Activity Learning from Sensor Data

Diane J. Cook

Chapter 2

Activities

2.1. Definitions

Activity learning is an important concept because it critical for understanding human behavior as well as designing human-centric technologies. Because activity learning projects tend to focus on a specific subset of activity patterns and use a subset of available sensor types, it can be difficult to reach an agreement about the expectations of activity learning and how to compare alternative approaches. Our goal is to provide a unifying treatment of activity learning. As a beginning, we provide definitions for sensor events and activities that we will use as the basis of our discussions throughout the rest of the book.

Sensor events. As Figure 1.1 illustrates, an activity learner receives data from sensors that are used to perceive the state of an individual and/or an environment. We specify input data to an activity learner as a sequence of sensor events. Each sensor event, e, takes the form $e = <t, s, m>$ where t denotes a timestamp, s denotes a sensor ID, and m denotes the sensor message. We define an *activity instance* or *activity occurrence* as a sequence of n sensor events $<e_1\ e_2\ ..\ e_n>$. An activity represents the collection of all of its instances. An activity learner may represent an activity as an abstraction of this collection of activity instances.

Types of activities. The terms "action" and "activity" are frequently used interchangeably in activity learning research. In actuality, there is a tremendous diversity of concepts that are classified as activities in the literature. The activity classes that are investigated differ in terms of activity complexity, the type of sensor modality that is typically used to capture the activity, and the computational techniques that are most effective for learning the activity. Figure 2.1 illustrates some of the activities that can be represented and learned using techniques described in this book. While Sitting, Standing, Waving, and Walking appear at one end of the spectrum of activities, the other end consists of longer, more complicated tasks such as Cooking, Marching in Formation, and Playing a Football Game.

Throughout this book, we refer to *action* as a simple ambulatory behavior executed by a single person and typically lasting short durations of time. Similarly, we refer to *interaction* as a short, single-movement action that involves multiple individuals. In contrast, by *activity* we refer to complex behaviors consisting of a sequence of actions which can be performed by a single individual or several individuals interacting with each other. They are typically characterized by longer temporal durations. At the other end of the spectrum are individual states such as human (or animal) postures and poses, environmental state, and object location. While such states are indicative of human behavior, they occur at a single point in time and can be differentiated from actions and activities in this way. Note that these definitions are hierarchical. An action may consist of a sequence of states, while an activity may be contain any number of actions. Activities may also be described in terms of environment states or influences that the environment has on the individual performing the activity. Figure 2.2 represents the relationship between these subsets of human behavior.

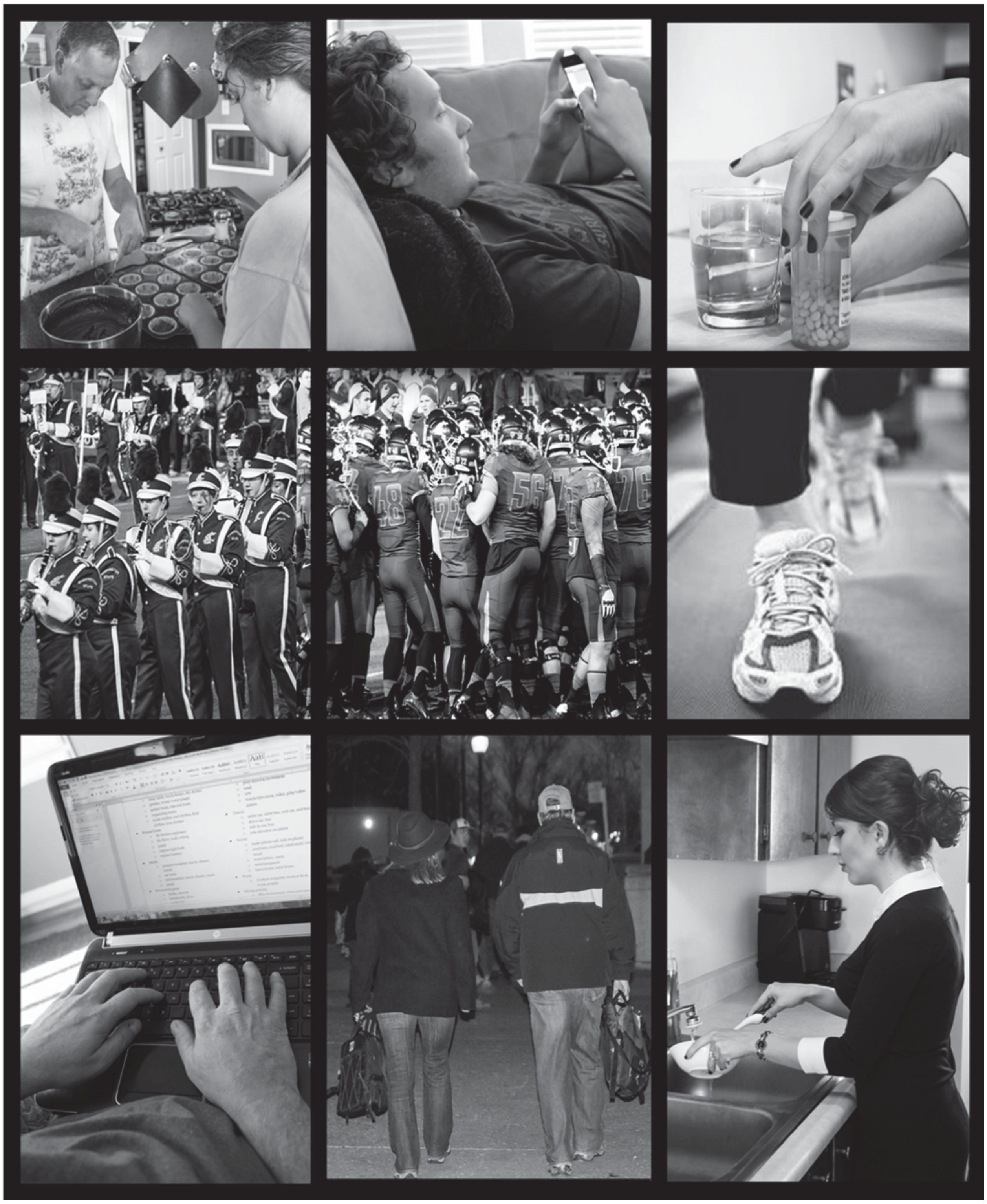

Figure 2.1. Examples of individual group actions and activities that are found in common everyday life settings.

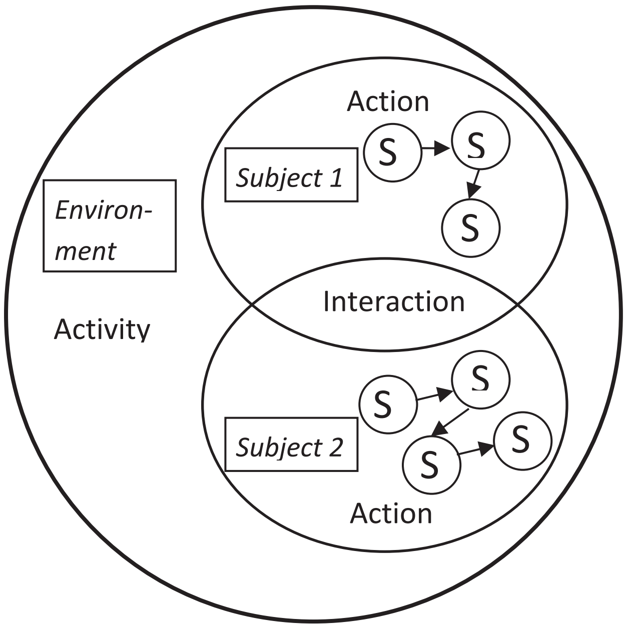

Figure 2.2. Relationship between state (S), action, and activity.

2.2. Classes of Activities

While activity sensor-based datasets are currently available for analysis, there is no single dictionary of activity classes that is accepted and used by all. We provide one possible taxonomy of activity classes in Table 2.1. As can be seen in this list, there exist implicit and explicit relationships between activity classes. A model that can be used to represent the activity Cooking may also be used to represent the subset Cooking Breakfast. On the other hand, a Cooking Breakfast model may not be an effective representative of all cooking tasks. Additionally, some activity classes may be considered functionally different and yet bear a striking resemblance in terms of the sensor events they generate. For example, an individual may fill a glass with water to drink or fill a kettle with water to boil. Such similarities between activities means that additional contextual information, including time of day and the previous activity, need to be considered when learning the activity class. Depending on the sensor modality, some pairs of classes may be virtually indistinguishable. For example, the movements corresponding to lying down on a rug and falling down may generate identical sensor sequences if only infrared motion sensors are used. In these cases, the activities can only be distinguished by incorporating additional sources of information such as different sensor modalities.

Table 2.1. Categories of routine activities.

Actions

- Walk, run, cycle, jump, sit down, stand up, bend over, throw, dig, kneel, skip
- Lie down, fall down, kneel down, faint
- Ascend stairs, descend stairs
- Open door, open cabinet, pick up item, push item, pull item, carry item, throw item
- Point, wave, talk, make fist, clap, gesture
- Chew, speak, swallow, yawn, nod
- Interactions
  - shake hands, hug, kiss, hit, chase, wrestle, high five
  - talk to someone, hand item to someone, throw item to someone

Activities

- Clean house
  - dust, vacuum, sweep, mop
  - make bed, change sheets
  - scrub floor, toilet, surface, windows, ceiling fans
  - clear table, wash dishes, dry dishes
  - garden, weed, water plants
  - gather trash, take out trash
  - organizing items
  - wash clothes, sort clothes, fold clothes, iron clothes
- Repair home
  - fix broken appliance
  - fix floor, wall, ceiling
  - paint
  - replace light bulb
  - replace battery
- Meals
  - prepare breakfast, lunch, dinner, snack
  - set table
  - eat breakfast, lunch, dinner, snack
  - drink
- Personal hygiene
  - bathe, shower
  - brush teeth, floss
  - comb hair
  - select outfit, dress
  - groom
  - shave, wash face, wash hands
  - toilet
  - trim nails, trim hair
- Health maintenance
  - take medicine, fill medicine dispenser, apply medicine
- Sleep
  - nighttime sleep
  - sleep out of bed
- Pet care
  - feed, water, groom
  - walk, play, train
  - clean pet home
- Exercise
  - lift weights
  - use treadmill, elliptical, cycle, rower
  - calisthenics
  - stretch
  - martial arts
  - dive, golf, swim, skate
- Leisure
  - play musical instrument
  - read
  - sew
  - watch television, video, play video games
- Travel
  - enter car, enter bus, exit car, exit bus
  - drive car, bus
  - ride in car, bus
  - ride elevator, escalator
- Social
  - make phone call, talk on phone
  - send text, read text, send email, read email
  - write letters, cards
  - entertain guests
  - leave home, enter home
- Work
  - work at computer, work at desk, work at table
- Group activity
  - Play board game, play card game
  - Play sport against opponent
  - Play sport with team
  - Gather crowd, disperse crowd, move crowd

2.3. Additional Reading

A number of authors have considered ways to categorize classes of activities for modeling, detecting, and recognizing. Candamo et al.[2] and Chaquet et al.[3] distinguish single person activities from multiple person activities and interactions. Borges et al.[4] offer a characterization that breaks activities into gestures, actions, and interactions in a manner similar to the discussion found in this chapter. Krishnan et al.[5] consider methods of learning activity taxonomies directly from sensor data. Chen et al.[6] and Bae[7] employ a method for activity learning that involves hand-constructing an activity taxonomy then using occurrences of the activities to refine the taxonomy. These learned activity ontologies can form a knowledge-driven approach to learning and reasoning about activities[8,9].

In this book, we will refer to examples activities that appear in Figure 2.1 and Table 2.1. However, this is by no means a comprehensive list nor is it the only taxonomical organization of activity classes. Other taxonomies have been generated for use in specific domains. For example, clinicians may evaluate an individual's cognitive and physical health based on their ability to independently complete known activities. As a result, in the health literature activities classes of activities are often categorized as Activities of Daily Living (ADLs)[10] or instrumental Activities of Daily Living (iADLs)[11]. Many resources are available that provide labels, categories, and examples of classes of activities. This includes work by Tapia[12], the Compendium of Physical Activities[13], the American time use survey[14], the Facial Action Coding System[15], the SCARE joint task corpus[16], and the UCF sports dataset[17].

Chapter 5

Activity Recognition

The field of activity recognition is concerned with the question of how to label activities from a sensor-based perception of the environment. The problem of activity recognition is to map a sequence of sensor events, $x=<e_1\ e_2\ ..\ e_n>$, onto a value from a set of predefined activity labels, $a \in A$. Activity recognition can be viewed as a type of supervised machine learning problem. An Activity Recognition (AR) algorithm learns a function that maps a feature vector, X, describing a particular sensor event sequence onto an activity label, $h:X \rightarrow A$. In this supervised machine learning problem the classes are the activity labels and the sensor events are represented by features combined into the input vector X of D dimensions. AR can use the learned function to recognize, or label, occurrences of the learned activity.

Activity recognition faces challenges that make it unique among supervised machine learning problems. The sequential nature of the input data, the ambiguous partitioning of data into distinct data points, and the common overlapping of activity classes mean that additional data processing must be performed in order to accomplish the goal of recognizing activities. As Figure 5.1 shows, the steps involved in activity recognition include collecting and preprocessing sensor data, then dividing it into subsequences that are converted to feature representations. The final feature vectors are either labeled by an oracle and provided as training data to learn an activity model or are sent to an already-trained classifier to identify the activity label. In Chapter 3 we described the data collection and preprocessing steps and discussed how raw sensor data can be represented as a set of high-level features. In Chapter 4 we described methods for supervised learning that map such a feature vector, X, onto class values.

In this chapter, we describe the remaining steps involved in the activity recognition process. When humans perform activities, they do so fluidly, in such a way that consecutive activities blur into each other rather than being clearly delineated with sufficiently-large gaps between activities. They can also perform several activities in parallel or interweave them for maximum efficiency. As a result, the continuous stream of sensor-based information needs to be separated into pieces that can more clearly be distinguished by a trained classifier. Several approaches can be taken to address this issue of sensor event segmentation for activity recognition. We introduce some of these methods, describe how they fit into the activity recognition process, and discuss the related problem of activity spotting. We also overview methods for evaluating the performance of the resulting activity recognition algorithms.

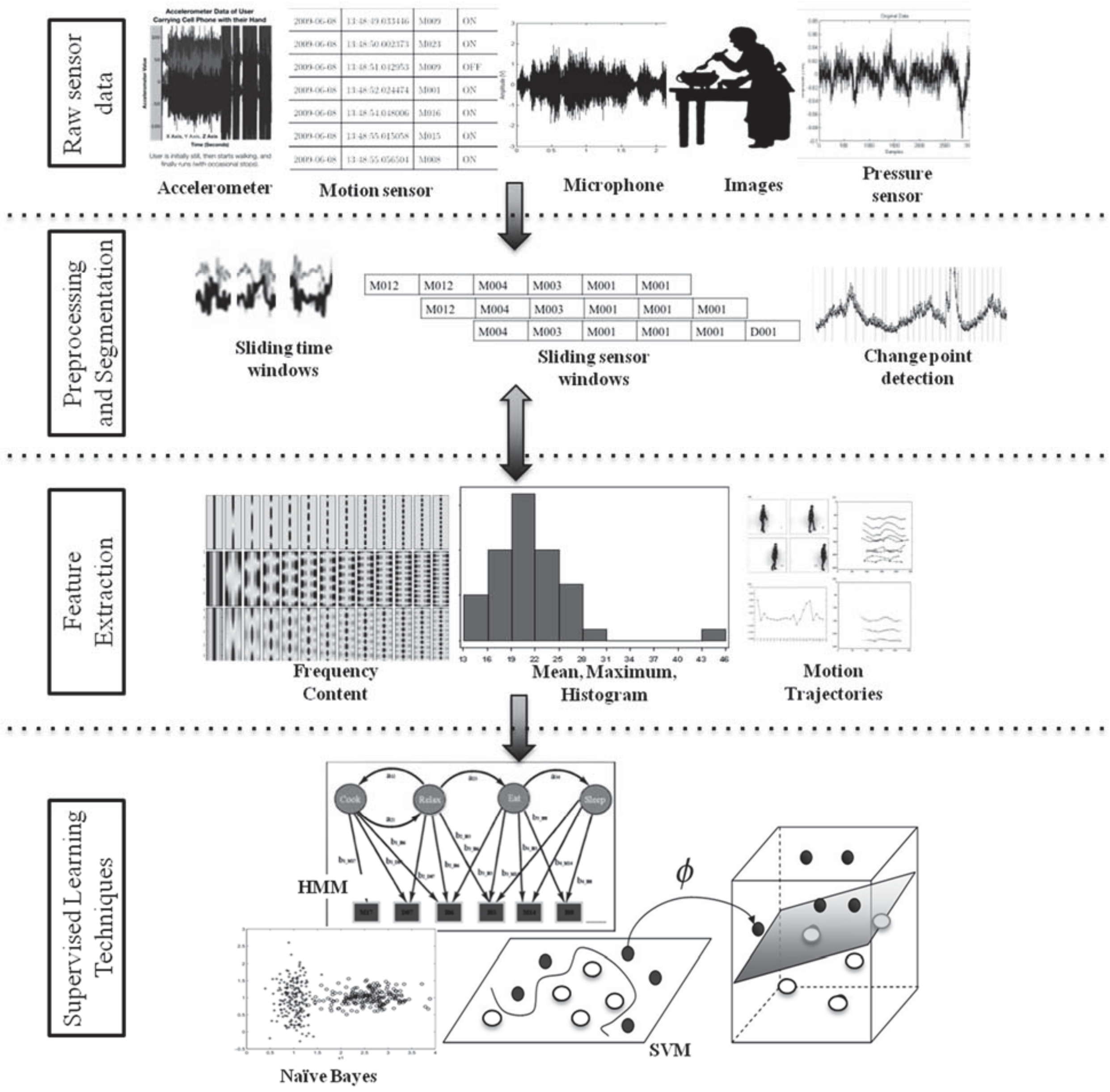

Figure 5.1. The activity recognition process includes the stages of raw sensor data collection, sensor preprocessing and segmentation, feature extraction and selection, classifier training and data classification.

5.1 Activity Segmentation

An activity recognition algorithm, AR, needs to recognize activities at the time they occur in unscripted settings at the time that an individual (or a group) performs the activity. To achieve this goal, AR must map the state of the user and the corresponding environment at the current point in time to an activity label. The first task then is to select the sensor events that comprise the current activity and define the corresponding context. The choice of segmentation or subsequence selection technique is critical because no classifier can produce meaningful results if the input features do not have discriminatory power. We consider two alternative approaches for making this selection: event segmentation and window sliding.

Event segmentation is a two-step process. In the first step, the sequence of streaming sensor events is separated into non-overlapping subsequences, or partitions. Each subsequence should be internally homogeneous and represent a single activity. AR can map each separate sequence to a corresponding activity label. The result of segmenting the data is a complete partitioning of the sensor events, as formalized in Definition 5.1.

Definition 5.1. Given a sequence of *n* sensor events $S = \langle e_1..e_n \rangle$, an *event segmentation* partitions the sequence into a set of *x* subsequences $P = \langle S_1,..,S_x \rangle$, such that each $S_i \subseteq S$ and the order of the elements in S_i is preserved. In addition, the set of subsequences is non-empty, non-overlapping, and $\bigcup_{i=1}^{i=x} S_i = S$.

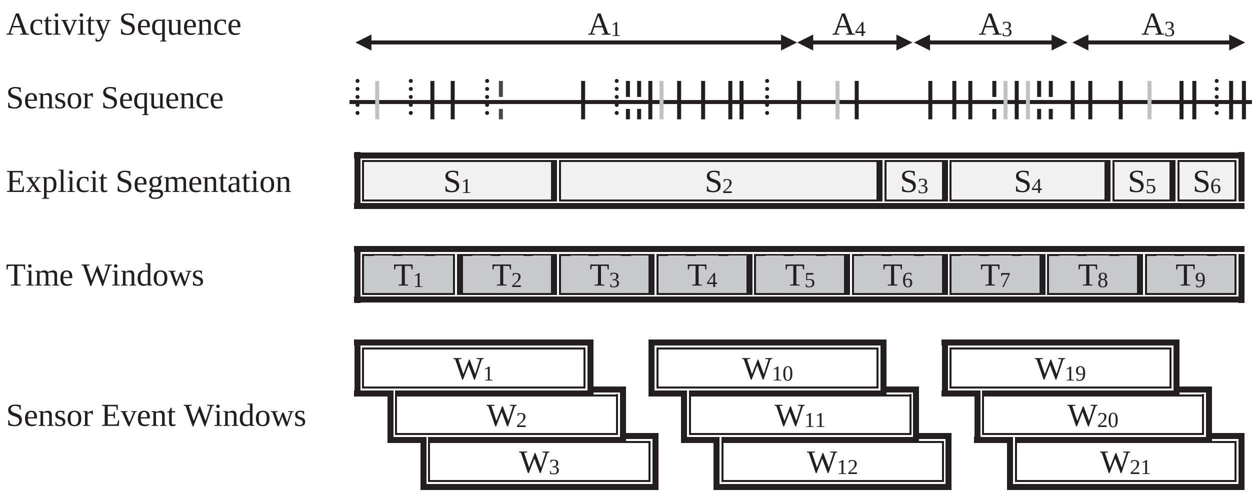

Figure 5.2. Illustration of alternative approaches to processing sensor data streams for activity recognition. Sensor event types and timings are depicted by the vertical lines, where the line type indicates the type, location, and value of the sensor message. The sensor windows are obtained using explicit segmentation, sliding window extraction with a fixed window time duration, or sliding window extraction with a fixed window length (number of sensor events).

Activity segmentation thus refers to the problem of segmenting a continuous stream of data into discrete meaningful units of activity execution. Each of these pieces, or segments, can be classified as an activity. For example, in Figure 5.2 the input sequence of sensor events is partitioned into the non-overlapping subsequences S_1.. S_6, each of which can then be mapped to an activity label. These subsequences represent activity segments and are defined by their corresponding start and end times in the sensor sequence. As we can see in Figure 5.1, the segments may not always align perfectly with activity boundaries and this needs to be take into consideration when evaluating the entire activity recognition process.

There are two classes of approaches that are common for event segmentation. The first relies on supervised machine learning, in which sample data is provided along with ground truth labels from which appropriate segment boundaries can be learned. The second utilizes features of the data itself, without supervised guidance, to identify activity boundaries in the event sequence data.

The first approach to sensor data segmentation that we will discuss relies on information about the activities that are modeled and are known to occur in the data sequence. Classifier-based and rule-based approaches can be designed to identify activity boundaries based on the known activity information.

Classifier-based segmentation. When a supervised approach is employed for activity segmentation, machine learning algorithms are trained to recognize activity beginnings and endings (activity boundaries or breakpoints) or changes between activities (transitions) based on labeled examples. The sequence of sensor events between the activity beginning and ending, or between detected transitions, is partitioned into a separate sequence and used for activity labeling. Here we describe types of data samples that could be used to train a classifier to recognize activity boundaries.

- Examples of activity starts and stops. To employ this approach, a sufficient number of start and stop instances must be provided for each activity class. Thus for a set of A possible activities, $2A$ models need to be learned.

- Examples of activity transitions. The switches from one activity to another, or activity transitions, themselves can be explicitly modeled. The input to this type of learning problem consists of sensor events at the end of one activity combined with sensor events at the beginning of the next together with any events that occur between the two activities. In order to use this learning approach to activity segmentation, a model needs to be learned for every pair of activities, resulting in A^2 models. In addition, this approach runs into problems when sensor events may be observed that do not correspond to a known activity. In such situations, the transition may not be recognized because the individual is transitioning to or from an unknown activity. In other cases, the transition itself may be a separate activity that recurs and could be modeled. For example, transitioning to a Leave Home activity may itself comprise several steps such as collecting car keys, gathering supplies for an outing, and turning out lights. This could be labeled as Preparing to Leave Home and may be a predictable behavior that is itself recognizable. In these situations the transition data could actually contain more sensor data then the activities that precede and follow the transition.

Another alternative is to treat transition recognition as a binary class problem, where the set of all possible transitions represents one class and all within-activity sequences represents the second class. While only one model needs to be learned in this case, this is a much more complex problem that will be difficult to learn if the number of possible activities is large.

- Examples of activity occurrences. The idea here is that if each activity is modeled and a data point does not sufficiently follow any of the learned activity patterns, then it must represent a transition between the known activities. A one-class classifier can be trained on each known activity and used to “reject” points that do not fit into the known activities, in which case they represent transitions. A one-class classifier distinguishes examples of the target class from all other possible data points. With this method, A unique models must be learned in total. Alternatively, the sequence can be rejected if none of the activity models recognize the sequence with sufficient confidence. As with the activity transition method, a situation may occur where not all possible activities are modeled. In this situation, the rejected data points may represent other unknown activities and not simply transitions between known activities.

```
Algorithm RuleBasedSegment(S, MSA)
  // S is an input sequence of sensor events <$e_1$ $e_2$ .. $e_n$>
  // MSA contains the set of activities associated with each sensor event type or value range
  Boundaries = {}
  begin = 1
  Boundaries.append(begin)    // Indicate the beginning of the sequence is an activity boundary

  i = 1
  while i<n do
     i = i + 1
     $SA_{begin}$ = MSA($e_{begin}$)          // Find the activities associated with the boundary sensor event
     $SA_i$ = MSA($e_i$)                  // Find the activities associated with the current sensor event
     if $SA_{begin} \cap SA_i = \varnothing$
        end = i - 1
        Boundaries.append(end)
        begin = i
        Boundaries.append(begin)
     end if
  done
  Boundaries.append(i)           // Indicate the end of the sequence is an activity boundary
  return Boundaries
```

Figure 5.3. Rule-based activity segmentation algorithm.

Rule-based segmentation. An alternative method to locate activity boundaries is to identify points in the sensor event sequence when the data is better supported by two or more activities than by a single contiguous activity (which indicates a change of activities at that point in the sequence). A common method to detect this change is to construct rules that look for particular types of change in the sensor events. This approach is based on the assumption that activities tend to be clustered around particular types of sensor events or other easily-detectable sensor features such as locations, times, or objects. From the perspective of sensor events, this means that the same types of sensor readings will occur each time a particular activity is performed. For each sensor type (in the case of discrete event sensors), the set of associated activities can be determined and stored. In the case of sampling-based sensors, the sensor values can be partitioned into ranges that occur when the activity is performed. The association of activities for each sensor event type or value range can be determined from domain knowledge or can be determined based on sample data.

As summarized in Figure 5.3, the sensor-activity mappings, stored in the vector *MSA*, can be used to detect activity boundaries. When neighboring sensor events $<e_{i-1}, e_i>$ do not have any associated activities in common, this indicates that an activity boundary has been reached and a transition is occurring. In this case, event e_{i-1} is marked as the end of one activity and event e_i is marked as the beginning of the next activity. In addition to determining the activities associated with sensor events, other feature similarities (such as time of day) can be determined as well and used to identify activity boundaries.

The rule-based segmentation algorithm provides a fairly conservative approach to activity boundary detection for segmentation. Consider the sensor event streams provided in Appendix A. Notice that there are some sensor types that are associated with both of the activities. For example, the sensor "M017" appears in both the Hand Washing and the Sweeping activities. In contrast, "WATER" appears only in the Hand Washing activity and "BURNER" appears only in the Sweeping activity. The MSAs for these sensors are thus MSA(M017) = {HandWashing, Sweeping}, MSA(WATER) = {HandWashing}, and MSA(BURNER) = {Sweeping}. If the following sensor sequence appears in the data stream:

10:00:00.00000	M017	ON
10:00:00.00000	M017	ON
10:01:00.00000	M017	ON
10:02:00.00000	WATER	ON
10:03:00.00000	BURNER	ON
10:04:00.00000	M017	ON
10:05:00.00000	M017	ON
10:06:00.00000	M017	ON

then a boundary will be detected between the WATER and BURNER events because their associated activities do not overlap. Note that some false negatives can result from this approach. If a M017 event occurred between the WATER and BURNER entries then no boundary would be detected because M017 belongs to both activities. This problem can be addressed by considering the probabilities of relationships between sensor events and activities as well as considering a subsequence of events before or after the candidate activity boundary.

5.2 Sliding Windows

A second approach to handling streaming data is to divide the entire sequence of sensor events into a set of time ordered, possibly-overlapping subsequences, or sliding windows. The windows follow the formalism given in Definition 5.2.

Definition 5.2. Given a sequence of *n* sensor events $S = \langle e_1..e_n \rangle$, *event windowing* identifies a set of *x* windows, $P = \langle S_1,..,S_x \rangle$, with window sizes $\{w_1,..,w_n\}$, such that each S*i* is an ordered subsequence of *S*. The set of windows is ordered, non-empty, possibly overlapping, and $\bigcup_{i=x}^{i=x} S_i \supseteq S$. Window *S*ᵢ can thus be represented by the sequence $< e_i, e_{i+w_i} >$.

Using a sliding window algorithm, the last (most recent) sensor event in each window S_i is mapped to an activity label by an activity recognition algorithm based on the learned mapping $h : S \rightarrow A$. The sequence of sensor events in the window provides a context for making an informed mapping. Because the windows are ordered, each window can be mapped to an activity label as it occurs, which makes this a valuable approach when labeling activities in real time from streaming data. This technique offers a simpler approach to learn the activity models during the training phase over the explicit segmentation approach. Furthermore, it reduces the computational complexity of activity recognition over the explicit segmentation process. This AR technique can also be used to facilitate *activity spotting*, which is the process of locating instances of a particular activity from a continuous data stream in which the activity is mixed with background noise and irrelevant actions.

There still remains a number of decisions to make with a sliding windowing approach. First, window sizes need to be determined based on the appropriateness for the context and the type of activities that will be recognized. Second, events may need to be weighted within a window based on their relevance to the current context.

5.2.1. Time based windowing

One approach to handling sliding windows is to divide the entire sequence of sensor events into equal-size time intervals as illustrated in Figure 5.1 by the subsequences denoted as T_1, T_2, .. T_9. This is referred to as *bursty*, or *timestamp-based sliding windows*. The bursty approach is valuable when data arrives asynchronously, as occurs with discrete-event sensors or when external events are included in the analysis. In these situations the timestamp of the sensor event is an important parameter that is used in describing each sensor event and generating activity feature vectors.

Using this approach, parameter w_i refers to a time duration. This is a good approach when dealing with data obtained from sensors that sample their state at constant time intervals. In such a scenario, every window is guaranteed to contain a fixed amount of data. This is a common approach when using accelerometers and gyroscopes, for example, where data is sampled at a constant rate from the sensors. However, one has to deal with the problem of selecting the optimal length of the time interval. If a very small interval is chosen, there is a possibility that it will not contain any relevant activity information for making any useful decision. If the time interval is too wide, then information pertaining to multiple activities can be embedded into it and the activity that dominates the time interval will have a greater influence in the classification decision. This problem manifests itself when dealing with sensors that do not have a constant sampling rate. In the current context of motion and door sensor events, it is very likely that some time intervals do not have any sensor events in them (e.g., T_6 in Figure 5.2). One approach that can be taken when the optimal window size is unknown is employ an ensemble of classifiers, as described in Chapter 4, each of which is trained for a different window size.

5.2.2. Size based windowing

A second approach to defining window sizes is to divide the sequence into windows containing an equal number of sensor events. This is commonly referred to as *fixed-size*, or *sequence-based sliding windows*. Fixed-sized windows are useful for situations in which the arrival rate of the data is not constant. Using this approach, parameter w_i refers to a number of sensor events. This is illustrated in Figure 5.2 by the subsequences denoted as W_1, W_2, .., W_{21}. Even of the size of the window (defined by number of sensor events in the window) is fixed, these windows may actually vary in their time duration. This is appropriate considering that during the performance of highly-mobile activities, multiple motion sensors could be triggered, while during more sedentary activities or silent periods, there will few, if any, sensor events. The sensor events preceding the last event in a window define the context for the last event.

Like the previous approach, this method also has some inherent drawbacks. For example, consider the sequence W_{11} in Figure 5.2. The last sensor event of this window corresponds to the beginning sensor event of activity A_3. It is possible that there exists a significant time lag between this event and its preceding sensor event. The relevance of all the sensor events in this window to the last event in the window might be minimal if the time lag is large. As a result, treating all the sensor events with equal importance may result in loss of recognition effectiveness. Note also that if all of the sensors provide constant-time sampling, the time based windowing and size based windowing approaches will yield the same results.

In the presence of multiple residents, sensor firings from two different activities performed by the different residents will be grouped into a single window, thereby introducing conflicting influences for the classification of the last sensor event. While by itself this approach may not be intuitively alluring, we will show that events can be weighted within the window to account for the relationship between the sensor events. This type of windowing approach does offer computational advantages over the explicit segmentation process and can perform in real time because it does not require knowledge of future sensor events to classify past or current sensor events.

The value of the window size parameter, *w*, may depend on the context in which activity recognition is performed. The value can be derived through an empirical process by studying the effect of the different values of *w* on the performance of the recognition system. Among the different factors that influence the value for *w* is the average number of sensor events that span the duration of alternative activities. At one end of the spectrum are activities such as Leave Home that may be defined by rapid firing of a small set of environment sensors, while at the other extreme is the activity Sleep that continues for hours but typically results in an occasional firing of one or two sensors due to minimal resident movement during this time. Ideally the size of a sensor event window should be large enough to define the context of the last sensor event. Heuristics such as the average length of the longest recognizable activity can be used to bound the window size.

2009-07-19	10:18:59.406	LivingRoom	ON	Personal_Hygiene
2009-07-19	10:19:00.406	Bathroom	OFF	Personal_Hygiene
2009-07-19	10:19:03.015	OtherRoom	OFF	Relax
2009-07-19	10:19:03.703	LivingRoom	OFF	Relax
2009-07-19	10:19:07.984	LivingRoom	ON	Relax
2009-07-19	10:19:11.921	LivingRoom	OFF	Relax
2009-07-19	10:19:13.203	OtherRoom	ON	Relax
2009-07-19	10:19:14.609	Kitchen	ON	Relax
2009-07-19	10:19:17.890	OtherRoom	OFF	Relax
2009-07-19	10:19:18.890	Kitchen	OFF	Relax
2009-07-19	10:19:24.781	FrontMotion	ON	Leave_Home
2009-07-19	10:19:28.796	FrontMotion	OFF	Leave_Home
2009-07-19	10:19:31.109	FrontDoor	CLOSE	Leave_Home
2009-07-19	12:05:13.296	FrontDoor	OPEN	Enter_Home

Figure 5.4. Illustration of time dependency in a sliding window of sensor events.

5.2.3. Weighting events within a window

Once the sensor window S_i is defined, the next step is to transform this window into a feature vector that captures relevant activity information content as described in Chapter 3. However, one of the problems associated with fixed-size windowing is that windows could contain sensor events that are widely spread apart in time. An illustration of this problem is presented in Figure 5.4. These are examples of a sequence of discrete-event sensor events collected by environment sensors in a smart home. Notice the time stamp of the last two events in the sequence in Figure 5.4. There is a gap of nearly one and a half hours between these sensor events. All the sensor events that define the context of the last event within this window have occurred in the "distant" past. In the absence of any weighting scheme, the feature vector may be biased toward the inclusion of misleading information. Even though the sensor event corresponding to the end of the Personal Hygiene activity occurred in the past, it would have an equal influence on defining the context of the event corresponding to Enter Home. To overcome this problem, a time-based weighting scheme can be incorporated to take into account the relative temporal distance between the sensors.

When the sliding window is a constant size, it is possible for two sensor events that are spread apart in time to be part of the same window. In order to reduce the influence of such sensor events on deciding the activity label for the most recent sensor event, a time-based weighting factor can be applied to each event in the window based on its relative time to the last event in the window.

Let $\{t_{i-w},..,t_i\}$ represent the time stamps of the sensor events in window S_i. For each sensor event e_j, the difference between the time stamp of e_j and the time stamp of e_i, the last event in the window, is computed. The contribution, or weight, of sensor event e_j can be computed using an exponential function as shown in Equation 5.1.

$$C(i, j) = \exp(-\mathrm{X}^{(t_i - t_j)}) \tag{5.1}$$

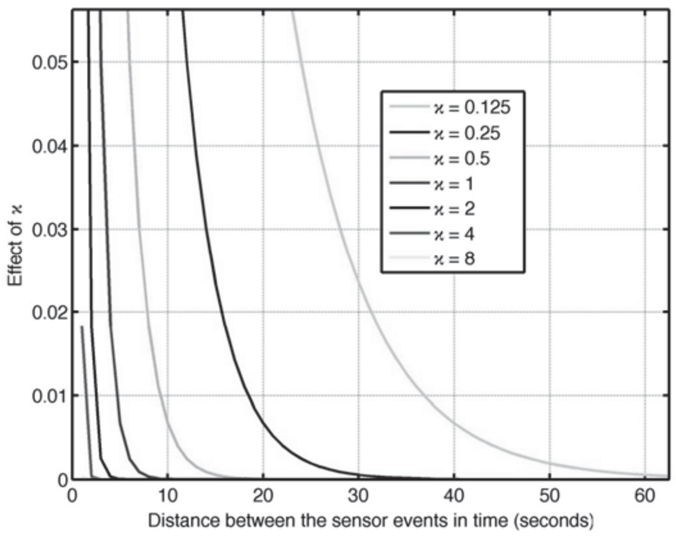

Figure 5.5. Effect of $\mathcal{X}$ on weights.

Features that are based on a simple count of the sensor events within a window can now be replaced by a sum of the time-based contributions of each sensor event within the window. Features which sum the values of sensor events can also employ this type of weighting approach to adjust the influence of each sensor event on the feature vector. The value of $\mathcal{X}$ determines the rate of decay of the influence. Figure 5.5 shows the effect of the choice of $\mathcal{X}$ on the rate of decay.

If $\mathcal{X}>1$, then only sensor events that are temporally close to the last event contribute to the feature vector. When $0 < \mathcal{X} < 1$, the feature vector is under the influence of a temporally wider range of sensor events. When $\mathcal{X} = 0$, the temporal distance has no influence on the feature vector, making it a simple count of the different sensor events.

Similarly, in situations when the sensor event corresponds to the transition between two activities (or in other settings when multiple activities are performed by more than one resident in parallel), the events occurring in the window might not be related to the sensor event under consideration. An example of this situation is illustrated in Figure 5.6. This particular sequence of sensor events from a smart home testbed represents the transition from the Personal Hygiene activity to the Leave Home activity. Notice that all the initial sensor events in the window come from a bathroom in the home, whereas the second set of sensor events are from an unrelated functional area of the apartment, namely the area near the front door. While this certainly defines the context of the activity, since the sensors from a particular activity dominate the window, the chances for a wrong conclusion about the last sensor event of the window are higher. This problem can be addressed by defining a weighting scheme based on the mutual information between the sensors.

2009-07-23	19:59:58.093	Bathroom	ON	Personal_Hygiene
2009-07-23	20:00:02.390	Bathroom	OFF	Personal_Hygiene
2009-07-23	20:00:04.078	Bathroom	ON	Personal_Hygiene
2009-07-23	20:00:08.000	LivingRoom	ON	Relax
2009-07-23	20:00:08.640	OtherRoom	ON	Relax
2009-07-23	20:00:09.343	LivingRoom	OFF	Relax
2009-07-23	20:00:12.296	Kitchen	ON	Relax
2009-07-23	20:00:25.140	LivingRoom	OFF	Relax
2009-07-23	20:00:27.187	FrontMotion	ON	Leave_Home
2009-07-23	20:00:27.437	Kitchen	OFF	Leave_Home
2009-07-23	20:00:30.140	FrontMotion	OFF	Leave_Home
2009-07-23	20:00:32.046	FrontMotion	ON	Leave_Home
2009-07-23	20:00:36.062	FrontMotion	OFF	Leave_Home
2009-07-23	20:00:39.343	FrontMotion	ON	Leave_Home
2009-07-23	20:00:43.671	FrontMotion	OFF	Leave_Home
2009-07-23	20:00:46.265	FrontDoor	CLOSE	Leave_Home

Figure 5.6. Illustration of sensor dependency in a sliding window of sensor events.

The mutual information measure reduces the influence of sensor events within the window that do not typically occur within the same time frame as the last sensor event in the window. In the context of environmental sensors, motion sensors that are geographically distant from the most recent sensor location will receive less weight than those that are close. Mutual information is typically defined as the quantity that measures the mutual dependence of two random variables. For sensor-based activity recognition, each individual sensor is considered to be a random variable. The mutual information or dependence between two sensors is then defined as the chance of these two sensors occurring successively in the entire sensor stream. If S_i and S_j are two sensors, then the mutual information between them, $MI(i,j)$, is defined as

$$MI(i, j) = \frac{1}{N} \sum_{k=1}^{N-1} \delta(s_k, S_i)\delta(s_{k+1}, S_j) \tag{5.2}$$

where

$$\delta(s_k, S_i) = \begin{Bmatrix} 0 & if\ s_k \neq S_i \\ 1 & if\ s_k = S_i \end{Bmatrix}. \tag{5.3}$$

The summed term thus takes a value of 1 when the current sensor is S_i and the subsequent sensor is S_j. If two sensors are adjacent to each other, such that triggering one sensor most likely results in also triggering the other sensor, then the mutual information between these two sensors will be high. Similarly, if the sensors are far apart such that they do not often occur together, then the mutual information between them will be low. Note that the computation of mutual information using this bi-gram model depends on the order in which sensor events occur.

The mutual information matrix is typically computed offline using sample data, with or without activity labels, from the set of sensors that will be employed for activity recognition. As an example, consider the event sequence shown in Figure 5.6. There are 16 events, or 15 pairs of successive events (N=15). The sequence {Bathroom, LivingRoom} appears 1 time and {LivingRoom, Bathroom} does not appear at all, so MI(Bathroom, LivingRoom) = 1/15. In contrast, MI(Bathroom, FrontDoor) = 0 and MI(LivingRoom, Kitchen) = 2/15.

Once computed, the MI matrix can then be used to weight the influence of sensor events in a window while constructing the feature vector. Similar to time-based weighting, each event in the window is weighted with respect to the last event in the window. Thus instead of computing feature values based on the count of different sensor events or an aggregation of the sensor values, it is the aggregation of the contribution of every weighted sensor event based on mutual information that can is included in the feature vector representing the data in the sliding window.

While Equations 5.2 and 5.3 are based on the occurrence of events from a particular sensor, a similar approach can be used to weight sensor events with values that do not commonly occur in the same window with other sensor values. By weighting sensors according to mutual information, the impact of sensor events can be reduced when their occurrence is due to noise, to interweaving of activities, or to activities of multiple individuals that are being detected by the same sensors.

Figure 5.7 shows intensity-coded mutual information scores between different sensors located in a smart home. It is evident from the figure that each of the sensors is functionally very distinct from the others. Furthermore, the relatively strong diagonal elements indicate the higher chance of sensor generating multiple similar messages in a row rather than transitioning to different sensors. Because each of these sensors is triggered by human motion, the MI values for this testbed indicate that the resident tends to remain in one location more than moving around. A few other observations can be made from Figure 5.7. For example, consider the similarities between sensors 5 and 6. These two sensors correspond to the *Front door* and *Kitchen* sensors that are geographically close to each other. As a result, when the resident enters the testbed the kitchen sensor is likely to sense motion as well as the front door motion sensor. However, the *Kitchen cabinet* sensors (#7) do not get triggered. Another subtle observation is the relatively high similarity between the *Medicine cabinet* sensor (#13), the *Kitchen* sensor, and the *Kitchen cabinet* sensors (#7). The resident of this home stores medicine in the kitchen cabinet. When the resident retrieves medicine each day all of the related kitchen sensors have a relatively high likelihood of firing.

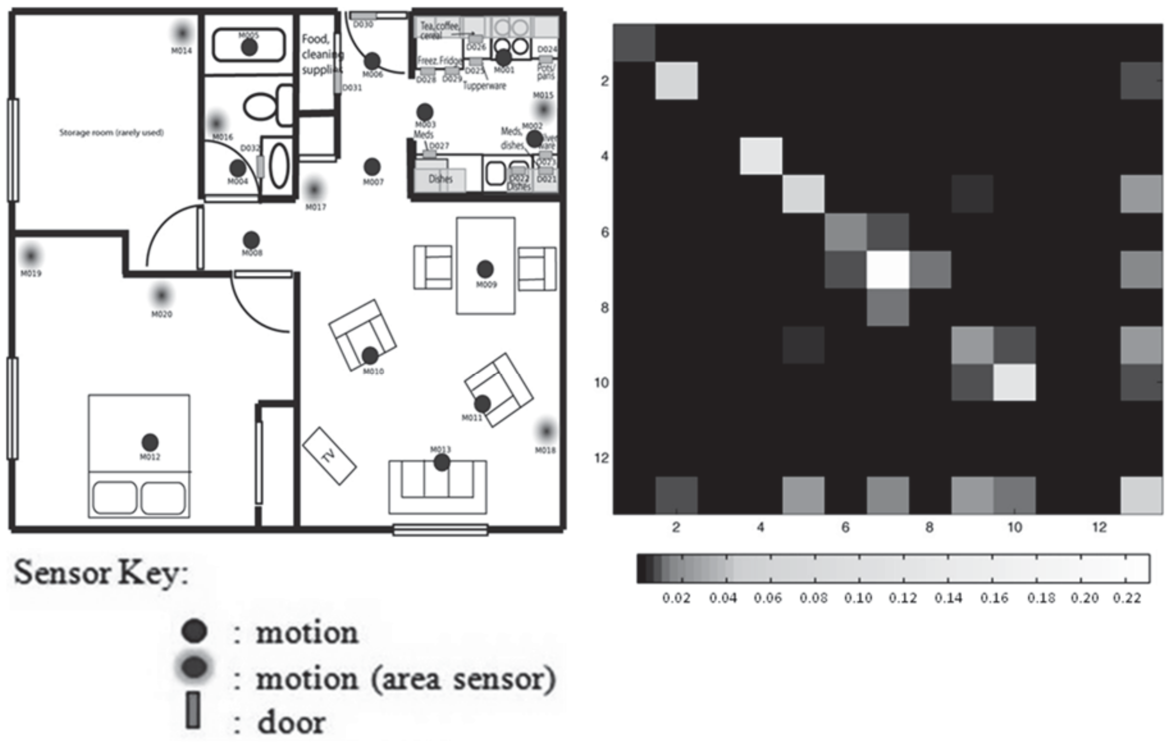

Figure 5.7. Intensity-coded mutual information between every pair of motion sensors (right) in a smart home (left).

5.2.4. Dynamic window sizes

The previously described approaches employ a fixed window size for computing the feature vectors. An important challenge with this approach is identifying the optimal window size. Various heuristics such as the mean length of the activities and sampling frequency of the sensors can be employed to determine the window size. For fixed-sized windows, lengths between 5 and 30 are common. For time-based windows containing events from high-frequency sampled sensors, common sizes are between 1 and 6 seconds. An alternative approach, however, is to automatically derive the window size from the data itself.

A probabilistic method can be used to derive an appropriate window size for each sensor event that is being classified with an activity label. Assuming that there are *M* activity labels, a set of candidate window sizes are defined, $\{w_1, w_2, .., w_L\}$ where w_1 corresponds to the minimum window size that is observed in the data for any activity, $w_1 = \min\{ws(A_1), ws(A_2), .., ws(A_M)\}$, w_L is the median window size observed for all of the activities in *A*, $w_L = median\{ws(A_1), ws(A_2), .., ws(A_M)\}$, and the remaining candidate window sizes are obtained by dividing the $[w_1, w_L]$ interval into equal-length bins. After generating candidate window sizes, the most likely window size for an activity A_m can be calculated as

$$w^* = \arg\max_{w_l} \{P(w_l \mid A_m)\}. \tag{5.4}$$

Note that we can also estimate the probability of an activity A_m being associated with sensor s_i as $P(A_m \mid s_i)$. Thus, if the most recent sensor event is generated by sensor s_i, the most likely activity A^* associated with the sensor is calculated as

$$A^* = \arg\max_{A_m} \{P(A_m \mid s_i)\} \tag{5.5}$$

To consider both of these influences in selecting an optimal size we can combine Equations 5.4 and 5.5. The optimal window size for events generated by sensor s_i can be determined by combining the two equations according to the factorization in Equation 5.6.

$$w^* = \arg\max_{w_l} P(w_l \mid s_i) = \arg\max_{w_l} [P(w_l \mid A_m) * P(A_m \mid s_i)]. \tag{5.6}$$

Each of the probability vectors can be estimated using available sample data. The window sizes for the sensor events that are used to train activity recognition classifiers and to label new events are computed using the probabilities estimated from the sample data.

A similar approach can be taken to dynamically compute time-based window sizes. As with size-based windows, time windows can be defined for every sensor event. For a sensor event s_i, a time window consists of all sensor events that fall within a fixed time duration starting from the timestamp of event s_i. The sensor events within a particular time window are aggregated to form features using the same process that was adopted for length-based windows. The time window can be labeled using the activity associated with the last sensor event in the time window.

As an example, consider a dataset that is formed by combining the events in Figures 5.4 and 5.6. For this dataset, the window sizes for the activities that occur are

Personal hygiene	{2, 3}	Relax	{5, 8}
Leave home	{3, 8}	Enter home	{1}

The minimum window size is 1 and the median size is 3. We consider the candidate window sizes 1 and 3. Consider that the most recent sensor event is generated by a *Front door* sensor. We see from the data that P(LeaveHome | FrontDoor) = 0.67 and P(EnterHome | FrontDoor) = 0.33, so the most likely activity given this sensor event is Leave Home. For the Leave Home activity, the only window sizes that are observed are sizes 3 and 8, each with probability 0.5. The probability of window size 1 for Leave Home is 0.0, so the most likely window size of 3 would be used to determine the context for this activity.

5.3 Unsupervised Segmentation

Although activity-centered segmentation allows activity breakpoints to be easily detected, it relies on knowledge of the predefined activities and sufficient training data to learn models of activities and their boundaries. It is more difficult to leverage activity-centered approaches to detect breakpoints within activities that are highly variable and do not appear in the vocabulary of predefined tasks. This is a limitation, because such free-form tasks are very common in routine behavior. Here we describe two alternative approaches, change point detection and piecewise representation, that can be used to identify potential activity boundaries without explicit guidance from a supervised learning algorithm.

Change point detection. An alternative method for activity segmentation is to detect activity breakpoints or transitions based on characteristics of the data without being given explicitly-labeled examples of activity breakpoints or transitions. We have already shown a number of approaches that exist to segmenting time series data. However, in some cases we want to 1) identify likely activity boundaries in real time as the data is observed and 2) perform the detection without explicit training a classifier on known activities or known activity transitions.

To accomplish this we can borrow from time series analysis to perform change point detection, which is the problem of detecting abrupt changes in time series data. Such abrupt changes may represent transitions that occur as individuals complete one activity and begin another activity. To perform change-point detection, probability distributions of the feature descriptions can be estimated based on the data that has been observed since the previous candidate change point. In order to ensure that change points are detected in real time, the algorithm only tries to identify the most recent change, or activity boundary. Sensor data from the current point in time back to the most recent change point is referred to as a run, *r*. If *t* sensor events or time units have occurred since the most recent detected change point, then there are *t* possible runs of different lengths that could be associated with the recent data. If the algorithm is applied after each new sensor event is received, then there are only two possibilities to consider: either the new event is a member of the current run and the run length increases by one, or a change point occurred, the run ended, and the new sensor event is part of a new run. Let r_t refer to the length of the most recent run. We can then calculate the probability of r_t given the previous *t* sensor events.

$$P(e_{1:t+1} \mid e_{1:t}) = \sum_{r_t} P(e_{t+1} \mid r_t \,\&\, e_{1:t}) P(r_t \mid e_{1:t}) \tag{5.7}$$

We can compute the probability of the run length after event *t* given the corresponding sequence of sensor events using Bayes rule.

$$P(r_t \mid e_{1:t}) = \frac{P(r_t \,\&\, e_{1:t})}{P(e_{1:t})} \tag{5.8}$$

The numerator can then be recursively computed based on the previous run length and corresponding sensor events.

$$\begin{aligned} P(r_t \,\&\, e_{1:t}) &= \sum_{r_{t-1}} P(r_t \,\&\, r_{t-1} \,\&\, e_t) \\ &= \sum_{r_{t-1}} P(r_t \,\&\, e_t \mid r_{t-1} \,\&\, e_{1:t-1}) P(r_{t-1} \,\&\, e_{1:t-1}) \\ &= \sum_{r_{t-1}} P(r_t \mid r_{t-1}) P(e_t \mid r_{t-1} \,\&\, e_{1:t-1}) P(r_{t-1} \,\&\, e_{1:t-1}) \end{aligned} \tag{5.9}$$

The algorithm to compute the probabilities does not require storing all of the intermediate probability values but can use the probabilities estimated up through the previous sensor event to compute the probabilities for the current sensor event. The process starts at the previous change point, so $P(r_0 = 0) = 1$. The term $P(e_t \mid r_{t-1} \,\&\, e_{1:t-1})$ represents the probability that the most recent sensor event belongs to each of the possible runs identified through the previous data point. Calculating this probability relies on knowing the type of data distribution. We let e_t^r represent the data point that is associated with run r_t and π_t^r represent the probability of data point e_t^r based on the run through *t* and the appropriate data distribution.

The probability of r_t given the previous value has two cases depending on whether a change point just occurred or whether a change point did not occur and the run length grew by one. To compute these two cases, we first compute the hazard function *H*. The hazard function is used to represent a failure rate, which is estimated here as the number of events that end a run (or segment) to the total number of events, based on sample data.

$$P(r_t \mid r_{t-1}) = \begin{cases} H(r_{t-1}+1) & \text{if } r_t = 0 \\ 1-H(r_{t-1}+1) & \text{if } r_t = r_{t-1}+1 \end{cases} \tag{5.10}$$

The change point detection algorithm is summarized in Figure 5.8. As the pseudocode shows, a change point can be detected and the corresponding activity boundary made after looking ahead only one sensor event. The parameters of the predictive distribution associated with a current run are indicated by X_t^r. Simple distributions such as the marginal predictive distribution can be used when the exact distribution of the data is unknown.

For detecting activity transitions, the sequence data is constructed as follows. The current state of the environment (including the sensors and the residents or the individual wearing sensors) is viewed as a single data point. This gives us a multi-dimensional data point where the number of dimensions corresponds to the number of features describing the state. The current state is then updated at every time step, or after every sensor event, to yield the sequence data. The probability of a run ending before the current time point or sensor event is computed and compared with the probability of the run extending to include the current data point. If the probability of the run ending is larger, then the activity boundary is noted before the most recent event and the process repeats with the new run starting at the most recent sensor event.

```
Algorithm DetectChangePoint()
  Boundaries = {}
  begin = 1
  Boundaries.append(begin)    // Indicate the beginning of the sequence is an activity boundary
  P(r_0 = 0) = 1              // The initial run length is 0
  $X_1^0 = X_{prior}$         // Initialize the parameters of the data distribution model based on priors
  i = 1
  while data do
    i = i + 1
                              // Calculate the probability that the run length grows by 1
    $P(r_i = r_{i+1} + 1 \& e_{1:i}) = P(r_{i-1} \& e_{1:i-1}) \times \pi_i^r \times (1 - H(r_{i-1} + 1))$
                              // Calculate the probability that a change point occurred
    $P(r_i = 0 \& e_{1:i}) = \sum_{r_{i-1}} P(r_{i-1} \& e_{1:i-1}) \times \pi_i^r \times H(r_{i-1})$
                              // Update the data evidence probability
    $P(e_{1:i}) = \sum_{r_i} P(r_i \& e_{1:i})$
                              // Determine the run length distribution
    $P(r_i \mid e_{1:i}) = P(r_i \& e_{1:i}) / P(e_{1:i})$
                              // Update the distribution statistics
    $X_{i+1}^{r+1} = X_{prior}$
                              // Predict the next data point
    $P(e_{i+1} \mid e_{1:i}) = \sum_{r_i} P(e_{i+1} \mid e_i^r \& r_i) \times P(r_i \mid e_{1:i})$
    if $P(r_i = 0) > P(r_i = r_{i+1} + 1)$
      end = i - 1
      Boundaries.append(end)
      begin = i
      Boundaries.append(begin)
    end if
  done
return Boundaries
```

Figure 5.8. Online change point detection algorithm.

Piecewise representation. The idea of piecewise approximation originates from time series analysis, in which the entire series cannot be accurately represented by a single simple function yet it can be segmented into pieces which can be individually represented by such functions. For an unsupervised approach to segmentation, we can try to model each segment with a simple regression or classifier model. Supervised versions of piecewise approximation can use recognition accuracy based on pre-trained activity models to determine the value of segment choice.

```
Algorithm HybridSegment(Buffer, BufferSize)
  // BufferSize is specified as 5 times the size of a typical activity segment
  Boundaries = {}
  begin = 1
  Boundaries.append(begin)    // Indicate the beginning of the sequence is an activity boundary

  while data do
      error = ModelError(Buffer)
      Segment = BottomUp(Buffer, error)
      end = begin + Segment[1].size
      Boundaries.append(end)
      begin = end + 1
      Boundaries.append(begin)
      Buffer.remove(Segment[1].size)          // Remove the first segment from the buffer
      Buffer.append(data, Segment[1].size)  // Add the next Segment[1].size events to the buffer
  done

  return Boundaries


Algorithm BottomUp(Seq, MaxError)
    Segment = {}                  // Segment contains the size of each segment in the input sequence

    for i = 1 to Seq.size - 1
        S.begin = i
        S.end = i
        S.size = 1
        Segment.append(S)
    done

    for i = 1 to length(Segment) - 1
        MergeCost[i] = ModelError(Merge(Segment[i], Segment[i+1]))
    done

    while min(MergeCost) < MaxError
        i = min(MergeCost)
        S[i] = Merge(Segment[i], Segment[i+1])
        Segment.remove(S[i+1])
        if i < length(Segment)
            MergeCost[i] = ModelError(Merge(Segment[i], Segment[i+1]))
        end if

        if i > 1
            MergeCost[i-1] = ModelError(Merge(Segment[i-1], Segment[i]))
        end if
    done

    return Segment
```

Figure 5.9. Hybrid bottom-up / sliding-window segmentation algorithm.

Given a method of representing each segment and an error function to determine the fit between the function and the data, we can employ several different techniques to efficiently search through the space of possible segment choices. A *top down* approach starts by viewing the entire sequence as a segment and chooses a point in the sequence (if one exists) where splitting the sequence into two smaller segments results in decreased error over representing the segments separately. The process repeats until performance is not improved by further splitting. A *bottom up* method starts by considering each event as a separate segment and merges neighboring segments until the performance does not improve with further merging. These techniques are similar to hierarchical clustering techniques that have been applied to non-sequential data. A third *sliding window* technique starts at the beginning of the input sequence with a segment of size one and grows the segment until the performance does not improve with further extensions of the segment. The process then repeats starting with the first data point not included in the most recent segment.

These techniques are effective for a variety of types of data with or without the use of trained activity models. However, they are designed to be applied in offline mode to the entire sequence of available sensor event data. A hybrid method can be used to combine these approaches in a semi-online approach. Using the hybrid algorithm, a subsequence is stored in a buffer that is long enough to store approximately five typical segments. The bottom-up technique is used to segment the data in the buffer and the left-most segment boundary is recorded. The data corresponding to the segment is removed and the next data points in the sequence are added to the buffer to maintain a constant-size buffer, then the process is repeated. This hybrid technique provides more of a "look ahead" then is used by a pure sliding window and thus the segment quality can be improved. Unlike a pure bottom up method, however, this hybrid method allows the data to be processed in near real time. Figure 5.9 provides pseudocode for the hybrid segmentation algorithm.

5.4 Measuring Performance

Once a sequence of sensor events is partitioned into overlapping windows or non-overlapping windows (segments), the activity recognition algorithm uses one of the classifiers described in Chapter 4 to map each window's subsequence of sensor events s_i to an activity label A_i based on the learned mapping $f : S \rightarrow A$. In order to compare alternative activity recognition algorithms and to estimate the expected performance of the recognition for future data, we want to be able to determine the performance of the algorithm on already available data as well as data we may collect in the future.

To illustrate the performance evaluation process, consider an example scenario in which we train a Decision Stump classifier to distinguish our Hand Washing activity from the Sweeping activity as described in Chapter 3. As mentioned in Chapter 3, a decision stump classifier is a one-level decision tree. An attribute is selected to be queried as the root of the tree and its children are leaf nodes, which provide an activity classification. To simplify the recognition scenario, we utilize a fixed sliding window of 10 sensor events with only two types of features: the time duration of the window and the sensor counts (bag of sensors) for each window. Due to the choice of features we only consider the discrete event sensors. This includes the motion sensors, door sensors, item sensors, and utilization of water and the stove burner. Note that with a window size of 10 the hand washing activity contains 10 data points while the sweeping activity contains 44 data points.

A first step at evaluating the performance of an activity recognition algorithm is to generate a table that summarizes how the activities were classified, typically referred to as the confusion matrix. Each row of the matrix represents data points and their actual activity label, while each column of the matrix represents the data points and the activity label they were assigned by the classifier. As shown in Table 5.1, each cell [*i*,*j*] provides the number of data points from activity class *i* that were categorized as activity *j* by the classifier. Notice that the cells along the diagonal indicate the correct classifications. A classifier that provides correct labels for each data point would generate 0 values for every cell except those along the diagonal. Thus the values represented by "a" and "d" represent the number of correctly classified data points for this two-class problem while the values represented by "b" and "c" represent the number of incorrectly classified data points.

Table 5.1. Example confusion matrix.

	Classified as 1	Classified as 0
True 1	a (TP)	b (FN)
True 0	c (FP)	d (TN)

Table 5.2 summarizes the results from our activity example. Mistakes are made in labeling data points from both classes. The sweeping class has fewer errors, which could be due to its larger number of training data points. To shed more light on the performance of the recognition algorithm we next introduce a range of metrics that could further evaluate the algorithm.

Table 5.2. Confusion matrix using decision stump for recognizing hand washing and sweeping activities.

		Classifier label	
		Hand washing	Sweeping
Activity label	Hand washing	6	4
	Sweeping	3	41

Table 5.3. Confusion matrix using naive Bayes for recognizing hand washing, sweeping, and cooking activities.

		Classifier label		
		Hand washing	Sweeping	Cooking
Activity label	Hand washing	9	1	0
	Sweeping	0	44	0
	Cooking	0	0	32

While Table 5.2 shows the confusion matrix that results from learning two activity classes, the techniques can be applied to any number of activity classes. Table 5.3 shows the confusion matrix that results from applying a naive Bayes learner to the hand washing and sweeping activities as well as a third activity, Cooking. The Cooking class contains 32 examples that represent data collected while an individual cooked a bowl of soup. Note that even though the number of classes has increased, the naive Bayes classifier actually classifies more data points correctly than the decision stump did for two activity classes.

Many performance metrics have been introduced to evaluate classifier algorithms. These are useful for evaluating activity recognition algorithms. However, the activity recognition problem has some unique characteristics that are not captured by traditional metrics so additional measures need to be considered as well. In this section we provide an overview of both, using our hand washing vs. sweeping problem to illustrate the metrics.

5.4.1 Classifier-based activity recognition performance metrics

Accuracy. The most common metric for evaluating classifier performance is accuracy. Accuracy is calculated as the ratio of correctly-classified data points to total data points. Referring to the example confusion matrix in Table 5.1 we see that accuracy can be calculated from the matrix by summing the values along the diagonal and dividing by the sum of all of the values in the matrix, or (a+d)/(a+b+c+d).

$$Accuracy = \frac{\#\, correctly\ classified\ activities}{\#\, total\ activities} \tag{5.11}$$

Accuracy is a measure that can be applied to multi-class problems as well as binary-class problems. This will be necessary for most activity recognition algorithms (just the "clean house" category in Table 2.1 alone lists 23 separate activities to be recognized). However, accuracy does not tell the whole story, because it does not provide insights on the source of the error or the distribution of error among the different activity classes. For our activity recognition example in Table 5.2 we see that the accuracy is 0.87. However, the accuracy for the individual classes varies quite a bit (the accuracy for Hand Washing is 0.60 and the accuracy for Sweeping is 0.93). The companion to accuracy is *Error Rate*, which is computed as 1 - Accuracy.

Sensitivity, specificity. Accuracy and Error Rate give a high-level idea about the recognizer's performance. However, in cases where there is a non-uniform distribution among the activity classes, these measures can be deceiving. Accuracy and Error Rate are ineffective for evaluating classifier performance in a class-imbalanced dataset because they consider different types of classification errors as equally important. For example, if the dataset consists of 5% Eat activities and 95% Sleep activities, a random prediction of all of the test instances being Sleep will yield an accuracy of 0.95. However, in this case the classifier did not correctly recognize any of the Eat activities. In order to provide a comprehensive assessment of activity recognition, we need to either consider metrics that can report the performance of activity recognition on two classes separately or not let the effect of class imbalance be reflected in the metric. Sensitivity, specificity, and g-mean are useful for reporting recognizer performance in such cases of class imbalance and for providing insight on recognition performance for individual classes. Sensitivity, also referred to as the *true positive rate* or *true activity rate (TP Rate)*, refers to the portion of a class of interest (the positive class) that was recognized correctly. Referring to the example confusion matrix in Table 5.1 we see that the true positive rate is a/(a+b). For a problem with more than two activity classes the true positive rate would be calculated in a similar manner. Using the example in Table 5.3, if Hand Washing is the positive class then the true positive rate would be calculated as the number of Hand Washing examples classified as Hand Washing divided by the total number of Hand Washing examples, or 9/10 = 0.9.

$$Sensitivity = TP\ Rate = \frac{TP}{TP+FN} \tag{5.12}$$

Similarly, the *false positive rate (FP Rate)* can be reported as the ratio of negative examples (examples not in the class of interest) classified as positive to the total number of negative examples. The false positive rate for the confusion matrix in Table 5.1 is computed as c/(c+d).

$$FP\ Rate \quad = \quad \frac{FP}{FP+TN} \tag{5.13}$$

In contrast, specificity refers to the portion of the negative class examples (examples not in the class of interest) that were recognized correctly.

$$Specificity = \frac{TN}{TN+FP} \tag{5.14}$$

G-mean. G-mean utilizes the Sensitivity and Specificity measures the performance of the activity recognition algorithm both in terms of the ratio of positive accuracy (Sensitivity) and the ratio of negative accuracy (Specificity). G-mean thus provides activity class-sensitive measure of the recognition performance. The sensitivity, specificity, and g-mean scores can be calculated separately for each class to see how well the recognition algorithm handles each activity. They can also be combined by averaging the scores over each class, multiplied by the class size to provide a weighted average and thus an overall performance measure for the algorithm. Note that the weighted average sensitivity and specificity scores yields the same result as the accuracy calculation.

$$G\text{–}mean \quad = \quad \sqrt{\frac{TP}{TP+FN} \times \frac{TN}{TN+FP}} \quad = \quad \sqrt{Sensitivity \times Specificity} \tag{5.15}$$

Precision, recall, f-measure. Like sensitivity and specificity, precision and recall provide class-sensitive measures. Precision is calculated as the ratio of true positive data points to total points classified as positive, while recall is calculated as the ratio of true positive data points to the total points that are true (should be classified as true).

$$Precision = \frac{TP}{TP+FP} \tag{5.16}$$

$$Recall = \frac{TP}{TP+FN} \tag{5.17}$$

While precision and recall can be used to evaluate activity recognition performance, they are traditionally used to evaluate the performance of retrieval approaches, such as document retrieval algorithms. In the context of activity learning, these can be useful measures if the goal is not to just label a particular data sequence with an activity name but to search through a set of sensor event sequences to retrieve all of the sequences in which a particular activity occurs. In this case, precision can be used to determine the proportion of identified sequences that are relevant (actually contain the activity of interest). This can be used, for example, to determine on which days a caregiver visited to provide physical therapy for a smart home resident. Similarly, recall can be used in this situation to calculate the fraction of relevant sequences that are actually identified and retrieved by the algorithm. In these cases, TP is measured as the number of relevant identified sequences. The precision denominator, TP+FP, is the total number of identified sequences, and the recall denominator, TP+FN, indicates the number of relevant sequences that exist. For activity recognition evaluation, recall can be viewed as synonymous with sensitivity or TP rate.

Finally, f-measure (also referred to as f-score or f1 score) provides a way to combine precision and recall as a measure of the overall effectiveness of activity classification. F-measure is calculated as a ratio of the weighted importance of recall or precision. The weight coefficient in Equation 5.7, β, is typically set to 1. Table 5.4 summarizes these performance metrics for our example activity recognition problem.

$$F\text{–}measure = \frac{(1+\beta)^2 \times \frac{TP}{TP+FN} \times \frac{TP}{TP+FN}}{\beta^2 \times \frac{TP}{TP+FN} + \frac{TP}{TP+FN}} = \frac{(1+\beta)^2 \times Recall \times Precision}{\beta^2 \times Recall + Precision} \tag{5.18}$$

Table 5.4. Traditional classifier-based performance evaluation for decision stump algorithm applied to hand washing (H) and sweeping (S) activities.

Performance Metric and Value			
Accuracy	0.87	G-mean (H)	0.56
Error rate	0.13	G-mean (S)	0.56
Sensitivity (H)	0.60	Precision (S)	0.67
Sensitivity (S)	0.93	Precision (H)	0.91
FP Rate (H)	0.07	F-measure (S)	0.55
FP Rate (S)	0.40	F-measure (H)	0.55
AUC-ROC	0.83	AUC-PRC	0.88

Kappa statistic. While the performance metrics we have described so far give an indication of the recognition algorithm's performance, the values may be difficult to interpret. If two competing algorithms are being considered they can be compared using these metrics. If one algorithm is being considered, it is useful to compare it to a baseline. The Kappa statistic, or Cohen's kappa coefficient, provides a way to compare the algorithm to one that assigns activity labels based on chance. The Kappa statistic is traditionally used to assess inter-rater reliability, or the degree to which two raters agree on the class value for a data point. When evaluating an activity recognition algorithm, the two raters represent the algorithm being evaluated and ground truth (the actual activity labels). The Kappa statistic can take on a value between -1 and 1, where a value of 1 indicates perfect agreement, a value of 0 indicates the agreement is equal to chance, and a value of -1 indicates perfect disagreement. P(A) is the probability of agreement between the activity recognition and ground truth (the accuracy of the algorithm) and P(E) is the probability that the accuracy is due to chance.

$$\kappa = \frac{P(A) - P(E)}{P(E)} \tag{5.19}$$

P(E) can be estimated using a number of methods. Here we estimate it by computing the distribution of class labels assigned by the algorithm as well as the actual class label distribution. Thus P(E) = P(H|Alg) × P(H) + P(S|Alg) × P(S) = 8/54 × 10/54 + 46/54 × 44/54 = 0.72 for our example scenario. The κ value can then be computed as (0.87 - 0.72) / (1.00 - 0.72) = 0.54.

In addition to using the Kappa statistic to evaluate the performance of a classifier, it is also a useful statistic for evaluating the reliability of labeled activity data. If multiple individuals examine sensor data in order to annotate the data with ground truth labels, the consistency of the labels can be determined between the annotators by calculating P(A) = the proportion of data points where the annotators agree on the label and P(E) = the proportion of data points where the annotators would agree if labels were randomly assigned.

Receiver Operating Characteristics Curve (ROC). ROC-based assessment facilitates explicit analysis of the tradeoff between true positive and false positive rates. This is done by plotting a two-dimensional graph with the false positive rate on the x axis and the true positive rate on the y axis. An activity recognition algorithm produces a (TP_Rate, FP_Rate) pair that corresponds to a single point in the ROC space, as shown in Figure 5.6. One recognition algorithm can generally be considered as superior to another if its point is closer to the (0,1) coordinate (the upper left corner) than the other. If the algorithm generates a probability or confidence value and uses a threshold to decide whether the data sample belongs to the activity class, the threshold value can be varied to generate a set of points in the ROC space. This set of points generates an ROC curve, as shown by the solid-line curve in Figure 5.10.

Figure 5.10. Example ROC curve (left) and PR curve (right).

To assess the overall performance of an activity recognition algorithm, we can look at the *Area Under the ROC curve*, or AUC. In general, we want the false positive rate to be low and the true positive rate to be high. This means that the closer to 1 the AUC value is, the stronger is the recognition algorithm. In Figure 5.10, the algorithm that generated curve A would be considered stronger than the algorithm that generated curve B. Another useful measure that can be derived from the ROC curve is the *Equal Error Rate* (EER), which is the point where the false positive rate and the false negative rate are equal. This point, which is illustrated in the ROC curve in Figure 5.6, is kept small by a strong recognition algorithm.

Precision-Recall Curve (PR Curve). A PRC can also be generated and used to compare alternative activity recognition algorithms. The PR curve plots precision rate as a function of recall rate. While optimal algorithm performance for an ROC curve is indicated by points in the upper left of the space, optimal performance in the PR space is near the upper right. As with the ROC, the area under a PRC can be computed to compare two algorithms and attempt to optimize activity recognition performance. Note that the PR curves in Figure 5.10 (right) correspond to the same algorithms that generated the ROC curves in Figure 5.10 (left). In both cases, Algorithm A yields a larger area under the curve and could be interpreted as outperforming Algorithm B. It should be noted that an algorithm that performs best using the ROC metric may not perform best using the PRC and vice versa. However, the metrics can be considered separately to better understand the nature of algorithm performance. The PR curve in particular provides insightful analysis when the class distribution is highly skewed, which often happens when modeling and recognizing activities.

5.4.2 Event-based activity recognition performance metrics

Activity recognition can be viewed as a classification task. However, there are aspects of activity recognition that are not like many other classification problems. First, the sequential nature of the sensor events means that the data points are not independent. In fact, they are related in time, in space, and in function. Second, the distinction between the class labels is not always clear. Some activities are very similar in function and purpose and thus the sensor events are also similar. Finally, if activities are not pre-segmented, then any given window of sensor events could represent a transition between activities rather than just one activity. As a result, additional performance metrics are needed to better understand possible mis-matches between the labels that are generated by an activity recognition (AR) algorithm and the ground truth activity labels for a given data point.

Using event-based evaluation relies on considering entire sensor event segments at a time. In contrast with pre-segmented data, a segment here is a subsequence within the input data where the AR label and the ground truth label remain constant. Figure 5.11 shows an artificial sequence of sensor events that are labeled with the hand washing and sweeping activities. Segment boundaries are indicated with vertical lines. With this approach to performance evaluation, each segment is analyzed to determine if it is correctly classified (the AR label matches the ground truth) or not. By considering one activity at a time, this can be viewed as a binary classification problem. For example, in Figure 5.9 the activity being considered, or positive class, is hand washing. A TP occurs when both rows show hand washing, a TN occurs when both rows show sweeping, and the other situations indicate a FP or FN. The false positive and false negative errors are further divided into the subcategories described here to better understand the type of error that is occurring.

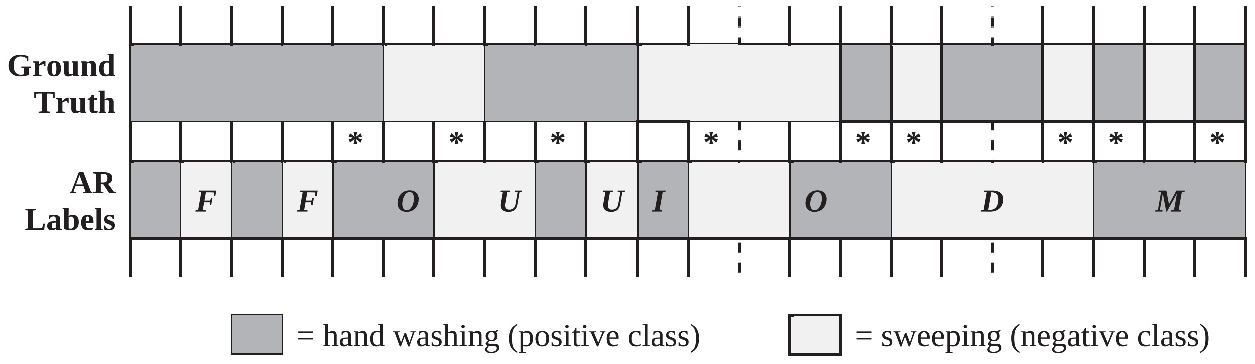

Figure 5.11. Event-based evaluation of hand washing activity recognition. The vertical solid lines indicate the performance segment boundaries. The top row shows the ground truth labeling of hand washing or sweeping and the bottom row shows the labels generated by an activity recognition algorithm. Each segment mismatch is labeled with the type of error that is represented, either an overfill (O), underfill (U), insertion (I), deletion (D), merge (M), or fragmenting (F).

Overfill. A false positive that occurs at the start or end of a segment that is partially matched is considered an overfill. This could occur at a segment boundary, for example, when the recognition algorithm is viewing a transition and has not received enough information to successfully detect the new activity. This is a possible explanation for the first overfill in Figure 5.11. The second overfill is found at the end of an activity and may occur when the activity recognition anticipates a new activity (perhaps based on the time of day) before it actually occurs.

Insertion. A false positive that represents an inserted positive label between two negative segments is considered an insertion. As seen in the example, the false positive segment due to insertion is not an extension of a true positive that is simply too long (as with overfill), but is an actual incorrectly-inserted positive label.

Merge. A merge is a false positive portion of a segment. The surrounding sensor events are correctly labeled with the positive class but the change in the middle of the segment to a different activity was not sensed by the AR algorithm.

Underfill. Each of the previous three error classes has a corresponding type that can be used for analyzing false negatives. In the same way that an overfill error corresponds to a positive segment that extends too far, an underfill error occurs when a positive segment is too short because the beginning or ending of the segment is incorrectly labeled as the negative class. In our example the fourth occurrence of a hand washing-labeled segment is too short, which results in an underfill error on either side of the segment.

Fragmenting. A fragmenting false negative occurs between two true positives in a single consecutive positive segment.

Deletion. The deletion error occurs when a false negative occurs between two true negatives. In our example, the AR algorithm combines sensor events into a single negative example because it failed to detect a switch to the positive class in the middle of the subsequence.

Once the correctly-labeled and incorrectly-labeled segments are identified and characterized, they can be used to generate rates or percentages for correct labels (C) and for each type of error for the sensor event sequence. Our example in Figure 5.9 has 22 sensor events total, so the O, U, F, and D rates are 2/22 = 0.090 while the I and M rates are 1/22 = 0.045. The error rates are fairly constant for each type of error in this artificial example. In realistic cases they can provide insights on whether the errors are due to slow transitions to new activities, failure to detect short, quick activities, or other types of problems with activity recognition.

Timeliness. A related sequence-based performance measure is the timeliness of activity recognition. Timeliness applies to a particular occurrence of an activity and is measured as the relative point in time for the activity at which the activity recognition algorithm correctly provides the correct label and does not switch to an inaccurate label for the remainder of the activity. In Figure 5.11, there are 11 distinct activities that are observed and the point in time at which a correct label is generated and not changed is indicated by a star (*). As can be seen in this figure, the first activity is not correctly predicted and held stable unless the fifth event of the sequence, so the timeliness is 0.2. In contrast, the last activity is correctly labeled from its onset, so the timeliness is 1.0. An activity recognition algorithm may need sufficient context to recognition the current activity, and the corresponding delay in outputting a correct label is captured by this performance metric.

5.4.3 Experimental frameworks for evaluating activity recognition

The next question is how to choose data on which the supervised learning algorithm will be tested. Because we want the learned model to generalize beyond the data it has already seen and correctly classify new data points that it has not previously seen, the model is usually trained on one set of data and tested on a separate, "holdout" set of data. In the context of activity recognition, the question is how to select subsets of available data for training and testing. Here we describe two techniques that are commonly applied to this process.

The first method is *k*-fold cross validation. This method is effective when the amount of labeled data is limited because it allows all of the data points to play a role both in training and testing the learned model. With this approach, the set of data points is split into *k* non-overlapping subsets. The model is trained and tested *k* times and the performance is averaged over the *k* iterations. On each iteration, one of the *k* partitions is held out for testing and the other *k-1* partitions are used to train the model. The choice of *k* varies, common choices are 3-fold (this is particularly common when there are few data points available because each partition should ideally have at least 30 data points) and 10-fold cross validation. The results reported in the confusion matrices of Table 5.2 and 5.3 and the performance evaluation of Table 5.4 are based on a 3-fold cross validation of the decision stump algorithm for the hand washing and sweeping activities.

While cross validation is a popular validation technique for machine learning algorithms, its use is trickier when applied to sequential data. This is because the long contiguous sequence must be separated into individual data points and subsets of data points. As a result, some of the context is lost that may be captured by some algorithms when training the model. When activities are pre-segmented, the segments can represent individual data points. When a sliding window approach is used, individual windows represent the data points. Some alternative selections are to separate the datasets by time boundaries such as hours, days, or weeks in order to retain much of the sequential relationships. This method also allows the activity recognition algorithm to be tested for its ability to generalize to new days, weeks, or months. When the recognition algorithm is trained to generalize over multiple users or physical settings, the individual users or environments can be treated as separate data points for testing. Finally, leave-one-out testing can be used to train the data on contiguous sequential data and test it on held-out data from the end of the sequence. The length of the training sequence can iteratively increase so that eventually all of the available data is used for both training and testing.

While most methods choose a holdout set for testing either through random selection or at the end of a sequence, holdout selection can also be performed strategically to demonstrate the generalizability of the activity recognition algorithm over selected dimensions. For example, an algorithm can be trained over multiple individual people or homes, selecting one person or home as the holdout set. Similarly, an entire activity can be held out to determine how the algorithm performs on a previously-unseen activity class.

Additionally, a need may arise to compare the performance of two alternative activity recognition algorithms. If minimizing error rate (or conversely, maximizing accuracy) of activity classification is considered as the performance metric of interest, then two alternative approaches f^1 and f^2 can be compared by, for example, looking at the mean error found through cross validation. However, because cross validation testing is performed only on a subset of the possible data points in the space, this comparison may not be a strong indicator of how the two approaches will compare in general.

Statistical tests such as the Student's t-test can be used to determine whether a set of performance differences between f^1 and f^2, such as generated through k-fold cross validation, is significant. To perform this computation from cross-validation results let $f_1^1, f_2^1, .., f_k^1$ represent the set of results for f^1, $f_1^2, f_2^2, .., f_k^2$ represent the set of values for f^2, and $d_1, d_2, .., d_k$ represent the differences between the results (i.e., $d_i = f_i^2 - f_i^2$). We are then trying to determine if the mean $\overline{f^1}$ is significantly different from the mean $\overline{f^2}$. This calculation assumes that the values for f^1, f^2, and *d* follows a Student's distribution, which approximates a normal distribution as *k* becomes large. We first reduce the difference to a zero-mean, unit-variable variable *t* as a function of the difference mean, number of folds, and difference variance σ_d^2 as shown in Equation 5.20.

$$t = \frac{\overline{d}}{\sqrt{\sigma_d^2 / k}} \tag{5.20}$$

If $t < -z$ or $t > z$, where *z* represents a confidence limit, then the difference in performance between the two approaches can be termed significant at the corresponding confidence level. The values of *z* are determined by the Student's distribution and the number of folds *k* (or, corresponding, the degrees of freedom *k-1*). For example, if *k=10* and we want to reject the hypothesis that there is no difference in performance between the approaches with probability p<.05 (a common threshold used to report statistical significance), then *z=2.262*.

5.5 Additional Reading

Ali and Aggarwal[71] learn activity breakpoints from video data using a supervised learning technique that is provided with activity begin and end frames for each activity class. Ho and Intille[72] and Feuz et al.[73] use supervised learning to recognize transitions between specific activity pairs. Iqbal and Bailey[74] use a multilayer perceptron to recognize and categorize breakpoints between any type of computer-related task. Niu and Abdel-Mottaleb[75] describe a method of activity segmentation that relies on rejecting sequences not clearly belonging to any one activity. Somewhat related is the idea of co-segmentation of event sequences introduced by Duchenne et al.[76] Co-segmentation is applied to two sequences that are known to contain the same activity of interest. The fact that they share the activity facilitates the process of identifying the activity boundaries within both sequences. The idea of employing an ensemble of classifiers to recognize activities from different window sizes is introduced by Zheng et al.[77]. Varying the size of a sliding window based on activity likelihood and relevant sensors has been explored by Krishnan and Cook[78] and by Okeyo et al[79]. A thorough treatment of one-class classifiers and their uses for outlier and anomaly detection is provided by Khan and Madden[80].

Hong and Nugent[81] introduce an activity segmentation method that learns the relationship between activities, sensors, and locations, and then uses changes in these parameters to identify activity boundaries. In addition, Gu et al.[82,83] propose the idea of identifying activity boundaries based on the difference between accumulated sensor relevance. In their case, the sensors were object sensors and weights were calculated based on the discriminatory relevance of each object to each activity. Yamasaki[84] takes a similar approach to identifying activity boundaries based on differences in activity signatures from video data. This approach pinpoints activity boundaries based on maximizing the difference between feature vectors describing successive frames in the video. Keogh et al.[85] propose the idea of combining sliding windows with bottom-up segmentation in their SWAB algorithm to provide real-time segmentation with improved performance over a pure sliding window approach.

The Bayesian online change point detection is based on work by Adams and MacKay[86]. An alternative approach to online change point detection is to directly estimate the ratio of the probability density functions before and after candidate change points. Direct density ratio estimation is useful when the individual probability distributions are unknown[87]. Techniques to perform this include kernel mean matching, logistic regression, Kullback-Leibler importance estimation, unconstrained least-squares importance fitting, and relative unconstrained least-squares importance fitting (RuLSIF).[88] Other changepoint detection variations were introduced by Guenterberg et al.[89], who identify an activity boundary when the differences in spectral energy (defined in Chapter 3) before and after the boundary point are greater than a threshold value. Feuz et al.[73] use change point detection to identify activity boundaries as a time to provide prompt-based interventions. Xie and al.[90] introduce a new variation on change point detection that scales well to high-dimensional data.

A number of survey articles exist which provide an excellent overview of activity recognition algorithms for different classes of sensors and activities. Many of the articles detail the algorithms and summarize achieved recognition performance on simulated and real-world datasets[30,91–99]. Activity recognition using NBCs has been explored by Cook[100] using smart home sensor data and by Bao et al.[35] and Ravi et al.[101] for accelerometer data. Popular methods for activity recognition also include hidden Markov models[102–105] and conditional random fields[41,100,106]. Other have found SVMs[107] and decision trees[35] to be effective. Many methods have been explored which combine these underlying learning algorithms, including boosting and other ensemble methods[101,108,109].

While activity recognition focuses on labeling activities, other work focuses on evaluating the quality of the activity that was performed[110–112]. Ogris et al.[113] and Amft[114] describe the problem of activity spotting and propose alternative approaches using data from wearable sensors.

Ward et al.[115] and Bulling et al.[93] introduce a number of performance metrics that fall into the categories of time-based and event-based evaluation of activity recognition algorithms. Reiss et al.[94] discuss the effectiveness of hold-one-activity-out evaluation of activity recognition algorithms for generalizability assessment. The notion of the timeliness of activity recognition was introduced by Ross and Kelleher[116].

Di Eugenio and Class[117] provide a good introduction to the kappa statistic and alternative ways to calculate P(E) in calculating the kappa statistic. Davis and Goadrich[118] provide a useful discussion of the relationship between ROC and PRC curves and how to compute the area under the curves. Additional methods for evaluating and comparing the performance of supervised learning algorithms are presented by Cohen[119] and by Dietterich[120]. Nuzzo[121] offers an interesting discussion of the limits of p values for determining statistical significance and the need to also consider effect sizes (in the discussion of this chapter, the effect size would be the amount of difference in performance between alternative activity recognition algorithms).