INFORMS Journal on Computing

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Validation Sequence Optimization: A Theoretical Approach

Gediminas Adomavicius, Alexander Tuzhilin,

To cite this article:
Gediminas Adomavicius, Alexander Tuzhilin, (2007) Validation Sequence Optimization: A Theoretical Approach. INFORMS Journal on Computing 19(2):185-200. https://doi.org/10.1287/ijoc.1050.0153

INFORMS Journal on Computing
Vol. 19, No. 2, Spring 2007, pp. 185–200
ISSN 1091-9856 | EISSN 1526-5528 | 07 | 1902 | 0185

DOI 10.1287/ijoc.1050.0153

Validation Sequence Optimization: A Theoretical Approach

Gediminas Adomavicius
Department of Information and Decision Sciences, Carlson School of Management, University of Minnesota, 321 19th Avenue South, Minneapolis, Minnesota 55455, USA, gedas@umn.edu

Alexander Tuzhilin
Department of Information, Operations, and Management Sciences, Stern School of Business, New York University, 44 West Fourth Street, New York, New York 10012, USA, atuzhili@stern.nyu.edu

The need to validate large amounts of data with the help of the domain expert arises naturally in many data-intensive applications, including data mining, data stream, and database-related applications. This paper presents a general validation approach that generalizes different expert-driven validation methods developed for specialized validation problems. In particular, we model the validation process as a sequence of *validation operators*, explore various properties of such sequences, and present theoretical results that provide for better understanding of the validation process. We also address the problem of selecting the best validation sequence among the class of equivalent sequence permutations. We demonstrate that this optimization problem is NP-hard and present two heuristic algorithms for improving validation sequences.

Key words: validation; validation operators; validation sequences; sequence optimization; computational complexity; heuristic algorithms; dynamic programming; data mining
History: Accepted by Amit Basu, Area Editor for Knowledge and Data Management; received June 2003; revised April 2004, October 2004; accepted May 2005.

1. Introduction and Motivation

This paper addresses the problem of validating large amounts of data and information generated in various applications. It is especially common in data mining, where the need to validate results arises naturally in the post-analysis stage of the knowledge discovery process. For example, Adomavicius and Tuzhilin (2001) describe validating user profiles in personalization applications, Tuzhilin and Adomavicius (2002) describe validating biological relationships in bioinformatics, and Klemettinen et al. (1994), Liu and Hsu (1996), Liu et al. (1999), and Adomavicius (2002) describe the general issues of validating frequent itemsets and association rules, discovered by the Apriori data-mining algorithm (Agrawal et al. 1996). The personalization problem can generate many rules (such as "when a person travels on business to Los Angeles, she tends to stay in expensive hotels there") that can be measured in hundreds of millions for large-scale CRM applications (Adomavicius and Tuzhilin 2001). Validation determines which of these rules are truly interesting and which can be discarded as obvious or irrelevant for a particular application. Thus, knowledge discovery significantly depends on the quality of validation.

While data-mining applications are the primary motivation for this research, the validation problem can occur in other types of applications. For example, Babu et al. (2004) consider efficient validation of data streams, where a continuous stream of data is processed by a set of commutative filters. This can be critical in many "data-stream" applications where data are updated continuously and need to be processed in real time, such as network monitoring, sensor processing, telecommunications-fraud detection, and financial stock monitoring (Babu et al. 2004). Another category of applications is validation of the transactional data using database queries. For instance, any application where records in a relational table have to be classified into several groups by the domain expert would fit into this category; this includes student admission processes into colleges, where all candidates are either accepted, rejected, or put on a waiting list based on a variety of criteria.

As has been observed (e.g., Brachman and Anand 1996, Fayyad et al. 1996, Silberschatz and Tuzhilin 1996, Provost and Jensen 1998, Lee et al. 1998, Adomavicius and Tuzhilin 1999, Sahar 1999), knowledge discovery should be an iterative and interactive process with explicit participation of the domain expert. Many advocate direct involvement of the user (e.g., domain expert) in validation, and rule validation in the post-analysis stage of knowledge discovery has been addressed before (Klemettinen et al. 1994, Liu and Hsu 1996, Srikant et al. 1997, Imielinski and

Virmani 1999, Liu et al. 1999). Although particular validation methods described by Klemettinen et al. (1994), Srikant et al. (1997), Imielinski and Virmani (1999), Adomavicius and Tuzhilin (2001), Tuzhilin and Adomavicius (2002), and Adomavicius (2002) are different and were developed for different applications, they have common ideas that go beyond data mining and can be applied to other important IT problems, such as processing data streams (Babu et al. 2004) or validation of transactional data using database queries. In this paper, we present a general theory behind different expert-driven validation methods for particular validation problems described above. In particular, we model the validation process as a sequence of *validation operators*. Validation operators were introduced for data-mining rule validation by Adomavicius and Tuzhilin (1999, 2001) and Tuzhilin and Adomavicius (2002), where the methods for *generating* sequences of these operators were also studied.

We assume that an initial sequence of validation operators is already defined. This is usually done by the domain expert who interactively and iteratively generates the sequence using validation tools (Adomavicius and Tuzhilin 2001). We focus on the optimization problem of how to replace this initially specified sequence of validation operators with an equivalent but more *efficient* sequence. This is important in data mining for the following reasons. First, new data keep arriving over time and this affects previously generated patterns, e.g., these patterns may no longer hold in light of the newly available data. The new data may also facilitate discovery of completely new patterns. As a result, it is necessary to re-evaluate and re-validate the data-mining results. In large applications having frequently changing data, this re-evaluation and re-validation process can be computationally intensive, so it is crucial to validate sequences in the most efficient manner. Second, validation is an iterative and interactive process, where the domain expert usually waits for the results of the previous validation step. Therefore, generation of more efficient sequences would reduce the domain expert's waiting time. This is crucial in large-scale applications dealing with millions of generated patterns where excessive waiting time can easily make the validation process impractical.

Moreover, the need for efficient validation exists beyond data mining. For example, work on data streams (Babu et al. 2004) considers not only data-stream processing, but also investigates *optimizing* the sequences of pipelined filters. This is important for data streams because stream validation often must occur in real time, and it is crucial to validate these streams faster than the data arrive.

To keep the problem applicable to a broad set of applications, we study validation of such sequences in a setting that is more general and abstract than a specific application of data mining rules. We also study properties of these sequences, including sequence equivalence, sequence permutation, and sequence optimality properties. In addition, we present theoretical results providing for a better understanding of the validation process. Finally, we address the problem of selecting the *best* validation sequence among the class of equivalent sequences. We show that this sequence-optimization problem is NP-hard, and present two heuristic algorithms for improving validation-sequence performance.

We not only build the theory behind validation methods, but also generalize and abstract them. Our approach is applicable to any data-validation problem having the following properties: (a) it generates many data points that cannot be validated individually; (b) generated data points can belong to different categories and need to be classified using several labels; and (c) the problem is knowledge-intensive and requires direct involvement of the domain expert since data-generation methods cannot leverage this knowledge as well as the domain expert. For example, personalization applications can generate millions of rules comprising profiles of individual customers via data mining, and numerous such rules can be spurious, obvious, or irrelevant (Adomavicius and Tuzhilin 2001). Such rules would have to be labeled as "bad" by the domain expert, separated from the rules labeled as "good" or "undetermined," and removed from user profiles.

In the next section, we present a general approach to validation that is based on sequences of validation operators. In the rest of the paper, we develop the solutions to the sequence-optimization problem.

2. General Validation Problem

Assume that we have a finite set $\mathscr{E}$ containing all possible data points that may need to be validated by the domain expert. Then a *data set* D is simply some subset of all possible data, i.e., $D \subseteq \mathscr{E}$. The domain expert "validates" data set D by assigning *labels* from the label set $\mathscr{L} = \{L_1, L_2, \ldots, L_n\}$ to each input element $e \in D$. More formally, the goal of the validation process is to split the input set D into $n+1$ pairwise disjoint sets $V_1, V_2, \ldots, V_n$, and U, where each V_i represents the subset of D that was labeled with L_i, and U denotes the subset of D that remains unlabeled after the validation process (i.e., some input elements may remain unvalidated).

Since D may contain many data elements, it may not be feasible for the domain expert to validate all the elements in D manually, i.e., by inspecting and labeling each of them. To make expert-driven validation feasible, we use *validation operators*, i.e., methods that allow the expert to validate multiple elements of D at a time. This is achieved by letting the domain

expert specify logical *predicates* that label a *class* of data elements with a particular label.

Formally, let $\mathcal{P}$ be the set of all possible predicates for validating input data from $\mathcal{E}$, i.e., $\mathcal{P}$ contains all predicates p of the form

$$p\colon \mathcal{E} \longrightarrow \{\text{True}, \text{False}\}. \tag{1}$$

Then, the validation operator is defined as follows.

Definition 1 (Validation Operator). A *validation operator* is defined as a tuple (l, p), where $l \in \mathcal{L}$ and $p \in \mathcal{P}$.

In other words, for an unvalidated data set D and an expert-specified validation operator $o = (l, p)$, all data points $e \in D$ for which $p(e) = \texttt{True}$ are labeled with l and are considered validated. Data points $e \in D$ for which $p(e) = \texttt{False}$ remain unvalidated. Therefore, validation is an iterative process, where in each iteration the domain expert can specify a new validation operator that validates still another portion of yet-unvalidated parts of D. Thus, the validation process is a *sequence* of validation operators.

Definition 2 (Validation Sequence). A sequence of validation operators is called a *validation sequence*. The validation sequence is $\langle o_1, o_2, \ldots, o_k \rangle$, where $o_i = (l_i, p_i)$, $l_i \in \mathcal{L}$ and $p_i \in \mathcal{P}$.

The schematic description of the validation process is in Figure 1. The following examples describe validation's role.

Example 1 (Validation in Databases). D contains records in a relational table to be classified into categories by a domain expert. A table might contain job candidates, where we must decide whether each candidate should be extended an offer, called for an additional interview, or not considered. The labels of validation operators would correspond to categories of job candidates (`job offer`, `additional interview`, `reject`). The predicates would be database queries issued by the domain expert on the candidate table, such as "Find the candidates who have excellent communication skills and who passed their previous job interviews with ratings of at least 5."

Example 2 (Generating Labels for Supervised Learning). To generate labels for supervised-learning algorithms in machine learning, class labels need to be assigned to the training and testing data by the domain expert "by hand" in many applications. This is expensive for large data sets containing many records. D is the set of training and testing records (as in Example 1), and $\mathcal{L} = \{L_1, L_2, \ldots, L_n\}$ is the set of class labels. The predicates would be the label-assignment operators specified by the domain expert using the interactive and iterative assignment process similar to Adomavicius and Tuzhilin (2001).

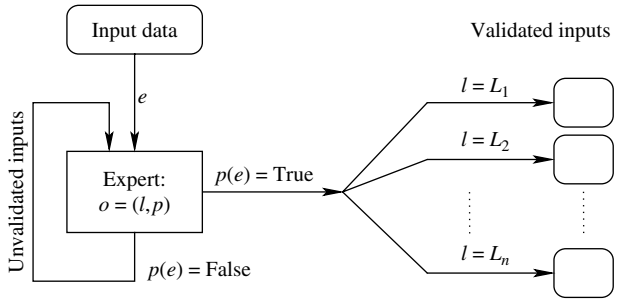

Figure 1 Expert-Driven Validation Process

Example 3 (Validation in Data Streams). D is a packet stream going through a router or firewall of an internet service provider (ISP). An example of a data-stream processing task is "monitor the amount of common traffic flowing through four ISP's routers over the last ten minutes" (Babu et al. 2004). The network analyst could use this task to monitor network health and find opportunities for load balancing. Data streams from the above routers are then processed (validated) by a sequence of pipelined filters (Babu et al. 2004), which would correspond to validation operators in our framework.

Thus, validation problems are general and can be applied in a variety of applications. However, since expert-driven validation is important in data mining, we use mostly data mining examples throughout the paper.

Example 4 (Validation in Data Mining). D is set of association rules (Agrawal et al. 1996) about customer purchasing behavior, like *"people who buy milk and yogurt, also buy bread,"* or *milk & yogurt → bread*. The set of labels $\mathcal{L}$ is $\{\texttt{good}, \texttt{bad}\}$, i.e., the domain expert wants to label the association rules of interest as "good" and of no interest (e.g., irrelevant or obvious rules) as "bad." Since the number of discovered association rules can be very large (Adomavicius and Tuzhilin 2001, Tuzhilin and Adomavicius 2002), it may be hard for the domain expert to validate them *all*, so some rules are likely to remain unvalidated.

One way to specify validation operators for association rules would be *template*-specification languages (Klemettinen et al. 1994, Srikant et al. 1997, Imielinski and Virmani 1999, Adomavicius and Tuzhilin 2001, Tuzhilin and Liu 2002). A rule template takes a set of rules as an input and returns only those rules that satisfy the template, like those that make inferences about bread purchasing, i.e., having attribute *bread* in their heads (consequents). Using the template specification language from (Adomavicius and Tuzhilin 2001), this filter can be specified as the following validation predicate: *"HEAD = bread."* Moreover, the expert may be interested in all such rules and accept them as "interesting" by assigning the label "good" to them. Formally, the validation operator is $o = (\texttt{good}, \text{"}HEAD = bread\text{"})$. For example, rule *milk & yogurt → bread* would match the above template and, thus, be labeled as

"good," whereas *bread* → *butter* would not match and would remain unvalidated.

As another example, to filter the discovered rules that have *milk* (possibly among other attributes) in their bodies (antecedents), one can specify *"BODY* ⊇ *milk*." Again, *milk & yogurt* → *bread* would match this template, and *bread* → *butter* would not.

Generating validation sequences in data mining is described in Adomavicius and Tuzhilin (1999, 2001), who argue that the domain expert is needed to drive the validation process. However, a validation sequence selected by a domain expert may not be the most efficient one. We focus on the performance-optimization problem: how to select the most efficient sequence that is equivalent to the validation sequence initially specified by a domain expert. This is relevant where we need to re-validate the data periodically. Since it would be a waste of human resources to let the domain expert conduct the re-validation process every time, we could have the domain expert validate the data (or a sample) only once and generate a validation sequence. Subsequent periodic validations then can be performed *automatically* using the sequence initially generated by the expert. Therefore, we should optimize this initial sequence, the more efficient version of which could then be automatically applied in multiple subsequent validations.

3. Sequence-Optimization Problem

In this section, we formalize the sequence-optimization problem. We first define the concepts of *equivalence* and *performance measure* for validation sequences.

As a result of the validation by sequence s, the input set D is divided into $n+1$ pairwise disjoint sets $V_1, V_2, \ldots, V_n$, and U, where each V_i is the subset of D that was labeled with L_i, and U is the subset of D that remains unlabeled after the validation process. Let $VI_s(D) := (V_1, V_2, \ldots, V_n)$ (validated input elements) and $UI_s(D) := U$ (unvalidated input elements).

3.1. Equivalence of Validation Sequences

Intuitively, validation sequences are equivalent if they always produce the same validation results for *any* data set to which they are applied; this is formalized below. $\mathscr{E}$ is a finite set of all possible data points that may need to be validated by the domain expert, and $D \subseteq \mathscr{E}$.

Definition 3 (Equivalent Validation Sequences). Validation sequences s and s' are *equivalent* (write $s \sim s'$) if and only if $VI_s(D) = VI_{s'}(D)$ for every input $D \subseteq \mathscr{E}$.

Lemma 1. $s \sim s' \Leftrightarrow VI_s(\mathscr{E}) = VI_{s'}(\mathscr{E})$.

(Proofs of the more important theoretical results are provided in the Online Supplement to this paper on the journal's website.)

The equivalence of validation sequences is a *binary relation* $R_\sim$ on the set of all possible validation sequences.

Lemma 2. *Relation* $R_\sim$ *is an equivalence relation, i.e.,* $R_\sim$ *is reflexive, symmetric, and transitive.*

Example 5 (Equivalent Validation Sequences). Consider validation sequences for association rules that are expressed using the template specification language mentioned in Example 4. Let $\mathscr{L} = \{\texttt{good}, \texttt{bad}\}$, $s = \langle(\texttt{good}, \text{"}HEAD = Bread\text{"}), (\texttt{good}, \text{"}BODY \supseteq Milk\text{"})\rangle$, and $s' = \langle(\texttt{good}, \text{"}BODY \supseteq Milk\text{"}), (\texttt{good}, \text{"}HEAD = Bread\text{"})\rangle$. Both sequences have two validation operators and clearly $s \neq s'$ since s' is a reverse of s. However, $s \sim s'$, since both s and s' will validate (i.e., will label with good) the same inputs, given any input data set.

The validation-sequence optimization problem is

$$s^* = \arg\min_{s' \sim s} cost(s'), \tag{2}$$

where $cost(s)$ is some performance measure for validation sequences. This is a variant of scheduling, i.e., we need to rearrange (reschedule) operators in s to minimize the cost function. We explore this connection to scheduling in Section 5.4.

To address (2), we should consider how $cost(s)$ should be defined, how we find sequences equivalent to s, i.e., what the *search space* of our optimization problem is, and how we find the optimal sequence efficiently. That is, even if we can determine all possible equivalent sequences and calculate the cost function for each of them, there may be too many of them and exhaustive search may be impractical.

Note that (2) assumes that the initial sequence is given and finds some better sequence equivalent to it. How the initial sequence is generated is beyond the scope of this paper and has been addressed elsewhere; e.g., in data mining (Adomavicius and Tuzhilin 2001) the initial sequence is generated by the domain expert during post analysis of data mining results.

3.2. Defining the Cost of a Validation Sequence

The time it takes to perform validation would be one natural way to define the cost of a validation sequence. However, the cost of validating a sequence may depend significantly on the input data set. For example, validation sequence s might be able to validate all of D_1 with its first operator (i.e., if all elements of D_1 match the predicate of the first operator), so subsequent operators would not even have to be invoked. On the other hand, the same s might not be able to validate any elements of data set D_2, so *all* validation operators would have to be invoked on *all* elements of D_2.

Therefore, we define the validation-sequence optimization problem for a specific data set D. That is,

which of the sequences that are equivalent to a given expert-specified validation sequence s would have the best performance on data set D? This approach enables the validation process to be adaptive, i.e., if the characteristics of data change over time, the validation sequence can always be dynamically adjusted (optimized) based on its performance on the *latest* portion of D.

Ideally, we would use run time as the performance measure. While we generally would be able to compute the run time of the expert-specified sequence s (i.e., during the initial validation process), it obviously would be difficult to estimate run time of *other* sequences theoretically. We overcome this difficulty with certain simplifying assumptions. In particular, we assume that it takes a *fixed* time for a validation operator $o = (l, p)$ to validate an input element $e \in D$, i.e., to check whether e satisfies predicate p. In other words, we assume that all predicates are "similar" in their capabilities and that all individual predicate/input checks take about the same time. This assumption is often used in similar situations, e.g., Babu et al. (2004) use this assumption to develop algorithms for optimizing the sequence of pipelined filters for data-stream processing. So our cost function is the total number of predicate/input satisfaction checks performed by s to validate input D. Specifically, assume that sequence s consists of k validation operators $o_1, o_2, \ldots, o_k$ and each of the operators o_i validated n_i number of elements from D. That is, operator o_1 checked all $|D|$ inputs and validated n_1 of them. Subsequently, o_2 checked the remaining $|D| - n_1$ inputs and validated n_2 of them, and so on. Then, the cost of validating D using sequence s can be defined as

$$\begin{aligned} cost(s, D) &= |D| + (|D| - n_1) + (|D| - n_1 - n_2) \\ &\quad + \cdots + \left(|D| - \sum_{j=1}^{k-1} n_j\right) \\ &= k|D| - \sum_{i=1}^{k-1} (k-i) n_i. \end{aligned} \tag{3}$$

Furthermore, let $cost_0(s, D)$ be the worst possible cost scenario, when not a single input from D is validated by s, i.e., all $n_i = 0$, so $cost_0(s, D) = k|D|$. Define *benefit* function as

$$\begin{aligned} benefit(s, D) &= cost_0(s, D) - cost(s, D) \\ &= \sum_{i=1}^{k-1} (k-i) n_i. \end{aligned} \tag{4}$$

Our optimization problem is thus

$$s^* = \arg\min_{s' \sim s} cost(s', D) = \arg\max_{s' \sim s} benefit(s', D). \tag{5}$$

How will we search for equivalent sequences? An important question is: how can we change sequence s so that the newly obtained sequence s' remains equivalent to s? Note that the general sequence equivalence specified in Definition 3 constitutes a *semantic* concept so it is difficult to find a general solution to (5) over the space of *all* equivalent sequences. This is common in computer science, e.g., consider finding equivalent schedules of transaction executions in databases for the purpose of concurrency control (Bernstein et al. 1987). Equivalence of two schedules of transactions is a semantic concept that is hard to verify formally. Therefore, for analytical tractability, this semantic notion of equivalence was replaced with a simpler *syntactic* notion of serializability of a schedule of transactions (Bernstein et al. 1987).

We follow a similar approach and replace the semantic notion of equivalence of validation sequences with a more restrictive but a simpler syntactic concept of equivalence between the *permutations* of some validation sequence. To illustrate this, consider Example 5, in which we had two equivalent sequences s and s', where s' was a permutation of s. So given validation sequence s, we search for the solution to our optimization problem among the *permutations* s' of sequence s such that $s \sim s'$.

The permutation-based approach allows us to keep our validation framework very general, since we need only to examine the positions of the predicates in the validation sequence. We would not be able to do this with the semantics-based approach to sequence equivalence, because it would require taking into account the complexities of the domain knowledge pertaining to data and predicates used in each specific application.

4. Permutation-Based Sequence Optimization

The permutation-based approach has been used for optimizing sequences of pipelined filters in data-stream-processing applications (Babu et al. 2004). However, Babu et al. (2004) analyze the optimization problem where all filters are *commutative*, i.e., any two filters can be switched without any consequences to the correctness of the processing result (only the processing time may be affected). Such an approach is not applicable to our case since not all validation operators are commutative for our validation problem in general, as will be demonstrated below. Moreover, unlike Babu et al. (2004), we deal with a general problem that goes beyond validation of data streams. Therefore, this problem needs a different permutation-based approach.

4.1. Permutations of Validation Sequences

Definition 4 (Sequence Permutation). Let s and s' be validation sequences, i.e., $s = \langle o_1, o_2, \ldots, o_k \rangle$

and $s' = \langle o'_1, o'_2, \ldots, o'_k \rangle$. s' is called a *permutation* of sequence s if and only if $\{o_1, o_2, \ldots, o_k\} = \{o'_1, o'_2, \ldots, o'_k\}$.

In other words, s' is a permutation of s if it contains exactly the same validation operators, but not necessarily in the same order. Let $s = \langle o_1, o_2, \ldots, o_k \rangle$ be a validation sequence and let u be some validation operator. We will say that s *contains* u (and denote $u \in s$) if there exists $x \in \{1, \ldots, k\}$ such that $u = o_x$. In such case we will say that s contains u at a position x and denote $pos_s(u) = x$.

Let u and v be validation operators contained in the validation sequence s. Then we will say that u *precedes* v in sequence s if $pos_s(u) < pos_s(v)$ and denote as $u \prec_s v$.

Definition 5 (Permutation Distance). Let s be a validation sequence $\langle o_1, o_2, \ldots, o_k \rangle$. Let $s' = \langle o'_1, o'_2, \ldots, o'_k \rangle$ be some permutation of s. The *distance* between s and s' is the number of operator inversions between s and s', i.e., the number of all distinct pairs of validation operators u, v, such that $u \prec_s v$ and $v \prec_{s'} u$:

$$dist(s, s') := |\{(u, v): u \in s,\ v \in s,\ u \prec_s v,\ v \prec_{s'} u\}|. \quad (6)$$

Example 6 (Permutation Distance). Let $s = \langle o_1, o_2, o_3, o_4 \rangle$ and $s' = \langle o_3, o_2, o_4, o_1 \rangle$. Then the distance between s and s' is 4, because there are 4 operator inversions in s' with respect to s: (o_3, o_1), (o_2, o_1), (o_4, o_1), (o_3, o_2).

Definition 6 (Simple Permutation). Let s be a validation sequence, and let s' be a permutation of s such that $dist(s, s') = 1$. Then s' is called a *simple* permutation of s.

Thus, $s' = \langle o'_1, o'_2, \ldots, o'_k \rangle$ is a simple permutation of $s = \langle o_1, o_2, \ldots, o_k \rangle$ if and only if there exists $i \in \{1, \ldots, k-1\}$ such that $o_i = o'_{i+1}$ and $o_{i+1} = o'_i$, and for all j ($j \neq i$ and $j \neq i+1$) $o_j = o'_j$ holds true. More generally, $dist(s, s')$ is the minimal number of simple permutations needed to obtain s' from s.

Definition 7 (Safe Permutation). Let s be a validation sequence. Then s' is a *safe* permutation of s if and only if s' is a permutation of s and $s \sim s'$.

Lemma 3. *If s' is a permutation of s, then $UI_s(D) = UI_{s'}(D)$ for every input data set D.*

4.2. Deriving Equivalence Criteria for Sequence Permutations

Theorem 4. *Let $s = \langle (l_1, p_1), \ldots, (l_k, p_k) \rangle$ and $s' = \langle (l'_1, p'_1), \ldots, (l'_k, p'_k) \rangle$ be validation sequences, where s' is a permutation of s. Then $s \nsim s'$ if and only if they contain a pair of validation operators $u = (l_u, p_u)$ and $v = (l_v, p_v)$ that satisfy all of the following conditions*:

1. *u precedes v in s, but v precedes u in s', i.e., $u \prec_s v$ and $v \prec_{s'} u$;*
2. *u and v have different labels, i.e., $l_u \neq l_v$;*
3. *There exists an element $e \in \mathscr{E}$ such that the Boolean expression*

$$p_u(e) \wedge p_v(e) \wedge \bigwedge_{i=1}^{x-1} \neg p_i(e) \wedge \bigwedge_{j=1}^{y-1} \neg p'_j(e) \quad (7)$$

is true, where $x = pos_s(u)$ and $y = pos_{s'}(v)$.

Intuitively, $s \nsim s'$ if and only if there exists a data element $e \in \mathscr{E}$ such that e is validated differently by s and s', and Theorem 4 provides the precise conditions for this.

Example 7 (Nonequivalent Validation Sequences). Let $s = \langle (\texttt{good}, \text{“}HEAD = Bread\text{”}), (\texttt{bad}, \text{“}BODY \supseteq Milk\text{”}) \rangle$ and let $s' = \langle (\texttt{bad}, \text{“}BODY \supseteq Milk\text{”}), (\texttt{good}, \text{“}HEAD = Bread\text{”}) \rangle$. Both sequences have two validation operators and clearly $s \neq s'$ since s' is a reverse of s. It is easy to see that s and s' satisfy the first two conditions of Theorem 4. As for the third condition, consider rule $Milk \rightarrow Bread$. Since this rule matches both of the predicates in s (and its permutation s'), sequence s would label this rule as `good` and s' would label it as `bad`. Hence, $s \nsim s'$.

The following corollary states necessary and sufficient conditions for $s \sim s'$, where s' is a permutation of s. It can be straightforwardly derived from Theorem 4 by taking the logical negation of the necessary and sufficient conditions for $s \nsim s'$.

Corollary 5. *Let $s = \langle (l_1, p_1), \ldots, (l_k, p_k) \rangle$ and $s' = \langle (l'_1, p'_1), \ldots, (l'_k, \ldots, p'_k) \rangle$ be validation sequences, where s' is a permutation of s. Then $s \sim s'$ if and only if every possible pair of validation operators $u = (l_u, p_u)$ and $v = (l_v, p_v)$ (i.e., $u \in s$, $v \in s$, $u \neq v$) satisfies at least one of*:

1. *Either u precedes v in both s and s', or v precedes u in both s and s';*
2. *u and v have the same label, i.e., $l_u = l_v$;*
3. *For all possible input elements $e \in \mathscr{E}$,*

$$\neg(p_u(e) \wedge p_v(e)) \vee \bigvee_{i=1}^{x-1} p_i(e) \vee \bigvee_{j=1}^{y-1} p'_j(e) \quad (8)$$

is true, where $x = pos_s(u)$ and $y = pos_{s'}(v)$.

Finally, the following corollary states necessary and sufficient conditions for $s \sim s'$, where s' is a *simple* permutation of s. It follows immediately from Corollary 5 by restricting s' to be a simple permutation of s.

Corollary 6. *Let $s = \langle o_1, \ldots, o_k \rangle$ be a validation sequence where $o_i = (l_i, p_i)$, and let $s' = \langle o'_1, \ldots, o'_k \rangle$ be a simple permutation of s. I.e., $(\exists! x \in \{1, \ldots, k-1\})$ $((o_x = o'_{x+1}) \wedge (o_{x+1} = o'_x))$, and also $o_i = o'_i$ for all $i \in \{1, \ldots, k\}$ such that $i \neq x$ and $i \neq x+1$. Then $s \sim s'$ if and only if*

at least one of the following conditions is satisfied:

1. o_x *and* o_{x+1} *have the same label, i.e.,* $l_x = l_{x+1}$;
2. *For all possible input elements* $e \in \mathscr{E}$,

$$\neg(p_x(e) \wedge p_{x+1}(e)) \vee \bigvee_{i=1}^{x-1} p_i(e) \tag{9}$$

is true.

The practical implication of the first condition is clear: whenever we switch any two adjacent operators in a validation sequence (i.e., perform a simple permutation), the new sequence is equivalent to the initial one if the labels of the inverted operators are the same. It is somewhat more difficult to derive the precise practical implications of the second condition, and we will discuss this later.

4.3. "Connectedness" of Equivalent Sequence Permutations

In this section, we prove several facts that provide understanding about the structure of the "space" of equivalent validation sequence permutations.

LEMMA 7. *Let* $s = \langle o_1, o_2, \ldots, o_k \rangle$ *be a validation sequence. Let* s' *be some permutation of* s. *Then, for every pair of validation operators* o_i *and* o_j *such that* $o_i \prec_s o_j$ *(i.e.,* $i < j$*), but* $o_j \prec_{s'} o_i$, *there exists* x *such that* $i \le x \le j-1$ *and* $o_{x+1} \prec_{s'} o_x$.

The following theorem is the main result about the space of all equivalent permutations.

THEOREM 8. *Let* $s = \langle o_1, o_2, \ldots, o_k \rangle$ *be a validation sequence. Let* s' *be some safe permutation of* s *such that* $dist(s, s') = d$, *where* $d \ge 1$. *Then there exists a sequence* s'' *that is a safe simple permutation of* s, *such that* $s' \sim s''$ *and* $dist(s', s'') = d - 1$.

This theorem states that, if $s \sim s'$, where s' is a permutation of s and $dist(s, s') = d$, then there always exists an "intermediate" equivalent validation sequence s'' that is a simple permutation of s, i.e., s'' is equivalent to both s and s', $dist(s, s'') = 1$, and $dist(s', s'') = d - 1$.

By applying the above theorem repeatedly, it is easy to demonstrate the "connectedness" of *all* equivalent validation sequences via their equivalent simple permutations, as stated by the following theorem.

THEOREM 9. *Let* $s = \langle o_1, o_2, \ldots, o_k \rangle$ *be a validation sequence. Let* s' *be some permutation of* s *such that* $dist(s, s') = d$, *where* $d \ge 1$. *Then,* $s \sim s'$ *if and only if there exist* $d + 1$ *validation sequences* $s_0, s_1, \ldots, s_d$, *such that* $s_0 = s$, $s_d = s'$, *and* s_i *is a safe simple permutation of* s_{i-1} *for every* $i = 1, \ldots, d$.

The above results give us a better understanding about equivalent permutations of a given validation sequence s. We can visualize the space of *all* permutations of s by constructing a *permutation graph*, where each vertex corresponds to a different permutation of s. Furthermore, in this graph, two vertices will have an edge connecting them if one of them is a simple permutation of the other. An example of such a graph for a sequence consisting of four validation operators is in Figure 2.

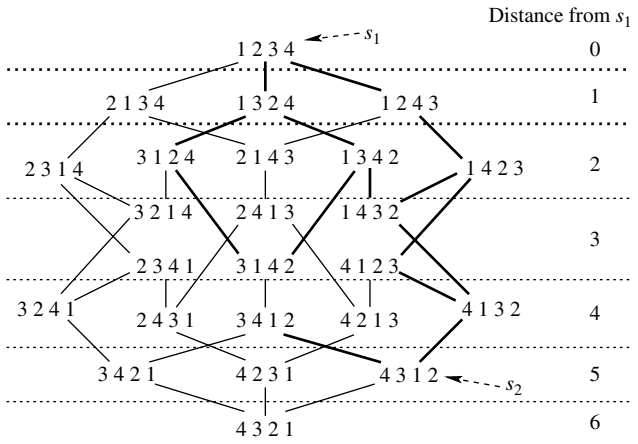

Figure 2 Permutation Graph of a Validation Sequence

Obviously, any two permutations are connected by a path (in fact, multiple paths) in this permutation graph. Theorem 9 states that if permutations s_1 and s_2 are equivalent, there exists a minimal path from s_1 to s_2 that goes only through "intermediate" permutations that are equivalent to s_1 and s_2. Furthermore, not only does such a path exist, but it is also *minimal*, i.e., its length (number of edges comprising the path) is always equal to the permutation distance between s_1 and s_2. To illustrate this, assume that $s_1 := \langle o_1, o_2, o_3, o_4 \rangle$, and let s_2 be a permutation of s_1 such that $s_2 := \langle o_4, o_3, o_1, o_2 \rangle$. Then the distance between s_1 and s_2 is 5, since there are 5 operator inversions in s_2 with respect to s_1, i.e., (o_4, o_3), (o_4, o_2), (o_4, o_1), (o_3, o_2), (o_3, o_1). Figure 2 highlights all the paths of length 5 between s_1 and s_2.

Theorem 9 identifies a "search space" of equivalent validation sequence permutations, and we can use this in solving our sequence-optimization problem. Assume that we have an expert-specified validation sequence s, and that we have a cost function defined for each permutation of s (we will address this issue later). Then, we could search for the optimal permutation by traversing the permutation graph (such as depicted in Figure 2) by doing only *safe* simple permutations, starting from vertex s. Theorem 9 guarantees that we will encounter the optimal permutation s^* along the way.

We could use various graph-traversal techniques, such as depth-first or breadth-first search. However, if the validation sequence s has k validation operators, then the number of possible permutations is $k!$ so exhaustive search techniques are not scalable. On the other hand, the largest possible distance between two permutations is $(k-1) + (k-2) + \cdots + 2 + 1 = k(k-1)/2$ (i.e., the largest possible number of inver-

sions between two permutations – where every pair of operators is inverted). Therefore, by Theorem 9, there exists a path between the expert-specified validation sequence s and the optimal sequence s^* that is not longer than $k(k-1)/2$. Consequently, based on the specific cost function, we would like to find a greedy algorithm that allows us to choose the right path without traversing the whole graph.

One obstacle with this approach is that we must be able to determine whether a given simple permutation is safe. Corollary 6 gives us two conditions – at least one of them has to be met by a simple permutation in order for it to be safe. Let $s = \langle o_1, \ldots, o_k \rangle$ be a validation sequence where $o_i = (l_i, p_i)$, and let $s' = \langle o'_1, \ldots, o'_k \rangle$ be a simple permutation of s. That is, $(\exists! x \in \{1, \ldots, k-1\})\ ((o_x = o'_{x+1}) \wedge (o_{x+1} = o'_x))$, and also $o_x = o'_x$ for all $i \in \{1, \ldots, k\}$ such that $i \neq x$ and $i \neq x+1$. As mentioned earlier, the first condition of Corollary 6, $l_x = l_{x+1}$, is easily verifiable. The second condition, i.e., whether

$$\neg(p_x(e) \wedge p_{x+1}(e)) \vee \bigvee_{i=1}^{x-1} p_i(e)$$

holds for all possible inputs $e \in \mathscr{E}$, depends on the semantics of predicates and in general may not be easily solvable analytically.

We will show later that even when we restrict the validation-sequence optimization problem, it remains computationally complex. More specifically, to study complexity of the validation-sequence optimization problem and further understand the structure of the space of the equivalent validation sequence permutations, we introduce in Section 5 the concepts of *strong* and *very strong* equivalence of sequence permutations and study their properties. We later show that the optimization problem stated above, and even several of its special cases, are NP-hard.

5. Computational Complexity of the Validation-Sequence Optimization Problem

In this section we consider two special classes of equivalent validation sequence permutations based on the concept of *orthogonality* of predicates. We call them *strongly* and *very strongly* equivalent permutations and explore their theoretical properties. Moreover, we use the class of very strongly equivalent permutations to restrict our sequence-optimization problem and show that this restricted optimization problem is NP-hard, so the general optimization problem is NP-hard as well. Furthermore, we use the class of strongly equivalent validation sequence permutations in heuristic algorithms for validation-sequence improvement.

5.1. Predicate Orthogonality

Definition 8 (Predicate Orthogonality). Predicates p and q are *orthogonal* if their conjunction is not satisfiable, i.e., if $(\forall e \in \mathscr{E})\ (\neg(p(e) \wedge q(e)))$. If p and q are orthogonal, write $p \perp q$ (and $p \not\perp q$ otherwise).

Thus, two predicates are orthogonal if they can never match the same input element.

Example 8 (Orthogonal Predicates). According to the rule-template specification language proposed by Adomavicius and Tuzhilin (2001), templates *"HEAD = bread"* and *"HEAD = milk"* clearly would never match the same association rule, so predicates *"HEAD = bread"* and *"HEAD = milk"* are orthogonal.

Example 9 (Nonorthogonal Predicates). Templates *"HEAD = bread"* and *"BODY ⊇ milk"* would both match association rule *milk → bread*. Hence, *"HEAD = bread"* and *"BODY ⊇ milk"* are not orthogonal.

Lemma 10. $(\forall p \in \mathscr{P})\ (p \perp \neg p)$.

Predicate orthogonality can be viewed as a *binary relation* $R_\perp$ on the set of all predicates. $R_\perp$ has the properties specified in the following lemma. (These are the properties that one may intuitively expect an orthogonality relation to have, e.g., orthogonality relation on the set of straight lines on the plane has the same properties.) The orthogonality conditions for predicate conjunctions and disjunctions are also stated below.

Lemma 11. *Binary relation $R_\perp$ is symmetric, but neither reflexive nor transitive.*

Lemma 12. *Let $p_1, \ldots, p_m$ and q be predicates. Let $p := p_1 \wedge \cdots \wedge p_m$. If there exists $i \in \{1, \ldots, m\}$ such that $p_i \perp q$, then $p \perp q$.*

Lemma 13. *Let $p_1, \ldots, p_m$ and q be predicates. Let $p := p_1 \vee \cdots \vee p_m$. Then $p \perp q$ if and only if $(\forall i \in \{1, \ldots, m\})$ $(p_i \perp q)$.*

5.2. Strongly Equivalent Permutations

Definition 9 (Strongly Equivalent Permutation). Let $s = \langle (l_1, p_1), \ldots, (l_k, p_k) \rangle$ and $s' = \langle (l'_1, p'_1), \ldots, (l'_k, p'_k) \rangle$ be validation sequences, where s' is a permutation of s. s' is said to be a *strongly equivalent* permutation of s (write $s \approx s'$) if and only if every possible pair of validation operators $u = (l_u, p_u)$ and $v = (l_v, p_v)$ (i.e., $u \in s$, $v \in s$, $u \neq v$) satisfies at least one of the following:

1. Either u precedes v in both s and s', or v precedes u in both s and s';
2. u and v have the same label, i.e., $l_u = l_v$;
3. p_u and p_v are orthogonal, i.e., $p_u \perp p_v$.

This definition mirrors the necessary and sufficient conditions for sequence equivalence (Corollary 5), except that the third condition is strengthened here (hence the term "strong equivalence").

Lemma 14. $s \approx s' \Longrightarrow s \sim s'$

Example 10 (Strongly Equivalent Permutations). Consider validation sequences from Example 5, i.e., $s = \langle (\texttt{good}, \text{"}HEAD = Bread\text{"}), (\texttt{good}, \text{"}BODY \supseteq Milk\text{"}) \rangle$ and its permutation $s' = \langle (\texttt{good}, \text{"}BODY \supseteq Milk\text{"}), (\texttt{good}, \text{"}HEAD = Bread\text{"}) \rangle$. According to Definition 9,

we have that $s \approx s'$, since the permuted validation operators have the same label, i.e., "good."

Strong equivalence of validation-sequence permutations can be viewed as a *binary relation* $R_{\approx}$ on the set of all possible permutations of a given validation sequence.

LEMMA 15. *$R_{\approx}$ is an equivalence relation: it is reflexive, symmetric, and transitive.*

Also, for strongly equivalent sequence permutations there exist necessary and sufficient conditions as well as "connectedness" results that are very similar to the corresponding results for equivalent permutations (i.e., Corollary 6 and Theorem 9).

THEOREM 16. *Let $s = \langle o_1, \ldots, o_k \rangle$ be a validation sequence where $o_i = (l_i, p_i)$, and let $s' = \langle o'_1, \ldots, o'_k \rangle$ be a simple permutation of s. I.e., $(\exists! x \in \{1, \ldots, k-1\})$ $((o_x = o'_{x+1}) \wedge (o_{x+1} = o'_x))$, and also $o_i = o'_i$ for all $i \in \{1, \ldots, k\}$ such that $i \neq x$ and $i \neq x+1$.*

Then $s \approx s'$ if and only if at least one of the following is satisfied:

1. *o_x and o_{x+1} have the same label, i.e., $l_x = l_{x+1}$;*
2. *$p_x \perp p_{x+1}$.*

THEOREM 17. *Let $s = \langle o_1, o_2, \ldots, o_k \rangle$ be a validation sequence. Let s' be a permutation of s such that $dist(s, s') = d \geq 1$. Then $s \approx s'$ if and only if there exist $d+1$ validation sequences $s_0, s_1, \ldots, s_d$, such that $s_0 = s$, $s_d = s'$, and s_i is a strongly equivalent simple permutation of s_{i-1} for every $i = 1, \ldots, d$.*

Now that we have defined the notion of strong equivalence for validation-sequence permutations, we can formulate the following special case of our validation-sequence optimization problem (i.e., a restriction of the search space to very strongly equivalent permutations):

$$s^* = \arg\min_{s' \approx s} cost(s', D) = \arg\max_{s' \approx s} benefit(s', D)$$
$$= \arg\max_{s' \approx s} \sum_{i=1}^{k-1} (k-i) n'_i. \quad (10)$$

We know the number of data points n_i from D validated by each validation operator o_i in the initial sequence s. However, the corresponding values n'_i for validation operators o'_i in s' are different from the n_i for strongly equivalent sequences s' in (10) in general. Since we do not know the n'_i values, this makes it difficult to solve the problem (10).

In the next section, we will consider a more restricted version of strong equivalence—very strong equivalence—that guarantees that the set of values $\{n'_i\}$ is always the same as $\{n_i\}$ for very strongly equivalent sequence permutations. This will be useful for determining the computational complexity of the optimization problem for that class of sequences and, consequently, for optimization problems (10) and (5).

5.3. Very Strongly Equivalent Permutations

We restrict the class of strongly equivalent permutations even further and introduce the class of *very strongly equivalent permutations.* We use this new class to prove NP-hardness of the restricted optimization problem and, consequently, NP-hardness of the general optimization problem (5).

DEFINITION 10 (VERY STRONGLY EQUIVALENT PERMUTATION). Let $s = \langle (l_1, p_1), \ldots, (l_k, p_k) \rangle$ and $s' = \langle (l'_1, p'_1), \ldots, (l'_k, p'_k) \rangle$ be validation sequences, where s' is a permutation of s. s' is said to be a *very strongly equivalent* permutation of s (write $s \cong s'$) if and only if every possible pair of validation operators $u = (l_u, p_u)$ and $v = (l_v, p_v)$ (i.e., $u \in s$, $v \in s$, $u \neq v$) satisfies at least one of the following:

1. Either u precedes v in both s and s', or v precedes u in both s and s';
2. $p_u \perp p_v$.

This essentially mirrors the definition of strongly equivalent permutations (Definition 9), except the second condition ($l_u = l_v$) is omitted here.

EXAMPLE 11 (VERY STRONGLY EQUIVALENT PERMUTATIONS). Consider validation sequence $s = \langle$(good, "*HEAD = bread*"), (good, "*HEAD = milk*")$\rangle$ and its permutation $s' = \langle$(good, "*HEAD = milk*"), (good, "*HEAD = bread*")$\rangle$. Then $s \cong s'$ because the predicates of permuted validation operators are orthogonal according to Example 8.

EXAMPLE 12 (NOT VERY STRONGLY EQUIVALENT PERMUTATIONS). Consider validation sequence $s = \langle$(good, "*HEAD = bread*"), (good, "*BODY $\supseteq$ milk*")$\rangle$ and its permutation $s' = \langle$(good, "*BODY $\supseteq$ Milk*"), (good, "*HEAD = Bread*")$\rangle$. Then $s \ncong s'$ because the predicates of permuted operators are not orthogonal, according to Example 9.

LEMMA 18. $s \cong s' \Rightarrow s \approx s'$

Very strong equivalence is a true equivalence relation, and the necessary and sufficient conditions as well as the "connectedness" results for very strongly equivalent permutations can also be obtained.

Let s' be a very strongly equivalent permutation of s and consider an arbitrary operator $o_i \in s$, i.e., $pos_s(o_i) = i$. Also, assume that in the permuted sequence s', o_i would be at some position j, i.e., $o_i = o'_j$ or $pos_{s'}(o_i) = j$, and that o_i validated n_i data points from data set D. Then o_i will also validate n_i points in s' while being at position j.

LEMMA 19. *Let $s = \langle o_1, \ldots, o_k \rangle$ be a validation sequence and let $n_1, \ldots, n_k$ be the numbers of data points validated by each of the validation operators in s, given some data set D. Let $s' = \langle o'_1, \ldots, o'_k \rangle$ be a very strongly equivalent permutation of s (i.e., $s \cong s'$) and let $n'_1, \ldots, n'_k$ be the numbers of data points validated by each of the validation operators in s', given the same data set D. Then for every $i \in \{1, \ldots, k\}$: $n_i = n'_j$ where $j = pos_{s'}(o_i)$.*

Further restrict (10) to

$$s^* = \arg\max_{s' \cong s} benefit(s', D). \quad (11)$$

In the next section, we show that this problem is NP-hard.

5.4. NP-Hardness of the Optimization Problem for Very Strongly Equivalent Permutations

Given validation sequence $s = \langle o_1, \ldots, o_k \rangle$ and some permutation s', Definition 10 specifies alternative conditions to be satisfied by every pair of validation operators $u \in s$ and $v \in s$, in order for $s \cong s'$. We will show that these conditions are equivalent to specifying a certain partial order on the set of validation operators in s.

5.4.1. Precedence Graph of Validation Operators. Let $G_s = (V, E)$ be a directed acyclic graph with k vertices, where each vertex $i \in V$ is associated with a different validation operator o_i ($i \in \{1, \ldots, k\}$). Furthermore, the set E of edges is defined as follows. For every pair of validation operators $o_i = (l_i, p_i)$ and $o_j = (l_j, p_j)$ such that $o_i \prec_s o_j$ (i.e., $i < j$), add an edge from vertex i to vertex j to set E if $p_i \not\perp p_j$. We will call this graph a *precedence* graph of sequence s.

If precedence graph G_s has an edge from i to j, then any permutation s' that is very strongly equivalent to s must have $o_i \prec_{s'} o_j$. If that were not the case, i.e., if there existed a very strongly equivalent permutation s' such that $o_j \prec_{s'} o_i$, then we would derive a contradiction, since validation operators o_i and o_j would not satisfy either of the two conditions from Definition 10 and it would imply that $s \ncong s'$.

G_s represents a *partial order* over the set of validation operators $o_1, \ldots, o_k$. As shown above, those permutations of s that satisfy this partial order are very strongly equivalent to s, and the ones that do not satisfy this order are not very strongly equivalent to s. Also, since we have not placed any restrictions on what kind of predicates can be used in validation operators, the resulting precedence graph in general can represent any partial order. Therefore, we transform the restricted sequence optimization problem

$$s^* = \arg\max_{s' \cong s} \sum_{i=1}^{k-1} (k-i) n'_i$$

into the following validation operator *scheduling* problem:

$$s^* = \arg\max_{s' \text{ satisfies } G_s} \sum_{i=1}^{k-1} (k-i) n'_i, \quad (12)$$

where $n'_i = n_x$, if $pos_{s'}(o_x) = i$ (according to Lemma 19). In other words, the problem is to find a "scheduling" of operators $o_1, \ldots, o_k$ such that it obeys the precedence graph G_s and the corresponding permutation $\{n'_i\}$ of $\{n_i\}$ maximizes the *benefit* function.

Assume that we can efficiently compute whether two predicates are orthogonal, i.e., given validation sequence s, we can efficiently construct precedence graph G_s.

THEOREM 20. *Optimization problem* (12) *is NP-hard*.

Proof is based on the fact that solving the scheduling problem (12) is equivalent to solving the "task sequencing on a single processor to minimize weighted completion time," a known NP-hard problem (Garey and Johnson 1979); see the Online Supplement to this paper on the journal's website. NP-hardness of the validation-sequence optimization problems (5), (10), and (11) stated throughout the paper follows immediately from Theorem 20, equivalence of problems (11) and (12), and Lemmas 14 and 18.

5.4.2. Validation as Scheduling. Besides demonstrating that the validation-sequence optimization process is NP-hard, the above analysis also shows that our optimization process can be viewed as a certain type of scheduling. To illustrate this point further, consider the following analogy to be used subsequently for drawing connections to the scheduling problem.

Consider a factory that produces items of a certain kind. After items are manufactured, they must be sorted by quality, e.g., `excellent`, `good`, `medium`, or `bad` (in general, the number of degrees of quality is arbitrary), and are placed on a moving conveyor for final inspection. A group of inspection robots is standing along the side of the conveyor, and each robot inspects each passing item for a particular property. For example, robot R_1 only knows that all the "defective" items should be labeled as `bad`, robot R_2 only knows that all "small" and "green" items should be labeled as `good`, robot R_3 only knows that all "big" and "red" items should be labeled as `medium`, and so on. If the item matches the inspection criteria for a robot, it is removed from the conveyor by that robot and placed into an appropriate container, otherwise it stays on the conveyor and moves to the next robot.

Inspection of a single item by a single robot costs c, so the problem is to arrange the robots alongside the conveyor to minimize the total inspection cost. One way to solve this problem is to put robots that "capture" more items at the beginning of the conveyor.

EXAMPLE 13 (VALIDATION AS SCHEDULING). Suppose three robots are initially arranged R_1, R_2, R_3. After the first 1,000 inspections, R_1 labeled 20 (out of 1,000 items), R_2 labeled 20 (out of the remaining 980 items), and R_3 labeled 400 (out of the remaining 960 items). The total cost in this case is 2,940 inspections. However, if the sequence had been R_1, R_3, R_2, then R_1 again would have labeled 20 (out of 1,000 items), R_3 would have labeled 400 (out of remaining 980), and R_2 would have labeled 20 (out of remaining 580).

The total cost in this case would be 2,560 inspections, a significant improvement.

With respect to validation optimization, the items on the conveyor represent the data points to be validated, the sequence of robots represents the sequence of validation operators, and the total inspection cost represents our validation cost function. Moreover, various rearrangements of robots along the conveyor belt represent various permutations of validation operators. Not all robot rearrangements are possible. In Example 13 we could switch robots R_2 and R_3, because they represent *orthogonal* validation operators, i.e., no item can be both "small/green" and "big/red" at the same time. However, we could not have switched R_1 and R_2, because an item can be "small/green" and "defective" at the same time. Such an item would be labeled differently than in the original sequence, producing an incorrect outcome.

Our validation problem can be formulated as a scheduling problem: given a set of validation operators $\{o_1, o_2, \ldots, o_k\}$ schedule them in a sequence s' so that (a) s' is equivalent to the original sequence s specified exogenously (e.g., by the domain expert), and (b) its cost on some representative data set D, $cost(s', D)$, is minimal among all the equivalent sequences. This problem constitutes a nontraditional scheduling problem for the following reasons. First, it depends on the concept of equivalence between two sequences that we define differently from scheduling theory. Second, the cost structure (3) and (4) is such that the cost of each validation operator o_i contributing to the total validation cost not only depends highly on the *position* of o_i in the validation sequence, but also on its relative position with respect to other validation operators. This is also a nonstandard assumption in scheduling. For these reasons, we could not apply standard methods from scheduling to solve our problem. Instead, we used our theoretical results in the algorithms presented below.

6. Heuristics for Validation Sequence Improvement

In Section 5 we showed that, given validation sequence s, the problem of finding the optimal sequence among all the sequences that are very strongly equivalent to s is NP-hard. This problem is already NP-hard without even taking into account computation of the precedence graph G_s. For the task-scheduling problem, described in Section 5.4, the precedence graph is given (i.e., it is part of the input). However, in our problem, we have to calculate the precedence graph ourselves. In other words, we have to be able to calculate which pairs of operators must preserve their precedence in the permuted sequence, based on orthogonality of their predicates.

Therefore, the precedence-graph calculation depends on the class of predicates used in validation operators. If the predicates are complex, it may be very difficult (or impossible) to show whether two given predicates are orthogonal. Since the problem is NP-hard even when the graph is already given, we present two general (i.e., independent of the class of predicates used) heuristic-based approaches to improving the validation sequence when a precedence graph G_s is given as an input.

6.1. Precedence Graph for Strongly Equivalent Permutations

To prove NP-hardness of the optimization problem, we showed earlier how to construct the precedence graph based on *very strong equivalence* constraints. For our heuristic approaches, we construct the precedence graph based on less restrictive equivalence—*strong equivalence*—constraints. In this scenario, our heuristics will potentially have a much larger search space to work with and so may generate permutations with better performance improvements.

Given validation sequence $s = \langle o_1, \ldots, o_k \rangle$ and some its permutation s', Definition 9 specifies several conditions that must be satisfied by every pair of validation operators $u \in s$ and $v \in s$ so that $s \approx s'$. We show that these conditions are equivalent to specifying a certain partial order on the set of validation operators in s.

More specifically, let $G_s = (V, E)$ be a directed acyclic graph with k vertices, where each vertex $i \in V$ is associated with a different validation operator o_i ($i \in \{1, \ldots, k\}$). Furthermore, the set E of edges is defined as follows. For every pair of validation operators $o_i = (l_i, p_i)$ and $o_j = (l_j, p_j)$ such that $o_i \prec_s o_j$ (i.e., $i < j$), add an edge from vertex i to vertex j to set E if both $l_i \neq l_j$ and $p_i \not\perp p_j$.

If G_s has an edge from i to j, then any permutation s' that is strongly equivalent to s must have $o_i \prec_{s'} o_j$. If that were not the case, i.e., if there existed a strongly equivalent permutation s' such that $o_j \prec_{s'} o_i$, then we would derive a contradiction, since validation operators o_i and o_j would not satisfy all three conditions from Definition 9 and it would imply that $s \not\approx s'$.

6.2. Sequence Improvement Using a Simple Permutation

Let the *improvement* of sequence s' over sequence s be

$$\Delta_{s \to s'} := cost(s, D) - cost(s', D). \tag{13}$$

Based on *cost* and *benefit* functions (3) and (4),

$$\Delta_{s \to s'} = benefit(s', D) - benefit(s, D), \tag{14}$$

Let $s = \langle o_1, \ldots, o_k \rangle$ be a validation sequence and G be a precedence graph based on s. Also, let $s' = \langle o'_1, \ldots, o'_k \rangle$ be a *simple* permutation of s, i.e., ($\exists! x \in$

$\{1, \ldots, k-1\})$ $((o_x = o'_{x+1}) \wedge (o_{x+1} = o'_x))$, and also $o_i = o'_i$ for all $i \in \{1, \ldots, k\}$ such that $i \neq x$ and $i \neq x+1$. Assume that there is no precedence constraint between operators o_x and o_{x+1}, i.e., there is no edge from o_x to o_{x+1} in G. Therefore, $s \approx s'$. Consequently, $s \sim s'$ and therefore s' will produce the same validation results as s. What is the cost of s' (or the value of $\Delta_{s \to s'}$)? To answer this, we estimate n'_i, i.e., the number of data points from data set D that each validation operator o'_i (from permuted sequence) would validate.

THEOREM 21. *Let s' be a simple permutation of s, obtained by swapping two adjacent validation operators o_x and o_{x+1}. Then $\Delta_{s \to s'} \geq n_{x+1} - n_x$.*

Therefore, whenever we perform a simple permutation that is permissible (i.e., allowed by the precedence graph), we are guaranteed to decrease the cost (or increase the benefit) of the validation sequence by at least $n_{x+1} - n_x$.

6.3. Greedy Heuristic Based on Simple Permutations

Our heuristic algorithm for improving validation sequences relies in part on the ability to calculate *predicate orthogonality*. Assume $s = \langle o_1, \ldots, o_k \rangle$ is an expert-specified validation sequence that was used to validate data set D, and let n_i be the number of elements validated by operator o_i, where $i \in \{1, \ldots, k\}$. We will construct an improved validation sequence $s' = \langle o'_1, \ldots, o'_k \rangle$ where $o'_i = (l'_i, p'_i)$ as a permutation of s. In the beginning, let $o'_i := o_i$ and $n'_i = n_i$ for each i. Then construction of s' can be described as follows:

(1) done := **False**; $\Delta_{Total} := 0$;
(2) **while** ¬done
(3) $S := \{1 \leq i \leq k-1 \mid l_i = l_{i+1} \vee p_i \perp p_{i+1}\}$
(4) **if** $S = \varnothing$ **then** done := **True**
(5) **else** $x := \arg\max_{i \in S}(n'_{i+1} - n'_i)$
(6) $\Delta_x := n'_{x+1} - n'_x$
(7) **if** $\Delta_x \leq 0$ **then** done := **True**
(8) **else** $\Delta_{Total} := \Delta_{Total} + \Delta_x$
(9) swap o'_x and o'_{x+1} in the validation sequence s'
(10) swap values of n'_x and n'_{x+1}
(11) **end if**
(12) **end if**
(13) **end while**

The intuition behind this algorithm is that in every iteration of the algorithm (lines 3–12), given the current sequence s', in a greedy fashion we choose to make a simple permutation of s' that remains equivalent to s'. That is, we consider only those pairs of adjacent operators o'_i and o'_{i+1} that either have identical labels or their predicates are orthogonal (line 3). Based on Definition 9, by swapping these two operators we would produce an equivalent permutation. Furthermore, in each iteration, among all available equivalent simple permutations we choose the one that maximizes the *improvement* of the sequence performance (line 5). The simple permutations (lines 9–10) continue in this greedy fashion until there is no incremental improvement to the sequence performance (determined at line 7).

While the greedy heuristic is not guaranteed to give us an optimal solution to the sequence-optimization problem, it can be shown that

$$\Delta_{s \to s'} \geq \Delta_{Total}, \tag{15}$$

where Δ_{Total} is calculated by the above heuristic algorithm. More precisely,

$$\Delta_{Total} = \sum_{i:\,(o_i \prec_s o_j) \wedge (o_j \prec_{s'} o_i)} (n_j - n_i). \tag{16}$$

To show that $\Delta_{s \to s'} \geq \Delta_{Total}$, one can easily extend the argument from Section 6.2, which showed that $\Delta_{s \to s'} \geq n_{x+1} - n_x$ for simple strongly equivalent permutations. Furthermore, based on the results from Section 6.2, we have that the above lower bound is tight, i.e., $\Delta_{s \to s'} = \Delta_{Total}$, when *all* simple permutations performed by the above heuristic involved pairs of operators with orthogonal predicates (i.e., all permutations are *very strongly equivalent*).

EXAMPLE 14 (VALIDATION SEQUENCE IMPROVEMENT). Consider validation sequence $s = \langle o_1, o_2, o_3 \rangle$. Assume that this sequence was used to validate data set D consisting of 1,000 data elements, and that o_1 validated 150 (n_1), o_2 validated 100 (n_2), and o_3 validated 750 (n_3) of them. Thus, $cost(s, D) = 1{,}000 + 850 + 750 = 2{,}600$. Also assume that our heuristic produced the equivalent permutation $s' = \langle o_3, o_1, o_2 \rangle$ by first swapping operators o_2 and o_3, and then o_1 and o_3. According to the greedy heuristic, $\Delta_{Total} = (n_3 - n_2) + (n_3 - n_1) = (750 - 100) + (750 - 150) = 1{,}250$. Therefore, based on the lower-bound result, we are guaranteed to have $cost(s', D) \leq 1{,}350$, i.e., the cost was cut *nearly in half*.

The heuristic traverses the permutation graph (e.g., Figure 2) by doing one equivalent simple permutation at a time, starting from vertex s. Since at every iteration we perform an equivalent simple permutation, the sequence s' produced by the above heuristic remains equivalent to the original sequence s (because of the transitivity of the equivalence relation). In addition, note that on every iteration we perform a simple permutation only if $\Delta_x > 0$. Because of this, we are guaranteed not to swap the same two validation operators more than once. Therefore, the maximal number of iterations performed by the above heuristic is equal to the number of possible validation-operator inversions, i.e., $k(k-1)/2$ (assuming the validation sequence has k operators). However, $k(k-1)/2$ iterations are possible only when all simple permutations are permissible,

which is a worst-case scenario that is far from realistic, e.g., in many data-mining validation scenarios, the number of permissible permutations typically is $O(k)$ because of the predicate nonorthogonality between numerous pairs of operators that have different labels. Moreover, since computational complexity of a single iteration is $O(k)$, i.e., dealing with $k-1$ pairs of adjacent operators, the worst-case computational complexity of the heuristic is $O(k^3)$ with the typical real-life complexity being much lower, i.e., $O(k^2)$, as mentioned earlier. This is a significant improvement over the exhaustive search-based techniques that consider all validation-sequence permutations and has worst-case computational complexity of $O(k!)$.

6.4. Performance of the Greedy Heuristic

Since the sequence optimization problem is NP-hard, the greedy heuristic is not guaranteed always to produce the optimal validation sequence because it has only polynomial computational complexity. Therefore, it is important to understand how well the proposed heuristic typically performs.

Unfortunately, theoretical analysis of the heuristic performance is a difficult problem, because performance of the heuristic depends significantly on its inputs, i.e., on the specific data set to be validated and on the initial validation sequence specified by the domain expert. In other words, given validation sequence s, the heuristic may be able to find an optimal permutation on one input data set, but would produce a highly suboptimal permutation if a different data set is given. Conversely, given data set D, the heuristic would be able to find the optimal permutation for one expert-specified sequence on D, but may produce only a suboptimal permutation for a different sequence.

Example 15 (Optimality of Heuristic Results: Dependence on Inputs). In Example 13, consider the set of only two robots: robot R_1 labels all "red" items as `good` and robot R_2 labels all "small" items as `good`. Since both robots use the same label, permutations $s_{12} = \langle R_1, R_2 \rangle$ and $s_{21} = \langle R_2, R_1 \rangle$ are equivalent (even strongly equivalent). In general, the optimal sequence depends on the input data set: If we manufacture more "red" than "small" items, then s_{12} is optimal, otherwise s_{21} is optimal. Furthermore, assume that s_{12} is the initial expert-specified sequence of robots and consider the input data set D of 100 items: 20 items are "red" and "small," 70 are "green/small," and 10 are "green/big." Obviously, R_1 will validate 20 of the 100 items ($n_1 = 20$) and R_2 will validate 70 of the remaining 80 items ($n_2 = 70$). Based on this information ($n_1 < n_2$) the heuristic would swap R_1 and R_2 and would produce the optimal result in this case, since $cost(s_{12}, D) = 100 + 80 = 180$ and $cost(s_{21}, D) = 100 + 10 = 110$. Alternatively, assume data set D consists of 100 items: 40 items are "red" and "small," 30 are "green/small," and 30 are "green/big." Obviously, R_1 will validate 40 of the 100 items ($n_1 = 40$) and R_2 will validate 30 of the remaining 60 items ($n_2 = 30$). Based on this information ($n_1 > n_2$) the heuristic would not swap R_1 and R_2. However, the optimal result would not be produced in this case, since $cost(s_{12}, D) = 100 + 60 = 160$ and $cost(s_{21}, D) = 100 + 30 = 130$.

So it may not be possible to find any closed-form solutions describing the heuristic performance without understanding the specifics of the domain knowledge—the underlying data and the validation predicates, e.g., in Example 15 the understanding of the "overlap" between "small" and "red" items was more crucial than knowing the performance n_1 and n_2 of individual validation operators.

For the same reasons, i.e., because of the dependence of heuristic performance on the domain-specific information, it is difficult to produce not only theoretical, but also simulation-based results about the performance of the heuristic, unless we choose to make oversimplifying or hugely restrictive assumptions, e.g., restrict the input data and validation predicates to be of some very limited types. Since our focus is on the general validation problem, we plan to pursue the domain-specific validation issues in our future research.

However, as discussed in Section 6.3, the proposed heuristic provides some theoretical guarantees. As (15) and (16) indicate, the performance improvement $\Delta_{s \to s'}$ from the initial expert-specified sequence is guaranteed to be at least

$$\sum_{i:\,(o_i \prec_s o_j) \wedge (o_j \prec_{s'} o_i)} (n_j - n_i). \tag{17}$$

In other words, the heuristic is always able to present feedback regarding the guaranteed cost savings it produces, based only on its knowledge of initial sequence performance (i.e., n_i values).

While the greedy heuristic is not guaranteed to produce optimal results in *all* cases, it *can* generate optimal validation sequences in some circumstances. The most obvious example is when all simple permutations are very strongly equivalent, e.g., when all the predicates are pairwise orthogonal, as follows from the results in Section 6.2. In such cases, the heuristic would be able simply to sort the operators o_i in the descending order based on the number n_i of data points that each of the operators has validated.

Since the heuristic does not guarantee optimality, it is important to understand some of its worst-case scenarios, even without taking the domain specifics into account. Figure 3 illustrates one such scenario. s has six validation operators, and each can label the appropriate input with an `A` or `B` label. Assume that all predicates are pairwise orthogonal except for p_5 and p_6, i.e., $p_5 \not\perp p_6$. The numbers n_i of data points (out

i	p_i	l_i	n_i
1	p_1	A	2
...	...	...	...
4	p_4	A	2
5	p_5	A	1
6	p_6	B	1,000

(a) Initial sequence (s)
$cost(s, D) = 6{,}025$

i	p'_i	l'_i	n'_i
1	p_5	A	1
2	p_6	B	1,000
3	p_1	A	2
...	...	...	...
6	p_4	A	2

(b) Optimal sequence (s')
$cost(s', D) = 2{,}037$

Figure 3 "Problematic" Example

of 1,009 total data points) validated by each operator are also provided in Figure 3(a).

Based on our definition of sequence cost, the cost of s is 6,025. The greedy heuristic would not perform any changes to the sequence because $n_{i+1} - n_i \leq 0$ for every permissible simple permutation in s. Note that it is not permissible to swap operators 5 and 6, since neither $l_5 = l_6$ nor $p_5 \perp p_6$. However, if we do not follow the greedy approach and move operator 5 all the way up (because p_5 is orthogonal to $p_1, \ldots, p_4$), we can move operator 6 up to the second position in the sequence (again, because p_6 is orthogonal to $p_1, \ldots, p_4$), obtaining sequence s' (presented in Figure 3(b)) that is still equivalent to s, but significantly less costly. Specifically, the cost of s' is 2,037. Therefore, the greedy heuristic does not provide any performance improvements in this "worst-case" scenario, whereas an optimal solution improved performance by 3,988 operations (cost reduction of 66.2%).

Worst-case validation sequences, as shown in Figure 3(a), do not occur frequently. As was argued by Adomavicius and Tuzhilin (2001), if possible, domain experts usually try to specify more general validation operators validating large amounts of data first (e.g., more general rule filters in data-mining applications), and the more specific operators later. It is not typical to specify many very specific validation operators (i.e., $p_1, \ldots, p_4$) before applying a very strong and general one (i.e., p_6). The heuristic would be able to find the optimal sequence (s') in the above example if we had $n_5 = 3$ (instead of $n_5 = 1$). Clearly, the heuristic algorithm can be improved so that it can detect some of the "problematic" situations at the expense of increasing computational complexity of sequence optimization. In Section 6.5 we provide a better (but more complex) heuristic based on dynamic programming.

Finally, to illustrate performance of the greedy approach more comprehensively, we applied it to a large-scale realistic data stream. In particular, we generated a data stream of 1,000,000 records, each containing the values of ten different attributes ($A_1, \ldots, A_{10}$) generated randomly based on ten predefined probability distributions (one for each attribute). Then we specified eight validation operators using labels from set $\mathcal{L} = \{\texttt{Accept}, \texttt{Reject}\}$ and using Boolean combinations of value constraints as predicates. Examples of operators include (`Accept`, "$A_2 < 1$") and (`Reject`, "$(A_3 = 1) \wedge (A_7 > 0)$"). After just eight operators, 789,964 records were already validated (accepted or rejected), or about 79% of the input.

The value of the cost function for our initial sequence s of validation operators was 6,248,799. Using the greedy heuristic, we obtained sequence s' for which the value of the cost function was 4,697,646. Therefore, while the greedy heuristic obviously does not guarantee optimality, it can still significantly improve the performance of the validation sequence (e.g., by about 25% in our case).

6.5. Dynamic-Programming Approach to Sequence Optimization

Assume $s = \langle o_1, \ldots, o_k \rangle$ is an expert-specified validation sequence used to validate data set D. We construct an improved validation sequence $s' = \langle o'_1, \ldots, o'_k \rangle$ as a permutation of s. In the beginning, let $s' := s$. Construction of s' can be described as

```
(1)  for n := 2 to k
(2)    CurrS := s'[1..n]
(3)    BestS := CurrS
(4)    Candidate := n − 1
(5)    for p := n downto 2
(6)      Success := False
(7)      while ¬Success and Candidate ≥ 1
(8)        Success := SWAP(CurrS[Candidate],
             CurrS[Candidate + 1, p])
(9)        Candidate := Candidate − 1
(10)     end while
(11)     if ¬Success then exit loop
(12)     if BestS.Cost < CurrS. Cost then
           BestS := CurrS
(13)     Candidate := Candidate − 1
(14)   end for
(15)   s'[1..n] := BestS
(16) end for
```

Every iteration of this algorithm (lines 2–15) tries to "optimize" a prefix of sequence s of length n, where $n = 2, \ldots, k$ (k is the length of the whole sequence), i.e., for each n the algorithm tries to optimize the sequence $\langle o'_1, \ldots, o'_n \rangle$. Clearly, we do not need to start from $n = 1$, since $\langle o'_1 \rangle$ is by itself optimal, i.e., no permutations are possible in a sequence of length 1. During every iteration, the optimization procedure follows a dynamic-programming approach (Cormen et al. 2001), i.e., the sequence of length n is optimized by taking the already "optimized" sequence of length $n - 1$ (the result from the previous iteration) and by trying to "insert" the nth operator o'_n in all permissible positions. Clearly, we have n possible such positions in the sequence of length n.

More specifically, each iteration starts with the sequence of length n that is constructed from the sequence of length $n-1$ (obtained from the previous iteration) by adding operator o'_n to the end of it, i.e., by keeping o'_n in its current, nth, position (line 2). Furthermore, we assume that this sequence is the best sequence of length n so far (line 3). Then we try to move the operator o'_n up through the sequence (**for** loop, lines 5–14) one position at a time. To keep moving o'_n up through the sequence, some other operator has to be put in its place. The first candidate for that is o'_{n-1} (line 4), i.e., the heuristic will first try to swap o'_{n-1} and o'_n. However, if at any point o'_{p-1} cannot be swapped with o'_p (because their labels are different and their predicates are not orthogonal), we then try to swap o'_{p-2} with *both* o'_{p-1} and o'_p, and so on, increasing the block of operators to be moved up together with the o'_p as necessary (**while** loop, lines 7–10). If we are not able to swap any operator into the pth position, then, naturally, we are done with all permissible permutations for this iteration (line 11). On the other hand, as long as we are able to swap, we compare every new sequence against our best sequence so far, and update the best sequence, if needed (line 12). After we are done with all permissible permutations for this iteration, we keep the best permutation of the first n operators (line 15).

As with our greedy heuristic, the dynamic-programming approach can never increase the cost of the validation sequence because in each iteration we choose the sequence that has lower cost than the default sequence, i.e., than the sequence with o'_n in its initial nth place.

Furthermore, the dynamic-programming approach provides the same theoretical guarantees as does our greedy heuristic, i.e., the performance improvement $\Delta_{s\to s'}$ from the initial expert-specified sequence is guaranteed to be at least

$$\sum_{i:\,(o_i \prec_s o_j)\wedge(o_j \prec_{s'} o_i)} (n_j - n_i), \tag{18}$$

and the above algorithm is always able to present feedback regarding the guaranteed cost savings it produces, based only on its knowledge of initial sequence performance (n_i values). However, the dynamic-programming approach provides better optimization performance than does the greedy heuristic. In particular, it would be able to produce the optimal validation sequence in the "problematic" example from Figure 3(a), which represented the worst-case scenario for our greedy heuristic, because in the sixth iteration ($n = 6$) the above algorithm would be able to move operators 5 and 6 up through the sequence *together*. In other words, it would produce the optimal result, as presented in Figure 3(b).

Computational complexity of the dynamic-programming heuristic is $O(k^3)$. This can be estimated by noticing that each iteration (lines 2–15), i.e., when $n = 2, \ldots, k$, has the worst case computational complexity of $O(n^2)$. In particular, computational complexity of the SWAP operation is proportional to the block size that is being swapped, and this operation cannot be attempted more than $n-1$ times (based on the dynamics of *Candidate* variable), e.g., it can indeed take $O(n^2)$ operations to move a block of $n/2$ operators through the remaining $n/2$ positions. Based on the outer loop (lines 1–16), n goes from 2 to k so the overall computational complexity of the dynamic-programming approach to sequence optimization is $2^2 + 3^2 + \cdots + (k-1)^2 + k^2 = O(k^3)$. While computational complexity of the above algorithm is greater than for the greedy heuristic in most real-life situations, the dynamic-programming approach produces better optimization results, as indicated earlier. However, as with the greedy heuristic, it is very difficult to derive more precise theoretical results about the performance of the above algorithm without understanding the specifics of the domain knowledge, i.e., the underlying data and the validation predicates.

Finally, to illustrate the optimization performance of our dynamic-programming approach more comprehensively and compare it to the greedy heuristic, we tested the dynamic-programming approach in the same experimental setting used to test the greedy heuristic, as described at the end of Section 6.4. Specifically, we used the same expert-specified validation sequence s and the same data stream of 1,000,000 records. Using the dynamic-programming approach, we obtained sequence s'' for which the value of the cost function was 4,440,347 – about a 29% improvement over the initial expert-specified validation sequence (where the cost value was 6,248,799) and a 5.5% improvement over the result of the greedy heuristic (where the cost value was 4,697,646).

7. Summary and Future Work

This paper presented an optimization approach for sequencing a set of validation operators provided by experts in the validation phase of data mining or similar exercises. We explored various properties of such validation sequences, such as sequence equivalence, sequence permutation, and sequence optimality, and derived several theoretical results providing for a better understanding of the validation process. We also addressed the problem of optimizing the expert-specified validation sequence by searching the set of equivalent sequence permutations. This problem can also be viewed as a scheduling problem, where costs of performing individual validation tasks depend highly on the position of this task (and other tasks) in the schedule. We showed that this

problem is NP-hard and presented two algorithms—a greedy heuristic and a dynamic-programming-based approach—that can be used to improve validation performance of a given sequence.

In our approach we view validation as an entirely expert-driven process and rely on the domain expert to provide a validation sequence based on the expert's validation decisions. Moreover, we take the validation sequence produced by the domain expert as exogenous input and try to transform it into an equivalent but computationally faster sequence. In other words, this paper addresses the problem of how to do the validation *faster* given the initial validation results, which is important for several reasons, as discussed in Section 1.

This paper focused on a *general* approach to optimizing validation sequences and did not consider any domain-specific information. In future work we plan to extend our general validation results in several ways. First, by considering *specific* application domains, such as data-mining rules, data streams, database records, and the assignment of classification labels for supervised learning algorithms, we can enhance validation capabilities of the domain expert by incorporating particular knowledge about these domains into the validation process and letting the validation system assist the domain expert to generate better and more efficient validation sequences. For example, a significant amount of research has been done on rule-based *expert systems*. While the area of expert systems does not directly address the expert-driven rule-validation problem, we believe that some of the results on how to deal with evolving rule bases (Agarwal and Tanniru 1992) or how to improve the performance of such systems (Gulati and Tanniru 1993) may lead to improvements of the rule validation approaches. We also believe that incorporating domain knowledge into the validation process will not only result in better validation sequences, but also will allow us to derive more precise theoretical results. Second, we did not address uncertainty as a part of the validation problem, i.e., in our approach we assume that the validation operators assign labels to validated data without any uncertainty. As an initial step, we wanted to study and understand the validation process that is fully deterministic, but we plan to explore incorporating uncertainty into the validation process.

References

Adomavicius, G. 2002. Expert-driven validation of set-based data mining results. Ph.D. thesis, Computer Science Department, New York University, New York, http://www.cs.nyu.edu/web/Research/theses.html.

Adomavicius, G., A. Tuzhilin. 1999. User profiling in personalization applications through rule discovery and validation. *Fifth ACM SIGKDD Internat. Conf. Knowledge Discovery and Data Mining*, 377–381.

Adomavicius, G., A. Tuzhilin. 2001. Expert-driven validation of rule-based user models in personalization applications. *Data Mining and Knowledge Discovery* **5** 33–58.

Agarwal, R., M. Tanniru. 1992. A structured methodology for developing production systems. *Decision Support Systems* **8** 483–499.

Agrawal, R., H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo. 1996. Fast discovery of association rules. U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, R. Uthurusamy, eds. *Advances in Knowledge Discovery and Data Mining*, Chap. 12. MIT Press, Cambridge, MA.

Babu, S., R. Motwani, K. Munagala, I. Nishizawa, J. Widom. 2004. Adaptive ordering of pipelined stream filters. *ACM SIGMOD Internat. Conf. Management Data*, 407–418.

Bernstein, P. A., V. Hadzilakos, N. Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA.

Brachman, R. J., T. Anand. 1996. The process of knowledge discovery in databases: A human-centered approach. U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, R. Uthurusamy, eds. *Advances in Knowledge Discovery and Data Mining*, Chap. 2. MIT Press, Cambridge, MA.

Cormen, T., C. Leiserson, R. Rivest, C. Stein. 2001. *Introduction to Algorithms*, 2nd ed. MIT Press, Cambridge, MA.

Fayyad, U. M., G. Piatetsky-Shapiro, P. Smyth. 1996. From data mining to knowledge discovery: An overview. U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, R. Uthurusamy, eds. *Advances in Knowledge Discovery and Data Mining*, Chap. 1. MIT Press, Cambridge, MA.

Garey, M. R., D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York.

Gulati, D., M. Tanniru. 1993. A model-based approach to investigate performance improvement in rule-based expert systems. *Decision Sci.* **24** 42–59.

Imielinski, T., A. Virmani. 1999. MSQL: A query language for database mining. *Data Mining and Knowledge Discovery* **3** 373–408.

Klemettinen, M., H. Mannila, P. Ronkainen, H. Toivonen, A. I. Verkamo. 1994. Finding interesting rules from large sets of discovered association rules. *Third Internat. Conf. Inform. and Knowledge Management*, 401–407.

Lee, Y., B. G. Buchanan, J. M. Aronis. 1998. Knowledge-based learning in exploratory science: Learning rules to predict rodent carcinogenicity. *Machine Learn.* **30** 217–240.

Liu, B., W. Hsu. 1996. Post-analysis of learned rules. *AAAI Conf.*, 828–834.

Liu, B., W. Hsu, Y. Ma. 1999. Pruning and summarizing the discovered associations. *Fifth ACM SIGKDD Internat. Conf. Knowledge Discovery and Data Mining*, 125–134.

Provost, F., D. Jensen. 1998. Evaluating knowledge discovery and data mining. *Tutorial Fourth Internat. Conf. Knowledge Discovery and Data Mining*.

Sahar, S. 1999. Interestingness via what is not interesting. *Fifth ACM SIGKDD Internat. Conf. Knowledge Discovery and Data Mining*, 332–336.

Silberschatz, A., A. Tuzhilin. 1996. User-assisted knowledge discovery: How much should the user be involved? *SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, Montreal, Canada.

Srikant, R., Q. Vu, R. Agrawal. 1997. Mining association rules with item constraints. *Third Internat. Conf. Knowledge Discovery and Data Mining*, 67–73.

Tuzhilin, A., G. Adomavicius. 2002. Handling very large numbers of association rules in the analysis of microarray data. *Eighth ACM SIGKDD Internat. Conf. Knowledge Discovery and Data Mining*, 396–404.

Tuzhilin, A., B. Liu. 2002. Querying multiple sets of discovered rules. *Eighth ACM SIGKDD Internat. Conf. Knowledge Discovery and Data Mining*, 52–60.