IJS delovno poročilo
DP-15587
2026

**Marko Bohanec**

# DexLearning: Software for Learning and Revision of DEX Models from Data

Jožef Stefan Institute
Ljubljana, Slovenija

DEPARTMENT OF KNOWLEDGE TECHNOLOGIES
Jožef Stefan Institute

# Abstract

*DexLearning* is a research-level open-source software package for machine learning and revision of DEX (Decision EXpert) multi-criteria decision models. *Learning* is the task of constructing a complete DEX model automatically or semi-automatically from a data set of labelled examples, such as data on previously made decisions. The examples associate input features (attributes) with decision outcomes. The learning algorithms construct both the hierarchical (tree) structure of the attributes and the associated decision tables. *Revision* is the task of modifying an existing DEX model to better conform to new data evidence. Learning and revision are implemented in two main command-line programs, *DexLearn* and *DexRevise*. Three supporting programs are also provided in the package: *Experiment*, for conducting multiple learning experiments; *TestModel*, for measuring the accuracy of DEX models on given data sets; and *SplitData*, for splitting a data set into multiple subsets. This report documents these tools, describes the underlying algorithms, and explains their command-line and script-level interaction, illustrated with examples.



## Keywords

Multi-criteria model, method DEX (Decision Expert), machine learning, model revision, software.

# Contents

# 1 Introduction

Multi-criteria models are common types of models used In Decision Making and Decision Analysis (Buede, 2013). A *multi-criteria model* (Greco, et al., 2016) is a way of evaluating and analyzing decision alternatives when there are multiple factors to consider. The *DEX* (*Decision EXpert*) method is a qualitative multi-criteria decision analysis approach (Bohanec, 2022), where all attributes are qualitative and represented by words, attributes are organized hierarchically, and evaluation is defined by decision rules.

Usually, DEX models are developed by human decision makers and experts, who manually develop the hierarchical structure of attributes and decision rules. DEX Software (https://dex.ijs.si/) is a collection of computer programs that support this process. However, there are multiple use-cases that involve data, which can be used to support the construction or adaptation of DEX models. There are several good reasons to learn or revise DEX models from data:

- *Developing a new model from scratch*. Data, such as examples of past decisions, can provide evidence about relationships between attributes and decision outcomes. Machine learning can help discover useful rules, value functions, and even hierarchical structures that can be expressed in terms of a DEX model.
- *Supporting and checking expert modelling*. Even when experts construct the model, empirical data can reveal patterns, inconsistencies, or rules that were overlooked.
- *Adapting an existing model to new evidence*. A DEX model may be well established but become less appropriate as new observations accumulate or the population, environment, or decision context changes. Revision from new data can update model's rules while preserving as much of the original expert knowledge as possible.
- *Reconstructing existing subtrees*. Sometimes the overall hierarchy and most of the model remain useful, but a particular subtree is inappropriately specified. Learning can reconstruct just that subtree while keeping the rest of the DEX model intact.
- *Dealing with incomplete knowledge*. Data-driven revision is useful when experts can specify the concepts and hierarchy but cannot reliably fill in all the detailed decision rules. Learning can suggest or complete the missing parts.

There were several previous attempts at DEX model learning. In the 1990's, Zupan (1997), Zupan et al. (1999), and Bohanec and Zupan (2004) developed two algorithms for learning complete DEX models from data: *HINT* (HIerarchy Induction Tool) and its *Minimum Error* variant (Zupan, 1997). In the early 2000's, Žnidaršič (2007), Žnidaršič and Bohanec (2004), and Žnidaršič et al. (2009) developed a model revision algorithm that updates DEX decision rules while leaving the model structure intact. Initially, all of these algorithms were actually implemented and operational. Over time, however, they were no longer regularly maintained and gradually disappeared from practical use. Nevertheless, the use cases for such tools remain, and there is a clear practical motivation for operational software supporting DEX model learning and revision. *DexLearning* is a software package that aims to bridge this gap, reimplementing the algorithms and providing research-level tools for skilled decision analysts and researchers.

As a side note, there have also been other approaches to DEX model learning. Bohanec and Delibašič (2015) proposed a semi-automated method for developing decision tables based on conditional probabilities, while Radovanović et al. (2023) developed DIDEX (Data-Induced DEcision eXpert), a method that automatically constructs interpretable hierarchical DEX models from data by iteratively discovering attribute interactions and aggregation rules. These approaches are currently not explicitly supported by DexLearning.

This report describes DexLearning from the user's perspective, focusing on the needs of analysts and model developers. Section 2 introduces the purpose of the software, its main components, and its limitations. Section 3 introduces a simple model for evaluating cars, which is used throughout the report to illustrate the algorithms and programs. Section 4 describes the characteristics and formats of the data sets used by the software. The following five sections are devoted to the five individual programs that constitute the package.

A companion report (Bohanec, 2026) presents results of an experimental evaluation of DexLearning using a variety of datasets.

# 2 DexLearning Software

*DexLearning* is a research-level software for machine learning and revision of DEX models from data. This report describes version 0.1 (2026) of the package. The software is open-source, available at GitLab https://repo.ijs.si/markobohanec/dexlearning under the MIT license.

## 2.1 Components

DexLearning consists of five distinct, but related, command-line programs. There are two main programs:

- *DexLearn*: Learning complete DEX models from data. The learning task is defined in terms of a script (`.json`) or commands (`.dxl`) file. Requires a tab-delimited dataset of examples (`.tab`). Scales of attributes may be optionally defined in a separate json-formatted `.scr` file. Implements two learning algorithms: HINT and MinError. Facilitates automatic and semi-automated (supervised) learning.
- *DexRevise*: Model revision. Requires an already defined `.dxi` model, a learning dataset to which to adapt the model, and an optional validation dataset.

The remaining three programs are supplementary utilities:

- *Experiment*: Perform a series of experiments with DexLearn, based on a `.json` script and varying learning parameters, such as the percentage of examples used for learning.
- *TestModel*: Assess classification accuracy of a DEX model on given datasets.
- *SplitData*: A tool for splitting a dataset into multiple subsets.

The software package also contains:

- *DexLearning:* A common class library used by the programs. It implements various utilities and classes that implement essential elements: data tables, decision trees, learning algorithms, DEX models and their components, partition algorithms and Bayesian statistics.
- *TestLearning:* Unit testing of DexLearning.

## 2.2 Implementation

The software is implemented using *RemObjects Oxygene*, a new-generation Object Pascal. While the DexLearning code itself is open-source, RemObject Oxygene is proprietary and requires a license for software development. Building programs may still be possible using the open-source builder EBuild, https://elementscompiler.com/elements/ebuild.aspx.

All the five programs are pre-built and provided as Windows `.exe` files on the `./Executables/` directory.

## 2.3 Limitations

- DexLearning is a free *research-level software*. Its functionality may change without notice, and its documentation may be incomplete or delayed.
- All programs are of the *command-line* type, and require some understanding of scripts, data formats and computer file systems.
- DEX models are *qualitative*, and so is the data that can be handled by DexLearning. DexLearning can only read data sets from text files, with one-line headings and tab-delimited lines. All input data items are interpreted as character strings.

- DexLearning is not designed for big data. The theoretical upper limit for data set size is 2,147,483,591 elements, but the practical limit is much lower due to high time complexity of involved algorithms.
- Software is implemented for the *Windows* operating system. No emulation under Linux of macOS has been tested so far.
- The DexLearning algorithms have been *reimplemented* from the scratch, considering only the available literature, without having access to any legacy code. Therefore, some algorithmic steps and control parameters might be different from the originals, and they may likely yield different results.
- The current implementation of DexLearning prioritizes *functionality* over efficiency. Consequently, certain components are likely suboptimal, both in runtime performance and architectural design, and may benefit from future refinement.

## 2.4 Repository Structure

The GitLab repository consists of the following directories:

- `.` (root directory): Essential files:
  - `DexLearning.sln`: A Visual Studio solution, the root of all software projects that constitute this package;
  - `README.md` and `LICENSE.md`: Basic information about the package;
  - `.git/` and `.gitignore`: Git-related information.
  - There may be other files located at the root directory, to be systematized and documented later, or removed.

- `./Data/`: Multiple data (`.tab`) and scale-definition (`.scl`) files to be used in experiments.
- `./DexiModels/`: Multiple DEX models to be used in experiments.
- `./Scripts/`: Collection of scripts for running DexLearn.

- `./DexLearning/`: Sources of common utility (`Utils`) and `Learning` programming units.
- `./DexLearn/`: Sources of the `DexLearn` program.
- `./DexRevise/`: Sources of the `DexRevise` program.
- `./Experiment/`: Sources of the `Experiment` program.
- `./SplitData/`: Sources of the `SplitData` program.
- `./TestModel/`: Sources of the `TestModel` program.
- `./TestLearning/`: DexLearning unit testing.

- `./Documentation/`: Software documentation; for instance, this report.
- `./Executables/`: Compiled programs and software libraries, prepared for direct execution.
- `./Results/`: Collection of experimental results; maintained and possibly overwritten while running DexLearn scripts.
- `./Examples/`: Examples used in documentation.
  - `./Examples/Car_Rev/`: Examples used in this report.
- There may exist other temporary directories, such as `./Test/` and `./Outputs/`.

## 2.5 Software Installation

In principle, no installation is needed. Just copy the contents of the `./Executables/` folder to your computer and run the programs from there.

Make sure to adapt directory names in script files to match your computer.

For safety, you may want to check the provided MD5 and SHA205 checksums.

# 3 Running Example

In order to illustrate the DexLearning programs, and the DEX method itself, we shall use a simple didactic DEX model for evaluating cars. The model is adapted from Žnidaršič and Bohanec (2004), where it was used to illustrate model revision. This model is different and smaller than the CAR model that is distributed with the DEXiWin software (Bohanec, 2024) and widely referenced in decision-support and machine-learning literature. On the GitLab repository, the model is stored under the name `./Examples/Car_Rev/Car_Rev0.dxi`. This model, as well as all other models stored in `.dxi` files, can be viewed and edited using the DEXi or DEXiWin software, see https://dex.ijs.si/.

The model is aimed at the selection of a family car. According to the structure of attributes shown in Figure 1, the evaluation of the goal attribute "car" is decomposed into individual evaluation of two subordinate subtrees, "costs" and "safety". Each of them is further decomposed into basic model attributes: "price" and "maintenance", and "ABS" (Anti-lock Braking System) and "size" (of the car). Figure 1 also shows attribute *scales*: the sets of values that can be assigned to individual attributes. Since DEX is a qualitative method, the scales are qualitative, too, and consist of a few words, typically from two to five, rarely more. In DEX, scales are usually *preferentially ordered* from "bad" to "good" values, where particularly bad and good values are traditionally displayed in red and green, respectively.

| Attribute | Scale |
|---|---|
| car | bad; acceptable; middle; good; *very good* |
| ├─costs | high; middle; *low* |
| │ ├─price | high; middle; *low* |
| │ └─maintenance | high; middle; *low* |
| └─safety | bad; acceptable; good; *very good* |
| ├─ABS | no; *yes* |
| └─size | small; middle; *big* |

*Figure 1: Model Car_Rev0: Structure and value scales of attributes.*

| | costs | safety | car |
|---|---|---|---|
| 1 | high | bad | bad |
| 2 | high | acceptable | bad |
| 3 | high | good | middle |
| 4 | high | *very good* | good |
| 5 | middle | bad | bad |
| 6 | middle | acceptable | acceptable |
| 7 | middle | good | good |
| 8 | middle | *very good* | good |
| 9 | *low* | bad | bad |
| 10 | *low* | acceptable | middle |
| 11 | *low* | good | good |
| 12 | *low* | *very good* | *very good* |

| | price | maintenance | costs |
|---|---|---|---|
| 1 | high | high | high |
| 2 | high | middle | high |
| 3 | high | *low* | middle |
| 4 | middle | high | high |
| 5 | middle | middle | middle |
| 6 | middle | *low* | *low* |
| 7 | *low* | high | middle |
| 8 | *low* | middle | *low* |
| 9 | *low* | *low* | *low* |

| | ABS | size | safety |
|---|---|---|---|
| 1 | no | small | bad |
| 2 | no | middle | acceptable |
| 3 | no | *big* | good |
| 4 | *yes* | small | bad |
| 5 | *yes* | middle | good |
| 6 | *yes* | *big* | *very good* |

*Figure 2: Decision tables corresponding to the "car", "costs" and "safety" aggregate attributes, respectively.*

The aggregation of attributes is represented in terms of *decision tables*, as shown in Figure 2. Each aggregate attribute (internal node in the tree) is associated with a decision table that maps all combinations of its subordinate attributes' values to the values of that attribute. For example, the "safety" table (Figure 1, far right) maps all the six combinations of the two "ABS" and three "size" values to the four possible values of "safety". Usually, such tables are defined interactively by decision makers, decision analysts and experts, using the software such as DEXi and DEXiWin.

In addition to facilitating table editing, the software encourages the users to define the tables to the maximum extent possible, while obeying the "principle of dominance". Namely, the fact that DEX attributes are usually, but not necessarily, *preferentially ordered* implies that the corresponding decision tables should be *monotone*: when the values of an ordered attribute increase, the outcome should increase, too, or remain constant. All decision tables in Figure 2 are monotone.

# 4 Datasets: Format and Basic Principles

DEX models are qualitative and contain only symbolic variables (attributes). Consequently, DexLearning interprets all datasets qualitatively and views all data items, including those composed of numbers, as character strings.

Data processing in DexLearning proceeds through three main stages:

- Reading data from a tab-delimited text file and representing it internally in the memory.
- Ordinalizing the data: determining attribute scales and converting text values to ordinal numbers.
- Making DEX-like decision tables based on: (1) selecting input attributes (table arguments), (2) selecting a single output attribute, and (3) creating decision rules that cover the whole decision space.

The data corresponding to the former two stages are internally represented and processed by classes `StringDataTable` and `IntDataTable`, respectively. The latter stage involves decision tables of two types:

- `IntTable`, where decision outcomes are represented as single values of the output,
- `ProbTable`, where outcomes are represented by value frequencies and probability distributions.

## 4.1 StringDataTable: Input Dataset

DexLearning reads datasets from tab-delimited text files and internally stores them in a `StringDataTable` object. The input file consists of lines where data items are separated by tab characters. The first line is a heading that defines the names of attributes corresponding to individual columns. The example in Figure 3 illustrates a possible beginning (first ten lines) of a dataset (stored on `./Examples/Car_Rev0-.tab`) that can be used to learn the *Car_Rev0* model. Note that only the basic attributes and the output attribute "car" are present in the dataset (aggregate attributes are created by learning). There is also an index column labelled "#", which is not needed for further processing and is going to be eventually discarded.

| # | price | maintenance | ABS | size | car |
|---|---|---|---|---|---|
| 1 | high | high | no | small | bad |
| 2 | high | high | no | middle | bad |
| 3 | high | high | no | big | middle |
| 4 | high | high | yes | small | bad |
| 5 | high | middle | no | middle | bad |
| 6 | middle | high | yes | small | bad |
| 7 | high | low | yes | small | bad |
| 8 | high | middle | no | small | bad |
| 9 | high | middle | no | big | middle |

*Figure 3: First ten lines of a string data file for learning the Car_Rev0 model.*

It also important to understand that DexLearning, while reading a string data file, tries to establish value scales of attributes. A scale is composed of an ordered list of distinct string values corresponding to some attribute, such as "high", "middle", "low" for the attribute "maintenance". Scale order is formed according to the *order of appearance of values* in the dataset. This is often, but not always, appropriate. In the example from Figure 3, all scale orders are properly established, except for the attribute "car"; there, the order of appearance in the file is "bad", "middle", "good", "acceptable", "very good", while the correct order (Figure 1) is "bad", "acceptable", "middle", "good", "very good". For the time being we shall leave this incorrect order and show how to handle such situations later in sections 5.3.2 and 5.3.3.

## 4.2 IntDataTable: Dataset Represented Using Ordinal Numbers

After reading the dataset and establishing value scales, the data is ordinalized by converting string values to ordinal numbers according to each attribute's scale. Internally, this data is stored in a `IntDataTable` object. Figure 4 shows the heading and first nine rows of the `IntDataTable`, which was converted from the `StringDataTable` shown in Figure 3. Ordinal numbers are zero-based.

| # | price | maintenance | ABS | size | car |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 2 | 1 |
| 3 | 0 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 1 | 1 | 1 |
| 5 | 0 | 0 | 1 | 2 | 2 |
| 6 | 0 | 1 | 0 | 0 | 0 |
| 7 | 0 | 1 | 0 | 1 | 0 |
| 8 | 0 | 1 | 0 | 2 | 1 |
| 9 | 0 | 1 | 1 | 0 | 0 |

*Figure 4: Car dataset represented as an `IntDataTable`, converted from `StringDataTable` in Figure 3.*

## 4.3 IntTable: A DEX Decision Table with Single-Value Output

DEX models operate with *decision tables*. Each decision table (such as the three examples in Figure 2) is defined in terms of its input attributes (called *arguments*) and a single *output* attribute. The table itself is composed of an ordered collection of decision rules that *cover the whole decision space*. In other words, decision table contain *decision rules* that correspond to *all possible combinations* of arguments' values.

In DexLearning, decision tables are represented by objects of two types, `IntTable` and `ProbTable`. Both are created from some `IntDataTable` after specifying: (1) the list of arguments, and (2) the output attribute.

| price | maintenance | ABS | size | car |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 2 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | -1 |
| 0 | 0 | 1 | 2 | -1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 2 | 1 |

*Figure 5: Heading and the first nine decision rules of the `IntTable` obtained from Figure 4.*

Decision rules in an `IntTable` are restricted to having only a single output value (called *outcome*). This can be either an ordinal number corresponding to the output attribute scale, or the value `-1` that denotes an undefined outcome. Figure 5 displays the topmost part of the `IntTable` obtained from the previous `IntDataTable` (Figure 4) by declaring "price", "maintenance", "ABS", and "size" arguments of the table, and "car" the output attribute. Notice that the column "#" has been excluded, and that "car", as a distinguished output attribute, is always displayed at the rightmost column. Also notice two undefined (`-1`) "car" values, which indicate that the corresponding arguments' values are not present in the original dataset.

`IntTable` is suitable for representing data that has no or little noise, and where multiple data items do not contradict each other by assigning different outcomes to the same combinations of input attribute values. DexLearning handles such situations by issuing warnings during conversion and assigning only

one of the multiple outcomes to conflicting rules. Consequently, the more the noise, the less reliable is the `IntTable` data representation. In general, conversion from `IntDataTable` to `IntTable` is lossy.

## 4.4 ProbTable: A DEX Decision Table with Probabilistic Output

Similar to `IntTable`, `ProbTable` assigns outcomes to all possible value combinations of decision table arguments. However, the outcomes are represented in terms of value frequencies and/or probability distributions. Figure 6 shows the same contents as Figure 5 represented in this way. Frequencies are integer arrays counting the numbers of occurrences of the corresponding output values in the data. Probabilities are real-numbered arrays usually obtained from frequencies by normalizing the sum of elements to 1. In comparison with `IntTables`, `ProbTables` are better suited for representing noisy data. Conversion from `IntDataTable` to `ProbTable` is lossless.

| price | maintenance | ABS | size | car(freq) | car(prob) |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1;0;0;0;0 | 1;0;0;0;0 |
| 0 | 0 | 0 | 1 | 1;0;0;0;0 | 1;0;0;0;0 |
| 0 | 0 | 0 | 2 | 0;1;0;0;0 | 0;1;0;0;0 |
| 0 | 0 | 1 | 0 | 1;0;0;0;0 | 1;0;0;0;0 |
| 0 | 0 | 1 | 1 | 0;0;0;0;0 | 0;0;0;0;0 |
| 0 | 0 | 1 | 2 | 0;0;0;0;0 | 0;0;0;0;0 |
| 0 | 1 | 0 | 0 | 1;0;0;0;0 | 1;0;0;0;0 |
| 0 | 1 | 0 | 1 | 1;0;0;0;0 | 1;0;0;0;0 |
| 0 | 1 | 0 | 2 | 0;1;0;0;0 | 0;1;0;0;0 |

*Figure 6: Heading and the first nine decision rules of the `IntTable` obtained from Figure 4.*

# 5 DexLearn: Machine Learning of DEX Models from Data

DexLearn is the central program of the DexLearning software package. It is aimed at machine learning of DEX models from data. It implements two learning algorithms based on function decomposition: HINT and MinError. Supports both the automatic in semi-automatic (supervised) development of DEX models. Also, it implements two decision tree-learning algorithms, C4.5 and ID3, which are mainly used for comparison with HINT and MinError.

Learning of DEX models proceeds in different stages, from data preparation to actual learning and assessment of model quality. In terms of user interface, DexLearning uses two types of script files, called `.json` and `.dxl`, that serve to define the steps of carrying out a given learning task.

DexLearn is invoked by the command:

```
DexLearn script [more scripts...]
```

Script files are of two types: (1) containing a json-formatted description of tasks (standard file extension `.json`), and (2) containing commands to be run in sequence (`.dxl`). The type is determined from file contents, not its extension: json-formatted files are assumed to begin with "`[{`", separated by any amount of whitespace.

In the remaining part of this section, the algorithms HINT and MinError are first described. This is followed by a description of script files and an example of using DexLearn.

## 5.1 HINT: Learning DEX Models by Decomposition

The *Hierarchy Induction Tool* (HINT), developed by Zupan and colleagues in the 1990s (Zupan, 1997; Zupan et al., 1999; Bohanec and Zupan, 2004), is a machine learning method that tackles complex problems by breaking them down into a hierarchy of simpler subproblems. Its core idea is *function decomposition*: given a set of training examples that define a target concept (a function), HINT looks for intermediate, "hidden" concepts that can be defined from a subset of the original input features. The goal is to transform the original, complex function $y = F(X)$ into a hierarchy of simpler functions, such as $y = G(A, c)$ and $c = H(B)$, where the set of input attributes $X$ is partitioned into two disjoint subsets, the *free set* ($A$) and the *bound set* ($B$). This process is applied repeatedly to the newly created sub-functions, building a hierarchy of attributes (Algorithm 1). The resulting structure is a *hierarchy of concepts*: unlike a decision tree, its internal nodes represent functions/concepts and its leaves represent original attributes. The new aggregate attributes $c$ are not present in the input data, and are constructed by HINT itself; this characterizes it as a *constructive induction* machine learning algorithm.

```
function HINT(F, X):
    hierarchy = []
    decomposition = DECOMPOSE(F, X)

    while decomposition is not None and complexity_improved(decomposition):
        hierarchy.append(decomposition)
        // Replace F with the new sub-functions G and H
        // and add c to the set of attributes for further decomposition
        F = decomposition.G
        X = decomposition.A + decomposition.c
        decomposition = DECOMPOSE(F, X)

    return hierarchy
```

*Algorithm 1: Top-level pseudocode of HINT.*

At the heart of HINT's single-step decomposition (Algorithm 2) is a clever use of *graph coloring*. To find the intermediate concept $c = H(B)$, the algorithm creates a *partition matrix*. The rows of this matrix

are labelled by the values of the free set $A$, and the columns are labeled by the combinations of values in the bound set $B$. Each column represents the behavior of the target function $F$ for a given combination of bound attributes. The algorithm then builds an *incompatibility graph* where each node is a column from the matrix. An edge connects two nodes if their corresponding columns are incompatible, meaning they have conflicting output values for the same row in the free set. The problem of finding the simplest new feature $c$ (with the fewest possible values) is then reduced to finding the optimal coloring of this incompatibility graph. Each color represents a unique value for the new feature $c$, ensuring that incompatible columns, which must map to different concepts, are assigned different labels. DexLearn employs DSATUR, a heuristic coloring algorithm (Brélaz, 1979).

```
function DECOMPOSE(target_function F, attributes X):
    best_decomposition = None
    best_complexity = INFINITY

    for each partition of X into free_set A and bound_set B:
        // 1. Build the partition matrix
        matrix = create_partition_matrix(F, A, B)

        // 2. Build the incompatibility graph
        graph = create_incompatibility_graph(matrix)

        // 3. Find the optimal coloring for the graph
        //    The number of colors needed is the number of values for new feature
c
        coloring = graph_color(graph)
        num_colors = count_colors(coloring)

        // 4. Derive new functions G and H from the colored matrix
        H = derive_function(B, coloring) // H maps B to c
        G = derive_function(A, c, matrix) // G maps A and c to y

        // 5. Evaluate the complexity of this decomposition
        current_complexity = estimate_complexity(G, H, num_colors)

        if current_complexity < best_complexity:
            best_complexity = current_complexity
            best_decomposition = (G, H, A, B, c)

    return best_decomposition
```

*Algorithm 2: Function decomposition step of HINT.*

The actual implementation of HINT in DexLearn has the following characteristics:

- All HINT's "functions" are internally represented by `IntTables`. That is, the initial dataset must be represented in this way, as are all the decomposed functions. This makes them directly representable in the resulting DEX models.
- While the original HINT typically restricted the size of the bound set to 3, DexLearn, in its automatic mode, searches only for bound sets of size 2.
- However, DexLearn does not only look for a single model hierarchy, but, in general, generates multiple solutions. At each single-step decomposition of $F$, it tries all "promising" bound sets $B$, searching for multiple alternative hierarchies. The "promising" criterion can be parameterized, but generally looks for new concepts with the smallest number of scale values.
- In order to `estimate_complexity()` in Algorithm 2, DexLearn assesses the *information content* of tables $G$ and $H$ (in bits). Bit size of an `IntTable` that contains $r$ rules and whose output attribute has $v$ values is $r \log_2 v$.
- The quality of produced models is assessed in terms of *classification accuracy* on given (training or testing) datasets.
- DexLearn's HINT accepts the following parameters:

- `ProcessInputs: Boolean`: Whether or not to pre-process the initial `IntTable` by finding redundant attributes and scale values (default: `true`)
- `CanDeleteInputs: Boolean`: Whether or not to delete redundant attributes from the initial `IntTable` prior to running HINT or MinError (default: `true`)
- `MaxSolutions: Integer`: Maximum number of solutions to be generated (default `-1`: unlimited)
- `MaxSave: Integer`: Maximum number of best generated solutions to be saved on `.dxi` files (default `-1`: unlimited)
- `MaxPerNode: Integer`: Maximum number of solutions to be generated per node (default `-1`: unlimited)
- `MaxOutValues: Integer`: Maximum allowed number of class values (default `-1`: unlimited)
- `BestMinSelect: Integer`: Number of minimal graph colorings to be considered (default `-1`: as many as there are)
- `BestTolerance: Integer`: Allowed tolerance when selecting graph colorings (`0` means that only the best ones are selected, `1` selects the best and those having 1 value more, etc.)
- `MEstimate: Float`: M-estimate parameter (Cestnik, 1990) for assessing class probabilities in MinError (default: `2.0`). See Appendix 1 for more information about the m-estimate.

## 5.2 MinError: Minimum Error Decomposition

*Minimum Error* (*MinError*) is a somewhat less known variant of HINT (Zupan, 1997), better adapted to handling noisy data. The standard HINT algorithm searches for a decomposition that is exact on the training data. However, real-world data can be noisy or inconsistent. In such cases, an exact decomposition might be overly complex, overfit the data, and fail to generalize well. The MinError variant tackles this by changing the objective of the decomposition search.

Instead of looking for a decomposition that perfectly matches all training examples, MinError aims to find a hierarchy that minimizes a *classification error* estimate (like the m-estimate) on the training set. The core idea is to trade off a perfect fit on the training data for a simpler, more robust model that is less likely to have memorized noise. Essentially, MinError performs a trade-off, guided by the m-estimate, to select the most promising decomposition in terms of its expected predictive performance on unseen data.

In contrast with the basic HINT, MinError operates on a `ProbTable` rather than `IntTable`. If necessary, `ProbTable` is converted from `IntTable` prior to execution. Both HINT and MinError algorithms can be run in a single learning task, if defined so in the script files.

## 5.3 DexLearn Script File

DexLearn script file (default extension `.json`) is a static json-formatted description of one or more tasks to be performed by DexLearn. The description generally consists of several json objects, enclosed in '`{`' and '`}`', hereafter called *sections*.

### 5.3.1 Declaration Sections

The script file typically begins with five sections that declare the task name, provide an optional description, define input and output directories, and parameterize logging and saving of files.

Figure 7 shows an example of these five sections for learning the *Car_Rev0* model:

- `Task` and `Description` name and describe the task. `Name` is occasionally used as a default name for files when they are not specified otherwise.

- `Directories`: `Input` and `Output` are the names of two main folders for reading data and writing results. They are used unless explicitly overridden in file names in the remaining sections. `Create` and `Clear` control whether or not the output directory is created and/or cleared before the execution. `Subdirectories` specifies whether results are written directly in the output directory (`false`), or are related groups of results written in subdirectories below the output (`true`).
- `Logging` controls the amount of information displayed on the console and, optionally, in a log file while executing the program. `Verbosity` level is a number between 0 and 2, where 2 is the most verbose. `LogStatistics` and `LogStructure` determine whether or not to log the statistics and structure, respectively, of generated models. `LogToFile` specifies whether or not logs are written to a file in addition to the console.
- `Saving` determines the contents and extent of results that are written to the output directory. Learned DEX models can be written in the `.dxi` format (`SaveDexi`) to be later processed by DEXi or DEXiWin software, and their structure can be written to `.gml` files (`SaveGml`) to be inspected by graphic programs, such as `Graphwiz dot` or `yed`. `SaveSolutions` controls whether or not generated solutions are saved at all, and `SaveOnlyBest` controls whether only best solutions (those with the smallest bit size) or all solutions are saved. The parameters `Delimiter`, `Decimal` and `Format` control the format of displayed floating point numbers.

```
[{
    "Task": "CarDP",
    "Description": "Car_Rev0.dxi example for the report",
    "Directories": {
        "Input": "/DexLearning/Data",
        "Output": "/DexLearning/Output",
        "Create": false,
        "Clear": false,
        "Subdirectories": false
    },
    "Logging": {
        "Verbosity": 1,
        "LogStatistics": true,
        "LogToFile": true,
        "LogStructure": false
    },
    "Saving": {
        "SaveDexi": true,
        "SaveGml": true,
        "SaveSolutions": true,
        "SaveOnlyBest": true,
        "Delimiter": "\t",
        "Decimal": ",",
        "FloatFormat": "0.00"
    },
```

*Figure 7: First five sections of an example DexLearn script file.*

### 5.3.2 Data Section

`Data` is an important section of the script file that specifies how to prepare data for learning (Figure 8).

```
    "Data": {
        "Name": "Car_Rev0",
        "DataFileName": "Car_Rev0.tab",
        "LoadStringData": true,
        "MakeIntData": true,
        "Scales": [{
                "AttributeName": "car",
                "ScaleValues": [ "bad", "acceptable", "middle", "good", "very good" ],
                "Ordered": true
            }],
        "ExcludeAttributes": ["#"],
        "OutputAttribute": "car",
        "MakeIntTable": true,
        "MakeProbTable": true,
        "MakeFlatModel": true
    },
```

*Figure 8: Data section of a DexLearn script file.*

Elements of the Data section are:

- Name: Name of the data. Used mainly for logging.
- DataFileName: Name of the .tab file that contains initial StringDataTable.
- LoadStringData: Whether or not to actually read StringDataTable. Required for any further data processing.
- MakeIntData: Whether or not to convert StringDataTable to IntDataTable.
- IncludeAttributes and ExcludeAttributes: Lists of IntDataTable attributes (arguments) to be included in or excluded from, respectively, the generated IntTable.
- OutputAttribute: Name of the IntTable and ProbTable output attribute.
- MakeIntTable and MakeProbTable: Whether or not to actually construct IntTable and/or ProbTable.
- MakeFlatModel: After creating an IntTable, whether or not to save a flat DEX model consisting of only that table.
- Scales: As already indicated in section 4.1, automatic detection of attribute scales does not always work correctly. This Scales subsection offers a way to enforce different scales for particular attributes. It consists of a list of elements, specifying:
  - Name: Attribute name;
  - ScaleValues: A *properly* ordered list of scale values;
  - Ordered: Whether or not the scale is preferentially ordered in an ascending order.

### 5.3.3 Scales Section

Scales is an optional section that can be also used for overriding inappropriately detected attribute scales (Figure 9). The ScalesFileName element may specify the name of external file that contains scale definitions, using the same json format as described above. This file can be read prior to processing (ReadScales) and/or written out afterwards (WriteScales). The idea is that upon the first reading of an IntDataFile, scales are written to a file, which is then edited and, corrected, read back by DexLearn at each next running session. ReadScalesIfExists and WriteScalesIfNotExists are useful variations of reading and writing scales, conditioned to whether the scale file already exists or not.

```
    "Scales": {
        "ScalesFileName": "CarDP.scl",
        "ReadScales": false,
        "ReadScalesIfExist": true,
        "WriteScales": false,
        "WriteScalesIfNotExist": true
    },
```

*Figure 9: Scales section of a DexLearn script file.*

### 5.3.4 RunHINT Section

The RunHINT section of the script file controls the execution of the HINT learning algorithm in automatic mode, where decomposition steps are determined by the algorithm itself. Figure 10 shows the default settings. The parameters Run, RunHINT and RunMinError determine whether this entire task, or particular algorithms HINT and MinError, are run or not. Checking refers to internal program checking, which verifies whether the heuristic and exact graph coloring algorithms produce compatible results. Such checking is very slow and is recommended only for debugging. ConvertProbToInt determines whether the resulting ProbTables, constructed by MinError, are converted to IntTables. Namely, only IntTables can be included in .dxi files to be read by DEXi and DEXiWin software. However, bear in mind that conversion from ProbTables to IntTables is lossy.

```
    "RunHINT": {
        "Run": true,
        "RunHINT": true,
        "RunMinError": false,
        "Checking": false,
        "ProcessInputs": true,
        "CanDeleteInputs": true,
        "MaxSolutions": -1,
        "MaxSave": -1,
        "MaxPerNode": -1,
        "MaxOutValues": -1,
        "BestMinSelect": -1,
        "BestTolerance": 0,
        "MEstimate": 2.0,
        "ConvertProbToInt": true
    },
```

*Figure 10: Default settings of the RunHINT section.*

All the remaining parameters in Figure 10 have been already explained in section 5.1.

### 5.3.5 SupervisedHINT Section

`SupervisedHINT` is an optional section in the script file that invokes HINT in the supervised mode: instead of automatic searching for multiple solutions, the program only follows the user's directions and carries out the requested decompositions. The core of this section is the `Decompose` element, which defines a sequence of how to bind input attributes so as to create new aggregate ones. Figure 11 shows an example that, based on the `Car_Rev0.tab` dataset, generates a DEX model that closely resembles the structure of the original model (Figure 1).

```
        "SupervisedHINT": {
            "Run": true,
            "MinError": false,
            "Checking": false,
            "ProcessInputs": true,
            "CanDeleteInputs": true,
            "Decompose": [
                { "Name": "costs", "Bind": ["price", "maintenance"]},
                { "Name": "safety", "Bind": ["ABS", "size"]}
            ]
```

*Figure 11: Example of a SupervisedHINT section.*

### 5.3.6 ConditionalProbabilities Section

`ConditionalProbabilities` is an optional section that triggers a calculation of various measures and statistics of input data, which are potentially useful for manual construction of DEX models, in the spirit of Bohanec and Delibašić (2015). The section has three Boolean parameters (Figure 12):

- `Run`: Whether or not to run this section at all.
- `Measures`: Whether or not to calculate measures related to pairs of input attributes: Entropy, Gini, Average Entropy, Average Gini, Mutual Information, Synergy, Gain Increase, and Bit contents.
- `Pairs`: Whether or not to produce Bayesian statistics of probability tables composed of pairs of input attributes.

```
    "ConditionalProbabilities": {
        "Run": true,
        "Measures": true,
        "Pairs": true
    }
```

*Figure 12: Example of a ConditionalProbabilities section.*

Results are stored on a comma-separated file, which can be further processed in spreadsheet software.

## 5.4 DexLearn Commands

Commands are a recent addition to DexLearn, aimed at tasks for which a static json specification is too rigid. A command file (`.dxl` by default) is composed of a sequence of commands, which are interpreted sequentially. Commands are of the form

```
command {command parameters in json format}
```

The `command` itself can be abbreviated to the extent that can still be distinguished from other commands. The `{}` part can be omitted, in which case the default parameter values are assumed. The `{}` part can also span multiple lines. Lines starting with `#` or `//` are comments and are ignored.

DexLearn's commands are:

- `help`: write the list of all commands and their arguments (such as in Appendix 2)
- `exit`: terminate the execution of commands
- `directories`: define directories for reading data and writing results
- `logs`: define the outputs and extent of logging
- `scales`: read and write scale definitions
- `saving`: define the extent and formats of saved data
- `loaddata`: load a `StringDataTable`
- `savedata`: define tables to be saved, and their format
- `preparedata`: carry out the data preparation
- `hint`: run HINT in automatic mode
- `minerror`: run MinError in automatic mode
- `supervised`: run HINT in supervised mode
- `c45`: run the C4.5 decision tree learning algorithm
- `id3`: run the ID3 decision tree learning algorithm
- `probabilities`: calculate measures and/or conditional probabilities of attribute pairs

Parameters to these commands are very similar to the ones in the script file (section 5.3) and are thus not repeated here. Instead, all commands are listed in Appendix 2 together with their default parameters.

## 5.5 Example

An example for running DexLearn on `Car_Rev0.tab` data is prepared in the `./Examples/Car_Rev` directory.

The script file is called `DexLearn_CarRev0.json` and is activated using the command

```
DexLearn DexLearn_CarRev0.json
```

The script carries out the following:

- Loads dataset `Car_Rev0.tab`
- Creates (on the first run) or reads (on subsequent runs) the scale file `Car_Rev0.scl`
- Creates and clears the output directory `./Examples/CarRev0/DexLearn`
- Internally creates all types of data tables: `IntDataTable`, `IntTable`, `ProbTable`
- Creates and saves on the output a flat DEX model (`DexLearningCarRev0[Flat].dxi`) and its structure (`DexLearningCarRev0[Flat].gml`)
- Runs HINT and saves the best model on `DexLearningCarRev0[1].dxi` and `DexLearningCarRev0[1].gml`

- Writes the statistics of generated solutions on `Solutions.tab`
- Writes measures and Bayesian statistics of attribute pairs on `AttributePairs.csv`

In order to run `SupervisedHINT` instead of automatic HINT, just toggle the corresponding `Run` parameters in the script and rerun the command. In order to avoid overwriting files with equal names, consider changing the output directory.

***Important***: It is assumed that this example is run while being positioned in the directory `./Examples/Car_Rev`, and that the program `DexLearn.exe` is on the system's search path. For running this and other scripts on a computer other than this author's, make sure to properly set the `Input` and `Output` directories in the json file, and properly formulate the program call.

# 6 Experiment: Perform Multiple Experiments with DexLearn

Experiment is a supplementary program, designed to run multiple DexLearn tasks, using different machine learning methods and varying their parameters. It uses the same script (`.json`) files as DexLearn and is invoked using the command

```
Experiment script.json [more scripts...]
```

The command format (section 5.4) is not supported.

The experiments carried out by Experiment are defined in an optional section of the script file, called `ExperimentParameters`. Figure 13 shows the default settings of this section. The `Run` parameter determines whether or not experiments are carried out. `TestOnAll` selects whether the models are tested on testing data (`false`) or all data available (`true`). `Count` defines the number of repetitions on the same data. `Sizes` is a list of numbers between 0 and 100, specifying the percentage of data to be used for training. `Methods` defines the list of methods to be run and compared with each other; all available methods are listed in Figure 13. `MEstimates` is a list of m-estimates to be used by MinError (see Appendix 1). `RandomSeed` provides an integer random seed so as to ensure the repeatability of results. Finally, `Output` defines the output directory on which to write the results. In contrast with DexLearn, this directory must be created manually prior to execution, in order to avoid unwanted overwriting of data.

```
···"ExperimentParameters":·{
·····"Run":·false,
·····"TestOnAll":·false,
·····"Count":·10,
·····"Sizes":·[100,·90,·80,·70,·60,·50,·40,·30,·20,·10],
·····"Methods":·["HINT","MinError",·"C4.5"],
·····"MEstimates":·[·1·],
·····"RandomSeed":·123456,
·····"Output":·"D:/DexLearning/Results/Experiments"
···},
```

*Figure 13: ExperimentParameters: Default settings.*

Running the example script `./Examples/Car_Rev/Experiment_CarRev0.json` creates a `.csv` file on the output directory `./Examples/Car_Rev/Experiment/`, which summarizes the elapsed time (in milliseconds), achieved classification accuracies, and bit size of the generated models in all experiments. With some postprocessing, charts can be obtained from this data, such as the learning curve in Figure 14.

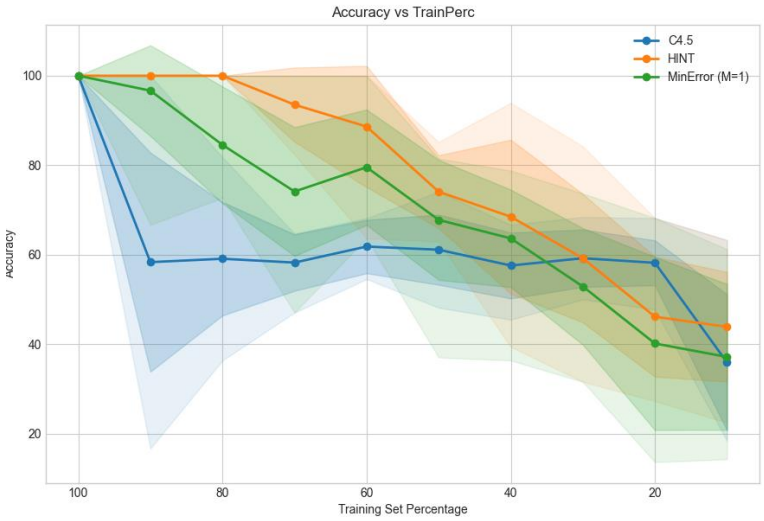


*Figure 14: Learning curve experimenting with the Car_Rev0 dataset.*

# 7 DexRevise: Data-Based Revision of DEX Models

DexRevise is another central program of the DexLearning software package, aimed at the revision of DEX models from data. Given a DEX model and a dataset, DexRevise adapts the model to match the dataset as much as possible. In contrast with DexLearn, which creates both the model structure and decision rules from a dataset, DexRevise only modifies decision tables, while leaving model structure intact. The quality of model revision is measured in terms of classification accuracy, assessed either on the same dataset that has been used in revision (referred to as the "training" or "learning" dataset), or on a separate "validation" dataset.

DexRevise is invoked by the command:

```
DexRevise model.dxi train_data.tab [validation_data.tab] [/options ...]
```

Here, `model.dxi` is an existing DEX model to be revised. This file is accessed in a read-only mode and is not modified by the program; revised models are written to other `.dxi` files. The two `.tab` files are datasets, represented in the same tab-delimited way as described in section 4.1. If only `train_data.tab` is specified, it is used for both model revision and validation. Otherwise, `train_data.tab` is used for revision, while revised models are validated on `validation_data.tab`.

Options, which can be abbreviated, are:

- `/progressive`: Updates best future accuracy after adding each new model; considers less future models, but may miss some promising revisions.
- `/quick`: Consider only changes with the highest weight; may miss some revisions and terminate too soon.
- `/median`: Consider only changes with median or higher weights.
- `/average`: Consider only changes with above average weights.
- `/monotone`: Disregard non-monotone revisions.
- `/saveiterations`: Save the best model found after each iteration of the algorithm.

Options `/quick`, `/median` and `/average` are mutually exclusive; only the last option applies.

The meaning of these options is better explained in connection with the model revision algorithm, presented in the next section.

## 7.1 Model Revision Algorithm

The algorithm implemented in DexRevise in based on the work of Žnidaršič (2007), Žnidaršič and Bohanec (2004), and Žnidaršič et al. (2009), however it introduces some novel elements. The distinction between the learning and validation datasets is already novel.

Žnidaršič et al.'s algorithms consider only one input data item at a time, which makes them sensitive to the incoming order of data items. In contrast, DexRevise first traverses the whole learning dataset (Algorithm 3), identifying all data items that disagree with the model. For each disagreement found, DexRevise recursively traverses the model structure and identifies decision rules that can be changed so as to establish the agreement between the data item and the model (method `FIND_CHANGES`, not elaborated further). In general, multiple such changes are possible for each data item. At this stage, no changes are made to the model; they are just recorded. A cumulative *weight* is recorded for each change, counting the number of data items that indicated that particular change. Each change is uniquely identified by: decision table, decision rule index, initial outcome and changed outcome. Changes with the highest weights are most promising for actual model modifications.

```
function CHANGES(model M, training dataset T):
    changes = ∅
    for each data item d from T:
        o = output(d)  // outcome according to data item d

        // compare o with model evaluation M(d)
        if o ≠ M(d):

            // find changes that modify M so that M(d) = o
            found = FIND_CHANGES(M, o)

            // record changes found
            for each c in found:
                // add c to changes, if not there, and increase c's "weight"
                RECORD_CHANGE(changes, c)
```

*Algorithm 3: Identifying candidate changes of model M from dataset T.*

Another novelty of DexRevise is that it runs iteratively, traversing the training dataset multiple times (Algorithm 4). In each iteration, it keeps the set of most promising models, and tries to improve their classification accuracy by applying candidate changes, one at a time. Classification accuracy of revised models is assessed either on the validation dataset `V`, if it has been specified as an argument to DexRevise, or on the training dataset (`V=T`) otherwise. The algorithm stops when no further improvement of models is possible.

```
function REVISION(model M, training dataset T, validation dataset V):
    bestM = M       // best model found so far
    bestQ = CA(M,V) // best classification accuracy measured so far

    models = { M }  // set of models considered in each iteration
    while models ≠ ∅:
        nextM = ∅     // candidate models for the next iteration
        iterQ = bestQ // accuracy threshold for the current iteration
        for each m in models:
            C = CHANGES(m,T)
            for each c in C:
               if ChangeAllowed(m,c):
                   newM = ApplyChange(m,c)
                   newQ = (newM,V)
                   if newQ > bestQ: // newM is the best model so far
                       bestM = ewM
                       bestQ = newQ
                   if newQ > iterQ: // newM is a candidate for the next iteration
                       if /progressive: iterQ = newQ; nextM = ∅
                       nextM = nextM ∪{ newM }
        models = nextM
```

*Algorithm 4: The top-level model revision algorithm.*

The main loop of Algorithm 4 checks the current model candidates `m ∈ models` and keeps track of the best model `bestM` with best quality `bestQ` achieved that far, while also preparing candidate models `nextM` for the next iteration.

By default, the set of `models` in each iteration is kept as large as possible. After setting `iterQ`, the accuracy threshold for the current iteration, all models `newM` that improve over that measure are included for consideration in the next iteration. This is to ensure that revision paths to obtaining the best final solution are thoroughly investigated. In some cases, however, this may lead to a very extensive search and long execution times.

To speed up the process, the progressive subvariant of the algorithm (activated using the `/progressive` program option) raises the threshold `iterQ` for the current iteration whenever a better

model is found, and discards models already found in that iteration. This generally results in fewer models being included in the next iteration, and substantially shortens execution times; however, it may miss some good model candidates.

The condition `ChangeAllowed(m, c)` determines whether the change `c` is allowed on model `m`. The main condition is that `c` must not revert `m` to one of the already investigated models (actually, behind the scene, DexRevise keeps track of already investigated revised models). Additionally, `ChangeAllowed()` considers conditions that constitute the subvariants of the algorithm, determined at invocation using these program options:

- `/quick`: consider only changes `c in C` that have the maximum weight in recorded changes `C`;
- `/median`: Consider only changes whose wights are better than or equal to the median weight in `C`;
- `/average`: Consider only changes whose wights are better than or equal to the average weight in `C`;
- `/monotone`: change `c` is allowed only if it preserves the monotonicity of changed rule with respect to all comparable rules in the corresponding decision table.

## 7.2 Example

As an example, let us rerun the revision experiment presented in Žnidaršič and Bohanec (2004). There, the base model was the one presented in Figures 1 and 2. The training dataset was generated from a modified model in which three, one and three decision rules of decision tables “costs”, “safety” and “car”, respectively, were modified. The task is to revise the base model and see if the revision algorithm reconstructs the same changes. The input data is prepared on the `./Examples/` folder:

- `Car_Rev0.dxi`: base model
- `Car_Rev1.tab`: a learning dataset with introduced changes

Running

```
DexRevise Car_Rev0.dxi Car_Rev1.tab
```

produces, after 7 iterations, the revised model `Car_Rev0_Revised.dxi` that exactly matches the learning dataset (classification accuracy is 100%). The revised model file, under section `DESCRIPTION`, contains a short revision log, including the list of changes made:

```
Revised model 165 from D:/DexLearning/Examples/Car_Rev0.dxi
Data D:/DexLearning/Examples/Car_Rev1.tab
Parameters: All, NoProgressive, NoMonotone
Initial accuracy: 57.4074%
1. "car" W4 2: 2 to 1
2. "costs" W3 2: 1 to 0
3. "car" W3 8: 0 to 1
4. "costs" W3 6: 1 to 2
5. "safety" W4 3: 0 to 1
6. "car" W2 5: 1 to 2
7. "costs" W4 3: 0 to 1
Accuracy: 100.0000%
Models checked: 1527
Runtime: 354.1659 ms
```

Each individual change is described by the sequence index, attribute name, weight (the number of data items that indicated the change), decision rule index (0-based), and changed ordinal value (from – to, also 0-based) of that decision rule’s outcome.

# 8 TestModel: Assess Classification Accuracy of a DEX Model

TestModel is a simple supplementary program, aimed at testing the classification accuracy of a given DEX model on one or more datasets. The command is:

```
TestModel model.dxi data.tab [data.tab ...]
```

Here, `model.dxi` is required and must be represented in a standard `.dxi` file (written by DEXi, DEXiWin or any of the DexLearning programs). The `data.tab` files must be in a standard DexLearning tab-delimited format containing one heading row (section 4.1). The input and output attributes are determined from the model, and must exactly match the names present in the data header.

The program loads the model and evaluates all the data items in all the data files, comparing the output values in the dataset with those obtained by model evaluation. Classification accuracy is defined as the proportion of data items where these two values match.

# 9 SplitData: Split a Dataset to Subsets

SplitData is another simple DexLearning program that splits given datasets into several subsets. The command is

```
SplitData data.tab [data.tab ...] [/option ...]
```

The options are:

- `/Fn`: Make `n` folds. The data is split proportionally to $\frac{n-1}{n}$ items for training and $\frac{1}{n}$ items for testing. Testing subsets are guaranteed to be partitions of the original dataset.
- `/Pn`: Split the dataset to $n$ % training and $(100 - n)$ % test data.
- `/Hn`: Indicates the number of header lines in `data.tab`.

# 10 References


Bohanec, M., Zupan, B. (2004): A function-decomposition method for development of hierarchical multi-attribute decision models. *Decision Support Systems* 36(3), 215–233. https://doi.org/10.1016/S0167-9236(02)00148-3.

Bohanec, M., Delibašić, B. (2015): Data-mining and expert models for predicting injury risk in ski resorts. In B. Delibašić, J. E. Hernández, J. Papathanasiou, F. Dargam, P. Zaraté, R. Ribeiro, S. Liu, & I. Linden (Eds.), *Decision Support Systems V – Big Data Analytics for Decision Making*, 46–60. Springer International Publishing. https://doi.org/10.1007/978-3-319-18533-0_5.

Bohanec, M. (2022): DEX (Decision EXpert): A qualitative hierarchical multi-criteria method. *Multiple Criteria Decision Making* (ed. Kulkarni, A.J.), Studies in Systems, Decision and Control 407, Singapore: Springer, 39–78. https://doi.org/10.1007/978-981-16-7414-3

Bohanec, M. (2024): DEXiWin: DEX Decision Modeling Software, User's Manual, Version 1.2. Institut Jožef Stefan, Delovno poročilo IJS DP-14747, 2024. https://kt.ijs.si/MarkoBohanec/pub/2024_DP14747_DEXiWin.pdf

Bohanec, M. (2026): *Experimental Evaluation of DexLearning Software for Machine Learning and Revision of DEX Models*. Report IJS DP-15586, Ljubljana: Jožef Stefan Institute. https://kt.ijs.si/MarkoBohanec/pub/2026_DP15586_DexLearning-Experiments.pdf.

Brélaz, D. (1979): New methods to color the vertices of a graph. *Communications of the ACM* 22 (4), 251–256. doi:10.1145/359094.359101. https://doi.org/10.1145/359094.359101.

Buede, D.M. (2013): Decision Making and Decision Analysis. In: Gass, S.I., Fu, M.C. (eds) *Encyclopedia of Operations Research and Management Science*. Boston: Springer. https://doi.org/10.1007/978-1-4419-1153-7_217.

Cestnik, B. (1990): Estimating probabilities. In: Carlucci Aiello, L. (ed.). *ECAI 90*. London: Pitman, 147–149. ISBN 0-273-08822-X.

Greco, S., Ehrgott, M., Figueira, J. (Eds.) (2016): *Multi Criteria Decision Analysis: State of the art Surveys*. New York: Springer. http://dx.doi.org/10.1007/978-1-4939-3094-4.

Radovanović, S., Bohanec, M., Delibašić, B. (2023): Extracting decision models for ski injury prediction from data. *International Transactions in Operational Research* 30(6), 3429–3454. https://doi.org/10.1111/itor.13246.

Zupan, B. (1997): *Machine learning based on function decomposition*. Doctoral dissertation. University of Ljubljana, Faculty of Computer and Information Science.

Zupan, B., Bohanec, M., Demšar, J., Bratko, I. (1999): Learning by discovering concept hierarchies. *Artificial Intelligence* 109(1–2), 211–242. https://doi.org/10.1016/S0004-3702(99)00008-9.

Žnidaršič, M. (2007): *Revizija verjetnostnih večparametrskih hierarhičnih modelov*. Doctoral dissertation. University of Ljubljana, Faculty of Computer and Information Science.

Žnidaršič, M., Bohanec, M. (2004): Revision of qualitative multi-attribute decision models. In R. Meredith, G. Shanks, D. Arnott, S. Carlson (Eds.), *DSS2004: IFIP International Conference on Decision Support Systems: Decision Support in an Uncertain and Complex World*, 881–888.

Žnidaršič, M., Bohanec, M., Zupan, B. (2009): Data-driven revision of decision models. In J. Wang (Ed.): *Encyclopedia of Data Warehousing and Mining*, 2nd Edition, 617–623. IGI Global. https://doi.org/10.4018/978-1-60566-010-3.

# Appendix 1: M-Estimate

The *m-estimate* (Cestnik, 1990s) is a probability estimation technique used primarily in machine learning, especially in decision trees and naive Bayes classifiers. It is designed to address the *zero-frequency problem*, the issue where a particular event (e.g., a class label given a feature value) never appears in the training data, leading to a probability estimate of zero. A zero probability is dangerous because it can entirely wipe out all other evidence when multiplying probabilities (e.g., in naive Bayes), or it can prevent a branch from being selected in a decision tree.

The m-estimate solves this by *smoothing* the empirical frequency toward a prior probability, using a parameter $m$ that controls the strength of this smoothing. The formula estimates the conditional probability $P(y \mid x)$ (the probability of class $y$ given that feature $x$ is observed) as:

$$P(y|x) = \frac{n_{xy} + m \cdot p(x)}{n_y + m}$$

Where:

- $n_{xy}$: the number of training examples that have both feature value $x$ and class $y$.
- $n_x$: the total number of training examples that have feature value $x$.
- $p(y)$: the prior probability of class $y$ (often estimated from the training data as the fraction of examples belonging to class $y$.
- $m$: a constant (the "$m$" parameter) that determines the weight given to the prior. It can be interpreted as the size of an imaginary "virtual" sample that is added to the real data.

The term $m \cdot p(y)$ acts as virtual counts added to the actual counts. Think of it as adding $m$ imaginary examples to the dataset, of which $m \cdot p(y)$ class $y$.

The denominator $n_y + m$ is the total effective sample size (real + virtual).

When $n_x = 0$ (the feature value never appears), the estimate collapses to the prior $p(y)$, avoiding zero.

As $n_x$ grows large (with many real observations), the influence of the virtual counts shrinks, and the estimate converges to the raw observed frequency $\frac{n_{xy}}{n_x}$.

Recommendations for choosing “m” are:

$m = 0$: gives the raw maximum likelihood estimate (no smoothing, risks zeros).

$m = 1$ is known as “Laplace smoothing” (add-one smoothing), which assumes a uniform prior.

$m > 1$ applies stronger smoothing, which is useful when the prior is highly reliable or when the data is very sparse.

A common heuristic, also proposed by Cestnik (1990), is to set $m$ proportional to the number of possible values of the feature or class, or to use $m = k$ where $k$ is the number of classes (for conditional probabilities). In practice, $m$ is often treated as a tunable hyperparameter, selected via cross-validation (see Section 6).

# Appendix 2: DexLearn Commands and Parameters

```
help

exit

directories {
  "Input": null,
  "Data": null,
  "Scales": null,
  "Output": null,
  "Logs": null,
  "CreateOutput": true,
  "ClearOutput": false
}

logs {
  "Verbosity": 1,
  "LogStatistics": true,
  "LogStructure": false,
  "LogFile": null
}

scales {
  "FileName": null,
  "ReadScales": false,
  "ReadIfExist": true,
  "WriteScales": false,
  "WriteIfNotExist": false,
  "WriteAttributesOnly": true
}

saving {
  "SaveDexi": true,
  "SaveGml": true,
  "SaveSolutions": true,
  "SaveOnlyBest": true,
  "SaveDecisionTree": true,
  "Delimiter": "\t",
  "Decimal": ",",
  "Digits": -1,
  "FloatFormat": "0.00"
}

loaddata {
  "FileName": null,
  "Quote": null,
  "Separators": null
}

savedata {
  "StrData": null,
  "IntData": null,
  "IntTable": null,
  "ProbTable": null,
  "Quote": null,
  "Separator": null,
  "AsText": true,
  "Invariant": true
}
```

```
preparedata {
  "FileName": null,
  "LoadStringData": true,
  "MakeIntData": true,
  "HandleScales": {
    "FileName": null,
    "ReadScales": false,
    "ReadIfExist": true,
    "WriteScales": false,
    "WriteIfNotExist": false,
    "WriteAttributesOnly": true
  },
  "Scales": null,
  "IncludeAttributes": null,
  "ExcludeAttributes": null,
  "OutputAttribute": null,
  "MakeIntTable": true,
  "MakeProbTable": true,
  "MakeFlatModel": true,
  "HandleModel": {
    "SaveDexi": true,
    "SaveGml": true,
    "SaveSolutions": true,
    "SaveOnlyBest": true,
    "SaveDecisionTree": true,
    "Delimiter": "\t",
    "Decimal": ",",
    "Digits": -1,
    "FloatFormat": "0.00"
  }
}

hint {
  "Name": null,
  "ProcessInputs": true,
  "CanDeleteInputs": true,
  "MaxSolutions": -1,
  "MaxSave": -1,
  "MaxPerNode": -1,
  "MaxOutValues": -1,
  "BestMinSelect": -1,
  "BestTolerance": 0,
  "MEstimate": 2.0
}

minerror {
  "Name": null,
  "ProcessInputs": true,
  "CanDeleteInputs": true,
  "MaxSolutions": -1,
  "MaxSave": -1,
  "MaxPerNode": -1,
  "MaxOutValues": -1,
  "BestMinSelect": -1,
  "BestTolerance": 0,
  "MEstimate": 2.0
}

supervised {
  "Name": null,
  "ProcessInputs": true,
  "CanDeleteInputs": true,
  "Decompose": null
}

c45 {
  "Name": null
}
```

```
id3 {
  "Name": null
}

probabilities {
  "Name": null,
  "Measures": true,
  "Pairs": true
}
```