27% with respect to a system that sends all the raw data to the server for remote processing.

Chapter 6

New Fusion and Selection Strategies in Multiple Classifier Systems

Pattern classification has been a very productive research area in past years. Innumerable applications can be visualized in forecasting, speech recognition, image processing, and bioinformatics, among others. In most cases, a single classification model is trained and evaluated fixing its parameters to maximize the classification accuracy. Still, selecting the best classification algorithm for a given dataset is not always an easy task. Even though cross validation and statistical hypothesis testing (e.g., 5×2-fold cross validation with a paired t-test) are often used to perform such selection, there are cases where no significant evidence can be found to assure that one classifier is better than another. Then, considering predictions not from one, but from a set of classifiers, turns out to be a good alternative to enrich a pattern recognition system. This is the main idea behind a *multiple classifier system* (MCS), which relies on the hypothesis that a set of *base classifiers* (i.e., the individual expert's part of the MCS) may provide more accurate and diverse predictions [74]. Of course, MCSs entail additional complexity, not only because a number of classifiers should be trained and evaluated, but also because they require defining criteria to generate a prediction from their outputs.

Since classification is one of the most important components in activity recognition, this chapter explores new strategies to combine a set of learners in a multiple classifier system. Particularly, the sort of MCS studied in this chapter is intended to solve the problem stated in Definition 1.3 (see Chapter 1). That definition assumes the following input is provided:

- A classification problem with a set $\Omega = \{\omega_0, .., \omega_{n-1}\}$ of n classes.
- A dataset with N instances.
- A set $\mathcal{D} = \{D_0, ..., D_{k-1}\}$ of classifiers.
- A set $S = \{s_0, ..., s_{k-1}\}$ with k predictions from each classifier for a given instance x, such that $s_i = D_i(x)$.
- A set $\mathcal{M} = \left\{M^0, ..., M^{k-1}\right\}$ of k confusion matrices for each classifier, where the sum of columns is the total number of predictions for each class and the sum of rows is the number of actual instances.

The goal is to find the correct label ω^* iff $\exists s_i \in S$ such that $\omega^* = s_i$.

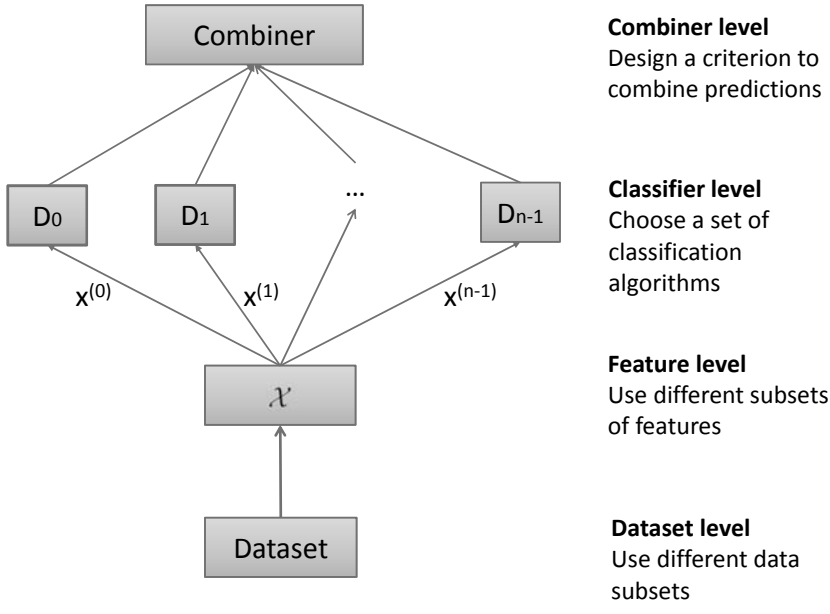

FIGURE 6.1: Different approaches in multiple classifier systems [74].

6.1 Types of multiple classifier systems

Although there is no consensus on the taxonomy of multiple classifier systems, most authors agree on four levels for building classifier ensembles, as it is shown in Figure 6.1:

- In the *dataset level*, the most noticeable methods are *bagging* and *boosting*. Bagging uses k identical classifiers, each with a different *bag* — a subsample of the training dataset. The bags are generally overlapping and the final decision is generated by voting. Instead, boosting iteratively builds a new classifier, assigning more weight to the instances that were incorrectly classified. In that way, new models become experts in feature subspaces where earlier models were not successful. As does bagging, boosting also uses a voting principle to output the final prediction.

- In the *feature level*, feature selection algorithms discussed in Section 2.2.1.5 yield different feature subsets which could be the input of each of the base classifiers in the ensemble.

- In the *classifier level*, different classification algorithms could be ap-

plied. These should be carefully selected since redundant or inaccurate classifiers might degrade the ensemble performance. Thus, quantitative criteria such as the *correlation* [54] and *disagreement* [62] are suggested practices to assure a successful MCS. In this book, we also present an algorithm to select base classifiers based on the concept of *level of collaboration*.

- Finally, in the *combiner level*, the main goal is to define a mechanism to estimate the correct label given a set of predictions from the base classifiers. This problem is more rigorously formulated in Definition 1.3.

This book focuses on the design of new combination and classifier level strategies in multiple classifier systems. Hence, the rest of this section is intended to cover the most relevant approaches in these categories.

6.2 Classifier-level approaches

A successful multiclassifier system should maintain base classifiers with high diversity. If the base classifiers are rather redundant, the ensemble would not deliver significant improvement. This is a hard problem to be solved deterministically as for k possible base classifiers, there are $O\left(2^k\right)$ possible subsets. Therefore, a variety of heuristic methods based on diversity metrics have been proposed. Two of the most common metrics are the *correlation* [54] and the *disagreement* [62].

Definition 6.1 ***Classifier correlation:*** *The pairwise correlation ρ_{ij} between classifiers D_i and D_j is defined as follows:*

$$\rho_{ij} = \frac{ad - bc}{\sqrt{(a+b)(c+d)(a+c)(b+d)}} \tag{6.1}$$

where:

- a is the number of instances correctly classified by both classifiers.
- b is the number of instances correctly classified by classifier D_i but incorrectly classified by classifier D_j.
- c is the number of instances correctly classified by classifier D_j but incorrectly classified by classifier D_i.
- d is the number of instances incorrectly classified by both classifiers.

Definition 6.2 ***Disagreement level:*** *The disagreement level between classifiers D_i and D_j is defined as follows:*

$$R(D_i, D_j) = \frac{b + c}{a + b + c + d} \tag{6.2}$$

Given these definitions, the idea is to choose a set of classifiers with low correlation values and high disagreement levels, thereby inducing more diversity in the ensemble. However, the combiner could be affected if diverse yet inaccurate classifiers are chosen. Section 6.5.4 provides a deeper understanding of this matter.

6.3 Combination-level approaches

There exist two main approaches to combine multiple base classifiers, namely *fusion* and *selection* [118]. The former works under the assumption that all base classifiers are competent in the entire feature space. Thus, voting and averaging are common methods in classifier fusion. In the latter, each classifier is assumed to be an expert in a certain feature subspace so the selection of a classifier usually requires to explore the neighborhood of the instance to be classified. According to Zhu et al. [117], classifier selection might be either *static* (i.e., the classifier is selected at training time) or *dynamic* (i.e., the classifier is selected at evaluation time). Dynamic selection usually yields more accurate predictions, yet it also entails higher computational complexity than static selection. In general, classifier fusion has been the focus of more profound research.

6.3.1 Classifier fusion

The very first and most intuitive approaches in decision fusion apply voting. Some examples are the *simple majority* (i.e., returning the prediction with more than a half of the votes) and the *plurality* (i.e., returning the prediction with the highest number of votes) [44]. Despite their simplicity, these have been shown to be effective in many learning problems [117]. However, if the accuracies of the base classifiers are very dissimilar, these two approaches may not be beneficial, as each base classifier has the same impact on the final decision. The weighted majority vote [74] was then introduced to overcome the issue by assigning a weight to each classifier according to an estimation of its prediction probabilities.

A different approach consists of estimating probabilities for each class using the confusion matrix. This is the main philosophy behind the *naïve Bayes combination* [74], which assumes conditional independence among the classifier predictions $S = \{s_0, ..., s_{k-1}\}$ given a class ω_i. The probability of each

class is calculated as follows:

$$P(\omega_j|S) \propto P(\omega_j) \prod_{i=0}^{k-1} P\left(s_i|\omega_j\right) \tag{6.3}$$

Then, the class with the highest probability is selected as the ensemble's prediction. One of the drawbacks of this method is that, if a classifier performs poorly on a certain prediction, it will highly affect the output of the ensemble.

6.3.2 Classifier selection

In classifier selection, the goal is to design a mechanism to always (or at least in most of the cases) choose the *best* classifier for a given input. Generally, this is achieved by estimating the *posterior probability* (i.e., the probability that a classifier's output is correct given a class) for each prediction in local regions of the feature space. Such regions can be determined by the classes themselves or could be defined by a vicinity criterion.

Classes as local regions: In the first case, the aim is to find the probability $P(\omega^* = \omega_j \mid s_i = \omega_j)$ that a prediction $s_i = D_i(x)$, corresponding to class ω_j, matches the correct class ω^*. By virtue of the Bayes' theorem, Giacinto et al. [53] expressed this probability as follows:

$$\hat{P}(\omega^* = \omega_j \mid s_i = \omega_j) = \frac{\hat{P}(s_i = \omega_j \mid \omega^* = \omega_j)\hat{P}(\omega^* = \omega_j)}{\hat{P}(s_i = \omega_j)} \tag{6.4}$$

Now, such probabilities could be estimated from the confusion matrix:

$$\hat{P}(\omega^* = \omega_j \mid s_i = \omega_j) = \frac{\frac{TP_{ij}}{TP_{ij}+FN_{ij}} \frac{TP_{ij}+FN_{ij}}{N}}{\frac{TP_{ij}+FP_{ij}}{N}} = \frac{TP_{ij}}{TP_{ij}+FP_{ij}} \tag{6.5}$$

where True Positives (TP_{ij}), False Positives (FP_{ij}) and False Negatives (FN_{ij}), are from the evaluation of classifier D_i on class ω_j. Observe that this result is nothing but the *precision* or *positive predictive value* of classifier y_j for class i. Using the values in the confusion matrix, Equation 6.5 can be rewritten as follows:

$$\hat{P}(\omega^* = \omega_j \mid s_i = \omega_j) = \frac{M^i_{\omega_j\omega_j}}{\sum_{r=0}^{n-1} M^i_{r\omega_j}} \tag{6.6}$$

where $M^i_{n\times n}$ is the confusion matrix for classifier D_i.

Vicinities as local regions: Woods et al. [112] proposed a dynamic classifier selection approach via *local accuracy estimates* (DCS-LA). Given an unknown instance x to be classified, the ensemble's output is given by the most accurate classifier in a local region defined by the k nearest neighbors

of x in the training set. Later, other approaches adopted this methodology, proposing several improvements and extensions [101, 74, 97]. Nonetheless, this family of methods exhibits a high computational cost, as each new instance to be classified should be compared to the entire training dataset — which could be fairly large. Moreover, this approach is very sensitive to noise and is highly biased by the size of the neighborhood, which is usually problem-dependent.

Hybrid methodologies have also been the subject of study. As an example, the approach proposed by Cavalin et al. [34] applies the concept of multistage organization for dynamic classifier selection, yet it is computationally expensive, requiring a hundred classifiers with different samples of the dataset and a genetic algorithm to select a prediction.

As stated before, an intuitive solution to the classification problem is to estimate the probability that each prediction $s_i \in S$ is correct and then select the prediction s^* with the maximum probability. Another approach is to rather estimate the probability of each class ω_i by combining the predictions of all classifiers. Nonetheless, selecting the prediction or the class with the highest estimated probability is not always an appropriate solution, especially when some classes are harder to distinguish than others. If we are dealing with recognizing physical activities, for instance, *sitting* and *running* are clearly differentiable, so higher probabilities are expected for these classes. Instead, *walking upstairs* and *walking downstairs* might be confusing in some cases [79], yielding to smaller probabilities. Therefore, always choosing the prediction with the highest probability value may bias the ensemble to select the easiest classes, thereby affecting the overall accuracy and diversity.

In the following sections, two new probabilistic strategies for fusion and selection in a multiclassifier system are presented to address the aforementioned issues. Both of them are based on simple heuristic rules, maintaining ease of implementation and low computational cost. An extensive analysis with seven classification algorithms and eleven datasets demonstrates that the proposed methods are effective to improve the classification accuracy with respect to the individual classifiers and other well known fusion and selection algorithms. Also, an algorithm to select base classifiers is presented in order to guarantee a significant accuracy improvement by the proposed strategies.

6.4 Probabilistic strategies in multiple classifier systems

6.4.1 Failure product

As it was shown before, the probability $\hat{P}(\omega^* = \omega_j \mid s_i = \omega_j)$ that a prediction $s_i = D_i(x)$, corresponding to class ω_j, is correct can be expressed as follows:

TABLE 6.1: Highest probability fallacy.

Prediction	Probability
$s_0 = \omega_1$	**0.9**
$s_1 = \omega_2$	0.85
$s_2 = \omega_2$	0.82
$s_3 = \omega_2$	0.8

$$\hat{P}(\omega^* = \omega_j \mid s_i = \omega_j) = \frac{M^i_{\omega_j \omega_j}}{\sum_{r=0}^{n-1} M^i_{r\omega_j}} \tag{6.7}$$

Given this result, it would sound logical to always select the prediction s^+ with the highest probability, as proposed by Giacinto et al. [53]:

$$s^+ = \arg\max_{s_i \in S} \left\{ \hat{P}(\omega^* = \omega_j \mid s_i = \omega_j) \right\} \tag{6.8}$$

Nonetheless, this estimation is highly biased when some classes are harder to predict than others, affecting the diversity of classification and, of course, the overall accuracy. This approach may also make an incorrect decision if several classifiers support a prediction s_r different than s^+. Consider the case in Table 6.1. Classifier D_0 predicted class ω_1 with the highest probability (i.e., 0.9). But all other classifiers selected ω_2 with high probability (i.e., among 0.8 and 0.85). In this scenario, prediction $s_0 = \omega_1$ is very likely to be wrong although it has the maximum probability.

In such direction, the following new metric is presented. From Equation 6.7, the probability that prediction s_i is not correct, would be given by:

$$\hat{P}(\omega^* \neq \omega_j \mid s_i = \omega_j) = 1 - \frac{M^i_{\omega_j \omega_j}}{\sum_{r=0}^{n-1} M^i_{r\omega_j}} \tag{6.9}$$

Definition 6.3 ***Failure product:*** *The* failure product $FP(\omega_j)$ *for a given class* ω_j *is defined as follows:*

$$FP(\omega_j) = \begin{cases} \infty & \text{if } s_i \neq \omega_j \forall s_i \in S \\ \prod_{i \mid s_i = \omega_j} \left[1 - \frac{M^i_{\omega_j \omega_j}}{\sum_{r=0}^{n-1} M^i_{r\omega_j}} \right] & \text{otherwise} \end{cases} \tag{6.10}$$

TABLE 6.2: Precision vs. recall fallacy. $D_0(x) = s_0 = \omega_1$ and $D_1(x) = s_1 = \omega_0$

	Classifier D_0		Classifier D_1	
Class	Precision	Recall	Precision	Recall
ω_0	0.9	0.5	**0.7**	0.8
ω_1	**0.8**	0.6	0.4	**0.99**
ω_2	0.4	0.7	0.5	0.5

Then, the prediction s^*_{FP} with the smallest failure product will be the output of the multiple classifier system:

$$s^*_{FP} = \arg\min_{\omega_j \in \Omega} \{FP(\omega_j)\} \tag{6.11}$$

6.4.2 Precision recall difference

As it was seen, the precision is a natural estimation for the probability that a prediction s_i is correct. However, better estimations could be done by also considering the recall of those classifiers which have a different prediction than s_i. Observe the situation in Table 6.2. For a given instance x, the prediction s_0 by classifier D_0 is ω_1 whereas the output s_1 of D_1 is ω_0. Notice that the precision for prediction s_0 (i.e., 0.8) is higher than the precision of s_1 (i.e., 0.7). However, D_1 has a very high recall for class ω_1 (i.e., 0.99) so it is very unlikely that D_1 misses an instance of class ω_1. Hence, a more informed decision should not ignore $s_1 = \omega_0$ as a potential prediction.

The recall of classifier D_i in a particular class can be estimated from the confusion matrix in the same fashion as the precision:

$$recall_i(s_i) = \hat{P}(s_i = \omega_j \mid \omega^* = \omega_j) = \frac{TP_{ij}}{TP_{ij} + FN_{ij}} = \frac{M^i_{\omega_j \omega_j}}{\sum_{r=0}^{n-1} M^i_{\omega_j r}} \tag{6.12}$$

Now, a metric that incorporates both precision and recall is defined to estimate the quality of each base classifier's prediction.

Definition 6.4 ***Precision-Recall difference:*** *The precision-recall* difference $PRD(s_i)$ *for a given prediction* s_i *is defined as follows:*

$$PRD(s_i) = \frac{M^i_{\omega_j \omega_j}}{\sum_{r=0}^{n-1} M^i_{r \omega_j}} - \max_{t | s_t \neq \omega_j} \left\{ \frac{M^t_{\omega_j \omega_j}}{\sum_{r=0}^{n-1} M^t_{\omega_j r}} \right\} \tag{6.13}$$

The first term of this difference is nothing but the precision of classifier D_i

TABLE 6.3: Dataset specifications.

Dataset	Attributes	Classes	Instances	Source
balance-scale	4	3	625	[15]
diabetes	8	2	768	[15]
glass	9	6	135	[15]
$D_{acc+tra}$	90	5	619	[79]
D_{acc}	90	5	277	[79]
ionosphere	34	2	351	[15]
lymph	18	4	148	[15]
segment	19	7	2310	[15]
sonar	60	2	208	[15]
soybean	35	19	683	[15]
vehicle	18	4	846	[15]

for class $s_i = \omega_j$. The second term is the maximum recall of all other classifiers with a prediction different than s_i. The main idea behind this strategy is to penalize predictions that were rejected by classifiers with a high recall. It was experimentally found that the *max* operator performs better than the average in this case. The value of the precision-recall difference is in the interval $[-1, 1]$ and the prediction s^*_{PR} with the maximum PRD will be returned by the MCS:

$$s^*_{PR} = \arg \max_{0<i<k-1} \{PRD(s_i)\} \tag{6.14}$$

6.5 Evaluation

6.5.1 Experiment design

An experimental analysis was carried out to assess the effectiveness of the proposed strategies. All the ensembles were implemented in Java using the WEKA API [18]. Eleven different datasets were evaluated in this work. Most of them are part of the UCI machine learning repository [15]. The datasets and their characteristics are described in Table 6.3.

Seven base classification algorithms were considered, namely Naïve Bayes (NB), Bayesian Network (BN), Instance Based Learning (IBK), Repeated Incremental Pruning (RIP), C4.5 decision tree, Logistic Regression (LR), and Sequential Minimum Optimization (SMO).

The two proposed strategies were compared to the following three well-known classifier fusion and selection approaches:

- **Naïve Bayes Combination (NBC)** [74]: This scheme estimates the probability of each class under the assumption that the classifiers are

conditionally independent given a class ω_i, using Equation 6.3. The class with the maximum probability is chosen as the ensemble's output.

- **Plurality (PL)** [44]: This approach evaluates all classifiers and selects the prediction with the greatest number of votes.
- **Local Class Accuracy (LCA)** [53]: This method chooses the prediction with the highest precision (see Equation 6.6).
- **Oracle (ORA):** The oracle is the optimal selector and it returns the correct label s^* if $s^* \in S$. Otherwise, it returns a random prediction s_i. Evidently, the oracle is not practical to be implemented in real applications as it requires a fully labeled dataset. However, it is shown as an upper bound for the classification accuracy.

Each dataset was divided in three folds: *training set, estimation set,* and *evaluation set,* as suggested by [46]. As usual, the first one was used to train the base classifiers. Then, the posterior probabilities were estimated from the confusion matrices of each base classifier after using the second fold. Finally, the third fold served to evaluate the performance of both base classifiers and ensembles. The evaluation was repeated three times using each fold as training set and randomly dividing the remaining dataset in two halves (one for estimation and another for evaluation). This entire process was completed for five different random seeds to provide statistical robustness, and the mean values were calculated.

6.5.2 Results

Table 6.4 displays the percentage average accuracies for each dataset given by all base classifiers and ensembles. The best accuracy values per dataset are in bold, the best ensembles are marked with an asterisk (*), and the best base classifiers are marked with a dot (•).

Note that in eight out of eleven datasets, at least one of the ensembles improved the overall classification accuracy with respect to the base classifiers. In six datasets, the FP strategy achieves the highest accuracy while in four of them, the PRD is the best ensemble. In nine cases, the accuracy of PRD is higher than LCA and PL, whereas in ten cases, FP outperforms LCA and PL. This shows that the proposed strategies are effective in improving the classification accuracy with respect to previously proposed fusion and selection methods. However, in three datasets (*vehicle*, *soybean*, and *balance-scale*), none of the ensembles improved the performance of the best base classifier. It was found that the base classifier set plays a significant role in such matters. This situation is examined next.

TABLE 6.4: Percentage average accuracies.

	glass	diabetes	ionosphere	$D_{acc+tra}$	D_{acc}	sonar	lymph	vehicle	segment	balance	soybean
NBC	39.00	67.79	79.83	83.33	65.48	68.36	63.96	47.75	71.17	80.33	56.4
PRD	**58.25***	73.07	86.44	92.29	81.57	75.98	**78.39***	66.5*	74.21	90.16*	**66.81***
FP	**58.25***	**73.54***	**90.26***	**93.11***	**82.3***	**76.38***	77.45	63.74	75.45	89.55	66.49
LCA	**58.25***	72.19	88.26	92.15	78.23	73.78	77.32	62.70	72.35	89.3	66.73
PL	54.93	71.64	85.36	92.52	80.98	73.90	77.34	63.00	**75.84***	85.92	67.49
ORA	81.17	86.59	96.13	97.48	90.73	96.85	93.70	82.25	78.55	96.84	70.41
NB	38.45	71.22	76.81	86.58	71.42	67.77	77.47•	41.56	66.44	83.37	65.00
BN	51.48	69.30	86.67•	88.36	74.61	66.16	76.65	52.01	72.74	67.61	65.85
RIP	47.29	68.26	82.28	83.17	71.27	65.76	66.76	55.82	73.66	76.84	63.36
C4.5	53.71	66.07	82.45	89.56	77.01	66.53	65.68	59.36	74.88•	75.98	63.48
LOG	51.85	71.77•	80.17	82.58	76.76	69.47	74.22	**67.80•**	74.87	**90.8•**	65.73
SMO	41.17	70.81	79.54	86.37	70.63	72.26	76.53	56.31	72.86	87.77	**67.51•**
IBK	53.99•	65.26	79.26	92.67•	78.62•	75.99•	74.64	57.04	74.73	77.96	65.20

TABLE 6.5: Correlation matrix for the *soybean* dataset.

	NB	**BN**	**RIP**	**C4.5**	**LOG**	**SMO**	**IBK**
NB	1	0.96	0.87	0.87	0.88	0.89	0.91
BN	0.96	1	0.86	0.86	0.88	0.90	0.90
RIP	0.87	0.86	1	0.86	0.84	0.86	0.86
C4.5	0.87	0.86	0.86	1	0.85	0.88	0.86
LOG	0.88	0.88	0.84	0.85	1	0.91	0.87
SMO	0.89	0.90	0.86	0.88	0.91	1	0.90
IBK	0.91	0.90	0.86	0.86	0.87	0.90	1

6.5.3 Correlation analysis

A successful multiclassifier system should maintain base classifiers with high diversity. If the base classifiers are rather redundant, the ensemble would not deliver a significant improvement. A well-known measure of diversity is the pairwise correlation ρ_{ij} [54] between classifiers D_i and D_j, defined as follows:

$$\rho_{ij} = \frac{ad - bc}{\sqrt{(a+b)(c+d)(a+c)(b+d)}} \tag{6.15}$$

where:

- a is the number of instances correctly classified by both classifiers.
- b is the number of instances correctly classified by classifier D_i but incorrectly classified by classifier D_j.
- c is the number of instances correctly classified by classifier D_j but incorrectly classified by classifier D_i.
- d is the number of instances incorrectly classified by both classifiers.

For each dataset, the correlation matrix $P = [\rho_{ij}]_{k \times k}$ has been calculated. An interesting phenomenon was found for the *soybean* dataset (see Table 6.5). Observe that all the correlation values are very high (between 0.84 and 0.96). This means that all the classifiers yielded the same predictions for most of the instances, which does not allow for a successful classifier ensemble. Note that even the oracle only improved the classification accuracy by less than 3% in regards to SMO (i.e., the best base classifier). Such analysis suggests that no successful ensemble can be built with the given set of base classifiers. Now, the *vehicle* and the *balance-scale* datasets do not follow a clear correlation pattern, so a different metric will be utilized to analyze them.

6.5.4 Collaboration among classifiers

Although low correlation is a desirable feature, it is not sufficient to guarantee a successful MCS. Imagine, for instance, that an additional classifier which

outputs uniformly random predictions is included in the ensemble. That classifier is expected to have a very low correlation with all other classifiers, but its predictions are also expected to be very inaccurate, especially when there are many classes. Of course, such a random classifier could affect the overall performance of the ensemble and should be avoided.

Now, in order to properly select the set of the base classifiers, the *collaboration matrix* B is introduced such that the element $0 \leq b_{ij} \leq 1$ is the proportion of instances correctly classified by classifier D_i but incorrectly classified by classifier D_j. Table 6.6 shows the B matrix for the *balance-scale* dataset; for this dataset, the most accurate classifier was LOG. Note that, in 27% of the cases, the BN classifier disagrees with LOG while the latter is giving correct predictions. At the same time, in only 4% of the cases, the predictions by BN are correct while LOG's are not. This means that including the BN classifier does not seem to be beneficial but rather detrimental for the ensemble. On the other hand, SMO erroneously disagrees with LOG in only 5% of the instances, but correctly disagrees with LOG in 2% of the cases. Therefore, SMO is more likely to help LOG and contribute to the ensemble performance.

TABLE 6.6: B matrix for the *balance-scale* dataset.

	NB	**BN**	**RIP**	**J48**	**LOG**	**SMO**	**IBK**
NB	0	0.18	0.11	0.12	0.03	0.02	0.10
BN	0.02	0	0.07	0.07	**0.04**	0.02	0.06
JRIP	0.05	0.16	0	0.07	**0.05**	0.03	0.07
J48	0.05	0.15	0.07	0	**0.05**	0.04	0.06
LOG	0.11	**0.27**	**0.19**	**0.20**	0	0.05	0.17
SMO	0.06	0.23	0.14	0.15	0.02	0	0.13
IBK	0.04	0.17	0.09	0.08	0.04	0.03	0

Then, the *level of collaboration* $-1 \leq \kappa_{ij} \leq 1$ of classifier D_i to classifier D_j is defined as follows:

$$\kappa_{ij} = b_{ij} - b_{ji} \tag{6.16}$$

Greater values of κ_{ij} indicate more chances of D_j being able to help D_i. Algorithm 6.1 formalizes the proposed procedure to select the base classifiers. Initially, the only member of the ensemble is the most accurate classifier (line 3). Then, classifier D^+ — which has the greatest average level of collaboration to all $D_j \in \mathcal{E}$ — is iteratively appended to the set $\mathcal{E} \subseteq \mathcal{D}$, containing the suggested classifiers to be part of the ensemble.

Algorithm 6.1 ***Base classifier selection based on classifier*** **level of collaboration** ***(LOC):*** *Be $\mathcal{D} \equiv$ the pool of all possible k base classifiers of the ensemble and $\mathcal{E} \equiv$ the suggested set of base classifiers*

1. *Compute the $B_{k \times k}$ matrix for all classifiers in $\mathcal{D}$*

2. $D^* \leftarrow$ *most accurate classifier in* $\mathcal{D}$
3. $\mathcal{E} \leftarrow \{D^*\}$
4. $\mathcal{D} \leftarrow \mathcal{D} - \{D^*\}$
5. ***while*** $\mathcal{D} \neq 0$
6. $\quad D^+ \leftarrow \arg\max_{D_i \in \mathcal{D}} \left\{ \frac{1}{\|\mathcal{E}\|} \sum_{D_j \in \mathcal{E}} \kappa_{ij} \right\}$
7. $\quad$ ***if*** *(accuracy of* $\mathcal{E}$*) <* *(accuracy of* $\mathcal{E} \cup \{D^+\}$*)* ***then***
8. $\quad\quad \mathcal{E} \leftarrow \mathcal{E} \cup \{D^+\}$
9. $\quad\quad \mathcal{D} \leftarrow \mathcal{D} - \{D^+\}$
10. $\quad$ ***else***
11. $\quad\quad$ ***return*** $\mathcal{E}$
12. $\quad$ ***end if***
13. ***end while***
14. ***return*** $\mathcal{E}$

After executing the LOC selection algorithm to the *balance-scale* dataset, four classifiers were selected, namely LOG, SMO, NB, and IBK. With this setting, the maximum accuracy was 91.12%, given by the PRD strategy, which is higher than the 90.8% reached by the individual LOG classifier. Figure 6.2 shows the variation of the accuracy versus the number of classifiers using the level of collaboration criterion. Observe that the oracle is always improved when a new classifier is appended to the pool, but this is not the case for the other ensembles. FP reaches its best accuracy for five classifiers, while LCA and PRD achieve their maximum potentials with four classifiers.

Finally, Table 6.7 displays the B matrix for the *vehicle* dataset. In this case, the best base classifier is also LOG. In 31% of the cases, the LOG classifier predicts the correct class while NB does not, whereas in only 5% of the cases, NB helps LOG. A similar situation occurs for all other classifiers, thus, the level of collaboration with respect to LOG is so low that the ensemble cannot be successful unless different classifiers or different sampling procedures are incorporated.

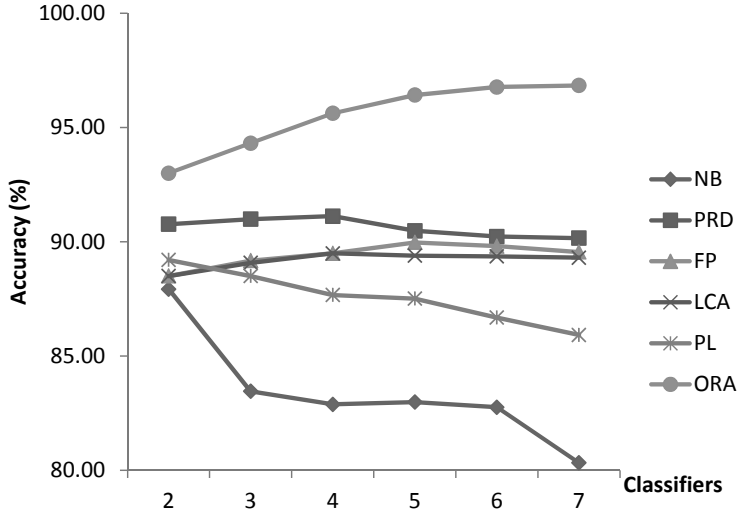

FIGURE 6.2: Overall accuracy of the ensembles varying the number of classifiers.

TABLE 6.7: B matrix for the vehicle dataset.

	NB	**BN**	**RIP**	**J48**	**LOG**	**SMO**	**IBK**
NB	0	0.07	0.10	0.10	**0.05**	0.09	0.11
BN	0.17	0	0.11	0.09	**0.06**	0.09	0.12
JRIP	0.25	0.15	0	0.09	**0.06**	0.10	0.12
J48	0.27	0.17	0.13	0	**0.07**	0.13	0.13
LOG	**0.31**	**0.22**	**0.18**	**0.16**	0	**0.16**	**0.18**
SMO	0.23	0.13	0.11	0.10	**0.04**	0	0.12
IBK	0.26	0.17	0.13	0.11	**0.07**	0.13	0

6.5.5 MCS in HAR

This chapter has shown that the proposed strategies improve the classification accuracy in a number of machine learning problems. Even for the HAR datasets, the FP and PRD produce — on average — more accurate results than the base classifiers and other ensembles. Nevertheless, even the highest accuracy achieved by FP in HAR (i.e., 93.10%) is not significantly better than for the ALR algorithm presented in Chapter 4 (i.e., 93.13% in a 5×2-fold cross validation and up to 95.37% in a 5×10-fold cross validation) (See Table 4.4 and Section 4.2.2.3). Since ALR achieves such a high accuracy, attempting to outperform it could cause overfitting; in other words, *noise* (i.e., instances with incorrect labels) would become part of the model, thereby degrading the overall accuracy of the classifier in unseen data. As a matter of fact, one of the issues with the ALR algorithm and other approaches based on boosting is that they poorly tolerate noise [111]. This is because boosting iteratively assigns higher weights to incorrectly classified instances, forcing base classifiers to make such instances part of the model. And, if there is a considerable amount of noise, the classifier will be overfitted.

6.5.5.1 Impact of noise in HAR

In most machine learning problems, labeling the data requires human intervention, making this process error prone. A more detailed analysis of the impact of noise in pattern classification is presented in [85]. Such work evaluates different types of attribute noise and class noise with a number of classification algorithms within different paradigms. Particularly, in a human activity recognition context, label errors could occur due to different factors. A malicious or lazy user who is required to follow a routine of exercise could report training data labeled as *running* while he or she is actually *lying* or *sitting*. Also, an incorrect use of the mobile applications presented in Chapters 4 and 5 could lead to mistaken labels if the user selects the wrong activity or they change activities before pressing the *stop* button. These situations emphasize the need for classification algorithms that are able to handle noise in activity recognition.

6.5.5.2 Experiments

In this study, noise was induced by arbitrarily modifying the labels of a subset $Y \in D_{acc+tra}$. The new labels and the instances in Y were chosen uniformly at random whereas the size of Y was varied between 0% and 25%. The MCSs were composed by four classifiers, namely MLP, RIP, SMO, and BN. These were chosen after an experimental analysis. The evaluation results are summarized in Table 6.8. As it was expected, the classification accuracy diminishes as the level of noise increases. Interestingly, some algorithms are more noise tolerant than others. While LOG, IBK, ALR, and NB are significantly affected by only introducing 5% of noise (accuracy drops down by up

TABLE 6.8: Accuracy of base classifiers and ensembles with different noise levels.

	0%	5%	10%	15%	20%	25%
MLP	90.59	85.4	77.98	70.14	65.49	55.11
C4.5	89.56	81.18	73.77	65.64	60.29	50.88
IBK	92.37	84.07	75.17	65.26	61.05	50.89
LOG	82.58	71.18	60.66	51.82	46.31	43.77
RIP	83.1	76.75	73.19	67.55	62.52	56.51
BN	88.36	84.16	77.92	75.48	69.33	62.07
SMO	86.37	80.6	75.84	72.14	64.96	59.03
NB	86.58	73.55	60.09	58.66	53.19	51.77
ALR	**92.89**	86.3	77.32	73.25	65.86	57.39
BJ48	88.43	84.52	78.88	74.74	68.66	61.77
LCA	90.66	87.99	79.9	71.85	66.51	58.96
NBC	78.43	77.21	75.25	71.77	66.21	58.21
PLU	90.81	85.63	79.99	75.93	69.33	62.73
PRD	90.96	87.03	80.13	72.75	69.4	60.88
FP	91.25	**88.3**	**81.76**	**76.74**	**70.81**	**65.56**

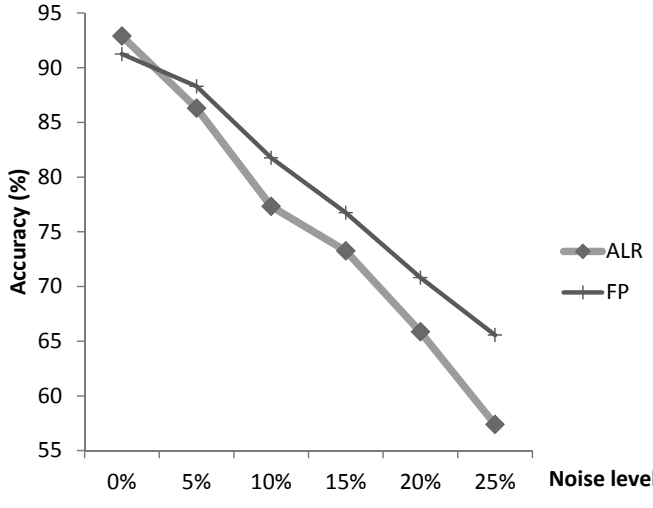

FIGURE 6.3: Overall accuracy of ALR and FP with different noise levels.

to 20%), MLP and BN are able to handle noise more effectively. But note that in all the cases, the FP strategy achieves higher accuracy than all other 15 algorithms — including ALR — for all noise levels. This fact demonstrates that FP is more noise tolerant than the base classifiers and ensembles. Finally, the overall accuracy of ALR and FP is plotted in Figure 6.3.

6.6 Concluding remarks

In this chapter, two probabilistic strategies are proposed for decision selection and fusion in multiple classifier systems. The evaluation results support the hypothesis that the proposed methods significantly improve the classification accuracy with respect to the base classifiers and other selection and fusion methods. Furthermore, an algorithm to select the best subset of base classifiers (LOC) is presented, not only reducing the complexity, but also improving the classification accuracy. More specifically, in a HAR context, the proposed strategies are demonstrated to be more effective to handle label noise.

Chapter 7

Conclusions

This chapter comprises two parts: First, it summarizes the most relevant findings and results presented throughout first part of the book. Then, a number of ideas are proposed for future research consideration in order to extend the field of Human Activity Recognition towards more realistic and pervasive scenarios.

7.1 Summary of findings and results

Chapter 3 presents a two-level taxonomy of HAR systems according to their response time and learning approach. Twenty eight systems are qualitatively compared under a number of design issues such as obtrusiveness, flexibility, recognition accuracy, and energy consumption, among others.

In Chapter 4, Centinela [79] is presented as an effective human activity recognition system combining acceleration along with physiological signals. Five activities are recognized by Centinela: *running*, *walking*, *still*, *walking upstairs* and *walking downstairs*. An experimental evaluation supports the hypothesis that vital signs are beneficial to improve the recognition accuracy of human activities. The evaluation also indicates that the Additive Logistic Regression classifier using 5s time windows with 50% overlap yields the highest accuracy (i.e., up to 95.7%). To achieve the peak accuracy, time- and frequency domain features should be extracted from acceleration signals whereas two features should be calculated from physiological signals: (1) the signed magnitude of change η and (2) the parameter b of the line $y = m(t) + b$ that best fits the points in the signal. Another important point of discussion is the placement of the sensor. Placing the accelerometer on the individual's chest had not been quite explored but it certainly avoids confusions that may arise if it is placed on the wrist [69].

In Chapter 5, the recognition of activities is taken one step further, introducing Vigilante [76, 77] as a new platform for real-time HAR. The system partially integrates the WEKA API in the Android platform to enable the evaluation of a number of classification algorithms. A new library, MECLA, is also proposed to enable tree-based classification algorithms in mobile devices — which are not fully functional with WEKA. The evaluation shows that

Vigilante can be effectively deployed on current smartphones with regard to *accuracy*, recognizing activities with an overall accuracy of up to 96.8%; *response time*, since its computational time accounts for less than 8% of the window length; and *energy consumption*, enabling energy savings of up to 27% with respect to a system that sends all the raw data to the server for remote processing.

Finally, Chapter 6 proposes and evaluates two new probabilistic strategies for decision selection and fusion in multiple classifier systems. The evaluation results support the hypothesis that the proposed methods significantly improve the classification accuracy with respect to the base classifiers and other selection and fusion methods. An algorithm to select the best subset of base classifiers is also included, allowing to reduce the complexity and improve the classification accuracy.

7.2 Future research considerations

In order to realize the full potential in HAR systems, some topics need further investigation. Next, a list of those topics is included.

- **Activity recognition datasets:** The quantitative comparison of HAR approaches has been hindered by the fact that each system works with a different dataset. While in research areas such as *data mining* there exist standard datasets to validate the effectiveness of a new method, this is not the case in activity recognition. Each research group collects data from different individuals, uses a different activity set, and utilizes a different evaluation methodology. In that direction, datasets should be publicly open to the research community to be used as benchmarks to evaluate new approaches. Several universities and institutions have published their datasets in [5, 3, 4, 2]. Another dataset is provided by the 2011 Activity Recognition Challenge [1], in which researchers worldwide were invited to participate.

- **Composite activities:** The activities explored in this book are rather simple. In fact, many of them could be part of more complex routines or behaviors. Imagine, for example, the problem of automatically recognizing when a user is *playing tennis*. Such an activity is composed by several instances of *walking*, *running*, and *sitting*, among others, with certain logical sequence and duration. The recognition of these *composite* activities from a set of *atomic* activities would surely enrich context awareness but, at the same time, brings additional uncertainty. Blanke et al. [29] provide an overview on this topic and propose a solution through several layers of inference.

- **Concurrent and overlapping activities:** The assumption that an individual only performs one activity at a time is true for basic ambulation activities (e.g., walking, running, lying, etc.). In general, human activities are rather overlapping and concurrent. A person could be *walking* while *brushing their teeth*, or *watching TV* while *having lunch*. Since only few works have been reported in this area, we foresee great research opportunities in this field. The interested reader might refer to the article of Helaoui et al. [61] for further information.

- **Multiattribute classification:** The purpose of a HAR system is, of course, providing feedback on the user's activity. But, context awareness may be enriched by also recognizing a user's personal attributes. A case study could be a system that not only recognizes an individual *running*, but also identifies them as a female between 30 and 40 years old. We hypothesize that vital signs may have an important role in the determination of these attributes. To the best of our knowledge, there is no previous work on this topic.

- **Cost-sensitive classification:** Imagine an activity recognition system monitoring a patient with heart disease who cannot make significant physical effort. The system should never predict that the individual is *sitting* when they are actually *running*. But confusion between activities such as *waking* and *sitting* might be tolerable in this scenario. *Cost-sensitive classification* works exactly in that direction, maintaining a cost matrix C where the value C_{ij} is the cost of predicting activity i given that the actual activity is j. The values in this matrix depend on the specific application. At prediction time, the classifier can be easily adapted to output the activity class with the smallest misclassification cost. Also, at training time, the proportion of instances can be increased for the most expensive classes, forcing the learning algorithm to classify them more accurately. Additional information on cost-sensitive classification is available in [111, 47, 114].

- **Crowd-HAR:** The recognition of human activities has been somehow individualized, i.e., the majority of the systems predict activities in a single user. Although information from social networks has been shown effective to recognize human behaviors [75], recognizing collective activity patterns can be taken one step further. If we could gather activity patterns from a significant sample of people in certain area (e.g., a city, a state, or a country), that information could be used to estimate levels of sedentarism, exercise habits, and even health conditions in a target population. Furthermore, this sort of *participatory-human-centric* application would not require an economic incentive method. The users would

be willing to participate in the system as long as they receive information on their health conditions and exercise performance, for example. Such data from thousands or millions of users may also be used to feed classification algorithms, thereby enhancing their overall accuracy.

- **Predicting future activities:** Previous works have not only estimated activities but also behavior routines [110]. Based on this information, the system could predict what the user is about to do. This becomes especially useful for certain applications such as those based on advertisements. For instance, if the user is going to *have lunch*, he or she may receive advertisement on restaurants nearby.

- **User flexibility:** People certainly perform activities in a different manner due to particular physical characteristics. Thus, acceleration signals measured from a child versus an elderly person are expected to be quite different. As it was seen in Section 2.1.8, a human activity recognition model might be either *monolithic* or *user-specific*, each one having its own benefits and drawbacks. A middle ground to make the most of both worlds might be creating *group-specific* classifiers, clustering individuals with similar characteristics such as age, weight, gender, and health conditions, among others. Then, our hypothesis is that a HAR system would be more effective having a recognition model for *overweight young men*, one for *normal male children*, another one for *female elderly*, and so forth.

Part II

HAR in an Android Smartphone: A Practical Guide

This part of the book is included to help readers develop human activity recognition systems in Android-based smartphones. After a brief introduction about Android and how to set up the software development environment in your computer, the following chapters explain how to connect the smartphone to external sensors, store information, and transmit data using the Bluetooth interface, all critical tasks that most HAR-based applications need to perform. Finally, two additional sections are included to show how to perform the feature extraction process and how to integrate Weka in an Android smartphone.

Chapter 8

Introduction to Android

The Android operating system for mobile devices is mainly the response from Google to the slow evolution of the Java ME platform for constrained devices and the carrier-controlled process to make mobile applications available to the general public. At that time, back in 2005, the Java ME platform was the platform most widely used to develop applications for resource constrained devices, such as cellular phones. Although there are still many business-oriented applications developed in Java ME, this platform presented several important drawbacks:

- Java ME applications were very tightly controlled by carriers. Applications could not be made available for download freely but needed the authorization of the carrier that provided the cellular service to the application users.
- Java ME was designed around a device whose main functionality was to handle cellular phone calls, a phone-centric design. Nowadays, smartphones are rather general purpose computers that also handle cellular phone calls. Android was designed from the ground up to be run in a computer, not in a cellular phone.
- The debugging process in Java ME was tedious, as the application had to be compiled, signed, uploaded, and then downloaded over the air to run it and test it in the real device. No on-device debugging was possible.
- Java ME had a very limited support for user interfaces and very limited persistent storage.
- No marketplace was available to download applications from.
- Java ME was very slow to evolve. The Java Specification Request (JSR) process for new APIs used to take years to complete a specification.

Therefore, Google, in 2005 bought a company called Android, to address all these issues and make available a powerful, unified, and open platform that developers could use to provide services and applications to the general public without much or no carrier control. Since then, Google has introduced many enhancements to the Android platform in the form of new versions, going from version 1.0 in 2008 to version 4.2, called Jelly Bean, in 2012.

Android is a very rich and powerful platform. It runs over a Linux kernel, so it inherits all the advantages of the Linux operating system in terms of security and robustness. Android applications are developed using the Java programming language but they are run on a different virtual machine called Dalvik, which is not a Java virtual machine. This means that Java ME applications do not run on the Android platform without modification. In addition, the Android platform includes the following:

- APIs to simplify the development process. Android includes APIs to use the GPS, camera, audio, network connections, Wi-Fi and Bluetooth networks, accelerometers and other sensors, the touch screen, and power management, among the most important ones.
- Native support for Google Maps, geocoding, and location-based services.
- Shared data and inter-application communication. The platform includes Content Providers to manage access to private databases and has support to send notifications and pass messages within and between applications.
- Support to implement and use a SQLite database in the smartphone for data storage and retrieval.
- Extensive media support, such as 2D and 3D graphics, libraries to handle still images, video, and audio files in different formats (e.g., MPEG4, H.264, MP3, JPG, PNG, GIF, others).
- Optimized memory and process management.

8.1 Android platform

The Android platform consists of the Linux kernel, common libraries, the Android runtime with core libraries and the Dalvik virtual machine, a number of APIs, and general purpose applications, as shown in Figure 8.1.

The first layer at the bottom of the framework consists of the Linux kernel and its core services including all the drivers to interact with the hardware devices in the smartphone, process and memory management, security, network, power management, and so on. The Linux kernel provides an abstraction between the hardware and the remainder of the stack. On top of the kernel there are a series of C++ libraries that provide support to play audio and video media, handle 2D and 3D graphics, support SQLite native databases, and SSL and WebKit for integrated web browsing and Internet security. The Android run time, which consists of the Dalvik virtual machine and its core libraries, is also at that level. One of the most important differences between

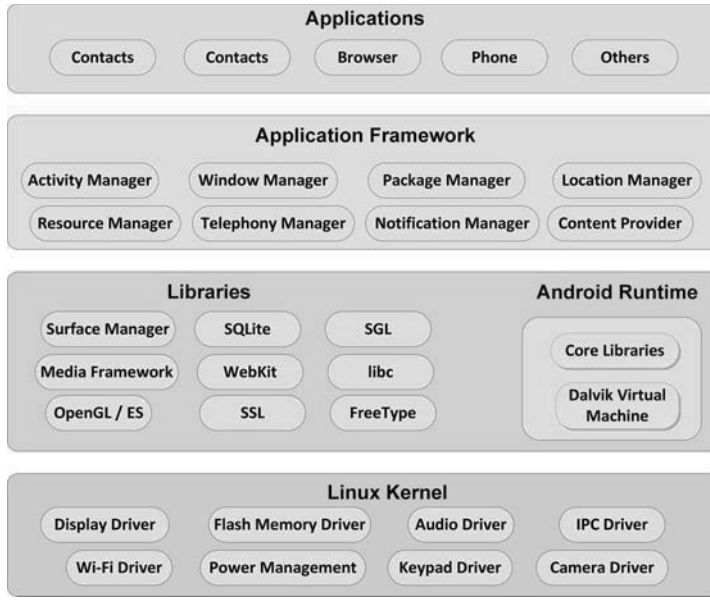

FIGURE 8.1: Android platform.

the Android platform and the Java ME platform is that the Dalvik virtual machine has been optimized to run multiple instances efficiently. This means that each application runs in a separate process with its own instance of the Dalvik virtual machine. The next layer is the application framework, which consists of all those APIs that help software developers make their applications. For example, the Location Manager is the API that handles all the low level details to work with the GPS device, and creates location-based services. Finally, the application layer contains applications already developed and available to all Android users, such as the application that manages the contacts, the application that helps you make phone calls, Internet browsers, etc.

8.2 Android application components

Android applications consist of loosely coupled components bound by the application manifest file, which describes each component and how they interact. Android consists of the following seven components:

- **Activities:** Applications are made of activities, with each activity implementing a piece of the functionality of the application. Activities use

Views to form graphical user interfaces. Each activity has an associated layout file, which is shown to the user when the activity is run.

- **Services:** Services are used to perform regular processing tasks that need to continue even when the application's activities are not active or visible. Services run in the background updating the data sources and visible activities and triggering notifications.
- **Content Providers:** Content providers are sharable data stores used to manage and share application databases.
- **Intents:** Using intents, broadcast messages can be sent system-wide or to a target activity or service. Intents are used to change from one activity to another as a result of a user command from the GUI.
- **Broadcast Receivers:** Broadcast receivers are intent broadcast consumers. When a broadcast receiver is created and registered, the application can listen to broadcast intents. Broadcast receivers will automatically start the application to respond to an incoming intent.
- **Widgets:** Widgets are visual components that can be added to the GUI.
- **Notifications:** Notifications are meant to get the user's attention without interrupting the current activity. For example, it is commonly used to alert the user to an incoming phone call from a service or broadcast receiver.

As can be inferred, applications are made of a series of activities that interact with the user through the graphical user interface. Therefore, at any given point in time, only one activity is active and visible to the user. The Android platform provides several methods to change the state of the activities, so that activities can be run, stopped, paused, resumed, and destroyed as needed by the application. Figure 8.2 shows the life cycle of an Android activity. It is worth noticing that the Android operating system may act by itself, terminating an activity if by the time a new activity needs to be launched there are not enough resources in the system to accommodate it. Android uses a priority system to choose which activity to terminate in that situation.

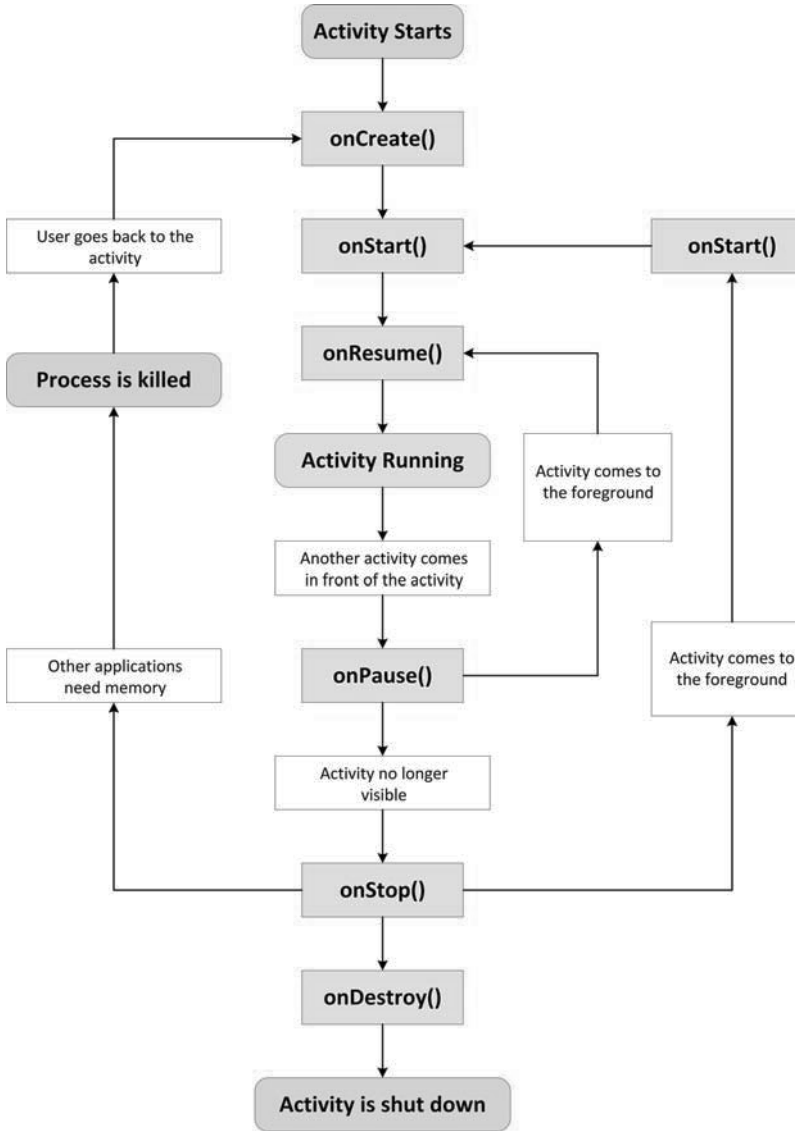

FIGURE 8.2: Life cycle of an Android activity.

Chapter 9

Getting Ready to Develop Android Applications

This chapter is a quick start guide to developing Android applications. The first section presents the procedure to install and set up the software development environment in your computer. Then, a typical Hello World application is developed and described including an explanation of the most important folders and files in the Eclipse Hello World project. The final section concludes the chapter describing how to run applications in software using the Eclipse emulator and in hardware using a smartphone.

9.1 Installing the software development environment

In order to get a computer ready to develop Android applications, the following software must be downloaded and installed in sequential order:

1. The Java SE Development Kit (JDK).
2. The Android Software Development Kit (SDK).
3. The Eclipse Integrated Development Environment (IDE).
4. The Android Development Tools (ADT) Plug-in for Eclipse.

9.1.1 Java SE development kit (JDK)

Android applications are written in Java; therefore, a Java development environment must be installed in your computer. The Java Development Kit can be downloaded from `http://java.oracle.com`. Once in the main page, click on downloads, and click on Java SE under the Java category and, in the next screen, click on the Java SE JDK. This will take you directly to the Java SE development kit download page where the appropriate version according to the operating system of your computer can be downloaded. For example, the name of the JDK file for Windows is `jdk-7u11-windows-i586.exe`. Download the file and run it, double clicking on it from the folder in the Windows

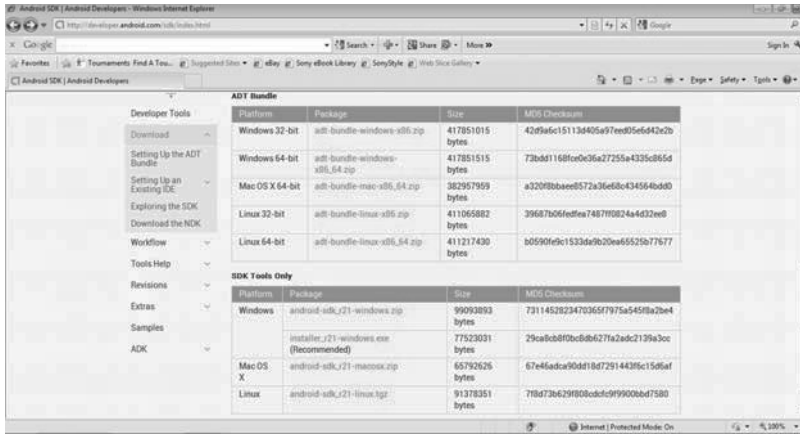

FIGURE 9.1: SDK Tools Only table.

Explorer in which the file was saved. After that, just follow the instructions given by the installation wizard. Choosing the default options, the kit will be successfully installed without doing anything else.

9.1.2 Android SDK

Once the JDK is installed, the next step is to download and install the Android SDK. The Android SDK can be downloaded from `http://developer.android.com/sdk/index.html`. In that page there are two options, either installing the ADT bundle, which contains everything that is needed to begin developing applications, or taking a more customized approach. Here, the second option is chosen and explained, so click on download for other platforms and go to the "SDK Tools Only" table (shown in Figure 9.1). For Windows users, it is recommended to download the installer file (`installer_r21-windows.exe`). Once downloaded, double click on the file and follow the instructions given by the installation wizard. If the Java SDK was successfully installed, the wizard will find it and let you know. Choose all the default options and the wizard will extract and install the appropriate files in your system (see Figure 9.2). Once the installation is complete, click Next and Finish and the installation will launch the Android SDK Manager, as shown in Figure 9.3.

The next step is to configure the Android SDK Manager. The SDK Manager allows the installation of any of the available Android versions as well as special tools, extras, and documentation. Each Android version is identified by an API number (e.g., Android 4.2 API 17). In addition, each version has an associated Google API version (e.g., Android 4.2 API 17, Google API 17), as shown in Figure 9.3, which contains additional APIs provided by Google, such as the Google Maps API. So, if the application includes the use of Google Maps, the Google API version must be chosen.

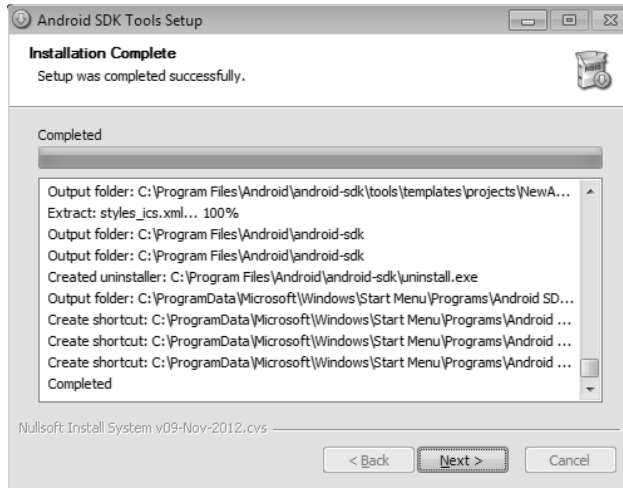

FIGURE 9.2: Extracting and installing the SDK files.

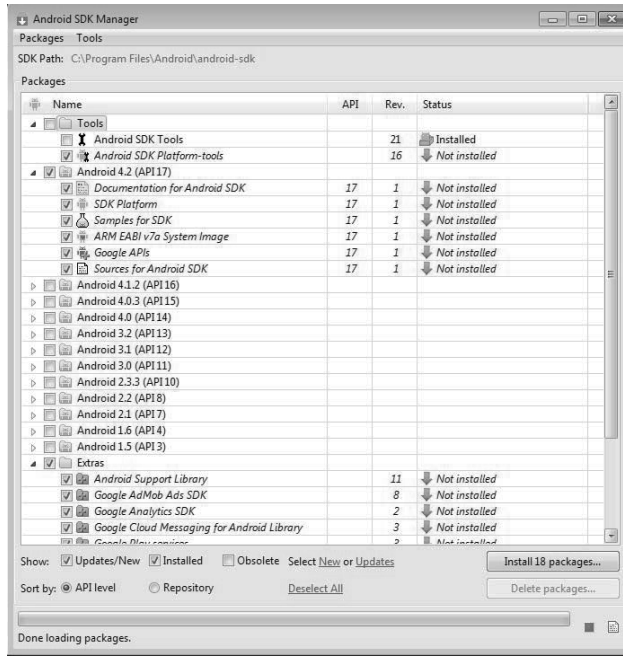

FIGURE 9.3: Android SDK manager.

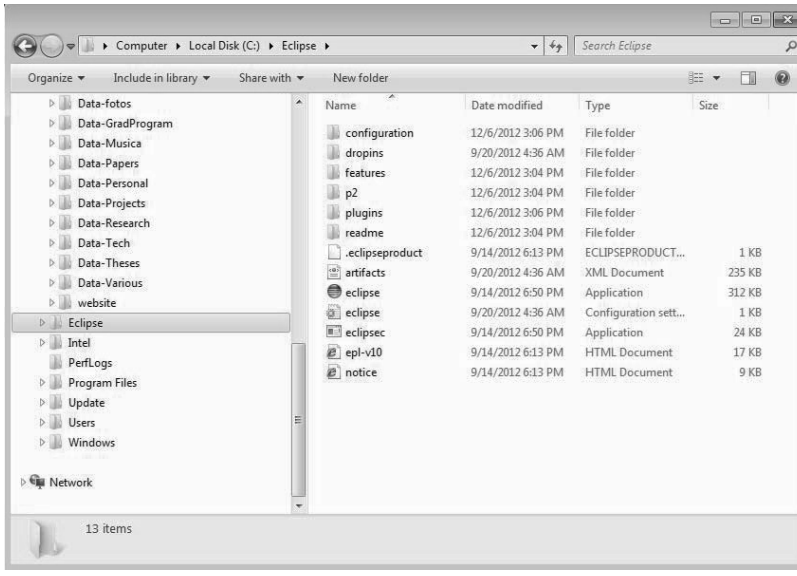

FIGURE 9.4: Files in the Eclipse folder.

After making the selections, click on the Install button and the SDK Manager will install the chosen options in your computer.

9.1.3 Eclipse IDE

The next step is to install the integrated development environment (IDE), i.e., where the applications will be developed. Eclipse, which is the recommended IDE for Android development, can be downloaded from `http://www.eclipse.org/downloads`. Once in that page, download the appropriate Eclipse for your computer. For example, the Eclipse IDE for Java EE Developers for a Windows 32 bit operating system is (`eclipse-jee-juno-SR1-win32.zip`). At this time, this version of Eclipse is named Eclipse Juno.

Make a folder in the place of your preference (e.g., `c:\Eclipse`) and download and save the zip file in it. After unzipping the file, the folder must contain the files shown in Figure 9.4. Launch Eclipse by double clicking on the `eclipse.exe` file. Eclipse will ask for the workspace path. Use the default path and click OK. After that, Eclipse's main screen will appear, as shown in Figure 9.5.

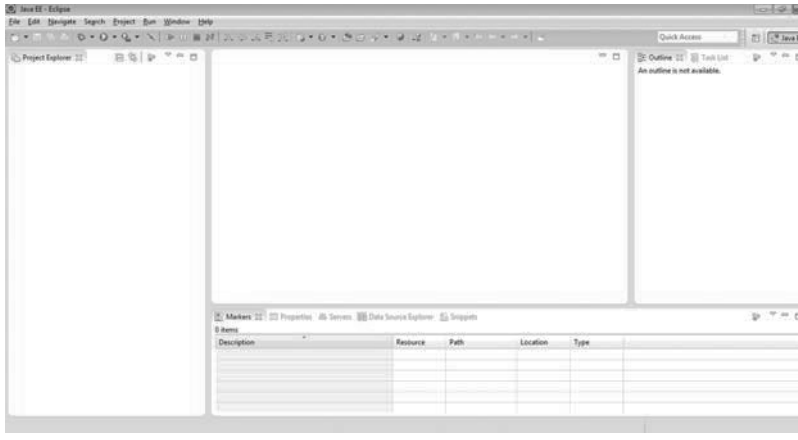

FIGURE 9.5: Eclipse's main screen.

9.1.4 Android ADT plug-in for Eclipse

The last step in the installation process is the installation of the Android Development Tools (ADT) plug-in for Eclipse. The ADT plug-in for Eclipse is very important as it facilitates many important tasks in the software development process of the Android applications. For example, the plug-in performs the following tasks from Eclipse:

- Creates new Android projects.
- Compiles, debugs, and runs Android applications.
- Creates Android package files (`.apk`), which are the files needed to run the applications.
- Signs the `.apk` file.

The ADT plug-in for Eclipse can be installed from the Eclipse IDE. Open Eclipse and once in the main screen, go to the Help tab and choose Install New Software. The install window shown in Figure 9.6 will appear. Click the Add button and include a name (e.g., Google ADT) and the `https://dl-ssl.google.com/android/eclipse` URL. Click Enter and the same install window populated with the tools and plug-ins available will be shown. Selecting the Developer Tools box will install the Android DDMS, the Android Developmetn Tools, the Android Hierarchy Viewer, the Android Trace view, and the Tracer for OpenGL ES tools. Click the Next button to install them all. At this point, you will be asked to restart Eclipse. Once restarted, the final check is to make sure Eclipse has the right path to the Android's SDK. For this, from Eclipse, go to Windows, Preferences and make sure that the path shown in the SDK Location part of the preferences window

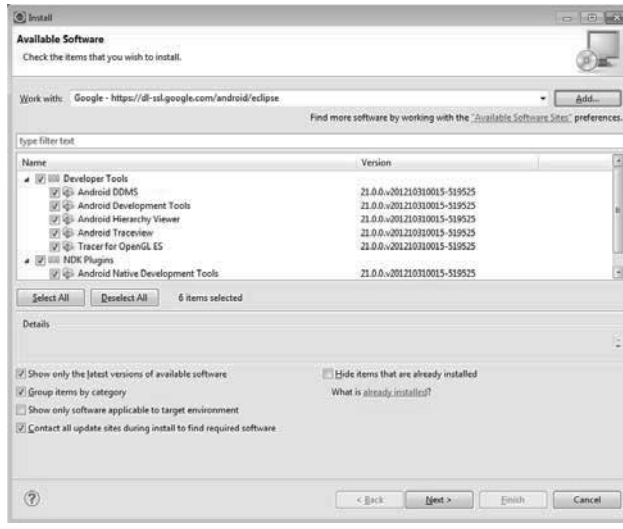

FIGURE 9.6: Installing the ADT plug-in for Eclipse.

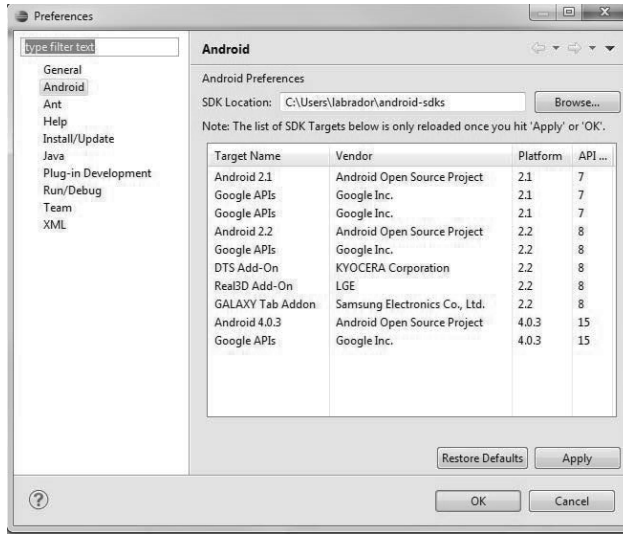

FIGURE 9.7: Verifying the location of the Android SDK in Eclipse.

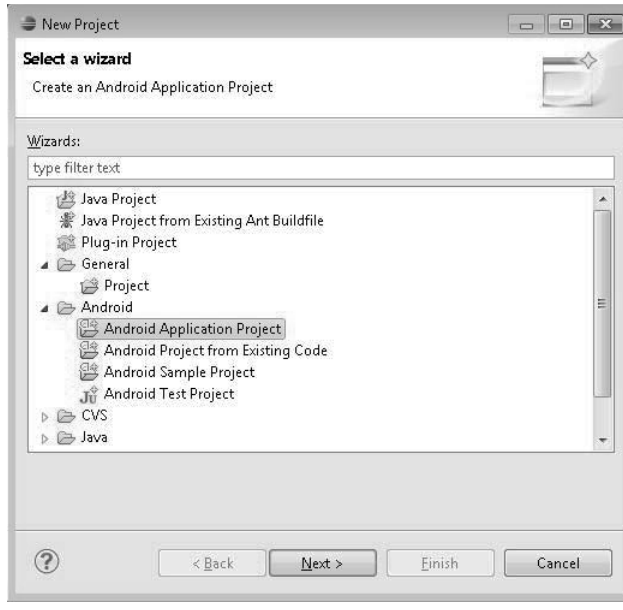

FIGURE 9.8: Creating a new Android project.

(shown in Figure 9.7) corresponds to the location in your machine where the Android SDK was installed. After this, the development environment will be ready to start developing Android applications.

9.2 A Hello World application

This section shows how to create a Hello World application. The first step is to create a new Android project. For that, open Eclipse and select File, New, Project. Expand the Android folder and select Android Project, as shown in Figure 9.8. Click Next and the New Android Application window shown in Figure 9.9 will appear. Type in the name of the project and the package name. The package name identifies the domain of the company that developed the application. Select the appropriate Android platforms and click Next.

The icon that will identify the application in the smartphone can be selected in the next screen. The first choice is to utilize the default icon. This is convenient if a custom made icon has not been designed yet. Otherwise, use the Choose button to select it. Click Next and the Create Activity window will appear. Select BlankActivity and click Next. The next screen asks for the name of the main activity of the application. This activity is the

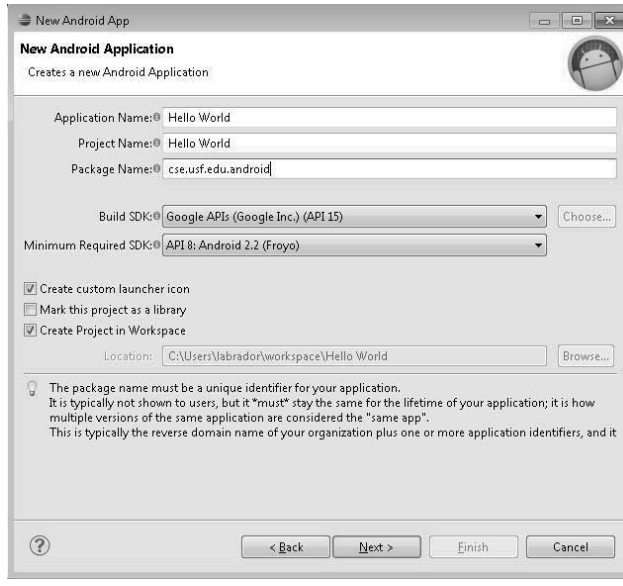

FIGURE 9.9: Creating the Hello World application.

one that will be run whenever the application is run. Type the name (e.g., `HelloWorldActivity`) and click Next. At this point, Eclipse will ask for the installation of the missing platform versions to support the application if they are not available in your system. Please do so if needed and click Finish. After this, Eclipse will create many folders and files automatically and it will show the main screen of the Hello World application. Figure 9.10 shows both, the folders and files in the Package Explorer section and the Hello World screen.

9.3 Skeleton of an Android application

This section explains the main folders and files that the Android ADT automatically created for the Hello World application. The main folders and files are the following:

- The `src` folder: This folder contains the source code files. For example, it contains the `HelloWorldActivity.java` file. Double clicking on that file, will show the Java code of the Hello World application, as shown in Figure 9.11. The most important part of this code is the `onCreate()` method. Once the application is launched, this is the first method that is run. As it can be seen, all this method does for the Hello World

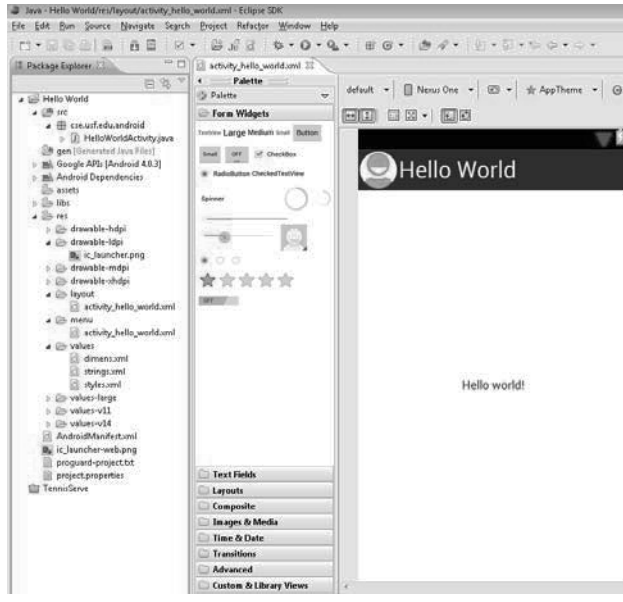

FIGURE 9.10: Hello World application.

application is to show the `activity_hello_world` layout, which is a file included in the layout folder. More on this later.

- The `res` folder: This folder contains several subfolders, as follows:
  - The `drawable` folders: These folders contain pictures and figures used by the application in low, medium, and high resolution. For example, the icon of the application must be included in one of these folders. In this case, the default icon name is `ic_launcher.png`.
  - The `layout` folder: This folder contains the layouts of the application. Each application activity has its corresponding graphical user interface, which is given by the GUI layout files. Notice that many files in Android are `.xml` files and the layouts are one of them. The GUI of each activity can be designed in two manners. One way is dragging the widgets available in the Graphical Layout screen (shown in Figure 9.10). The other way is to write the code directly using `xml-type` tags and commands, as shown in Listing 9.1.
  - The `values` folder: This folder contains `.xml` files with the names of many variables and constants used by the application. This is quite convenient because if a variable name needs to be changed later, only one change in one place is needed. Listing 9.2 shows the names of the variables used in the Hello World application.

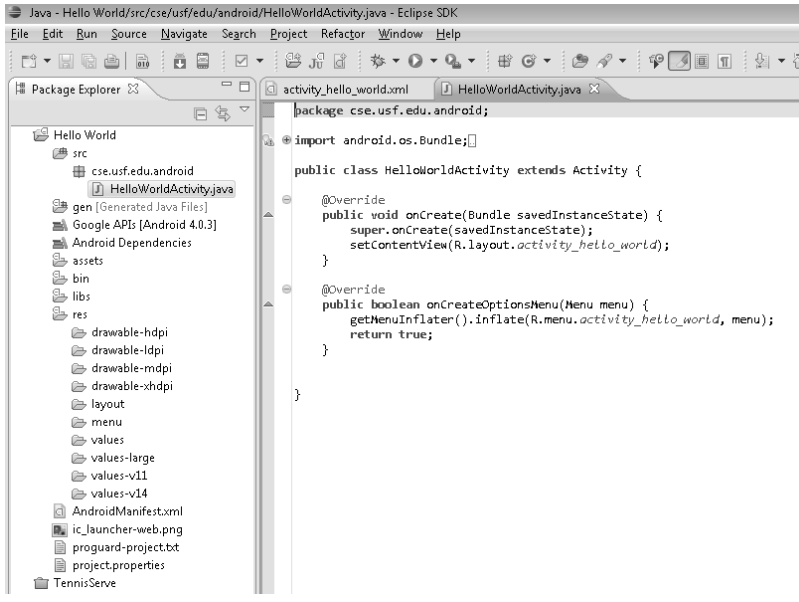

FIGURE 9.11: Hello World Activity Java code.

- The `AndroidManifest.xml` file. The manifest file is one of the most important files in any Android project. The manifest glues together all the pieces of the application indicating the version of the application, the API level, the application name, icon, and style, the activities and services, and also the permissions that the application needs to have to use some other files, applications, and services available in the smartphone, such as the database with your contacts, the email application, the Wi-Fi interface, and so on. Listing 9.3 shows the manifest file of the Hello World application.

```
1  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
       android"
2      xmlns:tools="http://schemas.android.com/tools"
3      android:layout_width="match_parent"
4      android:layout_height="match_parent" >
5
6      <TextView
7          android:layout_width="wrap_content"
8          android:layout_height="wrap_content"
9          android:layout_centerHorizontal="true"
10         android:layout_centerVertical="true"
11         android:padding="@dimen/padding_medium"
12         android:text="@string/hello_world"
13         tools:context=".HelloWorldActivity" />
14
15 </RelativeLayout>
```

Listing 9.1: Hello World layout (`activity_hello_world.xml`) file.

```
<resources>

    <string name="app_name">Hello World</string>
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>
    <string name="title_activity">HelloWorldActivity</string>

</resources>
```

Listing 9.2: Hello World `strings.xml` file.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="cse.usf.edu.android"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="15"
        android:targetSdkVersion="15" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".HelloWorldActivity"
            android:label="@string/title_activity_hello_world" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.
                    LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Listing 9.3: Hello World manifest file (`AndroidManifest.xml`).

9.4 Running Android applications

At this point, the Hello World application is ready; however, it cannot be run yet. Before running the application, Eclipse needs to know where to run it. There are two options, the Eclipse emulator or an actual smartphone. Let us describe these two options.

9.4.1 Running applications using the Android emulator

Eclipse allows for the definition and creation of virtual smartphones so that, if a smartphone is not available, the application can be run and tested in software. If this is the case, before running the application, an Android virtual device must be created using the Android Virtual Device (AVD) Manager. The

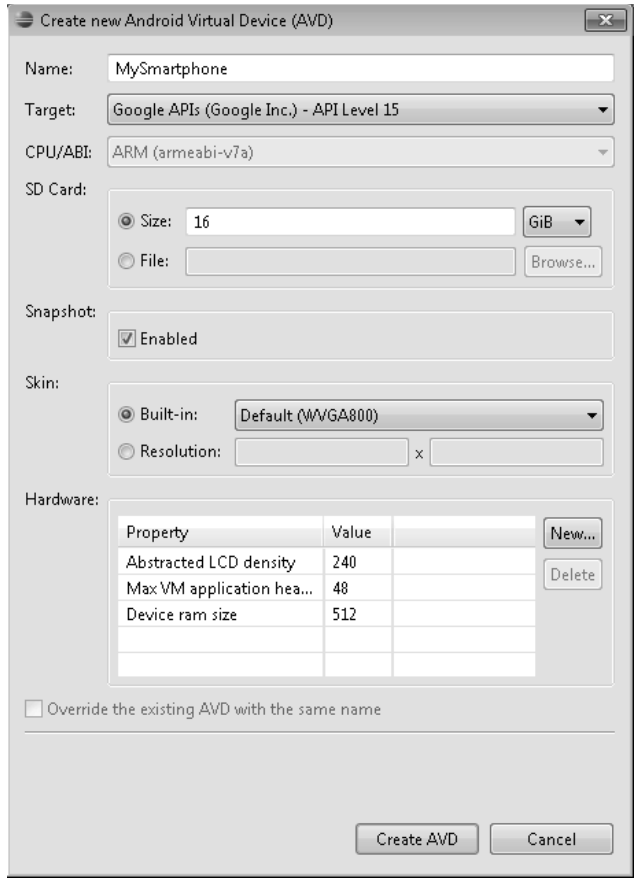

FIGURE 9.12: Android AVD manager.

AVD Manager can be launched by selecting it from the Windows tab. Once launched, click on the New button to create a new device and fill out the fields in the Create new Android Virtual Device (AVD) window, as shown in Figure 9.12. Click on Create AVD. After that, the virtual smartphone will be ready to run the Hello World application. Just make sure that the platform number (API level) by the application is the one included in the virtual device. Once the virtual device is available, the application can be run; just click the run button and Eclipse will launch the emulator and run the application in the virtual device, as shown in Figure 9.13.

9.4.2 Running applications using an Android smartphone

The second option, the most preferred, is to run the application directly in a smartphone. For this, in addition to having an Android smartphone, several

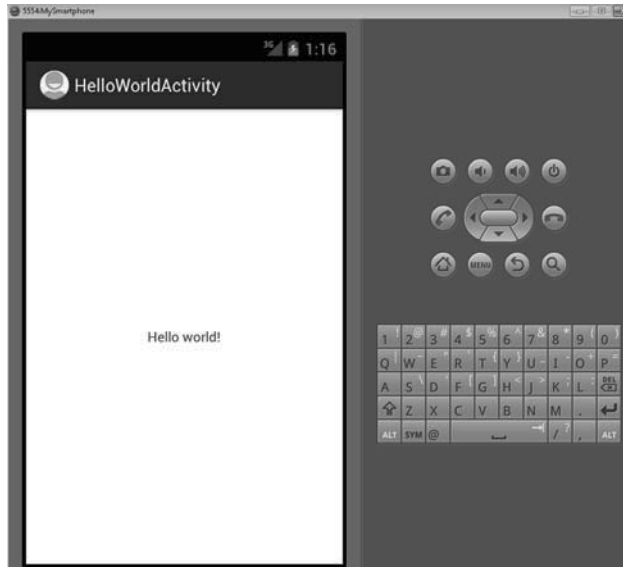

FIGURE 9.13: Android emulator running the Hello World application.

modifications need to be made in the Eclipse project, the smartphone, and the computer, as follows:

- Open the `AndroidManifest.xml` file and inside of the opening `<application>` tag insert `android:debuggable=‘‘true’’`.
- Turn on the smartphone, navigate to the home screen, press Menu, select Settings → Applications → Development, and enable the USB debugging option.
- Install the drivers of the smartphone in the computer. These drivers are usually found in the websites of the smartphone manufacturers. Then, download and install them so that when the smartphone is connected using the USB cable, the computer recognizes it.

After this, go to Run →Run Configurations and select the Always prompt to pick device option, as shown in Figure 9.14. With this option, every time the application is run, Eclipse will prompt the user on whether it should be executed in the emulator or in the smartphone. If the drivers where correctly installed, the smartphone should be in the list of available devices. So, select it and click the Run button.

At this point, the Eclipse ADT plug-in will sign the application with its own debug certificate, compile the application, generate the `.apk` file, and install it and run it in the smartphone. This is good enough for developing and testing the application. However, if the application is ready

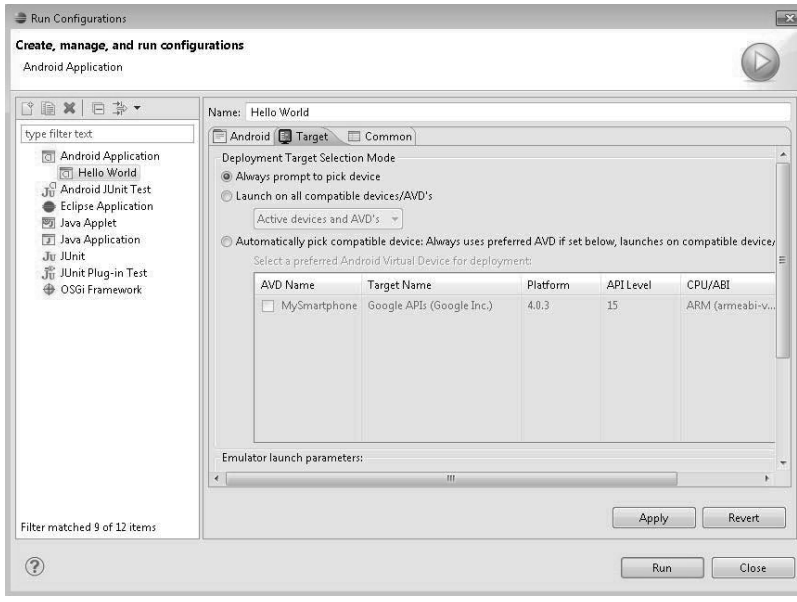

FIGURE 9.14: Run configurations window.

for the use of others, it must be signed with a different certificate than the debug certificate. The Eclipse ADT plug-in provides a simple export wizard that automates the entire process. Additional information can be found in the `http://developer.android.com/guide/publishing/app-signing.html#overview` and `http://developer.android.com/guide/publishing/app-signing.html#ExportWizard` websites.

Chapter 10

Using the Smartphone's Sensors

The Android Software Development Kit (SDK) provides software support in the form of Application Programming Interfaces (APIs) to interact with the sensors available in the smartphone. Nowadays, smartphones come equipped with a wide variety of sensors, which are usually classified into four big groups:

- **Motion sensors**, such as accelerometers and gyroscopes, which measure acceleration and rotational forces along three axes.
- **Environmental sensors**, such as barometers, photometers, and thermometers, which measure environmental variables like ambient light, pressure, humidity, and temperature.
- **Position sensors**, such as magnetometer, GPS, and proximity orientation sensors, which measure the physical orientation and position of the device.
- **Contextual sensors**, such as the camera and microphone, as well as Bluetooth and Wi-Fi interfaces, which are used to obtain information about the surrounding environment.

Table 10.1 shows the most common type of sensors supported by the Android APIs along with a brief description of the variables being measured and the most common use of such a variable. It is worth noticing that the APIs and sensors available depend on the Android version and the specific smartphone model.

The sensor API is part of the `android.hardware` package, which must be imported into the application. The package contains the following interfaces:

- `SensorManager`: Used to obtain a reference to the sensor service. It allows to list, register, and unregister sensor events.
- `Sensor`: Encapsulates the methods that determine the characteristics of a sensor.
- `SensorEventListener`: Allows a component to subscribe to a sensor and obtain its raw data. The subscriber needs to implement two callback methods to receive notifications from the sensor.

TABLE 10.1: Common sensors supported by the Android sensors API.

Sensor	**Description**	**Use**
Accelerometer	Acceleration force being applied to the smartphone on the x, y, and z physical axes	Directional movement of the device
Gyroscope	Rate of rotation of the device in rad/s on the three physical axes	Rotation detection
Gravity	Force of gravity being applied on all three physical axes of the device	Motion detection
Magnetic field	Geomagnetic field for all three physical axes	Compass
Orientation	Degrees of rotation that the smartphone makes around all three physical axes	Device position
Proximity	Proximity of an object to the screen of the device	Smartphone position during a call
Light	Ambient light level	Control brightness of the screen
Pressure	Ambient pressure	Monitoring pressure changes
Humidity	Relative humidity	Monitoring ambient humidity
Ambient temperature	Ambient temperature	Monitoring ambient temperature

- `SensorEvent`: Once a component subscribes to a sensor, the methods of the `SensorEventListener` return a `SensorEvent` object, which not only provides the sensor raw data but also the accuracy of the measurement, the time stamp, and the type of sensor that generated the event.

In order to utilize a sensor, the following three-step process must be followed:

1. Obtain a reference of the `SensorManager` to:
   - Verify if the sensor that you want to subscribe to is available using `SensorManager.List<Sensor> gwetSensorList(int type)`.
   - If the sensor is available, subscribe to it using `boolean registerListener(SensorEventListener listener, Sensor sensor, int rate)`.
2. Implement the `SensorEventListener` and write the code for the two methods of the interface:
   - `void onSensorChanged(SensorEvent event)`: This method is fired every time a sensor reads something new.
   - `void onAccuracyChanged(Sensor sensor, int accuracy)`: This method is fired whenever the accuracy of a sensor changes.
3. Unsubscribe (if needed) from the sensor using `SensorManager.unregisterListener (SensorEventListener lstr)`.

10.1 An example application

This section presents an application that acquires the raw data from different sensors and shows the readings in real-time on the screen. The sensors utilized in the application are the accelerometers, gyroscope, and the magnetic field. The application also measures the received signal strength from the communication carrier and the Wi-Fi networks available around the smartphone. Figure 10.1 shows a screen shot of the application while being connected to a private Wi-Fi network.

The application consists of two buttons, one to start the readings (only one shown in the figure) and another one to stop them that is automatically hidden when pressed, and one activity with its own layout. The manifest file is shown in Listing 10.1, showing that the application was developed using the Android 2.3.3 version, API level 10, it has permission to use the Wi-Fi interface, and launches the `ClientActivity` activity when initiated.

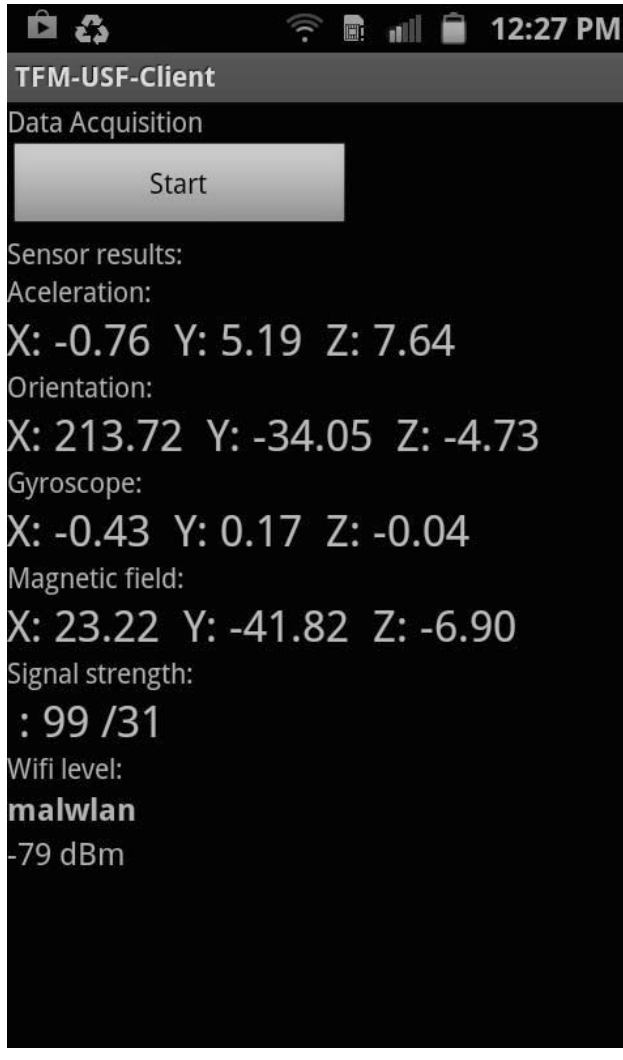

FIGURE 10.1: User interface of the example sensor application.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="android.tfmusf"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="10" />

     <uses-permission android:name="android.permission.
         CHANGE_WIFI_STATE"/>
    <uses-permission android:name="android.permission.
        CHANGE_NETWORK_STATE"/>
  <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"
      />


    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".ClientActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.
                    LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Listing 10.1: The `AndroidManifest.xml` file.

Listing 10.2 shows the `strings.xml` file, which contains the names of the variables used by the application. Looking at Figure 10.1 and Listing 10.2 it can be easily inferred how these names and variables were used.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="main">Data Acquisition</string>
    <string name="app_name">TFM-USF-Client</string>
    <string name="startButton">Start</string>
    <string name="stopButton">Stop</string>
    <string name="results">Sensor results:</string>
    <string name="aceleration">Aceleration:</string>
    <string name="orientation">Orientation:</string>
    <string name="gyroscope">Gyroscope:</string>
    <string name="magneticfield">Magnetic field:</string>
    <string name="signalstrength">Signal strength:</string>
    <string name="wifilevel">Wifi level:</string>

</resources>
```

Listing 10.2: The `strings.xml` file.

The following listing shows the content of the `main.xml` file, which contains the layout of the graphical user interface of the application. As it can be seen, it is a linear layout with lines containing buttons, text views, blank spaces, and a list view to include a list of all available Wi-Fi networks.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/main" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >

        <Button
            android:id="@+id/button1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="0.24"
            android:text="@string/startButton" />

        <Button
            android:id="@+id/button2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="0.18"
            android:text="@string/stopButton" />
    </LinearLayout>

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/results" />

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/aceleration" />

    <TextView
        android:id="@+id/acelerationView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=""
        android:textAppearance="?android:attr/textAppearanceLarge" />

    <TextView
        android:id="@+id/textView3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/orientation" />

    <TextView
        android:id="@+id/orientationView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text=""
        android:textAppearance="?android:attr/textAppearanceLarge" />

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
```

```
66          android:layout_height="wrap_content"
67          android:text="@string/gyroscope" />
68
69      <TextView
70          android:id="@+id/gyroscopeView"
71          android:layout_width="wrap_content"
72          android:layout_height="wrap_content"
73          android:text=""
74          android:textAppearance="?android:attr/textAppearanceLarge" />
75
76      <TextView
77          android:id="@+id/textView4"
78          android:layout_width="wrap_content"
79          android:layout_height="wrap_content"
80          android:text="@string/magneticfield" />
81
82      <TextView
83          android:id="@+id/magneticfieldView"
84          android:layout_width="wrap_content"
85          android:layout_height="wrap_content"
86          android:text=""
87          android:textAppearance="?android:attr/textAppearanceLarge" />
88
89      <TextView
90          android:id="@+id/textView5"
91          android:layout_width="wrap_content"
92          android:layout_height="wrap_content"
93          android:text="@string/signalstrength" />
94
95      <TextView
96          android:id="@+id/signalstrengthView"
97          android:layout_width="wrap_content"
98          android:layout_height="wrap_content"
99          android:text=""
100         android:textAppearance="?android:attr/textAppearanceLarge" />
101
102     <TextView
103         android:id="@+id/textView6"
104         android:layout_width="wrap_content"
105         android:layout_height="wrap_content"
106         android:text="@string/wifilevel" />
107
108     <ListView
109         android:id="@+id/wifiListView"
110         android:layout_width="match_parent"
111         android:layout_height="wrap_content"
112         tools:listitem="@android:layout/simple_list_item_2" >
113     </ListView>
114
115 </LinearLayout>
```

Listing 10.3: The `main.xml` file.

Finally, the `ClientActivity.java` file, which implements the main activity of the application, is shown in five parts in Listings 10.4, 10.5, 10.6, 10.7, and 10.8. The first part, shown in Listing 10.4, contains the name of the package, the libraries that need to be imported, and the names of the variables used in the activity. As usual, the `ClientActivity` class extends the `Activity` class and implements the `SensorEventListener` interface that contains the methods needed to use the sensors.

```
1  package android.tfmusf;
2
3  import java.util.List;
4  import android.app.Activity;
5  import android.content.BroadcastReceiver;
6  import android.content.Context;
7  import android.content.Intent;
8  import android.content.IntentFilter;
9  import android.hardware.Sensor;
10 import android.hardware.SensorEvent;
11 import android.hardware.SensorEventListener;
12 import android.hardware.SensorManager;
13 import android.net.wifi.ScanResult;
14 import android.net.wifi.WifiManager;
15 import android.os.Bundle;
16 import android.telephony.PhoneStateListener;
17 import android.telephony.TelephonyManager;
18 import android.view.View;
19 import android.widget.Button;
20 import android.widget.ListView;
21 import android.widget.TextView;
22
23 public class ClientActivity extends Activity  implements
       SensorEventListener  {
24   /** Called when the activity is first created. */
25
26   //Buttons
27   private Button startButton;
28   private Button stopButton;
29
30   //Views
31   private TextView acelerationView;
32   private TextView orientationView;
33   private TextView gyroscopeView;
34   private TextView magneticfieldView;
35   private TextView signalstrengthView;
36   private ListView wifiListView;
37
38   private Boolean started;
39
40   public static final int INVISIBLE = 4;
41
42   //Event Managers
43   //Sensors
44   private SensorManager sensorManager;
45   //Telephone
46   private TelephonyManager  telephonyManager;
47   private SignalStrengthListener signalStrengthListener;
48   //Wifi
49   private WifiManager mainWifi;
50   private WifiReceiver receiverWifi;
51   private List<ScanResult> wifiList;
```

Listing 10.4: The first part of the `ClientActivity.java` file.

Listing 10.5 presents the second part of the activity and contains the `onCreate()` method, which launches the main layout, initializes the variables of the views, and registers the different sensor managers to be used, such as the sensor manager for the different sensors, the Wi-Fi manager to scan and obtain the available Wi-Fi networks available, and the telephony manager to acquire the signal strength of the cellular carrier.

```
@Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        started = false;

        //Find Views
        acelerationView = (TextView) findViewById(R.id.acelerationView
            );
        orientationView = (TextView) findViewById(R.id.orientationView
            );
        gyroscopeView = (TextView) findViewById(R.id.gyroscopeView);
        magneticfieldView = (TextView) findViewById(R.id.
            magneticfieldView);
        signalstrengthView = (TextView) findViewById(R.id.
            signalstrengthView);
        wifiListView = (ListView) findViewById(R.id.wifiListView);

        // Sensor manager
        sensorManager = (SensorManager) getSystemService(
            SENSOR_SERVICE);

        // Telephony Manager
        telephonyManager = (TelephonyManager) getSystemService(
            TELEPHONY_SERVICE);

        // Signal Listener
       signalStrengthListener = new SignalStrengthListener();
       telephonyManager.listen(signalStrengthListener,
           SignalStrengthListener.LISTEN_SIGNAL_STRENGTHS);

       // Wi-Fi manager
       mainWifi = (WifiManager) getSystemService(Context.WIFI_SERVICE)
           ;
       receiverWifi = new WifiReceiver();
       registerReceiver(receiverWifi, new IntentFilter(WifiManager.
           SCAN_RESULTS_AVAILABLE_ACTION));

       addListenerOnButton();
    }
```

Listing 10.5: The second part of the `ClientActivity.java` file.

Listing 10.6 shows the third part of the activity, which includes the code to obtain a reference of the start and stop buttons and set up the click listener for each one, and the `onResume()` and `onPause()` methods to register and unregister the different sensors using their managers.

```
public void addListenerOnButton(){

      startButton = (Button) findViewById(R.id.button1);
      stopButton = (Button) findViewById(R.id.button2);

      startButton.setOnClickListener(new View.OnClickListener() {

      public void onClick(View v) {
        startButton.setVisibility(INVISIBLE);
        stopButton.setVisibility(0);
        started = true;
        mainWifi.startScan();
      }
    });

```

```
16        stopButton.setOnClickListener(new View.OnClickListener() {
17
18        public void onClick(View v) {
19          stopButton.setVisibility(INVISIBLE);
20          startButton.setVisibility(0);
21          started = false;
22        }
23      });
24    }
25
26    /** Register for the updates when Activity is in foreground */
27    @Override
28    protected void onResume() {
29      super.onResume();
30      //Register listeners
31      sensorManager.registerListener(this, sensorManager.
            getDefaultSensor(Sensor.TYPE_ACCELEROMETER), SensorManager.
            SENSOR_DELAY_GAME);
32      sensorManager.registerListener(this, sensorManager.
            getDefaultSensor(Sensor.TYPE_ORIENTATION), SensorManager.
            SENSOR_DELAY_GAME);
33      sensorManager.registerListener(this, sensorManager.
            getDefaultSensor(Sensor.TYPE_GYROSCOPE), SensorManager.
            SENSOR_DELAY_GAME);
34      sensorManager.registerListener(this, sensorManager.
            getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD), SensorManager.
            SENSOR_DELAY_GAME);
35      registerReceiver(receiverWifi, new IntentFilter(WifiManager.
            SCAN_RESULTS_AVAILABLE_ACTION));
36
37    }
38
39    /** Stop the updates when Activity is paused */
40    @Override
41    protected void onPause() {
42      super.onPause();
43      //Unregister Listener
44      sensorManager.unregisterListener(this, sensorManager.
            getDefaultSensor(Sensor.TYPE_ACCELEROMETER));
45      sensorManager.unregisterListener(this, sensorManager.
            getDefaultSensor(Sensor.TYPE_ORIENTATION));
46      sensorManager.unregisterListener(this, sensorManager.
            getDefaultSensor(Sensor.TYPE_GYROSCOPE));
47      sensorManager.unregisterListener(this, sensorManager.
            getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD));
48      unregisterReceiver(receiverWifi);
49    }
```

Listing 10.6: The third part of the `ClientActivity.java` file.

The fourth part of the class, shown in Listing 10.7, consists of the code that executes every time the readings of the sensors change (the `onSensorChanged()` method) after the start button is pressed (by means of the `started` flag). As it can be seen, each case of the switch statement handles each of the sensors being used. Appropriate variables are included to get the readings from the sensors and formating statements are executed to send the readings to the GUI (`setText()`) methods.

```
1  public void onSensorChanged(SensorEvent event) {
2        synchronized (this) {
3          if (started) {
4              switch (event.sensor.getType()){
5                case Sensor.TYPE_ACCELEROMETER:
6                    /*Get the acceleration force in m/s2 that is applied
                         to a device
7                    on all three physical axes (x, y, and z), including
                         the force of gravity.*/
8                    float xA = event.values[0];
9                    float yA = event.values[1];
10                   float zA = event.values[2];
11
12                   String acelerationOut = String.format("X: %.2f  Y:
                         %.2f  Z: %.2f", xA, yA, zA);
13                   acelerationView.setText(acelerationOut);
14
15               break;
16
17               case Sensor.TYPE_ORIENTATION:
18                   // Measures degrees of rotation that a device makes
                         around all three physical axes (x, y, z)
19                   float xO = event.values[0];
20                   float yO = event.values[1];
21                   float zO = event.values[2];
22
23                   String orientationOut = String.format("X: %.2f  Y:
                         %.2f  Z: %.2f", xO, yO, zO);
24                   orientationView.setText(orientationOut);
25
26               break;
27
28               case Sensor.TYPE_GYROSCOPE:
29                   //Measures a device's rate of rotation in rad/s
                         around each of the three physical axes (x, y,
                         and z)
30                   float xG = event.values[0];
31                   float yG = event.values[1];
32                   float zG = event.values[2];
33
34                   String gyroscopeOut = String.format("X: %.2f  Y: %.2
                         f  Z: %.2f", xG, yG, zG);
35                   gyroscopeView.setText(gyroscopeOut);
36
37               break;
38
39               case Sensor.TYPE_MAGNETIC_FIELD:
40                   //Measures the ambient geomagnetic field for all
                         three physical axes (x, y, z) in mircoT.
41                   float xM = event.values[0];
42                   float yM = event.values[1];
43                   float zM = event.values[2];
44
45                   String magneticfiedlOut = String.format("X: %.2f  Y:
                         %.2f  Z: %.2f", xM, yM, zM);
46                   magneticfieldView.setText(magneticfiedlOut);
47
48               break;
49             }
50         }
51       }
52     }
```

Listing 10.7: The fourth part of the `ClientActivity.java` file.

The fifth and final part of the activity, shown in listing 10.8, contains

the code that obtains the name and signal strength of the communication carrier providing the service to the smartphone and the name and signal strength of the Wi-Fi networks available around the smartphone. The first part is accomplished by the `signalStrength.getGsmSignalStrength()` method within the `onSignalStrengthsChanged()` method. The second part begins with the `getScanResults()` method that scans the Wi-Fi spectrum and gets the list of Wi-Fi networks (SSIDs) and their respective signal strengths.

```
1  private class SignalStrengthListener extends PhoneStateListener {
2
3        // Signal Carrier
4        String carrierName = telephonyManager.getNetworkOperatorName();
5
6      @Override
7       public void onSignalStrengthsChanged(android.telephony.
            SignalStrength signalStrength) {
8
9         super.onSignalStrengthsChanged(signalStrength);
10        if (started) {
11         // Get the signal strength (a value between 0 and 31)
12           int strengthAmplitude = signalStrength.getGsmSignalStrength
                ();
13           // Update a text view
14           String strengthsignalOut = carrierName + String.format(" :
                %02d /31",strengthAmplitude );
15           signalstrengthView.setText(strengthsignalOut);
16           mainWifi.startScan();
17        }
18      }
19   }
20
21   class WifiReceiver extends BroadcastReceiver {
22
23         public void onReceive(Context c, Intent intent) {
24             //Get the WiFi ssan results
25             wifiList = mainWifi.getScanResults();
26             Wifi[] wifis = new Wifi[wifiList.size()];
27             for(int i = 0; i < wifiList.size(); i++){
28               wifis[i] = new Wifi(wifiList.get(i).SSID, wifiList.get(i
                     ).level +" dBm");
29             }
30             wifiListView.setAdapter(new WifiArrayAdapter(c, wifis));
31         }
32     }
33
34   public class Wifi {
35       public String ssid;
36       public String level;
37
38       public Wifi(String ssid, String level){
39         this.ssid = ssid;
40         this.level = level;
41       }
42   }
43
44   public class WifiArrayAdapter extends TwoLineArrayAdapter<Wifi> {
45       public WifiArrayAdapter(Context context, Wifi[] wifis) {
46           super(context, wifis);
47       }
48
49       @Override
50       public String lineOneText(Wifi w) {
51           return w.ssid;
```

```
52          }
53
54          @Override
55          public String lineTwoText(Wifi w) {
56              return w.level;
57          }
58      }
59  }
```

Listing 10.8: The fifth part of the `ClientActivity.java` file.

Chapter 11

Bluetooth Connectivity in Android

This chapter provides a brief guide to connecting an Android smartphone with an external Bluetooth-enabled sensor. Bluetooth is a wireless technology to exchange data over short distances interconnecting fixed and mobile devices in Personal Area Networks (PANs). The standard is managed by the Bluetooth Special Interest Group, which consists of more than 17,000 companies in the information technology field. Bluetooth broadcasts in the 2.4GHz Industrial, Scientific, and Medical (ISM) frequency band and is frequently used to connect headsets and printers, perform file/data transfers and, as in this case, connect other external devices, such as sensors.

Bluetooth works based on a star network topology in which one device is the master and the others are slaves and the master switches between slaves in a round-robin fashion. Each device must wait for its time slot to transmit. Depending on the transmission power, Bluetooth devices are classified as:

- Class 1: Maximum transmission power: 100 mW; Range: $\approx$ 100 m.
- Class 2: Maximum transmission power: 2.5 mW; Range: $\approx$ 10 m.
- Class 3: Maximum transmission power: 1.0 mW; Range: $\approx$ 5 m.

In addition to the class of the device, there are currently four versions of the standard:

- Version 1.2: Up to 1 Mbps.
- Version 2.0 + Enhanced Data Rate (EDR): Up to 3 Mbps.
- Version 3.0 + High Speed (HS): Up to 24 Mbps.
- Version 4.0 Low Energy (LE): Up to 500 Kbps, aimed for low energy.

The Bluetooth standard specification is managed through a series of protocols in a stack. The stack consists of three major components: The *Application Layer*, the *Middleware Layer*, and the *Transport Layer*. The application layer is not part of the standard and consists of some applications developed to work using the underlying Bluetooth network. The middleware layer consists of several protocols meant to control the exchange of different types of data in a reliable manner. Finally, the Transport layer corresponds to the Physical

layer of the *Open System Interconnect* (OSI) model. It is worth mentioning that Bluetooth technology, in OSI parlance, is a data link layer technology.

The Bluetooth standard defines several profiles, which are specifications that a device must implement in order to provide certain services. For example, when a Bluetooth headset is to be used with a smartphone, both devices must implement the Hands-Free Profile (HFP). As such, there are different profiles aimed at different applications and services. From a programmatic point of view, one of the most utilized profiles is the Serial Port Profile (SPP), which enables reliable serial communication over Bluetooth (this profile is also called RFCOMM). Android's support is based on this profile. More information about Bluetooth technology can be found in `http://www.bluetooth.com/Pages/Bluetooth-Home.aspx`.

11.1 Exchanging data with an external device via Bluetooth

Connecting to a Bluetooth-enabled device consists of a five-step process, as follows:

1. Obtain an adapter to use Bluetooth.
2. Search for the Bluetooth device.
3. Create a socket.
4. Establish the connection.
5. Read and write operations.

An adapter in Android is usually an object that acts as a bridge. In the case of a Bluetooth adapter, the `BluetoothAdapter` class lets you interact with the Bluetooth device providing methods to discover Bluetooth devices, query paired devices, instantiate a Bluetooth device, and create sockets to listen for connection requests from other devices. Therefore, the starting point to work with Bluetooth devices is to get an instance of a `BluetoothAdapter`, which represents the local Bluetooth adapter. This is achieved by the following line of code:

```
BluetoothAdapter adapter = BluetoothAdapter.getDefaultAdapter();
```

If the MAC address of the remote Bluetooth device is known, an instance of that device can be obtained with the following line of code:

```
BluetoothDevice dev = adapter.getRemoteDevice(MACAddress);
```

The `BluetoothDevice` class represents a remote Bluetooth device and allows for the creation of a connection with the respective device or query information about it, such as the name, address, class, and bonding state.

If the MAC address of the remote device is unknown, the same code can be used after obtaining that MAC address using the available device discovery methods in the `BluetoothAdapter` class.

The next step is to create a socket to communicate with the device. This can be accomplished by the following lines of code:

```
Method m = dev.getClass().getMethod("createRfcommSocket",
new Class[] int.class);
```

```
BluetoothSocket socket = (BluetoothSocket) m.invoke (dev,
Integer.valueOf(1));
```

Once the socket is created, the next step is to establish the connection, as follows:

```
socket.connect();
```

The final step is to initialize the stream objects to read and write data via Bluetooth. This is accomplished by the following lines of code:

```
DataInputStream in = new DataInputStream(socket.getInputStream());
```

```
DataOutputStream out = new DataOutputStream(socket.getOutputStream());
```

Once the stream objects are initialized, the reading and writing functions are used to exchange data over the socket. For example, to read a byte from the device, using the `DataInputStream`, the following line of code is used:

```
byte b = in.readByte();
```

Similarly, to write a message to the device using the `DataOutputStream` object, the following line of code is enough where message is an array of bytes that make up the format of the data link layer frame used by the device:

```
out.write(message);
```

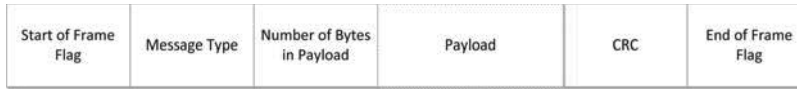

FIGURE 11.1: A typical data link layer frame.

In order to perform the read and write operations appropriately, it is imperative to know the format of the data link layer frame used by the device; otherwise, it will be extremely difficult, if not impossible, to decode the information coming from the device, nor the device will understand what you are sending to it. Although data link layer frames are fairly standard, as shown in Figure 11.1, some manufacturers implement their own protocols and field meanings, so it is imperative to obtain the detailed information about the frame from the device manufacturer. For example, the start-of-frame and end-of-frame characters need to be known in order to determine when the frame begins and ends. Similarly, the size of the payload needs to be known, as they are of variable length, to be able to decode the data correctly. Equally important is to know the types of messages implemented by the protocol and the characters that identify each one. Finally, the developer needs to know how to interpret the bytes coming in the payload. For example, the Bluetooth device may be sending in the payload data from several sensors such as accelerometer data, pulse rate data, temperature data, etc. The developer needs to know where in the payload these bytes are located, so the data can be extracted and used correctly.

Listing 11.1 includes sample code for reading a packet of data assuming a data link layer frame like the one shown in Figure 11.1.

```
1  // Create a byte array to store the read data
2  byte[] rawData = new byte[256];
3
4  // Wait until a STX byte is received
5  while ((t = in.readByte()) != STX && t != 0);
6
7  // If t == 0 is because connection was lost
8  if (t != 0) {
9      rawData[0] = ByteOperations.STX;
10     rawData[1] = in.readByte();
11     rawData[2] = in.readByte();
12
13     // DLC is the size of the actual data in the packet
14     int dlc = Integer.parseInt("" + rawData[2]);
15     i = 3;
16     for (int j = 0; j < dlc; j++) {
17       rawData[j + i] = in.readByte();
18       i++;
19     }
20
21   // Read the last two bytes: CRC and ETX
22   rawData[i++] = in.readByte();
23   rawData[i] = in.readByte(); ...
```

Listing 11.1: Sample code to read a packet from a Bluetooth device.

The result of running the code included in Listing 11.1 is an array of bytes containing the data from the Bluetooth device. If for example, this device is a sensor that measures a person's heart rate and temperature and these readings are included in bytes 12 and 13 and 14 and 15 of the payload, respectively, then the following two lines of code will provide those readings:

```
int heartRate = mergeUnsigned(rawData[12], rawData[13]);
int Temperature = mergeUnsigned(rawData[14], rawData[15]);
```

In the two lines of code above, the `mergeUnsigned()` method assumes that the readings are given in two bytes (low and high order). The function then merges and converts these two bytes into an integer that can be read by humans.

Chapter 12

Saving and Retrieving Data in an Android Smartphone

The Android platform offers several options for saving and retrieving the data used by applications. This chapter shows the reader how to use these options, namely the **Shared Preferences**, **Files**, and **SQLite Databases**.

12.1 Shared preferences

Shared preferences are useful to store and retrieve application settings and/or user profiles. The main idea is to make the application work as the user wants it to and also automate some repetitive tasks to enhance the user experience. For example, if the application requires the user to enter their user name and password to log into the application, the user name can be stored and retrieved from the shared preferences so that when they launch the application, the user name is already shown in the screen.

The `SharedPreferences` class provides a general framework to save and retrieve persistent key-value pairs of primitive data types such as booleans, floats, ints, longs, and strings. Internally, the data are saved into special files within the applications data directory.

The first step to working with shared preferences is to create a file. This is accomplished with the line of code below:

```
SharedPreferences settings = getSharedPreferences(fileName, mode);
```

The method has two parameters, the `fileName`, which is a string identifying the name of the file, and the `mode`, an integer that identifies the file creation mode, which has the following options:

- `MODE_PRIVATE`: Gives access to a single process in the application.
- `MODE_WORLD_READABLE`: Gives access to other applications to read the file.

- MODE_WORLD_WRITABLE: Gives access to other applications to write to the file.
- MODE_MULTI_PROCESS: Gives access to several processes that can be run within the application.

In order to store data in the shared preferences file just created, a SharedPreferences.Editor object is required. This editor is used to input all required data; however, the data are not copied to the file until the commit() method is called. Listing 12.1 shows a simple method that stores the username in the shared preferences file.

```
1 public void saveUserNamePreference (String userName){
2   SharedPreferences settings = getSharedPreferences(filename, 0);
3   SharedPreferences.Editor editor = settings.edit();
4   editor.putString(``user_name'', userName};
5   editor.commit();
6 }
```

Listing 12.1: Saving the user's login name in the shared preferences file.

The procedure to read the data from the shared preferences file is similar. The two lines of code below show how to retrieve the username of the user.

```
SharedPreferences settings = getSharedPreferences(filename, 0);
username = settings.getString(``user_name'', ``'');
```

12.2 Working with files

The second option to save/retrieve data in the Android platform is working with files. To store a file in the internal memory of the smartphone, the first step is to create the file. This is done by invoking the method Context.openFileOutput(filename, mode). This method returns a FileOutputStream object that provides the methods to write and append.

When creating the file for writing purposes, the available modes are the following:

- Context.MODE_PRIVATE: Opens the file to read/write and avoids other processes to access the file currently.
- Context.MODE_APPEND: Appends data to the end of the file if the file exists; if it does not exists, it creates the file.
- Context.MODE_WORLD_READABLE: Gives access to other applications to read the file.

- `Context.MODE_WORLD_WRITABLE`: Gives access to other applications to write to the file.

All the files created in internal storage are stored in a folder located in the internal memory of the device `(/data/data/[app_name]/files/)`.

Listing 12.2 shows the content of the `saveInternalMemoryFile()` method used as an example to save data into an internal file. Here the `theWriteString.getBytes()` method encodes the string to be written in the file into a sequence of bytes using the platform's default character set.

```
1  private void saveInternalMemoryFile(String username){
2    FileOutputStream fos = openFileOutput(fileName, Context.MODE_PRIVATE
         );
3    String theWriteString = "User Name: "+ username+"\n";
4
5    fos.write(theWriteString.getBytes());
6
7    fos.close();
8  }
```

Listing 12.2: Saving data into an internal file.

The reading operation is similar. Listing 12.3 consists of the method that shows how to open the file just created `(readInternalMemoryFile())` and read its content. To open a file for reading purposes the method utilized is `Context.openFileInput(filename)`. This will open a file for reading in private mode and it will return a `FileInputStream` object. Whether the file is open for reading or storing, the API documentation states that the filename should be written without path separators.

```
1  private void readInternalMemoryFile(filename){
2    FileInputStream fis = Context.openFileInput(filename, Context.
         MODE_PRIVATE);
3
4    StringBuffer sBuffer =  new StringBuffer();
5
6    DataInputStream dataIO = new DataInputStream(fis);
7
8    String strLine;
9
10   while((strLine = dataIO.readLine()) != null)
11     {
12     sBuffer.append(strLine + "\n");
13     }
14
15   fis.close();
16 }
```

Listing 12.3: Reading data from an internal file.

To utilize the external memory (SD-Card) of the device, the procedure is similar to the one used for internal memory. However, there are three major differences:

- The application must explicitly check for the presence of the external storage before opening a file for reading or writing. (Check if the external memory is mounted).

- The files will be visible to any application. This does not happen with internal storage because all files created by an application are private by default.

- When using external storage, the following permission should be added to the manifest: `android.permission.WRITE_EXTERNAL_STORAGE`.

After the program checks that the external storage is mounted, the next step consists of opening the file for reading/writing/appending and getting the input/output stream objects to perform the operations on the file. Finally, the file must be closed. Listing 12.4 includes an example to save data in the SD card of a smartphone. In the code, the `Environment.getExternalStorageState()` method is used to obtain the current state of the external media and the `Environment.MEDIA_MOUNTED.equals(stat)` method checks if the external media is mounted. After that, the procedure to write to the SD card is equal to writing a file in internal memory. Listing 12.4 is missing a few more lines of code to handle the case where the SD card is not available. The reading operation can be accomplished by inserting the code in Listing 12.3 in the appropriate place in Listing 12.4.

```
1  private void saveExternalMemoryFile(){
2    String state = Environment.getExternalStorageState();
3
4    if (Environment.MEDIA_MOUNTED.equals(state))
5      {
6      File file = new File(Environment.getExternalStorageDirectory(),
           fileName);
7      OutputStream fos = new FileOutputStream(file);
8
9      String theWriteString = "User Name: "+ username+"\n";
10
11     fos.write(theWriteString.getBytes());
12
13     fos.close();
14     }
15 }
```

Listing 12.4: Saving data to a file in an external SD card.

12.3 SQLite databases

Android allows every application installed in the device to have its own database managed by the SQLite engine. Each application maintains its database in different files that are separated from the databases of other applications. There is no need for data base administration or installing other components, as the Android framework provides this functionality. All classes

that are needed to create, query, store, modify, and delete tables and records in the database are under the `android.database.sqlite.*` package.

Because the design goal is to be embeddable and lightweight, SQLite does not support all the data types available in the SQL standard. SQLite only handles integer types, similar to Java's long type, real type, similar to Java's double type, and text type, similar to Java's string type.

Working with SQLite involves a three-step procedure:

1. Create a class to create, delete, and update the database table(s).
2. Create a class with functions to populate and query the tables.
3. Modify the user interface of the application's activity to save/retrieve data to/from the database.

The class `SQLiteOpenHelper`, included in the `android.database.sqlite.*` package, contains all the methods to create, destroy, and update a database schema. For example, the `SQLiteOpenHelper` class contains the abstract `onCreate()` method, which is called when the database is created for the first time, to handle the creation of the database schema as well as the `onUpgrade()` method, which is invoked when the database schema needs to be changed. As an example, Listing 12.5 shows the content of the class `MyNewDatabase` extending from the `SQLiteOpenHelper` class used to create a new database by the name "my_new_database." The database consists of one table called "classgrades," with columns to include automatic ida, and the names, last names, and grades of the students in a class.

```
1  public class MyNewDatabase extends SQLiteOpenHelper{
2    private static final String DB_NAME = ``my_new_database'';
3    private static final String DB_VERSION = 1;
4
5    // Database creation SQL statement
6    private static final String CREATE_MYNEWTABLE =
7        ``create table classgrades('' +
8        ``_id integer primary key autoincrement, `` +
9        `` name text not null, '' +
10       ``lastname text not null, '' +
11       ``grade integer not null)'';
12
13   public void onCreate(SQLiteDatabase database){
14     database.execSQL(CREATE_MYNEWTABLE);
15     }
16 }
```

Listing 12.5: Creating a new SQLite database.

Another important class in the `android.database.sqlite.*` package is the `SQLiteDatabase` class, which exposes the methods to manage a SQLite database such as create, insert, delete, execute SQL commands, and perform other common database management tasks. In order to store data in a SQLite database, data must be encapsulated in a `ContentValues` object first. Listing 12.6 shows how the data is first wrapped in a `ContentValues` object

and then inserted into the database. In the listing, it is assumed that a variable `database` of the `SQLiteDatabase` type has been instantiated and that this variable contains a handle to the `my_new_database`, which was opened for writing using the `getWritableDatabase()` method of the `SQLiteOpenHelper` class.

```
1  private ContentValues createStudentValues(String name, String lastname
       , int grade){
2    ContentValues values = new ContentValues();
3    values.put(new_name, name);
4    values.put(new_lastname, lastname);
5    values.put(new_grade, grade);
6    return values;
7    }
8
9  public long createStudentRecord(String name, String lastname, int
       grade){
10   ContentValue initialValues = createStudentValues(name, lastname,
         grade);
11   return database.insert(DB_NAME, null, initialValues);
12   }
```

Listing 12.6: Inserting data in the database.

Similarly, when reading records from a SQLite database, the `Cursor` interface provides random read-write access to the result set returned by a database query. Listing 12.7 shows a function that gets all student records from the database into a `cur` object. Then, this object can be manipulated in many different ways using any of the methods available from the `Cursor` interface. For example, the `cur.getCount()` method provides the number of records in the object; `getColumnCount()` returns the number of columns; `moveToFirst()` and `moveToLast()` move the cursor to the first and last row, respectively; and many others. A complete list of the methods exposed by the `Cursor` interface can be found in `http://developer.android.com/reference/android/database/Cursor.html`.

```
1  public Cursor getAllStudentRecords(){
2    return database.query(classgrades, new String[] {_id, name, lastname
         , grade}, null, null, null, null, null};
3    }
4
5  Cursor cur = myGradesDBManager.getAllStudentRecords();
```

Listing 12.7: Querying the database.

Chapter 13

Feature Extraction

As was mentioned in Section 4.1.2, *feature extraction* aims to filter the most relevant information within the collected signals and helps the classifier to effectively discover patterns within the data. In our context, measurements collected from the sensors are naturally indexed over the time dimension, which makes HAR a time-series classification problem. Nonetheless, the sampling rate of the sensors is not necessarily the same, thus, an accelerometer could provide observations at 100Hz whereas a heart monitor might deliver measurements at 1 Hz. This fact differentiates HAR from a typical multi-dimensional time series classification problem. The present chapter addresses the methodology followed by Vigilante to extract features from the measured signals in real time, including the data abstraction and the implementation of a number of well-known feature extraction procedures.

13.1 Data representation

Vigilante employs four basic data abstractions to enable feature extraction: namely `Point`, `TimeSeries`, `TimeWindow`, and `FeatureSet`. The first three classes are part of the package `edu.usf.cse.vigilante.common.data`, while `FeatureSet` and other classes directly related to feature extraction are in package `edu.usf.cse.vigilante.common.feature`.

13.1.1 Point

Each sensor measurement is represented as an instance of the class `Point`, as shown in Listing 13.1. This class defines an ordered pair $(time, value)$ which abstracts a data sample in a time series. Class `Point` implements interface `java.lang.Comparable` to define the total order between two given points using the time instant value as the comparison criterion. For two given points `p1` and `p2`, `p1` will be greater than `p2` if `p1.time` is greater than `p2.time`, and `p1.compareTo(p2)` will return 1. In turn, `p2` will be greater than `p1` if `p2.time` is greater than `p1.time` and `p1.compareTo(p2)` will be -1. Finally, in the

case of `p1.time` equals `p2.time`, `p1.compareTo(p2) == p2.compareTo(p1) == 0`. The main purpose of defining comparable points is to detect and avoid multiple points on a single time instant in a time series and facilitate temporal sorting. This will be later used by class `TimeSeries`.

```
1
2  package edu.usf.cse.vigilante.common.data;
3
4  public class Point implements Comparable <Point> {
5
6    /** The time instant*/
7    private double time;
8
9    /** The value of the variable of interest at the given time*/
10   private double value;
11
12   public Point(double time, double value) {
13     this.time = time;
14     this.value = value;
15   }
16
17   public double getTime() {
18     return time;
19   }
20
21   public void setTime(double time) {
22     this.time = time;
23   }
24
25   public double getValue() {
26     return value;
27   }
28
29   public void setValue(double value) {
30     this.value = value;
31   }
32
33   /** Compares two points according to their time instant value*/
34   public int compareTo(Point p) {
35     if (p == null) {
36       throw new IllegalArgumentException("Cannot compare to a null
             value");
37     } else if (this.time < p.time) {
38       return -1;
39     } else if (this.time > p.time) {
40       return 1;
41     } else {
42       return 0;
43     }
44   }
45 }
```

Listing 13.1: The implementation of class `Point`.

13.1.2 Time series

A time series is a set of measurements from one single variable over time such as, for instance, the heart rate signal or the acceleration in the X axis; in other words, an array of points. Class `TimeSeries` (see Listing 13.2) is then

introduced as an extension of `ArrayList <Point>` taking advantage of the methods provided by class `ArrayList` to easily add, remove, insert, and sort elements, among others. The constructor of class `TimeSeries` receives two parameters, namely `id` and `label`. The `id` uniquely identifies a time series in the system (say "HR"), whereas `label` is a more meaningful description of the variable (e.g., "heart rate").

The class `TimeSeries` incorporates a mechanism to avoid having more than one point at a single time instant within method `addPoint(Point)`. Such method invokes inherited method `contains(Point)` from class `ArrayList` to search for any point in the same time instant and, if it finds one, it refrains from adding the point to the series. The comparison is achieved by an internal call to method `compareTo(Point)` from class `Comparable`, which was implemented in class `Point` and defines two points as equal if they are defined in the same time instant. Finally, in order to guarantee data integrity, inherited methods `add(int, Point)` and `add(Point)` — which do not account for duplicated points — are logically disabled by overriding them and throwing an exception if they are invoked.

```
1
2  import java.util.ArrayList;
3
4  public class TimeSeries extends ArrayList <Point> {
5
6    /** The time series description*/
7    private String label;
8
9    /** The time series unique ID*/
10   private String id;
11
12   public TimeSeries(String id, String label) {
13     this.label = label;
14     this.id = id;
15   }
16
17   /** Adds a point to the series if there is no other point at the
         same time instant*/
18   public boolean addPoint(Point p) {
19     if ((p != null) && (!this.contains(p))) {
20       return super.add(p);
21     } else {
22       return false;
23     }
24   }
25
26   public String getLabel() {
27     return label;
28   }
29
30   public String getId() {
31     return id;
32   }
33
34   public boolean add(Point p) {
35     throw new UnsupportedOperationException("This method is not
           supported. Use addPoints instead.");
36   }
37
```

```
38   public void add(int x, Point p) {
39     throw new UnsupportedOperationException("This method is not
           supported. Use addPoints instead.");
40   }
41 }
```

Listing 13.2: The implementation of class `TimeSeries`.

13.1.3 Time window

Since a HAR system usually collects data from more than one variable (e.g., heart rate, acceleration in all three dimensions, etc.), a higher abstraction has been made to group time series into the so-called *time windows*. A time window, represented as an instance of class `TimeWindow` (see Listing 13.3), encapsulates a collection of time series defined between the same time interval by extending from class `Hashtable <String, TimeSeries>`. This abstraction allows to organize a set of instances of `TimeSeries` by a `String` value, which will be the series `id`, and enables an easier and quicker access to the elements (i.e., time series) in the collection.

The method `put(String, TimeSeries)`, inherited from `Hashtable` and used to append an element to the collection, requires two parameters: a key (i.e., index) and the actual element (i.e., time series). To simplify that, the method `addTimeSeries(TimeSeries)` was incorporated, which only requires a `TimeSeries` object and automatically uses its `id` as the key. Method `put(String, TimeSeries)` has therefore been overridden to guarantee key consistency and an exception will be thrown if it is called from any class. In order to maintain naming consistency, method `get(String)` has been renamed as `getTimeSeries(String)` using the same concept. As it will be discussed in further sections, instances from class `TimeWindow` are the input of feature extraction algorithms.

```
1
2  package edu.usf.cse.vigilante.common.data;
3
4  import java.util.Hashtable;
5  import edu.usf.cse.vigilante.common.feature.FeatureSet;
6
7  public class TimeWindow extends Hashtable <String, TimeSeries> {
8
9    /** The activity label associated with this time window*/
10   private String label;
11
12   /** The set of extracted features from this time window*/
13   private FeatureSet featureSet;
14
15   public TimeWindow(String label) {
16     this.label = label;
17   }
18
19   public void addTimeSeries(TimeSeries series) {
20     super.put(series.getId(), series);
21   }
```

```
22
23    public TimeSeries getTimeSeries(String seriesId) {
24      return super.get(seriesId);
25    }
26
27    public TimeSeries put(String key, TimeSeries series) {
28      throw new UnsupportedOperationException("Use method addTimeSeries
            rather than put!");
29    }
30
31    public TimeSeries get(String id) {
32      throw new UnsupportedOperationException("Use method getSignal
            rather than get!");
33    }
34
35    public FeatureSet getFeatureSet() {
36      return featureSet;
37    }
38
39    public void setFeatureSet(FeatureSet featureSet) {
40      this.featureSet = featureSet;
41    }
42
43    public String getLabel() {
44      return label;
45    }
46
47    public void setLabel(String label) {
48      this.label = label;
49    }
50  }
```

Listing 13.3: The implementation of class `TimeWindow`.

13.1.4 Feature set

A *feature set* is a collection of the outputs generated by the feature extraction algorithms, and hence a classification instance from the machine learning point of view. Programmatically, it is represented by class `FeatureSet` (defined in Listing 13.4), which extends from `Hashtable <String, Double>`. The keys are feature identifiers (e.g., "accX_mean" standing for the mean acceleration in the X axis), whereas the values are the result of the feature extraction algorithms. A feature set might be either labeled or unlabeled according to the `activityLabel` attribute. If `activityLabel` is different than `null`, the feature set would be a training instance; otherwise, it would be an instance to be classified (i.e., a window whose activity needs to be determined).

Class `FeatureSet` has three public constructors. The first one, `FeatureSet(String)`, creates an empty feature set with the given activity label. The second constructor, `FeatureSet(double[], String[])` creates a feature set from the given list of double values using the attribute names as keys. Finally, `FeatureSet(TimeWindow)` is the most powerful constructor and it automatically extracts features from each time series within the given time window. For that purpose, instances of `StatisticalFeatureExtractor`, `TransientFeatureExtractor`, and `StructuralFeatureExtractor` are created. These classes incorporate a variety of feature extraction algorithms and

will be covered in further sections. Once all features are extracted and assigned to double variables (e.g., `mean`, `stdv`, etc.), they are appended to the feature set instance by means of method `put(String, Double)`. Class `FeatureSet` also provides method `addFeatures(FeatureSet)` to perform the union of two feature sets.

The reader may notice that the presented implementation extracts the same set of features for all time series. In real scenarios, the nature of each signal usually leads to particular feature extraction mechanisms to exploit specific signal characteristics. For that purpose, an additional matching between signals and features could be easily implemented by adding an attribute `type` to the class `TimeSeries` and then computing specific subsets of features for each signal type.

Finally, method `toInstance()` converts the current feature set into its WEKA equivalent, `weka.core.Instance`, enabling the integration of WEKA capabilities into Vigilante. Chapter 14 will cover the implementation details on WEKA integration.

```
1
2  package edu.usf.cse.vigilante.common.feature;
3
4  import java.util.Enumeration;
5  import java.util.Hashtable;
6  import weka.core.Attribute;
7  import weka.core.Instance;
8  import weka.core.Instances;
9  import edu.usf.cse.vigilante.common.data.TimeWindow;
10
11 public class FeatureSet extends Hashtable <String, Double> {
12
13   private String activityLabel;
14
15   public FeatureSet(String activityLabel) {
16     this.activityLabel = activityLabel;
17   }
18
19   public FeatureSet(double[] values, String[] attributes) {
20     if ((values == null) || (attributes == null) || (values.length) !=
           (attributes.length)) {
21       throw new IllegalArgumentException("Invalid arguments to create
             a feature set");
22     }
23     for (int i = 0; i < values.length; i++) {
24       setAttribute(attributes[i], values[i]);
25     }
26   }
27
28   /** Creates a new feature set by extracting all features from the
         given time window*/
29   public FeatureSet(TimeWindow window) {
30     if (window == null) {
31       throw new IllegalArgumentException("Cannot extract features from
             a null time window!");
32       return;
33     }
34
35     for (TimeSeries series : window) {
```

```
36        StatisticalFeatureExtractor sta = new
              StatisticalFeatureExtractor(series);
37        TransientFeatureExtractor tra = new TransientFeatureExtractor(
              series);
38        StructuralFeatureExtractor str = new StructuralFeatureExtractor(
              series);
39
40        /** Compute some statistical features*/
41        double mean = sta.mean();
42        double stdv = sta.stdv();
43        double var = sta.variance();
44
45        /** Compute some structural features */
46        int degree = 3;
47        double[] coef = str.computeLeastSquares(degree);
48
49        /** Compute all transient features*/
50        FeatureSet fs = tra.computeFeatures();
51
52        /** Add them to the feature set*/
53        String id = series.getId()
54
55        this.put(id + "_mean", mean);
56        this.put(id + "_stdv", stdv);
57        this.put(id + "_var", var);
58        this.put(id + "_energy", energy);
59        this.put(id + "_coef1", coef[0]);
60        this.put(id + "_coef2", coef[1]);
61        this.put(id + "_coef3", coef[2]);
62
63        this.addFeatures(fs);
64      }
65    }
66
67    public void addFeatures (FeatureSet featureSet) {
68      for (String name : featureSet.keySet()) {
69        Double value = featureSet.get(name);
70        this.put(name, value);
71      }
72    }
73
74    public String getActivityLabel() {
75      return activityLabel;
76    }
77
78    public Double getValue(String attName) {
79      return super.get(attName);
80    }
81
82    public void setAttribute(String attName, Double value) {
83      super.put(attName, value);
84    }
85
86    /**
87     * Converts a FeatureSet to a WEKA Instance
88     * Parameter instanceHeader is an Instances object containing the
          attributes that the Instance should have.  */
89    public Instance toInstance(Instances instanceHeader) {
90      Instance instance = null;
91      if (instanceHeader != null) {
92        instance = new Instance(instanceHeader.numAttributes());
93        instance.setDataset(instanceHeader);
94
95        Enumeration e = instanceHeader.enumerateAttributes();
96        while (e.hasMoreElements()) {
97          Attribute attr = (Attribute) e.nextElement();
98          if (this.containsKey(attr.name())) {
```

```
99            if (attr.isNominal()) {
100             instance.setValue(attr, "" + this.getValue(attr.name()));
101             } else {
102                 instance.setValue(attr, this.getValue(attr.name()));
103               }
104             } else {
105               /* Attributes not found in this featureSet are set to
                      zero.
106               Otherwise the classifier cannot evaluate the instance */
107               if (attr.isNominal()) {
108                 instance.setValue(attr, "0.0");
109               } else {
110                 instance.setValue(attr, 0);
111               }
112             }
113           }
114     }
115
116     return instance;
117   }
118
119 }
```

Listing 13.4: The implementation of class `FeatureSet`.

13.2 Feature extraction computations

Listing 13.5 presents the implementation of class `FeatureExtractor`, an abstract class that defines the generic behavior of any feature extraction algorithm. The constructor takes as input an instance of `TimeSeries`, which is the signal from which the features will be extracted, along with an alphanumeric `activityLabel`. The actual calculations are done by abstract method `computeFeatures()`, which every class extending from `FeatureExtractor` must implement. In the end, the extracted features should be stored in the protected attribute `featureSet`.

```
1
2  package edu.usf.cse.vigilante.common.feature;
3
4  import edu.usf.cse.vigilante.common.data.TimeSeries;
5
6  public abstract class FeatureExtractor {
7
8    /** The series features will be extracted from*/
9    protected TimeSeries series;
10
11   /** The set of features extracted from the given time series*/
12   protected FeatureSet featureSet;
13
14   public FeatureExtractor(TimeSeries series, String activityLabel) {
15     this.series = series;
16     featureSet = new FeatureSet(activityLabel);
17   }
```

```
18
19    /** Every class extending from FeatureExtractor must implement a
          particular feature extraction algorithm here*/
20    public abstract FeatureSet computeFeatures();
21
22    public final FeatureSet getFeatureSet() {
23      return featureSet;
24    }
25  }
```

Listing 13.5: The implementation of class `FeatureExtractor`.

13.2.1 Statistical features

Class `StatisticalFeatureExtractor` — shown in Listing 13.6 — contains implementations of the following features: mean, standard deviation, root mean square (RMS), mean absolute deviation (MAD), median, and variance. Note that there are some common computations for these features, which could be leveraged to optimize the implementation. For instance, the mean value is used in the standard deviation calculations whereas the mean and the sum of the squared values could be computed within a single loop; the same optimization also applies to the summations for the MAD and the variance. Therefore, these common calculations are compiled in method `prepareFeatures()`, which is called directly from the constructor to assure their availability. An external class may either compute all statistical features at once using the method `computeFeatures()` or only a subset of them by individually invoking the corresponding public methods.

Note that these implementations run in linear time except for the median, which requires sorting in $O(n \log n)$. However, if the time series is already sorted (assuming there is no jitter in the sensor measurements), the median could be computed in constant time.

```
1
2  package edu.usf.cse.vigilante.common.feature;
3
4  import java.util.Collections;
5  import edu.usf.cse.vigilante.common.data.TimeSeries;
6  import edu.usf.cse.vigilante.common.data.Point;
7
8  public class StatisticalFeatureExtractor extends FeatureExtractor {
9
10   private double mean;
11   private double sumSquared;
12   private double sumMad;
13   private double variance;
14   private double mad;
15
16   public StatisticalFeatureExtractor(TimeSeries s, String label) {
17     super(s, label);
18     prepareFeatures();
19   }
20
21   public FeatureSet computeFeatures() {
```

```
22      featureSet.put("MEAN_" + series.getId(), mean);
23      featureSet.put("STDV_" + series.getId(), computeSTDV());
24      featureSet.put("RMS_" + series.getId(), computeRMS());
25      featureSet.put("MAD_" + series.getId(), computeMAD());
26      featureSet.put("VAR_" + series.getId(), computeVariance());
27      return featureSet;
28    }
29
30    public double getMean() {
31      return mean;
32    }
33
34    public void recomputeFeatures(TimeSeries series) {
35      this.series = series;
36      computeFeatures();
37    }
38
39    private void prepareFeatures() {
40      mean = 0;
41      sumSquared = 0;
42      sumMAD = 0;
43      for (Point p : series) {
44        mean += p.getValue();
45        sumSquared += Math.pow(p.getValue(), 2);
46      }
47      mean = mean / series.size();
48      for (Point p : series) {
49        sumMad += Math.abs(p.getValue() - mean);
50        sumVar += Math.pow(p.getValue() - mean, 2);
51      }
52      variance = sumVar / (series.size() - 1);
53      mad = sumMad / series.size();
54    }
55
56    public double computeSTDV() {
57      return Math.sqrt(variance);
58    }
59
60    public double computeRMS() {
61      return Math.sqrt(sumSquared/series.size());
62    }
63
64    public double computeMedian(TimeSeries series) {
65        Collections.sort(series);
66
67      if (series.size() % 2 == 1) {
68          return series.get((series.size() - 1) / 2).getValue();
69      } else {
70        double lower = series.get(series.size() / 2 - 1).getValue();
71        double upper = series.get(series.size() / 2).getValue();
72        return (lower + upper) / 2;
73      }
74    }
75
76    public double computeMAD() {
77      return mad;
78    }
79
80    public double computeVariance() {
81      return variance;
82    }
83  }
```

Listing 13.6: The implementation of class `StatisticalFeatureExtractor`.

13.2.2 Structural features

Class `StructuralFeatureExtractor` allows to compute the coefficients of the polynomial that best fits a given time series using the Least Squares (LS) algorithm. The implementation is shown in Listing 13.7, and it was adapted from [10]. The method `computeLeastSquares(int)` takes as input the number of coefficients (i.e., `coefCount`), which is one plus the degree of the polynomial. If there are more coefficients than points, the degree is set to the number of points and the remaining coefficients are filled with zero values; this is to maintain consistency with the coefficient array size. For the LS method, the JAMA library [9] is required to solve a system of linear equations via LU decomposition. Since JAMA implements a naïve matrix multiplication, the computational complexity of structural features is $O(n^3)$. But better algorithms have been devised for matrix multiplication, such as Strassen's ($O(n^{2.907})$) and Coppersmith's ($O(n^{2.376})$) [41]. Furthermore, recent results show that a system of linear equations could be solved in $O(m(\log n)^2)$ where m is the number of non-zero elements in the matrix [73]. Yet, these algorithms hide a large constant coefficient, being effective only for very large matrices.

Additional details on structural feature extraction can be found in Section 4.1.2.2 and [87].

```
1
2  package edu.usf.cse.vigilante.common.feature;
3  import edu.usf.cse.vigilante.common.data.TimeSeries;
4
5  public class StructuralFeatureExtractor extends FeatureExtractor {
6
7    public FeatureSet computeFeatures() {
8      double[] features = computeLeastSquares(degree + 1));
9      int i = 0;
10     for (double f : features) {
11       featureSet.put("COEF_" + degree + "^" + i + "_" + series.getId()
             , f);
12       i++;
13     }
14     return featureSet;
15   }
16
17   public double[] computeLeastSquares(int coefCount) throws Exception
         {
18
19     if (coefCount > series.size()) {
20       double[] coef1 = new double[coefCount];
21       double[] coef2 = computeLeastSquares(series.size());
22       for (int i = 0; i < series.size(); i++) {
23         coef1[i] = coef2[i]
24       }
25       for (int i = series.size(); i < coefCount; i++) {
26         coef1[i] = 0;
27       }
28       return coef1;
29     } else {
30       Jama.Matrix M = new Jama.Matrix(series.size(), coefCount);
31       Jama.Matrix B;
32
33       for (int j = 0; j < coefCount; j++) {
```

```
34          for (int i = 0; i < series.size(); i++) {
35            M.set(i, j, Math.pow((series.get(i).getTime() - series.get
                  (0).getTime()), j));
36          }
37        }
38
39        B = new Jama.Matrix(series.size(), 1);
40        for (int i = 0; i < series.size(); i++) {
41          B.set(i, 0, (series.get(i).getValue()));
42        }
43
44        return M.solve(B).getRowPackedCopy();
45      }
46    }
47  }
```

Listing 13.7: The implementation of class `StructuralFeatureExtractor`.

13.2.3 Transient features

Transient features are structural in nature but provide different measures of the data shape. The reader can refer to Section 4.1.2.3 for the formal definition of these features. Their implementations are contained in class `TransientFeatureExtractor` (see Listing 13.8). In the first place, the *trend* is calculated by method `computeTrend()`, and requires a linear regression (method `slope()`) to calculate the slope of the line that best fits the points in the time series. Then, according to the value of the variable `SLOPE_THRESHOLD`, the trend could be either 1 (i.e., increasing), 0 (i.e., constant), or -1 (decreasing).

On the other hand, the *magnitude of change* (MOC) is computed by method `computeMOC()` and uses the parameter `MOC_SERIES_PERCENTAGE` to calculate the symmetric maximum deviation between the beginning and the end of the signal. These implementations run in linear time, providing a more efficient way to describe the signal shape with respect to structural features.

```
1  package edu.usf.cse.vigilante.common.feature;
2
3  import edu.usf.cse.vigilante.common.data.Point;
4  import edu.usf.cse.vigilante.common.data.TimeSeries;
5
6  public class TransientFeatureExtractor extends FeatureExtractor{
7
8    public FeatureSet computeFeatures() {
9      featureSet.put("trend_" + series.getId(), computeTrend());
10     featureSet.put("moc_" + series.getId(), computeMOC());
11     return featureSet;
12   }
13
14   /** This method performs a linear regression on a TimeSeries,
         returning the slope.*/
15   private double slope() {
16     double sumT = 0;
17     double sumV = 0;
18     for (Point p : series) {
19       sumT += p.getTime();
```

```
20        sumV += p.getValue();
21        }
22      double meanT = sumT / series.size();
23      double meanV = sumV / series.size();
24
25      double w = 0, z = 0;
26      for (Point p : series) {
27        w += (p.getTime() - meanT) * (p.getTime() - meanT);
28        z += (p.getTime() - meanT) * (p.getValue() - meanV);
29        }
30      return  = z / w;
31    }
32
33    private double computeTrend(double slope) {
34    if (slope >= SLOPE_THRESHOLD) {
35      return 1;
36      }
37      else if (slope <= TREND_THRESHOLD) {
38        return -1;
39      }
40      return 0;
41    }
42
43    public double computeTrend() {
44      return computeTrend(slope());
45    }
46
47    public double computeMOC() {
48      if (series.size() == 1) {
49        return 0;
50        }
51        double start = series.get(0).getTime();
52        double end = series.get(series.size() - 1).getTime();
53
54        double spMinus = start+(end-start) * MOC_SERIES_PERCENTAGE;
55        double spPlus = start+(end-start)*(1 - MOC_SERIES_PERCENTAGE);
56
57        double maxPlus = series.get(0).getValue();
58        double minPlus = maxPlus;
59        double maxMinus = series.get(series.size() - 1).getValue();
60        double minMinus = maxMinus;
61
62      for (int i = 0; series.get(i).getTime() <= spMinus; i++) {
63        double value = series.get(i).getValue();
64        maxMinus = Math.max(maxMinus, value);
65        minMinus = Math.min(minMinus, value);
66      }
67
68      for (int i = series.size()-1; series.get(i).getTime() >= spPlus; i
            --) {
69        double value = series.get(i).getValue();
70        maxPlus = Math.max(maxPlus, value);
71        minPlus = Math.min(minPlus, value);
72      }
73
74      return Math.max(Math.abs(maxPlus - minMinus), Math.abs(maxMinus -
            minPlus));
75    }
76  }
```

Listing 13.8: The implementation of class `TransientFeatureExtractor`.

Chapter 14

Real-Time Classification in Smartphones Using WEKA

In previous chapters, it has been shown that human activity recognition systems rely on machine learning algorithms to predict an individual's activity during a certain period of time. In addition, it has been emphasized that different classification methods could be used in HAR systems, depending on the specific characteristics of each scenario (e.g., the set of activities, the type of sensors, and so forth). Chapter 5 elaborates on the advantages of implementing a completely mobile HAR system in terms of reliability, scalability, and energy consumption, just to mention a few. But such a task entails the evaluation of a classification model on the smartphone, which brings about an additional challenge: implementing each and every classifier under the Android platform. This could be very time consuming given the underlying complexity in the implementation of machine learning algorithms, along with the computational constraints present in mobile devices. The focus of this chapter, therefore, is to leverage the implementations of a number of classification methods provided by WEKA to enable mobile classifier evaluation under the Android framework.

14.1 A first look into WEKA

The *Waikato Environment for Knowledge Analysis* (WEKA) is one of the most widely used tools in the machine learning research community. It encompasses a variety of machine learning algorithms including classification, clustering, and association rules. Particularly, in the activity recognition context, the focus here will be on classification.

WEKA integrates an intuitive graphical user interface that enables training and evaluating a classification algorithm with just a few clicks. In this chapter, the WEKA Java API and its integration with the Android mobile platform will be explained in more detail. The interested reader may find more information on the WEKA explorer and experimenter in [18].

14.2 WEKA API

WEKA features a very powerful Java programming interface for the rapid prototyping of machine learning applications. This section presents the API fundamentals before covering the mobile WEKA integration. For supervised learning, it is necessary to first understand four main classes:

- `weka.core.Instance`: a learning instance, also called *example* or *sample*. It represents a vector of values from the given set of attributes in a learning problem.
- `weka.core.Isuchnstances`: a learning dataset, conformed by a set of instances which may or may not be labeled.
- `weka.classifiers.Classifier`: the super class for all classification algorithms, declaring abstract methods `buildClassifier()` and `classifyInstance(Instance)`.
- `weka.classifiers.Evaluation`: a class for evaluating a previously trained classification model using cross-validation and random split, among other methodologies. It also computes the confusion matrix, accuracy, precision, recall, and other classification metrics.

The following source code fragments have been adapted from [14, 16]. In these references, the interested reader may find further information.

14.2.1 ARFF format

The *attribute-relation file format* (ARFF) is the main format supported by WEKA. It embeds a header specifying the dataset schema along with the data contents into one single file. Listing 14.1 shows an example of an ARFF file for the well-known weather dataset [15]. The first line ‘‘`@relation weather`’’ specifies the dataset name. Then, the next `@attribute` tags define the attribute names and types. If an attribute is nominal, the set of possible values must be specified after the attribute name. In the case of real-valued attributes, the keyword `real` must be included. After all attributes have been specified, the `@data` tag must appear preceding the data themselves in a *comma-separated value* (CSV) format.

```
1
2  @relation weather
3
4  @attribute outlook {sunny, overcast, rainy}
5  @attribute temperature real
6  @attribute humidity real
7  @attribute windy {TRUE, FALSE}
8  @attribute play {yes, no}
9
10 @data
11 sunny,85,85,FALSE,no
12 sunny,80,90,TRUE,no
13 overcast,83,86,FALSE,yes
14 rainy,70,96,FALSE,yes
15 rainy,68,80,FALSE,yes
16 rainy,65,70,TRUE,no
17 overcast,64,65,TRUE,yes
18 sunny,72,95,FALSE,no
19 sunny,69,70,FALSE,yes
20 rainy,75,80,FALSE,yes
21 sunny,75,70,TRUE,yes
22 overcast,72,90,TRUE,yes
23 overcast,81,75,FALSE,yes
24 rainy,71,91,TRUE,no
```

Listing 14.1: An example of the ARFF format for the well-known weather dataset.

14.2.2 Reading an ARFF file

The WEKA API provides an easy way to read an ARFF file. Listing 14.2 presents a code snippet to do so. First, a `DataSource` object needs to be instantiated, passing the data location as a parameter, which could even be a URL repository or a JDBC connection. In this example, a local file path was used for the sake of simplicity. Then, method `getDataSet()` is invoked to return a reference to the dataset itself as an object from class `Instances`. Finally, the target class is specified as the last attribute by method `setClassIndex(int)`.

```
1
2  import weka.core.converters.ConverterUtils.DataSource;
3  import weka.core.Instances;
4
5  ...
6
7    DataSource source = new DataSource("/some/where/data.arff");
8
9    Instances data = source.getDataSet();
10
11   /* Set the very last attribute as the target class */
12   data.setClassIndex(data.numAttributes() - 1);
13
14 ...
```

Listing 14.2: Reading an ARFF file.

14.2.3 Classifier training

Listing 14.3 shows the Java code to build a C4.5 decision tree classifier using two different approaches: (1) an instance of wrapper class `J48` and (2) a generic instance of the `Classifier` class. In the first approach, a new instance of `J48` is created and the classifier parameters are passed as a `String` array to the method `setOptions(String[])`. In this particular case, options `["-C 0.25", "-U"]` refer to an unpruned tree with a confidence factor of 0.25 — each classifier may have different parameters and options. Then, method `buildClassifier()` is invoked to train the C4.5 classifier. In the second approach, static method `forName(String, String[])` in class `Classifier` is called to create an instance of the C4.5 classifier. Two parameters are passed to such a method: the classifier name and the classifier options.

```
1
2  import weka.classifiers.trees.J48;
3
4  ...
5
6    /** First approach: using classifier wrapper classes, i.e., J48*/
7    String[] options = new String[2];
8
9    /** Options: Unpruned tree with confidence factor of 0.25 */
10   options[0] = "-C 0.25";     // confidence factor
11   options[1] = "-U";          // unpruned tree
12
13   /** Create a new instance of J48 with the given options*/
14   J48 tree = new J48();
15   tree.setOptions(options);
16
17   /** Train the classifier*/
18   tree.buildClassifier(data);
19
20 ...
21
22   /** Second approach: using weka.classifiers.Classifier superclass*/
23
24   String classifierParameters = "-C 0.25 -U";
25
26   Classifier classifier = Classifier.forName("weka.classifiers.trees.
         J48", weka.core.Utils.splitOptions(classifierParameters));
27
28   classifier.buildClassifier(data);
29
30 ...
```

Listing 14.3: Training a C4.5 classifier with WEKA.

14.2.4 Classifier evaluation

Once a model is trained, method `double classifyInstance(Instance)` can be used to generate a prediction for a given instance. If the target class is nominal, the predictions will be integer numbers ranging between zero and the

number of classes minus one. For real-valued learning problems, the output of `classifyInstance(Instance)` is the prediction itself.

The WEKA API makes it also possible to evaluate a model with an entire given test dataset. Listing 14.4 shows how to do so, assuming `train` and `test` are the corresponding datasets. After the classifier is built, an instance of class `Evaluation` is created, and method `evaluateModel(Classifier, Instances)` is immediately invoked to evaluate the given classifier with the specified testing dataset.

```
1
2  import weka.core.Instances;
3  import weka.classifiers.Evaluation;
4  import weka.classifiers.trees.J48;
5
6  ...
7
8    Instances train = ...   // a training dataset
9    Instances test = ...    // a testing dataset
10
11   // Train classifier
12   Classifier cls = new J48();
13   cls.buildClassifier(train);
14
15   // Evaluate classifier and print some statistics
16   Evaluation eval = new Evaluation(train);
17   eval.evaluateModel(cls, test);
18   System.out.println(eval.toSummaryString("\nResults\n", false));
19   Instance instance = test.get(i)
20   double pred = tree.classifyInstance(instance);
```

Listing 14.4: Evaluation of a classification model.

Class `Evaluation` also supports cross validation by means of method `crossValidateModel(Classifier, Instances, int, Random)`. The first parameter is the classifier to be used, followed by the dataset. A crossvalidation example is shown in Listing 14.5 using a decision tree. Then, the number of folds should be specified — a 10-fold cross validation is usually preferred. The last parameter is the random seed for the cross validation data split as an instance of class `Random`.

After the cross validation has been completed, the most standard classification metrics can be obtained using methods `confusionMatrix()`, `precision(int)`, `recall(int)`, `falsePositiveRate(int)`, `fMeasure(int)`, and `falseNegativeRate(int)`, among others.

```
1
2  import weka.classifiers.Evaluation;
3  import java.util.Random;
4  ...
5
6    Evaluation eval = new Evaluation(newData);
7
8    eval.crossValidateModel(tree, newData, 10, new Random(1));
9
```

```
10   double accuracy = eval.pctCorrect();
11
12   double[][] confusionMatrix = eval.confusionMatrix();
13
14   double[] precision = new double[nClasses];
15   double[] recall = new double[nClasses];
16   double[] fpr = new double[nClasses];
17   double[] fnr = new double[nClasses];
18   double[] f = new double[nClasses];
19
20   for (int i = 0; i < nClasses; i++) {
21     precision[i] = eval.precision(i);
22     recall[i] = eval.recall(i);
23     fpr[i] = eval.falsePositiveRate(i);
24     fnr[i] = eval.falseNegativeRate(i);
25     f[i] = eval.fMeasure(i);
26   }
27 ...
```

Listing 14.5: Crossvalidation example.

14.3 Enabling WEKA in an Android smartphone

It is a well known fact that training a classifier is generally more expensive than evaluating it except for instance-based algorithms, especially as the volume of input data increases. Hence, the proposed approach is to first train the classifier(s) on the server. Yet, the challenge is: how to make a previously trained classification model available in a smartphone? For that purpose, the Java serialization API is used to *serialize* (i.e., export) a `weka.core.Classifier` object — encapsulating a classification model — to a binary file. The file will later be downloaded and *de-serialized* by the smartphone. This implies that the `weka.jar` file containing WEKA libraries must also be in the Android's classpath.

14.3.1 Training and exporting a HAR classifier

Listing 14.6 displays the implementation of class `ClassifierSerializer`, which trains a classifier for a particular dataset and exports it to a file. The constructor receives four parameters: (1) the class name of the classification algorithm to be used, (2) the classifier parameters, (3) the training dataset file location (in ARFF format), and (4) the binary output file path. Methods `readFile()` and `train()` contain corresponding code snippets already covered in the previous sections of this chapter. Method `writeToFile()` uses method `writeObject(Object)` from class `ObjectOutputStream` to perform the classifier file export.

```
1
2  import java.io.FileOutputStream;
3  import java.io.ObjectOutputStream;
4  import weka.classifiers.Classifier;
5  import weka.core.Instances;
6  import weka.core.converters.ConverterUtils.DataSource;
7
8  public class ClassifierSerializer {
9
10   /** A WEKA object to read an ARFF dataset*/
11   private DataSource source;
12
13   /** The classifier class name*/
14   private String classifierName;
15
16   /** Some parameters each particular classifier could have*/
17   private String classifierParameters;
18
19   /** The input ARFF dataset*/
20   private String dataSourceFile;
21
22   /** The output file path*/
23   private String outputFileName;
24
25   /** A WEKA object to handle a classification algorithm */
26   private Classifier classifier = null;
27
28   /** A WEKA object to represent the dataset*/
29   private String Instances dataset;
30
31   public ClassifierSerializer(String classifierName, String
         classifierParameters,
32     String dataSourceFile, String outputFileName) {
33     this.classifierName = classifierName;
34     this.classifierParameters = classifierParameters;
35     this.dataSourceFile = dataSourceFile;
36     this.outputFileName = outputFileName;
37   }
38
39   /** Outputs the trained classifier to the specified file*/
40   public void serialize() {
41     readFile();
42     train();
43     writeToFile();
44   }
45
46   /** Reads an ARFF file*/
47   private void readFile() {
48
49     /** Create a data source from a given ARFF file */
50     try {
51       source = new DataSource(dataSourceFile);
52     } catch (Exception ex) {
53       source = nulll;
54       ex.printStackTrace();
55     }
56
57     // Read the dataset from source
58     try {
59       dataset = source.getDataSet();
60     } catch (Exception e1) {
61       dataset = null
62       e1.printStackTrace();
63     }
64
```

```
65      /** If the given ARFF file does not specify the class attribute,
            it is
66          automatically set to the very last column */
67
68      if (dataset.classIndex() == -1) {
69        dataset.setClassIndex(dataset.numAttributes() - 1);
70      }
71    }
72
73    /** Trains a classification model using the specified classifierName
          and classifierParameters */
74    private void train() {
75      try {
76        classifier = Classifier.forName(classifierName, weka.core.Utils.
              splitOptions(classifierParameters));
77      } catch (Exception e2) {
78        e2.printStackTrace();
79      }
80
81          try {
82        classifier.buildClassifier(data);
83      } catch (Exception e1) {
84        e1.printStackTrace();
85      }
86    }
87
88    /** This method converts a Classifier object to binary format and
          stores it into a file*/
89    private void writeToFile() {
90
91      // Serialize classifier
92      FileOutputStream fileStream;
93      try {
94        fileStream = new FileOutputStream(outputFileName);
95            ObjectOutputStream objectStream = new ObjectOutputStream(
                  fileStream);
96            objectStream.writeObject(classifier);
97            objectStream.close();
98            fileStream.close();
99            System.out.println(outputFileName + "written to file
                  successfully");
100     } catch (Exception e) {
101       System.out.println(e.getMessage());
102     }
103   }
104 }
```

Listing 14.6: The implementation of class `ClassifierSerializer`.

Listing 14.7 shows an example of the usage of class `ClassifierSerializer` to train and serialize different classifiers with the default parameters. Figure 14.1 shows how to obtain these parameters from the WEKA user interface.

```
1
2       String datasource = "C:\\Development\\android\\VigilanteLibraries
            \\wekaFiles\\HAR-12s-ALL-Filtered.arff";
3
4       // An instance based learner (K-nearest-neighbor classifier) using
            k=1 and the euclidean distance as a metric.
5       serializeClassifier("weka.classifiers.lazy.IBk", "-K 1 -W 0 -A \"
            weka.core.neighboursearch.LinearNNSearch -A \\\"weka.core.
            EuclideanDistance -R first-last\\\"\"", "ibk.data", datasource
            );
```

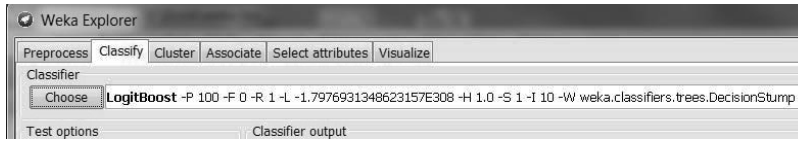

FIGURE 14.1: Default classifier parameters in the WEKA user interface.

```
6
7       // A Sequential Minimal Optimization (SMO) classifier using
            polynomial kernel functions.
8       serializeClassifier("weka.classifiers.functions.SMO", "-C 1.0 -L
            0.0010 -P 1.0E-12 -N 0 -V -1 -W 1 -K \"weka.classifiers.
            functions.supportVector.PolyKernel -C 250007 -E 1.0\"", "smo.
            data", datasource);
9
10      // An Additive Logistic Regression classifier using 10 iterations
            and the DecisionStump classifier as base learner
11      serializeClassifier("weka.classifiers.meta.LogitBoost", "-P 100 -F
             0 -R 1 -L -1.7976931348623157E308 -H 1.0 -S 1 -I 10 -W weka.
            classifiers.trees.DecisionStump", "logit.data", datasource);
12
13      // A Bayesian Network Classifier using the K2 search algorithm.
14      serializeClassifier("weka.classifiers.bayes.BayesNet", "-D -Q weka
            .classifiers.bayes.net.search.local.K2 -- -P 1 -S BAYES -E
            weka.classifiers.bayes.net.estimate.SimpleEstimator -- -A 0.5"
            , "bayesNet.data", datasource);
```

Listing 14.7: The implementation of class `Point`.

14.3.2 Retrieving models in the smartphone

In order to make the classification model available in the smartphone, the exported serialized file should be present in the device. One way to do so is to use a Web server to store the models and allow mobile clients to download them. In this fashion, models could be dynamically upgraded using additional data or more effective techniques and such changes would be reflected on the mobile side. Listing 14.8 depicts method `getClassifier(String)`, which connects to the given URL and de-serializes the classifier using class `ObjectInputStream`. Finally, the read object `obj` is casted to class `Classifier` and returned by the `getClassifier(String)`.

```
1
2  import weka.classifiers.Classifier;
3
4  ...
5
6  public static Classifier getClassifier(String urlPath) {
7      try {
8        String filename;
9        URL url = new URL(urlPath);
10         URLConnection conn = url.openConnection();
11         InputStream fileStream = new BufferedInputStream(conn.
               getInputStream());
```

```
12          ObjectInputStream objectStream =  new ObjectInputStream(
                fileStream);
13          Object obj = objectStream.readObject();
14
15          if (obj instanceof Classifier) {
16            return (Classifier) obj;
17          }
18          return null;
19      } catch (Exception e) {
20        return null;
21      }
22    }
```

Listing 14.8: De-serialization of a classification model.

14.4 Real-time activity recognition

As it was shown in Chapter 5, the sensor interaction and data flow are handled by Vigilante's class `SensorManager` from package `edu.usf.cse.vigilante.sensor`. A partial implementation of this class is presented here, including the most relevant functionalities to support feature extraction and classification. However, the complete code is not included here, as it entails additional components that are beyond the scope of this book. Nonetheless, the provided code will be useful for the reader to prototype a HAR system.

As it was discussed in Chapter 5, every time a new measurement (encapsulated in an instance of class `Packet`) is recorded from any sensor, the sensor manager is notified via the method `update(Object, Object)` from interface `Observer`. Then, the values in the packet are appended to the corresponding time series (e.g., `hrSeries`, `rrSeries`, and so forth), which are maintained as buffers. Since feature extraction should be accomplished in a time window, variable `windowBegTime` is maintained to keep track of the length of the current time series. When an entire time window has passed (i.e., `System.currentTimeMillis() - windowBegTime > WINDOW_LENGTH`), the window is ready to be issued (i.e., extract features and predict the activity). In this case, all series are wrapped up in a time window and the method `issueTimeWindow(TimeWindow)` is invoked. Such a method, in turn, creates a new `FeatureSet(window)`, which internally extracts all features from the given time window. Now, in order to generate a prediction for the user's activity, the feature set, it needs to be converted to an instance of `weka.core.Instance` class via method `toInstance(Instances)`. This method requires a parameter from class `Instances` to specify the attribute names and types. An easy way to achieve this is to let the mobile application read an "empty" ARFF file which only defines the data schema, using the code in Listing 14.2. Finally, the prediction is obtained by invoking

method `classifyInstance(Instance)` which returns the predicted activity as an integer number ranging between 0 and the number of activities minus one.

```
1
2  package edu.usf.cse.vigilante.android.sensor;
3
4  public class SensorManager implements Observer  {
5
6    private long windowBegTime = -1;
7
8    /** A classification model trained in the server*/
9    private Classifier classifier;
10
11   /** The data schema read from an empty ARFF file*/
12   private Instances instanceHeader;
13
14   private TimeSeries hrSeries, rrSeries, stSeries;
15   private TimeWindow window;
16   ...
17
18   public void update(Observable sensor, Object packet) {
19
20     if (packet instanceof BioHarnessGeneralPacket) {
21       heartRate.addPoint(new Point(pc.getTimeStamp(), pc.getHeartRate
             ()));
22       respirationRate.addPoint(new Point(pc.getTimeStamp(), pc.
             getRespirationRate()));
23       skinTemperature.addPoint(new Point(pc.getTimeStamp(), pc.
             getSkinTemperature())));
24     } else if (packet instance of ...) {
25       ...
26     }
27     if (System.currentTimeMillis() - windowBegTime > WINDOW_LENGTH) {
28       if (windowBegTime > 0) {
29         TimeWindow window = new TimeWindow();
30         window.addTimeSeries(heartRate);
31         window.addTimeSeries(respirationRate);
32         window.addTimeSeries(skinTemperature);
33
34         issueTimeWindow(window);
35         resetTimeSeries();
36
37       }
38       windowBegTime = System.currentTimeMillis();
39     }
40   }
41
42   /** Extract features and recognize the activity performed over the
         given window*/
43   public void issueTimeWindow(TimeWindow window) {
44     FeatureSet extractedFeatures = new FeatureSet(window);
45
46     if (classifier != null) {
47       try {
48         Instance instance = extractedFeatures.toInstance(
               instanceHeader);
49         int activityId = (int) classifier.classifyInstance(instance);
50
51         ....
52
53       } catch (Exception e) {
54         e.printStackTrace();
55       }
```

```
56        }
57    }
58
59    ...
60
61    public void initializeClassifier() {
62        instanceHeader = getInstanceHeader();
63        classifier = getClassifier();
64    }
65
66 }
```

Bibliography

[1] 2011 Activity Recognition Challenge. www.oportunity-project.eu/challenge.

[2] Datasets for Human Activity Recognition from ETH's Wearable Computing Lab. http://www.wearable.ethz.ch/resources/Dataset.

[3] Datasets for Human Activity Recognition from MIT Media Lab. http://architecture.mit.edu/house_n/data/PlaceLab/PlaceLab.htm.

[4] Datasets for Human Activity Recognition from Tanzeem Choudhury's website. http://www.cs.dartmouth.edu/~tanzeem/teaching/CS188-Fall08/dataset.html.

[5] Datasets for Human Activity Recognition from Tim Van Kasteren's website. https://sites.google.com/site/tim0306/datasets.

[6] Google Places API. http://code.google.com/apis/maps/documentation/places/.

[7] Google Scholar. http://scholar.google.com/.

[8] Google's Android becomes the world's leading smart phone platform. http://www.canalys.com/newsroom/google%E2%80%99s-android-becomes-world%E2%80%99s-leading-smart-phone-platform.

[9] JAMA: A Java Matrix Package. http://mste.illinois.edu/users/exner/java.f/leastsquares/.

[10] Source code for the Least Square algorithm. http://mste.illinois.edu/users/exner/java.f/leastsquares/.

[11] The Java Data Mining Platform. http://www.jdmp.org/.

[12] The Observer-Observable pattern in Java. http://download.oracle.com/javase/6/docs/api/java/util/Observable.html.

[13] The R Project. http://www.r-project.org/.

[14] The Weka API. http://www.cs.waikato.ac.nz/ml/weka/index_documentation.html.

[15] UCI datasets. http://archive.ics.uci.edu/ml/datasets.html.

[16] Use Weka in your Java code. http://weka.wikispaces.com/Use+WEKA+in+your+Java+code.

[17] Vigilante: A Mobile Platform for Real-time Human Activity Recognition. http://www.youtube.com/watch?v=4Jen6O1lfms.

[18] Waikato environment for knowledge analysis (weka). http://www.cs.waikato.ac.nz/ml/weka.

[19] Zephyr Bioharness BT Website. http://www.zephyr-technology.com/bioharness-bt.html.

[20] M. Ahad, J.K. Tan, H.S. Kim, and S. Ishikawa. Human activity recognition: Various paradigms. In *International Conference on Control, Automation and Systems*, pages 1896–1901, 2008.

[21] A. Ali, R.C. King, and G-Z. Yang. Semi-supervised segmentation for activity recognition with multiple eigenspaces. In *International Summer School and Symposium on Medical Devices and Biosensors*, pages 314–317, 2008.

[22] K. Altun and B. Barshan. Human activity recognition using inertial/-magnetic sensor units. In *Human Behavior Understanding*, Lecture Notes in Computer Science, pages 38–51. Springer Berlin/Heidelberg, 2010.

[23] P. Antal. Construction of a classifier with prior domain knowledge formalised as bayesian network. In *Proceedings of the 24th Annual Conference of the IEEE Industrial Electronics Society*, volume 4, pages 2527–2531, 1998.

[24] D. Arvidsson, F. Slinde, and L. Hulthén. Free-living energy expenditure in children using multi-sensor activity monitors. *Clinical nutrition (Edinburgh, Scotland)*, 28:3, s. 305, 2009.

[25] L. Bao and S. S. Intille. Activity recognition from user-annotated acceleration data. In *Pervasive*, pages 1–17, 2004.

[26] M. Berchtold, M. Budde, D. Gordon, H.R. Schmidtke, and M. Beigl. Actiserv: Activity recognition service for mobile phones. In *International Symposium on Wearable Computers*, pages 1–8, 2010.

[27] M. Berchtold, M. Budde, D. Gordon, H.R. Schmidtke, and M. Beigl. Actiserv: Activity recognition service for mobile phones. pages 1–8, 2010.

[28] M. Berchtold, M. Budde, H. Schmidtke, and M. Beigl. An extensible modular recognition concept that makes activity recognition practical. In *Advances in Artificial Intelligence*, Lecture Notes in Computer Science, pages 400–409. Springer Berlin/Heidelberg, 2010.

[29] U. Blanke and B. Schiele. Remember and transfer what you have learned - recognizing composite activities based on activity spotting. In *International Symposium on Wearable Computers*, pages 1–8, 2010.

[30] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100. ACM, 1998.

[31] T. Brezmes, J. L. Gorricho, and J. Cotrina. Activity recognition from accelerometer data on a mobile phone. In *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, volume 5518, pages 796–799. Springer Berlin/Heidelberg, 2009.

[32] T. Brezmes, J. L. Gorricho, and J. Cotrina. Activity recognition from accelerometer data on a mobile phone. In *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, volume 5518 of *Lecture Notes in Computer Science*, pages 796–799. Springer Berlin/Heidelberg, 2009.

[33] J. Candamo, M. Shreve, D.B. Goldgof, D.B. Sapper, and R. Kasturi. Understanding transit scenes: A survey on human behavior-recognition algorithms. *IEEE Transactions on Intelligent Transportation Systems*, 11(1):206–224, 2010.

[34] P. Cavalin, R. Sabourin, and C. Suen. Dynamic selection approaches for multiple classifier systems. *Neural Computing and Applications*, pages 1–16, 2011.

[35] Y-P. Chen, J-Y. Yang, S-N. Liou, G-Y. Lee, and J-S. Wang. Online classifier construction algorithm for human activity detection using a tri-axial accelerometer. *Applied Mathematics and Computation*, 205(2):849–860, 2008.

[36] J. Cheng, O. Amft, and P. Lukowicz. Active capacitive sensing: Exploring a new wearable sensing modality for activity recognition. In *Pervasive Computing*, volume 6030 of *Lecture Notes in Computer Science*, pages 319–336. Springer Berlin/Heidelberg, 2010.

[37] T. Choudhury, S. Consolvo, B. Harrison, J. Hightower, A. LaMarca, L. LeGrand, A. Rahimi, A. Rea, G. Bordello, B. Hemingway, P. Klasnja, K. Koscher, J.A. Landay, J. Lester, D. Wyatt, and D. Haehnel. The mobile sensing platform: An embedded activity recognition system. *Pervasive Computing, IEEE*, 7(2):32–41, 2008.

[38] D. Choujaa and N. Dulay. Tracme: Temporal activity recognition using mobile phone data. In *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, volume 1, pages 119–126, 2008.

[39] D. Christin, A. Reinhardt, S. S. Kanhere, and M. Hollick. A survey on privacy in mobile participatory sensing applications. *Journal of Systems and Software*, 84(11):1928–1946, 2011.

[40] R. Cilla, M.A. Patricio, A. Berlanga, and J.M. Molina. Creating human activity recognition systems using pareto-based multiobjective optimization. In *Sixth IEEE International Conference on Advanced Video and Signal Based Surveillance*, 2009.

[41] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987.

[42] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995.

[43] G. Dahlquist and A. Bjorck. *Numerical methods in scientific computing. vol 1.* SIAM, 2008.

[44] W. H. E. Day. Consensus methods as tools for data analysis. In *Conference of the International Federation of Classification Societies*, pages 317–324, 1988.

[45] T. G. Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation*, 10:1895–1923, 1998.

[46] R. P. W. Duin. The combining classifier: to train or not to train? In *Proceedings of the 16th International Conference on Pattern Recognition*, volume 2, pages 765–770, 2002.

[47] C. Elkan. The foundations of cost-sensitive learning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 973–978, 2001.

[48] M. Ermes, J. Parkka, and L. Cluitmans. Advancing from offline to online activity recognition with wearable sensors. In *Engineering in Medicine and Biology Society. 30th Annual International Conference of the IEEE*, pages 4451–4454, 2008.

[49] F. Foerster, M. Smeja, and J. Fahrenberg. Detection of posture and motion by accelerometry: a validation study in ambulatory monitoring. *Computers in Human Behavior*, 15(5):571–583, 1999.

[50] K. Frank, M. Rockl, M.J.V. Nadales, P. Robertson, and T. Pfeifer. Comparison of exact static and dynamic bayesian context inference methods for activity recognition. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 189–195, 2010.

[51] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28:2000, 1998.

[52] S.I. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, 1990.

[53] G. Giacinto and F. Roli. Dynamic classifier selection based on multiple classifier behaviour. *Pattern Recognition*, 34:1879–1881, 2001.

[54] K. Goebel and W. Yan. Using correlation-based measures to select classifiers for decision fusion, 2005.

[55] D. Guan, W. Yuan, Y-K. Lee, A. Gavrilov, and S. Lee. Activity recognition based on semi-supervised learning. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 469–475, 2007.

[56] M. A. Hall. *Correlation-based Feature Selection for Machine Learning.* 1999.

[57] Y. Hanai, J. Nishimura, and T. Kuroda. Haar-like filtering for human activity recognition using 3d accelerometer. In *IEEE 13th Digital Signal Processing Workshop and 5th IEEE Signal Processing Education Workshop*, pages 675–678, 2009.

[58] Z. He and L. Jin. Activity recognition from acceleration data based on discrete consine transform and svm. In *IEEE International Conference on Systems, Man and Cybernetics*, pages 5041–5044, 2009.

[59] Z. He, Z. Liu, L. Jin, L-X. Zhen, and J-C. Huang. Weightlessness feature; a novel feature for single tri-axial accelerometer based activity recognition. In *19th International Conference on Pattern Recognition*, pages 1–4, 2008.

[60] Z-Y. He and L-W. Jin. Activity recognition from acceleration data using ar model representation and svm. In *International Conference on Machine Learning and Cybernetics*, volume 4, pages 2245–2250, 2008.

[61] R. Helaoui, M. Niepert, and H. Stuckenschmidt. Recognizing interleaved and concurrent activities: A statistical-relational approach. In *IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 1–9, 2011.

[62] T. K. Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, Aug 1998.

[63] K. L. Huang, S. S. Kanhere, and W. Hu. Preserving privacy in participatory sensing systems. *Computer Communications*, 33(11):1266–1280, 2010.

[64] T. Huynh and B. Schiele. Towards less supervision in activity recognition from wearable sensors. In *10th IEEE International Symposium on Wearable Computers*, pages 3–10, 2006.

[65] J. Iglesias, J. Cano, A. M. Bernardos, and J. R. Casar. A ubiquitous activity-monitor to prevent sedentariness. In *IEEE PerCom*, pages 668–670, 2011.

[66] L. C. Jatoba, U. Grossmann, C. Kunze, J. Ottenbacher, and W. Stork. Context-aware mobile health monitoring: Evaluation of different pattern recognition methods for classification of physical activity. In *30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 5250–5253, 2008.

[67] Y. Jia. Diatetic and exercise therapy against diabetes mellitus. In *Second International Conference on Intelligent Networks and Intelligent Systems*, pages 693–696, 2009.

[68] C. N. Joseph, S. Kokulakumaran, K. Srijeyanthan, A. Thusyanthan, C. Gunasekara, and C.D. Gamage. A framework for whole-body gesture recognition from video feeds. In *International Conference on Industrial and Information Systems (ICIIS)*, pages 430–435, 2010.

[69] T-P. Kao, C-W. Lin, and J-S. Wang. Development of a portable activity detector for daily activity recognition. In *IEEE International Symposium on Industrial Electronics*, pages 115–120, 2009.

[70] A. M. Khan, Y. K. Lee, and S. Y. Lee. Accelerometer's position free human activity recognition using a hierarchical recognition model. In *12th IEEE International Conference on e-Health Networking Applications and Services (Healthcom)*, pages 296–301, 2010.

[71] A.M. Khan, Y-K. Lee, S.Y. Lee, and T-S. Kim. A triaxial accelerometer-based physical-activity recognition via augmented-signal features and a hierarchical recognizer. *IEEE Transactions on Information Technology in Biomedicine*, 14(5):1166–1172, 2010.

[72] A.M. Khan, Y.K. Lee, and S.Y. Lee. Accelerometer's position free human activity recognition using a hierarchical recognition model. In *IEEE International Conference on e-Health Networking Applications and Services (Healthcom)*, pages 296–301, 2010.

[73] I. Koutis, G.L. Miller, and R. Peng. Approaching optimality for solving sdd linear systems. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 235 –244, Oct. 2010.

[74] L. I. Kuncheva. *Combining pattern classifiers*. John Wiley and Sons publishers, 2004.

[75] N.D. Lane, Ye Xu, Hong Lu, A.T. Campbell, T. Choudhury, and S.B. Eisenman. Exploiting social networks for large-scale human behavior modeling. *Pervasive Computing, IEEE*, 10(4):45–53, 2011.

[76] O. D. Lara and M. A. Labrador. A mobile human activity recognition system. In *IEEE Consumer Communications and Networking Conference (CCNC)*, pages 38 –39, January 2012.

[77] O. D. Lara and M. A. Labrador. A mobile platform for real-time human activity recognition. In *IEEE Consumer Communications and Networking Conference (CCNC)*, pages 667 –671, January 2012.

[78] O. D. Lara and M. A. Labrador. A survey on human activity recognition using wearable sensors. *IEEE Communications Surveys and Tutorials*, (to appear), 2012.

[79] O. D. Lara, A. J. Perez, M. A. Labrador, and J. D. Posada. Centinela: A human activity recognition system based on acceleration and vital sign data. *Pervasive and Mobile Computing*, 8:717–729, 2012.

[80] B. Longstaff, S. Reddy, and D. Estrin. Improving activity classification for health applications on mobile devices using active and semi-supervised learning. In *Pervasive Computing Technologies for Healthcare (PervasiveHealth)*, pages 1–7, 2010.

[81] U. Maurer, A. Smailagic, D. Siewiorek, and M. Deisher. Activity recognition and monitoring using multiple sensors on different body positions. In *Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*, Washington, DC, USA, 2006. IEEE Computer Society.

[82] U. Maurer, A. Smailagic, D. Siewiorek, and M. Deisher. Activity recognition and monitoring using multiple sensors on different body positions. In *Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*, pages 113–116, 2006.

[83] D. McGlynn and M. G. Madden. An ensemble dynamic time warping classifier with application to activity recognition. In *Research and Development in Intelligent Systems XXVII*, pages 339–352. Springer London, 2011.

[84] D. Minnen, T. Westeyn, D.l Ashbrook, P. Presti, and T. Starner. Recognizing soldier activities in the field. In *4th International Workshop on Wearable and Implantable Body Sensor Networks*, volume 13, pages 236–241. Springer Berlin/Heidelberg, 2007.

[85] D. Nettleton, A. Orriols-Puig, and A. Fornells. A study of the effect of different types of noise on the precision of supervised learning techniques. *Artificial Intelligence Review*, 33:275–306.

[86] K. Oh, H-S. Park, and S-B. Cho. A mobile context sharing system using activity and emotion recognition with bayesian networks. In *International Conference on Ubiquitous Intelligence Computing*, pages 244–249, 2010.

[87] R. T. Olszewski, C. Faloutsos, and D. Banks Dot. *Generalized Feature Extraction for Structural Pattern Recognition in Time-Series Data*. 2001.

[88] J. Parkka, M. Ermes, P. Korpipaa, J. Mantyjarvi, J. Peltola, and I. Korhonen. Activity classification using realistic data from wearable sensors. *IEEE Transactions on Information Technology in Biomedicine*, 10(1):119–128, 2006.

[89] H. Peng, F. Long, and C. Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(8):1226–1238, 2005.

[90] J. Penttil, J. Peltola, and T. Seppnen. A speech/music discriminator based audio browser with a degree of certanity measure. In *Proceedings of the International Workshop Information Retrieval*, page 125131, 2001.

[91] A. J. Perez, M. A. Labrador, and S. J. Barbeau. G-sense: A scalable architecture for global sensing and monitoring. *IEEE Network*, 24(4):57–64, 2010.

[92] N. Pham and T. Abdelzaher. Robust dynamic human activity recognition based on relative energy allocation. In *Distributed Computing in Sensor Systems*, volume 5067 of *Lecture Notes in Computer Science*, pages 525–530. Springer Berlin/Heidelberg, 2008.

[93] J.R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann series in machine learning. Morgan Kaufmann Publishers, 1993.

[94] C. Randell and H. Muller. Context awareness by analysing accelerometer data. In *The Fourth International Symposium on Wearable Computers*, pages 175–176, 2000.

[95] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava. Using mobile phones to determine transportation modes. *ACM Transactions on Sensor Networks*, 6(2):1–27, 2010.

[96] D. Riboni and C. Bettini. Cosar: hybrid reasoning for context-aware activity recognition. *Personal and Ubiquitous Computing*, 15:271–289, 2011.

[97] A. Santana, R. Soares, A. Canuto, and M. de Souto. A dynamic classifier selection method to build ensembles using accuracy and diversity. In *SBRN. Ninth Brazilian Symposium on Neural Networks*, pages 36–41, Oct. 2006.

[98] J. Sarkar, L. Vinh, Y-K. Lee, and S. Lee. Gpars: a general-purpose activity recognition system. *Applied Intelligence*, pages 1–18, 2010.

[99] O. Schlmilch, B. Witzschel, M. Cantor, E. Kahl, R. Mehmke, and C. Runge. Detection of posture and motion by accelerometry: a validation study in ambulatory monitoring. *Computers in Human Behavior*, 15(5):571–583, 1999.

[100] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. Real-time human pose recognition in parts from single depth images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2011.

[101] P. C. Smits. Multiple classifier systems for supervised remote sensing image classification based on dynamic classifier selection. *Geoscience and Remote Sensing, IEEE Transactions on*, 40(4):801–813, Apr. 2002.

[102] M. Stikic, D. Larlus, S. Ebert, and B. Schiele. Weakly supervised recognition of daily life activities with wearable sensors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(12):2521–2537, 2011.

[103] M. Stikic, D. Larlus, and B. Schiele. Multi-graph based semi-supervised learning for activity recognition. In *International Symposium on Wearable Computers*, pages 85–92, 2009.

[104] E. Munguia Tapia, S. S. Intille, W. Haskell, K. Larson, J. Wright, A. King, and R. Friedman. Real-time recognition of physical activities and their intensities using wireless accelerometers and a heart monitor. In *Proceedings of the International Symposium on Wearable Computers*, 2007.

[105] A. Teller. A platform for wearable physiological computing. *Interacting with Computers*, pages 917–937, 2004.

[106] A. Tolstikov, X. Hong, J. Biswas, C. Nugent, L. Chen, and G. Parente. Comparison of fusion methods based on dst and dbn in human activity recognition. *Journal of Control Theory and Applications*, 9:18–27, 2011.

[107] P. Turaga, R. Chellappa, V.S. Subrahmanian, and O. Udrea. Machine recognition of human activities: A survey. *IEEE Transactions on Circuits and Systems for Video Technology*, 18(11):1473–1488, 2008.

[108] T. van Kasteren, G. Englebienne, and B. Krse. An activity monitoring system for elderly care using generative and discriminative models. *Journal on Personal and Ubiquitous Computing*, 2010.

[109] I. J. Vergara-Laurens and M. A. Labrador. Preserving privacy while reducing power consumption and information loss in lbs and participatory sensing applications. In *Proceedings of IEEE GLOBECOM*, 2011.

[110] La Vinh, Sungyoung Lee, Hung Le, Hung Ngo, Hyoung Kim, Manhyung Han, and Young-Koo Lee. Semi-markov conditional random fields for accelerometer-based activity recognition. *Applied Intelligence*, 35:226–241, 2011.

[111] I. H. Witten and E. Frank. *Data Mining, Practical Machine Learning Tools and Techniques*. Elsevier, 2nd. edition, 2005.

[112] K. Woods, Jr. Kegelmeyer, W.P., and K. Bowyer. Combination of multiple classifiers using local accuracy estimates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):405 –410, Apr. 1997.

[113] J. Yang, J. Lee, and J. Choi. Activity recognition based on rfid object usage for smart mobile devices. *Journal of Computer Science and Technology*, 26:239–246, 2011.

[114] J. Yin, Q. Yang, and J. J. Pan. Sensor-based abnormal human-activity detection. *IEEE Transactions on Knowledge and Data Engineering*, 20(8):1082–1090, 2008.

[115] H. Zhang. The Optimality of Naive Bayes. In *FLAIRS Conference*. AAAI Press, 2004.

[116] C. Zhu and W. Sheng. Human daily activity recognition in robot-assisted living using multi-sensor fusion. In *IEEE International Conference on Robotics and Automation*, pages 2154–2159, 2009.

[117] X. Zhu, X. Wu, and Y. Yang. Dynamic classifier selection for effective mining from noisy data streams. *Data Mining, IEEE International Conference on*, 0:305–312, 2004.

[118] X. Zhu, X. Wu, and Y. Yang. Effective classification of noisy data streams with attribute-oriented dynamic classifier selection. *Knowl. Inf. Syst.*, 9(3):339–363, March 2006.

Computer Science/Computer Engineering/Computing

Chapman & Hall/CRC Computer & Information Science Series

The pervasiveness and range of capabilities of today's mobile devices have enabled a wide spectrum of mobile applications, from smartphones equipped with GPS to integrated mobile sensors that acquire physiological data, that are transforming our daily lives. **Human Activity Recognition: Using Wearable Sensors and Smartphones** focuses on the automatic identification of human activities from pervasive wearable sensors—a crucial component for health monitoring and also applicable to other areas, such as entertainment and tactical operations.

Developed from the authors' nearly four years of rigorous research in the field, the book covers the theory, fundamentals, and applications of human activity recognition (HAR). The authors examine how machine learning and pattern recognition tools help determine a user's activity during a certain period of time. They propose two systems for performing HAR: Centinela, an offline server-oriented HAR system, and Vigilante, a completely mobile real-time activity recognition system. The book also provides a practical guide to the development of activity recognition applications in the Android framework.

Features

- Describes the potential of HAR in application areas, including health monitoring, entertainment, and tactical operations
- Presents the fundamentals of feature extraction and machine learning tools
- Looks at the design of HAR systems from a research point of view
- Shows how to perform real-time HAR
- Offers a hands-on guide to developing mobile HAR applications for an Android smartphone
- Explores future research considerations

K20375

ISBN: 978-1-4665-8827-1

90000

9 781466 588271

CRC Press
Taylor & Francis Group
an informa business
www.crcpress.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
711 Third Avenue
New York, NY 10017
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK