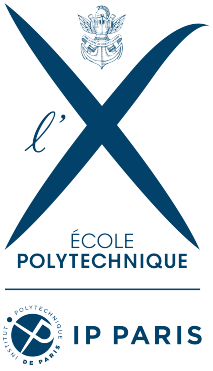

École Polytechnique

BACHELOR THESIS IN COMPUTER SCIENCE

Design and Implementation of a Statistical Module For the MCP System

Author:

Oskar Dąbkowski, École Polytechnique

Advisor:

Nicolas Hermann, LIX

Academic year 2023/2024

Abstract

To face the negative drawbacks of neural networks, various machine learning methods are being introduced. The MCP project was created to provide solutions to the lack of transparency and inexplicability of the neural networks. MCP has quite a different approach to them as it uses logical analysis to model data's behaviour. This tool is still under development, so it suffers from other drawbacks like a lack of data cleaning. As MCP has a modular construction, this problem could be solved by adding a module performing data clearing. This document shows the design and implementation of such a module which allows the user to clean the data. The module interprets the commands provided by the user by compilation methods and executes them on the data file. It allows the cleaning of the outliers from the data which makes it possible for the rest of the MCP to achieve greater accuracy in the creation of a conjunction normal form formula describing the data.

Contents

1 Introduction

Recently, computer science has faced a rapidly increasing amount of data which requires capturing, treating and interpreting, commonly known as Big Data [2]. During the last years, there has been a great development in the area of Machine Learning, which tries to tackle the issues related to Big Data. In particular, neural networks have lately played a significant role in that improvement, to which we owe progress in such important areas as computer vision or natural language processing.

Despite this progress, neural networks face two major drawbacks. Firstly, the model created by them does not explain the decision-making process. Behind thousands or millions of neurons of the model, we can mostly just see the weights which are impossible to understand by a human. This lack of transparency causes issues with making potential changes to the network, which may lead to unexpected biases. As neural networks become more significant in our lives, it may lead to increasing number of ethical problems. Secondly, the development of neural networks is non-systematic. It is heavily based on a trial-and-error approach, where developers compose the model architecture from different layers, experiment with hyperparameters and use different training techniques. This does not provide a certainty behind the model and may lead to some unexpected drawbacks.

The Multiple Characterization Problem (MCP) system proposed by Miki Hermann and Gernot Salzer[2] is an alternative to neural networks and does not suffer from the drawbacks mentioned above. It makes a logical analysis of the data, based on propositional logic and produces a conjunctive propositional formula describing the data set. It provides a binary classification of the data based on the input of the variables of the formula.

The current version of MCP, Danube, suffers from two drawbacks. Firstly, the potential exponential explosion of transforming data to binary vectors, which drastically increases the complexity of the program. This problem is expected to be solved with the Mekong version of MCP, which is currently under development. It will use similar techniques but use a many-valued logic approach. Secondly and more importantly for the accuracy of the model is the lack of cleaning of the data. This causes the formulas to take into account outliers, decreasing their efficiency. This problem can be tackled by the introduction of the module which will perform such cleaning.

In this document, we will introduce the Danube version of the MCP project, describing its modular build and the method of how it produces the conjunctive formula from the data set. Then, we will introduce the problem that was approached, which is creating a statistical module for MCP, describing its design and methodology. After this, we will describe its implementation in Python. Finally, we will discuss the difficulties related to the design and implementation of the module and possible future improvements.

2 MCP project description

In this section, we will describe the functionality of the MCP system [2]. The idea behind MCP is based on performing 3 major steps:

- Data binarization – transforming the data to the boolean form, operational to the module
- Core operations – obtaining the propositional formula based on boolean vectors
- Evaluation – checking the accuracy of the model

Each of these steps contains several independent modules:

2.1 Data Binarization Modules

MCP system creates a propositional formula based on binary data vectors. Therefore, the first step of the tool is to get them from the original data, which is assumed to be stored as a CSV file, where each row corresponds

to one data vector. This process is called binarization and it is important to perform it on both discrete and continuous data, which is done in 2 modules.

- **mcp-guess** – the program scans an input CSV file and outputs a template of a metafile, which is used for the next module
- **mcp-trans** – the program takes the data file as an input and the metafile and outputs a binarized file, which is used for the next steps of MCP

Meta template obtained by **mcp-guess** contains indications on the type of coordinates of data (columns). The template needs to be manually modified so that the final file contains transformation commands, which allow to change of the continuous data into intervals and turn them and discrete data into boolean vectors. After the transformation, some boolean vectors can appear that contain the same data. For this reason, there is a module **mcp-uniq** which cleans the repeated vectors.

In the metafile, one of the columns is indicated as a concept column (also referred to as a group). This column is not binarized as it is the target variable, the one we want to predict. In the **mcp-trans**, the group column is moved to become the first column, so we do not need to include the information about the concept in the next modules.

2.2 Core modules

The core of the MCP is responsible for the production of the conjunctive formulas describing the data from their binarized version. To get the data which will be used to obtain the formulas, the module **mcp-split** can be used. It allows to split the data into 2 parts with the given ratio, one of which will be the learning part and the other the testing part. This can be done after the data binarization or before, the module works in both cases.

Then in the next step, the conjunctive formulas will be produced. The core generates such a formula for each of the different data appearing in the concept column. Therefore, if there are 3 different target variables in our data set, the core will produce 3 conjunctive formulas, each of them corresponding to one target variable. To generate a formula for any target variable, we first assign all the data with the target variable as positive examples and the other ones as negative examples. Then, based on this assignment, the core calculates the conjunctive formulas in one of the 4 variants of the module:

- **mcp-seq** which performs the tasks sequentially, is a preferable solution if we have only a few target variables
- **mcp-mpi** and **mcp-pthread** which calculate the formula in parallel, but in a different way
- **mcp-hybrid**, which also runs in parallel and uses methods from 2 previous modules

In case of a greater number of target variables in a group, it is usually preferred to run the parallel modules, however, depending on the case, there is no module which always works more efficiently than the other ones.

There are 4 types of conjunctive formulas which the core might produce depending on the choice: Horn, dual Horn, bijunctive and general CNF formula. The reason for the choice of the 3 first ones is that they are well-studied CNF formulas and the algorithm generating them from a set of boolean vectors is known and runs in polynomial time. The algorithm of the last one is based on the work coauthored by Nicolas Hermann[1]. Additionally, after obtaining the formulas, there is a possibility of postprocessing of the data to reduce it by the redundant clauses. The propositional formula is kept in a DIMACS CNF format.

2.3 Evaluation Modules

The most important module for the evaluation, which allows to test the other part of data in the obtained CNF formula is **mcp-check**. It returns a file which gives information about efficiency measures like ratio and number of true positive examples, false negative or accuracy .

3 Relevance of statistical module for MCP

Classical Machine learning methods, like Neural Networks, which learn information from the data, although efficient, suffer 2 major drawbacks. The first one concerns the fact that their design is usually non-systematic, tailored to the particular problem. The other is not the explainability of the obtained model, where usually it is not possible to retrieve its decision-making process.

MCP Danube, the current version, does not suffer from these drawbacks as it does not need tweaking parameters (making it systematic) and the formula it provides gives a clear explanation of the classification decision. Currently, the MCP Mekong version is under development, which will tackle the problem of the exponential explosion during the transformation of raw data to Boolean vectors, which is present in the current version.

Moreover, MCP Danube faces the problem of a lack of cleaning the data. Because of that, the formula obtained not only takes the outliers into account but also has the same weight as all other data, worsening the efficiency of the model.

To avoid this problem for the current and future versions, the statistical module needed to be introduced, which would allow to clean the data from the outliers and use it to train the model.

4 Presentation of the problem

The problem is to create a module further called **mcp-clean**, which will be used as the first module of the MCP. The module needs to take a data file as an input, which will be a CSV file. It will produce a cleaned data file as an output, which will also be a CSV file. The file needs to be cleaned according to certain instructions which will be present in a metafile, another input file, with a .cln extension. These instructions will need to have a pre-defined grammar, which needs to be understood by the module.

Therefore, the problem of **mcp-clean** can be narrowed down to the design of grammar for the possible meta files, designing a compiler which will translate the metafile to the set of instructions and finally impose those instructions on the input file, to output the modified CSV file.

4.1 Instructions functionality

The instructions should perform data cleaning of the input file. In particular, we want to include 3 types of data cleaning:

- Statistical cleaning – we want to be able to clean rows where the value in a certain column stands out from the others. In particular, we want to apply the 68–95–99.7 rule and clean data which are 1, 2 or 3 variances away from the mean, depending on the user's choice
- Cleaining with an interval – we want to be able to impose an interval in which the values of a certain column need to be and clean the rows which are not within the intervals
- Filtering with a regular expression – we want to be able to filter values with respect to the regular expression given by the user. In case the value does not match the regular expression, the row containing it will be deleted from the dataset.

Moreover, we want to be able to perform statistical cleaning not only on a certain column but also with respect to some groups of features, included in different columns. We want to include this functionality to possibly clean data for local trends (clusters) if there are any.

Therefore, we want to include a "concept", or differently, a "group" instruction, which sets one of the columns with features in relation to which we will perform statistical cleaning, to prevent deleting features from the file.

4.2 Grammar of the meta file

The metafile needs to include instructions which will perform row cleaning with respect to the data coordinates. Therefore, we want for each cleaning instruction to include: the column number (where we want to operate), the type of data of the column (to make sure that the data has a correct type) and a cleaning operation. Moreover, we want to include a concept instruction, which will include a column number and the keyword "concept" or "group".

4.2.1 Syntax of a grammar

The grammar in the Backus-Naur form is presented in Table 1 on page 8.

The elements written in angle brackets correspond to non-terminal symbols and the ones in quotation marks refer to terminal symbols. We can observe that among operations we either have statistical cleaning operations (‘1sigma’, ‘2sigma’, ‘3sigma’) or a set of intervals delimited by "||". This corresponds to the union of intervals so that the filter is looking at whether the value falls to any of the intervals.

4.2.2 Semantics

The program consists of a list of instructions, each of which (except for concept/group instruction) is composed of 3 parts. The first part, which is represented either by a number or a range of natural numbers, corresponds to <columns> non-terminal symbol and describes on which columns the operation is going to be performed. We will design a program such that on none of the columns the operation cannot be made more than once, therefore none of the column ranges and values cannot have the same elements. Then, this part is separated by the syntactic sugar symbol ‘:’.

The second part is the type of the values which will be imposed on that column. In case the type of the value in a certain row does not match the type of the instruction, the row is erased. There are 4 possible types: ‘nat’, ‘int’, ‘real’ and ‘string’. The type is then separated by the syntactic sugar symbol ‘=’.

The last part is the operation that will be performed. In the syntax description, we separated ‘string’ from the other types. It is because for this type there is a unique operation to which <regex> corresponds. Within this symbol, there can be put any regular expression which can be put into Python after space clearing. Therefore, if we want to put a character into the regular expression which might be understood as a special character, we would put ‘b́efore it. For the ‘nat’, ‘int’ and ‘real’ types, we have 2 possible types of operations: <intervals> which represents the union of interval(s) in which the values of the column need to appear, and ‘1sigma’, ‘2sigma’ and ‘3sigma’ operations – accepting respectively values at most 1, 2 or 3 variances from the mean.

The intervals can be represented in 2 ways:

- In a form <int> ‘..’ <int>, which initially was supposed to be prescribed for ‘int’ and ‘nat’ types and meaning to take all the integers between the first and the second one (included), but for flexibility it extends to include also all the real numbers between the first and the second integer
- In a form <lbracket><real> ‘..’ <real><rbracket>, where the <real> represents a real number, <lbracket> represents either ‘[’ or ‘(’ and analogically <rbracket> represents either ‘]’ or ‘)’, meaning open and closed intervals

For the concept instruction, we only have 2 parts: the first one is a column number (so we cannot put the range here) and after the ‘:’ symbol, the second part is ‘group’ or ‘concept’ instructions which are the same and assign the chosen column to the features with respect to which we perform statistical cleaning.

Let us look at an example of iris.cln metafile:

```
0..3: real = 3sigma;
4: string = setosa | versicolor | virginica;
```

Table 1: Backus-Naur form of a grammar of a metafile

⟨*instructions*⟩ ::= ⟨*instruction*⟩ | ⟨*instruction*⟩ ⟨*instructions*⟩

⟨*instruction*⟩ ::= ⟨*columns*⟩ '`:`' ⟨*type_operation*⟩ '`;`' | ⟨*nat*⟩ '`:`' ⟨*concept*⟩ '`;`'

⟨*columns*⟩ ::= ⟨*nat*⟩ | ⟨*nat*⟩ '`..`' ⟨*nat*⟩

⟨*type_operation*⟩ ::= ⟨*type*⟩ '=' ⟨*operation*⟩ | '`string`' '=' ⟨*regex*⟩

⟨*type*⟩ ::= '`int`' | '`nat`' | '`real`'

⟨*operation*⟩ ::= ⟨*intervals*⟩ | '`1sigma`' | '`2sigma`' | '`3sigma`'

⟨*intervals*⟩ ::= ⟨*interval*⟩ | ⟨*interval*⟩ '`||`' ⟨*intervals*⟩

⟨*interval*⟩ ::= ⟨*int_interval*⟩ | ⟨*real_interval*⟩

⟨*int_interval*⟩ ::= ⟨*int*⟩ '`..`' ⟨*int*⟩

⟨*real_interval*⟩ ::= ⟨*lbracket*⟩ ⟨*real*⟩ '`..`' ⟨*real*⟩ ⟨*rbracket*⟩

⟨*lbracket*⟩ ::= '`(`' | '`[`'

⟨*rbracket*⟩ ::= '`)`' | '`]`'

⟨*concept*⟩ ::= '`concept`' | '`group`'

⟨*real*⟩ ::= ⟨*int*⟩ | ⟨*int*⟩ '`.`' ⟨*nat*⟩ | ⟨*int*⟩ '`.`' ⟨*nat*⟩ '`e`' ⟨*int*⟩ | ⟨*int*⟩ '`.`' ⟨*nat*⟩ '`E`' ⟨*int*⟩

⟨*int*⟩ ::= ⟨*nat*⟩ | '+' ⟨*nat*⟩ | '−' ⟨*nat*⟩

⟨*nat*⟩ ::= '`0`' | ⟨*nzdigit*⟩ | ⟨*nzdigit*⟩ ⟨*digits*⟩

⟨*digits*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*digits*⟩

⟨*digit*⟩ ::= ⟨*nzdigit*⟩ | '`0`'

⟨*nzdigit*⟩ ::= '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`'

⟨*regex*⟩ ::= extended regular expression

And, for reference, a snip of iris.csv, on which the operations are going to be performed:

```
4.6 3.2 1.4 0.2 setosa
5.3 3.7 1.5 0.2 setosa
5.0 3.3 1.4 0.2 setosa
7.0 3.2 4.7 1.4 versicolor
6.4 3.2 4.5 1.5 versicolor
6.9 3.1 4.9 1.5 versicolor
```

The first line of iris.cln describes that the columns from 0 to 3 are going to be treated with 3-sigma statistical cleaning (the values accepted are within 3 variances from the mean), while the column number 4 is going to accept only strings matching the regular expression 'setosa | versicolor | virginica', so only the strings 'setosa', 'versicolor' and 'virginica'. The spaces from the regular expression should be erased unless they're quoted or the space follows the backslash character.

5 Implementation of mcp-clean

The **mcp-clean** module has been developed as a Python program which can be divided into several parts. In total, it consists of around 750 lines of code, most of which are dedicated to compiling the metafile. We can distinguish the main parts of the module:

- Document parser, which parses file names from a command-line argument
- Metafile Lexer, which reads the metafile and generates tokens used for Parsing. In case the token is not recognised, the Lexer raises a syntax error
- Metafile Parser, which works in parallel with Lexer and produces instructions which will be applied to the input CSV file. In case the instruction is not recognised, the Parser raises a runtime error
- Operations class, which enables the program to create an Operations object, which saves the data about the instructions which will be performed
- Interpreter class, which uses Operation class to impose the instructions on the input CSV file, to create an output CSV file

The **mcp-clean** module can be found on the GitHub with the example csv and cln files under this link: https://github.com/oskdbk/mcp-clean

5.1 Document parsing

At the beginning, **mcp-clean** parses the file paths of the input file, metafile and output files. For this purpose, argparse Python library has been used. These documents will be opened in the next modules.

5.2 Tokens

The program introduces a Token class. When a Token object is created, it contains information about the token's type, its lexeme and the position of the token in the metafile. The latter is used for later error handling. The list of the 3 character long tokens is introduced as global variables for later use for simplicity, as they are extensively used for lexing. The token types with their explanations look as follows:

```
### Tokens:
#numbers
NUM = "NUM" #integers
FLT = "FLOAT" #floats / real numbers
INF = "INF" #infinity (+-)

#syntactic sugar and symbols
HYP = "HYPHEN" # "-"
COL = "COLON" # ":"
EQU  = "EQUAL" # "="
SEM = "SEMI-COLON" # ";"
LCP = "LEFT BRACKET" # "["
RCP = "RIGHT BRACKET" # "]"
LOP = "LEFT PARANTHESIS" # "("
ROP = "RIGHT PARENTHESIS" # ")"
DDT = "DOUBLE DOT" # ".."
IOR = "INTERVAL OR" # "||"

#types
INT = "INT"
NAT = "NAT"
REL = "REAL"
STR = "STRING"

#statistical cleaning operations
SG3 = "3SIGMA" # 3 variances from the mean
SG2 = "2SIGMA" # 2 variances from the mean
SG1 = "1SIGMA" # 1 variances from the mean

#group/concept
GRP = "GROUP"  # group/concept operation
#regex
RGX = "REGEX" # to save the regular expression
```

5.3 Lexing

To perform Lexing of the metafile, the program introduces a Lexer class, which reads the metafile and keeps track of its behaviour. Moreover, it is equipped with an error method, which is triggered in case of encountering a syntax error and pointing to the place where it is encountered.

The main part of this class is the next_token() method which returns the next token found in the file. This method takes the next characters from the file by moving the file pointer and reading it as long as the token is recognised. However, there are some exceptions where additional flags are kept by the Lexer object to keep track of them.

5.3.1 Double dot flag

Usually, the next_token() method needs to look 1 character ahead to recognise a token. In such a case, when the next character does not belong to the currently analysed token, we put it in an attribute self.is_char, which

otherwise carries a False value. However, there is a case where we need to look 2 characters ahead to recognise a current token. Once again, let us look at the abalone.cln example:

```
0: concept;
1..7: real = 3sigma;
8: nat  = [1..2] || [2..29];
```

On the third line of this file, we want to impose a union of closed intervals, denoted as '[1..2] || [2..29]' (which is equivalent to 1..29 in this case). When we start reading the interval the Lexer denotes the '[' character and returns an LCP token. Then, we observe a character '1', however, we do not know whether it represents a natural number or a real number, especially because the next character is '.', which would usually imply a real number, but in this case, it is part of a double dot (DDT) token. Therefore, in this case, we need to look at 2 characters ahead, but only 1 token ahead. To take care of this exception, there is an attribute self.is_ddot which is True when the next token is a double dot token in case it could not be recognised as one after reading one character ahead, otherwise, it is false.

5.3.2 Comment flag

For a better user experience, the module gives the possibility of making comments. Similarly, as in Python, the '#' character makes the program treat the rest of the line as a comment. The only exception from that functionality of this symbol is when it is inside a regular expression and when it is either quoted with single or double quotation marks or when it has a backslash character before. Otherwise, when the program reads a '#' character, it switches the flag self.is_comment to the True value and turns it back to false when the '\' character is encountered.

5.3.3 Regex and pre-regex flag

The functionality of filtering by a regular expression may lead to an overcomplicated lexing process if we just simply look for 1 character ahead. Fortunately, thanks to the distinctive grammar of the instruction which performs filtering by a regular expression, we can predict then we can expect an RGX token.

By the build of grammar, we know that the RGX token appears if and only if it is preceded by STR and EQU tokens. Therefore, we introduce 2 flags: self.is_preregex() which becomes True when the program encounters the STR token and self.is_regex() which becomes True when self.is_preregex() is True and when the program observes an EQU token. The latter flag corresponds to the mode where we put all the elements into the next RGX token and goes back to the False value (along with self.is_preregex) when the program reads an ';' character. However, we also want to be able to potentially put this character as part of some regular expression. We will enable it if this character is preceded by a '\' character or if it is in a quotation.

5.3.4 Backslash and quotation marks flags

These flags give the possibility for the user to include any symbol literally in the RGX token. In particular, it allows the addition of a ';' character as well as a space character to the regular expression (otherwise the spaces are cleaned).

The program recognizes 3 attributes, which can be changed only inside the regular expression. The first one is self.is_slash which can change the value to True when the program reads the '\' character and changes to False on the next character. It allows to pass the next character to the RGX token.

We can obtain a similar effect with self.is_q1 and self.is_q2 flags, the former detecting if we started a double quote and the latter if we started a single quote inside a regular expression. If one of these flags is True, its value becomes False once the program reads the same character again. everything that is read when the flag is

True is put into the RGX token (without quotation marks). In case one wants to include the quotation marks in a regular expression, the backslash should be used.

The lexer is written so that only one of these three flags can be True at one time.

5.4 Parsing

For parsing, we created a Parser class, which takes a Lexer and Operations class object as an attribute, the latter will be described in the next section. The idea behind the Parser is to analyse the next tokens, which represent terminal symbols or sets of terminal symbols according to the formal grammar described in section 4.2.1. This parser is an LL(1) parser as it is a top-down parser with left-to-right derivation which looks 1 token ahead. It works in parallel with a lexer which generates the new tokens. The parser passes the tokens through an automaton trying to match their sequences with a particular instruction, which is passed further. If it is not recognised, it makes the Token object raise a RuntimeError.

5.5 Operation class

The Operation class saves the set of instructions taken by the Parser and is later used for an Interpreter class to execute them on the CSV file. Because we will read the CSV file line-by-line (as these may become too big to open), we will need to group the instructions in the operation class so that we can access them by the column number. This class contains only attributes which are:

- self.concept – which at the beginning takes the value -1 (meaning that no concept was established) and when the concept instruction is read, the value of the attribute changes the index of the column where the concept is set
- self.nr_rows – which keeps track of how many lines are in the CSV input file, used for statistical analysis when no concept is present
- self.list_instructions and self.list_stats – arrays which contain the indices of columns on which respectively there will be instruction and there will be statistical instruction performed. This information is collected during parsing and is used to iterate over the desired indices when the instructions are interpreted
- self.dict_instructions – a dictionary, which is modified during parsing. It contains column numbers as keys and instructions to be performed as values. The instructions are saved as 2-element tuples. The first element represents the type of values which are expected to appear in the column. The second one depends on the instruction:
  - if the instruction is represented by the regular expression (for the string type), then the second element is just a string representing a regular expression
  - if the instruction is a statistical cleaning (represented by either ‘1sigma’, ‘2sigma’ or ‘3sigma’), then this element contains the string which represents this instruction
  - if the instruction is filtering by the union of intervals, then this element is an array of intervals. Each interval is represented by a 2-tuple of 2-tuples, where the first represents the left bound of the interval and the second the right bound of the interval. Each of them contains information on the value stored and whether the interval from this side is open or closed. For example, the interval denoted by ‘[2.1 .. 3,5)’ in the metafile will be saved as ((2.1, ’c’),(3.5, ’o’)), where ’c’ and ’o’ correspond to closed and open intervals respectively.
- self.dict_stats – a dictionary which will contain information about the columns on which the statistical analysis will be performed. It is initialised during parsing where the keys are represented by the indices

of columns on which the statistical analysis will be performed. In the beginning, the values for each of the keys are empty dictionaries.

During the interpretation, these empty dictionaries are being filled with statistical information about the column. These dictionaries take 2 different forms

- If the concept is not present, then the values of the keys represented by the column indices are the dictionaries of statistical measures of the column. This dictionary has 4 keys: 'sum' whose value is initialised by 0 and adds the values from the column, 'sqsum', similar, to the 'sum', but adds squared values from the column, 'mean', whose value is the mean of the values of the column and 'std' whose value is the variance of the values.
- If the concept is present, then the build of the dictionary is similar, but there is an intermediate dictionary which takes as keys possible values in the concept column and as values the aforementioned statistical measures. These dictionaries are nested such that at first, we want to call a column index, then the name of some value from the concept column and finally the statistical measure. We want to do it to keep track of such measures for each value from the concept, as the statistical clearing will be done with respect to each of these values.

5.6 Interpreter class

The Interpreter class takes as an attribute an Operation class object initialised during parsing and applies the instruction included in it on the data. Therefore, it also takes input and output CSV files' pointers as attributes and based on them creates objects from the CSV Python library which allows us to read and write line-by-line, with the arrays.

The class is equipped with self.interpret() method which executes the instructions on the input CSV file. Until now the metafile is no longer open. Because of the different implementations of the Operation class object depending on whether there is a concept introduced or not, this method triggers either self.interpret_con() or self.interpret_nocon(). The former is taken into account when the concept instruction appears in the metafile, the latter otherwise.

The interpretation is based on 2 parses through the CSV input file. During the first parse, the program collects statistical data from the columns on which statistical analysis will be imposed. We need to do it in order to calculate the mean and variance which will be used for '1sigma', '2sigma' and '3sigma' operations. After the first parse, the program puts the file pointer to the beginning of the file and reads it again, now applying all the cleaning included in the Operation class object.

In the case of the concept, we also need to include the tracking of how many repeating target values there are in the concept column. In this case, the Operation class has an attribute which tracks it during the first parse.

When the interpreter encounters data which is not of the correct type (which is included in the Operation class object) then the row is not written to the output file. If the data is missing, the missing cell is replaced by a '?' character, which is expected to be handled by other modules of MCP.

6 The result of the implementation

The results can be divided into 2 parts. Firstly, we will discuss the results when it comes to the implementation of the code, whether the program works against different metafiles and the efficiency of the code. Secondly, we will see the impact of the work on the accuracy improvement of the whole MCP module.

6.1 The quality of the module

The module covers numerous cases during lexing and parsing when deciding on the token and instruction which will be passed further. During the testing procedure, there was no issue with an unexpected program error. The

program can handle 3 types of errors for which it gives a customized message:

- Error related to opening the file – it is triggered when the file is not found or meets some unexpected issue
- SyntaxError, which is raised by the lexer when the token cannot be completed (for example the next character does not match any token). Moreover, the error shows on which line and on which position the error occurs
- RuntimeError, which is raised by the parser through the Token class when the instruction cannot be completed. The token object contains its position in the original file and it shows it with the characters which are after the place where the error is present

For example, the module was run on iris2.cln metafile which looks like the following:

```
0..3: reall= 2sigma;
4: string = setosa | versicolor | virginica;
```

We can notice a syntax error in the first line of the file, because of the misspelling of the 'real' token. The error message looks like the following:

```
Traceback (most recent call last):
  File "C:\Users\oskar\Desktop\University\Semester 6\Thesis\mcp-clean_3.py", line
    parser.parse()
  File "C:\Users\oskar\Desktop\University\Semester 6\Thesis\mcp-clean_3.py", line
    token = self.lexer.next_token()
            ^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\oskar\Desktop\University\Semester 6\Thesis\mcp-clean_3.py", line
    self.error(f"name \"{t}\" is not defined", char)
  File "C:\Users\oskar\Desktop\University\Semester 6\Thesis\mcp-clean_3.py", line
    raise SyntaxError(f"in line {self.lin_num}, on the position {self.pos_num}\
SyntaxError: in line 1, on the position 7: name "reall" is not defined
= 2sigm
^
```

The position to which the error refers shows where the wrong token starts. The arrow at the bottom points to the next token after the mistake is noted.

In order to show the tokenization process, we slightly modified the source code which printed all the created tokens. We will show it in the example of ff.cln (metafile which corresponds to the Forest Fires dataset) which looks like follows:

```
0: nat = 0..9;
1: nat = 2..9;
2: string = jan | feb | mar | apr
| may | jun | jul | aug
| sep | oct | nov | dec;
#3: string = mon | tue | wed
#| thu | fri | sat | sun;
#4..8: real = 3sigma;
#9: nat = 15..100;
#10..12: nat = 3sigma;
```

In this example, we can also see the use of 2 other functionalities: comments (for the second half of the file) and the possibility of splitting the regular expression with an enter. For this example, the tokens look like follows:

```
[(NUM, 0, 1, 1),
(COLON, :, 1, 2),
(NAT, , 1, 4),
(EQUAL, =, 1, 8),
(NUM, 0, 1, 10),
(DOUBLE DOT, .., 1, 11),
(NUM, 9, 1, 13),
(SEMI-COLON, ;, 1, 14),
(NUM, 1, 2, 1),
(COLON, :, 2, 2),
(NAT, , 2, 4),
(EQUAL, =, 2, 8),
(NUM, 2, 2, 10),
(DOUBLE DOT, .., 2, 11),
(NUM, 9, 2, 13),
(SEMI-COLON, ;, 2, 14),
(NUM, 2, 3, 1),
(COLON, :, 3, 2),
(STRING, , 3, 4),
(EQUAL, =, 3, 11),
(REGEX, jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec, 3, 13),
(SEMI-COLON, ;, 5, 24)]
```

We can check that all the tokens correspond to the metafile. Moreover, the third and fourth elements show the line and column number in the metafile where the token starts.

When it comes to the execution time, for the CSV files whose size is up to 5MB, the execution time is around a second. To test the module's performance on bigger files, I used an example of creditcard_2023.csv which has 317MB. The statistical cleaning was performed on this data with the following creditcard.cln file:

```
1..28: real = 3sigma;
30: concept;
```

The module terminated after around 50 seconds and produced a cleaned CSV file which has 270MB.

6.2 Accuracy improvement for the MCP

MCP accuracy depends on the chosen CNF formula which needs to be produced. Usually bijunctive formula has the smallest accuracy, sometimes comparable to flipping a coin. However, it is produced the quickest. For the general CNF formula, the results are usually accurate, with accuracy over 80%, but it takes more time to generate it. Finally, the Horn formulas give usually the greatest accuracy among these three, but take the longest to produce.

To show the impact of the **mcp-clean** module on the accuracy improvement, we took two examples of data sets, one which had low accuracy without cleaning and the other whose accuracy was already great. We will only compare the general CNF formulas for the target variable with the smallest improvement (which are still significant), but the other results are available on above mentioned Github.

6.2.1 Iris example

The iris dataset contains 3 types of irises: Setosa, Versicolour, and Virginica, which usually serve as a target variable, which is the case for us. There are also 4 other types of variables which correspond to the flowers' characteristics and are real numbers. The smallest improvement (by percentage points) of accuracy for general CNF formulas, when the dataset iris.csv was treated by the MCP system, was in the case of the 'Setosa' target variable. Figure 1 shows a snippet of the results for the 'Setosa' CNF formula's efficiency without cleaning: The accuracy of 85% is already high but is improved after the cleaning. Figure 2 shows a snippet of the results

Figure 1: 'Setosa' CNF formula's efficiency without cleaning

```
+++ sensitivity    (tpr) = 100     %  [tp / (tp + fn)]
+++ miss rate      (fnr) = 0       %  [fn / (fn + tp)]
+++ fall-out       (fnr) = 20.3704 %  [fp / (fp + tn)]
+++ specificity    (tnr) = 79.6296 %  [tn / (tn + fp)]
+++ precision      (ppv) = 63.3333 %  [tp / (tp + fp)]
+++ accuracy       (acc) = 84.9315 %  [(tp + tn) / (tp + tn + fp +fn)]
+++ F_1 score      (F_1) = 77.551  %  [tp / (tp + 0.5 * (fp +fn))]
```

for the 'Setosa' CNF formula's efficiency after '3sigma' cleaning for all the numerical rows: The CNF formulas

Figure 2: 'Setosa' CNF formula's efficiency with '3sigma' cleaning

```
+++ sensitivity    (tpr) = 100     %  [tp / (tp + fn)]
+++ miss rate      (fnr) = 0       %  [fn / (fn + tp)]
+++ fall-out       (fnr) = 0       %  [fp / (fp + tn)]
+++ specificity    (tnr) = 100     %  [tn / (tn + fp)]
+++ precision      (ppv) = 100     %  [tp / (tp + fp)]
+++ accuracy       (acc) = 100     %  [(tp + tn) / (tp + tn + fp +fn)]
+++ F_1 score      (F_1) = 100     %  [tp / (tp + 0.5 * (fp +fn))]
```

for two other target variables had a greater improvement concerning percentage points, as the results before the cleaning had smaller accuracy. However, the 100% accuracy obtained is also because the iris dataset contains only 150 elements.

6.2.2 Banknote example

The Banknote example contains only 2 types of target variables: genuine and forged (indicating whether the banknote is forged or not). Similarly to the iris example, the banknote dataset also contains 4 other types of variables, which are real numbers. Therefore for comparison, this dataset will also be treated with '3sigma' cleaning on each of the 4 other columns. In this example, the 'forged' CNF formula had a smaller accuracy improvement (by percentage points) after cleaning. Therefore, Figure 3 shows a snippet of the results for the 'forged' CNF formula's efficiency without cleaning: In this case, the original dataset contains 1371 elements,

Figure 3: 'forged' CNF formula's efficiency without '3sigma' cleaning

```
+++ sensitivity    (tpr) = 100     %  [tp / (tp + fn)]
+++ miss rate      (fnr) = 0       %  [fn / (fn + tp)]
+++ fall-out       (fnr) = 7.57576 %  [fp / (fp + tn)]
+++ specificity    (tnr) = 92.4242 %  [tn / (tn + fp)]
+++ precision      (ppv) = 93.4211 %  [tp / (tp + fp)]
+++ accuracy       (acc) = 96.3504 %  [(tp + tn) / (tp + tn + fp +fn)]
+++ F_1 score      (F_1) = 96.5986 %  [tp / (tp + 0.5 * (fp +fn))]
```

significantly more than the iris example. Although the accuracy is already on the level 96.4%, it is substantially increased after the cleaning, which can be observed in Figure 4 presenting the snippet of the results for the 'forged' CNF formula's efficiency with cleaning. Although the accuracy without cleaning was already high,

Figure 4: 'forged' CNF formula's efficiency with '3sigma' cleaning

```
+++ sensitivity    (tpr) = 100     %  [tp / (tp + fn)]
+++ miss rate      (fnr) = 0       %  [fn / (fn + tp)]
+++ fall-out       (fnr) = 0.983607 %  [fp / (fp + tn)]
+++ specificity    (tnr) = 99.0164 %  [tn / (tn + fp)]
+++ precision      (ppv) = 99.2177 %  [tp / (tp + fp)]
+++ accuracy       (acc) = 99.5624 %  [(tp + tn) / (tp + tn + fp +fn)]
+++ F_1 score      (F_1) = 99.6073 %  [tp / (tp + 0.5 * (fp +fn))]
```

the cleaning helped to obtain an accuracy close to 100%.

Similarly to those examples, in most cases, there is a significant improvement in accuracy for the formulas which were obtained from the cleaned learning data set. Usually, the '3sigma' cleaning was providing the best improvements.

7 Possible improvements and discussion

The **mcp-clean** module is an important part of the MCP system. Without the data cleaning, the system was obtaining unsatisfying results, especially for bijunctive formulas. This program can not only serve MCP Danube but also will be able to work for the next MCP Mekong version as it is independent from the other modules. However, after it is introduced, the data split will need to be done before binarization. This is because the statistical cleaning of the data will not serve well for the binary vectors. It also shows that if the input data from which we want to obtain a CNF formula is binary at the beginning, the module might not help with the data cleaning.

By performing statistical analysis with the established concept, we were performing it on some pre-settled clusters of data which are usually real numbers. One of the possible improvements would be to include cleaning in such clusters for binary vectors.

However, the module could also give more possibilities for statistical analysis. The data might be expected to have a gamma distribution. The functionality of imposing different distributions than gamma could be added.

The part of the compiler created for **mcp-clean** serves its role, but leaves the space for improvements. For example, there could be another flag for the parser which contains the previous character (or characters) to enable showing them in an error message. It would allow the user to see the exact position of the error made, similarly as it can be seen in Python. Another issue with the error handling of the module is that sometimes when RuntimeError is detected, the program points out the wrong reason for an error. That does not impose problems when the metafile can be parsed but could be improved by more extensive testing or introducing new flags for this purpose.

When the program takes big CSV files as input, it takes some time to clean it. As metafile is described with an easy language, the problem lies in a CSV file. The potential improvement in this field could be achieved by checking how it affects the execution time to read more lines of a CSV file at once and if it is better to move the pointer to the beginning of the CSV file after the first parse or to close it and open again.

Developing this module not only improved my programming skills but also allowed me to implement part of the compiler, which I did not have the opportunity to encounter. During the internship, although the module does not need to interpret a very difficult language, I could discover how a more systematic approach could be implemented and why it serves better in the long-run for the developer than the ad-hoc solution.

8 References

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

[2] Miki Hermann and Gernot Salzer. MCP: capturing big data by satisfiability. In Chu-Min Li and Felip Manyà, editors, *Proceedings 24th International Conference on Theory and Applications of Satisfiability Testing (SAT 2021), Barcelona (Spain)*, volume 12831 of *Lecture Notes in Computer Science*, pages 207–215. Springer, 2021.

A Appendix