Time Series Classification under More Realistic Assumptions

Bing Hu Yanping Chen Eamonn Keogh
University of California, Riverside
{bhu002, ychen053}@ucr.edu , eamonn@cs.ucr.edu

ABSTRACT

Most literature on time series classification assumes that the beginning and ending points of the pattern of interest can be correctly identified, both during the training phase and later deployment. In this work, we argue that this assumption is unjustified, and this has in many cases led to unwarranted optimism about the performance of the proposed algorithms. As we shall show, the task of correctly *extracting* individual gait cycles, heartbeats, gestures, behaviors, etc., is generally much more difficult than the task of actually *classifying* those patterns. We propose to mitigate this problem by introducing an alignment-free time series classification framework. The framework requires only very weakly annotated data, such as "*in this ten minutes of data, we see mostly normal heartbeats...*," and by generalizing the classic machine learning idea of *data editing* to streaming/continuous data, allows us to build robust, fast and accurate classifiers.

We demonstrate on several diverse real-world problems that beyond removing unwarranted assumptions and requiring essentially no human intervention, our framework is both significantly faster and significantly more accurate than current state-of-the-art approaches.

1. INTRODUCTION

In virtually all time series classification research, long time series are processed into short equal-length "template" sequences that are representative of the class. For example, individual and complete gait cycles for biometric classification[1][7][11][16], individual and complete heartbeats for cardiological classification [5][12], individual and complete gestures for gesture recognition [37], etc.

In most cases, the segmentation of long time series into these idealized snippets is done by hand [7][11][16][17]. However, for many real-world problems this either cannot be done, or only done with great effort [10][18][23].

As a concrete example, consider the famous Gun/Point problem [13][30], which has appeared in at least one hundred works [6][14][20]. To create this dataset, the original authors [29][30] used a metronome that signaled every three seconds to cue both the actor's behavior and the start/stop of the recording apparatus [29]. This allowed the extraction of perfectly aligned data, containing *all* of the target behavior and *only* the target behavior. Unsurprisingly, dozens of papers report less than 10% classification error rate on this problem. However, does such an error rate reflect our abilities with real-world data?

Such contriving of time series datasets seems to be the norm. For example, [37] notes, "*one subject performed one trial of an action* (in exactly) *10 seconds*." and [22] tells us that human editors should carefully discard "*all transient activities between performing different activities.*" Likewise, a recent paper states: "*We assume that the trajectories are segmented in time such that the first and last frames are already aligned* (and) *the resulting model has the same length*" [34]. Note that these authors are to be commended for stating their assumptions so concretely. In many cases, no such statements are made, but we suspect that similar "massaging" of the data has occurred.

We believe that such contriving of the data has led to unwarranted optimism about how well we can classify real-time series data streams. For real-world problems, we cannot always expect the *training* data to be so idealized, and we certainly cannot expect the *testing* data to be so perfect.

A more realistic idea for data gathering is to capture data "in the wild" as in [2][25][31], etc. However, this opens the problem of data editing and cleaning. For example, a one-hour trace of data labeled "*walking*" will almost certainly contain non-representative subsequences, such as the subject pausing at a crosswalk, or introducing a temporary asymmetry into her gait as she answers her phone. The current solution to preprocess such data requires human intervention to examine and edit such traces, and keeping data that demonstrates the sought-after variability (walking `uphill`, `downhill`, `level`, walking `fast`, `normal`, `slow`), while discarding data that is atypical of the class.

Moreover, in virtually all time series classification research, the data must be arranged to have equal length [34]. For example, in the world's largest collection of time series datasets, the UCR classification archive, all forty-five time series datasets contain *only* equal-length data [13].

Finally, most of the literature assumes that all objects to be classified belong to exactly one of two or more well-defined classes. For example, in the Gun/Point problem, every one of the instances is *either* a gun-aiming *or* a finger-pointing (unarmed) behavior. However, the vast majority of normal human actions are clearly neither. How well do current techniques work when most of the data is *not* from the well-defined classes?

To summarize, much of the progress in time series classification from streams in the last decade is almost certainly optimistic, given that most of the literature implicitly or explicitly assumes one or more of the following:

1. Copious amounts of perfectly aligned *atomic* patterns can be obtained [11][35][37].
2. The patterns are all of *equal length* [11][13][16][23][31].
3. Every item that we attempt to classify belongs to *exactly one* of our well-defined classes [10][13][23][30].

In this work, we demonstrate a time series classification framework that does not make *any* of these assumptions.

Our approach requires only very *weakly-labeled* data, such as "*This ten-minute trace of ECG data consists mostly of arrhythmias, and that three-minute trace seems mostly free of them*", removing assumption (1). Using this data we automatically build a "data dictionary", which contains only the minimal subset of the original data to span the concept space. This is because the data dictionary can contain, say, one example of walking `fast`, one example of walking `normal`, etc. This mitigates assumption (2).

As a byproduct of building this data dictionary, we learn a *rejection threshold*, which allows us to address assumption (3). A query item further than this threshold to its nearest neighbor is assumed to be in the `other` class. Finally, we show that using the *Uniform Scaling* distance measure [15] instead of *Euclidean* distance also addresses assumption (2).

The rest of this paper is organized as follows: In Section 2, we introduce definitions and notation used in this paper. In Section 3.1, we show how classification is achieved with our data dictionary model. In Section 3.2, we illustrate how to actually *learn* the data dictionary by utilizing data editing techniques [21][28][33][36]. In Section 4, we present a detailed empirical evaluation of our ideas. We discuss related work in Section 5. Finally, in Section 6 we offer conclusions and directions for future work.

2. DEFINITIONS AND NOTATION

We begin with the definition of *time series*:

Definition 1: *Time Series*: T = $t_1,\ldots t_m$ is an ordered set of *m* real-valued variables.

We are only interested in *local* properties of a time series, thus we confine our interest to *subsequences*:

Definition 2: *Subsequence*: Given a time series T of length *m*, a subsequence S_k of T is a sampling of length $n \leq m$ of contiguous position from T with starting position at *k*, $S_k = t_k,\ldots t_{k+n-1}$ for $1 \leq k \leq m\text{-}n+1$.

The extraction of subsequences from a time series can be achieved by use of a *sliding window*:

Definition 3: *Sliding Window*: Given a time series T of length *m*, and a user-defined subsequence length of *n*, all possible subsequences can be extracted by sliding a window of size *n* across T and extracting each subsequence, S_k. For a time series T with length *m*, the number of all possible subsequences of length *n* is *m*-*n*+1.

For concreteness, we take the step of explicitly defining training data, as our definition of *training data* explicitly removes the assumptions inherent in most works [7][11][13][16][23][31][34].

Definition 4: *Training Data*: A *Training Data* C is a collection of the *weakly-labeled* time series annotated by behavior/state or some other mapping to the ground truth.

By *weakly-labeled* we simply mean that each long data sequence has a single global label and not lots of local labeled pointers to every beginning and ending of individual patterns, e.g., individual gestures. There are two important properties of such data that we must consider:

- *Weakly-labeled* training data may contain *extraneous/irrelevant sections*. For example, after a subject reaches down to turn on an ankle sensor to record her gait, there may be a few seconds before she actually begins to walk [31]. Moreover, during the recording session, the subject may pause to shop, or jump to avoid a puddle. It seems very unlikely that such recordings could *avoid* having such spurious data. Note that this claim is not mere speculation; we observed this phenomenon in the first few seconds of the BIDMC Congestive Heart Failure dataset [3] as shown in Figure 1, and similar phenomena occur in all the datasets we examined.
- *Weakly-labeled* training data will almost certainly contain significant *redundancies*. While we want lots of data in order to learn the inherent variability of the concept we wish to learn, significant redundancy will make our classification algorithms slow when deployed. Consider Figure 1 once more. Once we have a single normal heartbeat, say pattern **A**, then there is little utility in adding any of the 14 or so other very similar patterns, including pattern **B**. However, to robustly learn this concept (beats belonging to Record-08), we must add *either* example of the Premature Ventricular Contraction (PVC).

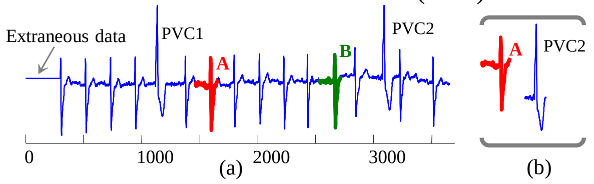

Figure 1: A snippet of BIDMC Congestive Heart Failure Database ECG - Record-08 [3]. (a) is weakly-labeled data, which exhibits both *extraneous* data, a section of recording when the machine was not plugged in, and *redundant* data (only one pair of redundancies are shown in bold (red/green). (b) A minimally redundant set of representative heartbeats (a *data dictionary*) could be used as training data

Rather than these large *weakly-labeled* training datasets, we desire a smaller "smart" training data subset that does not contain *spurious* data, while maintaining coverage of the target concept by having one (ideally, *exactly* one) instance of each of the many ways the targeted behavior is manifest. For example, from the training data shown in Figure 1, we want just one PVC example and just one example of a normal heartbeat (perhaps *either* **A** *or* **B**). However, we do not want to require costly human effort to obtain this. While the time series shown in Figure 1 would be fairly easy to edit by hand, it is only 0.16% of the full ECG dataset we consider in Section 4. Therefore, our objective is to build this idealized subset of the training data automatically. We begin by defining it more concretely as a *data dictionary*.

Definition 5: A *Data Dictionary* **D** is a (potentially very small) "smart" subset of the training data. We allow an input parameter *x*, where *x* is the percentage of the training data C used in data dictionary **D**. The range of *x* is (0,100%], and a dictionary with the percentage *x* of the original data is denoted as $\mathbf{D}_x$.

As the *Data Dictionary* is at the heart of our contribution, we will take the time to discuss it in detail.

2.1 A Discussion of Data Dictionaries

As defined above, there are a huge number of possible data dictionaries for any percentage *x*, as any random subset of C satisfies the definition. However, we obviously wish to create one with some desirable properties.

Clearly, the classification error rate obtained from using just **D** should be no worse than that obtained from using all the training data. We do not wish to sacrifice accuracy. As we shall show, this is a surprisingly easy objective to achieve. In fact, as we shall show later, the classification error rate using a judiciously chosen **D** is generally significantly lower than using all of C. This is because the data dictionary contains less spurious –and therefore, potentially *misleading*–data.

Another desirable property of **D** is that it be a very small percentage of the training data. This is to allow real-time deployment of the classifier, especially on resource limited devices (embedded devices, smartphones, etc. [2][9]). This requirement may be seen as conflicting with the above classification *error rate* requirement; however, again we will show that in most real-world problems we can judiciously throw away more than 95% of C to obtain a $\mathbf{D}_{5\%}$ that is *at least* as accurate as using all the data in C.

Note that the number of subsequences within each class in **D** may be different. That is to say, our algorithm for building **D** is *not* round-robin; rather the algorithm adaptively adds more subsequences to cover the more "complicated" classes of **D**. For example, the ECG data from Record-08 shown in Figure 1 is relatively simple. In contrast, the ECG of Record-03 shown in Figure 2 has a more complicated trace, and at least four kinds of beats (normal, S, PVC and Q). Therefore, we might expect the number of subsequences for Record-03 in **D** to be greater than that for Record-08, something that is empirically borne out in our experiments (Section 4).

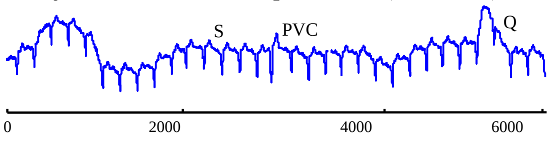

Figure 2: A snippet of BIDMC Congestive Heart Failure Database ECG: Record-03 [3]. Note that this section of ECG data exhibits more variability than the data in Figure 1

Finally, there is the question of what value we should set x to. In fact, we can largely bypass this issue by providing an algorithm that produces a "spectrum" of data dictionaries in the range of x = (0,100%], together with an estimate of their error rate on unseen data. The user can examine this error rate vs. value-of-x curve to make the necessary trade-offs. Note that these data dictionaries are "nested", that is to say, for any value of x we have $\mathbf{D}_x \subseteq \mathbf{D}_{x+\varepsilon}$. Thus, we can consider our data dictionary creation algorithm an *anyspace* algorithm [36].

Given the above considerations, how can we build the best data dictionary? As we will later show, we can heuristically search the space of data dictionaries using the simple algorithm in Section 3.2.

2.2 An Additional Insight On Data Redundancy

Based on our experience with real-world time series problems, we noted the following: in many cases, **D** contains many patterns that appear to be simply (linearly) rescaled versions of each other. For clarity, we illustrate our point with a synthetic example in Figure 3; however, we will later show some real examples.

This situation is a consequence of our requirement that data dictionary **D** has the most representative subsequences of training data C. For example, if one class contains examples of `walk`, we hope to have at least one representative of each type of `walk`—perhaps one example of a `leisurely-amble`, one example of a `normal-paced-walk`, one example of a `brisk-walk`, etc. It is important to note that in this example, the three walking styles are *not* simply linearly rescaled versions of each other. They have different foot strike patterns, and thus produce different prototypical time series templates [4][19]. Nevertheless, *within* each sub-class of `walk`, there may also be a need to allow some linear rescaling of the time series. Using the *Euclidean* distance our search algorithm can achieve this by attempting to ensure that the data dictionary contains each gait pattern over a range of speeds. This is what our toy example in Figure 3 illustrates.

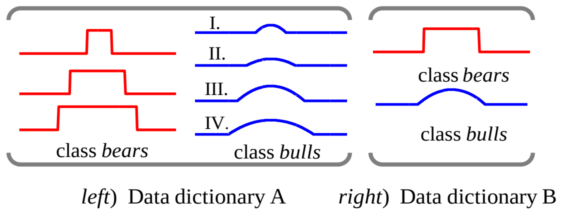

Figure 3: *left*) A toy example data dictionary which was condensed from a large dataset. These seven subsequences in data dictionary A span the concept space of the *bulls/bears* problem. *right)* Note that if we had a distance measure that was invariant to linear scaling, we could further reduce data dictionary A to data dictionary B

For example, when reducing a dataset of daily human activities, we may have to extract examples of a `brisk-walk` at 6.0km/h, 6.1km/h, 6.2km/h, etc. However, by generalizing from the *Euclidean* distance to the *Uniform Scaling* distance [15], we allow our algorithm to keep just one example of the `walk`, and *still* achieve coverage of the target concept by using a flexible measure *instead* of lots of data. The *Uniform Scaling* distance is a simple generalization of the *Euclidean* distance that allows limited invariance of the length of the patterns being matched [15]. The maximum amount of linear scaling allowed is a user-defined parameter [15]. As we later show, allowing just a small amount of scaling, say 25%, can greatly improve accuracy.

To see this in a real dataset, consider Figure 4.*left* which shows one of fifteen classes that was processed into a data dictionary in an experiment we performed in Section 4.2. At first glance, the two patterns seem redundant[1], violating one of the requirements stated above.

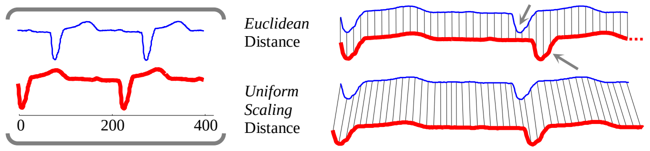

Figure 4: *left*) A data dictionary learned from a 15-class ECG classification problem (just class 01 is shown here). At first glance, the two exemplars seem redundant apart from their (irrelevant) phases. *right*) By using the *Euclidean* distance between the two patterns we can see that the misalignment of the beats would cause a large error. The problem solved by using the *Uniform Scaling* distance [15]

Instead of having two similar but different scaled patterns, just a single pattern is kept using the *Uniform Scaling* distance. We have found that using the *Uniform Scaling* distance allows us to have a significantly smaller data dictionary. In Figure 4, we could delete either one of the two

[1] Note the fact that the two patterns are out of phase does not make them non-redundant, as at query time only queries half their length are used, and they are sliding across the entire length of the patterns. Details in Section 4.2.

patterns and cover the space of possible heartbeats from Record-01. For example, in Figure 3, we could further delete patterns I, II and IV and still cover the space of possible "*bulls*".

However, beyond reducing the size of data dictionaries (thus speeding up classification), there is an additional advantage of using *Uniform Scaling*; it allows us to achieve a lower error rate. How is this possible? It is possible because we can generalize to patterns *not seen* in the training data.

Imagine the training data does contain some examples of gaits at speeds from 6.1 to 6.5km$_{/h}$. As noted above, if the data dictionary has enough examples to cover this range of speeds, we should expect to do well. However, suppose the unseen data contains some walking at 6.7km$_{/h}$. This is only slightly faster than we have seen in the training data, but the *Euclidean* distance is very sensitive to such changes [15]. Using the *Uniform Scaling* distance allows us to generalize our labeled example at 6.5km$_{/h}$ to the brisker 6.7km$_{/h}$ instance. This idea is more than speculation. As we show in Section 4, using the *Uniform Scaling* distance does produce a significantly lower error rate.

2.3 On the Need for a Threshold

As noted above, the training set may have extraneous data. Likewise, in most realistic deployment scenarios, we expect some (often *most*) of the data to be classified as the `other` class. In these cases, we wish our algorithm to label the objects as such. To achieve this, the data dictionary must have a distance threshold `r` beyond which we reject the query as unclassifiable (i.e., the `other` class). As we will show, we can learn this threshold as we build the dictionary.

3. ALGORITHMS

In order to best explain our framework, we first assume a *data dictionary* with the appropriate threshold has already been created and begin by explaining how our *classification* model works. Later, in Section 3.2, we revisit the more difficult task of *learning* the data dictionary.

3.1 Classification using a Data Dictionary

Our classification model requires just a data dictionary with its accompanying threshold distance, `r`.

For an incoming object to be classified `q`, we classify it with the data dictionary using the classic nearest neighbor algorithm [33]. In Table 1, we show how to determine the class membership of this query, including the possibility that this query does not belong to any class in this data dictionary. For our purposes, there are exactly two possibilities of interest:

- If the query's nearest neighbor distance is larger than the threshold distance, we say this query does not belong to any class in this data dictionary (line 12).
- If the query's nearest neighbor distance is smaller than the threshold distance, then it is assigned to the same class as its nearest neighbor (line 14).

The algorithm begins by initializing the `bsf` distance to infinity and the predicted `class_label` to `NaN` in lines 1 and 2. From lines 3 to 9, we find the nearest neighbor of the query `q` in data dictionary **D**. The subroutine `NN_search` (shown in Table 2) returns the nearest neighbor distance of `q` within a time series. If the nearest neighbor distance within a time series in line 4 is smaller than the `bsf`, then in lines 6 and 7 we update the `bsf` and the `class_label`.

Table 1: Classification Algorithm using Data Dictionary

```
Input:   D, a data dictionary that has N classes; The
         total number of time series in D is k
         r, a threshold distance of D
         q, a query
Output:  The class membership of q, including the
         possibility of a special class 'other'

1    bsf = ∞;    //initialize the best-so-far distance
2    class_label = NaN;
3    for i = 1 to k
4        dist = NN_search(q, D(i));
5        if dist < bsf
6            bsf = dist;
7            class_label = class of D(i);
8        endif
9    endfor
10   NN_dist = bsf;
11   if  NN_dist  > r
12     return q belongs to 'other' class;
13   elseif NN_dist <= r
14     return q belongs to 'class_label'th class;
15   endif
```

From lines 11 to 15, we compare the nearest neighbor distance to the threshold distance `r`. If the nearest neighbor distance is smaller than `r`, then this query belongs to the same class as its nearest neighbor. Otherwise, this query does not belong to any class within this data dictionary and is thus classified as the `other` class.

As we show in Table 1 line 4, the function `NN_search` is slightly different from the classic nearest neighbor search algorithm [13]. `NN_search` returns not only the nearest neighbor distance of a query, but also a distance vector that contains distances between the query and *all* the possible subsequences in a time series. This distance vector is not exploited at classification time, but as we show in Section 3.2, it is exploited when building the data dictionary. For concreteness, we briefly discuss the `NN_search` function in Table 2 below.

Table 2: Nearest Neighbor Search within a Time Series

```
Input:   q, a query                T, a time series
Output:  dist_vector,a vector that contains distances
         between q and all possible subsequences in T
         NN_dist, the nearest neighbor distance

1    w = set of all possible subsequences in T;
2    dist_vector = zeros(1,|w|);
3    for i = 1 to |w|
4          dist_vector(i) = distance(q,w(i));
5    endfor
6    NN_dist = minimum(dist_vector);
7    return dist_vector ;
8    return NN_dist ;
```

In line 1, using a sliding window (cf. Definition 3), we extract all the subsequences of the same length as the query. From lines 3 to 5, the distances between `q` and all the possible subsequences are calculated. We calculate the nearest neighbor distance in line 6. Note that in line 4, the distance could be *Euclidean* distance [13], or *Uniform Scaling* distance [15], etc. We will revisit this choice in Section 4.

In addition to finding the nearest neighbor, this function also returns a distance vector. This additional information is exploited by the dictionary building algorithm discussed later in Section 3.2. Figure 5.*bottom* shows an example of such a distance vector.

Having demonstrated how the classification model works in conjuction with the data dictionary, we are in position to illustrate how to actually build the data dictionary, which is a more difficult task.

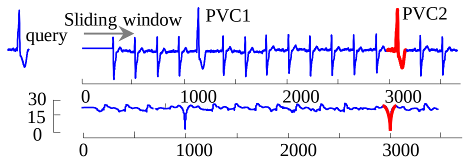

Figure 5: *top*) A snippet of BIDMC Congestive Heart Failure Database ECG data: Record-08 [3]. *bottom*) the distance vector of an incoming query.The nearest neighbor and its distance of q is colored in red/bold

3.2 Building the Data Dictionary

As discussed in Section 2, we want to build the data dictionary automatically. Using human effort to *manually* edit the training data into a data dictionary is clearly not a realistic solution: as it is not scalable to large datasets and invites human bias into the process.

Before introducing our dictionary building algorithm, we will show a worked example on a toy dataset in the *discrete* domain. We use a small discrete domain example simply because is it easy to write intuitively; our real goal remains large real-valued time series data.

1) *The intuition behind data dictionary building*

Suppose we have a training dataset that contains two classes, C1 and C2:

C1 = { d**pace**kfjkl**walk**fl**walk**kl**pace**dalyutek**walk**sfj}
C2 = { jhjh**leap**ashl**jump**okdjkl**leap**hf**leap**fj**jump**acgd}

In this toy example, the data is *weakly-labeled*. The colored/ bolded text is for the reader's introspection only; it is *not* available to the algorithm. Here the reader can see that in C1, there appears to be two ways a shorter subsequence query might belong to this class; if it contains the word *pace* or *walk*. This is similar to the situation shown in Figure 1 where a query will be classified to the class of Record-08 if it contains pattern **A** or pattern PVC.

We want to know whether any incoming queries belong to either class in this training data or not. In our proposed framework, we search *just* the data dictionary.

Recall that one of the desired properties of the data dictionary is that it contains a minimally redundant set of patterns that is representative of the training data. In this example for C1, these are clearly the substrings *pace* and *walk*. Likewise for C2, *leap* and *jump* seem to completely define the class. Thus, the data dictionary **D** should be the following:

D = C1:{ *pace* ; *walk* } C2: { *leap* ; *jump*}, r = 1

Consider now two incoming queries *ieap* and *kklp*. The former is a noisy version of a pattern found in our dictionary, but as it is within our rejection threshold of (hamming) distance r of 1, it is correctly labeled as C2. In contrast *kklp* has a distance of 3 to its nearest neighbor in **D**, so it is correctly rejected.

Note that had we attempted to classify against the raw data rather than the dictionary, the query *kklp* would have been classified as C1 (it appears in the middle of ..l**wal**kkl**pace**d.). This misclassification is clearly contrived, but it *does* happen frequently in the real data. Consider the flat section of time series at the beginning of Figure 1. As noted above, it is *extraneous* data, due to a temporary disconnection of the sensor. However, many other patients' ECG traces also have these flat sections, but clearly that does not mean we should classify them as belonging to patient Record-08.

In our example, we have considered two separate queries; however a closer analogue of our real-valued problem is to imagine an endless stream that needs to be classified:

.. ttgpacedgrteweerjumpwalkflqrafertwqhafhfahfahfbseew..

Up to this point we have not explained how we built our toy dictionary. The answer is simply to use the results of leaving-one-out classification to score candidate substrings. For example, by using leaving-one-out to classify the first substring of length 4 in C1 *dpac*, it is incorrectly classified as C2 (it matches the middle of ..**ump**acgd.. with a distance of 1). In contrast, when we attempt to classify the second substring of length 4 in C1, *pace*, we find it is correctly classified. By collecting statistics about which substrings are often used for correct predictions, but rarely used for wrong predictions, we find that the four substrings shown in our data dictionary emerge as the obvious choices. This basic idea is known as *data editing* [21][24][33]. In the next section, we formalize this idea, and generalize it to *real*-valued data streams.

2) *Building the data dictionary*

The high-level intuition behind building the data dictionary is to use a ranking function to score every subsequence in C. These "scores" rate the subsequences by their *expected utility* for classification of future unseen data. We use these scores to guide a greedy search algorithm, which iteratively selects the best subsequence and places it in **D**. How do we know this utility? We simply estimate it by cross validation, e.g. looking at the classification error rate and some additional information as explained below.

As previously hinted, our algorithm iteratively adds subsequences to the data dictionary. Each iteration has three steps. In Step 1, the algorithm scores the subsequences in C. In Step 2, the highest scoring subsequence is extracted and placed in **D**. Finally, in Step 3, we identify all the queries that cannot be correctly classified by the current **D**. These incorrectly classified items are passed back to Step 1 to re-score the subsequences in C.

There is an important caveat. Once we have removed the best subsequence in Step 2, the scores of all the other subsequences may change in the next iteration. To return to our running example in Figure 1, *either* subsequence **A** and **B** would rank highly. However once we have placed one, say **A**, in **D**, there is little utility in adding **B**, since having **A** in **D** is sufficient to correctly classify similar patterns in Step 3. Thus we expect the scores of **B** will be low in the next iteration, given that the correctly classified queries by the current **D** will not be used to re-score C in the next iteration.

The process iterates until we run out of subsequences to add to **D** or the unlikely event of *perfect* training error rate having been achieved. In the dozens of problems we have considered, the training error rate plateaus well before 10% of the training data has been added to the data dictionary.

Below we consider each step in detail.

Step 1 : In order to rank every point in the time series, we use the leaving-one-out classification algorithm[2]. However, we do not want to use *just* the classification error rate to score the subsequences. Imagine we have two subsequences S_1 and S_2, either of which is found to correctly predict 70% of the queries tested with them. Either appears to be a good candidate to add to **D**. However, suppose that in addition to being close enough to many objects with the *same* class label (*friends*), allowing its 30% error rate, further suppose that S_1 is also very close to many objects with *different* class labels (*enemies*). If S_2 keeps a larger distance from its enemy class objects, it is a much better choice for inclusion in **D**.

This idea, that instead of using just the error rate of classification, you must also consider the relative distance to "*friends*" and "*enemies*" has been investigated extensively in the field of data editing [24][33].

Given a query length *l*, we randomly choose a query `q` from the training data C[3]. In Table 3, lines 2 and 3, we first split the training data into two parts, Part A (*friends* only) and Part B (*enemies* only). Using the `NN_search` algorithm in Table 2, we find *nearest neighbor friend* in Part A (lines 5 to 13) and *nearest neighbor enemy* (lines 14 to 22) in Part B.

In lines 23 to 27, the nearest neighbor *friend* distance and the nearest neighbor *enemy* distance are compared. If the nearest neighbor *friend* distance is smaller than the nearest neighbor *enemy* distance, we discover all the distances of the query `q` in Part A that are also smaller than the *nearest neighbor enemy* distance. Such subsequences are *likely true positives*. That is to say, our confidence that these subsequences can produce correct classifications of unseen data has increased.

Similarly, if the nearest neighbor *friend* distance is larger than the nearest neighbor *enemy* distance, we find all the distances of the query `q` in Part B that are also smaller than the *nearest neighbor friend* distance. We call the corresponding subsequences *likely false positives*.

Table 3: Classification of Training Data

```
Input:    C, the training data
Output:   likely true/false positive subsequences

 1   q = a randomly selected subsequence in C;
 2   A = friends ;   //all the time series in C that have the same class as q, q is removed from A;
 3   B = enemies ;  // all the time series in C that have different class from  q;
 4   dists_A = [];  dists_B = [];
 5   bsf = ∞;             //initialize the best-so-far distance
 6   for i = 1 to |A|
 7   [dist_vector, NN_dist] = NN_search(q, A(i));
 8   if NN_dist < bsf
 9      bsf = NN_dist;
10   endif
11   dists_A = [dists_A ; dist_vector];
12   endfor
13   NN_friend_dist = bsf;   // nearest neighbor distance in same class
14   bsf = ∞;                  //initialize the best-so-far distance
15   for j = 1 to |B|
16   [dist_vector, NN_dist] = NN_search(q, B(j));
17   if NN_dist < bsf
18      bsf = NN_dist;
19   endif
20   dists_B = [dists_B ; dist_vector];
21   endfor
22   NN_enemy_dist = bsf;  // nearest neighbor distance in different class
23   if NN_friend_dist < NN_enemy_dist
24   likely_true_positives = find(dists_A < NN_enemy_dist)
25   elseif NN_friend_dist >= NN_enemy_dist
26   likely_false_positives = find(dists_B < NN_friend_dist)
27   endif
```

[2] Where tractably is an issue, we may sample a subset of the queries.

[3] We defer the discussion on how to choose a query length to Section 6.

Given the *likely true/false positives* found in Table 3, we are now in a position to discuss how to rank them.

By utilizing the simple rank function introduced in [33], we generalize an algorithm that gives positive score to *likely true positives* and negative score to the *likely false positives*.

$$rank(S) = \sum -2/(num\_of\_class - 1), \quad likely \quad false \quad positives$$

Note that subsequences that are not used to classify any queries (correctly or not) get a zero score. Using a large number of queries, we compute a score vector for every time series in C. We denote *rank(S)* as the score for a subsequence S in the time series.

In the next step, we demonstrate how to extract the current best subsequence using the score vectors.

Step 2: We extract the highest scoring subsequence and place it in **D**. We demonstrate this step by using the example in Figure 6. Suppose in one of the iterations in Step 1, the starting point of the red/bold heartbeat has the highest score. We therefore need to extract this heartbeat. Because the *Euclidean* distance is very sensitive to even slight misalignments, and our scoring function is somewhat "blurred" as to its exact location in the x-axis. Extracting *exactly* the subsequence with query length *l* would be very brittle. Therefore, we "pad" the chosen subsequence some time series from the left and to the right, in particular with the *l*/2 data points to either side.

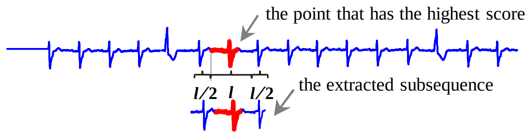

Figure 6: *top*) A snippet of BIDMC Congestive Heart Failure Database ECG data: Record-08 [3]. *bottom*) the extracted subsequence has twice the query length

Note that there is a slight difference between the first iteration and the subsequent iterations. Before the first iteration, **D** is empty. After the first iteration, **D** should contain exactly one subsequence from each class. This is the smallest **D** logically possible. Therefore instead of splitting C to the *friends* part and the *enemies* part, the algorithm finds the most representative subsequence in *each* class in Step 1, and then adds them into **D** in Step 2.

After the first iteration, we extract only the one subsequence that holds the highest score in C and add it into **D**. Thus the class sizes in **D** can be skewed, as the algorithm adds more exemplars to the more diverse/complicated classes. While we are iteratively building **D**, the size of C becomes smaller, as the extracted subsequence is removed from C in each iteration.

Step 3 : The algorithm examines the quality of the current **D** by doing classification using all the queries. The queries that are correctly classified by the current **D** will not be used to re-score C in the next iteration Step 1, since the current **D** is sufficient to correctly classify them. Only the misclassified queries will proceed back to Step 1 to re-score C. In each iteration Step 3, we redo classification experiments on D using all the queries, since the correctly classified queries in $\mathbf{D}_x$ may become misclassified in $\mathbf{D}_{x+\varepsilon}$.

After building a data dictionary for a training data, our last obligation is to learn the distance threshold.

3.3 Learning the Threshold Distance

After the data dictionary is built, we learn a threshold to allow us to reject future queries which do not belong to any of our learned classes. We begin by recording a histogram of the nearest neighbor distances of testing queries that are *correctly* classified using **D**, as shown in Figure 7. Next, we compute a similar histogram for the nearest neighbor distances of queries which should *not* have a valid and meaningful match within **D** (i.e., the `other` class). Where can we get such queries? In the example shown in Figure 7, we simply used gesture data as the `other` class, knowing *gestures* should not match a set of *heartbeats*. Note that it is occasionally possible that a gesture might match a heartbeat by coincidence; but our approach is robust to such spurious matches so long as they are relatively rare. If external datasets are in short supply, we can also simply permute subsequences of **D** to produce the `other` class, for example flipping heartbeats upside-down and backwards.

Given the two histograms, we choose the location that gives the equal-error-rate as the threshold (about 7.1 in Figure 7). However, based on their tolerance to false negatives, users may choose a more liberal or conservative decision boundary.

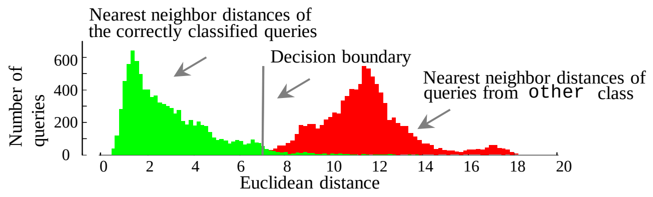

Figure 7: The *green*/left histogram contains the nearest neighbor distances of correctly classified queries for the ECG datas used in Section 4.2. The *red*/right histogram shows nearest neighbor distances for queries from the `other` class

3.4 Uniform Scaling Technique

Finally we can trivially replace the *Euclidean* distance with *Uniform Scaling* [4] distance in the above data dictionary building and threshold learning process [15]. We choose the maximum scaling factor based on the variability of time series in the domain at hand, see discussion in Section 4. A naive implementation of *Uniform Scaling* would be slow, but [15] shows that it can be computed in essentially the same time as *Euclidean* distance.

4. EXPERIMENTAL EVALUATION

We begin by discussing our experimental philosophy. To ensure that our experiments are easily reproducible, we have built a website, which contains all the datasets and code [38]. In addition, this website contains additional experiments which are omitted here for brevity. Our experimental results support our claim that using *only* the data dictionary is more accurate and faster than using *all* the available training data.

We compare our algorithm with several widely used rival approaches. The most widely used rival approach extracts feature vectors from the data and reports the best result among multiple models [2][25][31]. In addition, we compare with the obvious strawman of using *all* the training data, which is just a special case of our framework, in which all the training data is used (i.e. $\mathbf{D}_{100\%}$).

To support our claim that the real-world streaming data is not as clean as the contrived datasets used in most literature, we report the percentage of the rejected queries produced by the learned threshold and show some examples[5].

We report the error rate using both *Euclidean* distance and *Uniform Scaling* distance to support our claim that the latter can be very useful for time series classification problems.

While we are ultimately interested in the *testing* error rate, we also report the *training* error rate, as this can be used to predict the best size of the data dictionary for a given problem. However, for completeness, we build and test the data dictionary $\mathbf{D}_x$ for every value of x, from the smallest logically possible size to whatever value minimizes the holdout error rate (this is generally much less than $x = 10\%$).

The reader may object that error rate is not the correct measure here. Imagine that our rejection threshold is so high that we reject 999 of 1,000 queries, and just happen to get one classified object correct. In this case, reporting a 0% error rate would be dubious at best. This is of course what precision/recall and similar measurements are designed to be robust to. However, in all our case studies, our rejection rate is much less than 10%, so reporting just the error rate is reasonable, and allows us to present more visually intuitive figures. Moreover, we will show experiments where we consider the *correctness* of rejections made by our algorithm.

Finally, we defer experiments that consider the scalability of dictionary building to [38], noting in passing that this is done *offline*, and that in any case we can do this *faster* than real-time. In other words we can learn the dictionary for an hour heartbeats in much less than one hour.

4.1 An Example Application in Physiology

We consider a physical activity dataset containing eight subjects performing activities such as: `normal-walking`, `walking-very-slow`, `descending-stairs`, `cycling`, and `inactivity` (an umbrella term for lying-in-bed/sitting-still/standing-still), etc [22]. Approximately eight hours of data at 110Hz was collected from wearable sensors on the subjects' wrist, chest and shoes.

For simplicity of exposition, we consider only a *single* time series, recording the roll-axis from the sensor placed in the subjects' shoe. However, our algorithm trivially extends to multi-dimensional data (examples appear at [38]). Note that although our algorithm only uses a *single* axis from the sensor, we demonstrate that our results are significantly better than rival algorithms that use all *three*-axis data (roll, pitch and yaw) from the same sensor [31].

We randomly choose 60% of the data as training data, and treat the rest as testing data. In Figure 8, we show the training/testing error rates as our algorithm grows **D** from the smallest logically possible size (about 0.39% of all the training data) to the point where it is clear that our algorithm can no longer improve. Although our algorithm bottoms out earlier in the plot, we wish to demonstrate that the output is very smooth over a wide range of values.

[4] The reader may ask why not Dynamic Time Warping? Empirically, we tried it and it does not help. Moreover we should *not* expect it to help this problem [38].

[5] Due to space limitations we only show the rejected queries in the first case study. See [38] for examples of rejected queries from the other case studies.

We compare with the widely-used rival approach [2][25][31], which extracts signal features from the sliding windows. For fairness to this method, we used their suggested window size [31], and tested *all* of the following classifiers: K-nearest neighbors (K=5), SVM, Naïve Bayes, boosted decision trees and C4.5 decision tree [2][25][31]. The *best* classification result is 0.364 achieved by the C4.5 decision tree.

For the commonly used strawman of using all the training data, the testing error rate is 0.221. However, our framework equals this testing error rate using only 1.6% (i.e.$\mathbf{D}_{1.6\%}$) of the training data and obtains the significantly lower error rate of 0.152 at $\mathbf{D}_{8.3\%}$. Moreover, given that we are using only about one twelfth the data, we are able to classify the data about twelve times faster.

Our algorithm is clearly highly competitive, but does it owe its performance to choice of *which* subsequences are placed in **D** by our algorithm? To test this, we built another **D** by *randomly* extracting subsequences from C. As Figure 8 also shows, our systematic method for ranking subsequences is significantly better than *random* selection.

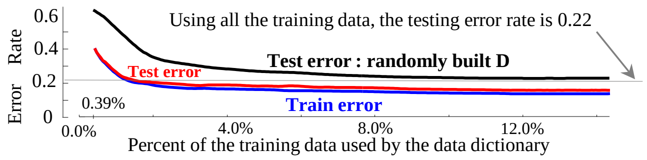

Figure 8: The classification error rates for D from $\mathbf{D}_{0.39\%}$ to $\mathbf{D}_{14.2\%}$ for the physical activity dataset [22]

A final observation about these results is that the training error rate is a very good predictor of the test error rate. As Figure 8 shows, the training error is only *slightly* optimistic.

We are now ready to test our claim that *Uniform Scaling* (c.f. Section 2.2) can help in datasets containing signals acquired from human behavior/physiology. We repeated the experiments above under the exact same conditions, except we replaced *Euclidean* distance with *Uniform Scaling* distance in both the training and testing phases.

Based on studies of variability for human locomotion [1][4][19], we chose a maximum scaling factor of 15%; that is to say, queries are tested at every scale from 85% to 115% of their original length. *Uniform Scaling* obtains a 0.085 testing error rate at $\mathbf{D}_{8.1\%}$, significantly better than *Euclidean* distance, as shown in Figure 9.

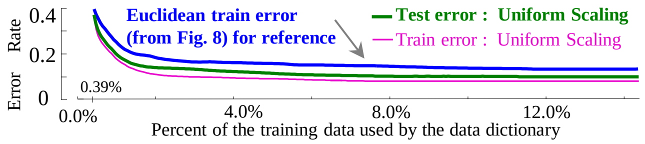

Figure 9: The *pink*/*green(bold)* curves are train/test error rates obtained when we replaced *Euclidean* distance with *Uniform Scaling* distance

We learned a threshold distance of 14.5 for **D**[6]. With this threshold, our algorithm rejects 9.5% of the testing queries. In Figure 10, we see that the vast majority of rejected queries do belong to the `other` class and are thus correctly rejected.

[6]Experimental results show that the threshold distances for **D** built with *Euclidean* distance and *Uniform Scaling* distance are almost identical. Therefore, we only report one threshold distance.

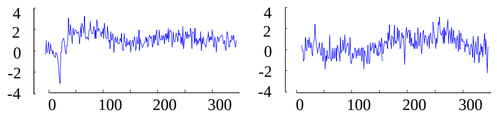

Figure 10: Two examples of rejected queries. Both queries contain significant amount of noise

We do not present formal numerical results for the rejected queries, as the weakly-annotated format of the original data does not provide the label of the objects with certainty.

This dataset draws from sporting activities. We also consider a similar but independent dataset [2], which considers more quotidian activities such as `tooth-brushing` etc. We achieve near identical improvements on this dataset, thus we relegate a discussion of it to [38].

4.2 An Example Application in Cardiology

We apply our framework to a large ECG dataset: the BIDMC Congestive Heart Failure Database [3]. The dataset includes ECG recordings from fifteen subjects with severe congestive heart failure. The individual recordings are each about 20 hours in duration, sampled at 250 Hz.

Ultimately, the medical community wants to classify patient-independent *types* of heartbeats. However in this experiment, we classify *individuals'* heartbeats. This is simply because we are able to obtain huge amounts of labeled data this way. Note that as hinted at in Figure 2, the data *is* complex and noisy. Moreover, a single (unhealthy) individual may have many different types of beats. Cardiologist Helga Van Herle from USC informs us this is a perfect proxy problem.

We use a randomly selected 150 minutes of data for training, and 450 minutes of data for testing.

In Figure 11, we show the training/testing error rates as our algorithm grows the data dictionary from the smallest possible size ($\mathbf{D}_{0.28\%}$) to the point where it is clear that our algorithm can no longer improve.

Note that the testing error rate is 0.102 using the strawman of using *all* the training data, which is significantly better than the default error rate 0.933. However, our framework duplicates this error rate using only 2.1% (i.e. $\mathbf{D}_{2.1\%}$) of the training data**,** and obtains the much lower error rate of 0.076 at $\mathbf{D}_{4.5\%}$. From Figure 11 we again see that our method for building dictionaries is much better than random selection.

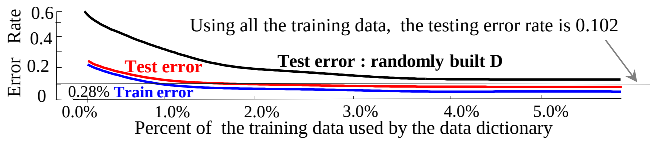

Figure 11: The classification error rates for D from $\mathbf{D}_{0.28\%}$ to $\mathbf{D}_{5.82\%}$ for BIDMC Congestive Heart Failure Database[3]

We again test the *Uniform Scaling* distance instead of *Euclidean* distance in both the training/testing phases. Based on studies of variability for human heartbeats [3][8] and advice from a cardiologist, we chose a maximum scaling factor of 25%. In Figure 12 , *Uniform Scaling* obtains a 0.035 testing error rate at $\mathbf{D}_{4.6\%}$, significantly better than using the *Euclidean* distance.

As illustrated in Figure 7, the threshold distance for **D** is 7.1. With this threshold, the algorithm rejects 4.8% of the testing queries. Once again, these rejections (which can be seen at

[38]) all seem like reasonable rejections due to loss of signal or extraordinary amounts of noise/machine artifacts.

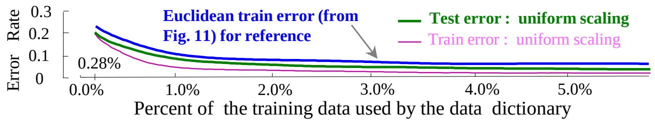

Figure 12: The *pink*/*green(bold)* curves are train/test error rates obtained when we replaced *Euclidean* distance with *Uniform Scaling* distance

5. RELATED WORK

There is significant literature on time series classification [2][6][9][20][30][32] both in the data mining community and beyond. However, almost all of these works make the three assumptions we relaxed in this work, and are thus orthogonal to the contributions here. Our algorithm can be seen as building a data dictionary of primitives for the very long streaming/continuous time series [26][27]. Other works have also done this, such as [27], but they use significant amount of human effort to *hand-edit* the time series into patterns. In contrast, we build dictionaries automatically, with no human intervention.

6. CONCLUSION AND FUTURE WORK

We introduced a novel framework that requires only very weakly-labeled data and removes the unjustified assumptions made in virtually all time series classification research. We demonstrated over several large, real-world datasets that our method is significantly more accurate than several common strawman algorithms. Moreover, with less than one tenth of the original data kept in **D**, we are at least ten times faster at classification time.

Our algorithm has just one parameter, the length of queries. In our activity datasets, we simply used the original authors values [2][22], and for ECGs we used a cardiologist's suggestion. By changing these suggested values we empirically found that we are not sensitive to this parameter. Nevertheless in future work, we plan to learn it from the data.

7. ACKNOWLEDGEMENTS

We thank all the donors of dataset. We would like to acknowledge the financial support for our research provided by NSF grants IIS – 1161997.

8. REFERENCES

[1] K. Aspelin, *Establishing Pedestrian Walking Speeds*. Portland State University. *www.usroads.com/journals/p/rej/9710/re971001.htm*, retrieved 2009-08-24.

[2] L. Bao and S.S. Intille, *Acitivity Recognition from User-Annotated Acceleration Data*, In Proc' of the 2nd International Conference on Pervasive Computing, pages1-17, 2004.

[3] The BIDMC Congestive Heart Failure Database, *www.physionet.org/physiobank/database/chfdb/*

[4] G.A. Cavagna, N.C. Heglund and C.R. Taylor, *Mechanical work in terrestrial locomotion: two basic mechanisms for minimizing energy expenditure*, Journal of Physiology 233(5): R243-R261, 1977.

[5] P.de Chazal, M. O'Dwyer, and R. B. Reilly, *Automatic classification of ECG heartbeats using ECG morphology and heartbeat interval features*, IEEE Trans. Biomed. Eng., vol. 51, pp. 1196-06, Jul.2004.

[6] L. Chen, M. T. Özsu and V.Oria, *Robust and fast similarity search for moving object trajectories* , In Proc' of the ACM SIGMOD, 2005.

[7] CMU Graphics Lab Motion Capture Database, *mocap.cs.cmu.edu/*, retrieved 2012-04-24.

[8] Electrocardiography,*en.wikipedia.org/wiki/Electrocardiography*.

[9] D. Gafurov, K. Helkala and T. Søndrol, *Biometric Gait Authentication Using Accelerometer Sensor*, Journal of Computers , (1) 6, 2006.

[10] D. Gafurov and E. Snekkenes, *Towards Understanding the Uniqueness of Gait Biometric*, 8th IEEE International Conference on Automatic Face & Gesture Recognition, 2008.

[11] M.A. Hanson, H.C. Powell Jr, A.T. Barth, J. Lach, M.B.C, Brown, *Neural Network Gait Classification for On-Body Inerital Sensors*, In Proc' of the 2009 Sixth International Workshop on Wearable and Implantable Body Sensor Networks, 2009.

[12] B. Hu, T. R Rakthanmanon, Y. Hao, S. Evans, S. Lonardi, and E. Keogh, *Discovering the Intrinsic Cardinality and Dimensionality of Time Series using MDL*, ICDM, 2011.

[13] E. Keogh, Q. Zhu, B. Hu, Y. Hao , X. Xi, L. Wei, and C. A. Ratanamahatana. The UCR Time Series Classification/Clustering Homepage: *www.cs.ucr.edu/~eamonn/time_series_data/*, 2006.

[14] E. Keogh, S. Lonardi and C. Ratanamahatana, *Towards Parameter-Free Data Mining* , In Proc' of the tenth ACM SIGKDD, 2004.

[15] E. Keogh, T. Palpanas, V.B. Zordan, D. Gunopulos and M. Cardle, *Indexing Large Human-Motion Databases*, VLDB, 2004.

[16] P. Koch, W. Konen and K. Hein, *Gesture Recognition on Few Training Data using Slow Feature Analysis and Parametric Bootstrap* , IJCNN, 2010.

[17] J. Lester, T. Choudhury, N. Kern, G. Borriello and B. Hannaford, *A Hybrid Discriminative/Generative Approach for Modeling Human Activities*, IJCAI, 2005.

[18] J. Liu, K. Yu, Y. Zhang and Y. Huang, *Training Conditional Random Fields Using Transfer Learning for Gesture Recognition*, ICDM,2010

[19] T.A. McMahon, G.C. Cheng, *The mechanics of running : How does stiffness couple with speed*, Journal of Biomechanics, Vol 23, 1990.

[20] M. Morse and J.M. Patel, *An Efficient and Accurate Method for Evaluating Time Series Similarity*, Proc SIGMOD, 2007.

[21] V. Niennattrakul, E. Keogh and C.A. Ratanamahatana, *Data Editing Techniques to Allow the Application of Distance-Based Outlier Detection to Streams*, ICDM, 2010.

[22] PAMAP, Physical Activity Monitoring for Aging People, *www.pamap.org/demo.html* , retrieved 2012-05-12.

[23] J. Pärkkä, M. Ermes, P. Korpipää, J. Mäntyjärvi, J. Peltola, and I. Korhonen, *Activity classification using realistic data from wearable sensors*, IEEE Trans. Inf. Tech. Biomed., vol. 10, pp. 119-28, 2006.

[24] E. Pekalska, R.P.W. Duin and P. Paclík, *Prototype selection for dissimilarity-based classifiers*, Pattern Recognition, 39, 2006.

[25] C.Pham, T. Plötz, P. Olivier, *A dynamic time warping approach to real-time activity recognition for food preparation*, In Proc' of the First international joint conference on Ambient intelligence, 2010.

[26] M. Raptis, D. Kirovski, and H. Hoppes, *Real-Time Classification of Dance Gestures from Skeleton Animation*, In Proc'of the ACM SIGGRAPH symposium on Computer animation, 2011.

[27] M. Raptis, K. Wnuk, and S. Soatto, *Flexible Dictionaries for Action Recognition*, In Proc' of the 1st International Workshop on Machine Learning for Vision-based Motion Analysis, 2008.

[28] T.Rakthanmanon, E. Keogh, S. Lonardi, and S. Evans. *Time Series Epenthesis: Clustering Time Series Streams Requires Ignoring Some Data*. ICDM 2011.

[29] C.A. Ratanamahatana (2012). Personal communcation. May 2012.

[30] C.A. Ratanamahatana and E. Keogh, *Making Time-series Classification More Accurate Using Learned Constraints*, SDM, 2004.

[31] A. Reiss and D. Stricker, *Introducing a Modular Activity Monitoring System*, 33th International EMBC, 2011.

[32] J. Song and D. Kim, *Simultaneous Gesture Segmentation and Recognition based on Forward Spotting Accumulative HMM*, In Proc' of the 18th ICPR, 2006.

[33] K. Ueno, X. Xi, E. Keogh and D. Lee, *Anytime Classification Using the Nearest Neighbor Algorithm with Applications to Stream Mining*, ICDM, 2010.

[34] J. Usabiaga, G. Bebis, A. Erol, M. Nicolescu, *Recognizing simple human actions using 3D head movement*, Computational Intelligence, 23(4), 2007.

[35] R.D. Vatavu, *The Effect of Sampling Rate on the Performance of Template-based Gesture Recognizers*, Proc of ICMI, 2011.

[36] L. Ye, X. Wang, E. Keogh and A. Mafra-Neto, *Autocannibalistic and Anyspace Indexing Algorithms with Applications to Sensor Data Mining*, SDM, 2009.

[37] A.Y. Yang, A. Giani, R. Giannatonio, K. Gilani, etc. "*Distributed Human Action Recognition via Wearable Motion Sensor Networks* ", *www.eecs.berkeley.edu/~yang/software/WAR/index.html,*

[38] Project URL: sites.google.com/site/sdm13realistic/