AN IMPLEMENTATION FOR A HEURISTIC RANGE-BASED CLASSIFICATION RULE DERIVING ALGORITHM

PHUONG ANH TRAN

MSc Computing

A DISSERTATION SUBMITTED FOR THE DEGREE OF MSC COMPUTING

DEPARTMENT OF COMPUTER SCIENCE
CARDIFF UNIVERSITY

2016

Declaration

CANDIDATE'S ID NUMBER	1571743
CANDIDATE'S SURNAME	MS TRAN
CANDIDATE'S FULL FORENAMES	PHUONG ANH

This work has not previously been accepted in subtance for any degree and is not concurrently submitted in candidature for any degree.

Signed .... (candidate) Date ....

STATEMENT 1

This dissertation is being submitted in partial fulfillment of the requirements for the degree of ....

Signed .... (candidate) Date ....

STATEMENT 2

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A Bibliography is appended.

Signed .... (candidate) Date ....

STATEMENT 3

I confirm that the electronic copy is identical to the bound copy of the dissertation.

Signed .... (candidate) Date ....

STATEMENT 4

I hereby give consent for my disseration, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .... (candidate) Date ....

STATEMENT 5 - BAR ON ACCESS APPROVED

I hereby give consent for my dissertation, if accepted, to be available for photocopying and for inter-library loans **after expiry of a bar on access approved by the Graduate Development Committee.**

Signed .... (candidate) Date ....

Acknowledgments

To my supervisor, Dr Shao Jinhua, whom inspires me with confidence and beyond.

Abstract

This project aims to implement an existing heuristic algorithm for deriving range-based classification rules in Java. Its highlighted features include using class values to guide its associated range search, accompanied by efficient methods used to split sub-ranges which help produce more accurate rules. The programme built for this algorithm will carry on those features with slight modifications in adaptation to the data structure of the chosen programming language.

Contents

List of Figures

List of Tables

List of Algorithms

Section 1

Introduction

The ability to search and extract meaningful information from a large set of data has been the core strength of data mining and machine learning research. One approach is to recognise certain patterns within a dataset, then transform these patterns into rules so later these rules could be used to predict a classifiable outcome for any given instance within that dataset. The process of transforming such patterns into rules thus requires efficient and accurate methods in order to produce correct predictions. It is important to choose the right methods and algorithms that could derive useful and relevant rules.

The goal of this project is to analyse and implement an existing classification rule mining algorithm, in which rules are derived for ranged-based data. As opposed to categorical values, continuous values pose a more challenging problem since it is more difficult to determine the effective range boundaries to produce fitting predictions. Subsequently, the effectiveness depends on how well ranges are split into 'right intervals' [1]. To determine the right split, the algorithm implemented in this project proposes an effective splitting method, as well as guiding its search for sub-optimal ranges using class values of the dataset. This project follows a similar structure set by these methods in its implementation.

In details, the existing algorithm (referred to as *original algorithm* for the rest of the report) withholds two distinctive features: firstly, optimal sub-ranges are found with an approach similar to Kadane's solution to the maximum subarray problem [2], which is highly efficient due to its linear time complexity. Secondly, inspired by association rule mining, sub-ranges can combine for larger associated ranges with an algorithm similar to *A-priori* method in generating large frequent itemsets [3]. These two features make for a fast and efficient way to classify range-based datasets which are implemented in this project, alongside some modifications for adaptions to Java data structure.

Section 2

Background

This section presents the concepts as well as the original algorithm structure. Dataset examples are provided for each concept. Step-by-step explanations are provided for the algorithm.

2.1 Preliminaries

Assuming that we have a dataset that could be presented as a table $T(A_1, A_2, A_3, ..A_m, C)$, where $A_{j,m}$ as $1 \leq j \leq m$ are range attributes and C is the categorical class value. A single tuple within T is denoted as $t_k = (v_1, v_2, v_3..v_m, C)$ where v_m is a value under A_m.

A sample dataset served as an example for this section can be shown in Table 2.1:

Table 2.1: A sample dataset as table

T	*A1*	*A2*	*A3*	*A4*	*C*
t1	0.75	1.45	2.13	4.56	c_1
t2	0.64	1.62	2.64	4.75	c_2
t3	0.71	1.21	3.11	3.97	c_1
t4	0.57	1.23	2.75	4.24	c_1
t5	0.80	1.53	2.34	4.11	c_2

Definition 1 (range) Assume within the domain of attribute A exists two values a and b that represent a continuous range over A_j, denoted $r = [a, b]_{A_j}$. This range covers a set of values in A that lies between a and b.

Example 1: From Table 2.1, a range of $[0.64, 0.75]_{A_1}$ would cover a set of values 0.64, 0.71

and 0.75 in A_1.

Definition 2 (cover) Assume $r = [a, b]_{A_j}$ to be a range over attribute A. This range r covers a set of tuples where their values are between a and b where $a \leq v_j \leq b$. This set of tuples that is covered by r is denoted $\tau(r)$.

Example 2: From Example 1, a range of $[0.64, 0.75]_{A_1}$ would have a cover of tuples t_2, t_3, t_1, so its covered tuple set is t_2, t_3, t_1.

Definition 3 (associated ranges) Assume $r_1 = [a_1, b_1]_{A_1}$ to be a range over A_1 and $r_2 = [a_2, b_2]_{A_2}$ to be a range over A_2. Those ranges are associated ranges if $\tau(r_1) \cap \tau(r_2) \neq \varnothing$

Example 3: Assume $r_1 = [0.64, 0.75]_{A_1}$ and $r_2 = [1.21, 1.45]_{A_2}$. Table 2.1 shows that r_1 covers t_1, t_2, t_3 and r_2 covers t_1, t_4, t_3, which means $r_1 \cap r_2 = \{t_1, t_2, t_3\} \cap \{t_1, t_3, t_4\} = \{t_1, t_3\}$. Since there are mutual tuples between these ranges, r_1 and r_2 are associated ranges.

Definition 4 (range-based classification rule) Assume c to be a class value from C and $r_1, r_2, r_3..r_h$ be a set of associated ranges. $r_1, r_2, r_3..r_h \rightarrow c$ is a range-based classification rule.

By definition, any tuple t_k can form a rule. However, this will not be accurate nor useful. To determine whether a rule is qualified, there are criteria to be used, which are to be discussed in the sub-section below.

2.2 Criteria

There are three criteria used in this algorithm: support, confidence and density. This sub-section presents the formal definitions for those measures, while examples are to be provided in the next sub-section where the actual algorithm is reviewed.

Definition 5 (support) Assume T to be a table and $r_1, r_2, r_3..r_h \rightarrow c$ to be a range-based classification rule derived from T. The support for r, provided $|\ .\ |$ being the size of a set, is:

$$\sigma(r) = \frac{|\ \tau(r_1) \cap \tau(r_2) \cap ... \cap \tau(r_h)\ |}{|\ T\ |}$$

Definition 6 (confidence) Assume similar settings from Definition 5, the confidence for r in T is:

$$\delta(r) = \frac{\mid \tau(r_1) \cap \tau(r_2) \cap ... \cap \tau(r_h) \cap \tau(c) \mid}{\mid \tau(r_1) \cap \tau(r_2) \cap ... \cap \tau(r_h) \mid}$$

Definition 7 (density) Assume similar settings from Definition 5, the density for r in T is:

$$\gamma(r) = \frac{\mid \tau(r_1) \cap \tau(r_2) \cap ... \cap \tau(r_h) \cap \tau(c) \mid}{\mid \tau(r_1) \cup \tau(r_2) \cup ... \cup \tau(r_h) \mid}$$

2.3 Original algorithm

The original algorithm provides a more efficient way to discover and combine associated ranges for larger rule sets. Instead of traversing through all possible combinations of ranges among all classes and attributes, this algorithm first divides the data into sub-data pools in accordance to its class value. Then, under each attribute, it searches for sub-ranges that could pass certain user-defined thresholds (confidence, support and density). These sub-ranges are then collected and ready to be tested for possible combinations between each other that passes certain thresholds similar to the previous step, which are then collected and ready to be tested for larger combinations. Additionally, after larger combinations are found, an extra adjustment step is done on class values of relevant tuples within the new associated ranges. Afterwards, new associated ranges are ready to be analysed again to find more possible sub-ranges and new combination of sub-ranges. This iteration continues until no possible combinations can be found, so that the last standing associated ranges are translated into the concluding classification rule for that class value.

To get a clearer structure of the algorithm, its proceedings can be divided into two phases:

Phase I (Analyse) This phase has its primary concern as to analyse ranges. Based on the algorithm description, it includes:

1. Partition data into sub-data pool using class values.

2. For each attribute in sub-data pool, find a maximum and minimum range within that attribute and use it as a starting range for next step.

3. From min-max starting ranges, search for sub-ranges that passes certain user-specified

thresholds such as confidence, support and density. Sub-ranges are collected and passed onto Phase II to generate larger associated ranges.

Phase II Generate This phase has its primary concern as to generate larger associated ranges for the next iteration. Based on the algorithm, it includes:

1. For each sub-range obtained from last phase, check for possible combination of that range with the remaining ones.
2. Adjust class values of relevant tuples $r_1, r_2, r_3...r_h \rightarrow c$ under the new combined sub-range.

Details for each phase are as follows:

2.3.1 Original Phase I Analyse

Given a table $T(A_1, A_2, A_3, ..A_m, C)$, the algorithm attemps to seek for classification $r_1, r_2, r_3..r_h \rightarrow c_j$ by finding, among instances that are under class $c_j \in C$, a number of associated ranges $r_1, r_2, r_3..r_h$ that have at least a minimum support (σ_{min}), density (δ_{min}) and confidence (γ_{min}) that had previously been specified by the user. This process can be explained in Phase I (Analyse) in which the algorithm seeks for associated ranges and checks if thresholds are satisfied.

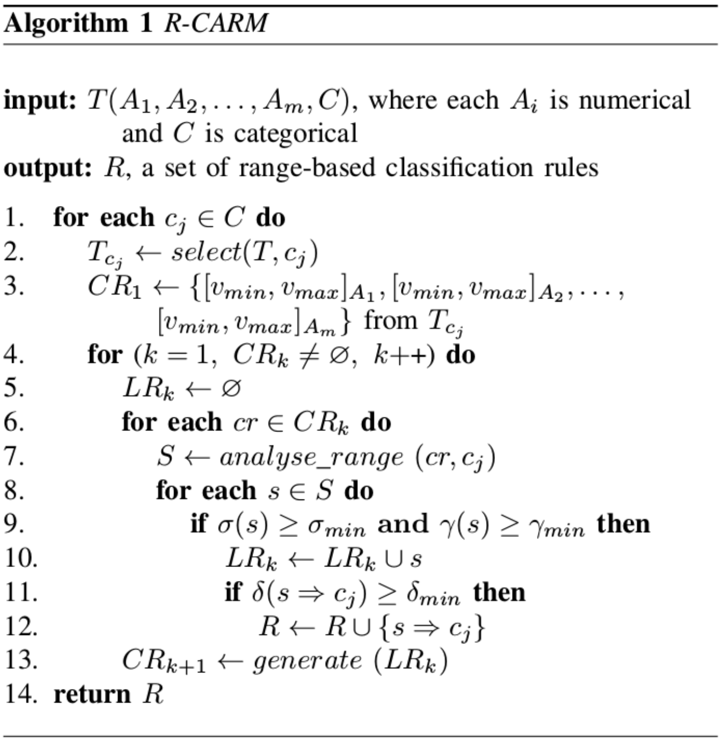
From Figure 2.1, a step-by-step explanation of this algorithm is as follows:

Step 1-2 As the beginning steps of the algorithm, they selects a set of tuples $t_1, t_2, t_3..t_N$ from T that has class value $c_j \in C$ and store them in T_{c_j}.

Example: Using Table 2.1 from last section, for class c_j where $j = 1$ (class 1), tuple set T_{c_1} under this class are:

Table 2.2: A subset of tuples under class 1

T_{c_j}	*A1*	*A2*	*A3*	*A4*	C_j
t1	0.75	1.45	2.13	4.56	1
t3	0.71	1.21	3.11	3.97	1
t4	0.57	1.23	2.75	4.24	1



```
Algorithm 1 R-CARM

input: $T(A_1, A_2, \ldots, A_m, C)$, where each $A_i$ is numerical
        and $C$ is categorical
output: $R$, a set of range-based classification rules

1.  for each $c_j \in C$ do
2.      $T_{c_j} \leftarrow select(T, c_j)$
3.      $CR_1 \leftarrow \{[v_{min}, v_{max}]_{A_1}, [v_{min}, v_{max}]_{A_2}, \cdots,$
            $[v_{min}, v_{max}]_{A_m}\}$ from $T_{c_j}$
4.      for $(k = 1,\ CR_k \neq \varnothing,\ k{+}{+})$ do
5.          $LR_k \leftarrow \varnothing$
6.          for each $cr \in CR_k$ do
7.              $S \leftarrow analyse\_range\ (cr, c_j)$
8.              for each $s \in S$ do
9.                  if $\sigma(s) \geq \sigma_{min}$ and $\gamma(s) \geq \gamma_{min}$ then
10.                     $LR_k \leftarrow LR_k \cup s$
11.                     if $\delta(s \Rightarrow c_j) \geq \delta_{min}$ then
12.                         $R \leftarrow R \cup \{s \Rightarrow c_j\}$
13.         $CR_{k+1} \leftarrow generate\ (LR_k)$
14. return $R$
```

Figure 2.1: Original Phase I Analyse algorithm

Step 3 In this single step, from T_{c_j}, for each attribute A_i the algorithm derives a set of ranges $[v_{min}, v_{max}]_{A_i}$ for that attribute. These ranges are formed by seeking for the minimum and the maximum values of A_i in T_{c_j}. This is denoted as candidate ranges cr, since they are still subjected to a check to see whether they have the required minimum thresholds needed to form the rules.

Example: From Table 2.2, for each attribute a candidate range can be generated as follows: $A_1 = [0.57, 0.75]$, $A_2 = [1.21, 1.45]$, $A_3 = [2.13, 3.11]$, $A_4 = [3.97, 4.24]$.

Step 4-13 In these remaining steps lie the core analysis of Phase I, which could be separated into three main components: sub-range derivation, threshold check and larger range generation. Firstly, for each range in cr, an analysis is performed to derive possible sub ranges from this range. The reason for this analysis is because for continuous ranges, $[v_{min}, v_{max}]_{A_i}$ could cover more than necessary tuples that consequently reduces the rule accuracy if not trimmed down.

Example: Consider the example $A_1 = [0.57, 0.75]$ from the previous step. Using this range on the original set of tuples T in Table 2.1, the tuples that the range covers are:

Table 2.3: A subset of tuples under class 1 and range r of attribute 1

T	*A1*	*C*
t1	0.75	1
t2	0.64	2
t3	0.71	1
t4	0.57	1

Under this range $A_1 = [0.57, 0.75]$ which is a candidate rule, its cover of t_1, t_2, t_3, t_4 gives a $\frac{3}{4}$ correct classification. However, by adjusting the range to $A_1 = [0.68, 0.75]$ which covers t_1, t_3, t_4, it now gives $\frac{4}{4}$ correct classification. Although the support for this range is reduced, higher classification accuracy is achieved. This adjustment step lies within step 7.

Step 7 The pruning process is represented by the *analyse_range* function, where a method is implemented to find sub-ranges. This method is based on the solution to the max sum problem [SRC], which is for finding a maximal gained sub-array within a given array of discrete numbers (its details will be discussed further in the next section). To apply the similar approach to finding maximum sub-ranges, for class c_j, each tuple in T_{c_j} that is covered by range $[v_{min}, v_{max}]_{A_i}$ is scored with 1 if its class is c_j and -1 otherwise. This is to convert continuous values into discrete (in this case, binary) scores of numbers, that are then used to calculate the max sum scores. Those max sum scores represents the candidate sub-ranges, which could be the desired sub-ranges once they satisfy the support confidence and density constraints. There can be multiple sub-ranges and thus all sub-ranges are collected and store the result in set S. Sub-ranges in set S is checked in the following step 8-12 for threshold requirements.

Example: From Table 2.3, T_{A_1} where $A_1 = [0.57, 0.75]$ and the class value c_1 is 1, the tuples t_1, t_2, t_3, t_4 are covered under this rule and can be presented in ascending order as:

Assume the max-sum approach requires 100% confidence i.e. correct classification ($c = 1$) for each covered tuple, the resulted optimal binary array will be 111 which represents a

Table 2.4: A subset analysed by max-sum method for sub-ranges

	t_2	t_4	t_3	t_1
Attribute value	0.64	0.68	0.71	0.75
Class value	c_2	c_1	c_1	c_1
Binary values	-1	1	1	1

sub-range $s = [0.68, 0.75]$ that covers tuples t_4, t_3, t_1.

Step 8-10 These steps perform a support and density threshold check on sub-ranges in set S. For each sub-range s in S, a threshold check is performed to see whether it has sufficient support and density specified (Step 9). If sub-range s does, it is kept in LR_k where candidate sub-ranges are stored to form larger associated ranges in the next iteration (Step 10). However, if it does not satisfy the specified thresholds, sub-range s is discarded as it cannot be used to form larger associated ranges due to support and density measures being monotonic [4]. All checked ranges s in LR_k are then passed on for a final check on confidence threshold in the following step 11-13.

Example: From Table 2.4 where sub-range $s = [0.68, 0.75]_{A_1}$ is found for attribute A_1, the support and density can be computed as follows:

$$\sigma_s = \frac{\{t_3, t_4, t_1\}}{\{t_2, t_4, t_3, t_1\}} = \frac{3}{4} = 0.75$$

$$\gamma_s = 1$$

Assume the condition specified is $\sigma_s \geq 0.6$ and $\gamma_s \geq 0.9$, then sub-range $s = [0.68, 0.75]_{A_1}$ satisfies the required support and density threshold and therefore accepted. Assume, with the similar rule, another sub-range $s = [1.21, 1.45]_{A_2}$ is found for attribute A_2 from an initial range $[1.21, 1.45]_{A_2}$. The support and density can be computed as follows:

$$\sigma_s = \frac{\{t_3, t_4, t_1\}}{\{t_3, t_4, t_1, t_5, t_2\}} = \frac{3}{4} = 0.75$$

$$\gamma_s = 1$$

With the same support and density measures, sub-range $s = [1.21, 1.45]_{A_2}$ is also accepted. These two sub-ranges are stored in LR_k for the next steps, hence $LR_k = \{[0.68, 0.75]_{A_1}, [1.21, 1.64]_{A_2}\}$.

Step 11-12 These steps perform the last check for confidence threshold on ranges in LR_k. Using class value c_j, it checks whether sub-range s in LR_k can be used to form range-based classification rules with sufficient confidence (Step 11-12). If s satisfies the confidence test, sub-range s is then kept in LR_k which is then used to form larger association of ranges. The process of forming larger associated ranges is in Phase II (Generate) in Step 13. However, in cases where s fails to pass the confidence threshold, it is discarded and no longer considered in the next iteration.

Example: In Step 8-10 example, sub-range $s = [0.68, 0.75]_{A_1}$ is shown to have passed threshold for support and density. It is then checked for confidence measure, which is as follows:

$$\delta_s = \frac{\{t_4, t_3, t_1\}}{\{t_4, t_3, t_1\}} = 1$$

Assume the condition specified is $\delta_s \geq 0.8$, sub-range $s = [0.68, 0.75]_{A_1}$ passes the test and therefore accepted. Similarly for sub-range $s = [1.21, 1.45]_{A_2}$:

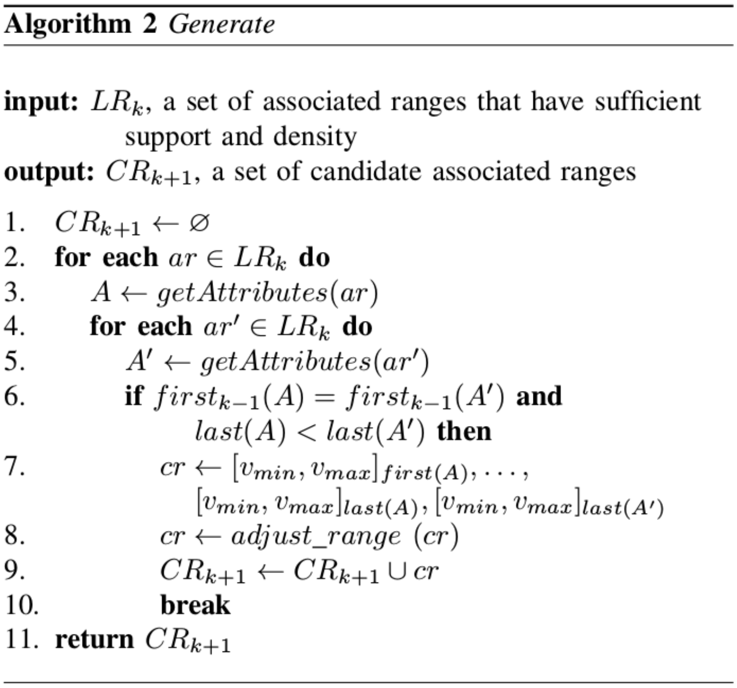
$$\delta_s = \frac{\{t_3, t_4, t_1\}}{\{t_3, t_4, t_1\}} = 1$$

Hence $LR_k = \{[0.68, 0.75]_{A_1}, [1.21, 1.64]_{A_2}\}$ is accepted and passed onto Phase II (Generate) in Step 13 to form larger associated ranges.

Step 13 This step, which represents Phase II (Generate), perform the generation of larger associations of ranges CR_{k+1} by a generate function. The process is the core in Phase II (Generate) is described in Phase II sub-section below.

2.3.2 Original Phase II Generate

In this phase, the algorithm prepares CR_{k+1} for the next iteration by combining two relevant ranges in LR_k, thus extending the current ranges. The combining process mirrors the A-priori algorithm in generating frequent itemsets in association rule mining. Although A-priori applies on categorical values, it is adapted to this range-based classification algorithm by using an additional step on range adjustment. The algorithm is denoted in Figure 2.2 below:



Algorithm 2 *Generate*

```
input: $LR_k$, a set of associated ranges that have sufficient
          support and density
output: $CR_{k+1}$, a set of candidate associated ranges
1.  $CR_{k+1} \leftarrow \varnothing$
2.  for each $ar \in LR_k$ do
3.      $A \leftarrow getAttributes(ar)$
4.      for each $ar' \in LR_k$ do
5.          $A' \leftarrow getAttributes(ar')$
6.          if $first_{k-1}(A) = first_{k-1}(A')$ and
                $last(A) < last(A')$ then
7.              $cr \leftarrow [v_{min}, v_{max}]_{first(A)}, \dots,$
                $[v_{min}, v_{max}]_{last(A)}, [v_{min}, v_{max}]_{last(A')}$
8.              $cr \leftarrow adjust\_range\ (cr)$
9.              $CR_{k+1} \leftarrow CR_{k+1} \cup cr$
10.             break
11. return $CR_{k+1}$
```

Figure 2.2: Original Phase II Generate algorithm

Step 1-5 Given LR_k from previous phase, these steps traverses through all sets of associated ranges in LR_k and for each set of associated ranges ar, it looks for another remaining set ar as a possible set to combine to make a larger range. The condition to combine is in the following Step 6-7.

Step 6-7 These steps decide whether ar and ar can be combined by searching whether ar and ar differs by only one last attribute. $first_{k-1}$ is a function to retrieve the first attribute in each set, and $last$ is a function to retrieve the last attribute in the same set. So if the last function of ar returns an attribute that is larger than the that which is returned by the last function of ar, this pair is accepted. A new extended set of associated range is formed based on this pair, in which the starting range is the same $first_{k-1}$ range and the ending range is the larger *last* range between the two ranges (Step 7).

Example: From Phase I example, $LR_k = \{[0.68, 0.75]_{A_1}, [1.21, 1.64]_{A_2}\}$ is found. Assume another $LR_k = \{[0.68, 0.75]_{A_1}, [2.75, 3.11]_{A_3}\}$ is found. A pair between these *LRs* can be formed since $A_1 = A_1$ and $A_3 > A_2$. A new extended set of assciated ranges *cr* is $cr = \{[0.68, 0.75]_{A_1}, [1.21, 1.64]_{A_2}, [2.75, 3.11]_{A_3}\}$

However, before this set is passed onto the next iteration, an adjustment is necessary since similar issue is encountered from Phase I : A combination of two continuous ranges may cover more than necessary set of tuples. It occurs when a tuple might be covered in range 1, but not covered in range 2 and vice versa. As the new extended set is passed onto the next iteration, those tuples that fail to be covered in all specified ranges shall be set aside. In this algorithm, the algorithm preserves a strict policy of selection: it presumes those tuples would produce wrong classification at the end, thus will change their classes to $Null$ which signifies a possible eliminiation on the next iteration (recall maxsum process in Phase I where it might eliminates tuples that are not under the same class value to trim down the candidate range).

Example: An extended set of associated ranges $cr = \{[0.68, 0.75]_{A_1}, [1.21, 1.64]_{A_2}, [2.75, 3.11]_{A_3}\}$ is found from the last step. This set *cr* covers tuples under A_1, A_2, A_3 respectively as:

$$cr = \{[0.68, 0.75]_{A_1}, [1.21, 1.64]_{A_2}, [2.75, 3.11]_{A_3}\}$$

$$\{t_4, t_3, t_1\}\{t_3, t_4, t_1\}\{t_4, t_3\}$$

It can be seen that tuple t_1 is not covered under A_3 range.Initially, t_1 has class c_1, but since it was not covered completely under this set, the algorithm will convert c_1 to $Null$ in t_1. From Table 2.1, t_1 is now presented as:

Its class will be kept for all iterations under class c_1.

Once Phase II (Generate) is completed, it repeats Phase I (Analyse) but on the new extended set of associated ranges instead. The iteration continues until no possible pair

Table 2.5: A tuple turned Null in class value during generate phase

T_{c_j}	A_1	A_2	A_3	A_4	C_j
t1	0.75	1.45	2.13	4.56	Null

can be formed, which is when the algorithm is concluded. The remaining pair become the classification rule for that class.

However, it must be noted that this null-class converting process is replaced in this project algorithm, as this project adopts a stricter policy that eliminates partly-covered tuples instead of leaving it for consideration in the next iteration. Explanation and reasoning for such approach is discussed in the next section where the project algorithm is reviewed.

Section 3

Project Algorithm

Based on the original algorithm discussed in the previous section, the initial plan for the project is to implement the exact algorithm into code. However, during the implementation process, several adjustments were agreed to be made. Some adjustments (e.g additional steps in scanning data input, display final rules to screen) are for pre-processing data for input and output, at beginning and end respectively. Others (e.g max-sum, threshold check, null-class converter) are for possible improvements in algorithmic in the programme.

Respectively, for the technical-related changes, section 4 will discuss them in depth where the actual code implementation is concerned. For the algorithmic changes, they are to be discussed in the remaining sections below.

3.1 Similarities

As the project‘s main goal is to implement the algorithm, the programme structure is entirely similar to the initial algorithm i.e. Phase I (Analyse) and Phase II (Generate) is kept in order. Details are explained in the following sub-sections.

3.1.1 Project Phase I Analyse

Phase I contains the range analysis method (max-sum) and the threshold checker (for support, density and confidence). In this project, similar steps are implemented: Firstly, the table of data is divided into sub-data by class value (step 1-2). Secondly, within this sub-data, a $[v_{min}, v_{max}]$

range is derived for each of the attribute (step 3). Thirdly, for each of these ranges within an attribute, a max-sum analysis is conducted so that sub-ranges can be deduced from the original range (step 4-7). Finally, each of these sub-ranges is then checked to see whether it passes the specified thresholds of support, confidence and density (step 8-12). The accepted sub-ranges are collected and sent to Phase II (Generate) (step 13).

However, in this project, several steps are done differently in terms of algorithm compared to the initial algorithm. Firstly, the max-sum analysis is adapted to fit in this range-based classification problem. While it was not discussed in details in the initial paper, there is a difference in this max-sum compared to the original max-sum solution. That is, this max-sum method will not seek for one maximal array but multiple ones depending on a set of pre-specified conditions. Secondly, in the threshold checking part, the order of threshold tested are no longer support - density - confidence, but support - confidence for first iteration and density - support - confidence for the second iteration onwards. It is to adapt to the different structural as well as technical requirements of different iteration e.g. density is not relevant in the first iteration where no extended ranges have been made yet. Details are to be discussed in the next section.

3.1.2 Project Phase II Generate

Phase II contains the range combination method (a-priori) and the additional adjustment step (null-class converter). In this project, the first method is implemented but the second part is replaced by an elimination step (eliminator): Firstly, the accepted sub-ranges from Phase I are considered and grouped with each other under the condition that one of them has an extendable range (step 1-7). Secondly, according to the original algorithm, an adjustment method is required by changing the class value of partly-covered ranges to Null (step 8).

However, instead of turning tuple's class value to *Null* only, this project adopts a stricter policy: all partly-covered ranges are eliminated and not considered for the next iteration. It is due to the idea taken from the paper that those tuples are deemed to badly influence the rule accuracy in the long run. While they might as well get eliminated in the next max-sum iteration, the algorithm in this project does not at all consider those tuples to save on processing time and enhance the rule accuracy in exchange for less support. Hence the null-class converting process (step 8) is replaced with an elimination process. Details will be discussed in the next section.

3.2 Differences

As mentioned in the previous section, there are two main algorithmic changes in this project compared to the initial algorithm:

3.2.1 Project Phase I Analyse

- Max-sum method → maxsum() in detail

  Original version: Max-sum on multiple ranges - not elaborated

  Project version: Max-sum on multiple ranges - elaborated

- Threshold checking method → two separated check()

  Original version: Threshold check consists of three orderly checks: support-density and confidence

  Project version: Threshold check consists of two orderly checks for first iteration and three orderly checks for second iteration onwards. The order of each threshold check is also altered (for first iteration: support-confidence; for second iteration: density-support-confidence)

3.2.2 Project Phase II Generate

- Null-class converting method → adjust()

  Original version: a conversion where partly covered tuples have their classes change to Null

  Project version: a strict adjustment where partly covered tuples are complelely removed and not considered during the next iteration

The differences can be seen in the project algorithm in the next section.

3.3 Final Construction

Algorithm 3.1 shows the Project version of the original algorithm, beginning with Phase I Analyse:

Algorithm 3.1: Project Algorithm Phase I Analyse

```
     Input  : $T(A_1, A_2, A_3, ..A_m, C)$, where $A_{j,m}$ as $1 \leq j \leq m$ is numerical and $C$ is
              categorical.
     Output : $R$, a set of range-based classification rules.
 1   foreach $c_j \in C$ do
 2       $CR_1 \leftarrow [v_{min}, v_{max}]_{A_1}, [v_{min}, v_{max}]_{A_2}, ...[v_{min}, v_{max}]_{A_m}$ from $T_{c_j}$
 3       for $(k = 1, CR_k \neq \varnothing, k++)$ do
 4           $LR_k \leftarrow \varnothing$
 5           foreach $cr \in CR_k$ do
 6               $S \leftarrow max\_sum(cr, c_j)$
 7               foreach $s \in S$ do
 8                   if $\sigma(s) \geq \sigma_{min}$ and $\gamma(s) \geq \gamma_{min}$ then
 9                       if $k = 1$ then $LR_k \leftarrow LR_k \cup s$
10                       else if $\delta(s \Rightarrow c_j) \geq \delta_{min}$ then $LR_k \leftarrow LR_k \cup (s \Rightarrow c_j)$
11
12                   end
13               end
14           end
15           $CR_{k+1} \leftarrow generate(LR_k)$
16       end
17   end
18   return $R$
```

The differences can be seen in Step 6 (*max_sum* method) and Step 8-10, where a conditional clause $k = 1$ is used to separate the iteration: only in iteration 2 onwards would density check occurs.

Algorithm 3.2 shows the Project version of Phase II Generate:

Algorithm 3.2: Project Algorithm Phase II Generate

```
    Input  : $LR_k, asetofassociatedrangesthatareacceptedfromlastphase.$
    Output : $CR_{k+1}$, a set of candidate associated ranges.
1   $CR_{k+1} \leftarrow \varnothing$
2   foreach $ar \in LR_k$ do
3       $A \leftarrow getAttributes(ar)$
4       foreach $ar' \in LR_k$ do
5           $A' \leftarrow getAttributes(ar')$
6           if $first_{k-1}(A) = first_{k-1}(A')$ and $last(A) < last(A')$ then
7               $cr \leftarrow [v_{min}, v_{max}]_{first(A)}, ..., [v_{min}, v_{max}]_{last(A)}, [v_{min}, v_{max}]_{last(A')}$
8               $cr \leftarrow strict\_adjust(cr)$ $CR_{k+1} \leftarrow CR_{k+1} \cup cr$
9           end
10      end
11  end
12  return $CR_{k+1}$
```

The differences can be seen in step 8, where a strict-adjust method is applied instead of the originally method.

Section 4

Implementation

4.1 Structure

4.1.1 Overview

Based on the project algorithm design, its Java implementation unit complies with the specified structure and thus maintain the two core phases (*Analyse* and *Generate*). However, hard code requires extra handling of data input and output. Hence additional two phases are added: *Scan* and $Monitor$, which read and display input respectively. Moreover, the differences in data type of different iterations require extra treatment would be too clunky to append to the *Analyse* phase. Thus, another phase is in line: $Sub - Analyse$, which inherits *Analyse* range-analysing functions while having additional threshold check (for density) and handle elimination round for partly-covered tuples.

The implementation process can now be divided into five Phases instead of two from the original algorithm. The reasoning is assessed in details below:

Table 4.1: 5-Phase Implementation and Reasoning

Reason	**Outcome**
To read and pre-process data input before analysis	Phase I Scan
To comply with the original structure	Phase II Analyse
To comply with the original structure	Phase II Generate
To adapt to different iteration needs during data analysing process	Phase IV Sub-Analyse
To monitor and record the result	Phase V Monitor

From Table 4.1, a detailed structure for the programme can be presented in an Activity Diagram below:

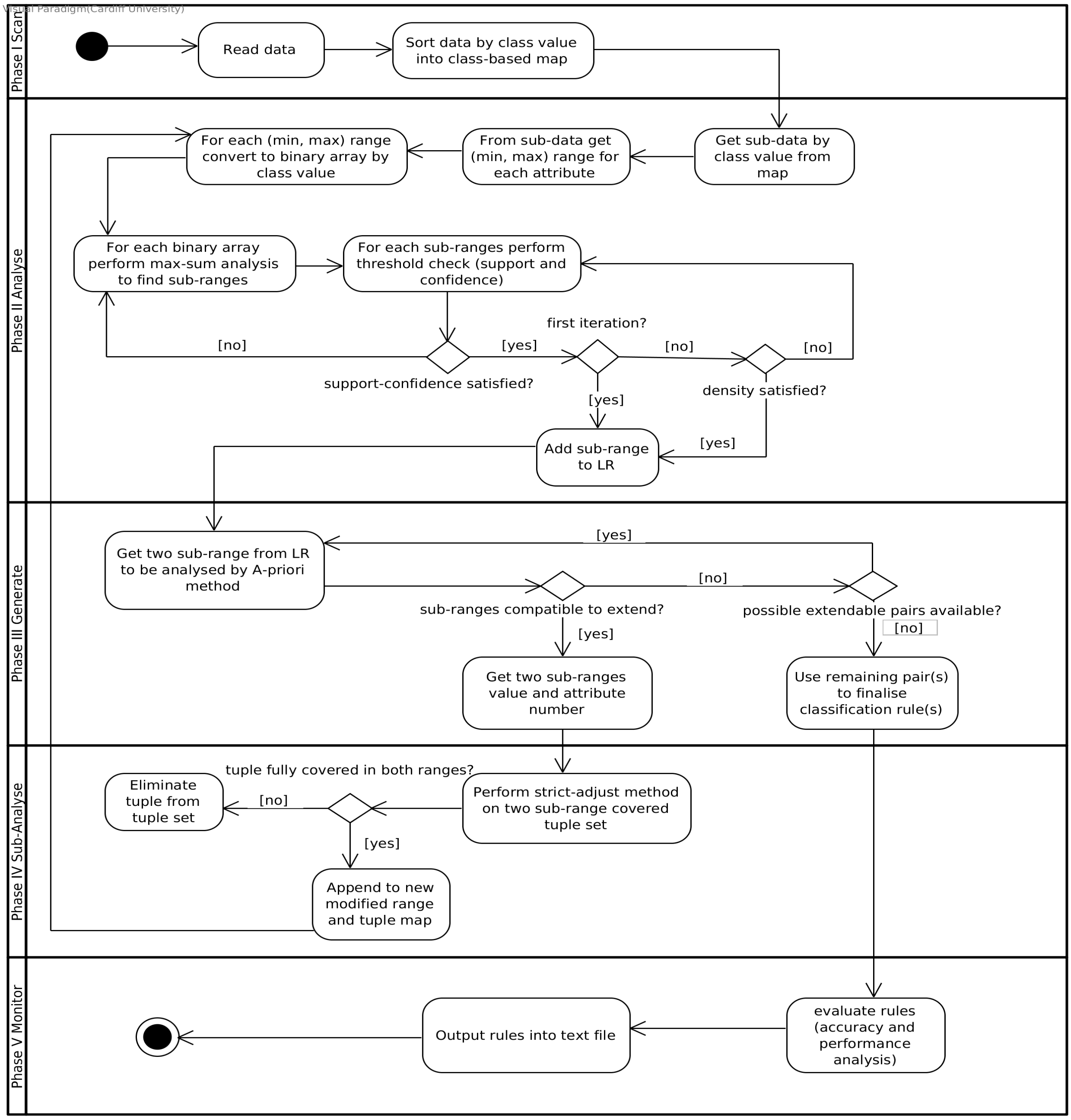

Visual Paradigm(Cardiff University)
Phase I Scan
Read data
Sort data by class value into class-based map
Phase II Analyse
Get sub-data by class value from map
From sub-data get (min, max) range for each attribute
For each (min, max) range convert to binary array by class value
For each binary array perform max-sum analysis to find sub-ranges
For each sub-ranges perform threshold check (support and confidence)
support-confidence satisfied?
[no]
[yes]
first iteration?
[no]
[yes]
density satisfied?
[no]
[yes]
Add sub-range to LR
Phase III Generate
Get two sub-range from LR to be analysed by A-priori method
[yes]
sub-ranges compatible to extend?
[no]
[yes]
possible extendable pairs available?
[no]
Get two sub-ranges value and attribute number
Use remaining pair(s) to finalise classification rule(s)
Phase IV Sub-Analyse
tuple fully covered in both ranges?
Perform strict-adjust method on two sub-range covered tuple set
Eliminate tuple from tuple set
[no]
[yes]
Append to new modified range and tuple map
Phase V Monitor
evaluate rules (accuracy and performance analysis)
Output rules into text file

Figure 4.1: Class Overview: Activity Diagram

To demonstrate the structure graphically, a Class Diagram and a Sequence Diagram is shown for each phase, that is represented by each class. Class Diagram is chosen as a common representation for an overview within and between classes [5]. A Sequence Diagram is also chosen against Activity Diagram since it shows the recursive calls in different iterations in depth.

4.1.2 Phase I Scan

Aim: There are three aims within this phase or class:

- To read data from `.csv` file
- To sort data into sub-data pools using class values found when reading data in
- To pass these data into the next phase or class (*Analyser*

Input: This class input is the data itself:

- Pre-processed .csv file (no null values within attributes and the order of element in an instance is attribute 1, attribute 2,..., attribute N, class C

Reason: Initial plan included an extra processing unit for filtering csv file when value is Null. However, time constraint did not allow the construction of this unit. Hence the programme only consider pre-processed csv data at the time the report is written.

Output: This class output are data taken from input source:

- `AllTuples`: A list of all instances of the data - a list of tuples that are presented as lists.
- `AllClassMap`: A partitioned map that contains class values as keys that are connected to their relevant sub-data (list of instances) as values.

Reason: Data taken from input sources need separate containers for different purpose: `AllTuples` is required to provide access to all original data. On a different note, `AllClassMap` is required to get the class-based data partition.

Data type: `LinkedList` (for tuples), `LinkedHashMap` (for class-based data partitioned map)

Reason: A `List` - without specified inner data type - is used for tuples since it needs multiple types of data (e.g. float number, string class value, integer ordered number). In this case, `LinkedList` is used instead of `ArrayList` for two reasons: Firsrly, its ability to add and remove first and last elements more quickly than the alternative. It is necessary to retrieve class value in each tuple at faster rate as class value is the last element in the tuple. Secondly, `LinkedList` or any type of `Linked`-collection type means its A set of tuples thus casts as `LinkedList<LinkedList>`.

Similarly, a `LinkedHashMap` is used for faster retrieval rate an. A `HashMap` is used for storing partitioned sub-data from class values since each partition of data needs to be recorded with its unique class value. It has String class as key and `LinkedList<LinkedList>` List of all tuples as values.

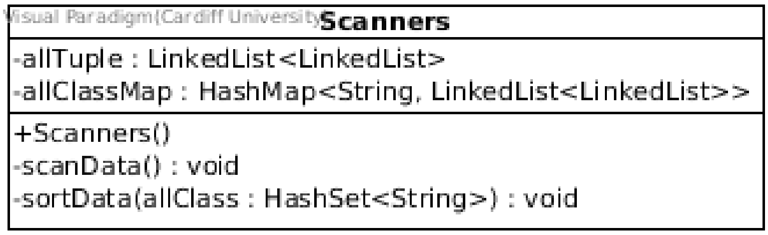

Scanners
-allTuple : LinkedList<LinkedList>
-allClassMap : HashMap<String, LinkedList<LinkedList>>
+Scanners()
-scanData() : void
-sortData(allClass : HashSet<String>) : void

Figure 4.2: Class Diagram for class Scanners

Class diagram: Its class diagram is in Figure 4.2, showing two features/methods:

- scanData()

  This method initialises two scanners, one to scan line-by-line data from csv file (i.e. tuples) and one to scan in-line elements (i.e. attributes). It processes data into 1) allTuple (where all instances are recorded from the first scanner) and 2) Class values, Tuple order numbering and Tuple attributes (which is sorted by the second scanners). The Class values become the parameter for the next method, where it helps create the Map that connects Class value to relevant set of Tuples.

- sortData()

  This method groups sets of tuples according to their Class value by checking their last String element. It sorts and partitions allTuple into sub-data pools, which are then passed on as parameter to the new class (*Analyser*).

Sequence diagram: Figure 4.3 shows the sequence diagram for *Scanners*, showing the data flow from *Scanner* → *Analyser*:

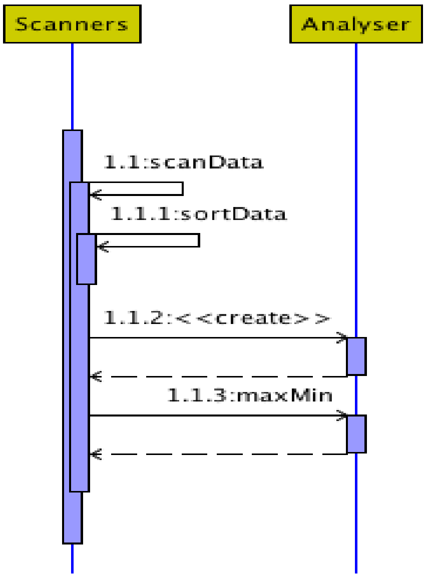


Figure 4.3: Sequence Diagram for class Scanners

4.1.3 Phase II Analyse

Aim: Being the core module of the programme, there are important aims for this phase or class:

- To find (min, max) range for each attribute from class-based map.
- To convert (min, max) range to binary array, which is ready for max-sum analysis.
- To perform max-sum analysis on (min, max), aim to obtain at least one sub-optimal range, which is ready for the threshold test.
- To perform threshold check. As discussed, this check diverses depending on which iteration. However, support and confidence is always required thus appear in this phase or class.
- To modify the new sub-data pools for the next iteration. Since data set would shrink after pruning process from max-sum and (if applicable) strict-adjust method, current data set is frequently changing. Threshold measures, however, are calculated depending on these sets, thus makes this modification step necessary.

- To pass the modified sub-data pools as well as new sub-ranges to the next phase or class (*Generator*).

Input: This class input is the original dataset and the class-based data map from passed on from previous class:

- `allTuple`: all original instances in dataset.
- `allClassMap`: with key as class tag and value as relevant sub-data partition.

Reason: `allClassMap` is needed for the class-guided search for the rest of the programme. This map, however, changes as the sub-data pool changes throughout the cycle. Therefore, `allTuple` is needed to refer back to the original, pre-modification input when necessary.

Output: A list of sub-ranges in float values that passed the support-confidence check (`attributeRangeList`). It also returns a modified version of allClassMap, which `AllClassMap` is now divided into two maps: *attributeRange* and *attributeTuple*.

Reason: As required for the next *Generate* phase, `attributeRangeList` stores potential associated ranges for new combination. Map, however, diversifies into two:

- One for storing sub-ranges (that are also stored in `attributeRangeList`)for each attributes. That means `attributeRangeMap` has its key quantity equal to the number of attributes, and each key maps to at least one sub-range found from analysing methods.
- One for storing tuples that each sub-range in `attributeRangeMap` covers. That means `attributeRangeMap` has its key quantity equal to the number of sub-ranges available (that are also recorded in `attributeRangeList`), and each key maps to a list of tuples that is covered by that range.

  Initially, only `attributeRangeMap` was implemented as it was thought that only attribute and its sub-ranges were necessary to be recorded formally. Tuples covered by sub-ranges were thought to be retrievable by tracing back to the original `AllClassMap`.

  However, as data was partitioned further the longer the algorithm persisted, there was no fixed map that remained relevant for the entire duration of the programme.

With the dynamic nature of tuple sets, it was safer to recorded them right when their sub-ranges were being processed. Thus there was a need for the additional `attributeTupleMap`.

Data type: For `attributeRangeMap`, a `LinkedHashMap` with `LinkedList<Integer>` as keys and `LinkedList<LinkedList<Float[]»` as values is used. This map stores attribute number as key (thus `Integer` type cast) and ranges as value (thus `LinkedList<Float>` type cast). It must be noted that despite attribute number being a singular integer (from 1 to n), `LinkedList` is implemented since attributes' ranges are combined at Generate phase. That means attribute number must be expandable to manage this new combination (e.g attribute 1 and 2 has associated ranges of $[1.1, 1.5]$ and $[2.2, 2.6]$. Once these two ranges are combined to form larger ranges, attribute numbers are also combined i.e. to $1, 2$ to manage this combined range in the map) - hence need for a `List`. Likewise, `LinkedList<Float>` is sufficient to manage a single range; when there are multiple ranges to be stored, an outer `LinkedList` must be used.

Similarly, For `attributeTupleMap`, a `LinkedHashMap` with `LinkedList<LinkedList<Float[]»` as keys and `LinkedList<LinkedList>` as values. Keys in `attributeTupleMap` are `attributeRangeMap` values; the cross-reference makes for a convenient key retrieval between two maps.

Class diagram: Its class diagram is in Figure 4.3, which additionally shows a parentage link to *SubAnalyser* class for Phase IV $Sub - Analyse$. It is the only instance of inheritence in the programme: *Analyser* is used in the first iteration and its framework is re-used for code efficiency in $Sub - Analyser$ for second iteration onwards. For this subsection, only *Analyser* is discussed; *SubAnalyser* is to be explained in the next subsection.

Class *Analyser* contains seven features/methods:

- maxMin()

  This method is responsible for collecting (min, max) ranges for each attribute. For each of the Class value in the map, it goes through each attribute value of every corresponding tuples. A `Collections.sort()` method is applied to find the minimum and maximum value of each attribute. In the first iteration, only one range is found for each attribute.

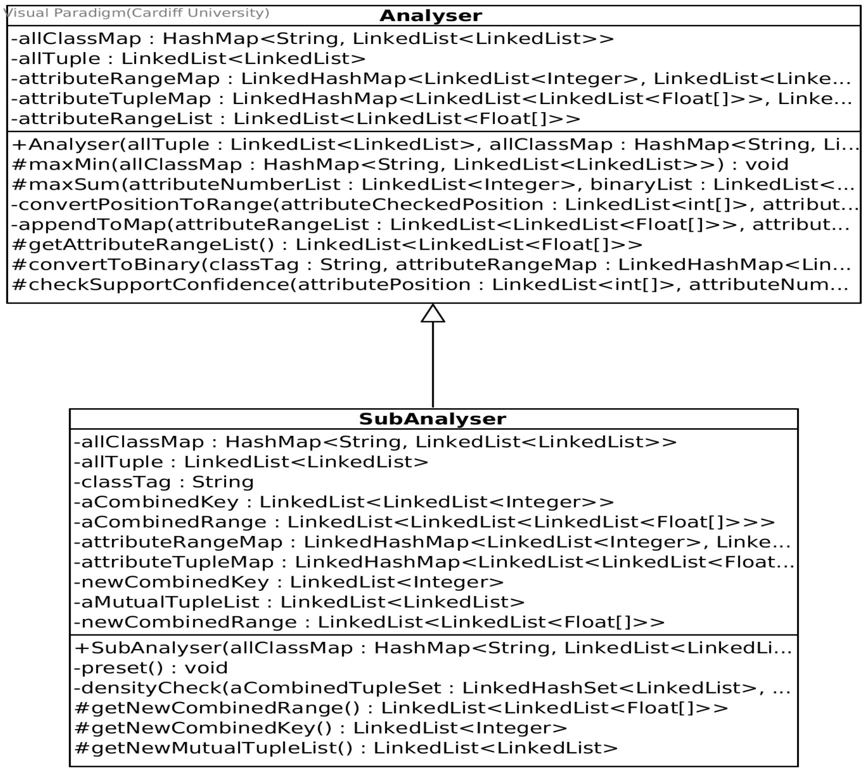

Visual Paradigm(Cardiff University)
Analyser
-allClassMap : HashMap<String, LinkedList<LinkedList>>
-allTuple : LinkedList<LinkedList>
-attributeRangeMap : LinkedHashMap<LinkedList<Integer>, LinkedList<Linke...
-attributeTupleMap : LinkedHashMap<LinkedList<LinkedList<Float[]>>, Linke...
-attributeRangeList : LinkedList<LinkedList<Float[]>>
+Analyser(allTuple : LinkedList<LinkedList>, allClassMap : HashMap<String, Li...
#maxMin(allClassMap : HashMap<String, LinkedList<LinkedList>>) : void
#maxSum(attributeNumberList : LinkedList<Integer>, binaryList : LinkedList<...
-convertPositionToRange(attributeCheckedPosition : LinkedList<int[]>, attribut...
-appendToMap(attributeRangeList : LinkedList<LinkedList<Float[]>>, attribut...
#getAttributeRangeList() : LinkedList<LinkedList<Float[]>>
#convertToBinary(classTag : String, attributeRangeMap : LinkedHashMap<Lin...
#checkSupportConfidence(attributePosition : LinkedList<int[]>, attributeNum...
SubAnalyser
-allClassMap : HashMap<String, LinkedList<LinkedList>>
-allTuple : LinkedList<LinkedList>
-classTag : String
-aCombinedKey : LinkedList<LinkedList<Integer>>
-aCombinedRange : LinkedList<LinkedList<LinkedList<Float[]>>>
-attributeRangeMap : LinkedHashMap<LinkedList<Integer>, Linke...
-attributeTupleMap : LinkedHashMap<LinkedList<LinkedList<Float...
-newCombinedKey : LinkedList<Integer>
-aMutualTupleList : LinkedList<LinkedList>
-newCombinedRange : LinkedList<LinkedList<Float[]>>
+SubAnalyser(allClassMap : HashMap<String, LinkedList<LinkedLi...
-preset() : void
-densityCheck(aCombinedTupleSet : LinkedHashSet<LinkedList>, ...
#getNewCombinedRange() : LinkedList<LinkedList<Float[]>>
#getNewCombinedKey() : LinkedList<Integer>
#getNewMutualTupleList() : LinkedList<LinkedList>

Figure 4.4: Class Diagram for class Analyser

For each range, it then appends attribute number, range to `attributeRangeMap`. It also retrieves tuples that are covered under this range and appends range, range-covered tuple to `attributeTupleMap`.

These two maps, accompanied by the Class value, are passed onto the next method, `convertToBinary` to prepare binary array for max-sum analysis.

- convertToBinary()

  For each attribute in `attributeRangeMap`, this method retrieves the respective (min, max) range. Each range is used as key to search in `attributeTupleMap` to obtain tuples that are covered by this range. It then goes through each tuple to retrieve its Class tag and if this tag matches the Class value received from previous method, it appends 1 to a new `binaryList`, otherwise appends -1.

  This binary list is sent to the next method `maxSum()` for sub-range analysis.

- maxSum()

For each of the binary list, this method performs a similar analysis to Kadane's max-sum algorithm [SRC]: Starting with a `sum` of 0, it traverses through the list values while adding this value to the initial `sum`, also keeping track of the element position in an `attributePosition` array. If `sum` is positive, it moves on while taking the same action. However, if it gets negative, it resets the `sum` to 0 and also restart the `attributePosition` array. This essentially means that the current array contains too many wrong class tag if continued (i.e. too many -1s), and thus not necessary to proceed into more negatives.

However, while a new `attributePosition` array is created starting from this position, the current array is recorded instead of discarded, which is also considered as a sub-range ready for next iteration. This step differentiates this method from the original Kadane's since Kadane's seeks for only one optimal range, constrasting to this method where multiple sub-ranges are desired.

The process is repeated until the end of array is reached. The outcome can be 1) no `attributePosition` array or sub-ranges found, 2) one `attributePosition` array or sub-range found, or 3) multiple `attributePosition` array or sub-ranges found. Regardless, if available, sub-ranges (in `attributePosition` array format) are recorded into an `List` and passed onto next method, `checkSupportConfidence()`, for threshold test.

It must be noted that, initially, whenever a `attributePosition` array is found, `checkSupportConfidence()` is called straight away so a threshold test is performed whenever `sum` turns negative. However, it means that `checkSupportConfidence()` might be called excessively in the case of a negative streak for `sum`, thus might be safer to call it after max-sum is finished.

- checkSupportConfidence()

  For each position array in `attributePosition`, support and confidence is measured:

  Support σ = array size / binary list size

  Confidence δ = number of positive occurence within array / array size

Array size is calculated by the subtraction of last and first position e.g for position array $[10, 21]$, the array size is $21-10 = 11$. Positive occurence is counted by traversing through binary values that are covered under this position range e.g for position array $[10, 21]$, the programme retrieves value of element number [10], [11],...,[21] in binary list and counts the quantity of 1s against -1.

These measures are compared to their corresponding user-specified thresholds in a conditional clause. If an array passes both thresholds, it is accepted as a candidate sub-range for next iteration. Accepted arrays are added into a new `list` to be passed onto the next method, `convertPositionToRange` where it is converted from integer position to its respective float range values.

(Initially, no new `List` was needed as a loop was constructed so that rejected arrays were removed from the list instead of adding the accepted ones into new list. However, Java ejected `ConcurrentModificationException` as it prohibited modification when a list is in iteration, hence the change.)

- convertPositionToRange()

  For each accepted position array in the new textttattributePosition, this method converts position (integer) to its relevant range value (float). It uses `attribtuteTupleMap` to retrieve the tuples covered under this range, then seeks for the two tuples that lies at specified position. For instance, a position array $[10, 21]$ represents the attribute value of tuples at position [10] and [21] in the relevant tuple set. Within these two tuples, attribute value is reclaimed i.e. tuple $t_1 = 0, 1.2, 1.5, 2.3, class1$ returns attribute 1 value as 1.2. Thus $[10, 21]$ is converted to range e.g $[1.2, 2.4]$ with this method.

  Once done, the converted ranges are recorded into an `attributeRangeList`. If it is in the first iteration, this list is passed onto the `appendToMap()` method in order to modify the `attributeRangeMap` and `attributeTupleMap` for the next iteration. However, if it is second iteration or above, nothing is done further although these ranges can be retrieved by a `get()` method to be sent to the next iteration. This is because for the first iteration, all attributes are processed at once and map is updated completely after loop. However, for the second iteration onwards, only one (combined) attribute is considered for each loop, thus the map is not completely changed so no need for modification.

- appendToMap()

  This method, while only applies to the first iteration, is responsible for updating the `attributeTupleMap` and `attributeRangeMap` for the next iteration by 1) new sub-range and 2) new sub-range covered tuple set. In details, for each (converted) sub-range in `attributeRangeList`, it obtains a new sub-set of tuples that are covered by this sub-range. This sub-set of tuples are cut out from `attributeTupleMap` by using a loop that searches for tuples with attributes falling under this range. The new covered tuple set is appended to `attributeTupleMap`. The sub-range itself is also appended to `attributeRangeMap`.

- getAttributeRangeList()

  This method is a `get()` method to grab `attributeRangeList` directly to the next iteration (only used for second iteration onwards where `appendToMap()` is skipped).

Sequence diagram: Below is a Sequence diagram to present data flow from *Scanners* → *Analyser* → *Generator*:

4.1.4 Phase III Generate

Aim: As another core module of the programme, this class concerns with the following aims:

- To combined sub-ranges found in the previous module and form larger associated ranges, by a method resembles A-priori.
- To check if one sub-range is extendable with another by checking its attribute value.
- To check if a candidate combination of two sub-ranges can satisfies support-confidence and extra density threshold since multiple ranges are involved.
- To finalise the rule if no possible combination can be made i.e use the last associated ranges as final rule and send them to the last phase $Monitor$.

Input: This class input is the updated range and tuple map from the previous class and the

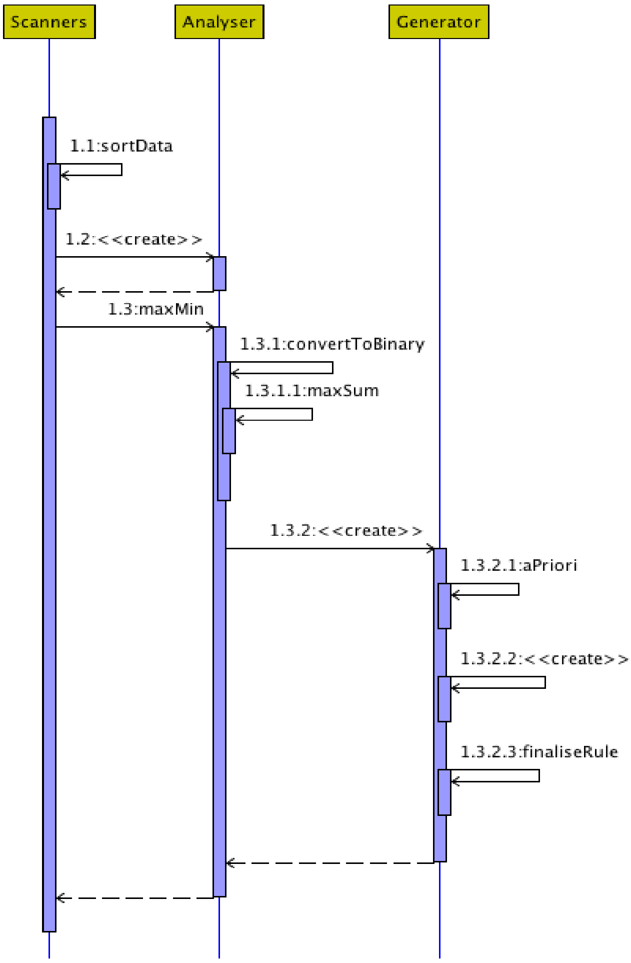


Figure 4.5: Sequence Diagram for class Analyser

original tuple set as reference when applicable:

- `allTuple`: all original instances in dataset.
- `attributeRangeMap`: with attribute number as kes and new (one or more) sub-ranges as value.
- `attributeTupleMap`: with each sub-range as key and tuple set covered under sub-range as value.

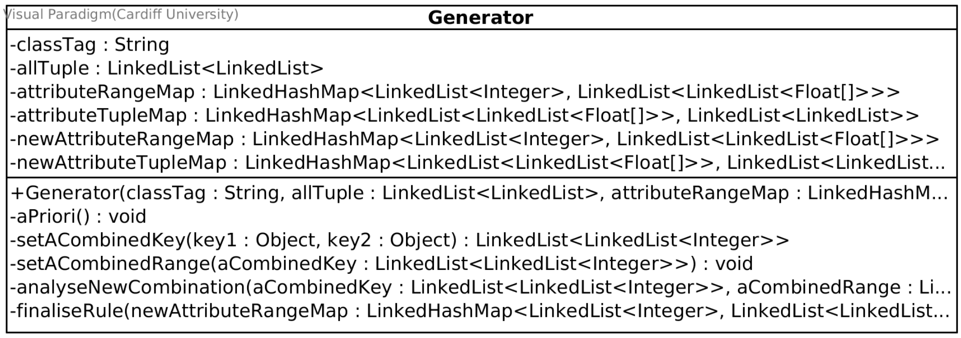
Reason: `allTuple` is needed to be passed onto the next *SubAnalyser* class as reference to the original dataset when applicable. `attributeRangeMap` and `attributeTupleMap` are needed as new data point for pairing associated ranges.

Output: Firstly, *Generator* generates new extended associated ranges and passes them as output to the next *SubAnalyser* class in order to do range analysis again (similar to *Analyser* content). Secondly, as long as there is possible pairing between associated ranges, *Generator* operates a recursive call to itself that keeps generating larger associated ranges. However, if no combination is possible, *Generator* ends the recursive call and move onto finalising the rule by calling its `finaliseRule()` method.

Reason: As *SubAnalyser* is essentially *Analyser* with additional density checking method, *Generator* can repeat an iteration where 1) new sub-ranges are merged to form candidate associated ranges and 2) candidate ranges are then checked for threshold measures and if they pass the test, sub-ranges are sent to *Generator* for another recursive call to repeat the whole iteration until no combination is possible.

Data type: Uniformly, the new `attributeRangeMap` and `attributeTupleMap` remains to be `LinkedHashMap` with similar structure to that in *Analyser* so that it could be recognised by Java as a compatible parameter in the next iteration.

Moreover, the `LinkedList` for attribute number and float range helps improve the efficiency in extending the new element, by simply adding new element to the end of array which is the strongest aspect of `LinkedList`.



```
Visual Paradigm(Cardiff University)
Generator
-classTag : String
-allTuple : LinkedList<LinkedList>
-attributeRangeMap : LinkedHashMap<LinkedList<Integer>, LinkedList<LinkedList<Float[]>>>
-attributeTupleMap : LinkedHashMap<LinkedList<LinkedList<Float[]>>, LinkedList<LinkedList>>
-newAttributeRangeMap : LinkedHashMap<LinkedList<Integer>, LinkedList<LinkedList<Float[]>>>
-newAttributeTupleMap : LinkedHashMap<LinkedList<LinkedList<Float[]>>, LinkedList<LinkedList...
+Generator(classTag : String, allTuple : LinkedList<LinkedList>, attributeRangeMap : LinkedHashM...
-aPriori() : void
-setACombinedKey(key1 : Object, key2 : Object) : LinkedList<LinkedList<Integer>>
-setACombinedRange(aCombinedKey : LinkedList<LinkedList<Integer>>) : void
-analyseNewCombination(aCombinedKey : LinkedList<LinkedList<Integer>>, aCombinedRange : Li...
-finaliseRule(newAttributeRangeMap : LinkedHashMap<LinkedList<Integer>, LinkedList<LinkedList...
```

Figure 4.6: Class Diagram for class Generator

Class diagram: Its class diagram is in Figure 4.4, which presents five features/methods:

- aPriori()

  This method combines sub-ranges by going through each range in `attributeRangeMap` and attempts to combine it with another among the remaining ranges. When a combination is planned, the next step is to get a new attribute number combination (combinedKey) and its respective sub-range combination (combinedRange).

- setACombinedKey()

  This method sets up the combination of attribute number e.g if attribute 1 and 2 has two ranges that are in pairing, a new combined key is generated to be $[1, 2]$.

- setACombinedRange()

  This method sets up the combination of range e.g if attribute 1 has range $[1.2, 2.4]$ that is in pairing with range $[3.4, 3.7]$ from attribute 2, then a new combined ranage is generated to be $[1.2, 2.4], [3.4, 3.7]$.

  Initially, this method is merged with `setACombinedKey` method; however, it is separated for readability.

- analyseNewCombination()

  Once new combined key and range are generated, this method sends both to the next *SubAnalyser* class to observe whether the new (candidate) combined range could satisfies specified conditions for the next iteration. If it does, the range is returned by a `get()` method within *SubAnalyser* to this method.

  If the returned range is *Null*, it means threshold was not satisfied thus the programme returns to `aPriori()` method to seek for new combination. However, if there is (one or more) returned range from *SubAnalyse*, this method will take the combined range and key to append it to map (similar method to `appendToMap`). A `newAttributeRangeMap` and `newAttributeTupleMap` will replace `attributeRangeMap` and `attributeTupleMap` in the last iteration respectively.

- finaliseRule()

  This method occurs only when there is no possible pairing left. A conditional clause in the *Generator* constructor calls this method when there is only one attribute number left in `newAttributeRangeMap`. Essentially, it means all keys are combined into one largest key combination and thus no other combination is possible. It acts as a stopping point to recursive *Generator* calls and procees to send the final combination as classification rules to the next class *Monitor* for evaluation and display.

Sequence diagram: Figure 4.5 shows a Sequence diagram for *Generator* to present data flow from *Analyser* → *Generator* → *Monitor*:

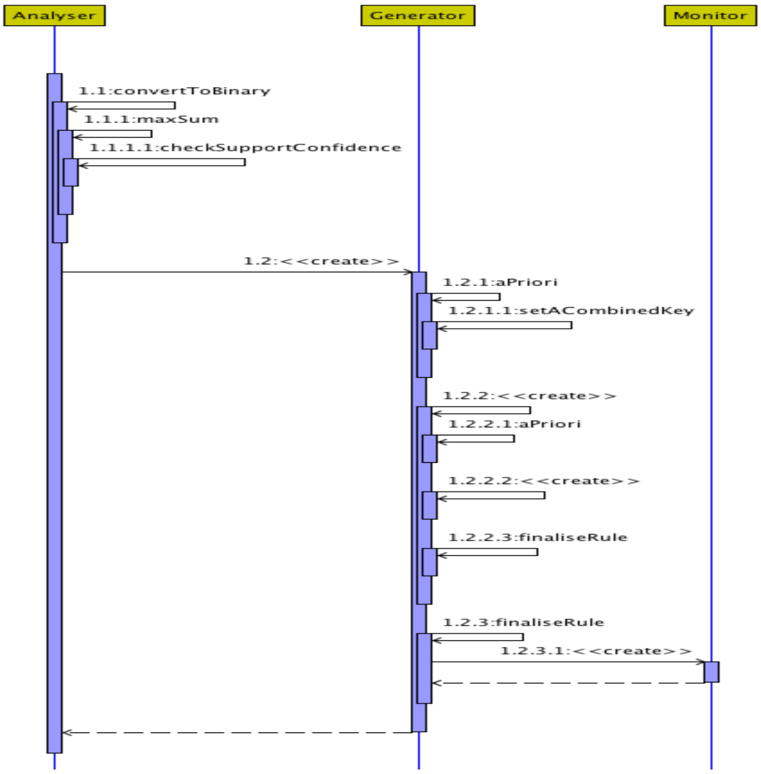


Figure 4.7: Sequence Diagram for class Generator

4.1.5 Phase IV Sub-Analyse

Aim: As a child-class of *Analyser*, this class analyses the combined range by calling some *Analyser* range-analysing method. It also provides extra density check and strict-adjust subset of tuples covered by candidate ranges. In other words, its aims are:

- To get tuples covered by both ranges and set it as new subset of tuples used to perform threshold tests.
- To check for density within combined ranges.
- To recall range analysing method in parent class *Analyser* including `convertToBinary`, `maxSum()`, `checkSupportConfidence()` and `convertPositionToRange`.

- To return new combined range as a confirmation of combined range passing threshold check to *Generator* through a `get()` method.

Input: This class input is the updated range and tuple map from the previous class, with the new combined key and combined range to analyse. Due to being a child class to *Analyser*, it has to intialises with `allClassMap` and `allTuple`. However, since they are not needed in this class, it was decided to drop them i.e. passing *Null* as paramaters to those two in constructor:

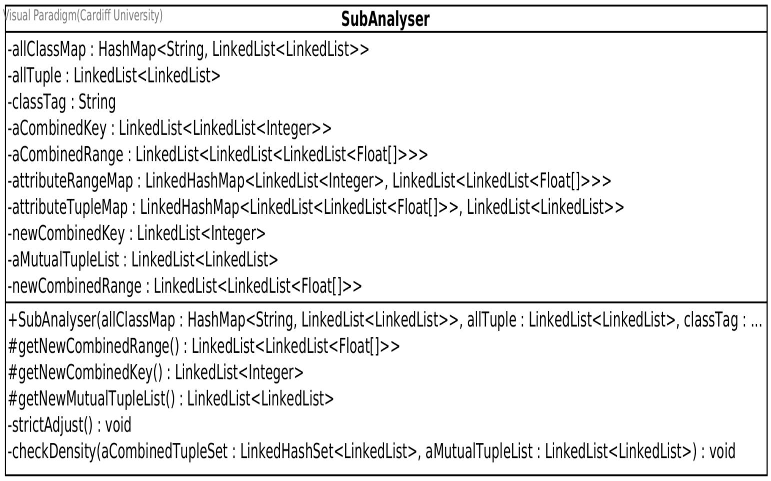
- `allTuple`: Now presented as *Null*.
- `allClassMap`: Now presented as *Null*.
- `attributeTupleMap`: Same as that in *Generator*.
- `attributeRangeMap`: Same as that in *Generator*.
- `aCombinedKey`: combined key from *Generator* to retrieve ranges when applicable.
- `aCombinedRange`: combined range from *Generator* to apply threshold tests upon.

Reason: Unlike *Analyser* in the first iteration where all attributes are considered, for the second iteration onwards with *SubAnalyser* only one (combined) attribute and one (combined) range are considered at one time. Therefore only one (combined) range `aCombinedRange` is analysed within this class.

Output: A new combined range is returned by a `get()` method to *Generator* once this class is done running, if the combined range passes specified threshold tests. However, it returns *Null* if threshold were not satisfied.

Reason: As *SubAnalyser* calls *Analyser* method, new sub-ranges can be found from the combined range through `maxSum()` analysis. Therefore, one or multiple ranges could be returned from *Analyser* once the methods finished. This set of range is then grabbed by *SubAnalyser* and returned to *Generator*, indicating that there is a possible pair tested to be applicable for the next iteration. By *Analyser* returning *Null* when threshold requirements failed to be met, *SubAnalyser* will also return *Null* to *Generator* which signifies *Generator* to seek for new pairing.

Data type: As discussed, using nested `LinkedList`, attribute number and range can be extended unrestrictedly. For the updated mutual tuple set, a `LinkedHashSet`. `LinkedHashSet` is used as it avoids duplication while keeping elements in order.



```
SubAnalyser
-allClassMap : HashMap<String, LinkedList<LinkedList>>
-allTuple : LinkedList<LinkedList>
-classTag : String
-aCombinedKey : LinkedList<LinkedList<Integer>>
-aCombinedRange : LinkedList<LinkedList<LinkedList<Float[]>>>
-attributeRangeMap : LinkedHashMap<LinkedList<Integer>, LinkedList<LinkedList<Float[]>>>
-attributeTupleMap : LinkedHashMap<LinkedList<LinkedList<Float[]>>, LinkedList<LinkedList>>
-newCombinedKey : LinkedList<Integer>
-aMutualTupleList : LinkedList<LinkedList>
-newCombinedRange : LinkedList<LinkedList<Float[]>>
+SubAnalyser(allClassMap : HashMap<String, LinkedList<LinkedList>>, allTuple : LinkedList<LinkedList>, classTag : ...
#getNewCombinedRange() : LinkedList<LinkedList<Float[]>>
#getNewCombinedKey() : LinkedList<Integer>
#getNewMutualTupleList() : LinkedList<LinkedList>
-strictAdjust() : void
-checkDensity(aCombinedTupleSet : LinkedHashSet<LinkedList>, aMutualTupleList : LinkedList<LinkedList>) : void
```

Figure 4.8: Class Diagram for class SubAnalyser

Class diagram: Its class diagram is in Figure 4.5, which presents five features/methods:

- strictAdjust()

  This method seeks to construct 1) a set that contains all tuples that is at least covered by one range and 2) a mutual set of tuples where only tuples that are covered under both ranges are appended. Both sets are then passed onto next method `checkDensity()`.

- checkDensity()

  This method uses the two tuple sets from `strictAdjust()` method to perform density check on the combined range. If the combind range satisfies density requirement, it is passed onto the same range-analysis as first iteration: `convertToBinary()`, `maxSum()`, `checkSupportConfidence()` and `convertPositionToRange()` from *Analyser* through inheritence calls. In this case density is checked first then the support and confidence is checked afterwards to reuse the code in *Analyser*.

- getNewCombinedRange()

  This method is a `get()` method to retrieve the new combined range for *Generator* after this class finishes.

- getNewCombinedKey()

  This method is a `get()` method to retrieve the new combined key for *Generator* after this class finishes.

- getNewMutualTupleList()

  This method is a `get()` method to retrieve the new mutual tuple set obtained in `strictAdjust()` for *Generator* after this class finishes.

Sequence diagram: Below is a sequence diagram used to present a data flow between *Generator* → *SubAnalyser* → *Analyser*:

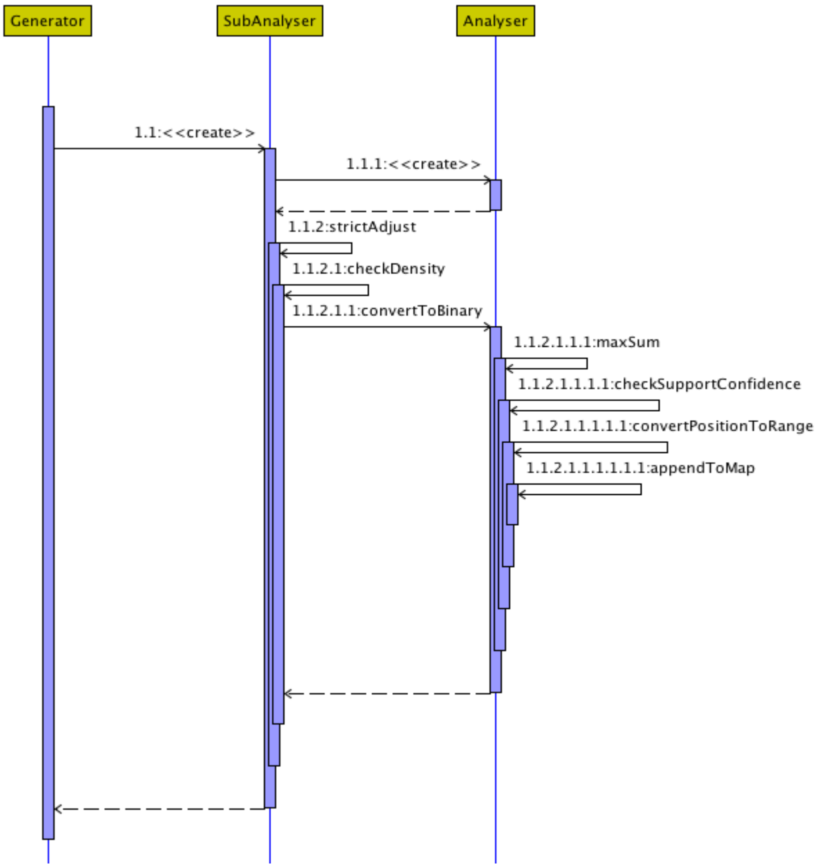

Generator
SubAnalyser
Analyser
1.1:<<create>>
1.1.1:<<create>>
1.1.2:strictAdjust
1.1.2.1:checkDensity
1.1.2.1.1:convertToBinary
1.1.2.1.1.1:maxSum
1.1.2.1.1.1.1:checkSupportConfidence
1.1.2.1.1.1.1.1:convertPositionToRange
1.1.2.1.1.1.1.1.1:appendToMap

Figure 4.9: Sequence Diagram for class SubAnalyser

4.1.6 Phase V Monitor

Aim: As the final phase, this class wraps up the programme with the following aims:

- To by reformating the remaining associated ranges into readable classification rules.

- To evaluate the accuracy and performance of the rules.
- To output the evaluation as well as the rules into a text file for the record.

Input: This class input is the final version of range and tuple map from *Generator*, as well as the original dataset to retrieve range values:

- `allTuple`: all original instances in dataset.
- `attributeRangeMap`: with combined attribute number as key and (one or more) associated ranges (classification rules) as value.
- `attributeTupleMap`: with each associated range as key and its covered tuple set as value.

Reason: `allTuple` is needed to retrieve the float ranges and calculate the rule accuracy. `attributeRangeMap` and `attributeTupleMap` are the containers of the associated ranges i.e. the classification rules.

Output: *Monitor* produces two outputs: the evaluation of rules (including runtime and accuracy) as `String` and the rules itself to the terminal and text `.txt` file as record.

Reason: Initially, class *Monitor* did not exist as the information is produced directly to the Terminal when the programme runs. However, for readability and back-tracking functionality, a text log is decided to be kept alongside Terminal output, hence *Monitor*.

Data type: All information is transformed into `String` as it is appended to the text `.txt` file.

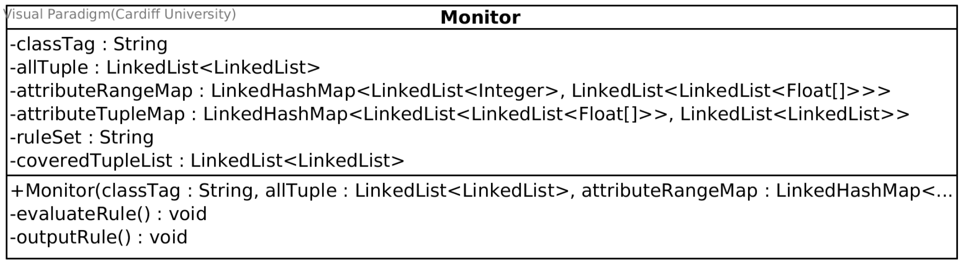

Visual Paradigm(Cardiff University)
Monitor
-classTag : String
-allTuple : LinkedList<LinkedList>
-attributeRangeMap : LinkedHashMap<LinkedList<Integer>, LinkedList<LinkedList<Float[]>>>
-attributeTupleMap : LinkedHashMap<LinkedList<LinkedList<Float[]>>, LinkedList<LinkedList>>
-ruleSet : String
-coveredTupleList : LinkedList<LinkedList>
+Monitor(classTag : String, allTuple : LinkedList<LinkedList>, attributeRangeMap : LinkedHashMap<...
-evaluateRule() : void
-outputRule() : void

Figure 4.10: Class Diagram for class Monitor

Class diagram: Its class diagram is in Figure 4.6, which presents five features/methods:

- evaluateRule()

  This method is responsible for re-formating the ranges and calculate the rule performance and accuracy. For re-formating, due to data structure requirements the ranges found are not in correct order e.g. $r = [a, b]$ where $a > b$, so this method swaps the ranges into correct band to get $r = [b, a]$.

For rule evaluation, initially it calculated the accuracy by applying the rule onto the original dataset and observed if its prediction matched the rules. As realised at later stage, this approach was not sufficient in determining the true accuracy of the programme, and training set was considered by spliting the original dataset into half. However, due to time constraint there was no working implementation for this testing part and thus `evaluateRule` only re-formats rules instead of evaluating them for the time being.

- outputRule()

  This method outputs the rules (that are formatted into `String`) onto a text file for logging purpose.

Sequence diagram: A Sequence diagram is used to present the data flow between $Generator \rightarrow Monitor$:

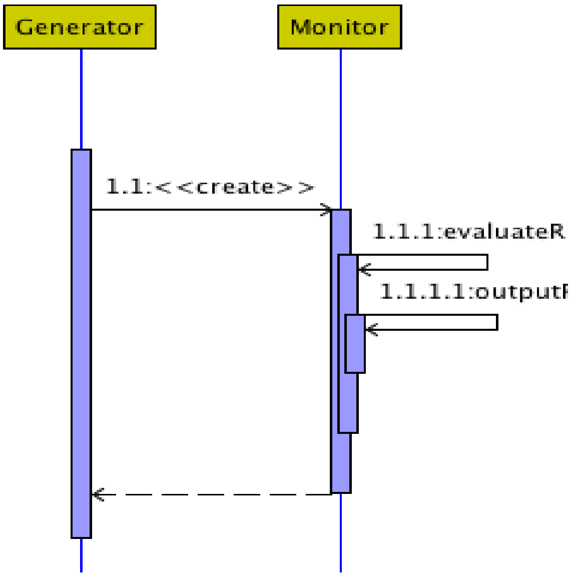


Figure 4.11: Sequence Diagram for class Monitor

4.2 Development Process

In this section, the software engineering aspects of the project is discussed: on planning, technical tools utilised and version control for project manangement.

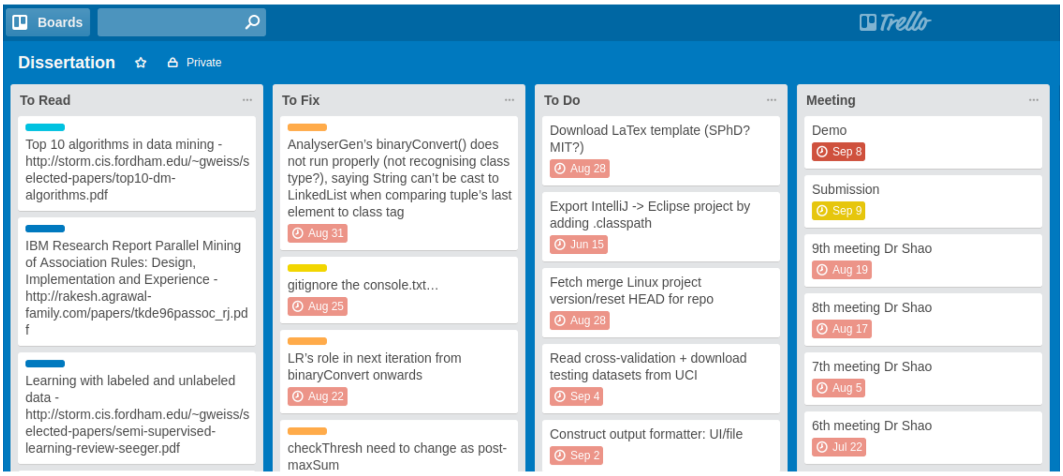
4.2.1 Planning

The most common way to do project planning is to rely on Gantt Chat]. Gantt chart provides an overview for project development process and monitors the progress based on specified deadlines for different tasks.

However, there are algorithmic as well as implementing changes that are considered frequently during the process. It was decided that Gantt chart would not be dynamic enough to accommodate changes, and as a replacement a Kanban board was used.

Kanban gives more focus to the tasks while still keeping track of the deadlines. Trello, an online Kandan tool, is used in this project for its cross-platform mobility (as a website) and the ability to sort and archive tasks if necessary.

A screenshot of the Trello board for this project is as below:



To Read

Top 10 algorithms in data mining - http://storm.cis.fordham.edu/~gweiss/selected-papers/top10-dm-algorithms.pdf

IBM Research Report Parallel Mining of Association Rules: Design, Implementation and Experience - http://rakesh.agrawal-family.com/papers/tkde96passoc_rj.pdf

Learning with labeled and unlabeled data - http://storm.cis.fordham.edu/~gweiss/selected-papers/semi-supervised-learning-review-seeger.pdf

To Fix

AnalyserGen's binaryConvert() does not run properly (not recognising class type?), saying String can't be cast to LinkedList when comparing tuple's last element to class tag
Aug 31

gitignore the console.txt...
Aug 25

LR's role in next iteration from binaryConvert onwards
Aug 22

checkThresh need to change as post-maxSum

To Do

Download LaTex template (SPhD? MIT?)
Aug 28

Export IntelliJ -> Eclipse project by adding .classpath
Jun 15

Fetch merge Linux project version/reset HEAD for repo
Aug 28

Read cross-validation + download testing datasets from UCI
Sep 4

Construct output formatter: UI/file
Sep 2

Meeting

Demo
Sep 8

Submission
Sep 9

9th meeting Dr Shao
Aug 19

8th meeting Dr Shao
Aug 17

7th meeting Dr Shao
Aug 5

6th meeting Dr Shao
Jul 22

Figure 4.12: Trello Board as Planner

4.2.2 Technical Platforms

For this project, there are several softwares utilised:

- IntelliJ [6]

  As the first IDE used for this project, it provides a simple and convenient integration with version control from GitHub. Its SequencePlugin [7] provides quick Sequence diagram generator that is used in this report.

- Eclipse [8]

  In addition to IntelliJ, Eclipse is used in computer labs. While having similar functions as IntelliJ, Eclipse also provides a free Class diagram generator plugin (ObjectAid [9]) for a quick UML generator. However, the Class diagrams used in this report is generated by Visual Paradigm instead, as Visual Paradigm provides more detailed information e.g. type of parameters.

- Git - GitHub [10] GitHub is used for mananging the project and the report itself, which ensures mobility of the programme and detailed records of code change history. GitHub is used as an integration to IntelliJ and Eclipse.

  GitHub links are available for the latest updates of project [11] and the report in both pdf and laTex format [12].

- Visual Paradigm [13] Visual Paradigm is used to generate Class diagram in this project, replacing Eclipse plugin.

- Weka [14] Weka is used for testing part of this project by cross-validating this project outcome against that of top ten algorithm [15].

- LaTex LaTex is used for constructing this report. Its class template can be found at [16].

4.2.3 Version Control

Version control for this project is purely done on GitHub. Since half of the project was done remotely, a distributed logging system shall be used, thus GitHub is the optimal choice for its progress tracking as well as revertible contents.

Section 5

Testing

This section contains all testings done on the programme output. To measure the rule accuracy, $k-foldcross-validation$ is chosen, in which both training and testing set are only used once for testing [17]. In this project, a two-fold cross-validation estimator is applied ($k = 1$). This means given a dataset, it is partitioned into two equal-sized sub-datasets. One is used for training (i.e being used as the programme input to get output rules) and another is used for testing (i.e being used for accuracy measurements for the output rules obtained earlier). The two are then swapped once, meaning the first half is for testing and the second half for training. Accuracy is taken as an average of that of both rounds.

While *holdout* methods could have been used, $k - foldcross - validation$ estimator variance is lower than the other by averaging its estimation over k different partitions [17], hence chosen for this project testing part.

In terms of performance, a *runtime* measuring method is set at the beginning of the programme, which records starting time and returning finishing time to calculate the total runtime (measurement unit: milliseconds).

However, due to time constraints, there are only two datasets being used as testing sets for this programme: Iris (4 attributes, 150 tuples) and Wine datasets (13 attributes, 178 tuples) [18].

5.1 Performance

In terms of performance, on Iris dataset, average runtime of 5 runs is around $234ms$. On Wine dataset, it is to be $475ms$ which is doubled from the Iris dataset but explanable due to its attribute size difference.

However, performance can slow down significantly if $System.out.println()$ is used within the programme. It results in $8126ms$ ($\approx 8s$) for Iris dataset and $23212ms$ ($\approx 23s$) for Wine dataset.

5.2 Accuracy

In terms of accuracy, as stated, a two-fold cross-validation technique is applied. The two dataset, Iris and Wine, is cross-validated twice: firstly with $\sigma_{min} = \delta_{min} = \gamma_{min} = 0.5$ and secondly with $\sigma_{min} = \delta_{min} = \gamma_{min} = 0.7$.

Figure 5.1 shows the result as:

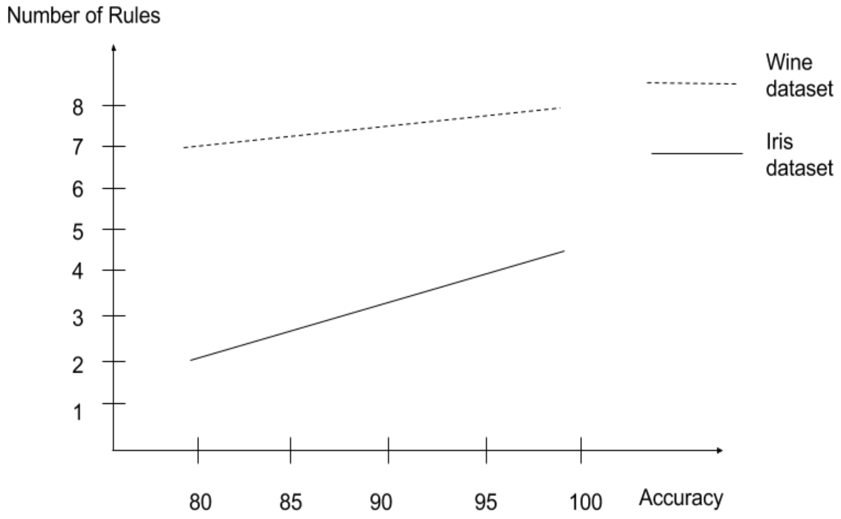


Figure 5.1: Accuracy measures obtained using two-fold cross-validation technique

From Figure 5.1, the programme shows a positive relationship between the number of rules and accuracy level, with both reaching 100% accuracy at the end of the test. This suggests the programme is in the right direction, matching the common trend. There are fewer rules in Iris dataset compared to that of Wine, which is explanable due to the significant gap in attribute numbers between them. However, upon testing, despite rules constructed successfully, it was observed that the programme seemed to produce similar rules at most times despite changes in threshold requirements. Another test was then done setting all thresholds at a low 0.1, where the programme was expected to produce much larger number of rules compared to the initial test. However, the programme freezed and a *Null* error appeared in the console. This suggests a structural error in the recursive loop, although due to time constraint no tests could be done further.

Section 6

Conclusion

In conclusion, the programme implements the original algorithm, from which it is able to derive classification rules given a set of user-specified thresholds. With minor modifications to range adjustment and adaptation to Java data structure, the programme is in shape to produce outcome in the required format.

However, the given output is suggested to be inherently inaccurate: output rules do not vary as expected, and at very low threshold (0.1 when tested), potential crashes could occur. It is suspected that there exists structural errors, as data partitions in the generating phase might have been passed on incorrectly, resulting in wrong output. Moreover, crashes occur at lower threshold boundary might concern with a bad behaviour within recursive calls. Because of inheritence calls being nested in recursive loop, bad inner calls might be difficult to be spotted.

Nevertheless, the programme could produce correctly formatted output and with more time allowance, systematic bugs can be tested and amended. It is believed that a revision to parameters passed within the recursive loop could redirect the algorithm to work on the right resources and put the programme on track.

Section 7

Future Work

Firstly, the programme could benefit from a revision in data structure within all classes. It also benefits from a simpification within recursive loop: Inheritance could be replaced by an independent class.

Secondly, once producing the right output, multi-threads could be implemented. Initial idea is to hand over each class and its data partition to each Java worker thread, which makes for a parallel structure in analysing ranges that could potentially make the programme faster.

Thirdly, reconsiderations can be made upon how data could be partitioned. In this project, following the original algorithm, data is partitioned using max-sum method and an adjustment method to range cover. Other alternatives can be considered i.e max-miner approach [19].

Section 8

Reflection

Theory

Upon the theoretical aspect of the project, I have felt that I take too long before getting a firm grip of the algorithm. Due to the lack of experience to data mining, I take long to pick up the basics and struggle to quickly apply my understanding upon a custom method. I should have also maintained a macro-control over the algorithm (maintaining an overall picture of the data flow) instead of having a tunnel vision to individual parts of the programme. The result is my algorithm design being structurally fragmented i.e. may work well at single phase level but get confusing once iteration between phases starts.

Subsequently, most of my time is spent fixing iterative data structure errors, which could have been used for understanding and analysing the algorithm itself and possibly make improvements for it. Instead, the project is merely an implementation of the algorithm; no significant improvements were made on the theoretical front, which is not my initial expectation to the project.

Implementation

In terms of implementation, I have chosen Java as I enjoy the precise structure required in its data. However, for the exact same reason, it does not benefit me for having a lack of exposure to data structure and algorithm in object-oriented languages. A lot of my obstacles lie in the confusion of how abstract data type (ADT) extracts its elements and how variables are kept internally within each class after forced re-initialisation in inherited classes. My persistence to maximum code reusability means I would like to keep my code as concise as possible, but this in turn forcing me to deal with inheritence within a large amount of recursive iterations. It might not worth insisting on code efficiency early in exchange for a simplier but working solution.

Project Management

As I realise my lack of in-depth knowledge in this area, doing a research-oriented project means that I have to spend extra time in acquiring only background knowledge. However, I have been impatient with the project and often jumped in before fully understanding what it means to do this and that. My planning is therefore not as concrete as I would like, as I keep changing my mind about little details.

Moreover, I should have better risk assessment even when the programme seems to work, only to find out it gives semantically wrong output at the end of the project.

Nevertheless, I thoroughly enjoy working with a research-based problem, especially in an area I would otherwise not have exposure to. I have learnt to use various technical tools and familiarised myself with new design patterns and data structures. It is an inspiring project and despite the outcome, I would like to continue doing it for pure enjoyment.

Bibliography

[1] R. Srikant and R. Agrawal, "In Proceedings of the ACMSIGMOD Conference on Management of Data," p. 112, 1996.

[2] J. Bentley, "Programming pearls: algorithm design techniques," *Commun. ACM*, pp. 865–873, 2011.

[3] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[4] G. Florin, *Data Mining: Concepts, Models and Techniques*. Springer-Verlag, 2011.

[5] H. Vliet, *Software Engineering: Third Edition: Principles and Practice*. Wiley, 2006.

[6] JetBrains, "IntelliJ IDEA the Java IDE." `https://www.jetbrains.com/idea/`.

[7] Kentaur, "JetBrains Plugin Repository: SequencePlugin." `https://plugins.jetbrains.com/plugin/22?pr=`.

[8] Eclipse, "Eclipse Neon." `https://eclipse.org/`.

[9] ObjectAid, "ObjectAid UML Explorer." `http://www.objectaid.com/`.

[10] GitHub, "How People Build Software - GitHub." `https://github.com/`.

[11] Repository, "Rangeclassification." `https://github.com/Phuongt994/rangedclassification`.

[12] Repository, "Rangeclassificationreport." `https://github.com/Phuongt994/rangedclassificationreport`.

[13] V. Paradigm, "Visual Paradigm Cardiff University License." `https://www.visual-paradigm.com/`.

[14] G. H. B. P. P. R. M. Hall, E. Frank and I. H. Witten, “The WEKA Data Mining Software: An Update,” *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, Nov. 2009.

[15] X. W. et al, “Top 10 algorithms in data mining,” *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, 2008.

[16] S. Garg, “CTAN: SPhD template for LaTex.” `https://www.ctan.org/pkg/sphdthesis?lang=en`.

[17] R. Kohavi, “A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection,” pp. 1137–1143, Morgan Kaufmann, 1995.

[18] A. Frank and A. Asuncion, “UCI Machine Learning Repository.” `http://archive.ics.uci.edu/ml`.

[19] R. Bayardo, “ in Proc. of the 1998 ACM-SIGMOD IntâĂŹl Conf. on Management of Data,” pp. 85–93, 1998.