SLIMME Vis - Interactive visual candidate exploration for SLIM

Matthias Müller-Brockhausen; s2084740

Universiteit Leiden

Abstract. In the Field of ITDM algorithms, such as SLIM, optimize compression based on the Minimum Description Length. They apply heuristics to cope with an otherwise incalculable size of options. This paper sheds some light on the effect of these heuristics by offering a website that enables the user to interactively influence the heuristics decision and directly see the effect on the achievable compression and built code table.

Keywords: Visualization · SLIM · Tree · Code Table · Candidate Exploration

1 Introduction

Information Theoretic Data Mining (ITDM) can seem like an overwhelming topic at first glance. Requiring good knowledge of the basics of statistics and then applying these intelligently to be able to mine *any* kind of data can seem daunting to newcomers. SLIM[6] an ITDM algorithm that optimizes compression by means of the Minimal Description Length (see Section 2) to compress transaction data is used as an example to ease the reader into the world of ITDM. The algorithms *heuristics* build a sorted candidate list (see Section 2.1) that **determines** the achievable compression of the algorithm.

Inspired by "Seeing Theory"[8], a website that interactively explains the basics of statistics, a web tool that allows the user to influence said *heuristics* decision is implemented for this paper. To make the web tool more accessible as a stand alone tool and also more closely resemble "Seeing Theory" the SLIM algorithm and its preliminaries are also visually explained on the tools website. Furthermore studies have proven that optimized visualizations and interaction (see Section 5.2) helps accelerate and deepen the understanding of theoretical concepts[3] such as the ones required for SLIM.

The closest work to this in the field of ITDM is *SPECTRA*[5] which allows estimating the number of frequent item sets and comparing that estimation with the actual amount via a line graph. This however does not feature an explanation of the algorithms, or offer a lot of interactiveness besides choosing a dataset to calculate on.

The programmed tool for the paper including it's source code is available online[1][2].

2 Preliminaries

The following is a concise overview of the basics and terms used throughout the paper. The same content is also explained visually and interactively on the produced website. These preliminaries try to only mention things relevant to understanding the problem statement of Section 3. For a complete explanation on the theory behind the algorithm, the original KRIMP paper[9] and its extension SLIM[6] are best suited as the following sections are based upon them.

SLIM works on a database D. A database is made up of transactions t. Each transaction is a subset of i $t \subseteq I$. A pattern language L also consists of sets of items like a transaction. A transaction can be summarized by item sets X, in which every single item may only occur once $X \subseteq I$. SLIM builds a code tables CT. A code table is made up of item sets X on the left side accompanied by a relevant numeric value concerning the set at hand. The website displays the *usage* of a set on the right hand side.

One value that is constant for a set given a database is its support. The support of a set is calculated by counting the number of transactions which contain X:

$$support_D(X) = |\{t \in D | X \subseteq t\}|. \tag{1}$$

The *usage* of a subset is defined as:

$$usage_D(X) = |\{t \in D | X \in cover(CT, t)\}|. \tag{2}$$

The usage is determined by covering the full database D. To cover a database take the ordered sets of a code table CT.

{1, 2} {1} {2} {3} {4}

To cover a transaction, one starts at the left. For every set check whether it is a subset of the transaction. If so it is added to the cover, and the values within the subset are removed from the transaction. This process is repeated, until the transaction is fully covered. A cover for the example would look like this:

Transaction: 1 2 3 4
Cover: {1, 2} {3} {4}

Covering the database is essential for calculating the usage (Equation 2).

Since the cover of a transaction is done from left to right or top to bottom, the order of the sets of items I is of importance! A badly ordered list could result in a suboptimal cover:

{1} {2} {3} {4} {1, 2}
Transaction: 1 2 3 4

[1] Demo hosted on GitHub Pages at https://hizoul.github.io/slimme-vis/

[2] Source code hosted on GitHub at https://github.com/Hizoul/slimme-vis

Cover: {1} {2} {3} {4}

SLIMS uses the so called *standard cover order* portrayed in Figure 1. In words: The longest subsets should be first. Along the same length of subsets the ones with the highest support come first. If those two criteria are equal, then just sort them lexicographically.

$$|X| \downarrow \quad supp_D(X) \downarrow \quad \text{lexicographically} \uparrow$$

Fig. 1: The *standard cover order* used in KRIMP and SLIM to sort a code table

Thanks to Shannons Entropy one can convert the usage into a probability distribution:

$$P(X|D) = \frac{usage_D(X)}{\sum_{Y \in CT} usage_D(Y)} \tag{3}$$

This in turn allows the assignment of a code length for each of the set in the code table CT (more frequent = shorter code).

The individual set code lengths can be combined to calculate the required amount of bits to encode a full database D via a code table CT

$$L(D|CT) = \sum_{t \in D} \sum_{X \in cover(CT,t)} -log(P(X|D)) \tag{4}$$

The goal of SLIM is to minimize this code length through the Minimum Description Length (MDL):

$$L(H) + L(D|H) \tag{5}$$

$L(H)$ is the amount of bits required to represent the description / code table CT itself. $L(D|H)$ is the amount of bits the Database can be described in / encoded with by using the code table CT.

2.1 Candidates

SLIM builds a list of candidate sets F. A candidate is a set of items, that is probably a frequent pattern. A frequent pattern is defined as a pattern which support passes a user defined *minimum support threshold* (minsup).

$$\{X \in L | supp_D(X) \geq minsup\} \tag{6}$$

Each candidate is considered in order. SLIM adds it to the code table then tests whether the code length to encode the whole database has reduced or not. If it reduced keep the candidate in the list and test the next one.

SLIM orders its candidates using the Gain Order. The gain is an approximation to the probable impact on the encoding length of the code table if the candidate were added.

The order of the candidates is of great importance, for similar reasons as for the cover order explained in Section 2. First adding one candidate might make the second candidate not reduce the code length any more, although it could have were it added first.

To numerically determine the quality of a given code table the relative compression is used. That is the percentage of saveable bits if it is encoded with the given code table instead of the standard code table.

$$L\% = \frac{L(D, CT)}{L(D, ST)} * 100 \tag{7}$$

3 Problem Statement

As mentioned in Section 2.1 the order of the code table CT and the order of the candidates F make or break the found optimum of the algorithm. With respect to runtime, it is infeasible to try out all possible orders, which is why the heuristic is needed. The achievable relative compression is in turn determined by the order that the heuristic produces. In the KRIMP paper, a variation called KRAMP is presented. This variant still uses the same order, but it always explores all possible paths, instead of only taking the best option. The runtime of course rose to be impractical, but the achieved compression was *slightly* better than the one of KRIMP. Meaning using alternative paths, that don't seem optimal to the heuristic can result in better code tables. On the question of how to incorporate subjective interestingness, Tijl de Bie states:

"[...]this can only be done if the data miner (the user) is an integral part of the analysis, considered as much as the data and the patterns themselves. More specifically, to understand what is interesting to a user, we will need to understand how to model the beliefs of that user, how to contrast a pattern with this model, and how that model evolves upon presentation of a pattern to that user"[1]

These problems are tried to be solved by providing the interactive candidate space explorer shown in Figure 2 and 3. Every time the candidates of F are ordered, a list of up to 10 candidates is shown together with the final compression ratio, if the candidate were added to the code table at this point in time. The final compression ratio means, adding the candidate shown, but then just letting the SLIM algorithm finish without further user intervention. Every candidate can be clicked to explore the next ten best candidates and further refine the candidates that influence the relative encoding. With this functionality the user can get a better understanding of the actual effect of the orders, and by choosing himself also gets a chance to apply his previous beliefs to the outcome of the code table. Additionally the analyst can vary the sorting between descending and ascending gain order as well as a random order to underline the effect of the order on the relative compression.

4 Results

The following experiment is made on a small prepared dataset, which is displayed (and modifiable) within the tool. The effect of the achievable compression (see Equation 7) already becomes clear by merely switching between the different offered sortings (namely Gain Descending, Ascending, Random). The standard descending gain order finds its optimum at a compression rate of 30.48%. Using a random sort however shows that the compression can go as low as 26.67% as evident from Figure 2. Since the order is random it might take a few tries of selecting the random sort to reproduce the result.

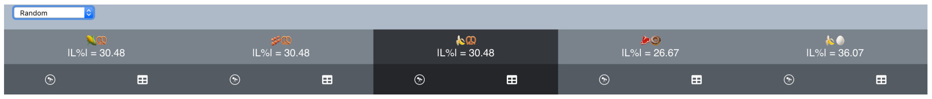


Fig. 2: The candidate explorer, displaying up to 10 candidates from F. Random sort is applied in this instance.

Manually exploring within the descending gain order, as shown in Figure 3, reveals that one has to use options that wouldn't be considered by the heuristic multiple times in order to achieve a compression of 26.67%. The ascending gain order, as to be expected, delivers the worst result with 63.35%. With random similarly bad results are possible.

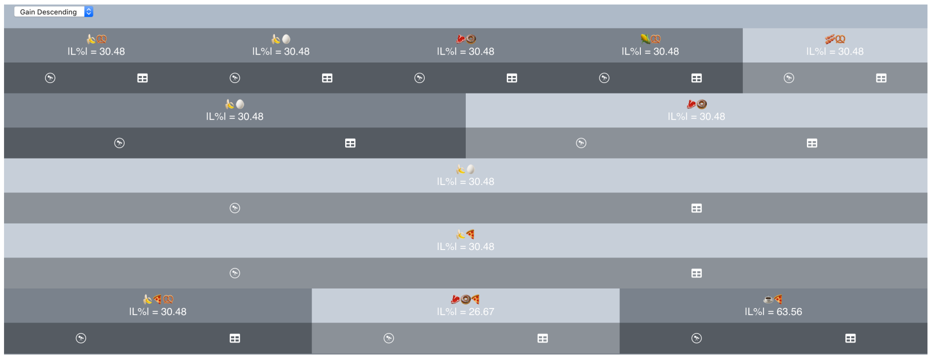


Fig. 3: The candidate explorer, displaying up to 10 candidates from F. The standard descending gain order is applied, and a custom path chosen to reproduce the achieved compression of a lucky random exploration displayed in Figure 2.

5 Discussion

5.1 Algorithm

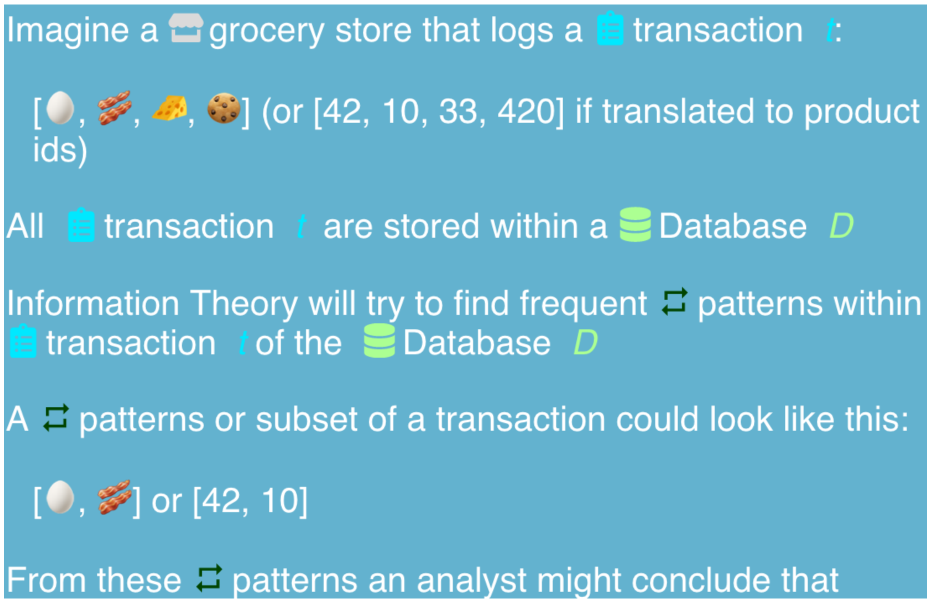
A problem encountered during implementation was that, that the post-prune mechanism does not verify if the code table is still valid. Some relative compressions were so good that they seemed to good to be true. And in fact they were because they could not fully cover every transaction of the database. Furthermore an endless loop of pruning and then re-adding the same pair over and over could happen. To solve this problem, the post-prune mechanism was extended to check, whether removing a set from the code table renders it invalid. Only if it is still valid the item will actually be pruned.

The implemented candidate exploration tool also offers some room for improvement. It would for example be beneficial for processing larger datasets if a specific pattern could be specified and the candidate space exploration only starts as soon as that specific pattern is reached. Without such a functionality it could be a very slow process to model the users belief onto a large dataset. Similar to that one could also say, do x iterations of SLIM and from that state on forward let me do the interactive candidate exploration.

5.2 Maximizing Learning Efficiency

As this paper is providing an explanatory website for the basics required for SLIM, it is relevant to assess the possibilities this offers in contrast to a traditional scientific paper. The field of psychology has long known, that there are certain features, our brain can process without any sort of focus via our subconsciousness. Some of the most prominently known *preattentive features* are color, orientation[2] and shape[7]. In 1993 Janiszewski has shown, that by implementing these preattentive features into advertisements, the probability that people subconciously remember a brand name while scanning through a newspaper is increased[4]. An interesting aspect of these preattentive features is that they can be processed in parallel, similar to multithreading on a computer, which makes it easier for people skimming through a document to see relevant parts[2]. However it is important to note, that using too many preattentive features at once, can actually decrease the subjects unconcious processing, and requires more manual focus[10]. Building upon this knowledge, every variable introduced in the preliminaries Section 2 (e.g. transaction t) is associated to a specific color and icon. An example of this can be seen in Figure 4.

Furthermore for every item i that occurs in the sets of transactions and code tables, a *translation* is used to translate the item id into a shape. That way while exploring possible candidates the user can more quickly recognize a path of items he would like to follow.



Imagine a grocery store that logs a transaction *t*:

[🥚, 🥓, 🧀, 🍪] (or [42, 10, 33, 420] if translated to product ids)

All transaction *t* are stored within a Database *D*

Information Theory will try to find frequent patterns within transaction *t* of the Database *D*

A patterns or subset of a transaction could look like this:

[🥚, 🥓] or [42, 10]

From these patterns an analyst might conclude that

Fig. 4: An example of color coding keywords in the explanation to enhance recognizability.

6 Conclusion

An interactive learning experience, optimized for learning efficiency (Section 5.2), has been created to ease the user into the world of ITDM at the example of SLIM. Furthermore the candidate space exploration of SLIM has been made interactive and now allows the analyst to influence the result according to his beliefs (Section 3). Section 4 underlined how the default heuristic used in SLIM only produces a local optimum. This was proven by showing that the default sort finds a relative compressiomn of 30.48% whereas a manual exploration and also a completely random one can achieve up to 26.67%. Additionally for people that wish to reimplement and play around with SLIM, further details relevant for a working implementation are shared in Section 5.1.

References

1. De Bie, T.: Subjective interestingness in exploratory data mining. In: International Symposium on Intelligent Data Analysis. pp. 19–31. Springer (2013)
2. Healey, C.G., Booth, K.S., Enns, J.T.: High-speed visual estimation using preattentive processing. ACM Transactions on Computer-Human Interaction (TOCHI) **3**(2), 107–135 (1996)
3. Hegarty, M.: Dynamic visualizations and learning: Getting to the difficult questions. Learning and Instruction **14**(3), 343–351 (2004)

4. Janiszewski, C.: Preattentive mere exposure effects. Journal of Consumer Research **20**(3), 376–392 (1993)
5. van Leeuwen, M., Ukkonen, A.: Estimating the pattern frequency spectrum inside the browser. arXiv preprint arXiv:1409.7311 (2014)
6. Smets, K., Vreeken, J.: Slim: Directly mining descriptive patterns pp. 236–247 (2012)
7. Treisman, A., Gormican, S.: Feature analysis in early vision: evidence from search asymmetries. Psychological review **95**(1), 15 (1988)
8. Tyler Devlin, Jingru Guo, D.K.D.X.: Seeing theory; visited on 16.11.2018; https://seeing-theory.brown.edu/
9. Vreeken, J., Van Leeuwen, M., Siebes, A.: Krimp: mining itemsets that compress. Data Mining and Knowledge Discovery **23**(1), 169–214 (2011)
10. Wolfe, J.M.: The parallel guidance of visual attention. Current Directions in Psychological Science **1**(4), 124–128 (1992)