On Reasoning About Black-box UDFs by Classifying Their Performance Characteristics

Michał Bodziony
IBM Poland Software Lab Kraków
Kraków, Poland *michal.bodziony@pl.ibm.com*

Batosz Ciesielski
Poznan University of Technology
Poznań, Poland *bartosz.ciesielski1997@gmail.com*

Anna Lehnhardt
Poznan University of Technology
Poznań, Poland *anula.le@wp.pl*

Robert Wrembel
Poznan University of Technology
Poznań, Poland *robert.wrembel@put.poznan.pl*

Abstract

User defined functions (UDFs) are frequent components of data integration processes, where UDFs are often available only as black boxes, i.e., their semantics and performance characteristics are unknown (such functions are further called BBUDFs). This feature prevents from optimizing the processes. Discovering the semantics of a BBUDF is often impossible due to high complexity of its code. However, discovering the performance model of a BBUDF seems to be feasible with the support of machine learning. In this paper, we present a solution for classifying BBUDFs into performance classes based on their performance characteristics. This way, if a performance class of a given BBUDF is known, it may allow to reason about some hidden features of the BBUDF. Our solution is supported by experimental evaluation, which reveals that our initial approach, in multiple cases, allows to classify BBUDFs to adequate performance classes.

Keywords: data integration process, user defined function, time series, time series similarity measure, time series classification

1. Introduction

Data integration (DI) has been an active research field for already six decades [48, 55]. A common goal of a DI task is to make heterogeneous and typically distributed data available for an end user in a unified format. Research and development in this field resulted in a few standard DI architectures, namely: federated, mediated, data warehouse, lambda, data lake, lake house, polystore, and data mesh, e.g., [9, 11, 14, 18, 46, 51]. In all of these architectures data are transported from source storage systems into an integrated system by means of an integration layer. This layer is implemented by a sophisticated software, which runs the so-called *DI processes* (a.k.a. extract-transform-load - ETL, data processing pipeline, data processing workflows). A DI process runs a sequence of tasks (steps), which ingest data from various sources and preprocess them into formats suitable for analytical and machine learning applications.

For years, the so-called big data (mainly heterogeneous and voluminous) are produced by various devices and systems, e.g., from simple to complex sensors, medical imaging machinery, agri-robots. Pre-processing and analyzing big data is challenging and it often requires a

custom data integration code, called a *user defined function* (UDF), e.g., [17, 56]. Such a code may be implemented in any programming language and it is called from a DI engine as an external program. UDFs are frequently treated as *black-boxes*, since their internal logic and performance characteristics are unknown to a DI process designer (typically, only their input and output parameters are known). Further in this paper such UDFs will be called *black-box UDFs* (BBUDFs). As a consequence, optimization means for DI processes with UDFs are very limited and often impossible.

In order to optimize a DI processes in a cost-based style (e.g., [12], cost models of single tasks in the process as well as an overall cost model of the whole process must be known. BBUDFs in a DI process make building such cost models challenging, since the semantics and performance characteristics of each BBUDF must be discovered. However, discovering the semantics of BBUDFs is very difficult and often impossible. On the contrary, applying machine learning (ML) algorithms to discovering their performance models seems to be promising [54]. In the paper we address this research direction.

The plurality of possible operations implemented and their combinations inside a BBUDF is infinite. Therefore, a natural approach is to start discovering the semantics and performance models of atomic operations and their basic combinations. Having received promising results on simple BBUDFs, the approach can be extended towards handling more complex BBUFDs.

Our solution is based on classifying BBUDFs into performance classes, based on performance characteristics (in the form of time series) produced by BBUDFs. The characteristics (typically CPU and RAM) are assigned labels of known performance classes. The known performance classes are created from performance characteristics of white-box DI tasks. By assigning an unknown BBUDF to a performance class we are able to reason about the BBUDF in terms of what operations it may execute and what are its performance characteristics, for example w.r.t. a data volume being processed. Our solution is supported by experimental evaluation. It reveals that in multiple cases the solution allows to correctly assign a performance class label to a BBUDF, thus it allows to reason about the semantics of the BBUDF.

This paper further provides: (1) the state of the art on black-box opening techniques, in Section 2, (2) the description of our contribution, in Section 3, (3) the description of our test environment and data characteristics, in Section 4, (4) the experimental evaluation of our concept, in Section 5, and (5) the summary and conclusions, in Section 6.

2. Black Box Opening Techniques

The existing solutions to 'opening' an encapsulated software code can be categorized as: (1) annotating, (2) static code analysis, (3) model discovery by means of ML algorithms, (4) efficient compilation and execution, (5) query reverse engineering, and (6) 'opening' by experimentation.

The solutions that use *annotations* assume that a programmer annotates a software with hints. They allow to: (1) figure out automatically some execution characteristics of the software, e.g., a cost function, resource consumption, a number of input and output rows, e.g., [40], (2) execute it in a parallel architecture, e.g., [21, 22], (3) generate an optimized execution plan for relational database operators, e.g., [16]. The annotations instruct a DI engine how to better orchestrate tasks in a workflow or how to execute them in parallel.

A *static code analysis* approach [22] allows to learn features of a Java bytecode. A code analyzer provides a control flow graph and a structure that for each statement associates variables used by the statement.

The *ML-based* solutions analyze performance characteristics (typically, CPU, I/O, and memory usage) collected during a normal execution of a software or an excessive testing phase, in order to learn performance models [19, 36, 47, 53]. A complementary research is being done in the area of learning states of a software. In this field, the solutions are based on the theory of the final state automata (see for example [50] for an overview).

Efficient compilation and execution of UDFs in a cluster was addressed in [7]. Paper [41] shows how to apply lambda functions implemented in C in SQL subqueries, whic are run in PostgreSQL. The goal of a lambda function is either to pre-materialize results of a table function or to return a cursor to a result of the table function. When UDF U is used in query Q, it is called and executed for every record processed by Q, introducing substantial performance overhead. Methods for decorrelating execution of U from Q were proposed in [37, 44]. They assume that the code of U is expressed in a procedural language, possibly with embedded SQL commands. The decorrelation techniques apply similar idea of representing U by means of an equivalent relational algebra expression and treat the expression as a nested sub-query in Q. In [6], the authors address UDFs with block operators. A block operator receives as an input a set of tuple groups and it runs some computations for the groups, returning for each group an output tuple. Finally, [45] proposes a technique for merging multiple UDFs, whose code snippets implement the same computations. To this end, the authors developed a formal language that reflects imperative constructs, including conditional statements and loops. In all of the aforementioned solutions it is assumed that a source code of a UDF is known.

The works on *query reverse engineering*, e.g., [25, 49], are the most related to what we are proposing in this paper. Paper [25] focuses on reverse engineering of project-join SQL queries. The technique is based on the following concepts: (1) eliminating column ambiguity, (2) the ranking of possible combinations of projection columns, (3) finding a right sequence of joins of candidate tables, and (4) validating the results of a constructed query. In [49], reverse engineering of a select-project-join query is modeled as a classification problem, where possible selection predicates are represented as a decision tree.

Finally, the idea behind 'opening' a BBUDF by *experimentation* is based on analyzing the dependencies between output data characteristics produced by a given BBUDF in response to a given input data set of given characteristics. The characteristics that describe the input and output data include: the number of input and output attributes, the types of input and output attributes, the number of input and output rows. Based on these characteristics and rules, it is deduced what type of operation the BBUDF performs. In [4] the authors showed that this technique can be successfully applicable to only simple BBUDFs (implementing filtering, projection, filtering+projection), as in general the problem transforms to the Boolean Satisfiability Problem (SAT) [32] (belonging to the NP-complete class).

Complementary research results on optimizing the execution of UDFs (treated as white boxes) in databases were published in [38, 43]. It is assumed in these publications that a UDF is translated either into relational algebraic expressions [38] or it is compiled into a low level executable code, merged with a compiled SQL code [43] for execution in a DBMS. The solution proposed in [13] allows to parallelize UDFs in the map-reduce framework, but UDFs must be implemented from scratch for this framework. In [1] the authors describe the so-called parallelization skeletons, which are code templates used for implementing UDFs for parallelized execution.

In the context of the contribution of this paper, there are three common limitations of all the outlined approaches. First, none of them addresses the problem of discovering the semantics or performance classes of BBUDFs. Second, they do no apply ML techniques to build performance models of BBUDFs. Third, the approaches do not support the classification of BBUDFs based on their performance models that, in turn, may ease in some cases reasoning about the semantics of BBUDFs.

3. Our Approach

We base our approach on the following three components, as shown in Figure 1: (1) the repository of performance characteristics of known UDFs; it includes CPU and RAM usage represented as time series (TSs), (2) classification algorithms for TSs and similarity measures for TSs.

We assume that classes of performance characteristics of known (white-box) UDFs have been built on TSs obtained from excessive experiments, based on the content of the repository. All TSs stored in the repository are labeled with performance classes, assigned by means of a classification algorithm. Notice that to classify TSs a similarity measure is needed (see Section 3.2). When a BBUDF is made available in the system, its performance characteristics are collected by means of experiments. Next, the classification algorithm assigns a class to the characteristics of the BBUDF at hand.

Knowing the performance class of the BBUDF one can get some insights (with a certain level of probability) on the kind of operations run and performance (for example w.r.t. a data volume) of the BBUDF by analyzing other (known) UDFs belonging to the same class. Moreover, in some cases, understanding on how to parallelize the BBUDF could also be figured out. Being able to apply parallelization to a given BBUDF is a step towards its performance optimization.

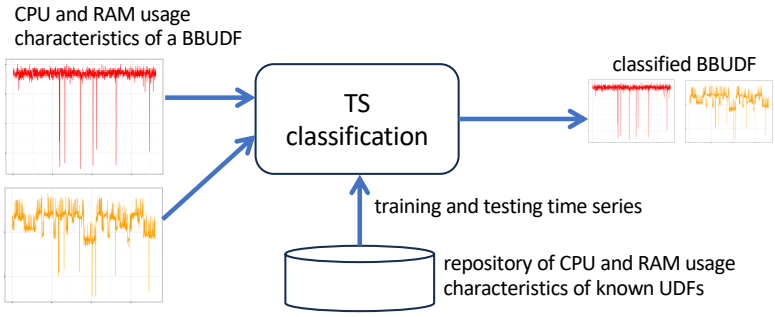

Fig. 1. A schematic illustration of the approach of discovering performance models of BBUDFs contributed in this paper

In our setting, TSs describe CPU usage and RAM usage by UDFs. TS classification consists in assigning a class label to a given TS based on the shape of the TS. Notice that the amplitude of a TS and its length often should not be taken into account by a classification algorithm, since they depend on the size of a data set processed. For example, filtering applied to a data set consisting of 1M of rows will last shorter and use less resources than filtering applied to a data set of 10M of rows. However, in both cases the shape of both TSs will expose similar landmark [24] characteristics. This observation was confirmed by our preliminary experiments. The difference in amplitude is typically removed by applying normalization. Two normalization techniques are the most frequently used, namely *min-max* [19] and *z-score* [36]. *min-max* is suggested to be used when the distribution of values is unknown or is non-normal. For this reason, in our approach we used *min-max*.

3.1. TS Classification

The following TS classification algorithms are the most popular, namely: *ROCKET* and *HIVE-COTE*, and *kNN*.

ROCKET [10] uses random convolutional kernels to transform TSs. These transforms are next used as classification features. Rocket computes two aggregate features from each kernel and feature convolution. The number of kernels defaults to 10000, i.e., 20000 features are generated by default (in our experiments we used the default number of kernels). Next, the features are ingested by a classification algorithm (linear classification algorithms like ridge regression or logistic regression are recommended).

HIVE-COTE, in versions 1.0 [27] and 2.0 [34] are ensemble classifiers. Version 2.0 uses the following component classifiers: Shapelet Transform, an ensemble of ROCKET classifiers,

Temporal Dictionary Ensemble, and Diverse Representation Canonical Interval Forest. Each component classifier is trained independently and it produces a probability estimate for each class. Then the final output is aggregated by combining the probabilities, which are weighted by an estimate of the quality of the module found on the training data. Since it was shown that version 2.0 produces very promising classification models [34], in our research we applied this version.

TS classification was proved to be efficient with the well-known *kNN* classification algorithm, e.g., [26, 33]. Furthermore, in [31] the authors showed that by applying the DTW similarity, the classification accuracy of TS reached approximately 80%. For this reason, in our approach we also applied *kNN* combined with *DTW* (see Section 3.2, further in this paper called *kNN-DTW*.

3.2. TS Similarity: DTW

Four the most popular similarity measures for time series include: (1) Euclidean distance, (2) longest common subsequence [2, 8], (3) landmark similarity [24, 35], and (4) dynamic time warping (DTW). Because DTW turned out to be the most popular end effective method for assessing TSs similarity [20, 29, 39, 57], we report its results in this paper. We have also run experiments with the other distances, but we do not report them here due to a space limit and because they offered worse results than DTW.

Dynamic time warping (DTW) [3] is used to compute a distance between two TSs, which reflects how similar these two TSs are. TSs being compared may differ in length and in phase, i.e., one TS may be shifted in time w.r.t. to the other one. The main idea of DTW is to compute the distance between similar points (curves) of two TSs. Typically, the Euclidean distance is computed between the similar points, but other distance measures can be applied, e.g., the Gover distance [15]. DTW compares the amplitude (the value of Y in the 2-dimensional space) of the first TS at time t with the amplitude of the second TS at time ($t-1$ and $t+1$) or ($t-2$ and $t+2$). This way, similar shapes, but in different phases in both TSs, contribute to a lower distance, thus a higher similarity. The following fundamental rules define how points in the compared TSs are related: (1) the begin and end point of TS1 and TS2 must be matched, (2) matching points in TS1 and their corresponding points in TS2 must be monotonically increasing, and vice versa (i.e., the lines drawn between matching points must not intersect), (3) matching points in TS1 and their corresponding matching points in TS2 cannot exceed a predefined distance, and vice versa.

4. Test Data Characteristics and Environment

Test data were collected during excessive experiments that were run in a micro-cluster composed of 9 physical workstations, running Ubuntu 20.04.2. One node was designated as master and the remaining eight nodes were designated as workers. All the workstations were equipped with: Intel Core 2 Quad CPU Q9650 3GHz, 8GB RAM DDR3-1333MHz, HDD Seagate ST3500418AS 500GB, and network card Intel Corporation 82567LM-3 1Gbit/s.

Five UDFs were tested in the micro-cluster and their CPU and RAM usage characteristics were collected. The UDFs implemented: *filtering*, *aggregation*, *filtering-aggregation*, *filtering-join*, *filtering-aggregation-join*. The UDFs processed real data on thermal energy consumption generated by sensors. Data were organized into records, each of which included 12 attributes, of types: numeric, short character string, and timestamp. Each UDF was processing two distinct data volumes: 1GB and 2GB. The data were stored in csv files.

Each UDF was run 25 times for the same data volume. For each run, its CPU and RAM performance characteristics were collected every 1/15 sec. These characteristics were stored in 33600 raw csv files. The 25 measurements for a given UDF were: (1) averaged, (2) min-max

normalized, (3) smoothed within 1-second, 2-seconds, and 3-seconds window, and (4) stored in another set of csv files. Such pre-processed data were **input into the classification algorithms**. Examples of CPU usage (in msec) of two UDFs, which implement filtering on two different attributes on the 2GB data set, smoothed within 1-second window, are shown in Figure 2.

Our preliminary experiments showed that **the best classification quality** was obtained on **min-max normalized and smoothed data in 1-second window**. Therefore, these results are outlined in this section. Moreover, we run additional experiments where we compared the similarity distances mentioned in Section 3.2. From those experiments **the most promising results were returned by DTW**. For this reasons, the experiments outlined in this paper use DTW.

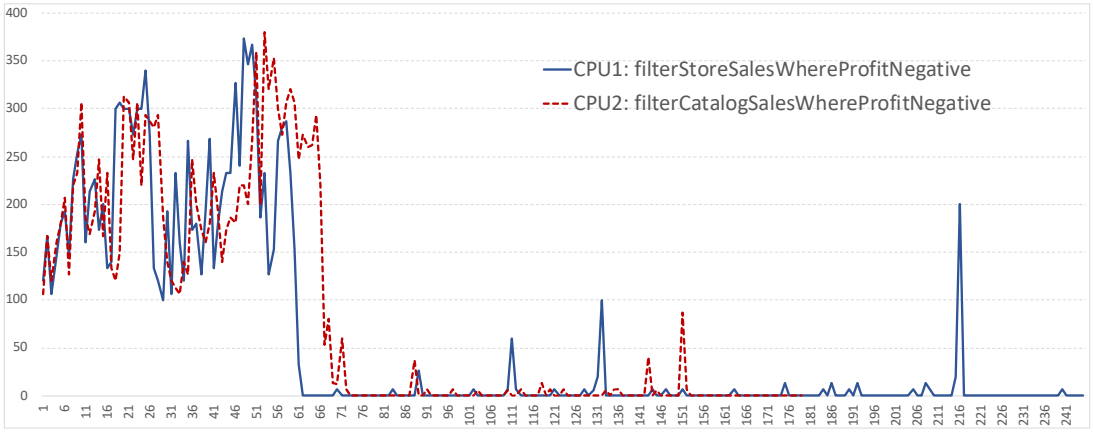

Fig. 2. Example CPU usage characteristics (in msec) of two UDFs filtering 2GB data set, on two different attributes, smoothed within 1-second window

The project reported in this paper is **open and available in a GitHub repository**, available at: `https://github.com/Put-MasterThesis-TimeSeries-Sim`. The project repository includes: (1) all the raw CPU and RAM usage characteristics (stored in 33600 csv files), (2) pre-processed files (min-max normalized, min-max normalized and smoothed within 1-second, 2-seconds, and 3-seconds window), (3) Python scripts (notebooks) for data visualization, normalization, smoothing, and computing averages, (4) Python scripts for training classification algorithms *ROCKET* and *HIVE-COTE-v2*, (5) Python scripts for *kNN-DTW* classification, (6) Python scripts for assessing TSs similarity measures and classification quality. All the tasks of data pre-processing, model building, and assessing are fully documented and organized in sequences of steps in Python notebooks. The repository includes also other experiments not reported here due to space limit.

5. Experimental Evaluation

The goal of the experimental evaluation was to: (1) evaluate the DTW distance w.r.t. the values it produces for various similar and dissimilar TSs; (2) evaluate the classification algorithms for TSs, outlined in Section 3.1, w.r.t. their quality, represented by the F1 measure; (3) evaluate the classification algorithms for TSs w.r.t. their time performance.

5.1. Results: Distance Values Produced by DTW

The DTW distance (see Section 3.2) was assessed w.r.t. the values it produced for various similar and dissimilar TSs. To this end, we compared with each other the DTW distances between TSs produced by five classes of UDFs (see Section 4). For each UDF, their raw results for 1GB and 2GB data sets were aggregated into a single TS. Figure 3 presents heatmaps for CPU and RAM

usage. Each cell in these heatmaps represents a distance between two classes of UDFs. Notice that, the lower the distance the more similar two UDFs are. As we can observe in the CPU heatmap, all the diagonal cells from top-left to bottom-right include the lowest values. It means that UDFs of the same class, e.g., *filtering-filtering* have the shortest distances, i.e., they are the most similar.

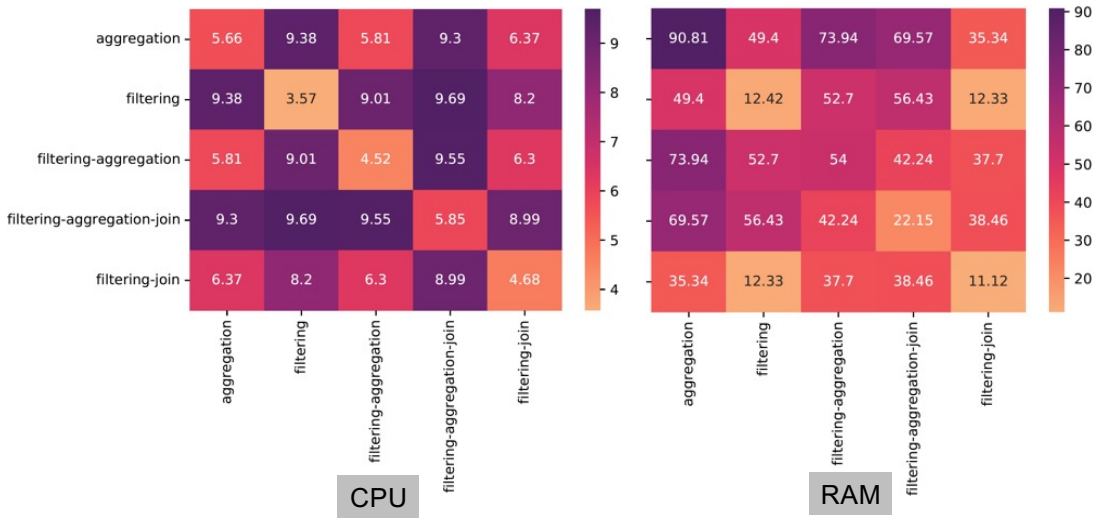

Fig. 3. Heatmaps for CPU and RAM usage characteristics, computed by means of DTW on data min-max normalized and smoothed within 1-second window

In the RAM heatmap, the same behaviour was confirmed for UDFs implementing: *filtering*, *filtering-aggregation-join*, and *filtering-join*. We suspect that the RAM usage characteristic for *aggregation* has substantially smaller RAM footprint than for *filtering*, thus the RAM usage characteristic of *filtering* dominates the usage of aggregation in *filtering-aggregation*. Similarly, the RAM usage of a *join* dominates the usage of *aggregation* in *filtering-aggregation-join*, which distorts the results involving *aggregation*.

5.2. Results: Classification Quality

ROCKET and *HIVE-COTE* are supervised learning algorithms that support multi-variate TSs. Therefore, they were trained on both the CPU and RAM usage characteristics. The characteristics obtained from running the UDFs on 1GB and 2GB data sets were combined. From such data sets, 2/3 of rows were randomly selected as training data and 1/3 as testing data, as suggested for example in [5, 28, 30, 42, 52]. On the contrary, *kNN-DTW* is an unsupervised learning algorithm - for it in our experiments k={1, 3, 5}. The results presented in this section represent averages for the three values of k.

As the classification quality we selected *F1*, which is the commonly used measure. Its values for the three tested algorithms and five UDFs are shown in Figure 4. As we can observe from the chart, TSs classification with *ROCKET* resulted in the highest F1 score for all the tested UDFs except *filtering*. For this UDF, *HIVE-COTE-v2* offered slightly better classification quality. From Figure 4 we can also observe that the most difficult operations to classify turned out to be *filtering-aggregation* and *filtering-aggregation-join*. Again, we suspect that the CPU and RAM usage characteristics by the *aggregation* component were dominated by those of *filtering* and *join*. The worst results were returned by *kNN-DTW* and with such low values of F1, the algorithm is not recommended to be applied to the problem addressed in this paper.

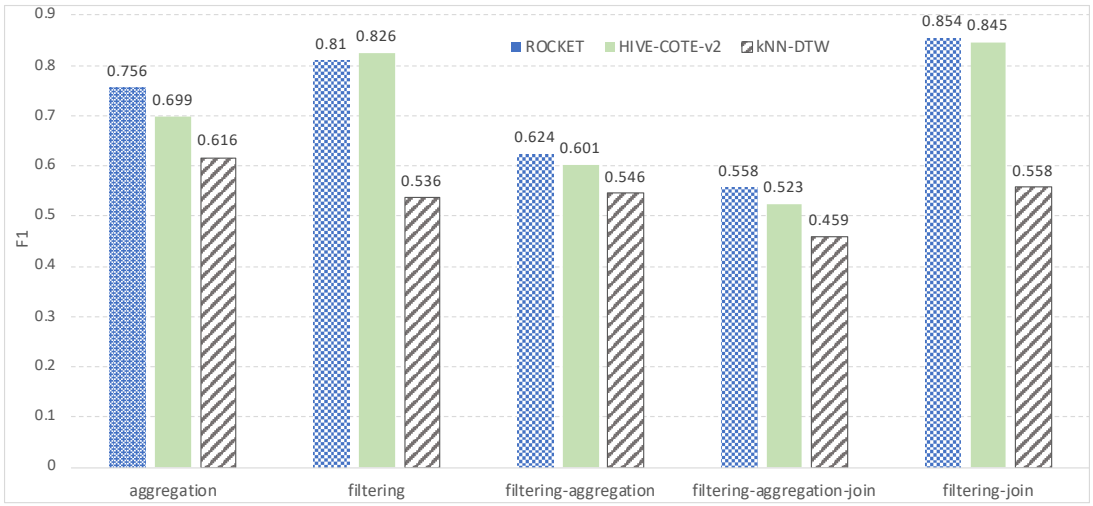

Fig. 4. Values of F1 for *ROCKET*, *HIVE-COTE*, and *kNN-DTW* computed on data min-max normalized and smoothed within 1-second window

5.3. Results: Classification Time Performance

Table 1 shows average processing times and their standard deviations of the tested classification algorithms, namely *ROCKET*, *HIVE-COTE-v2*, and *kNN-DTW*. Here, by processing time we mean time spent on computing similarities between TSs and time spent on classifying unknown TSs. The averages were computed from 10 runs of the algorithms in the setup described in Section 5.2. As we can observe from the table, the fastest is *ROCKET* and the slowest is *kNN-DTW*. The most time-expensive part of *kNN-DTW* is the DTW algorithm, which contributes to such a long processing time of this classification algorithm.

Table 1. Average processing time (from 10 runs) of classification algorithms *ROCKET*, *HIVE-COTE-v2*, and *kNN-DTW*

	AVG process. time [min]	STDEV
ROCKET	1.1	0.2
HIVE-COTE-v2	14.1	2.5
kNN-DTW	256.5	22.6

6. Summary and Conclusions

In this paper we addressed the problem of handling black-box UDFs in data integration processes, with the final goal of being able to optimize the execution of such processes. BBUDFs pose a serious challenge, as their semantics is very difficult and often impossible to discover. To make this problem easier, we proposed to classify performance characteristics of BBUDFs to known classes of performance models. Based on the classification, one can reason about the performance and scalability of a given BBUDF for various volumes of data processed. If the performance model of a BBUDF is known, i.e., it belongs to a given performance class, then such knowledge increases the means of optimizing a DI process that uses the BBUDF, for example by parallelization. Moreover, knowing the semantics of other UDFs belonging to the same class, one may further reason about a possible functionality implemented by such a BBUDF.

To make our approach working, we used three core components, namely: (1) the repository of performance characteristics, (2) similarity measures for TSs, and (3) classification algorithms

for TSs. We evaluated the approach on five different simple UDFs that are the fundamental building blocks of data integration processes (e.g., [23]). The obtained results showed that three UDFs were classified at a satisfactory value of the F1 measure, thus we were able to reason about the performance class and types of operations implemented by these UDFs. Unfortunately, UDFs that implemented aggregation and aggregation combined with join and filtering were much more difficult to classify correctly.

Since the problem of discovering performance characteristics and the semantics of BBUDFs is very difficult, we addressed it by experimenting with basic building blocks of DI processes, to verify whether our approach would be feasible. The next natural step in our research will focus on: (1) discovering classes of performance models of more complex code snippets implementing BBUDFs and (2) experimenting with neural networks for building classes of performance models.

Notice that, the problem of BBUDFs in DI processes can be generalized to building performance models of code snippets, which is important in real applications of performance optimization, including: (1) designing scalable execution environments (usually parallel); (2) allocating adequate computing resources for an efficient execution of a code snippet; (3) optimizing execution plans of DI processes and database queries. The current stage of the world wide research does not allow yet to fully 'open' BBUDFs, but we believe that this is a promising research direction. For this reason, the initial work presented in this paper may open new paths in research on optimization of systems with software components of unknown semantics.

Acknowledgements. The work of Michał Bodziony is related to his employment at IBM Polska Sp z o.o. Additionally, his work is supported by the Applied Doctorate grant no. DWD/4/24/2020 from the Polish Ministry of Education and Science.

References

[1] Ali, S. M. F., Mey, J., and Thiele, M.: Parallelizing user-defined functions in the ETL workflow using orchestration style sheets. In: *Int. Journal of Applied Mathematics and Computer Science* 29.1 (2019).

[2] Apostolico, A.: "String Editing and Longest Common Subsequences". In: *Handbook of Formal Languages, Volume 2. Linear Modeling: Background and Application*. Springer, 1997.

[3] Berndt, D. J. and Clifford, J.: Using Dynamic Time Warping to Find Patterns in Time Series. In: *Workshop Knowledge Discovery in Databases*. AAAI Press, 1994.

[4] Bodziony, M., Krzyzanowski, H., Pieta, L., and Wrembel, R.: On discovering semantics of user-defined functions in data processing workflows. In: *SIGMOD Workshops*. ACM, 2021.

[5] Brownlee, J.: *Train-Test Split for Evaluating Machine Learning Algorithms*. `https://machinelearningmastery.com/train-test-split-for-evaluating-machine-learning-algorithms/`. Accessed Jul, 2023. 2020.

[6] Chen, Q., Wu, R., Hsu, M., and Zhang, B.: Extend core UDF framework for GPU-enabled analytical query evaluation. In: *Int. Database Engineering and Applications Symposium (IDEAS)*. 2011.

[7] Crotty, A., Galakatos, A., Dursun, K., Kraska, T., Binnig, C., Cetintemel, U., and Zdonik, S.: An Architecture for Compiling UDF-Centric Workflows. In: *VLDB Endow.* 8.12 (2015).

[8] Das, G., Gunopulos, D., and Mannila, H.: Finding Similar Time Series. In: *European Symposioum Principles of Data Mining and Knowledge Discovery*. Vol. 1263. LNCS. Springer, 1997, pp. 88–100.

[9] Dehghani, Z.: Data Mesh: Delivering Data-Driven Value at Scale. O'Reilly, 2022.

[10] Dempster, A., Petitjean, F., and Webb, G. I.: ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels. In: *Data Mining and Knowledge Discovery* 34.5 (2020).

[11] Errami, S. A., Hajji, H., Kadi, K. A. E., and Badir, H.: Spatial big data architecture: From Data Warehouses and Data Lakes to the LakeHouse. In: *Journal of Parallel and Distributed Computing* 176 (2023).

[12] Forresi, C., Francia, M., Gallinucci, E., and Golfarelli, M.: Cost-based Optimization of Multistore Query Plans. In: *Information Syst. Frontiers* 25.5 (2023).

[13] Friedman, E., Pawlowski, P., and Cieslewicz, J.: SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. In: *VLDB Endow.* 2.2 (2009).

[14] Gillet, A., Leclercq, É., and Cullot, N.: Lambda+, the Renewal of the Lambda Architecture: Category Theory to the Rescue. In: *Int. Conf. Advanced Information Systems Engineering (CAiSE)*. LNCS 12751. Springer, 2021.

[15] Gower, J. C., Roux, N. J. le, and Gardner-Lubbe, S.: The Canonical Analysis of Distance. In: *Journal of Classification* 31.1 (2014).

[16] Große, P., May, N., and Lehner, W.: A study of partitioning and parallel UDF execution with the SAP HANA database. In: *Conf. on Scientific and Statistical Database Management (SSDBM)*. 2014.

[17] Grulich, P. M., Zeuch, S., and Markl, V.: Towards Efficient and Secure UDF Execution with BabelfishLib (Lightning Talk). In: *Proc. of VLDB Workshops*. Vol. 3462. CEUR-WS.org, 2023.

[18] Hai, R., Koutras, C., Quix, C., and Jarke, M.: Data Lakes: A Survey of Functions and Systems. In: *IEEE Trans. Knowl. Data Eng.* 35.12 (2023).

[19] Hernández, Á. B., Pérez, M. S., Gupta, S., and Muntés-Mulero, V.: Using machine learning to optimize parallelism in big data applications. In: *Future Generation Computer Systems* 86 (2018).

[20] Herrmann, M., Tan, C. W., and Webb, G. I.: Parameterizing the cost function of dynamic time warping with application to time series classification. In: *Data Mining and Knowledge Discovery* 37.5 (2023).

[21] Hueske, F., Peters, M., Krettek, A., Ringwald, M., Tzoumas, K., Markl, V., and Freytag, J.-C.: Peeking into the optimization of data flow programs with mapreduce-style UDFs. In: *Int. Conf. on Data Engineering (ICDE)*. 2013.

[22] Hueske, F., Peters, M., Sax, M. J., Rheinländer, A., Bergmann, R., Krettek, A., and Tzoumas, K.: Opening the black boxes in data flow optimization. In: *VLDB Endowment* 5.11 (2012).

[23] IBM: *Product documentation. InfoSphere Information Server v.11.7*. `https://www.ibm.com/docs/en/iis/11.7?topic=jobs-processing-data`. 2024.

[24] Jagadish, H. V.: Review - Landmarks: a New Model for Similarity-based Pattern Querying in Time Series Databases. In: *ACM SIGMOD Digit. Rev.* 1 (1999).

[25] Kalashnikov, D. V., Lakshmanan, L. V., and Srivastava, D.: FastQRE: Fast Query Reverse Engineering. In: *SIGMOD*. 2018.

[26] Levchenko, O., Kolev, B., Yagoubi, D. E., Akbarinia, R., Masseglia, F., Palpanas, T., Shasha, D. E., and Valduriez, P.: BestNeighbor: efficient evaluation of kNN queries on large time series databases. In: *Knowledge and Information Systems* 63.2 (2021), pp. 349–378.

[27] Lines, J., Taylor, S., and Bagnall, A. J.: HIVE-COTE: The Hierarchical Vote Collective of Transformation-Based Ensembles for Time Series Classification. In: *IEEE Int. Conf. on Data Mining (ICDM)*. 2016.

[28] Liu, X., Chang, W., Yu, H., Hsieh, C., and Dhillon, I. S.: Label Disentanglement in Partition-based Extreme Multilabel Classification. In: *Annual Conf. Advances in Neural Information Processing Systems (NeurIPS)*. 2021.

[29] Liu, Y., Zhang, Y., Zeng, M., and Zhao, J.: A novel distance measure based on dynamic time warping to improve time series classification. In: *Information Sciences* 656 (2024).

[30] Mahankali, A. V. and Woodruff, D. P.: Linear and Kernel Classification in the Streaming Model: Improved Bounds for Heavy Hitters. In: *Annual Conf. Advances in Neural Information Processing Systems (NeurIPS)*. 2021.

[31] Mahato, V., O'Reilly, M., and Cunningham, P.: A Comparison of k-NN Methods for Time Series Classification and Regression. In: *Irish Conf. on Artificial Intelligence and Cognitive Science*. Vol. 2259. CEUR-WS.org, 2018.

[32] Malik, S. and Zhang, L.: Boolean satisfiability from theoretical hardness to practical success. In: *Commun. ACM* 52.8 (2009).

[33] Martínez, F., Frías, M. P., Charte, F., and Rivera, A. J.: Time Series Forecasting with KNN in R: the tsfknn Package. In: *The R Journal* 11.2 (2019).

[34] Middlehurst, M., Large, J., Flynn, M., Lines, J., Bostrom, A., and Bagnall, A. J.: HIVE-COTE 2.0: a new meta ensemble for time series classification. In: *Machine Learning* 110.11 (2021), pp. 3211–3243.

[35] Perng, C., Wang, H., Zhang, S. R., and Jr., D. S. P.: Landmarks: a New Model for Similarity-based Pattern Querying in Time Series Databases. In: *Int. Conf. on Data Engineering*. IEEE Computer Society, 2000, pp. 33–42.

[36] Pumma, S., Feng, W., Phunchongharn, P., Chapeland, S., and Achalakul, T.: A runtime estimation framework for ALICE. In: *Future Generation Computer Systems* 72 (2017).

[37] Ramachandra, K. and Park, K.: BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid. In: *VLDB Endow.* 12.12 (2019).

[38] Ramachandra, K., Park, K., Emani, K. V., Halverson, A., Galindo-Legaria, C. A., and Cunningham, C.: Froid: Optimization of Imperative Programs in a Relational Database. In: *VLDB Endow.* 11.4 (2017).

[39] Rasines, I., Remazeilles, A., Prada, M., and Cabanes, I.: Minimum Cost Averaging for Multivariate Time Series Using Constrained Dynamic Time Warping: A Case Study in Robotics. In: *IEEE Access* 11 (2023).

[40] Rheinländer, A., Heise, A., Hueske, F., Leser, U., and Naumann, F.: SOFA: An extensible logical optimizer for UDF-heavy data flows. In: *Information Systems* 52 (2015).

[41] Schüle, M. E., Huber, J., Kemper, A., and Neumann, T.: Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In: *SSDBM*. ACM, 2020.

[42] Shao, Z., Bian, H., Chen, Y., Wang, Y., Zhang, J., Ji, X., and Zhang, Y.: TransMIL: Transformer based Correlated Multiple Instance Learning for Whole Slide Image Classification. In: *Annual Conf. Advances in Neural Information Processing Systems (NeurIPS)*. 2021.

[43] Sichert, M. and Neumann, T.: User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. In: *VLDB Endow.* 15.5 (2022).

[44] Simhadri, V., Ramachandra, K., Chaitanya, A., Guravannavar, R., and Sudarshan, S.: Decorrelation of user defined function invocations in queries. In: *Int. Conf. on Data Engineering (ICDE)*. 2014.

[45] Sousa, M., Dillig, I., Vytiniotis, D., Dillig, T., and Gkantsidis, C.: Consolidation of queries with user-defined functions. In: *ACM SIGPLAN PLDI*. ACM, 2014.

[46] Tan, R., Chirkova, R., Gadepally, V., and Mattson, T. G.: Enabling query processing across heterogeneous data models: A survey. In: *IEEE Int. Conf. on Big Data*. 2017.

[47] Tang, W., Desai, N., Buettner, D., and Lan, Z.: Job scheduling with adjusted runtime estimates on production supercomputers. In: *Journal of Parallel and Distributed Computing* 73.7 (2013).

[48] Timakum, T., Lee, S., Hu, H., Song, I., and Song, M.: DOLAP: A 25 Year Journey Through Research Trends and Performance (Invited Paper). In: *Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*. Vol. 3653. CEUR-WS.org, 2024.

[49] Tran, Q. T., Chan, C.-Y., and Parthasarathy, S.: Query Reverse Engineering. In: *The VLDB Journal* 23.5 (2014).

[50] Vaandrager, F. W.: Model learning. In: *Commun. ACM* 60.2 (2017).

[51] Vaisman, A. A. and Zimányi, E.: Data Warehouse Systems - Design and Implementation, Second Edition. Data-Centric Systems and Applications. Springer, 2022.

[52] Wickramanayake, S., Hsu, W., and Lee, M.: Explanation-based Data Augmentation for Image Classification. In: *Annual Conf. Advances in Neural Information Processing Systems (NeurIPS)*. 2021.

[53] Witt, C., Bux, M., Gusew, W., and Leser, U.: Predictive performance modeling for distributed batch processing using black box monitoring and machine learning. In: *Information Systems* 82 (2019).

[54] Wrembel, R.: Optimizing Data Integration Processes with the Support of Machine Learning - Is it really possible? In: *Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*. Vol. 3653. CEUR Workshop Proceedings. 2024.

[55] Wrembel, R., Abelló, A., and Song, I.: DOLAP data warehouse research over two decades: Trends and challenges. In: *Information Systems* 85 (2019).

[56] Yamada, M., Kitagawa, H., Amagasa, T., and Matono, A.: Augmented lineage: traceability of data analysis including complex UDF processing. In: *The VLDB Journal* 32.5 (2023).

[57] Zhang, Q., Zhang, C., Cui, L., Han, X., Jin, Y., Xiang, G., and Shi, Y.: A method for measuring similarity of time series based on series decomposition and dynamic time warping. In: *Applied Intelligence* 53.6 (2023).