Knowledge-Based Systems xxx (xxxx) xxx

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

Original software publication

jKarma: A highly-modular framework for pattern-based change detection on evolving data[☆]

Angelo Impedovo [*], Corrado Loglisci, Michelangelo Ceci, Donato Malerba

University of Bari Aldo Moro, Department of Computer Science, Knowledge Discovery and Data Engineering Laboratory, Bari 70125, Italy

ARTICLE INFO

Article history:
Received 7 August 2019
Received in revised form 23 October 2019
Accepted 28 November 2019
Available online xxxx

Keywords:
Change detection
Pattern mining
Evolving data
Software modularity

ABSTRACT

Pattern-based change detection (PBCD) describes a class of change detection algorithms for evolving data. Contrary to conventional solutions, PBCD seeks changes exhibited by the patterns over time and therefore works on an abstract form of the data, which prevents the search for changes on the raw data. Moreover, PBCD provides arguments on the validity of the results because patterns mirror changes occurred with any form of evidence. However, the existing solutions differ on data representation, pattern mining algorithm and change identification strategy, which we can deem as main modules of a general architecture, so that any PBCD task could be designed by accommodating custom implementations for those modules. This is what we propose in this paper through *jKarma*, a highly-modular framework written in Java for defining and performing PBCD.

1. Introduction

Pattern-based change detection (PBCD) refers to the class of change detection solutions able to find out data-points in which the data distribution changes by acting on the patterns rather than on raw data. Despite the attention it could raise, we ascertain lacking in comprehensive environments able to investigate the problem with alternative solutions or even with integrable implementations. Its main peculiarity is working in an unsupervised fashion, without relying on labeling, which often makes it preferable to the supervised approaches.

The blueprint relies on three main methodological decisions, that is, data description, pattern mining algorithm, and change identification strategy. Pattern mining algorithms are in charge of building an abstract representation of the evolving data (patterns). The change identification strategy is in charge of searching for changes expressed by the patterns by the effect of possible distribution drifts in the underlying data. In PBCDs, the changes correspond to variations that occurred on the patterns discovered over time. While the decision on which technique to use for the pattern mining and change identification components determines the algorithmic aspects of a PBCD solution, the data representation strictly concerns the formalism of the evolving data, characteristics of the original data to consider and pattern language. For instance, the PBCDs implemented in [1,2] identify the changes through a generic notion of Jaccard dissimilarity defined for three different types of patterns, that is, *frequent subnetworks*, and δ*-closed itemsets*.

Our purpose is to provide the users with a software framework that supports the study of a predictive problem (change detection) through an unsupervised data mining task (pattern mining) while disseminating existing PBCDs and promoting the development of new ones.

As our best knowledge, this is the first solution that combines change detection and pattern mining, while they have been explored as separated tasks in existing frameworks. MOA [3] and scikit-multiflow [4] have been designed to work on evolving data (data stream) and basically offers a toolkit of predictive algorithms which deal with concept drift (changes of the target concept), without particular attention on the change identification, which, in this work, is reached through the patterns. Several classes of patterns (such as sequential patterns, periodic patterns, etc.) have been considered in SPMF [5], and a wide list of implementations is available, but no type of patterns has been used for change identification and no algorithm has been designed for change detection.

We accomplish this with jKarma, a framework written in Java and released under Apache License 2.0 which offers loosely coupled modules, does not require particular programming efforts and enables the use of reusable, off-the-shelf or ad-hoc implementations for two algorithmic components above introduced. jKarma supports the users in building and performing custom ad-hoc PBCDs on-the-fly, through an API, which can be integrated into larger data analysis projects.

☆ No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.knosys.2019.105303.

* Corresponding author.
E-mail address: angelo.impedovo@uniba.it (A. Impedovo).

https://doi.org/10.1016/j.knosys.2019.105303
0950-7051/© 2019 Elsevier B.V. All rights reserved.

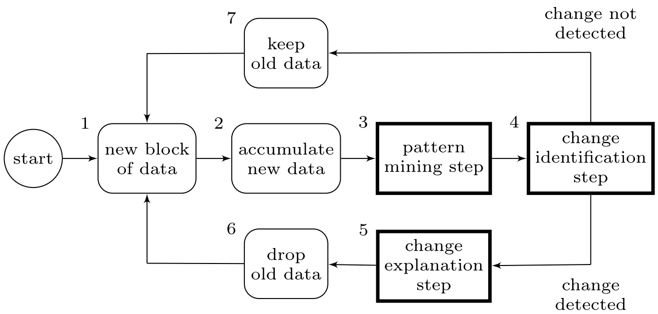

Fig. 1. Overview of the PBCD architecture.

2. Background and PBCD architecture

In this section we provide preliminary notions and explain the conceptual architecture under which PBCD solutions can be collocated.

Given the set of items I, a transactional database is the time-ordered sequence $D = \langle T_1, T_2, \ldots, T_n \rangle$. Each $T_i \subseteq I$ is a transaction observed in t_i and uniquely identified by id i. Thus, a pattern $P \subseteq I$ is a set of $|P|$ items, and, for PBCD purposes, they are discovered from transactions collected by means of time windows. More precisely, a window $W = [t_i, t_j]$, with $t_i < t_j$, is the sequence of $|W| = j - i + 1$ transactions $\{T_i, \ldots, T_j\} \subseteq D$. We will use the notation P_W to denote the set of patterns discovered on the window W.

In the blueprint of PBCD, the *Mining step* and the *Identification step* search for change-points on evolving data by using *Time-windows models*. In particular, two time-windows W and W', $W = [t_b, t_e]$ and $W' = [t'_b, t'_e]$ ($t_b \leq t'_b \leq t_{e+1}$, $t_e < t'_e$) are built (Fig. 1, Step 2) and input to a pattern mining algorithm, which discovers two pattern sets P_W and $P_{W'}$ (Fig. 1, Step 3). In these terms, the changes are attributed to the patterns which make P_W different from $P_{W'}$. In particular, we can determine the *(i)* amount of the change through a quantification of the difference between the two pattern sets, *(ii)* temporal collocation of the changes (change-points) as the time in which the difference-patterns occur (Fig. 1, Step 4).

For this core procedure, jKarma offers a general architecture that supports software modularity (Fig. 1). It makes the decisions on the specific implementation for *Time-windows models*, *Mining step* and *Identification step* independent from each other. Indeed, the time-window models allow us to build sub-sequences of data regardless of their original structure (such as, itemsets, subgraphs, subtrees) and the choice of the specific model to use (such as, sliding, landmark, tilted) is not constrained neither by pattern mining nor change identification, since the time-windows are only in charge of to scan evolving data and account for new (recent) transactions and old (past) transactions (Fig. 1, Steps 2, 6 and 7).

The sole assumption of jKarma is that evolving data, regardless of both their complexity and their source, must be stored as transactional databases. This makes jKarma flexible with respect to the possibility of both using alternative search strategies for mining transaction-based patterns (such as depth-first, breadth-first) and considering several notions of transaction-based patterns (such as closed and maximal).

The Identification step (Fig. 1, Step 4) is in charge of spotting variations which the new pattern set $P_{W'}$ presents in comparison with the old pattern set P_W. This activity is not a mere operation of complement in the set theory but considers the changes at the level of the evidence which characterizes the patterns individually. To do that, jKarma makes available different implementations of dissimilarity measures (such as Jaccard dissimilarity, etc.) defined on several notions of evidence of the patterns (such as relative frequency, frequency ratio, periodicity, etc.). Not all the dissimilarity values are worthwhile of interest, but only those that exceed a desired degree of change, as well as, not all the patterns exhibit a variation in the evidence, but only that exceed a desired degree of evidence. This enables jKarma to provide "explanations" of the changes in the form of patterns that better express the underlying changes (Fig. 1, Step 5).

Given the framework, it is possible to instantiate different computational solutions by plugging different combinations of components. An example is the KARMA algorithm [1], a PBCD specifically designed for graph data, which is based on exhaustive mining of frequent connected subgraphs (FCSs hereafter). KARMA iteratively consumes blocks Π of graph snapshots coming from a stream of graphs D (Fig. 1, Step 1). The algorithm accumulates Π by using two successive landmark windows W and $W' = W \cup \Pi$ (Fig. 1, Step 2). This way, it mines the complete sets of FCSs, P_W and $P_{W'}$, necessary to the detection step (Fig. 1, Step 3). The window grows ($W = W'$, Fig. 1, Step 7) with new graph snapshots, and the associated set of FCSs is kept updated until the *macroscopic change* between P_W and $P_{W'}$ is identified (Fig. 1, Step 4). In that case, the algorithm first characterizes the change by discovering *microscopic changes* (Fig. 1, Step 5) and then drops old data by retaining only the last block of transactions ($W = \Pi$, Fig. 1, Step 6). Then, the analysis restarts.

3. Software framework

jKarma is an highly-modular framework written in Java 8 for defining and executing custom PBCDs. Its main purpose is facilitating the rapid prototyping of custom PBCDs by implementing the general architecture seen in Section 2. Custom PBCDs are instantiated by composition, meaning that existing modules for the pattern mining, change identification and change explanation steps can be combined together to design PBCDs ready to be used. Alternatively, the framework exposes an API library for the definition of new modules.

3.1. Software architecture

jKarma is developed as a multi-module Maven project, in which five different modules coexist: *(i)* `jkarma-core` is the root module; *(ii)* `jkarma-dist` automates the jar and javadoc building with Maven; *(iii)* `jkarma-model` exposes different entity classes; *(iv)* `jkarma-mining` exposes the API for defining custom pattern mining strategies; *(v)* `jkarma-pbcd` exposes the API for assembly custom PBCD pipelines on top of pattern mining strategies.

3.2. Software functionalities

jKarma is a java library which exposes an API allowing *the definition and execution of custom PBCD strategies on transactional data sources.*

These functionalities are completely independent from other data mining and machine learning libraries, and third-parties data sources. This allows jKarma to offer two advantages, *(i)* integrability with existing projects using their own data sources (such as, relational databases, graph databases, xml documents), and *(ii)* potential interoperability with existing analytics frameworks. More specifically, two factory classes introduce the functionalities:

Please cite this article as: A. Impedovo, C. Loglisci, M. Ceci et al., jKarma: A highly-modular framework for pattern-based change detection on evolving data, Knowledge-Based Systems (2019) 105303, https://doi.org/10.1016/j.knosys.2019.105303.

- `org.jkarma.mining.structures.Strategies` for constructing generic `MiningStrategy` objects implementing the pattern mining algorithm to be used in the *Mining step* of the PBCD architecture.
- `org.jkarma.pbcd.detectors.Detectors` for constructing generic PBCD objects implementing the details of every step involved in the PBCD architecture.

3.3. Implementation details

The expressiveness of the programming interface enables the modular design of custom PBCD strategies. This is made through the reuse of existing software modules concerning the (*mining step* and *identification step*) in the PBCD architecture.

In the current version, it is possible to devise PBCDs based on 5 pattern mining algorithms (Eclat, diffEclat, LCM and LCM-Max, and PFPM), each of which is compatible with three pattern languages (itemsets, subgraphs, subtrees), four time-window models (blockwise sliding/landmark, cumulative sliding/landmark) and two space-search algorithms (depth-first search and beam search). Furthermore, the API allows the user to implement his own modules when necessary.

4. Illustrative examples

In this section we report some illustrative examples of how jKarma can be used for building different PBCDs. Since the definition of custom PBCDs is done by following a component-based architectural model, in the following we will show how the user can specify the details about the *Mining step*, the *Detection step*, and the *Explanation step*. In particular, this is done in a two-step approach, the first step uses the `Strategies` class to define a `MiningStrategy` object, while the second step injects that object into a custom PBCD object via the `Detectors` class. It is evident that the choices done have a determinant effect on the behavior of the PBCDs, which can result in different change detection results. Clearly, the choice of the details is domain-specific and depends on the problem at hand.

4.1. Definition of mining strategies

As discussed before, the mining strategy is configured in jKarma by instantiating a generic `MiningStrategy<A,B>` object. This implies the specification of the set of items, type of the items (`A`), pattern language, pattern evidence criterion (implemented in class of type `B`), pattern mining algorithm and search strategy of the patterns.

We report an introductory example showing the definition of a mining strategy, based on the Eclat algorithm, which searches for patterns in the form of FCSs. The pattern evidence criterion filters out FCSs whose frequency is lower than the minimum threshold (`minSupp`). The Eclat algorithm computes the frequency of a pattern by inspecting its *tidset*, a data structure collecting the identifiers of the transactions in which the pattern occurs.

```
public MiningStrategy<LabeledEdge, TidSet>
defineStrategy(double minSupp) {
    TidsetProvider<LabeledEdge> accessor = new
        TidsetProvider<>(Windows.blockwiseSliding());
    return Strategies.uponSubgraphs().eclat(minSupp)
        .limitDepth(3).dfs(accessor);
}
```

Listing 1: FCS mining strategy based on Eclat.

Here, the strategy, which is an object of type `MiningStrategy<LabeledEdge, TidSet>`, is initially instantiated by the `uponSubgraphs` method that specifies the FCSs pattern language. The `eclat` method injects the mining algorithm into the mining strategy, while the `limitDepth` method limits the maximum number of edges in every FCS. Then, an instance of type `TidsetProvider<LabeledEdge>` (`accessor`) scans the transactions and builds the tidsets. Finally, the `dfs` method finalizes the strategy and forces the Eclat algorithm to run in a depth-first search fashion.

An interesting aspect is that Eclat, in this case, is used to mine FCSs, while natively it is a frequent itemset mining algorithm. This represents an advantage because the pattern language is decoupled from the mining algorithm, so, equivalent strategies defined on different languages (for example, itemsets and subtrees) can be defined. This is illustrated in the following listing:

```
public MiningStrategy<LabeledEdge, TidSet>
defineStrategy(double minSupp) {
    TidsetProvider<LabeledEdge> accessor = new
        TidsetProvider<>(Windows.blockwiseSliding());
    return Strategies.uponSubtrees().eclat(minSupp)
        .limitDepth(3).dfs(dataAccessor);
}
```

Listing 2: Frequent subtrees mining strategy based on Eclat.

Both the strategies, illustrated in two listings above, are based on the Eclat algorithm and compute the frequencies of the patterns through an intersection set operation on the `TidSet` objects. While this is a good choice on sparse datasets, it could be time-consuming for dense datasets [6]. This is faced in jKarma through an alternative strategy based on the *diffEclat* algorithm, which uses the `DiffSet` data structures instead of `TidSet` instances. The implementation (i) invokes the `diffEclat` method instead of the `eclat` method, and (ii) replaces the `TidsetProvider` data accessor with a `DiffsetProvider`.

```
public MiningStrategy<LabeledEdge, DiffSet>
defineStrategy(double minSupp) {
    DiffSetProvider<LabeledEdge> accessor =
        DiffSetProvider<>(Windows.blockwiseSliding());
    return Strategies.uponSubtrees().diffEclat(minSupp)
        .limitDepth(3).dfs(dataAccessor);
}
```

Listing 3: Frequent subtrees mining strategy based on diffEclat.

However, the main pitfall of the examples illustrated above is their exhaustiveness, which leads to the discovery of complete sets of patterns. The exhaustive search is caused by the `dfs` method, which forces the mining algorithm to work in exhaustive mode. jKarma can be used to define non-exhaustive strategies based on beam-search and heuristics as done in [7]. In the following example, a non-exhaustive strategy, based on Eclat, for mining FCSs is built. In this listing, the search-space of the patterns is explored with a beam-search of size k.

```
public MiningStrategy<LabeledEdge, TidSet>
defineStrategy(double minSupp, int k) {
    TidSetProvider<LabeledEdge> accessor = new
        TidSetProvider<>(Windows.blockwiseSliding());
    return Strategies.uponSubgraphs().eclat(minSupp)
        .limitDepth(10)
        .beam(accessor, k, new AreaHeuristic());
}
```

Listing 4: Non-exhaustive FCS mining strategy based on Eclat.

4.2. Definition of PBCDs

As introduced in Section 2, PBCD relies on the sets of patterns P_W and $P_{W'}$ discovered on two time windows W and W' respectively. These are used to compute the dissimilarity score $d(P_W, P_{W'})$, which, in its turn, allows us to quantify the degree of change. The patterns P_W and $P_{W'}$ are again processed for the change explanation. The dissimilarity score is computed on two equally-sized vector encodings F_W and $F_{W'}$, in which the ith

KNOSYS: 105303

ARTICLE IN PRESS

Table 1
Running times and accuracies of PBCD-1, PBCD-2, KARMA, and StreamKrimp on synthetic data.

Dataset	Running times (s)			
	PBCD-1	PBCD-2	KARMA	StreamKrimp
synth-drifts-1	12.913	6.194	60.763	86.130
synth-drifts-2	12.284	6.522	55.982	77.138
synth-drifts-3	12.603	6.463	58.137	76.750
Dataset	**Accuracy**			
	PBCD-1	PBCD-2	KARMA	StreamKrimp
synth-drifts-1	0.987	0.918	0.804	0.931
synth-drifts-2	0.991	0.916	0.799	0.911
synth-drifts-3	0.988	0.918	0.796	0.916

element corresponds to the weight associated to the ith pattern, according with the enumeration of $P_W \cup P_{W'}$ with respect to W and W', respectively. This way, the change can be quantified by means of vector-based measures, instead of set-based ones. Clearly, different weighting scheme could determine different vector encodings for the same sets of patterns. Moreover, alternative measures could determine different change scores between the same vector encodings.

In jKarma, a PBCD pipeline is defined by injecting a `MiningStrategy<A,B>` instance into a `PBCD<C,A,B,D>` object via the `Detectors` class. This ensures the type-checking consistency between the patterns discovered in the *mining step* and those used in the *identification step*. The generic type `C` specifies the type of transactions that will be consumed by the PBCD, while the generic type `D` denotes the pattern weighting scheme adopted. Finally, a PBCD is finalized by providing details on the *identification step* and *explanation step*.

In the following example, a `PBCD` is built by passing a `MiningStrategy` to the `upon` method. Then, a binary weighting scheme and the Jaccard dissimilarity measure are specified via the `unweighted` method. The PBCD will use the `isFrequent` predicate when constructing the binary vector encodings, while the `UnweightedJaccard` computes the dissimilarity score. This PBCD explains changes by discovering emerging patterns via the `Descriptors.eps` method. Finally, the PBCD is finalized with the `build` method which (i) sets the minimum change threshold to 0.5, and (ii) arranges a data source with blocks of 15 transactions.

```java
public PBCD<TemporalGraph, LabeledEdge, TidSet, Boolean>
buildPBCD(MiningStrategy<LabeledEdge,TidSet> strategy) {
    UnweightedJaccard m = new UnweightedJaccard();
    return Detectors.upon(strategy)
        .unweighted((p,t)->Patterns.isFrequent(p,
            minFreq, t), m)
        .describe(Descriptors.eps(minGr)).build(0.5, 15);
}
```

Listing 5: PBCD based on the unweighted jaccard dissimilarity between binary-valued vector encodings of patterns.

However, it is possible to build PBCDs with alternative weighting scheme and dissimilarity by providing different arguments. The same holds for the *Explanation step*.

4.3. A complete example: the KARMA algorithm

We report a complete example[1] in which jKarma is used so as implementing the PBCD algorithm presented in [1]. The example also shows how to react to changes, by following the event-listener paradigm. In particular, the `changeDetected` method will be executed when a change has been detected, otherwise, the `changeNotDetected` method will be executed. Information associated to the change detection events are accessible through `ChangeDetectedEvent` and `ChangeNotDetectedEvent` instances.

```java
public PBCD<TemporalGraph, LabeledEdge, TidSet, Boolean>
getKARMA(double minSupp, double minChange, double minGr){
    //auxiliary components
    TidSetProvider<LabeledEdge> dataAccessor = new
        TidSetProvider<>(Windows.cumulativeLandmark());
    UnweightedJaccard m = new UnweightedJaccard();
    Descriptor descriptor =
        Descriptors.partialEps(minSupp, minGr);

    //mining strategy definition
    MiningStrategy<LabeledEdge, TidSet> strategy =
        Strategies.uponSubgraphs().eclat(minSupp)
        .limitDepth(3).dfs(dataAccessor);

    //PBCD definition
    return Detectors.upon(strategy)
        .unweighted((p,t)->Patterns.isFrequent(p,
            minSupp, t), m)
        .describe(descriptor).build(minChange, 15);
}

public void runKARMA(Stream<TemporalGraph> dataSource){
    PBCD<TemporalGraph, LabeledEdge, TidSet, Boolean>
        detector = this.getKarma(0.15, 0.2, 1.2);
    //change detection event listening
    detector.registerListener(new
        PBCDEventListener<LabeledEdge, TidSet>(){
        public void changeDetected(
        ChangeDetectedEvent<LabeledEdge,TidSet> e){
            //reaction to change detected
        }
        public void changeNotDetected(
        ChangeNotDetectedEvent<LabeledEdge,TidSet> e){
            //reaction to change not detected
        }
    });
    //consume the data source
    dataSource.forEach(detector);
}
```

Listing 6: Example of jKarma implementing the KARMA PBCD, presented in [1]. The PBCD is used on a stream of labeled graphs.

4.4. Comparative evaluation

To show the effectiveness of jKarma in deploying actionable PBCDs, we compare the detection accuracy and running times of four PBCD algorithms on three synthetic datasets.[2] Three algorithms are designed by means of jKarma, the fourth one is the method StreamKrimp[3] proposed in [8]. In particular, we have two non-exhaustive PBCDs (PBCD-1 and PBCD-2) and an exhaustive PBCD (KARMA). StreamKrimp is a non-exhaustive PBCD based on frequent itemsets discovered according to the MDL principle, while PBCD-1 and PBCD-2 are variants of the KARMA algorithm, as they are based on frequent subtrees discovered by adopting (i) a beam search approach, and (ii) two different time-window models, that is, landmark window model and sliding window model, respectively. The KARMA algorithm, is obtained by configuring jKarma as shown in Section 4.3. To guarantee a fair comparison, the algorithms have been executed with same minimum frequency and change thresholds (equal to 0.5).

The results (Table 1) show that non-exhaustive PBCDs (PBCD-1, PBCD-2, and StreamKrimp) are more accurate than those exhaustive (KARMA). Moreover, although exhaustive, KARMA is more efficient than StreamKrimp, which is not designed with the current framework (like PBCD-1, PBCD-2 and KARMA). We

1 https://bitbucket.org/jkarma/demo-karma-pbcd/.

2 https://bitbucket.org/jkarma/datasets.

3 https://people.mmci.uni-saarland.de/~jilles/prj/krimp/.

Please cite this article as: A. Impedovo, C. Loglisci, M. Ceci et al., jKarma: A highly-modular framework for pattern-based change detection on evolving data, Knowledge-Based Systems (2019) 105303, https://doi.org/10.1016/j.knosys.2019.105303.

Table 2
Software metadata.

Nr.	(executable) Software metadata description	Please fill in this column
S1	Current software version	1.0.0
S2	Permanent link to executables of this version	https://bitbucket.org/jkarma/jkarma/downloads/jkarma-1.0.0.jar
S3	Legal Software License	Apache License 2.0
S4	Computing platform/Operating System	Linux, macOS, Microsoft Windows.
S5	Installation requirements & dependencies	Java 8
S6	If available, link to user manual - if formally published include a reference to the publication in the reference list	https://bitbucket.org/jkarma/jkarma/wiki/Home
S7	Support email for questions	angelo.impedovo@uniba.it

Table 3
Code metadata.

Nr.	Code metadata description	Please fill in this column
C1	Current code version	1.0.0
C2	Permanent link to code/repository used of this code version	https://bitbucket.org/jkarma/jkarma/commits/tag/1.0.0
C3	Legal Code License	Apache License 2.0
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	Java 8, Maven, Eclipse
C6	Compilation requirements, operating environments & dependencies	Java 8, Maven
C7	If available Link to developer documentation/manual	https://bitbucket.org/jkarma/jkarma/wiki/Home
C8	Support email for questions	angelo.impedovo@uniba.it

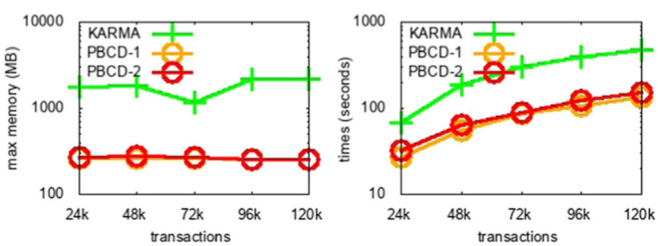

Fig. 2. Memory usage and running times of PBCD-1, PBCD-2, and KARMA as a function of the number of transactions.

also collected results on computational performances of KARMA, PBCD-1 and PBCD-2 working on five synthetic datasets. Fig. 2 shows the memory usage and running times as function of the number of transactions. We see the three algorithms scaling up linearly as the number of transactions grows (the running times and transactions increase of the same magnitude order). PBCD-1 and PBCD-2 are more efficient because implement non-exhaustive search methods. As to the memory usage, there is no substantial correlation with the transactions, which is quite expected because the search space of the patterns (the main subject of memory consumption) is built only once and the subsequent computation marginally influences the memory allocation. This highlights that the framework jKarma investigates a problem common to the several solutions offered, but, at the same time, fully leverages the peculiarities of the most efficient algorithms, in order to guarantee as lower usage of computational resources as possible.

5. Conclusions

We have introduced jKarma, an highly-modular framework for defining and executing customized pattern-based change detection approaches for evolving data, in Java. jKarma enables the modular definition of custom PBCDs, with reduced or none implementation efforts, by following a component-based architectural model. The framework comes as a Java software library which is completely independent from other data mining frameworks and existing data sources, making it integrable into existing projects.

Appendix

Required Metadata

Current executable software version

See Table 2.

Current code version

See Table 3.

References

[1] C. Loglisci, M. Ceci, A. Impedovo, D. Malerba, Mining microscopic and macroscopic changes in network data streams, Knowl.-Based Syst. 161 (2018) 294–312, http://dx.doi.org/10.1016/j.knosys.2018.07.011.

[2] D. Trabold, T. Horváth, Mining strongly closed itemsets from data streams, in: Discovery Science - 20th International Conference, DS 2017, Kyoto, Japan, October 15-17, 2017, Proceedings, 2017, pp. 251–266.

[3] A. Bifet, G. Holmes, R. Kirkby, B. Pfahringer, MOA: massive online analysis, J. Mach. Learn. Res. 11 (2010) 1601–1604.

[4] J. Montiel, J. Read, A. Bifet, T. Abdessalem, Scikit-multiflow: A multi-output streaming framework, J. Mach. Learn. Res. 19 (2018) 72:1–72:5.

[5] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C. Wu, V.S. Tseng, SPMF: a java open-source pattern mining library, J. Mach. Learn. Res. 15 (1) (2014) 3389–3393.

[6] M.J. Zaki, K. Gouda, Fast vertical mining using diffsets, in: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003, 2003, pp. 326–335.

[7] A. Impedovo, M. Ceci, T. Calders, Efficient and Accurate Non-exhaustive Pattern-based Change Detection in Dynamic Networks, in: Discovery Science - 22nd International Conference, DS 2019, Split, Croatia, October 28-30, 2019, Proceedings, pp. 396–411, http://dx.doi.org/10.1007/978-3-030-33778-0_30.

[8] M. van Leeuwen, A. Siebes, Streamkrimp: Detecting change in data streams, in: Machine Learning and Knowledge Discovery in Databases, European Conference, ECML/PKDD 2008, Antwerp, Belgium, September 15-19, 2008, Proceedings, Part I, 2008, pp. 672–687, http://dx.doi.org/10.1007/978-3-540-87479-9_62.

Please cite this article as: A. Impedovo, C. Loglisci, M. Ceci et al., jKarma: A highly-modular framework for pattern-based change detection on evolving data, Knowledge-Based Systems (2019) 105303, https://doi.org/10.1016/j.knosys.2019.105303.