Universiteit Leiden
The Netherlands

Opleiding Informatica

Diverse subgroup discovery

for big data

Sinan Zaeem

Supervisors:
Matthijs van Leeuwen & Marvin Meeng

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

23/08/2017

Abstract

Nowadays, the amount of the data used in different fields is increasing. The consequence is that the task of the data mining in finding the useful data becomes more and more difficult to achieve. The highly redundant result sets and a big search space are the most biggest problems when it comes to handle big data.

Subgroup Discovery (**SD**) is a data mining technique that has these two above mentioned problems. The task of SD is to discover interesting subgroups that show a special behavior with respect to some interest target property. The existing *diverse subgroup set discovery* (DSSD) algorithm solved the SD problems by considering subgroup set discovery rather than the individual subgroups.

During the process of searching for the subgroups, their covers should be kept. These covers show the subset of the dataset. The function of a subset is precisely to show whether a row is covered by a subgroup or not. One of the approaches to deal with subgroups subsets is to cache all these subsets. This in turn will be a problem when it comes to big datasets because in this case, the current subset implementation require a lot of memory.

In our study, we consider five subset implementation techniques and we conduct a number of experiments on 38 datasets. The results of these steps show that the performance of the implementation techniques depends on the characteristics of the datasets. In this research are nine most important characteristics of datasets used regarding of their way of predicting the final results in runtime and memory usage.

To predict the suitable implementation technique for a given dataset, linear regression models were built using *WEKA* tool in order to find the effect of each property on the end results. The results of *WEKA* show that the prediction measures of the memory complexity models are more accurate than the prediction measures of the time complexity models.

Contents

Chapter 1

Introduction

In every day and in every aspect of our life, data is being recorded and stored. For example, data in the social media are being used to describe people, their profiles and what they do like. Another example is in the health sector, where data describe patients information regarding their health background, drugs being used, infections and etc. All these data must be analyzed so that we can extract the useful information out it and make it understandable to use for further purposes. Therefore, the concept of the *data mining* is introduced.

Data mining [HPK12], is a process of extracting information from raw data. For the extraction process, there are many data mining techniques [Puj01]. The input of the data mining techniques are the datasets. A dataset is data that is arranged in a form of rows and columns. A dataset has a number of description attributes and one or more target attributes.
Subgroup discovery (**SD**) [Atz15] is a data mining technique that identifies a subset of rows in the dataset that shows unusual behavior relative to what is expected according to a specific target attribute.
The subgroup is a conjunctive of described attributes of the dataset. As an example of a subgroup; consider the following sentence: *People with high salary and fixed job are more likely to buy a house*. Here we can describe this subgroup by the conjunction of the following description conditions: (salary = high) and (fixed job = yes) with the target condition which is (buy a house= yes).
The goal of subgroup discovery is eventually to find the top k set of high-quality subgroups. The quality of a subgroup is determined by one of the quality measures. There are many quality measures [HCGdJ11] types depending on the target attribute itself. The target could be single or multiple attributes at once. The target attribute may be nominal, numeric or binary. The quality measure shows the importance of a subgroup according to the target attribute.
Subgroup discovery can be seen in many applications [Atz15] for example, Knowledge Discovery in the Medical Domain [GL02] [AP05], Subgroup discovery in Social Media [Atz12] [ALKH09] and Technical Fault Analysis [NP14] [JFW+14].
An efficient searching strategy in finding these subgroups is critical because, the search space for all the possible subgroup candidates is exponentially increasing. Therefore, the exhaustive search strategy (brute-force) is difficult to implement. The heuristic search strategy is more efficient but it does not guarantee to

mining the best candidates. SD-Map algorithm [AP06] for example, uses the exhaustive search while DSSD algorithm [vLK12] uses the heuristic search strategy and especially the beam search.
Extracting useful information from a big dataset is difficult for all data mining techniques inclusive SD technique because of the redundancy and the big complexity problems. In DSSD [vLK12], the subgroup set discovery is introduced and considered instead of individual subgroups. DSSD insures the diversity and less redundancy subgroups at the end result.
The challenge of the above mentioned algorithms is when we have big datasets. We refer to this dataset as a big set when it has more than 50000 rows or more than 300 columns. Large runtime and large memory usage problems can appear when we apply subgroup discovery on a big dataset. The reason is related on how we deal with the subgroup subset. Subgroup subset represents the covered rows in the datasets for a subgroup. Knowing the subset of each subgroup is important for calculating the quality of this subgroup. The subset implementations in this research are known as the mask implementations.

In DSSD [vLK12] there are two methods to deal with the subsets in the memory. The first method is to keep the all the subgroups subsets in the memory so that we can go back to it when its necessary. The second method is to keep only the required subgroups subsets in the memory. If we need another subset which is not already exist in the memory, then we must calculate it again. Therefore, the memory usage of the second method is not large compared to the first method. As a consequence, large runtime will be applied for this method. In this research, we choose to work with the first method and investigating the related problem.

The current subset implementation data structure uses one byte memory to show if a row in the dataset is covered by the subgroup. That means if we have a dataset consists of one million of rows then we need one megabyte of memory for each subgroup subset. If we have a million subgroups then we need one terabyte of the memory. Therefore, we need to find an efficient representation of the subsets so that we can improve the memory usage.
To apply SD in big datasets, we need to improve the memory usage limitations. To do that, we examine the limitations of the current subset data structure implementation and investigate the other data structure implementations.

The main goal of this study is to answer the following question: *How to discover diverse subgroup sets from big datasets up to* 500000 *rows with reasonable runtime and memory usage using different subset datastructure implementations?* Other sub-questions which need to be answered in this current study are:

1. Can another subset implementations reduce the memory usage?
2. Will the runtime also be reduced?
3. Is the subset implementation performance affected by the dataset characteristics?

In the current subset implementation, all dataset rows must be represented for each subgroup and that consume unnecessary memory.
In order to reach the goal of this study, we should first investigate the current subset implementation named the Bool-Array, which uses one byte of memory to represent the covering state of each row in the dataset. Next

we investigate three implementation techniques those are:

1. Uint32-Array, which uses four bytes memory to represent a covered row.
2. BitMask, which uses one bit for each row in the dataset.
3. Hash table, which uses eight bytes memory for each covered row.

We investigate also AutoMask technique which combines between the Bool-Array technique and the Uint32-Array technique. After that we analyze all these techniques, we select one or more of them based on their runtime and memory usage (Sect.4.1).

The Bool-Array and Uint32-Array techniques are already implemented in the existed software of the DSSD. The implementation task regarding the BitMask and AutoMask techniques has been done in this study. The Hash table technique has not been implemented see (Sect. 5.1). Next we perform several experiments including, testing these four above mentioned techniques on 38 datasets. Here, the following dataset properties will be considered in this experiment; number of rows, number of columns (binary, float and nominal), description attributes, model attributes, density, and average coverage. From these experiments we focus on the results of the memory usage and the runtime.

In the final step we want to know the effect of the dataset properties on the runtime and memory usage results. To do that, we use the information of the 38 datasets and the obtained results from our experiments to build several prediction models. Building of these models was done using WEKA tool, a 10-fold cross validation technique and linear regression method [WF05]. For each implementation technique, two prediction models are included, one for the runtime results and the other for the memory usage results. The predicting quality of each model will be measured using five measurements to show how good the prediction of these models are.

In the discussion section, we discuss the obtained results in details. One of these results for example is, the BitMask technique has low memory usage and high runtime comparing with the other techniques. Other obtained result indicated that, the memory usage prediction models show better results than the runtime prediction models see (Sect.6.2). By discussing these results, we can indicate which implementation technique are best to use for a provided dataset to get the best results.

Finally, the prime and the central goal of our paper is to discover diverse subgroup for datasets that have a size up to 500000 rows or number of columns up to 2000 columns within 48-hours using no more than 13 Gb of memory. We do that by selecting the suitable implementation technique according to the dataset characteristic. The experiments show that we have fulfilled our goal.

1.1 Definitions

In this section, some of the important definitions will be explained.

1.1.1 Subgroup Discovery

Subgroup discovery (**SD**) is one of the data mining techniques that specializes in finding interesting subsets of datasets that have a special behavior according to some interest target property. The subgroup discovery concept was introduced by Kloesgen [Klo96] and Wrobel [Wro97].

Let the examples in the datasets S consist of a set of attributes referred to A. A consists of one or more description attributes referred in this case to D and one or more target attribute referred to T Each description attribute D_i and target attribute T_i has a domain of possible values Dom(D_i) and Dom(T_i) respectively. The concept *subgroup* consists of; (1) *Subgroup cover* which defines as a subset of the examples $E \subseteq S$ and $|E|$ represents the subgroup size and (2) *Subgroup description* which is a function $s : (Dom(\mathrm{D}_1) \times ... \times Dom(\mathrm{D}_k)) \mapsto \{0, 1\}$, and its corresponding subgroup cover is $E_s = \{e \in S | s(e^D) = 1\}$. See [vLK12].
A subgroup description is a pattern, can be expressed by conjunctive of conditions on the description attributes [vLK12]. An example of that, The following dataset D has four attributes:

Age = {Less than 25, 25 to 60, More than 60}

Diploma = {Yes, No}

Experience = {0 to 2, 2 to 5, More than 5}

Target:

Salary = {Low, Medium, High}

The following subgroups are two possible subgroups which show unusual behavior:

S1: {Age = Less than 25 AND Diploma = No AND Experience = 0 to 2} with Target Salary = High.

S2: {Age = 25 to 60 AND Diploma = Yes AND Experience = 2 to 5} with Target Salary = Low.

S1 shows that young people with no diploma and a little experience have high salary which is unusual comparing with the rest of the population. **S2** shows that people between 25 and 60 with a diploma plus experience with more than two years have low salary which is also unusual.

One of the examples of SD application is the *SUBGROUP DISCOVERY IN SOCIAL MEDIA* [ALKH09]. In this application, the SD is used for example for discovering the spammers profiles in social media. This is done by describing the special and common features of the spammers by applying the SD technique. As a result, the patterns will be mined that help to identify the spammers subgroups. We will mark these subgroups as spammers candidates.
To know how interesting a given subgroup is, we need to apply a quality measure on its target. The task of SD is to find the top-K-subgroups according to their quality measure. For this task we need the following parameters: a dataset S with one target attribute, a k number and a quality measure φ. The search strategy is also important to find the top-k-subgroup. We use the top-down search strategy where we add in each level one condition to the subgroup description.

1.1.2 Quality measures

To evaluate the subgroups one of the quality measures is used. According to the value of a specific quality measure we will discard or select a specific subgroup. There are many quality measures exists [vLK12] depending on their main objective. Because the *WRAcc* [LKFT04b] quality measure will be used in our experiments, the mechanism of its work will be explained as following; The weighted relative accuracy *WRAcc* depends on the examples weights. Because this weight represents how many times the examples are covered by a subgroup. *WRAcc* depends directly on the size of a subgroup and the number of positive examples that the subgroup covers. We use *WRAcc* for single binary target attribute. The measure equation of *WRAcc* can be seen below see [vLK12]:

$$\varphi_{WRAcc}(G) = \frac{|G|}{|S|}(1^G - 1^S)$$

Where:

G: subgroup cover

S: The dataset

1^S : the fraction of ones in the target attribute within the subgroup

1^G : the fraction of ones in the target attribute within the dataset

1.1.3 Beam search

Beam search is a heuristic top-down search strategy. In beam search the bread-first search strategy is used because not all parts of the search space are explores. Only the selected subgroups will be included in the beam in each level. These subgroups will be used further in the refinement. In the refinement process we generate all possible subgroup descriptions by adding one condition in each level.

1.2 Thesis Overview

This chapter contains the introduction; Chapter 2 discusses related work; Chapter 3 explains how DSSD algorithm works; Chapter 4 describes the mask implementations techniques; Chapter 5 shows the experiments and the results; Chapter 6 discusses the experiments results; Chapter 7 concludes this thesis.

Chapter 2

Related Work

2.1 SD-Map

Because the search space of the exhaustive search become enormous in case of large data, the runtime become very large which makes the exhaustive search impossible. Therefore, the heuristic search is used instead, to discover the top k subgroups. The problem of the heuristic search is that, for multiple reasons [AP06] not all important patterns can be mined. The literature [AP06] shows how to use a fast exhaustive search method to discover all important patterns in the dataset. The algorithm **SD-Map** is introduced which used the *FP-Growth* [HPY00] algorithm to find the frequent patterns set. The *FP-Growth* algorithm uses a recursive divide-and-conquer algorithm to test each frequent pattern. This way has a benefit of reducing the number of database scans.

FP-Growth algorithm implement the FP-tree which stores count information about the frequent patterns. In *SD-Map* algorithm, we collect the frequent patterns using the *FP-growth* method and we test these patterns using a quality measure to measure the importance of a subgroup.

2.2 Subgroup Set Selection

Subgroup set selection plays a big role for improving the obtained subgroup results and preventing redundancy. The literature [Atz15] explains the options for diversity-aware and redundancy-aware subgroup set discovery. The explanation based on condensed representations, relevance criteria, covering approaches and causal subgroup analysis. We will explain briefly each of them:

1. Condensed Representations: In the association rules field the condensed patterns helps finding the interesting patterns e. g., [PBTL99] [BPT$^+$00]. That leads to reduce the size of the result sets and improves the redundancy.

2. Relevance of Subgroups: Considering the relevance of a subgroup [GL02] comparing with the rest of the subgroups in the set.

3. Covering Approaches: The selection of a subgroup depends on its coverage of the dataset e.g., [GL02] [LKFT04a].

4. Causal Subgroup Analysis: It finds all subgroups that are causal for the target e.g., [KM02]. The operations on the instances to belong to a causal subgroup has a significant effect on their belong to the target group.

Chapter 3

Diverse subgroup set discovery algorithm

In this chapter we explain how the DSSD algorithm works. For better illustration, we will cite the pseudo code of DSSD algorithm from [vLK12] reference. See Algorithm 1.

Algorithm 1 DSSD diverse subgroup set discovery

Input: A dataset S, a quality measure φ, parameters j , k, $mincov$ and $maxdepth$, and subgroup selection parameters P.
Output: $\mathcal{R}$, an approximation of the top-k subgroups $\mathcal{G}k$.

DSSD (S, φ, j, k, $mincov$, $maxdepth$, P):

$\mathcal{R} \leftarrow \emptyset, Beam \leftarrow \{\emptyset\}, depth = 1$
while $depth \leq maxdepth$ **do**
    $Cands \leftarrow \emptyset$
    for all $b \in Beam$ **do**
        $Cands \leftarrow Cands \cup GenerateRefinements(b, mincov)$
    end for
    for all $c \in Cands$ **do**
        $UpdateTopK(\mathcal{R}, j, c, \varphi(c))$
    end for
    $Beam \leftarrow SubgroupSelection(Cands, \varphi, P)$
    $depth \leftarrow depth + 1$
end while
for all $r \in \mathcal{R}$ **do**
    $ApplyDominancePruning(r, \varphi)$
end for
$\mathcal{R} \leftarrow RemoveDuplicates(\mathcal{R})$
$\mathcal{R} \leftarrow SubgroupSelection(\mathcal{R}, \varphi, P)$
return $\mathcal{R}$

DSSD algorithm has seven parameters. The parameter j represents the number selected subgroups for the beam. The parameter k represents the number selected subgroups for the end result. The parameters P represent the parameters of a specific selection strategy. The other four parameters are clear from their name and are mentioned above.

The algorithm consists of three phases:

Phase 1: The algorithm begins with an empty beam and subgroup result set (line 1). Next, a loop will be executed beginning from level one depth to the maximum depth (line 2-12). In this loop, refining will be executed to generate the candidates of each level (line 5). The refinement operation uses the subgroups in the beam and the minimum coverage parameter as parameters to achieve the required task. In the refinement, the description length of each subgroup in the beam will be extended by one in each iteration. The new description must satisfy the minimum coverage condition. If this is the case, the new candidate will be added to the search space for the next level. Otherwise it will be skipped. For the new candidates the algorithm will calculate their quality measure and it will choose the top j candidates to update the subgroups result set (line 8). In each iteration, the subgroups in the beam will be updated according to one of the the selection strategies that the algorithm uses (line 10). We will highlight one of these strategies in Sect. 3.1. From this phase we will get the top-j subgroups result set.

Phase 2: A post-processing step will be achieved in this phase. After we get the subgroups result set, dominance pruning will be applied for each subgroup to get better results (line 14). In dominance pruning the algorithm removes a condition if that has no effect of decreasing the quality of the subgroup. For more information about the concept of dominance pruning, see [vLK12]. After the pruning operation is done the results will be eliminated from any possible duplicated subgroups (line 16).

Phase 3: In the last phase, selection on the result subgroups set will be applied to decrease the possible redundancy problems. The selection achieved using one of the selection strategies. From the last phase we get the final top-k subgroup result set. The selection strategies are explained in Sect. 3.1.

3.1 Selection Strategies

The literature [vLK12] describes three selection strategies, the description-based subgroup selection, cover-based subgroup selection and compression-based beam selection. We will show how the cover-based subgroup selection strategy works because it will be used in the experiments. For more details about the other selection strategies see [vLK12].

3.1.1 Cover-based subgroup selection

The selection of the subgroups depends on their covers of which two variants have been introduced:

1. Fixed-size cover-based selection: In this version, the strategy calculates a score. This score depends on the rows of a candidate that are already covered by the selected subgroups in the beam. The score calculated by the following equation see [vLK12]:

$$\Omega(G, Sel) = \frac{1}{|G|} \sum_{t \in G} \alpha^{c(t,Sel)}$$

Where $\alpha \in \langle 0, 1]$ is the weight parameter. The strategy selects the highest k subgroups according to the result of multiplied this score by the quality measure of that subgroup. In this version we have always a fixed number of subgroups at the end.

2. Variable-size cover-based selection: In this version, a minimum score will be calculated. The minimum score is used to determine if the iteration of the selecting candidates must be stopped. It is the same as the Fixed-size cover-based selection except that the score of the candidate must meet the minimum score limitation. In this version, we mostly have different number of subgroups at the end.

Chapter 4

Mask implementations

In this chapter, we will explain the mask implementations techniques. Each subgroup has one mask. The mask represents the subset of a specific subgroup. Therefore, the mask implementations techniques play an essential role in memory and time complexity. We will explain how the existing and the new techniques work and we will compare between them.

1. **Bool-Array**: This is an existing technique. A boolean array is used in this technique. The size of this array is equal to the size of the dataset. The boolean mask array represents the covered and uncovered rows of the dataset for a specific subgroup.

2. **Uint**32**-Array**: The idea of this existing technique is about using an unsigned integer mask array. All the covered rows in the dataset for a specific subgroup can be represented in this array. That means, we do not have to represent the information of the uncovered rows in this mask array.

3. **AutoMask**:
   In this new technique we check the number covered rows for each subgroup to select one of the above two techniques. If the number of the covered rows for a specific subgroup is smaller or equal to θ, we use the Uint32-Array technique. Otherwise we use the Bool-Array technique. Where θ is a parameter that depends on the size of the dataset.

4. **BitMask**: The idea of this new technique is the same as in Bool-Array except that we represent the row information with one bit instead of one byte.

5. **Hash table**: In this new technique, we use an unsigned integer pointer array with a specific size that related to the size of the dataset. These pointers points to linked lists. These linked lists will be filled with the covered rows information using a specific hash function.

We will show the comparison between all the above mentioned implementations techniques in the following subsection.

4.1 Memory and Time Complexity

We have analyzed the complexity of all the mask implementations techniques. That help us to decide which technique can be better implemented. The memory and time complexity can be seen in Table 4.1 and Table 4.2 respectively. In these tables, we describe the worst case complexity. For time complexity, we have analyzed it in Table 4.2 for the most important operations. The AutoMask technique is not included in these tables because it is a technique that depends on the Bool and the Uint32 Arrays techniques.

Table 4.1: Memory Complexity

Technique	**Bool-Array**	**Uint32-Array**	**BitMask**	**Hash table**
Memory space (bytes)	GD	$4GS$	$G(D/8)$	G(4H + 8S)

Table 4.2: Time Complexity

Operations	**Bool-Array**	**Uint32-Array**	**BitMask**	**Hash table**
Create	$D(m+w)$	$4Smw + D(w+r)$	$(D/8)(m + b) + D(w+r)$	$4H(m+w) + S(h + 8m + 8w) + D(w+r)$
A&B	$D(2r+v+w)$	$S_A(4r + 4rvlgS_B)$	$D(2g+v+b)$	$D(h + 4r + 8(S_A/H)rv) + S_A(h + 4r + 8(S_B/H)rv)$
Enumerate	Drv	$4Sr$	Dgv	$D(h + 4r + 8(S_A/H)rv)$

D: size of the dataset

S: number of covered examples

H: size of the hash table

G: number of subgroups

m: time to reserve one byte

w: time to write one byte

r: time to read one byte

v: time to make one operation or one comparison

h: hash function calculation time

b: time to write one bit in the bitmask array

g: time to read one bit in the bitmask array

From Table 4.1 we note the following:

The memory complexity of Bool-Array and BitMask techniques depends directly on the size of the dataset. The difference is that BitMask memory usage is the one eighth of Bool-Array memory usage.
The memory complexity of Uint32-Array and Hash table techniques depends directly on the number of covered examples. Hash table technique memory usage depends also on the size of the hash table.
That means, if we have a dataset with big size and small number of covered examples we can better use Uint32-Array or Hash table techniques. If we have a dataset with small size and big number of covered examples then we can better use Bool-Array or BitMask techniques.

From Table 4.2 we note the following:

The time complexity of all mask implementations techniques depends directly on the size of the dataset, time to execute one operation, and the time to reserve, write, and read one byte in the memory.

The time complexity of BitMask technique depends also directly on the time to read and write one bit in the memory.

The time complexity of Uint32-Array technique depends also directly on the number of covered examples and the binary search time.
The time complexity of Hash table technique depends directly also on the number of covered examples, hash table size, and hash function calculation time.
That means if we have a dataset with small number of covered examples we can better use Uint32-Array or Hash table techniques. If we have a dataset with a large number of covered examples we can better use Bool-Array technique. The time complexity of BitMask technique in general is larger than the time complexity of Bool-Array technique because, the time to read and write a single bit in the memory cost more time than reading and writing one byte in the memory.

The last information is summarized it in the Table 4.3 which gives us a comparison between the four implementations techniques according to the corresponding operation.

Table 4.3: Informal time complexity comparison

Operations	**Bool-Array**	**Uint32-Array**	**BitMask**	**Hash table**
Create	Fast	Medium	Fast	Slow
A&B	Fast	Slow	Medium	Slow
Enumerate	Fast	Fast	Mediam	Slow

Chapter 5

Experiments and Results

This chapter provides a description of the experiments and our obtained results. In Sect. 5.1 we will show what the effect is of using another mask implementation technique on runtime and memory usage. In Sect. 5.2 we will illustrate how we can predict the memory usage and the runtime according to the dataset properties.

5.1 Mask techniques complexity

To evaluate the mask implementation techniques, experiments were conducted on 38 datasets. Since the Hash table technique showed the lowest performance compared to the other mask implementation techniques, as was described in Sect. 4.1, the Hash table technique was not included in the experiments. The DSSD implementation uses existing software [vL12] where all implementation techniques are implemented using the C++ programming language. The experiments were conducted on a Xeon(R) 2.40 GHz system with 512 GB of memory running Windows Server 2012 R2 operating system. The following parameters were fixed for all experiments:

1. The highest k quality subgroups = 10000 because we want to get a large number of subgroups in the first phase within a reasonable runtime.
2. Subgroup set selection (Post-processing selection) = 100 because from the highest 10000 quality subgroups we can get 100 diverse subgroups without redundancy.
3. Search type = beam search. See (Sect. 1.1.3).
4. Maximum depth = 5 because we want to experience the selection strategy and the pruning method within a reasonable runtime.
5. Minimum coverage = 10 because the candidates subgroups that covered less than 10 rows in the dataset have less importance.
6. Beam width = 100 because it must be less than the highest k quality subgroups.
7. $\theta = \frac{1}{4}$ of the dataset size. Because the Uint32-Array technique gives better results in this case.
8. Quality measure = WRAcc. See (Sect. 1.1.2).

The results of the experiments on memory usage and runtime are shown in Table 5.1 and Table 5.2 respectively. These tables also show the following dataset properties:

1. Rows: The number of rows of a dataset (the size of the dataset).
2. Column: The number of columns of a dataset which can be:
   (a) Binary.
   (b) Float: For all experiments, we treat the float numbers using *on-the-fly* discretisation technique with six equal-sized bins.
   (c) Nominal.
3. Description: represents the number of description attributes.
4. Model: represents the number of model attributes.
5. Density: used only in the binary columns case. It represents the density of the true value in the whole binary columns.
6. Average cov.: represents the average number covered examples in the dataset.

5.2 Prediction using Weka

It is important to know how the obtained results in Table 5.1 and Table 5.2 were affected by the dataset properties, because this information will help to make predictions on the results for new datasets. The *Weka* tool was used to achieve this task. We used 10-fold cross-validation technique and a linear regression method to build models. The models show the quality of the prediction and what the effect of each property is on the results. Table 5.3 and Table 5.4 show how good the prediction is according to the following five measures:

1. **Correlation coefficient (CC)**: It shows the correlation between the true and the predicted output. The range is from 1 to 0. Where 1 means perfect correlation and 0 means there is no correlation.
2. **Mean absolute error (MAE)**: This is the average of all absolute errors. Where the absolute error is the absolute difference between the predicted value and the true value.
3. **Root mean squared error (RMSE)**: It shows the concentration of the data points around the regression line. The smaller the value the better.
4. **Relative absolute error (RAE)**: It calculates the total absolute error divided by the total absolute error of a simple predictor. Where the simple predictor is the average of the true values.
5. **Root relative squared error (RRSE)**: It calculates the root of the total squared error divided by the total squared error of the simple predictor.
   For more details about *Weka*, the cross-validation measures and linear regression method see [WF05].

Table 5.5 and Table 5.6 show the effect of the dataset properties on the end result. We use the linear regression equation:

$$y = a + b_1 x_1 + b_2 x_2 + ... + b_n x_n$$

Where:

a: is a constant.

b: is a coefficient of a property.

x: is a property.

n: number of properties.

y: represents the memory or the time complexity.

5.3 Datasets

In this section we will show the source of all datastes that we have used in our experiments. The Emotions, CAL500, Scene, Bibtex and Yeast datasets were taken from the Mulan repository [TVS10]. The Mammals dataset were taken from [HFEM07]. Oils and Oils2 are private datasets that describe specific characteristics of oil. Housing and helsinki datasets were taken from [ZML+15]. Wages dataset is taken from [Ber91]. Tic dataset is taken from [vdPvS00]. The rest of the datasets are taken from the UCI repository [Aha87].

	Dataset properties									Memory complexity (Mb)			
Datasets	Rows	Column	Binary	Float	Nominal	Description	Model	Density	Average cov.	Bool-Array	Uint32-Array	AutoMask	BitMask
contactlenses	24	5	3	0	2	4	1	0.5	0.30	1.67	1.67	1.67	1.67
zoo	101	18	15	0	3	16	1	0.43	0.56	9.26	9.39	9.14	8.35
oils2	118	32	0	32	0	30	2	0	0.61	17.19	20.13	17.12	15.3
oils	121	41	0	41	0	32	9	0	0.57	19.35	21.51	19.03	17.8
iris	150	5	0	4	1	4	1	0	0.55	14.05	14.25	13.49	11.84
hayesroth	160	5	0	4	1	4	1	0	0.46	5.34	5.87	5.31	4.82
glass	214	10	0	9	1	9	1	0	0.51	19.8	21.56	19.41	17.39
haberman	306	4	1	2	1	3	1	0.26	0.43	9.05	9.2	7.94	6.97
ionosphere	351	35	1	34	0	34	1	0.64	0.56	46.93	63.96	44.51	35.43
dermatology	366	35	1	1	33	34	1	0.12	0.47	25.29	25.95	23.39	19.39
CAL500	502	242	174	68	0	68	174	0.14	0.33	811.85	838.61	813.02	783.38
housing	506	14	1	13	0	13	1	0.07	0.56	30.59	41.23	30.89	22.3
wages	534	11	4	4	3	10	1	0.4	0.51	21.49	23.47	19.37	14.38
emotions	593	78	6	72	0	72	6	0.31	0.39	112.97	148.49	105.86	76.95
balancescale	625	5	0	4	1	4	1	0	0.47	20.11	21.87	18.22	14.07
soybean	683	36	16	0	20	35	1	0.28	0.48	35.19	37.33	32.28	24.56
credita	690	16	5	6	5	15	1	0.46	0.50	32.74	37.38	31.17	22.44
breast	699	10	1	9	0	9	1	0.34	0.48	30.45	30.24	28.78	20.71
wisconsin	699	10	71	9	0	9	1	0.34	0.48	30.35	30.26	28.75	20.63
pima	786	9	1	8	0	8	1	0.34	0.62	34.9	44.82	35.45	24.93
tictactoe	958	10	1	0	9	9	1	0.65	0.22	10.59	12.59	10.09	6.16
creditg	1000	21	3	7	11	20	1	0.68	0.52	42.12	50.07	38.45	26.29
cmc	1473	10	3	2	5	9	1	0.55	0.37	34.32	30.79	25.63	17.27
yeastuci	1484	9	1	7	1	8	1	0.01	0.66	54.82	71.82	53.16	34.14
redwine	1599	12	0	12	0	11	1	0	0.57	60.88	82.71	60.58	35.56
car	1728	7	0	0	7	6	1	0	0.15	6.61	6.61	6.08	3.65
crime	1994	103	0	102	1	102	1	0	0.34	277.3	401.07	249.99	110.14
scene	2407	300	6	294	0	294	6	0.17	0.29	927.23	1126.95	805.5	330.63
yeast	2417	117	14	103	0	103	14	0.3	0.34	412.8	479.75	363.98	189.45
mammals	2624	263	194	69	0	67	194	0.16	0.14	1067.09	1010.85	962.04	894.28
abalone	4177	9	0	8	1	8	1	0	0.63	138.44	171.69	131.04	71.43
spambase	4601	58	1	57	0	57	1	0.4	0.44	215.79	247.54	187.15	91.19
tic	5822	86	6	0	80	85	1	0.02	0.24	392.2	227.11	199.97	130.15
bibtex	7395	1995	1995	0	0	1836	159	0.03	0.28	6064.91	5888.15	4691.24	3737.54
mushroom	8124	23	5	0	18	22	1	0.35	0.41	189.88	154.96	120.85	68.62
helsinki	8337	24	1	23	0	23	1	0.16	0.44	358.72	410.3	310.55	133.62
nursery	12960	9	1	0	8	8	1	0.5	0.14	103.92	98.77	91.24	26.33
covtype	581012	55	44	10	1	54	1	0.04	0.75	17644.77	42248.8	17754.51	13230.82

Table 5.1: Memory complexity

	Dataset properties									Time complexity (seconds)			
Datasets	Rows	Column	Binary	Float	Nominal	Description	Model	Density	Average cov.	Bool-Array	Uint32-Array	AutoMask	BitMask
contactlenses	24	5	3	0	2	4	1	0.5	0.30	0	0	0	0
zoo	101	18	15	0	3	16	1	0.43	0.56	3	3	2	2
oils2	118	32	0	32	0	30	2	0	0.61	14	16	14	16
oils	121	41	0	41	0	32	9	0	0.57	18	19	18	19
iris	150	5	0	4	1	4	1	0	0.55	5	6	5	7
hayesroth	160	5	0	4	1	4	1	0	0.46	1	2	2	1
glass	214	10	0	9	1	9	1	0	0.51	16	17	17	19
haberman	306	4	1	2	1	3	1	0.26	0.43	2	2	2	2
ionosphere	351	35	1	34	0	34	1	0.64	0.56	70	72	66	79
dermatology	366	35	1	1	33	34	1	0.12	0.47	15	18	14	19
CAL500	502	242	174	68	0	68	174	0.14	0.33	275	312	329	240
housing	506	14	1	13	0	13	1	0.07	0.56	35	33	34	39
wages	534	11	4	4	3	10	1	0.4	0.51	13	14	15	16
emotions	593	78	6	72	0	72	6	0.31	0.39	233	239	240	408
balancescale	625	5	0	4	1	4	1	0	0.47	8	8	8	7
soybean	683	36	16	0	20	35	1	0.28	0.48	22	23	27	32
credita	690	16	5	6	5	15	1	0.46	0.50	23	23	19	26
breast	699	10	1	9	0	9	1	0.34	0.48	19	16	18	20
wisconsin	699	10	71	9	0	9	1	0.34	0.48	27	26	27	31
pima	786	9	1	8	0	8	1	0.34	0.62	49	47	46	44
tictactoe	958	10	1	0	9	9	1	0.65	0.22	2	2	2	3
creditg	1000	21	3	7	11	20	1	0.68	0.52	35	37	33	42
cmc	1473	10	3	2	5	9	1	0.55	0.37	10	12	11	13
yeastuci	1484	9	1	7	1	8	1	0.01	0.66	82	83	80	88
redwine	1599	12	0	12	0	11	1	0	0.57	106	109	98	134
car	1728	7	0	0	7	6	1	0	0.15	0	1	1	1
crime	1994	103	0	102	1	102	1	0	0.34	721	776	731	922
scene	2407	300	6	294	0	294	6	0.17	0.29	6126	6689	5489	6643
yeast	2417	117	14	103	0	103	14	0.3	0.34	1192	1183	1003	1700
mammals	2624	263	194	69	0	67	194	0.16	0.14	313	315	309	382
abalone	4177	9	0	8	1	8	1	0	0.63	137	139	148	167
spambase	4601	58	1	57	0	57	1	0.4	0.44	349	329	327	404
tic	5822	86	6	0	80	85	1	0.02	0.24	287	332	333	594
bibtex	7395	1995	1995	0	0	1836	159	0.03	0.28	11953	11735	10025	11964
mushroom	8124	23	5	0	18	22	1	0.35	0.41	83	88	82	106
helsinki	8337	24	1	23	0	23	1	0.16	0.44	843	936	770	772
nursery	12960	9	1	0	8	8	1	0.5	0.14	12	13	16	18
covtype	581012	55	44	10	1	54	1	0.04	0.75	159924	114590	148034	124789

Table 5.2: Time complexity

Table 5.3: Memory prediction measures

Technique	**CC**	**MAE**	**RMSE**	**RAE**	**RRSE**
Bool-Array	0.9177	389.0827	1458.055	32.09%	48.32%
Uint32-Array	0.8559	983.4749	4948.2707	40.40%	71.08%
AutoMask	0.9962	303.9214	1370.8304	26.60%	46.04%
BitMask	0.9454	293.4541	1458.3983	33.24%	65.36%

Table 5.4: Time prediction measures

Technique	**CC**	**MAE**	**RMSE**	**RAE**	**RRSE**
Bool-Array	0.0466	6245.634	27344.2606	71.20%	104.02%
Uint32-Array	0.0739	4828.361	20162.037	73.74%	106.90%
AutoMask	0.0388	5817.67	25327.8759	72.19%	104.12%
BitMask	0.0678	4869.3365	21271.5894	69.00%	103.62%

Table 5.5: Coefficients of linear regression for memory usage

Technique	Rows	Column	Binary	Float	Nominal	Description	Model	Density	Average cov.	Constant
Bool-Array	0.0301	2.9987	-0.1383	0	0	0	1.3751	0	161.7215	-127.837
Uint32-Array	0.0723	12.3185	11.9514	13.0591	9.5583	21.5983	-20.0462	0	557.0192	-378.1928
AutoMask	0.0303	7.4481	7.2308	7.6963	6.1331	-12.5492	-10.6527	0	197.0911	-138.1859
BitMask	0.0226	2.8116	0.5281	0	0	-1.7367	1.0882	0	215.8115	-136.738

Table 5.6: Coefficients of linear regression for runtime

Technique	Rows	Column	Binary	Float	Nominal	Description	Model	Density	Average cov.	Constant
Bool-Array	0.2743	58.4816	56.5281	69.5529	50.3251	-108.8191	-117.0519	0	2303.4789	-1753.8759
Uint32-Array	0.1965	2.6478	3.6623	17.9366	0	0	-9.0699	0	1565.5575	-1295.0137
AutoMask	0.254	43.512	41.9946	54.1435	37.4702	-80.3905	-86.8694	0	2151.5704	-1624.5976
BitMask	0.214	4.38	2.0072	16.7097	0	0	-9.1678	0	1634.5305	-1353.2298

Chapter 6

Discussion

In this chapter we will further discuss the results described in Chapter 5. We will discuss the memory usage, the runtime, cross-validation measures and the properties effects results.

6.1 Performance analysis

In this section we will discuss the memory usage results and the runtime results in Table 5.1 and Table 5.2. We will relate these results to the analyzing results from Sect. 4.1.

6.1.1 Analysis of memory usage

In Sect. 4.1 we have shown that the memory complexity of Bool-Array and BitMask techniques directly depends on the size of the dataset. For example, when we look at the *zoo*, *spambase*, and *covtype* datasets, which are arranged in ascending order according to dataset size, we note that *zoo* has the smallest memory usage while *covtype* has the largest memory usage. We have also demonstrated that BitMask memory complexity is the one eighth of Bool-Array memory complexity. For all the 38 datasets we see that the memory usage of BitMask is less than that one of Bool-Array. In Sect. 4.1 we have shown that the memory complexity of Uint32-Array depends directly on the number of covered examples in the dataset. We represent that by using the average covered example property. We note that most datasets in which this property is lower than 0.45, have less memory usage in Uint32-Array than that one in Bool-Array. Furthermore, most datasets in which this property holds a value higher than 0.45, have more memory usage in Uint32-Array than that one in Bool-Array. As we know, the AutoMask technique depends on the selection between Bool and Uint32 Arrays techniques. This selection is based on the number of covered examples in each subgroup. Therefore we note that the results of AutoMask technique can be better or at least not worse than the results of Bool-Array technique. We see that in the datasets with average covered example property lower than 0.45 have lower memory usage in AutoMask than in Bool-Array while the other datasets have about the same memory usage for both techniques.

6.1.2 Analysis of runtime

In Sect. 4.1 we have shown that the time complexity of all mask implementation techniques depends directly on the size of the dataset. We see that in the *covtype* dataset, which has a large number of rows with large runtime. We also note that not only the dataset size property plays a role in these results, but also the number of columns property. A large number of columns means a large search space, which leads to a large runtime. When we look at the *bibtex*, *yeast* and *scene* dataset results, we note that they have large runtime because they have a large number of columns. We also see that in general the BitMask runtime is bigger than the Bool-Array runtime. As we have shown in Sect. 4.1, the reason for this is that it takes more time to read and write a single bit in the memory than to read and write one byte in the memory.

Most of the dataset properties have an effect on the complexity results for all the mask implementation techniques. The dataset properties that we have shown are the most important, but obviously there are also other dataset properties that affect the complexity results, for example, binary description space and binary model space. Because we have nine dataset properties, it is difficult to determine the effect of each property on the end result. Therefore, we will look at the results of Sect. 6.3.

6.2 Predictive accuracy

Because we want to know the predictive quality of our models built with *Weka*, we have looked at the cross validation measures.
Table 5.3 gives us the cross-validation measures of the memory usage models for all the four implementations techniques. We note that the correlation coefficient results are close to 1. This means that the correlation between the true and the predicted value is high. For the other measures we can say the lower the number, the better the prediction. Therefore, we conclude that the AutoMask model is the best prediction model and Uint32-Array model is the worst. In general we observed that in all cases, the correlation coefficient is higher than 0.85 and therefore we conclude that we always manage to built a good memory usage prediction models for all the implementation techniques.
Table 5.4 shows the cross-validation measures of the runtime models. The results are not encouraging comparing with the results of Table 5.3. The reason for that is, predicting the runtime of a machine for a specific algorithm is difficult [Che92] [KSTW06] [Knu75]. Therefore we do not expect to get good prediction results using these models.

6.3 Properties effects

The models that we have built help us to know the effect of each property on the end result by using the linear regression equation.
Table 5.5 shows the effect of each property on the memory usage for all the implementation techniques. We note that the effect of the number of rows and number of columns properties is the largest on Uint32-Array while it is the smallest on BitMask. We also note that the effect of the average covered examples property on Uint32 is the largest while it is the smallest on Bool-Array. That is logical because we know that Uint32 depends directly on the number covered examples while this is not the case in Bool-Array. We can say if we have a large dataset size with a large average covered examples, the memory usage of Uint32 technique will be large.
Table 5.6 shows the effect of each property on the runtime results for all the implementations techniques. We note that the effect of the average covered examples property is big on the runtime for all the implementations techniques. The reason for this case is that, the cover selection strategy consumes more time to achieve the required task.

From the two tables we note that the density property plays no role on the end results. The reason is that the pattern language in all the implementation techniques uses the two values of binary columns. Therefore in our case the density property has no effect. It will be of interest if we use another pattern language. See future work Sec. 7.1.
According to the dataset properties and the information from the last two tables we can decide which mask implementation technique is the best for the memory usage and which one is the best for the runtime. For example, in case of memory usage, if we have a dataset with average coverage property of 0.15 then it is better to use the Uint32-Array than Bool-Array technique. For the most datasets using the BitMask technique give the better memory usage results.

Chapter 7

Conclusions

DSSD guarantee diversity and less redundancy subgroups. It functions well on a normal dataset size. But when the datasets become large and complex with more than 50000 rows or more than 300 columns, the memory usage and the runtime will increase.

The mask implementation techniques play an essential role in increasing, decreasing, or in trade-off between runtime and memory usage. We analyzed four mask techniques then we have conducted experiments on these techniques using 38 datasets.

According to the memory usage, the results of the experiments show how large the dataset size is and how small the average number of covered examples is, the Uint32-Array technique will perform in a better way. The reason for that is that the Uint32-Array keeps only the covered rows. The results also showed that in the opposed conditions, we better use the Bool-Array or the BitMask techniques rather than the Uint32-Array.

Regarding to the runtime, the results of the experiments show that Uint32-Array technique is better to use when the dataset has small number of covered examples. In the opposed condition, we better use the Bool-Array technique.

In general, the runtime of the BitMask technique is the largest, because long time is required in order to read and write a single bit in the memory.

Auto-Mask technique selects between the Bool and Uint32 Arrays according to the number of covered examples of a subgroup.

To know if the other datasets properties affect also on the results of the complexity, we have built prediction models using *Weka* tool [WF05]. The predictive accuracy measures show that the memory usage prediction models are more accurate than the runtime prediction models. The explanation is that the memory usage prediction models has correlation coefficient higher than 0.85 while the runtime prediction models has no more than 0.08 correlation coefficient. We can use these models to predict the complexity of new datasets which is useful to select a suitable mask implementation technique.

The effect of the average covered examples property on the memory usage of the Uint32-Array technique is the largest while it is the smallest on the Bool-Array technique. The effect of the covered example property is big on the runtime of all the implementation techniques.

The results of our project show that we can discover diverse subgroup sets from big datasets by switching between the four mask implementations techniques according to the datasets properties by using the information of the prediction models to get a reasonable runtime and memory usage. By this way, our research goal will be fulfilled and reached.

7.1 Future work

As a future work, we can suggest the following points:

1. Using another pattern language that consider only the positive value and discard the negative value. For example, if we have the pattern $\{A = 1 \text{ AND } B = 0\}$ then we will only consider the pattern $\{A = 1\}$ by discarding $\{B = 0\}$. In this pattern language the density property will be of interest. Therefore, we will expect to get different prediction results because the density property will affect the end result.

2. Implementing auto switching between all the implementation techniques. To do that, we use the information of the predictive models and apply it on the datasets properties of the given dataset to know which mask implementation technique has the best efficiency to use. We can therefore use this information to do auto switching.

3. In this paper we have looked at what happened when we cache all the subgroups covers to the memory but there are another possible methods to dealing with the subgroups covers. For example, calculate the subgroups covers each time we need them instead of keeping them all in the memory. We can also cache some of the frequent used subgroups covers and when the memory become full we remove the minder frequent used subgroups covers to free some of the space. In both examples, we want to investigate the complexity results.

Bibliography

[Aha87] D. Aha. UCI machine learning repository. http://archive.ics.uci.edu/ml/datasets.html, 1987.

[ALKH09] M. Atzmueller, F. Lemmerich, B. Krause, and A. Hotho. Who are the spammers? understandable local patterns for concept description. 2009.

[AP05] M. Atzmueller and F. Puppe. Semi-automatic visual subgroup mining using vikamine. *Journal of Universal Computer Science*, 11(11):17521765, 2005.

[AP06] M. Atzmueller and F. Puppe. Sd-map a fast algorithm for exhaustive subgroup discovery. *Knowledge Discovery in Databases*, 42(13)):6–17, 2006.

[Atz12] M. Atzmueller. Mining social media. *Data Mining and Knowledge Discovery*, 2(5):411–419, 2012.

[Atz15] M. Atzmueller. Subgroup discovery advanced review. *Data Min Knowl Disc*, 5(1):3549, 2015.

[Ber91] R. Berndt. *The Practice of Econometrics*. 1991.

[BPT+00] Y. Bastide, N. Pasquier, R. Taouil, G. Stumme, and L. Lakhal. Mining minimal non-redundant association rules using frequent closed itemsets. *Computational Logic CL 2000*, 1861:972986, 2000.

[Che92] P.C. Chen. A method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, 21(2):295315, 1992.

[GL02] D. Gamberger and N. Lavrac. Expert-guided subgroup discovery: Methodology and application. *Journal of Artificial Intelligence Research*, 17:501527, 2002.

[HCGdJ11] F. Herrera, C. Jos Carmona, P. Gonzlez, and M. Jos del Jesus. An overview on subgroup discovery: foundations and applications. *Knowledge and Information Systems*, 29(3):495525, 2011.

[HFEM07] H. Heikinheimo, M. Fortelius, J. Eronen, and H. Mannila. Biogeography of european land mammals shows environmentally distinct and spatially coherent clusters. *J Biogeogr*, 34(6):10531064, 2007.

[HPK12] J. Han, J. Pei, and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, third edition, 2012.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *ACM SIGMOD Record*, 29(2):1–12, 2000.

[JFW+14] N. Jin, P. Flach, T. Wilcox, R. Sellman, J. Thumim, and A. Knobbe. Subgroup discovery in smart electricity meter data. *IEEE Transactions on Industrial Informatics*, 10(2):13271336, 2014.

[Klo96] W. Kloesgen. A multipattern and multistrategy discovery assistant. *Advances in Knowledge discovery and data mining*, page 249271, 1996.

[KM02] W. Klosgen and M. May. Spatial subgroup mining integrated in an object-relational spatial database. *Principles of Data Mining and Knowledge Discovery*, 2431:275286, 2002.

[Knu75] D.E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of computation*, 29(129):122136, 1975.

[KSTW06] P. Kilby, J. Slaney, S. Thiebaux, and T. Walsh. Estimating search tree size. *In Proceedings of AAAI06*, page 10141019, 2006.

[LKFT04a] N. Lavrac, B. Kavsek, P. Flach, and L. Todorovski. Subgroup discovery with cn2-sd. *Journal of Machine Learning Research*, 5:153188, 2004.

[LKFT04b] N. Lavrac, B. Kavek, P. Flach, and L. Todorovski. Subgroup discovery with cn2-sd. *J Mach Learn Res*, 5:153188, 2004.

[NP14] M. Natu and G. Palshikar. Interesting subset discovery and its application on service processes. *Data Mining for Service*, 3 of Studies in Big Data:245269, 2014.

[PBTL99] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. *Conference on Database Theory (ICDT 99)*, 1540:398–416, 1999.

[Puj01] A. K. Pujarir. *Data Mining Techniques*. Universities Press, first edition, 2001.

[TVS10] G. Tsoumakas, J. Vilcek, and L. Spyromitros. Mulan: a java library formulti-label learning. 2010.

[vdPvS00] P. van der Putten and M. van Someren. Coil challenge 2000: The insurance company case. *Sentient Machine Research*, 9:1–43, 2000.

[vL12] M. van Leeuwen. Patterns that Matter. http://www.patternsthatmatter.org/, 2012.

[vLK12] M. van Leeuwen and A. Knobbe. Diverse subgroup set discovery. *Data Min Knowl Disc*, 25(2):208–242, 2012.

[WF05] I. H. Witten and E. Frank. *Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2nd edition, 2005.

[Wro97] S. Wrobel. An algorithm for multi-relational discovery of subgroups. *Proceedings of the 1st European symposium on principles of data mining and knowledge discovery*, 12(63):7887, 1997.

[ZML+15] I. Zliobaite, M. Mathioudakis, T. Lehtiniemi, P. Parviainen, and T. Janhunen. A case study in helsinki region. *Mining Urban Data*, 1392:65–71, 2015.