MCP: Capturing Big Data by Satisfiability* (Tool Description)

Miki Hermann[1] and Gernot Salzer[2]

[1] LIX, CNRS, École Polytechnique, Institut Polytechnique de Paris, 91120 Palaiseau, France. hermann@lix.polytechnique.fr

[2] 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.

Keywords: data classification · bit vectors · information extraction · explainable AI · machine learning

1 Introduction and Related Work

Since several years, computer science applications are challenged by very large amounts of data, commonly referred to as *Big Data*, that must be understood, 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 *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 [5,9]. There also exist another approach through mechanized hypothesis formation, the GUHA Project developed in Prague by Hájek and his colleagues [12,14].

* Partially developed within the ACCA Project.

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 clause. 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 Core of the MCP System

MCP has a modular architecture. It is composed of several modules, which perform designated tasks. The core of the system is composed of different variants of the module generating a propositional formula from sets of binary tuples. The main task of the MCP system, solved by its core modules, is defined as follows:

Problem 1 (MCP Problem). Given two sets of Boolean vectors (tuples) of arity k over the Boolean domain $D = \{0, 1\}^k$, representing positive examples $T \subseteq D$ and negative examples $F \subseteq D$, **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 subcases 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. Horn formulas represent a theoretical background of Prolog

programs. Horn clauses (implications of the form antecedent $\rightarrow$ consequent) represent a natural explanation pattern — easy to explain also to a non-expert in computer science or logic. The posed problem is an instance of PAC-learning.

There are several caveats for this problem we must deal with, namely what to do if (1) $T \cap F \neq \emptyset$, (2) $\langle T \rangle_C \cap F \neq \emptyset$, (3) $\{0,1\}^k \setminus (\langle T \rangle_C \cup F) \neq \emptyset$. There is no solution for the first two cases, since we cannot satisfy the basic requirements of the MCP Problem. The third caveat is solved by means of **strategy**.

3.1 Strategies for Computing the Closure

Depending on how we want to treat the vectors absent from $\langle T \rangle_C \cup F$, we have two available *strategies*, depending on whether we consider the largest or the smallest closure of the set of positive examples T.

The **large** strategy, which is the default, computes the *largest* C-closure containing T that does not intersect with F. The computed formula φ satisfies the condition $f \not\models \varphi$ for each $f \in F$. The **exact** strategy computes the *smallest* C-closure containing T. It satisfies the conditions $\langle T \rangle_C \models \varphi$ and $f \not\models \varphi$ for each $f \in \{0,1\}^k \setminus \langle T \rangle_C$.

3.2 Minimal Section

We want to keep the sets $\langle T \rangle_C$ and F disjunct on the smallest number of coordinates, to keep the number of variables of the produced formula as small as possible. Given the sets of vectors $\langle T \rangle_C$ and F or T and F as binary codes, composed of codewords over Boolean domain, we want to compute their *minimal section*, i.e. their restriction to a maximal set of coordinates A, such that $\langle T \rangle_C|_A \cap F|_A = \emptyset$ or $T|_A \cap F|_A = \emptyset$. Computing the optimal minimal section is an NP-complete problem. Therefore we adopt several approximation approaches by means of **direction**, always skipping coordinates whose removal would render the problem unsolvable. Following directions are available:

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

end: Prefer coordinates to the right (at the end) of the codewords 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: Removing coordinates in random order.

There also exists the **nosect** option, where no minimal section is computed and all coordinates considered.

3.3 Effective Learning of Formulas

The MCP system learns *Horn* formulas by the following procedure. For each $f \in F$ it determines if $f \in \langle T \rangle_{\text{Horn}}$ efficiently, without computing the Horn closure. Then it computes the minimal section of $\langle T \rangle_{\text{Horn}}$ and F, followed by the computation of the corresponding Horn formula according to the chosen direction and strategy on the (approximate) minimal section of $\langle T \rangle_{\text{Horn}}$ 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 [13] 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 $f \in \langle T \rangle_{\text{bijunctive}}$ for each $f \in F$ without computing the bijunctive closure $\langle T \rangle_{\text{bijunctive}}$. Moreover, the bijunctive closure $\langle T \rangle_{\text{bijunctive}}$ 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 [11].

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.

3.4 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 [11].

3.5 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. [10]), where the measure of a clause is the number of covered tuples. Of course, this approach is inapplicable for the *exact* strategy.

3.6 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 values is equal to 1, the 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 [7,8].

Let G be the set of identified groups. The actual computation is determined by the **action**, which determines how the sets of positive examples T and negative examples 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 examples T and the vectors belonging to the group g' as the negative examples 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 examples T and all vectors belonging to any group from $G \setminus \{g\}$ as the negative examples 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.

3.7 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 examples T, the MCP core needs to determine if a given vector $f \in F$ from negative examples 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) [15], the **POSIX threads** (pthreads) [6], 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.

3.8 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.

4 Prequel and Sequel Modules

4.1 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 do to the amount of date 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*

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 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 *coordinate* and the *indicator* has one of the following forms:

ident		group identifier
bool	$[elem_0\ elem_1]$	boolean 2-element set
enum	$[elem_0 \ldots elem_\ell]$	enumerated set of $\ell+1$ elements
up	$[elem_0 \ldots elem_\ell]$	enumerated set of increasing $\ell+1$ elements
down	$[elem_0 \ldots elem_\ell]$	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
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 ℓ_1

4.2 Formula Evaluation

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 *out-put-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.

5 System Distribution and Examples

The MCP system is available at the `github.com/miki-hermann/mcp`. We also provided a companion `mcp.tar.gz` to this submission, containing the sources, the man pages, and the examples. Run the command "`tar xzf mcp.tar.gz`" to unpack the distribution and 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 33. All examples from [7,8] 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 by 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, *balance-scale* identifying psychological experiments balancing a scale, *balloons* — a toy example, where specific formulas are required to be produced, *breast-cancer-wisconsin* identifying benign and malignant breast cancer cases in Wisconsin, *car* identifying very good cars, *forest-fire* predicting forest fires in July, August, and September, *iris* identifying three types of iris flowers, *mushroom* identifying edible and poisonous mushrooms, and *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.

6 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 [11] 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. Butenhof, D.R.: Programming with POSIX threads. Addison-Wesley (1997)
7. Chambon, A., Boureau, T., Lardeux, F., Saubion, F.: Logical characterization of groups of data: a comparative study. Applied Intelligence **48**(8), 2284–2303 (2018)
8. Chambon, A., Lardeux, F., Saubion, F., Boureau, T.: Computing sets of patterns for logical analysis of data. Tech. rep., Université d'Angers (2017)
9. Crama, Y., Hammer, P.L.: Boolean Functions - Theory, Algorithms, and Applications, Encyclopedia of Mathematics and its Applications, vol. 142. Cambridge University Press (2011)
10. Garey, M.R., Johnson, D.S.: Computers and intractability: A guide to the theory of NP-completeness. W.H. Freeman and Co (1979)
11. 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)
12. Hájek, P., Holena, M., Rauch, J.: The GUHA method and its meaning for data mining. Journal of Computer and System Sciences **76**(1), 34–48 (2010)
13. Hébrard, J.J., Zanuttini, B.: An efficient algorithm for Horn description. Information Processing Letters **88**(4), 177–182 (2003)
14. Hájek, P., Havránek, T.: Mechanizing Hypothesis Formation. Springer (1978)
15. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The complete reference. MIT Press (1995)