Martin Hahmann, Claudio Hartmann, Lars Kegel, Dirk Habich, Wolfgang Lehner

Big by blocks: modular analytics

Erstveröffentlichung in / First published in:

Information Technology. 2016, 58 (4), S. 176–185 [Zugriff am: 05.11.2020]. De Gruyter. ISSN 2196-7032

DOI: https://doi.org/10.1515/itit-2016-0003

Special Issue

Martin Hahmann*, Claudio Hartmann, Lars Kegel, Dirk Habich, and Wolfgang Lehner

Big by blocks: modular analytics

DOI 10.1515/itit-2016-0003
Received January 20, 2016; accepted May 9, 2016

Abstract: Big Data and Big Data analytics have attracted major interest in research and industry and continue to do so. The high demand for capable and scalable analytics in combination with the ever increasing number and volume of application scenarios and data has lead to a large and intransparent landscape full of versions, variants and individual algorithms. As this zoo of methods lacks a systematic way of description, understanding is almost impossible which severely hinders effective application and efficient development of analytic algorithms. To solve this issue we propose our concept of modular analytics that abstracts the essentials of an analytic domain and turns them into a set of universal building blocks. As arbitrary algorithms can be created from the same set of blocks, understanding is eased and development benefits from reusability.

Keywords: Data analysis, clustering, forecasting, algorithm description.

ACM CCS: Theory of computation → Design and analysis of algorithms → Algorithm design techniques, Theory of computation → Theory and algorithms for application domains → Machine learning theory

1 Introduction

Big Data has been "all the rage" in recent years and, as a topic, has spawned a multitude of research and development projects. Regarding analytics, these efforts have been focused on improving the scalability as well as analysis performance of algorithms, i.e., enabling them to process and exploit the ever-growing mass of data. While scalability is generally realized via parallelization and adaption to specific system architectures, analysis quality is optimized by tightly fitting an algorithm to a specific application scenario or domain. Due to this practice, a zoo of individual algorithms exists, which is continually expanded with new members. While these new algorithms still contain genuine novel approaches, the majority of them are slightly modified variants, scenario specific adaptations or scalable versions of existing methods. Utilizing this large pool of algorithms is a considerable challenge that is further complicated by a lack of systematic descriptions. In general, algorithm descriptions, either in the form of program code or text, focus on implementation specifics or integration details of certain domain knowledge. Thus, similarities and relationships between algorithms are obscured, which makes it really hard to comprehend *how* algorithms work. The resulting lack of understanding hinders effective application of analytic algorithms as not every method fits every scenario. Furthermore, efficient modification of algorithms is complicated as optimization potentials or correct implementations of changes might not be recognized.

Recognizing potentials for improvement and finding novel promising approaches for analysis are challenges of algorithm design that actually predate the big data era. However, the fast growth of data has acted like a catalyst for the significance of tasks like algorithm adaptation and optimization. Algorithm adaptation is mostly driven by the *Variety* of big data that demands the inclusion of more and more data sources into analysis in order to improve result quality. The ever increasing *Volume* of data and the *Velocity* in which it is generated, pushes optimization in order to ensure scalability and include concepts like parallelization, optimized data structures, and modern hardware architectures into analytic algorithms. As big data expands, constantly and fast, the implementation of analytic algorithms must be agile and happen in short-cycles in order to keep pace with emerging data sources or execution platforms.

Currently this is not the case, as we will illustrate in the following by regarding the three existing general approaches for implementation of analytic algorithms. The first one uses standard programming languages like *C*. It allows the development of high-performance algorithms that utilize customized execution models and data struc-

***Corresponding author: Martin Hahmann,** Technische Universität Dresden, Dep. of Computer Science, Nöthnitzer Str. 46, D-01062 Dresden, Germany,
e-mail: martin.hahmann@tu-dresden.de
Claudio Hartmann, Lars Kegel, Dirk Habich, Wolfgang Lehner: Technische Universität Dresden, Dep. of Computer Science, Nöthnitzer Str. 46, D-01062 Dresden, Germany

tures in order to exploit domain knowledge and specific system architectures to the fullest. A slightly different approach is offered by analysis-specific development environments like the R project [13] or SystemML [8]. While not offering full customizability, these still provide specific data structures, execution models, and optimizers for analytics in general. The most user-friendly approach are workflow frameworks like KNIME [4] that offer prefabricated operators which are connected to form analytical processes. This offers a certain degree of reuse regarding adaptations, but limits the potential for optimization. Furthermore, major algorithm changes require the creation of new operators in an underlying programming language. In general, implementation considers analytic algorithms as individual monolithic entities that must be completely remade in order to be adapted or optimized. This, of course, makes implementation costly and time intensive.

In this article, we break with this philosophy and propose a contrasting view by interpreting algorithms as combinations of general modules instead. Basically, we want to achieve a user-experience similar to the operator combination of workflow frameworks like KNIME, but with a finer granularity. We want to apply modifications on a level that is more detailed than changing an operator/algorithm completely, but still not as detailed as the specification using programming languages. For this, we interpret an algorithm as a structured combination of interchangeable components. The composition of an algorithm is defined by a base template which is equipped with components that capsule an abstract functionality rather than a programming instruction. This approach enables agile and user-friendly modifications, as developers can simply change components in order to realize their desired algorithm design. This shortens development cycles and allows users to keep pace with the dynamics of big data.

To create these modules, we examine different analysis techniques, abstract from their algorithms, extract common similarities, and create sets of universal building blocks that can tackle the challenges mentioned earlier. In doing so, a conceptual model is created, that allows the formal and platform independent description of algorithms for a specific analysis technique. Besides description, our modular approach offers further potential benefits. As each building block formalizes a certain functionality, understanding and comparing different algorithms becomes more structured. By exploiting this, taxonomies and systematic approaches for selecting the appropriate method for a given use-case could be derived. Furthermore, our approach could be used as a frontend language for actual analytic systems like SystemML. By using system-specific compilers, our formal abstract descriptions could be realized on a variety of systems, without forcing users to be proficient in the target systems development mechanisms.

In the following two sections, we illustrate our building blocks concept for the analytic domains of clustering and forecasting. We describe the respective conceptual models with base templates, building blocks, and example algorithms. The building blocks for clustering have already been proposed in [9]. By translating our modularization concept to the domain of forecasting, we create a novel set of building blocks, which shows the versatility of our approach. We conclude the paper by discussing the future development of our concept and its potential for implementation and application.

2 Clustering

We begin the creation of a general set of building blocks for clustering, by decomposing the corresponding algorithms into their conceptual components. We concentrate only on the core clustering procedure and do not consider pre- and post-processing tasks like feature selection etc. As a starting point, we assume a general definition of data clustering [10]: *“Data clustering is the partitioning of a set of points into groups – so called clusters – in a way that similar points are put in the same cluster, while dissimilar points are located in different clusters.”*

From this definition, we derive the essential steps and components of a clustering algorithms in Section 2.1, turn them into a formalism for building blocks in Section 2.2, and show some algorithm examples in Section 2.3.

2.1 Base template

Based on the initial definition, we can identify three tasks, which are observable in all clustering algorithms: measuring the similarity of points, choosing the points that are similar and should be grouped together, and actual grouping of these points. As result of this abstraction, we define three *phases* of a clustering algorithm, that form a base template which needs to be fitted with building blocks to define an algorithm. In the following, we introduce each phase and investigate its defining elements, in order to find such blocks.

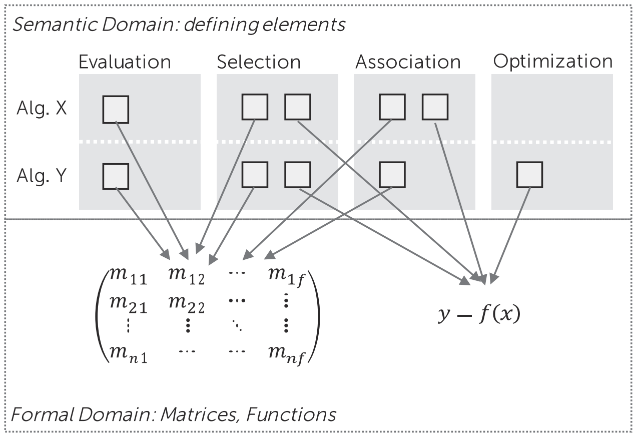
Evaluation Phase: During this first phase, the similarity of points is measured. Similarity between objects is determined with either a: (i) similarity function or (ii) a distance function [10]. While the former expresses the degree of equality, the latter points out the amount of disagreement between objects. As both options are analogous, we assume that similarity is expressed through distances. Based on this, the *distance measure* becomes the first defining element of the evaluation phase.

A distance measure takes at least two input values and produces one output value. One input contains the *points* which are to be clustered, the second input is variable. Algorithms like DBSCAN [6] calculate all point-to-point distances and thus, reuse *points* as second input. However, approaches like k-means [7] utilize a special set of representatives as second input for the distance computation. We unify both alternatives with the term *references* for the second input of the *distance measure*. *References* can be (i) equal to *points*, (ii) a subset of *points* or (iii) a set of objects from the same feature-space like *points*. These two inputs and the calculated *distances* form the remaining defining elements of evaluation. Evaluation can now be generally defined as the creation of *point-distance-reference* relation-triples that explicitly express the similarity between points and references.

Selection phase: In this phase, the points that are eligible to be grouped together are selected according to the algorithms specification. For this, the output generated by evaluation is checked for points which are similar and should pass selection in order to be clustered together. This requires the definition of conditions that describe the status "similar" and to test whether they are fulfilled or not. For this, we propose *filters*, which represent the defining element of this phase. A set of these *filters* is used to test the input triples from evaluation and passes on only those that fulfill the conditions. Regard k-means as an example, that groups each point with the reference that is closest to it. Thus, one filter is needed to select only the minimum point-distance-reference triple for each point and pass them to the grouping phase.

Association phase: During this phase, the points that passed selection are associated with a cluster to create a clustering. This *clustering* represents the output and a defining element of this phase. A clustering is made up of relations, i.e., the affiliation between points and clusters. Association transforms *point-distance-reference* triples into *point-cluster* tuples. This is done using the defining element *association function*.

Some algorithms directly create clusterings, e.g., the *association function* of k-means takes a *point-distance-reference* triple, removes the distance and adopts the reference as cluster. In contrast, DBSCAN [6] first associates a core-object with its neighborhood – by creating *point-*



Semantic Domain: defining elements
Evaluation
Selection
Association
Optimization
Alg. X
Alg. Y
$\begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1f} \\ m_{21} & m_{22} & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & \cdots & \cdots & m_{nf} \end{pmatrix}$
$y - f(x)$
Formal Domain: Matrices, Functions

Figure 1: Mapping semantics to a formalism.

reference tuples – before the actual clusters are formed from overlapping neighborhoods. This necessitates an indirection between association function and clustering, which is given by the defining element: *adjacencies*. With it, association is defined as follows: incoming triples from selection are transformed into *adjacencies* from which the clustering is derived.

Optimization phase: This optional phase exists in several algorithms and implements optimizations aimed at improving the results generated by the mandatory phases evaluation, selection, and association. This generally leads to multiple iterations over the mandatory core while parameters are adjusted, references are updated etc.

2.2 Building blocks

Our defining elements occur in every clustering algorithm, where they realize the same abstract functionality in different ways. In order to describe this variety of semantics in a systematic way and create building blocks, we need a minimal set of formal elements to which all individual occurrences of our defining elements are mapped as illustrated in Figure 1. For this, we utilize a mathematical formalism that uses *matrices* for data containers like inputs and outputs, functions, and a small set of control flow structures. Matrices are denoted with a single capital letter, e.g., D for the distances. Different versions of a matrix are denoted in the superscript: D^I and D^{II} are versions of D after 1 resp. 2 function applications, while D^x and D^{x+1} designate the versions of D that are valid in the current and next iteration of the algorithm. Subscripts, are used to describe matrices in more detail, e.g., D_R denotes the distances between all references R. In the following we il-

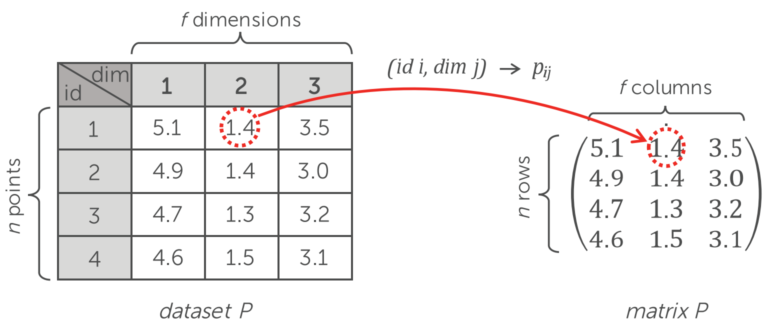


Figure 2: Mapping data to matrices.

lustrate the building blocks for the defining elements of clustering. By adding additional matrix or function blocks, arbitrary functionality can be added to a clustering algorithm.

Matrices: Generally, datasets for clustering are represented using multi-dimensional vectors inside a feature-space. Based on this procedure, *points* are defined as a set P of f-dimensional vectors $\vec{p} = \{p_0, \ldots, p_f\}$ where $n = |P|$. This set is converted into a matrix $P^{n \times f}$ by interpreting each vector $\vec{p}_i$ as a row $p_{i,*}$ of said matrix. *References* are defined accordingly as matrix $R^{k \times f}$ containing k reference vectors. This dataset to matrix conversion is illustrated in Figure 2. In addition to datasets, matrices can also describe relations between objects, e.g., *point-distance-reference* triples from evaluation, by using a row and column pair to address the matrix element holding the value of the actual relation. We define *distances* as a matrix $D^{n \times k}$, where $n = |P|$ and $k = |R|$. Each element d_{ij} of D relates a point/row $p_{i,*}$ of P to a reference $r_{j,*}$ of R. Thus, d_{ij} contains the distance between $p_{i,*}$ and $r_{j,*}$. This is adapted for tuples like point-cluster from *clustering*, by using a binary matrix to express an existing relation. A binary matrix element with value 1 at position (i, j) states a relation between the objects referenced by i and j. Accordingly, we define *adjacencies* as binary matrix $A^{n \times k}$ and *clustering* as binary matrix C with n rows and a number of columns matching the number of clusters.

Functions: The distance measure *dist*, takes a pair of rows $(p_{i,*}, r_{j,*})$ from P and R and assigns a scalar value to it, representing the distance between the corresponding objects. The abstract function *dist* can be defined as:

$$\begin{aligned} dist : (P, R) &\mapsto D \\ d_{ij} &= f(p_{i,*}, r_{j,*}) \end{aligned}$$

A filter checks whether a matrix or one of its elements fulfills certain conditions and passes them on or sorts them out accordingly. Thus a filter resembles an *if-then* statement. To describe the if-part, we use a *condition*, which is described as function with the co-domain (0, 1), representing the results false and true. A simple threshold condition, that is satisfied by all numbers smaller 10 could be defined as:

$$threshold : x^I = \begin{cases} 1, & \text{if} \quad x < 10 \\ 0, & \text{otherwise.} \end{cases}$$

To simplify notation, we only denote the condition leading to true as the function name. Thus, notation of the preceding definition is reduced to $\langle x < 10 \rangle$.

The 'then' part has to delete elements that fail the condition while leaving all others alone. Actual deletion of matrix elements cannot be modeled as a function, which necessitates a workaround. To replace removal, we define a neutral element to which all failing inputs are mapped. For our scenario, we choose 0 as neutral element, which allows us to define a minimum filter as:

$$\begin{aligned} minFilter : (D, \langle d_{ij} = min(D) \rangle) &\mapsto D^I \\ d^I_{ij} &= \langle d_{ij} = min(D) \rangle \cdot d_{ij} \end{aligned}$$

Assuming d_{23} is the minimum of a distance vector $d_{2,*}$, the filtered row becomes $d^I_{i,*} = (0, 0, d_{23}, 0)$. Obviously, subsequent functions must be aware of the neutral element. By executing filters, a modified version of the input is created, e.g., the modified distance matrix D^I, which is the output of the selection phase.

After filtering, the association function has to transform the remaining distances into adjacencies, i.e., *point-distance-reference* triples from selection must be converted into *point-reference* tuples. This turns the filtered distance matrix D^I into a binary matrix, where a value of 1 represents an existing adjacency. Non-existent adjacencies are already mapped to 0, which means the remaining non-zero values must be mapped to 1. For this, the sign function *sgn()* can be used. As distances cannot be negative it fits our requirements perfectly. With it, the association function *assoc* is defined as:

$$assoc : D^I \mapsto A$$
$$a_{ij} = sgn(d^I_{ij})$$

Control flow: Loops are a vital element of almost every clustering algorithm, but can hardly be modeled mathematically. Thus we define them outside the mathematical domain. Two loop types are essential: a *for-each* loop for element-wise traversal of datasets or clusterings and a *repeat-until* loop for conditioned iterations, e.g., during optimization.

For the for-each loop, the traversed matrix M and the element/granularity of traversal, i.e., row, column or component are specified in the head. In our case, element-wise traversal is done by splitting up the source matrix into element-matrices at the beginning of the loop, processing the individual elements according to the instructions in the loop body, and re-assembling them into complete matrices at the end of the loop. Loop output is denoted after the *end for* term. Furthermore, we use for-each loops to remove rows and columns of zeros from matrices. This is needed when algorithms delete references or filters empty whole clusters during optimization. By inserting a condition at the end of the loop we prevent zero element-matrices from being reassembled into the output matrix.

The *repeat-until* loop is used to represent conditioned loop for controlling algorithm iterations. Its stopping condition is specified after the closing *until* statement. The input matrices of a repeat-until loop are denoted in the head statement and are processed continuously until the loop stops. The output matrix of this loop type can be either a processed version of the input or an assembly of element-matrices generated during the loop.

2.3 Example algorithms

In the following, we demonstrate how clustering algorithms are described with our building blocks. We transcribe two prominent clustering algorithms: *k-means* [7] and DBSCAN [6]. Both belong to different classes, with k-means being part partitioning algorithm class and DBSCAN being a density-based method. Our descriptions will be kept brief due to page constraints. For more details, please refer to [9].

K-means creates clusters based on a given number of centroids by assigning points to their nearest centroid. Its description in Algorithm 1 begins with the evaluation phase where three matrix building blocks are placed: P contains the data points, R^x contains the k initial centroids, and D contains the distances between P and R^x. The superscript of R^x denotes that references will be updated during iterations of the algorithm. The fourth block is the euclidean distance, denoted as $dist.L_2$, that calculates D. The distance matrix D is passed on to the selection phase, where it is traversed row-wise using a for-each loop (3). Each row $d_{i,*}$ contains all distances between a point $p_{i,*}$ and R and is subjected to a filter. In accordance with the target function of k-means, this filter only keeps the minimum element d_{ij} per row and maps all other elements to 0. At the end of the loop, the processed rows are reassembled into the filtered matrix D^I (5), that is passed on to the association phase. There, the *assoc* function generates the binary adjacency matrix A (6). As k-means directly uses adjacencies as cluster assignments, A is simply adopted as result C (7).

Algorithm 1 k-means

```
 1: repeat with $R^x$
phase Evaluation
 2:     $dist.L_2(P, R^x) \rightarrow D$
phase Selection
 3:     for each $d_{i,*}$ of $D$ do
 4:         $filter(d_{i,*}, \langle d_{ij} = min(d_{i,*})\rangle)$
 5:     end for $\rightarrow D^I$
phase Association
 6:     $assoc(D^I) \rightarrow A$
 7:     $A \rightarrow C$
phase Optimization
 8:     $updt(C^T, P) \rightarrow R^{x+1}$
 9: until $R^x = R^{x+1}$ output $\rightarrow C$
```

With the core phases finished and a clustering result generated, k-means enters its optimization phase. There, centroids (references) are updated to the arithmetic average of all their assigned points and are passed on as references to the next iteration. This update procedure is realized with the matrix-multiplication of C and P. Binary matrix C contains all point-cluster assignments and has the dimensions $n \times k$ with k being the number of centroids. As P has the dimension $n \times f$, with n being the number of points and f being the number of features of the dataset, we have to transpose C to attain the required column-row-match for multiplication. Multiplying C^T and P creates an updated version of R^x with the correct dimension of $k \times f$. During matrix multiplication, each cluster represented by a binary row of C^T is used to select its assigned points

$p_{*,j}$ from the dataset P. The selection is summed up and divided by the number of cluster members, i.e., the sum of binary $c_{i,*}$. This realizes our desired update mechanism *updt*() (8) which creates the centroids R^{x+1} for the next iteration. The core of k-means is surrounded by a *repeat-until* loop that describes the iterative update of references R^x and stops when $R^x = R^{x+1}$(9). That means, optimization stops when the updated references no longer change.

DBSCAN defines clusters as dense regions separated by regions of lower density. The algorithm uses two parameters ε and *minPts* to define a density threshold. With ε a neighborhood is defined around each point p. If this neighborhood contains at least *minPts* additional points, p is considered as member of a dense area, i.e., a cluster and is named *core-object*. Sets of core-objects with overlapping ε-neighborhoods are merged in order to create clusters. This is done recursively, i.e., if p is a core-object each member of its ε-neighborhood is checked for the density condition.

The transcribed version of DBSCAN is shown in Algorithm 2. In contrast to k-means, DBSCAN calculates the distances between all points during evaluation, which means $R = P$. Selection traverses D row-wise and applies two filters. The first removes all distances that are bigger than the ε-neighborhood(3). The second filter utilizes *sgn()* to check if the number of remaining points in the neighborhood exceeds *minPts* (4).

Algorithm 2 DBSCAN

```
phase Evaluation
 1: $dist.L_2(P,P) \rightarrow D$
phase Selection
 2: for each $d_{i,*}$ of $D$ do
 3:     $filter(d_{i,*}, \langle d_{ij} < \varepsilon \rangle)$
 4:     $filter(d^I_{i,*}, \left\langle \sum_{j=0}^{n} sgn(d_{ij}) \geq minPts \right\rangle)$
 5: end for $\rightarrow D^I$
phase Association
 6: $assoc(D^I) \rightarrow A$
 7: $merge(A) \rightarrow C$
 8: $distinct(C) \rightarrow C_{distinct}$
```

Following selection, association starts with the known application of *assoc*() (6). This time, *assoc*() does not create final clusters, but associates each core-object with the members of its ε-neighborhood only. To determine the final clusters, overlapping ε-neighborhoods have to be merged. As our building blocks do not support recursion as proposed in [6], we use the *merge*() function (7). It uses a repeat-until loop to continuously multiply binary A^x with itself and make the resulting A^{x+1} binary again. In doing so, transitive cluster assignments are resolved and adjacencies are made explicit, i.e., each member of a cluster is associated with all other members. *Merge*() stops when A^{x+1} does not change anymore. The resulting C contains the final clusters but also duplicates due to *merge*(), e.g., a cluster with 4 members leads to 4 identical rows in the binary matrix C.

We apply the function *distinct* (8), that uses multiplications with binary matrices, filters and our loop constructs to select distinct clusters and eliminate their duplicates, creating the final clustering $C_{distinct}$. A more detailed definition of this function is given in [9].

Summary: Our building blocks allow the *systematic description* of arbitrary clustering algorithms. In order to translate existing methods into building blocks, they must be examined, decomposed, and re-thought. By following this process understanding is improved as our proposed base template and consistent building blocks offer structure and guidance. Furthermore, algorithms can be compared in more detail by examining identical and dissimilar building blocks in their descriptions. This also supports the development of novel algorithms, as building blocks can be easily re-combined. To illustrate this feature, we regard our example clustering algorithms k-means and DBSCAN again. We described k-means as a partitioning algorithm that assigns each point to its nearest reference. The corresponding pairs are selected using a minimum filter. While this keeps selection and association simple, it also makes the algorithm susceptible to noise. K-means assigns each point to a reference, and thus to a cluster, regardless of how far away it is from the references in general. In doing so, outliers can compromise cluster homogeneity. To tackle this issue, selection must be expanded with a filter that checks whether there are references in a certain neighborhood around a point. This functionality is already present in DBSCAN and can be added to the selection phase of k-means by simply placing the corresponding building block. Just like this, design and adaptation of algorithms becomes more agile and user-friendly.

3 Forecasting

Like with clustering, we first look for the essential conceptual components of forecasting to create a set of building blocks for this domain. We start by examining the basic idea of forecasting, which can be described as the process of making predictions of the future based on the analysis of historic and current data. For this, data is represented as a sequence of values ordered by time (time series). In more detail, forecasting analyses time series data in order to create a model that represents the time series as close as possible. An optimal fitting to the time series can be achieved by adapting one of the different available models with individual sets of parameters. Fitting is generally done iteratively by an optimizer that creates different parameterizations. For each set of parameters the fit between model and time series is evaluated with an error measure. When the best parameterization is found, the model is executed and extrapolates a forecasting value at a time stamp in the future.

From this definition, we derive the essential steps and components of a forecasting algorithms in Section 3.1, turn them into a formalism for building blocks in Section 3.2, and show algorithm examples in Section 3.3.

3.1 Base template

We derive the fundamental phases of forecasting as follows: First, the available data is analyzed and the parts for modeling are selected. Second, the selected parts are incorporated into a parameterized model that represents the time series. Third, the model is fitted to the data using an optimization technique that creates sets of parameters and an error measure that evaluates their quality. Finally, the fitted model is used to extrapolate a forecast value. Unlike the phases of clustering, our four forecasting phases are not independent and follow a less linear sequence of execution. Furthermore, they are more straight forward and less complex. While each phase of clustering can contain different kinds of building blocks, the forecasting phases are very homogeneous and are quite similar to building blocks themselves. In the following, we describe each phase and its basic tasks. We ignore the fourth as it is just plain application of the fitted model. Out of the three remaining phases, the fitting phase is optional.

Window generation: During this first phase, the existing data of a time series is partitioned and extracted for the forecast model. For this, data markers are defined that we call windows. Windows can have different lengths, can span one or more time series, can be overlapping or distinct, and can occur in different numbers. Furthermore, their placement can be regular or irregular. All this information is contained in a window definition that forms the defining element of this phase. Window generation contains all window definitions that are needed by the forecast model.

Modeling: In this phase, the model for the representation of the time series is described. Equations and formulas are the defining elements of modeling that incorporate data from the time series via the definitions from window generation. Furthermore, the parameters for fitting are added to the model. All equations are specified such that the whole model outputs one result, the forecast value.

Fitting: The fitting phase has two defining elements: the optimizer and the error measure. Both are specified as black box building blocks meaning that they are not modularly defined within the scope of forecasting but referenced as closed operators. In order to realize such modularity, additional specific sets of building blocks would be required. The feasibility of such sets remains doubtful due to the many degrees of freedom found in their definition.

3.2 Building blocks

Similar to clustering, we need formal elements to systematically describe the semantics of forecasting algorithms. We regard time series data as an ordered sequence of values like an array or vector. Each value is referenced via its time stamp resp. index. Parameters are denoted with greek letters. There are no specific control flow elements. We assume that the fitting phase is executed iteratively inside a repeat-until loop that stops when the optimal parameters are found. In the following, we describe the building blocks for the defining elements of forecasting.

Window definition: Each window is defined using the following schema:

$$\{\, \mathit{label}(\mathit{pnum}, \mathit{length}, \mathit{scope}) \cdots \} \# \, (\mathit{offset}, \mathit{nwindows})$$

The part before the hashtag describes the window shape, while the part after describes its number and placement. More complex windows can be defined by adding multiple shape descriptions in the curly braces. This can be necessary if different characteristics of the time series, e.g., a certain season and the near past, should be considered in the model. *Label* is used to uniquely identify each window and reference it in the forecast model. *Pnum* defines the number periods used by the window. When referenced by the model, only the first value of the period is used. For algorithms that work with a growing number of periods the marker $g(s)$ is used and indicates that the number of periods is constantly incremented by one. The argument s specifies the initial number of periods. *Length* defines how many time stamps are contained in one period. In combination with *pnum* this is used to realize intervals between considered values, e.g., for capturing seasonal behavior. For illustration, regard the following examples assuming a monthly time granularity: To consider the last three months *pnum* is 3 and *length* equals 1. To consider the data for the current month in the last two years, *pnum* becomes 2 while *length* is set to 12. *Scope* describes how many times series a model considers and can only have two values: 1 for a single time series or k for a set of k time series. For sliding windows, *offset* defines how many time stamps the window is moved, while *nwindows* specifies the number of windows that are considered in the model.

Model, optimizer, error measure: These building blocks are straightforward equations and functions. As a convention, the result of the forecast model, i.e., the forecast value is denoted as $\hat{x}_{t+1}$.

3.3 Example algorithms

In the following, we describe two example forecasting algorithms with our building blocks. We start with an *autoregressive integrated moving average (ARIMA)* model [3], which is one of the most common forecasting models. Second, we describe the *Holt-Winters algorithm* [3] as an example for exponential smoothing approaches.

ARIMA is exemplified, using the description of a non-seasonal AR3 model, which represents a time series value as the weighted sum of its 3 predecessors. For this, sliding windows of respective width are generated over the time series and optimization is used to find the three weights $\alpha_1, \alpha_2, \alpha_3$ that represent the time series values as best as possible. After this is done, the optimized weights are applied to the last three time series values to calculate the forecast value $\hat{x}_{t+1}$(2). This leads to the description in Algorithm 3.

Algorithm 3 ARIMA - AR3

```
phase Window Generation
  1: $\{x(3, 1, 1)\}$ # $(1, |t| - (3 \cdot 1))$
phase Modeling
  2: $\hat{x}_{t+1} = \alpha_1 x_t + \alpha_2 x_{t-1} + \alpha_3 x_{t-2}$
phase Fitting
  3: optimizer: NelderMead($\alpha_1, \alpha_2, \alpha_3$)
  4: error measure: SSE()
```

Our window definition (2) is illustrated in Figure 3 and describes the necessary three periods of length 1 as well as the offset for the sliding window. The term $|t| - (3 \cdot 1)$ describes the number of windows that are created, whereas the number of available time stamps $|t|$ is reduced by the

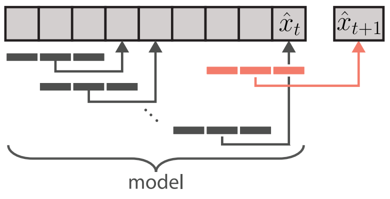


Figure 3: Windows for AR3.

last window which is used for calculating the forecast value. In the fitting phase, the downhill simplex method *NelderMead* [12] is used for the optimization of α_1, α_2, and α_3. Another common choice would be LBFGS [11]. For the error measure we use the *error-sum-of-squares SSE*, which is a common error measure that sums up the squared differences between predicted values and observed values. Other examples of such measures are SMAPE and RMSE [5].

Holt-Winters is also called triple exponential smoothing and incorporates the time series, a trend component, and a seasonality into the forecast. Seasonality is used to describe a certain behavior of the time series that repeatedly occurs every L periods. The basic working principle of exponential smoothing is recursion. Simply put, a smoothed value a_t^* is the weighted average of the current time stamp a_t and the previous smoothed value a_{t-1}^*. With Holt-Winters, this smoothing is applied to the observed time series, a trend component defined as difference between subsequent time stamps, and a recurring season. Each of these components has an individual smoothing factor, all of which are subject to parameter optimization. The forecasting value is calculated by applying the optimized smoothing parameters to the whole time series. This leads to the description shown in Algorithm 4.

Algorithm 4 Holt-Winters

```
phase Window Generation
  1: $\{x(g(L), 1, 1)\}$ # $(0, |t| - L)$
phase Modeling
  2: $a_t^* = \alpha \cdot \frac{x_t}{c_{t-L}^*} + (1 - \alpha) \cdot (a_{t-1}^* + b_{t-1}^*)$
  3: $b_t^* = \beta \cdot (a_t^* - a_{t-1}^*) + (1 - \beta) \cdot (b_{t-1}^*)$
  4: $c_t^* = \gamma \cdot \frac{x_t}{a_t^*} \cdot (1 - \gamma) \cdot (c_{t-L}^*)$
  5: $\hat{x}_{t+1} = (a_t^* + b_t^*) \cdot c_t^*$
phase Fitting
  6: optimizer: NelderMead($\alpha, \beta, \gamma$)
  7: error measure: SSE()
```

Window definition is quite different from ARIMA due to the recursive character of Holt-Winters that necessitates the use of $g(L)$ to define a growing number of periods. L defines the length of a season and must be used as initial number of periods in order to provide seasonal information from the beginning. No sliding window is used so the offset is set to 0, while the number of windows results from the incrementally growing periods. Modeling is quite complex and contains the smoothing equations for time series a_t^*, trend b_t^*, and season c_t^* as well as their respective parameters α, β, γ. The last equation shows how the components are combined to create the forecast value. The fitting phase is the same as used in ARIMA.

Summary: While we introduced only autoregressive methods as examples, i.e., methods that calculate a forecast value as a function of its previous values, our building blocks can also describe regressive approaches or methods based on machine learning like MARS [1] and SVR [14]. These methods generally take the whole time series into account as one and then apply a single optimization to it. For our descriptions, this means that window definition uses $pnum = |t|$ to encompass all data and optimization functionality is migrated into the modeling phase, making fitting somewhat obsolete.

4 Perspectives

With our modular building blocks concept, we approach Big Data analytics at a fundamental level. By offering a structured and systematic way to describe analytic algorithms we also change and improve how they are understood. In addition to the offered utility of our concept for describing, comparing, and organizing algorithms, it also offers significant benefits for their adaption and design. Due to the modularity of our approach, not only building blocks but also whole phases can be interchanged and recombined to form new analytic algorithms. With this, our approach realizes reusability in algorithm development as each additional building block or phase provides new combination options. These different modular algorithms, phases, and building blocks can be collected and organized in extensive repositories. By adding performance measures and case-based reasoning to these repositories, recommenders for algorithm selection could be developed.

We also see potential for cross-domain usage of our building blocks as similarities can be found between clustering an forecasting. Both domains show variations of the basic pattern: evaluate data, asses elements, combine certain elements, and optimize. This pattern has also emerged in our ongoing development of building blocks for compression algorithms. Cross-domain combination of building blocks offers further interesting options for algorithm design, e.g., putting window generation before evaluation in clustering could be used to create algorithms for time series clustering.

Besides its descriptive potential, our building blocks will also be used as the core of a system for developing and running modular algorithms. For this, the analysis-specific development environments like SystemML, we mentioned in the introduction, seem to be a promising starting point. By developing specific compilers, our building blocks can provide user-friendly frontends for designing specific types of analysis techniques in these systems. This allows us to take advantage of assets like efficient data structures and optimizers, without giving up on high-level algorithm specification. In addition, systems like Mahout-Samsara [2] that offer optimized processing of matrices, could be used to realize our building blocks for clustering and ensure the efficient execution of modular algorithms. Such a system setup would be similar to SQL query processing with sets of logical building blocks that are mapped to platform-specific physical implementations and a huge potential for application and optimization.

Acknowledgement: This work was, in part, funded by the German Federal Ministry of Education and Research (BMBF) in the context of the project "ScaDS – Competence Center for Scalable Data Services and Solutions Dresden/Leipzig" (01IS14014A).

References

1. Jerome H. Friedman. Multivariate Adaptive Regression Splines. *The Annals of Statistics*, 19(1):1–67, 1991.
2. Apache Mahout, http://mahout.apache.org.
3. J. Scott Armstrong. *Principles of forecasting: A handbook for researchers and practitioners*. Kluwer Academic Publishers, Norwell, 2001.
4. Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel. KNIME: The Konstanz Information Miner. In *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer, 2007.
5. Zhuo Chen and Y Yang. Assessing forecast accuracy measures. Technical report, 2004.
6. Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of KDD*, 1996.
7. E. W. Forgy. Cluster analysis of multivariate data: Efficiency versus interpretability of classification. *Biometrics*, 21, 1965.
8. Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda,

Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, pages 231–242, Washington, DC, USA, 2011.
9. Martin Hahmann, Dirk Habich, and Wolfgang Lehner. Modular data clustering – algorithm design beyond mapreduce. In *Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), Athens, Greece, March 28, 2014.*, pages 50–59, 2014.
10. A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3), 1999.
11. Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45:503–528, 1989.
12. J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.
13. R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.
14. N. Sapankevych and Ravi Sankar. Time series prediction using support vector machines: a survey. *Computational Intelligence Magazine, IEEE*, 4(2):24–38, 2009.

Bionotes

Dr.-Ing. Martin Hahmann
Technische Universität Dresden, Dep. of Computer Science, D-01062 Dresden, Germany
martin.hahmann@tu-dresden.de

Dr.-Ing. Martin Hahmann studied Computer Science at Technische Universität Dresden. He finished his diploma in 2007 and received his Phd in 2013. He is a member of the scientific staff of Prof. Dr. Wolfgang Lehner and the Competence Center for Scalable Data Services and Solutions. His research focuses on analytic processes, clustering and visualization.

Claudio Hartmann
Technische Universität Dresden, Dep. of Computer Science, Nöthnitzer Str. 46, D-01062 Dresden, Germany

Claudio Hartmann studied Computer Science at Technische Universität Dresden. He received his diploma in 2013. He is a member of the scientific staff of Prof. Dr. Wolfgang Lehner. His research focuses on time series analysis and forecasting.

Lars Kegel
Technische Universität Dresden, Dep. of Computer Science, Nöthnitzer Str. 46, D-01062 Dresden, Germany

Lars Kegel studied Computer Science at Technische Universität Dresden where he received his diploma in 2014. He is a member of the scientific staff of Prof. Dr. Wolfgang Lehner. His research focuses on time series modeling and model maintenance.

Dr.-Ing. Dirk Habich
Technische Universität Dresden, Dep. of Computer Science, Nöthnitzer Str. 46, D-01062 Dresden, Germany

Dr.-Ing. Dirk Habich studied Computer Science at the University of Halle-Wittenberg where he received his diploma in 2006. He received his Phd in 2008 from the Technische Universität Dresden, where he is a member of the scientific staff of Prof. Dr. Wolfgang Lehner. His research interest focuses on Database support for Data Mining, in-memory databases, and modern system architectures.

Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden, Dep. of Computer Science, Nöthnitzer Str. 46, D-01062 Dresden, Germany

Prof. Dr.-Ing. Wolfgang Lehner studied Computer Science at the University of Erlangen-Nürnberg where he also received his Phd and habilitation. Since 2002 Wolfgang Lehner leads the Database Systems Group at Technische Universität Dresden. He has been a visiting Researcher at multiple institutions including IBM Almaden, Microsoft Research Redmond and SAP Walldorf. His work focuses on real-time analysis in data-warehouse-systems, in-memory databases for analytic and transactional query processing and support for advanced analytics in databases.