MCP: Learning Propositional Formulas from Binarized Data

Miki Hermann
LIX, CNRS, École Polytechnique,
Institut Polytechnique de Paris,
Palaiseau, France
hermann@lix.polytechnique.fr

Gernot Salzer
Institut for Logic and Computation
Technische Universität Wien
Vienna, Austria
gernot.salzer@tuwien.ac.at

Abstract

Experimental data is often given as bit vectors, with vectors corresponding to observations, and coordinates to attributes, with a bit being true if the corresponding attribute was observed. Observations are usually grouped, e.g. into positive and negative samples. Among the essential tasks on such data, we have compression, the construction of classifiers for assigning new data, and information extraction.

Our system, MCP, approaches these tasks by propositional logic. For each group of observations, MCP constructs a (usually small) conjunctive formula that is true for the observations of the group, and false for the others. Depending on the settings, the formula consists of Horn, dual-Horn, bijunctive or general clauses. To reduce its size, only relevant subsets of the attributes are considered. The formula is a (lossy) representation of the original data and generalizes the observations, as it is usually satisfied by more bit vectors than just the observations. It thus may serve as a classifier for new data. Moreover, (dual-)Horn clauses, when read as if-then rules, make dependencies between attributes explicit. They can be regarded as an explanation for classification decisions.

1 Introduction and Related Work

Since several years, computer science applications are challenged by large quantities of data, commonly referred to as *Big Data*, that needs to be interpreted, captured, treated, and transformed. There exist several approaches to cope with this challenge, mainly from the field of Artificial Intelligence. One of these approaches is the *Logical Analysis of Data*. This document presents a tool called MCP, performing logical analysis of big data, producing a propositional formula. The basic idea behind this tool programmed in C++ is to describe a very large data set by a propositional formula.

Logical Analysis of Data is a part of Machine Learning, which has been developed by Hammer and his colleagues [5, 10]. There also exists another approach through mechanized hypothesis formation, the GUHA Project developed in Prague by Hájek and his colleagues [13, 15].

2 Preliminaries

We recall the main structures of Boolean algebra. A *literal* is either a variable, called positive literal, or its negation, called negative literal. A *clause* is a disjunction of literals. A *formula* in *conjunctive normal form* is a conjunction of clauses. A *Horn* clause is a clause with at most one positive literal. A

dual Horn clause is a clause with at most one negative literal. A *bijunctive* clause is a clause consisting of at most two literals. An *affine* clause is a linear equation of the form $x_1 + \cdots + x_k = b$, where x_i are variables, $+$ is the exclusive-or operator, and $b \in \{0, 1\}$ is a Boolean value. A Horn, dual Horn, bijunctive, or affine formula is a conjunction of only Horn, dual Horn, bijunctive, or affine clauses, respectively.

We will work with vectors, also called tuples, of finite arity over a domain D. This domain is either Boolean, i.e., $D = \{0, 1\}$, or finite, i.e., $|D| = n$ for some natural number $n \geq 2$. Vectors $(a_1, \ldots, a_k)$ of arity k will be shortened to $a_1 \cdots a_k$ when the elements a_i are clear.

Let $\boldsymbol{a} = a_1 \cdots a_k$, $\boldsymbol{b} = b_1 \cdots b_k$, and $\boldsymbol{c} = c_1 \cdots c_k$ be Boolean vectors of the same arity k. There exist different closures of these Boolean vectors.

- *Horn closure* of $\boldsymbol{a}$ and $\boldsymbol{b}$ is the vector $\boldsymbol{d} = d_1 \cdots d_k$, such that $d_i = a_i \wedge b_i$;
- *Dual Horn closure* of $\boldsymbol{a}$ and $\boldsymbol{b}$ is the vector $\boldsymbol{d} = d_1 \cdots d_k$, such that $d_i = a_i \vee b_i$;
- *Bijunctive closure* of $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ is the vector $\boldsymbol{d} = d_1 \cdots d_k$, such that $c_i = \mathrm{maj}(a_i, b_i, c_i)$, where maj is the associative-commutative majority operator;
- *Affine closure* of $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ is the vector $\boldsymbol{d} = d_1 \cdots d_k$, such that $d_i = a_i + b_i + c_i$,where $+$ is the exclusive-or operator in the Boolean ring $\mathbb{Z}_2$;

all for each $i = 1, \ldots, k$. Given a set of Boolean vectors S of arity k, we denote by $\langle S \rangle_C$ the C-closure of S for C being Horn, dual Horn, bijunctive, or affine. A basic result from universal algebra states that for an arbitrary set of Boolean vectors S of the same arity k, the C-closure is the set of satisfying assignments for some C-formula φ [3, 4].

3 Running Example: Mushrooms

To illustrate the MCP system, we will use the Mushroom Data Set from the UCI Machine Learning Repository as a running example.[1] It contains 8124 records with 22 attributes each. Each record describes an instance of one of 23 species of mushrooms, categorized as *edible* or *poisonous*. Among the attributes we find e.g. *cap-shape*, *odor* or *habitat*. Each attribute may take a value from a finite set. The odor, for instance, may be one of *almond*, *fishy*, *foul* and six further odors. A record looks like

```
edible,convex,smooth,white,yes,almond,...,purple,several,woods
```

where the first field specifies the category and the subsequent ones the values of the 22 attributes in a fixed order; e.g., the sixth field specifies the odor as *almond*.

To process data like the mushrooms with the MCP system, it needs to be *binarized*. MCP offers a utility, `mcp-trans`, for transforming the data. The particular encoding has to be specified in a file that describes the mapping for every attribute and value. To ease the burden on the user, the utility `mcp-guess` takes the original data and generates a draft of this specification. So the typical workflow consists of first running `mcp-guess`, then checking and manually adjusting the generated specification, then running `mcp-trans` on the data, and finally feeding its binary output to the core tools of the MCP system. The next section takes a closer look at the functionality of the MCP core, while the utility programs are discussed in the subsequent section.

[1] https://archive.ics.uci.edu/ml/datasets/mushroom

4 Core of the MCP System

MCP consists of several modules. The core modules generate a propositional formula from given sets of binary tuples, according to the following specification.

Problem (MCP Problem)**.** Given two sets of Boolean vectors (tuples) of arity k over the Boolean domain $D = \{0, 1\}^k$, representing positive samples $T \subseteq D$ and negative samples $F \subseteq D$, **compute** a Horn, dual Horn, bijunctive, or general CNF formula φ, respectively, such that (1) $T \models \varphi$ and (2) for each $f \in F$, $f \not\models \varphi$.

There are several reasons why we focus on the aforementioned four subclasses of propositional formulas. Horn, dual Horn, bijunctive, and affine formulas are the four families of Boolean formulas, whose satisfiability problem can be decided in polynomial time. Moreover, Horn formulas are the foundation of logic-oriented programming, with Horn clauses being naturally interpreted as rules.

Example. Suppose we aim for a formula that characterizes the *edible* mushrooms in our running example. Then the records categorized as *edible* become the basis for the positive samples, T, and the records labeled as *poisonous* result in the negative samples, F. Since the attributes of the example are mostly non-binary, the dimension of the tuples will not equal the number of attributes 22, but will be larger and will depend on the chosen transformation. The one described later results in 111 Boolean attributes.

4.1 Strategies for Computing the Closure

An instance of the MCP problem is not solvable if some tuple occurs in the set of positive and negative samples at the same time, hence we require $T \cap F = \emptyset$. For a C-formula (with C being Horn, dual Horn, bijunctive or affine), the condition has to be strengthened to $\langle T \rangle_C \cap F = \emptyset$, as each tuple in the C-closure of T necessarily satisfies any C-formula for T.

With this constraint, instances of the MCP problem are solvable. In general, there is a range of solutions. While every formula solving the instance is satisfied by the tuples in T and falsified by those in F, its behavior for the remaining tuples is undetermined. MCP offers two **strategies**. The **large** strategy (the default) computes a formula satisified by a maximal number of tuples, while the **exact** strategy minimizes the number of satisfying tuples.

4.2 Minimal Section

For a set of tuples, S, let $S|_A$ denote the restriction of the tuples to the coordinates in A. Clearly, if two sets S_1 and S_2 have an empty intersection for a subset of the coordinates, then the original sets have an empty intersection, too: $S_1|_A \cap S_2|_A = \emptyset$ implies $S_1 \cap S_2 = \emptyset$. In general, each coordinate contributes a variable to the formula, hence minimizing A will reduce the number of different variables in the formula.

Finding an A of minimal cardinality such that $T|_A \cap F|_A = \emptyset$ (or $\langle T \rangle_C|_A \cap F|_A = \emptyset$) is an NP-complete problem. MCP implements several approximatons differing in the **direction** the coordinates are tried, skipping coordinates whose removal would render the problem unsolvable. The following directions are available:

begin: Prefer coordinates to the left (at the begin) of the tuples by removing coordinates from the right. This direction is the default.

end: Prefer coordinates to the right (at the end) of the tuples by removing coordinates from the left.

lowcard: Prefer coordinates with a lower Hamming weight, by removing coordinates with high Hamming weight.

highcard: Prefer coordinates with a higher Hamming weight, by removing coordinates with small Hamming weight.

random: Remove coordinates in random order.

nosect: Use all coordinates, do not remove any.

4.3 Effective Learning of Formulas

The MCP system learns *Horn* formulas by the following procedure. For each $f \in F$ it determines if $f \in \langle T \rangle_{\mathrm{Horn}}$ efficiently, without computing the Horn closure. Then it computes the minimal section of $\langle T \rangle_{\mathrm{Horn}}$ and F, followed by the computation of the corresponding Horn formula according to the chosen direction and strategy on the (approximate) minimal section of $\langle T \rangle_{\mathrm{Horn}}$ and F. It uses different algorithms for the strategies: that of Angluin *et al* [1] for the large strategy and another of Hébrard and Zanuttini [14] for the exact strategy.

Learning of *dual Horn* formulas is done very easily. MCP system first swaps the polarity of the Boolean vectors in T and F, producing the new sets T' and F', respectively. Then it computes the Horn formula φ' for T' and F', followed by swapping the polarity of literals in φ', producing the dual Horn formula φ.

There is no known possibility to determine if $f \in \langle T \rangle_{\mathrm{bijunctive}}$ for each $f \in F$ without computing the bijunctive closure $\langle T \rangle_{\mathrm{bijunctive}}$. Moreover, the bijunctive closure $\langle T \rangle_{\mathrm{bijunctive}}$ can be (and usually also is) very much time and space consuming. We adopted the following solution to produce *bijunctive* formulas by MCP system: It computes the minimal section using an intersection test, followed by application of the *Baker-Pixley Theorem* [2] (projection on every pair of coordinates), which implicitly guarantees the bijunctive closure.

Learning a *general CNF* formula presents several challenges. Its advantage is that We get a propositional formula in any case, provided that $T \cap F = \emptyset$. Its drawback is that the produced formula is usually very big. We adopted two different approaches in the MCP system, depending on the applied strategy. In case of large strategy, for each false element $f \in F$ the MCP system produces the unique clause c_f which falsifies f. The resulting formula φ is the conjunction of all falsification clauses c_f. In case of exact strategy, the MCP system uses an algorithm producing a CNF formula in time $O(|T|\, k^2)$, where k is the arity of vectors in T, using a Boolean restriction of a larger algorithm from [12].

Learning *affine* formulas reveals more from linear computer algebra than from logic, therefore we did not implement it in the MCP system for the time being. We may implement it in a further version if there is demand.

4.4 First Postprocessing: Redundancy Elimination

The inferred formula φ can contain redundant literals and clauses, which can and must be eliminated to produce the smallest possible formula. There are several stages, which can be applied for *redundancy elimination*, called **cooking** inside the MCP system, with the following options: **raw** performs no redundancy elimination, **bleu** performs unit resolution, **medium** performs unit resolution and clause subsumption, and finally **well done**, which is the default, performs unit resolution, clause subsumption, and implied clause removal. Moreover, the *exact* strategy includes a **primality** step, reducing the clauses by elimination of unnecessary literals, using an algorithm from [12].

4.5 Second Postprocessing: Set Cover

In case of the *large* strategy, we are mainly interested in producing a formula φ falsified by each tuple $f \in F$. However, the inferred formula φ may contain more clauses than necessary, even after full redundancy elimination. Our task is to keep the smallest number of clauses in φ which are necessary to guarantee falsification by all tuples $f \in F$. For this purpose in the MCP system, we use *Set Cover* where a clause $c \in \varphi$ covers a vector $f \in F$ if f falsifies c. Set Cover is a well-known NP-complete problem, therefore we use Johnson's approximation algorithm (see e.g. [11]), where the measure of a clause is the number of covered tuples. Of course, this approach is inapplicable for the *exact* strategy.

4.6 Input Format and Action Possibilities

The input file of the MCP system core, is a Boolean matrix, one Boolean vector per row. Each vector is prefixed by a string g, identifying a group to which the vector belongs. The MCP system core collects first the vectors from the input matrix and distributes them into the identified groups. Each input file starts with an indication line, containing two boolean values. If both values are equal to 0, the following lines are the rows of the Boolean matrix with leading group indicators. If the first value is equal to 1, the following line contains the variable names ordered by coordinates. If the second value is equal to 1, there is one more line of supplementary information before the matrix. However, this supplementary information is unused by the MCP system, but it is still maintained for compatibility reasons with data sets used in [8, 9].

Let G be the set of identified groups. The actual computation is determined by the **action**, which determines how the sets of positive samples T and negative samples F are constituted. The are two options, **one** and **all**.

The option *one* consecutively selects two groups $g, g' \in G$, determines the vectors belonging to the group g as the positive samples T and the vectors belonging to the group g' as the negative samples F, then starts the computation of the corresponding formula with minimal section. If there are n groups in the set G, this action proceeds with the computation of $n(n-1)$ formulas.

The option *all*, which is the default, consecutively selects a group $g \in G$, determines the vectors belonging to the group g as the positive samples T and all vectors belonging to any group from $G \setminus \{g\}$ as the negative samples F, then starts the computation of the corresponding formula with minimal section. For n groups in the set G, this action proceeds with the computation of n formulas.

4.7 Parallelization

For a set of n groups, the MCP system computes either n or $n(n-1)$ formulas. These computations are independent, therefore they can be performed in parallel. This is called *outer parallelism* in the MCP core.

In case of Horn closure of the positive samples T, the MCP core needs to determine if a given vector $f \in F$ from negative samples belongs to $\langle T \rangle_{\mathrm{Horn}}$, without computing the closure itself. This procedure is quite time consuming when the set T is quite large. It can be computed in parallel, each time taking only a determined chunk of T. This is called *inner parallelism* in the MCP core.

We adopted three types of parallelization within the MCP core: the **Message Passing Interface** (MPI) [16], the **POSIX threads** (pthreads) [7], and a **hybrid** version combining both. These parallelizations are effective only on very large input data sets. The MPI version is applied only for outer parallelism, the pthreads version to both, and in the hybrid version MPI is applied for outer parallelism and pthreads for inner parallelism.

4.8 Invocation

MCP core is called by one of the following commands and options:

sequential version:	**mcp-seq**	`-i` *input-file*	`-o` *output-file*
MPI version:	**mcp-mpi**	`-l` *formula-prefix*	`-c` *closure*
POSIX threads version:	**mcp-pthread**	`-d` *direction*	`-s` *strategy*
hybrid version:	**mcp-hybrid**	`--cook` *cooking*	`--setcover` *y/n*

Each of these core modules produces files *formula-prefix*_*g*`.log` containing the learned formula for each group g inside *input-file*. Consult the manual pages for more detailed information.

Example. Suppose we have transformed the data of our running example to 15 858 binary tuples of length 111, stored in a file `mushroom.bin`, as described at the end of section 5.1. Running the command

```
mcp-seq -i mushroom.bin -o mushroom.frm
```

we obtain, within a minute or so, Horn formulas for the edible as well as for the poisonous mushrooms. E.g., the edible mushrooms are characterized by 17 rules like the following ones:

$$\begin{aligned} &\mathit{cap\text{-}color} \neq \mathit{yellow} \ \vee \ \mathit{odor} \neq \mathit{foul} \\ &\mathit{cap\text{-}color} \neq \mathit{gray} \ \vee \ \mathit{odor} \neq \mathit{foul} \\ &\mathit{spore\text{-}print\text{-}color} = \mathit{white} \ \rightarrow \ \mathit{odor} = \mathit{none} \end{aligned}$$

The first two formulas specify that for an edible mushroom, its *odor* is either not *foul*, or its *cap-color* is different from *yellow* and *gray*. Moreover, if the *spore-print-color* is *white*, then an edible mushroom has no *odor* at all.

5 Prequel and Sequel Modules

5.1 Data Binarization

The core of the MCP system accepts only Boolean vectors. However, data are usually spanning much larger domains: finite, or infinite but countable, or uncountable. In the latter two cases, every very large finite data set contains only a finite subset of the domain, but it can be intractable due to the amount of data to be treated. The MCP system copes with this situation by *binarization*.

Binarization is the process of transforming data of any domain into binary vectors to make classifier algorithms, in our case the MCP system core, more efficient. Its advantage is that we obtain the possibility to treat any data by propositional formulas. Its drawback is a possible exponential explosion. Binarization concerns both, particular values, especially for finite domains, as well as intervals, usually used for infinite ones. MCP system adopts both approaches.

Binarization in the MCP system is a two-step procedure. The first step consists of scanning of the CSV file and generating a meta-file template. This step is performed by the command

mcp-guess `-i` *csv-file* `-o` *meta-template*

where it is implicitly assumed that the *csv-file* contains one data vector per line, the vector elements are separated by commas or semicolons or space or tabs, vector element can be quoted, missing elements are denoted by a question mark. The template generated by *mcp-guess* cannot be used directly by

the next module, but it must be manually adapted to a proper meta-file. This command just creates indications if the values of a given coordinate are Boolean, enumerated strings, enumerated integers, integers in a range, or floats in a range.

The second step of the binarization process is performed by the command

mcp-trans `-i` *data-file* `-m` *meta-file* `-o` *binarized-file* [`--pvt` *pivot-file*] [`-r` y/n]

which generates a *binarized-file*, ready to be treated by the MCP system core, from the original *data-file* using a *meta-file*. This meta file consists of transformation commands. Each transformation command describes the treatment of one attribute and has the following format:

identifier **=** *coordinate* **:** *indicator* **;** {**#** *comment*}

where # starts an optional comment stretching until end of line, the symbols = and **:** and **;** are syntactic sugar, *identifier* will become the name of the variable for the given attribute *coordinate* and the *indicator* has one of the following forms:

concept		identifier of the concept to be learned
pivot		identifiers for pivot values in prediction
bool	'[' $elem_0$ $elem_1$ ']'	boolean 2-element set
enum	'[' $elem_0$ ... $elem_\ell$ ']'	enumerated set of $\ell + 1$ elements
up	'[' $elem_0$ ... $elem_\ell$ ']'	enumerated set of increasing $\ell + 1$ elements
down	'[' $elem_0$ ... $elem_\ell$ ']'	enumerated set of decreasing $\ell + 1$ elements
int	*min max*	integers in the range between *min* and *max*
dj	n *min max*	interval [*min*, *max*) cut in n disjoint chunks
cp	[$\wedge$] $number_0$ ... $number_\ell$ [$]	intervals determined by checkpoints (The numbers must build an increasing sequence. Consecutive numbers a and b determine the interval $[a, b)$. The optional symbols $\wedge$ and $ stand for minimal and maximal infinity, respectively)
over	n *min max* ℓ	[*min*, *max*) cut in n chunks with overlaps of length ℓ
span	ℓ *min max*	[*min*, *max*) cut in disjoint chunks, each of length ℓ
warp	ℓ_0 *min max* ℓ_1	[*min*, *max*) cut in chunks of length ℓ_0, overlaps of length ℓ_1

The square brackets '[' and ']', written in quotation marks to distinguish them from optional parameter indications, are just syntactic sugar for a better orientation of the parser.

Some data vectors can contain missing values indicated by a quation mark. The default treatment by **mcp-trans** is their elimination. If however we wish to include these data rows with missing values, we need to generate for them *robust* extensions [6], where the question marks are replaced by all other data for this coordinate gathered from all other data rows. This is achieved by the optonal `-r` flag. The *robust* option is incompatible with the *pivot* option.

The optional *pivot-file* will contain data identifiers used for prediction (see the command **mcp-predict**). The concept and pivot indicators are exclusive. The concept indicator is used during the learning process on training data, whereas the pivot indicator is used during the checking process on testing data, when a prediction for these data has to be done. Usually, the pivot coordinate contains the identifiers, one per data item, for which a prediction is done by means of the **mcp-predict** command.

After the transformation and binarization, the produced binary vectors do not need to be unique. To avoid this situation, we can use the command

mcp-uniq `-i` *input-file* `-o` *output-file*

which eliminates row doublets. Essentially, this command acts as the Linux command *uniq*, but the rows do not need to be sorted and duplicate rows need not be consecutive.

Example. Our mushroom example uses non-binary attributes. Therefore, we run `mcp-guess` on the original data to draft a specification for the binarization process. The command

```
mcp-guess -i mushroom.data -o mushroom.spec
```

analyzses the values occurring for every attribute and generates a file starting as follows.

```
id0 = 0: bool [edible poisonous];
id1 = 1: enum string [bell conical convex flat knobbed sunken];
id2 = 2: enum string [fibrous grooves scaly smooth];
id3 = 3: enum string [brown buff cinnamon gray green pink ...
id4 = 4: bool [no yes];
id5 = 5: enum string [almond anise creosote fishy foul musty ...
```

We adapt this file by marking `id0` as the column that defines the category (the 'concept'), and by replacing the generic identifiers `id`x by mnemonic labels, to improve readability. If we can also store the names of the attributes in the file `mushroom.names` and run the command

```
mcp-guess -i mushroom.data -o mushroom.spec -n mushroom.names
```

with the `-n` option. The identifiers of the attributes are then read from that file and replace the identifiers `id0`, ..., `id5`. Moreover, we check the encodings proposed by `mcp-guess`. Attributes taking more than two different, unordered values are tagged as `enum`, which tells the binarization utility to use a separate propositional variable for every value. Attributes with just two values are marked as `bool`, which results in a single variable for both values. While saving on variables reduces the size of the problem, it may make the information harder to access and prevent MCP from constructing, for instance, a Horn formula. In our final specification, the lines above take the following form.

```
class =  0: concept;
cap-shape = 1: enum [bell conical convex flat knobbed sunken];
cap-surface = 2: enum [fibrous grooves scaly smooth];
cap-color = 3: enum [brown buff cinnamon gray green pink purple ...
bruises = 4: bool [no yes];
odor = 5: enum [almond anise creosote fishy foul musty ...
```

Now we binarize the original data with the command

```
mcp-trans -i mushroom.data -m mushroom.spec -o mushroom.bin
```

The binarized data, `mushroom.bin`, starts with the following lines.

```
1 0
cap-shape_5:cap-shape==sunken:cap-shape!=sunken  ...
edible 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 ...
poisonous 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 ...
```

The first line with `1 0` indicates that line 2 is a header and that the actual data starts in line 3. The quite verbose header consists of space-separated labels, one for each binary attribute. Each label starts with the name of the attribute (`cap-shape_5` in the first label above), followed by two expressions

separated by colons. The first expression, `cap-shape==sunken`, specifies that a value `1` for the attribute `cap-shape_5` means that the original attribute `cap-shape` had the value *sunken*, whereas the second expression, `cap-shape!=sunken` reminds us that `cap-shape_5` being `0` means that `cap-shape` had a value different from *sunken*. This information is used by MCP to enhance the readability of the generated formulas. From line 3 to the end of the file, each line contains a tuple of 111 binary values, which is prefixed by one of the strings `edible` or `poisonous` giving the category of the tuple. The total number of lines is actually almost twice as large as in the original data file. The increase is caused by unknown values (appearing as a question mark, `?`) in the data file. Records with such values can be either dropped, decreasing the number of tuples, or can be replaced by all possible values of the respective attribute. The latter option may lead to a significant increase of the data set.

5.2 Formula Evaluation and Classification Prediction

If we are interested only in the produced formula, then the output file generated by the MCP core contains the satisfied formulas for each group of Boolean vectors. However, if we want to evaluate the accuracy of the produced formula, we must proceed further. The first prerequisite for a possibility to check the accuracy of a formula, is to have two sets of vectors: one for learning the formula, the other for checking its accuracy. Either we have these two sets of vectors already from the beginning or we need to split the original set of Boolean vectors into the learning part and the checking part before running the MCP core on the learning part. The latter is performed by the command

mcp-split `-i` *input-file* `-l` *learn-file* `-c` *check-file* `-r` *ratio*

that splits uniformly at random the *input-file* into a *learn-file* and *check-file*, where *ratio* is the percentage of vectors from the *input-file* populating the *check-file*. If the options `-l` or `-c` are not explicitly stated, the software deduces the file identifiers from the base name of the *input-file* and adding the suffix `.lrn` or `.chk` to it, respectively. The *ratio* default is 10.

The accuracy of the formula for a given group g is checked by the command

mcp-check `-i` *check-file* `-l` *formula-file* `-o` *output-file*

where *formula-file* is the file *formula-prefix_g*`.log` produced by the MCP core. Its *output-file* reproduces the formula and reports the following statistical entities, measured on the vectors from *check-file*: true positives (tp), true negatives (tn), false positives (fp), false negatives (fn), sensitivity $(tp/(tp + fn))$, miss rate $(fn/(fn + tp))$, specificity $(tn/(tn + fp))$, and precision $(tp/(tp + fp))$. The optimal situation would be to have neither false positives nor false negatives. If, however, these values are non-zero, it can be either due to an insufficient cardinality of learning data, or a wrong binarization, or else the data itself are not precise.

Once a learning process has been done and corresponding formulas have been generated, a prediction for testing data is performed by the following command:

mcp-predict `-i` *input* `-o` *output* `-l` *formula-prefix* `-pdx` *prediction* [`-pdt` *pivot*]

where *input* is the testing file containing Boolean vectors without group identifiers for which the prediction will be done; *output* is the file which will contain the report of the prediction run; *formula-prefix* is the prefix for files containing formulas produced by MCP core and where the corresponding *formula-file* is supposed to have the name *formula-prefix_g.log* for some group G; the file *prediction* will contain the prediction results in the form

pivot-value, *group identifiers*

where *group-identifiers* is a list of groups (only one in the best case) separated by the sign +, for which the corresponding formula from *formula-prefix_g* is satisfied by that *pivot-value*; and finally the optional file *pivot* contains the corresponding pivot identifiers to identify the prediction results, one per line.

The difference between check files and test files is only the presence (in check files) or absence (for test files) of group identifiers for attribute data. This is also visible in the difference in the semantics of the last two commands. The command **mcp-check** checks the accuracy of the solution with respect to existing knowledge contained in the data, whereas the command **mcp-predict** synthetises the missing attribute from data by means of a previously learned formula.

We can transform a check file into a test file by means of the following command:

mcp-chk2tst `-i` *input-file* `-o` *output-file*

which esentially discards the group information from data.

6 System Distribution and Examples

The MCP system is available at the github.com/miki-hermann/mcp. Please, follow the instructions in `README.md` file at the root. It is indispensable to run the installation instructions described in that file to be able to run the MCP system properly.

The overall performance of the MCP system is very competitive, both in terms of time, as well as in terms of quality of the produced formulas. The performance of the system has been measured on a DELL computer with an Intel Core™ i7-9700 CPU @ 3.00GHz × 8 with 16GB of memory, running under Linux Fedora 38. All examples from [8, 9] run under one second.

We have been testing the MCP system on several examples from the UCI Machine Learning Repository (archive.ics.uci.edu/ml). All examples in the subdirectories are equipped with a `Makefile` simplifying the application of the MCP system on them. The directory *uci* contains the following treated examples:

abalone	identifying abalone with 27 rings;
accent	identifying several accents in spoken English language;
balance-scale	identifying psychological experiments balancing a scale;
balloons	a toy example, where specific formulas are required to be produced;
banknote	identifying forged and genuine banknotes;
bean	identifying grains of 7 different dry beans;
breast-cancer-wisconsin	identifying benign and malignant breast cancer cases in Wisconsin;
car	identifying very good cars;
divorce	predicting if a marriage will end up in a divorce according to an analysis of responses to psychological investigation;
forest-fire	predicting forest fires in July, August, and September;
iris	identifying three types of iris flowers;
monk	the well-know monk examples;
mushroom	identifying edible and poisonous mushrooms;
nursery	for admition of children into a nursery;

optdigits	for determining digits from optical reading;
rice	identifying two rice varieties: cammeo and osmancik;
shuttle	the shuttle example;
vote	identifying democrats and republicans in the House of Representatives according to the 1984 US Congressional Voting Records.

We would especially drive the readers attention to the *mushroom* example, which identifies the edible and poisonous mushrooms always with 100% accuracy. This illustrates very well the strength of the MCP system.

7 Concluding Remarks

The MCP system consists of more than 7000 lines of `C++` code, using only the standard library. Parallel execution requires installation of the MPI software. Future versions of MCP will include a web GUI to enhance usability, as well as support for finite domains [12] to obviate the need for data binarization.

References

[1] Angluin, D., Frazier, M., Pitt, L.: Learning conjunctions of Horn clauses. Machine Learning **9**(2-3), 147–164 (1992)

[2] Baker, K.A., Pixley, A.F.: Polynomial interpolation and the Chinese Remainder Theorem for algebraic systems. Mathematische Zeitschrift **143**(2), 165–174 (1975)

[3] Böhler, E., Creignou, N., Reith, S., Vollmer, H.: Playing with Boolean blocks, part I: Post's lattice with applications to complexity theory. SIGACT News **34**(4), 38–52 (2003)

[4] Böhler, E., Creignou, N., Reith, S., Vollmer, H.: Playing with Boolean blocks, part II: Constraint satisfaction problems. SIGACT News **35**(1), 22–35 (2004)

[5] Boros, E., Crama, Y., Hammer, P.L., Ibaraki, T., Kogan, A., Makino, K.: Logical analysis of data: classification with justification. Annals of Operations Research **188**(1), 33–61 (2011)

[6] Boros, E., Ibaraki, T., Makino, K.: Monotone extensions of Boolean data sets. In: Li, M., Maruoka, A. (eds.) Proceedings 8th International Conference on Algorithmic Learning Theory, (ALT '97), Sendai (Japan). Lecture Notes in Computer Science, vol. 1316, pp. 161–175. Springer (1997)

[7] Butenhof, D.R.: Programming with POSIX threads. Addison-Wesley (1997)

[8] Chambon, A., Boureau, T., Lardeux, F., Saubion, F.: Logical characterization of groups of data: a comparative study. Applied Intelligence **48**(8), 2284–2303 (2018)

[9] Chambon, A., Lardeux, F., Saubion, F., Boureau, T.: Computing sets of patterns for logical analysis of data. Tech. rep., Université d'Angers (2017)

[10] Crama, Y., Hammer, P.L.: Boolean Functions - Theory, Algorithms, and Applications, Encyclopedia of Mathematics and its Applications, vol. 142. Cambridge University Press (2011)

[11] Garey, M.R., Johnson, D.S.: Computers and intractability: A guide to the theory of NP-completeness. W.H. Freeman and Co (1979)

[12] Gil, A., Hermann, M., Salzer, G., Zanuttini, B.: Efficient algorithms for constraint description problems over finite totally ordered domains. SIAM Journal on Computing **38**(3), 922–945 (2008)

[13] Hájek, P., Holeňa, M., Rauch, J.: The GUHA method and its meaning for data mining. Journal of Computer and System Sciences **76**(1), 34–48 (2010)

[14] Hébrard, J.J., Zanuttini, B.: An efficient algorithm for Horn description. Information Processing Letters **88**(4), 177–182 (2003)

[15] Hájek, P., Havránek, T.: Mechanizing Hypothesis Formation. Springer (1978)

[16] Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The complete reference. MIT Press (1995)