J.E. Witteveen

Mining hyperintervals
Getting to grips with real-valued data

Bachelor's thesis, 28 August 2012

Supervisors: Prof. Dr. P.D. Grünwald, Dr. A. Knobbe &
W. Duivesteijn

Mathematisch Instituut, Universiteit Leiden

Contents

Abstract

Many uses of data mining, such as clustering, classification, the construction of decision trees, subgroup discovery and itemset mining, often fail to be able to cope with real-valued data well. In fact, it is common for data mining methods to only work well on nominal data with little different values. We build the theory to fill this gap for data from arbitrary uncountable sets and introduce an efficient method to mine data, without the usual discretization as a pre-processing step. It is shown that discretization is not needed in order to make use of the MDL principle.

1 Introduction

We introduce the topics this thesis will be concerned with. An overview of the contents of this thesis is given, so that the reader can skip some or all of the thesis, depending on his/her (current) interests.

Put bluntly, data mining is concerned with finding regularities in data. There are lots of sources of data and likewise lots of niches in data mining. We will be concerned with data collected as entries in a database. To a statistician, a database is a record of values taken by a, possibly unknown, random variable. This view implies the applicability of a lot of theory, mainly from measure theory, even without assuming the existence of a fixed distribution that is generating the data. Remarkably, very little of the data mining literature explicitly tries to harness measure theory. Additionally, although mining nominal data is increasingly well explored in data mining research, the mining of data from sets with high cardinality is still a bit problematic.

Statistics provides methods to fit Gaussian mixture models, which can lead to successful classifiers, but some forms of data mining desire a different kind of output. In both the construction of decision trees and subgroup discovery, the data miner is interested in boundary values of, possibly interacting, attributes that capture some sort of regularity in the data. The same goes, in some sense, for the construction of histograms.

Inspired by Krimp's success at using the Minimum Description Length principle in frequent itemset mining, we set out to construct an equally advanced mining scheme for mining regularities in data from uncountable sets. Certainly, looking at measure theory helps in setting up the theory behind such a scheme.

This thesis is split into three parts, the first of which is purely theoretic in nature and of little interest to someone merely interested in practicing data mining, although Section 2 does contain an attempt at defining what data mining is. Section 3 is a bit of an odd one out in this thesis, as it's essentially philosophical. Read it if you want to have something to disagree on or take interest in complexity notions and Turing machines. The fourth section is the meat of Part I. It is full of theorems and even contains integrals. The mathematically inclined might enjoy the derivation of a generalisation of the Kraft–McMillan theorem.

Part II takes the theory of Part I and puts it into something useful. Section 5 and Section 6 are rather general and comprise, as their titles state, a simple model and a simple model class. The next section, Section 7, contains all the gory details of our novel data mining algorithm. Only those who take a sincere interest in the new method are advised to delve into this section.

The last part is where all, but one (hint: page 21), of the figures are. People who want to know if the new data mining method is actually any good might appreciate this part.

Part I
Theory

2 The concept of random variables

We give a definition of data mining within the paradigm of statistics. Also, we introduce random variables as the source of data for a data miner and contrast two definitions for random variables.

As in any investigation of statistics we start out with some space of interest, a universe Ω that behaves like a set. Common (for instance in [3,8,13]) definitions of random variables are more demanding with regard to Ω and take on a form equivalent or similar to Definition 1.

Definition 1. Let $(\Omega, \mathcal{F}, P)$ be a probability space. A *random variable* X is a function $X : \Omega \to \mathbb{R}$ that induces a measure on a measurable space $(\mathbb{R}, \mathcal{E})$.[*]

An obvious peculiarity of this definition is the fixation of $\mathbb{R}$. As we will see further on this choice restricts the definition of parameters and their estimates in statistical models and would eventually limit the use of parameter estimation as a means of data mining. This shortcoming was recognized by [6], who went on to redefine random variables as per Definition 2.

Definition 2. Let $(\Omega, \mathcal{F}, P)$ be a probability space and $(\Psi, \mathcal{E})$ a measurable space. A $(\Psi, \mathcal{E})$*-valued random variable* X is a function $X : \Omega \to \Psi$ that induces a measure on $(\Psi, \mathcal{E})$.

To see how random variables can be made of paramount importance to data mining, we have to agree on a definition of data mining. A very broad (and vague) definition is proposed by [14], including very general notions of sample data. The definition of data mining lets data mining be any technique by which data can be examined. However appealing this may seem from a real-world point of view, a far narrower definition is needed for this thesis. In its most constrained form we could state that data mining is the utilisation of a statistic, where a statistic is any function on the sample of a random variable[†]. As pointed out by [4], however, there is a great deal of subjectiveness to be taken into account when dealing with data mining. Data mining as a research field is not less about practitioners of data mining as it is about (mining) data. We settle with the view that *data mining is about the formation and application of a statistic by a data miner, where the statistic might be parameterized by the miner's prior understanding of the data and/or the applied statistic might be subjected to post-hoc treatment based on that understanding.* This definition makes data mining tractable for statisticians, while keeping it open for philosophical discussion regarding the subjective nature of gained knowledge. Without delving too deep into philosophy we note that if we take a statistic to be a kind of a random variable, we are probably better off with Definition 2 than with Definition 1

[*] There are a lot of unspecified variables in this definition. Their impact is of little importance to this thesis and is thus left out of consideration. It is sufficient to state that sensible choices lead to sane situations.

[†] Equivalently, we could say that a statistic is itself a random variable.

since it allows for mined information to be in a far more intuitive format than plain real numbers, as Example 1 shows.

Example 1. Let Ψ be the set of all products some supermarket sells. In a classical example of data mining, Ω represents the transactions done in that supermarket and a random variable maps each transaction to a subset of Ψ, based on which products are involved in the transaction. So if someone goes into the supermarket in the morning of a given date and buys two chocolate bars of brand `x` and an edition of the `y`-newspaper, and pays in cash, the random variable would map the transaction to the set {`chocolate/x`, `newspaper/y`}. This random variable induces a measure on $(\Psi, 2^\Psi)$, based on the sales of combinations of products.

A possible statistic that would make up the process of data mining is that given a subset of Ψ, we determine how often each product in that subset is present in the dataset consisting of images of the random variable for all recorded transactions. In this case, the subset functions as a parameter of the statistic. If the data miner goes on to only look at the items in the subset that are present more than a certain number of times in the dataset, this would be a form of post-hoc treatment.

3 Towards the MDL principle

A philosophical-mathematical foundation for the MDL principle as a means of data mining is given. In doing so, we arrive at a stronger version of the Church–Turing thesis. The main result of this section will be that we regard the MDL principle as a more suitable formalism for data mining than traditional statistical methods.

A common technique that adheres to the above definition of data mining is standard statistical inference. In statistical inference the involved statistic is mostly an estimator estimating a parameter of a statistical model or a model in a model class*. The choice of the model or model class make up for part of the prior knowledge of the data miner. When regarded in this context, it is worthwhile to take note of the principle of linguistic relativity, also known as the Sapir-Whorf hypothesis, which hinges between being a real hypothesis and being an axiom. In its weak form it is stated as follows by [2].

Axiom 1 (weak linguistic relativity). *Structural differences between language systems will, in general, be paralleled by non-linguistic cognitive differences, of an unspecified sort, in the native speakers of the two languages.*

Though initially concerned with natural languages, the axiom has been adopted by researchers and users of artificial languages such as mathematics and programming languages. In those formalized settings, the non-linguistic differences can often be made precise and provable, turning Axiom 1 into a theorem for those differences.

We can use Axiom 1 in our setting to pinpoint a discrepancy between data mining as perceived by the data miner and data mining as looked upon by the information theorist. However fitting the view of data mining we have come forth with, it may fail to capture the methods of the data miner. As Goethe has

*We use the nomenclature of [8].

put it: “Mathematicians are like Frenchmen: whatever you say to them they translate into their own language, and forthwith it is something entirely different.”[*] It is thus apparent that thinking of statistical inference as a data mining application means working in the ways of statistical inference when statistical inference is the application of data mining at hand. The problem with this is not so much that statistics will not mine data, but that it might not fit the conceptual framework of the data miner enough. This carries the risk of employing an unnecessarily complicated formalism. A report on a statistically relevant (or: statistically interesting) feature of some data might not be interesting to a particular data miner mining that data. Although [7] discerns several aspects of ‘interestingness’ and tries to mold those aspects into some sort of a definition, in the end, what is interesting differs from data to data and from data miner to data miner. In Example 1, the data miner appeared interesting in sales numbers for products in a given set, which does not say another data miner investigating the same data would deem that information interesting too.

One of the results that fortify the claim that statistics might not fit the conceptual framework of the data miner well is the list of no less then 39 probability based interestingness measures gathered in [7]. The article takes a pessimistic standpoint and states that selecting a suitable interestingness measure is an inherent difficulty of data mining. In an attempt to haggle a bit on this stance, we must take a closer look at ‘interestingness’. This has been done quite a few times before in the data mining literature. Both [4] and [7] are written at least partly for this reason. The following approach is, however, a bit unconventional. As it turns out, we can find a formalism, a ‘language system’ in the idiom of Axiom 1, that matches that (the language system) of a human data miner in general, thus making everything interesting to that formalism interesting to the data miner, regardless of the precise meaning of ‘interesting’.

To start with, we mention the Church–Turing thesis on effective computability, which again is more of an axiom than a hypothesis. From here on in this chapter, when we talk about partial functions we will be talking about partial functions with a fixed range R and as their domain the set of strings over a fixed, finite alphabet $\mathcal{A}$.[†] In our formulation ‘effectively calculable’ means humanly producible, in the intuitive sense. Partial recursive functions are often defined as the partial functions corresponding to the behaviour of Turing machines, but all one would ever need is that they form a recursive set.

Axiom 2 (effective computability)**.** *The class of effectively calculable partial functions coincides with the class of partial recursive functions.*

What constitutes ‘effectively calculable’ is a delicate matter, but not the only gripping aspect of the axiom. Although it seems to equate human computation with a formal abstraction, Axiom 2 only does so for the reachability of the results of such computation. In order to make use of Axiom 2 for our cause, we need more. Before we formulate our last axiom, we remind ourselves of a central theorem in the study of Kolmogorov complexity. In Kolmogorov complexity, every partial recursive function f has associated with it a complexity function

[*] “Die Mathematiker sind eine Art Franzosen; redet man zu ihnen, so übersetzen sie es in ihre Sprache, und dann ist es alsobald ganz etwas anders.” Johann Wolfgang von Goethe, Maximen und Reflexionen.

[†] In light of the subject this is a very common and not at all too prohibiting restriction. It certainly is in accordance with the views expressed in Turing’s 1936 paper.

$C_f : R \to \mathbb{Z}_{\geq 0} \cup \{\infty\}$ that assigns to an object r in the range of f the length (in characters of $\mathcal{A}$) of the shortest argument x for which $f(x)$ maps to r, or infinity if no such x exists. As in chapter 2 of [13], we fix a partial order on complexity functions C_f and C_g by $C_f \leq C_g$ if and only if there is a constant $c \geq 0$ such that for all r we have: $C_f(r) \leq C_g(r) + c$. This ordering gives rise to ordered equivalence classes of complexity functions. The theorem now states that there is a least element, called additively optimal, among these classes.

Theorem 1 (invariance of descriptive complexity)**.** *There is an additively optimal class of complexity functions belonging to partial recursive functions.*

This theorem is proven and explored extensively in [13]. What we take from it is that we can speak of 'descriptive complexity' and be precise in what we say up to an additive constant. In any case this constant is determined by the reference class representative of the additively optimal class and any class representative we wish to compare it with.

Having a formal view on descriptive complexity we can now augment Axiom 2 with a statement on the procedural background of human computation.

Axiom 3 (perceived descriptive complexity)**.** *Every humanly perceived descriptive complexity, for as far as it can be seen as a function, is in the additive optimal class of complexity functions for partial recursive functions.*

Although not designated as such, this axiom is a considerable part of the foundation of the minimum description length principle. The basic idea in [8], going back to Rissanen, is that learning from data is the same as becoming able to compress it. The one reasonable interpretation of compression in this view would relate to descriptive complexity, which reduces the idea to an intuitive formulation of Axiom 3. This approach to the MDL principle makes for a novel (not discerned in [8]) argument for the use of the MDL principle. Namely, we see the MDL principle come forth as a principle that naturally fits human perception. This does not say that MDL would be useless without accepting Axiom 3. Without the axiom, it still is a branch of statistics in general and its methods might have benefits over other statistical methods. Within the 'learning equals compressing' paradigm, however, we need everything that is informative to humans in a compressing sense to be compressing in a mechanical sense too (humans cannot compress better than machines), thus, by Theorem 1, we need Axiom 3.

With this insight we can expect that using the MDL principle, our concerns on interestingness will be covered. This is mainly on account of Axiom 1, that now holds that employing a system that is faithful to our measure of informativeness will have our measure of significance seen mirrored in its conclusions.* That is: there are no 'non-linguistic cognitive differences' (or, equally vague: differences in perceived interestingness) between the human data miner and the MDL principle. In particular, the MDL principle, as a modus operandi, should be preferred over standard statistical inference, our first example of a data mining application, whenever it can be used instead. When estimating parameters

* Additionally, when Axiom 3 is accepted, the differences mentioned in Axiom 1 are bounded (additively) in magnitude. Indeed, this means that in fact, any formalism could be used for data mining, especially since the MDL formalism is not in the additively optimal class of Theorem 1. However, our reasoning shows that, put informally, comparing the human data miner and the MDL principle, the differences mentioned in Axiom 1 are very small.

of a model or models in a model class, the MDL principle provides straightforward means of accounting for model (class) complexity. Moreover, in a very plain manner, estimators become variables in MDL based formulas. In this setting it is possible to exploit the generality of Definition 2, for often real-world compression is based on things like dictionaries instead of the reals.

There is, however, no free lunch and the MDL principle has its drawbacks in data mining too. Not only might the MDL principle not always provide a useful guide in data mining practice, there is also a very prominent theoretical impediment. A point that is stressed perhaps too little in [13], is that the constant in the partial order defined on complexity functions earlier is not bounded. Even in the additive optimal class of Theorem 1 the constants are, in general, large or even extremely large. Here, 'large' refers to the total complexity of a use case. As in statistics, we often want good results fast, meaning with little data available. Especially in these cases, it is of paramount importance to pick a reference partial recursive function carefully. Ideally, we want a reference partial recursive function such that the constant that arises from comparison with the human perceived descriptive complexity is very small. If we are not only interested in finding any regularity in the data, but pursue finding as many as possible, we would even want the constant arising from comparison the other way round to be small too.

4 Discretization and the Kraft inequality

In this section we introduce some information theoretic concepts. Using measure theoretical considerations, we manage to generalize the Kraft–McMillan theorem so that it is applicable in uncountable spaces.

4.1 At most countable spaces

Let $(\Omega, \mathcal{F}, P)$ be a probability space where Ω is at most countable. We will look at the smallest possible 'events' with non-zero probability:

$$\mathcal{E} := \{A \in \mathcal{F} : P(A) > 0, \forall B \subset A : B \notin \mathcal{F}\}.$$

Since every two elements of $\mathcal{E}$ are necessarily disjoint ($\mathcal{F}$ is a σ-algebra), $\mathcal{E}$ too is at most countable. It is easiest to think of $\mathcal{E}$ as singletons in Ω with positive probability, because in many cases, among which Example 2, that is precisely what $\mathcal{E}$ is. Next to $\mathcal{E}$, we will use a finite alphabet (set of characters) $\mathcal{A}$ and a uniquely decodable* subset $\mathcal{C}$ of $\mathcal{A}^\star$, where $\mathcal{A}^\star$ denotes the strings ('$\star$' is the Kleene star) over $\mathcal{A}$, called a *code*.

In its usual interpretation, the minimum description length principle utilizes the concept of a decoding function: a surjection $D : \mathcal{C} \twoheadrightarrow \mathcal{E}$. Additionally there is a length function $\ell_{\mathcal{A}} : \mathcal{C} \to \mathbb{Z}_{\geq 0}$ that assigns to each string its length in characters of $\mathcal{A}$.

Lengths are important to us, as they are related to probabilities through the Kraft–McMillan theorem.

* A set of strings is called *uniquely decodable* if every string that is a concatenation of strings from that set has a unique decomposition within that set. Note that in this thesis, a code is necessarily uniquely decodable.

Theorem 2 (Kraft–McMillan theorem). *For every code $\mathcal{C}$ over an alphabet $\mathcal{A}$ there exists a, possibly defective, probability mass function p on $\mathcal{C}$ that makes short lengths and high probabilities of the strings in the code correspond as follows:*

$$p(s) = |\mathcal{A}|^{-\ell_{\mathcal{A}}(s)}.$$

The theorem hinges on the Kraft–McMillan inequality, for which proofs of both the case where $\mathcal{C}$ is finite and the case that $\mathcal{C}$ is (countably) infinite can be found in [3].

Example 2. Consider the probability space $(\Omega, \mathcal{F}, P)$, where $\Omega = \mathbb{Z}_{>0}$, $\mathcal{F} = 2^{\Omega}$ and P is fixed by its definition on Ω as $P(z) = 2^{-z}$. In this case $\mathcal{E}$ consists of all the singletons of Ω, namely: $\mathcal{E} = \{\{z\} : z \in \Omega\}$.

If we settle on the three-character alphabet $\mathcal{A} = \{\mathtt{0}, \mathtt{1}, \$\}$, we can define a code $\mathcal{C}$ as all non-empty strings from $\{\mathtt{0}, \mathtt{1}\}^{\star}$ followed by a \$. This way, the \$ functions as a stopping character, making $\mathcal{C}$ uniquely decodable. There is a straightforward decoding function, D, into $\mathcal{E}$, where the string before the \$ is interpreted as the binary representation of a number n and D maps n to $\{n+1\} \in \mathcal{E}$. Theorem 2 now gives us a defective probability mass function on $\mathcal{C}$ as: $p(s) := |\mathcal{A}|^{-\ell_{\mathcal{A}}(s)} = 3^{-(\lfloor \log_2(D(s)+1) \rfloor + 1)} \approx \frac{1}{3}(D(s)+1)^{-\log_2 3}$, where we have loosely used $D(s)$ as if it were the value inside the singleton it actually is.

Note that in this particular case, p gives rise to a probability measure Q on $\mathcal{F}$, different from P. However, it is no coincidence that when we had started with the probability space $(\Omega, \mathcal{F}, Q)$, we would have obtained the same probability mass function p and hence the same probability measure Q.

With much subtlety Theorem 2 is used in [8] to redefine codes and length functions so that the converse of the theorem also holds. Codes become abstract, at most countable sets on which length functions $\ell_{\mathcal{A}} : \mathcal{C} \to \mathbb{R}$ are defined that make *identification* of codes and probability mass functions possible in a way that generalizes Theorem 2. As no alphabet is used in the new definition of a code, its role in the length function is merely one of cardinality. The $\mathcal{A}$ in $\ell_{\mathcal{A}}$ functions as a scaling parameter as can easily be seen when we formulate the length of an element of the code in terms of its probability p:

$$\ell_{\mathcal{A}}(s) = -\log_{|\mathcal{A}|} p(s). \tag{1}$$

When this 'scale' is unimportant, we will not write out the dependency on $\mathcal{A}$.

If a decoding function $D : \mathcal{C} \twoheadrightarrow \mathcal{E}$ is injective — if it is not, there is a code $\mathcal{C}' \subset \mathcal{C}$ so that the restriction of D to $\mathcal{C}'$ is — it is a one-to-one correspondence and both the length function and the probability mass function can be considered to be defined on $\mathcal{E}$ as well. We can therefore speak of the code length[*] and probability of elements of $\mathcal{E}$.

4.2 Uncountable spaces

Now, let $(\Omega, \mathcal{F}, P)$ be a probability space where Ω is uncountable. We will start by decomposing this space into a part that can be treated as if it were at most countable and a fundamentally uncountable part.

[*] Note the similarity of this code length function to the complexity functions of Section 3.

In our decomposition, we will use the notion of an *atom* from [5], for which we include the definition here for completeness. All other measure theoretic terminology used in this section is considered elementary. We refer to [1] and [5] for definitions.

Definition 3. Let $(\Omega, \mathcal{F})$ be a measurable space and μ a measure on that space. A set $A \in \mathcal{F}$ is an *atom* for μ if $\mu(A) > 0$ and for every measurable subset B of A it holds that $\mu(B) < \mu(A) \implies \mu(B) = 0$.

We would like the fundamentally uncountable part in our decomposition to be *atomless*, that is, free of atoms for P.

Lemma 3. *Let $(\Omega, \mathcal{F}, P)$ be a probability space. Every pairwise disjoint subset $\mathcal{E} \subseteq \mathcal{F}$ contains at most countably many atoms (for P).*

Proof. Because we are working in a probability space, we have: $P(\Omega) = 1$, thus there are at most $k \in \mathbb{Z}_{>0}$ atoms in $\mathcal{E}$ of measure at least $\frac{1}{k}$. In particular, this means the set $X_k := \{A \in \mathcal{E} : A \text{ is an atom of measure at least } \frac{1}{k}\}$ is finite for every k. Since every atom in $\mathcal{E}$ is contained in X_k for some k, we can enumerate all atoms in $\mathcal{E}$ through the countable union $\bigcup_{k \in \mathbb{Z}_{>0}} X_k$ of finite sets. □

Due to Lemma 3, we can make Ω atomless (for P) by the exclusion of at most countably many atoms. For this, it is important to note that it is always possible to make the subset $\mathcal{E}$ so that if $A \in \mathcal{E}$ contains an atom, A itself is an atom. The set of the excluded atoms, with its power set as σ-algebra and the induced probability measure forms a probability space that can be dealt with according to Section 4.1.

After the above exclusion, we are left with a, possibly defective, probability space $(\Omega, \mathcal{F}, P)$ (we reuse the same symbols), where P is atomless. The case of an at most countable Ω is already dealt with in Section 4.1, so we may assume Ω is uncountable. In what follows, we will generalize the code length function of Section 4.1 for such spaces. In order for this to be meaningful, it turns out we will need a σ-finite, strictly positive,* atomless reference measure μ on $(\Omega, \mathcal{F})$, so that P is continuous with respect to μ on $\mathcal{F}$. Although the three constraints on μ give a sense of limitation, they turn out to be met easily in practice. The requirement of σ-finiteness is very common in measure theory and present in many of the theorems in both [1] and [5]. Requiring that μ is strictly positive is a stronger demand as it demands a topological base for μ. Because of our previous analysis, atomless reference measures are a natural choice. In Example 3, the Lebesgue measure, which meets all three criteria, is used.

An easy way to get a grip on Ω is by discretizing. It makes it possible to mirror what we did in Section 4.1.

Definition 4. Given a measurable space $(\Omega, \mathcal{F})$ and a σ-finite measure μ on that space, a *discretization for μ of* $(\Omega, \mathcal{F})$ is a subset $\mathcal{E} \subseteq \mathcal{F}$ that forms a partition of Ω, such that:

$$\forall A \in \mathcal{E} : 0 < \mu(A) < \infty$$

holds. If in addition: $\forall A, B \in \mathcal{E} : \mu(A) = \mu(B)$ holds, the discretization is said to be *uniform.*

* Strict positivity is defined for Radon measures in [1, 5] based on the support of the measure. In short a measure μ on $(\Omega, \mathcal{F})$ is *strictly positive* if there is a Hausdorff topological space (Ω, T) such that: $T \subseteq \mathcal{F}$ and $\forall U \in T : U \neq \emptyset \implies \mu(U) > 0$.

An immediate consequence of the definition is the following.

Corollary 4. *Discretizations are at most countable.*

Proof. This is a consequence of the fact that μ is σ-finite and the discretization is a pairwise disjoint collection of non μ-negligible sets. A complete proof is available as the proof of Proposition 215B in [5]. $\square$

In order to prove the existence of uniform discretizations, we need the following proposition.

Proposition 5. *Let $(\Omega, \mathcal{F}, \mu)$ be an atomless measure space. If $A \in \mathcal{F}$ and $0 \leq \alpha \leq \mu(A) < \infty$, there is a $B \in \mathcal{F}$ such that $B \subseteq A$ and $\mu(B) = \alpha$.*

This proposition is proven as Proposition 215D in [5]. In the proof, the construction of B uses the axiom of choice. The proposition enables a proof of the following lemma.

Lemma 6. *For every σ-finite, atomless reference measure μ on a measurable space $(\Omega, \mathcal{F})$, there exists a uniform discretization for μ of $(\Omega, \mathcal{F})$.*

Proof. In case μ is finite, none of the other properties of μ matter and $\{\Omega\}$ is as required. For infinite μ, we use the fact that μ is σ-finite and write Ω as a countable union $\bigcup_{k\in\mathbb{Z}_{>0}} X_k$ of sets $X_k \in \mathcal{F}$ of finite measure. Proposition 5 allows us to split up X_1 into $\lfloor \mu(X_1) \rfloor$ measurable sets of measure 1 and a measurable remainder R_1. Inductively, we give the same treatment to $X_k \cup R_{k-1}$, for $k > 1$: split it up into $\lfloor \mu(X_k \cup R_{k-1}) \rfloor$ measurable sets of measure 1 and a measurable remainder R_k. This way all of Ω is covered by pairwise disjoint measurable sets of identical μ-measure. $\square$

To get a hold of the coarseness of discretizations, we introduce a manner of convergence.

Definition 5. Let $(\Omega, \mathcal{F})$ be measurable space containing a Hausdorff topological space (Ω, T) and let μ be a σ-finite, strictly positive measure on $(\Omega, \mathcal{F})$. Consider a sequence of discretizations, $(\mathcal{E}_k)_{k\in\mathbb{Z}_{>0}}$, for μ of $(\Omega, \mathcal{F})$. For all $x \in \Omega$, we fix a notation for the class x resides in under a given discretization by: $\forall k \in \mathbb{Z}_{>0} : x \in [x]_k \in \mathcal{E}_k$. The sequence of discretizations is *convergent* if and only if for all $U \in T$ it holds that:

$$\forall x \in U : \exists k_{U,x} \in \mathbb{Z}_{>0} : \forall k \geq k_{U,x} : [x]_k \subseteq U.$$

This is a way of ensuring the discretization becomes increasingly fine as its index grows. If a measure μ on a measurable space $(\Omega, \mathcal{F})$ is σ-finite, strictly positive and atomless, it follows from Lemma 6 and Proposition 5 that there exists a convergent sequence of uniform discretizations for μ of $(\Omega, \mathcal{F})$.

Example 3. Arguably the most common uncountable set is $\mathbb{R}$. All common atomless probability measures are continuous measures with respect to the Lebesgue measure on $\mathbb{R}$. Also the Lebesgue measure is σ-finite, strictly positive and atomless. Hence there must be a convergent sequence of uniform discretizations for the Lebesgue measure of $\mathbb{R}$ (with the Lebesgue measurable sets as σ-algebra). One example of such a sequence is given by: $\mathcal{E}_k = \{[n2^{-k}, (n+1)2^{-k}) : n \in \mathbb{Z}\}$.

When combined with discretizations, the Radon–Nikodym theorem is of great use to us.

Theorem 7 (Radon–Nikodym)**.** *Let μ be a σ-finite reference measure on a measurable space $(\Omega, \mathcal{F})$. For every, possibly defective, probability measure P that is absolutely continuous with respect to μ, there is a μ-measurable function $p : \Omega \to \mathbb{R}$ that satisfies, for all $A \in \mathcal{F}$:*

$$P(A) = \int_A p \,\mathrm{d}\mu.$$

A proof of Theorem 7 can be found in [1]. We cannot directly use the *probability density function* p that Theorem 7 gives us to define code lengths like we did in equation 1. Using discretizations, it is possible to define a sequence of functions that converges to p that do allow us to define code lengths.

Theorem 8. *Let μ be a σ-finite, strictly positive measure on a measurable space $(\Omega, \mathcal{F})$ that contains a Hausdorff topological space (Ω, T) and let a convergent series of discretizations for μ of $(\Omega, \mathcal{F})$ be given. For all μ-integrable functions f, the sequence, indexed by $k \in \mathbb{Z}_{>0}$, of functions of the form*

$$\frac{1}{\mu([x]_k)} \int_{[x]_k} f \,\mathrm{d}\mu$$

converges pointwise to a function that is μ-almost everywhere identical to f.

Proof. The procedure is much like the inverse of Riemann integration. Let f be represented as the upper envelope $f = \sup_{j \in \mathbb{Z}_{>0}} u_j$ of an isotone sequence $(u_j)_{j \in \mathbb{Z}_{>0}}$ of elementary functions (a common approach in [1]) for the Borel σ-algebra $\mathcal{B}(T)$ on (Ω, T). Note that the restriction to Borel sets is possible because every elementary functions for $\mathcal{F}$ is μ-almost everywhere identical to an elementary function for the Borel σ-algebra. For every $j \in \mathbb{Z}_{>0}$ we can look at a normal representation of u_j. In such a representation, every $x \in \Omega$ is contained in some set $A_{j,x} \in \mathcal{B}(T)$ of identical u_j-value $\alpha_{j,x}$. Within every $A_{j,x}$ of positive measure, we can find an open set $U_{j,x}$ of T containing x. Now, per definition, there is an index $k_{U_{j,x}} \in \mathbb{Z}_{>0}$ so that for all $k' > k_{U_{j,x}}$ it holds that $[x]_{k'} \subseteq U_{j,x} \subseteq A_{j,x}$ and thus that

$$\frac{1}{\mu([x]_{k'})} \int_{[x]_{k'}} u_j \,\mathrm{d}\mu = \frac{\alpha_{j,x}}{\mu([x]_{k'})} \int_{[x]_{k'}} \mathrm{d}\mu = \alpha_{j,x} = u_j(x).$$

Taking the supremum with regard to j both left and right yields equality for k', where k' is at least $\sup_{j \in \mathbb{Z}_{>0}} k_{U_{j,x}}$ for any given x. Since, for every x, the sequence $(k_{U_{j,x}})_{j \in \mathbb{Z}_{>0}}$ consists of merely positive integers, pointwise convergence holds. □

Using Theorem 8, we can write the equation of Theorem 7 as an approximation for $x \in \Omega$ that gets better as k tends to infinity: $P([x]_k) \approx p(x)\mu([x]_k)$. As μ is strictly positive, the σ-algebra on Ω contains a Hausdorff topology, which means that every two distinct points $x, y \in \Omega$ are distinguishable by sets of positive measure. Therefore, we have that $\lim_{k\to\infty}[x]_k$ is $\{x\}$. This makes it meaningful to define a length-like function on Ω by

$$\ell^k(x) := -\log p(x) - \log \mu([x]_k),$$

where p relates to some, possibly defective, probability measure through Theorem 2. If we additionally require μ to be atomless and restrict ourself to convergent series of *uniform* discretizations, $\mu([x]_k)$ does not depend on x and we can write:

$$\ell^k(x) = -\log p(x) + c_k, \tag{2}$$

with the *discretization constant* c_k going to infinity as k goes to infinity because μ is atomless.

Of course ℓ^k would only be a proper length function, i.e., one that satisfies Theorem 2, in the limit as k approaches infinity, and would then assign infinite length to all elements of X. Nevertheless a crucial observation is that the length function does not alter its behaviour with varying k other than that it is 'shifted upwards or downwards'.

All in all we have found a way of dealing with elements of uncountable sets based on the minimum description length principle. We have also identified the assumptions necessary for this way to be attainable. A suitable length function for a code corresponds to a probability distribution that captures some or all of the regularities in the data. The length function takes on the form $-\log p + c_k$, where p is determined by Theorem 7 and c_k is only important in model (class) complexity comparison and not for measuring compression performance. Properly accounting for the detail in, and impact of p by the choice of the discretization constant c_k can be an involved process.

Part II
Practice

5 A simple model

We devise a general model for use in coding. This model is only slightly more complex than assuming a uniform distribution over the range of some data. Formulas for both the uniform model and the slightly more complex one are derived, so that comparison of the two becomes possible for given databases and model parameters.

The success of minimum description length based reasoning for data mining use is shown by [15]. As that paper deals with pattern mining in terms of frequent itemsets, the method proposed, Krimp, is not very suitable for non-nominal data. Where such data is encountered, it is preprocessed by some discretization. Also auxiliary structure present in the data, such as an ordering, is not used by Krimp.

To Krimp, a pattern is a subset of some fixed finite itemset $\mathcal{I}$. We associate with a pattern a unit hypercube embedded* in $[0,1]^{|\mathcal{I}|}$ that contains $(1,1,\ldots,1)$, specifically: the Cartesian product of copies of the closed interval $[1]$ for items in the pattern and of the closed interval $[0,1]$ for items not in the pattern. In this view, we have a $[0,1]^{|\mathcal{I}|}$-valued random variable (in the sense of Definition 2) that maps data to the 'corners' of $[0,1]^{|\mathcal{I}|}$ and Krimp is a statistic that maps a sample ('multiset of corner points') to a set of patterns. It does so looking for good compression based on a decoding function that is constructed with meticulous effort in [15].

The embedding of data in the hypercube $[0,1]^{|\mathcal{I}|}$ makes for an obvious extension to measurable, ordered data. What follows is, however, not a generalisation of Krimp, but rather a Krimp-inspired model for measurable, ordered data that is designed for data mining. By no means we state that this model — or Krimp, for that matter — has the sought after small constant linked with Theorem 1. It is even impossible that the model defines a complexity function that is additively optimal. The success of Krimp nevertheless makes a case for the usefulness of this sort of modeling and therewith for the presence of 'nice enough properties' of the complexity functions resulting therefrom.

We broaden our scope to accommodate, given $n \in \mathbb{Z}_{>0}$, for $\mathbb{R}^n$-valued random variables†, and focus on embedded bounded hyperrectangles with edges parallel to coordinate axes. In contrast to Krimp, we do not demand any point to be covered by the hyperrectangle. If we think of each of the n dimensions as attributes of the data, this means we are interested in endpoints of intervals of attributes. Therefore, we will refer to such hyperrectangles as *hyperintervals*. As a foundation of our code length function we will take a very crude form of exceptional model mining as introduced in [12] and, in a way, treated in [11]. We want to attain better compression as the difference between the average density of a sample inside the hyperinterval and the average density of that sample outside the hyperinterval increases. Considering that the outside of a

* By 'embedded', we mean the dimension may be less than that of the surrounding space.
† In fact, what follows holds for random variables in any finite dimensional metric space.

hyperinterval is unbounded, we have to alter our approach somewhat to be able to compare both these average densities.

Since we hold data mining to be about statistics*, being observable random variables, we are only concerned with the observable implications of our model. Therefore, we start out with a sample of size m of a random variable that maps into $\mathbb{R}^n$. Now let M be the volume of the smallest hyperinterval covering the entire sample, according to the Lebesgue measure on $\mathbb{R}^n$. Without any further hyperintervals, we assign† to a data point a length of $-\log \frac{1}{M}$. This choice is the most obvious one when we do not take any prior information about the data to be known, as it is based on a uniform density of data across the range of the database. This would make the *complexity* of the sample equal to

$$m\left(-\log\frac{1}{M}\right) = m\log M. \tag{3}$$

In [11] it is shown that the assumption of uniformity in a reference model might give rise to trivial mining results. However, for simplicity we stick with our choice.

Suppose we have an additional hyperinterval, H, inside this interval of volume M and let m_{in} denote the number of sample points it covers and M_{in} its volume. Additionally take $m_{\text{out}} := m - m_{\text{in}}$ and $M_{\text{out}} := M - M_{\text{in}}$. Each point x is now naturally equipped with a length $\ell(x)$ as:

$$\ell(x) := \begin{cases} -\log\frac{1}{M_{\text{in}}} = \log M_{\text{in}} & \text{for } x \in H, \\ -\log\frac{1}{M_{\text{out}}} = \log M_{\text{out}} & \text{for } x \notin H. \end{cases}$$

These equations again are based on uniform densities, this time separately for the inside of the hyperinterval and for the outside of the hyperinterval. In order to be able to differentiate between 'inside points' and 'outside points' we need to add to this length. For the sample, the optimal additional length would be $-\log\frac{m_{\text{in}}}{m} = \log m - \log m_{\text{in}}$ for points inside and $-\log\frac{m_{\text{out}}}{m} = \log m - \log m_{\text{out}}$ for points outside, all of these lengths coming forth of equation 2. The complexity of the sample in this new case is thus equal to

$$\begin{aligned} m_{\text{in}}(\log M_{\text{in}} + \log m - \log m_{\text{in}}) + m_{\text{out}}(\log M_{\text{out}} + \log m - \log m_{\text{out}}) \\ = m_{\text{in}}\log\frac{M_{\text{in}}}{m_{\text{in}}} + m_{\text{out}}\log\frac{M_{\text{out}}}{m_{\text{out}}} + m\log m. \end{aligned} \tag{4}$$

To be fair, we should compensate for the fact that this added hyperinterval brings greater detail in the model to the table. A very reasonable way of assigning cost to this model, would be to encode the added hyperinterval in some way. An easy way to do this is by specifying two points within the surrounding hyperinterval: one containing the lower endpoints for each attribute of the hyperinterval we want to encode, the other containing the upper endpoints. Taking a single attribute $x \in \mathbb{R}$ with minimum value L and maximum value U in the sample, we may define a probability density function on the upper endpoint of the hyperinterval of interest for the attribute as $\frac{x-L}{(U-L)^2/2}$. Then, provided an upper endpoint u, we take a uniform probability density function on the lower

* The plural of statistic is meant, not statistics as a branch of mathematics.

† For now, the discretization constant is ignored. It will be dealt with a bit further on.

endpoint of the hyperinterval of interest for the attribute, which has constant probability density $\frac{1}{u-L}$. Any combination of lower and upper endpoint now has probability $\frac{x-L}{(U-L)^2/2} \cdot \frac{1}{x-L} = \frac{2}{(U-L)^2}$, which does not depend on the actual values of the endpoints! When we follow this procedure for all attributes, the likelihood of every hyperinterval becomes $\frac{2^n}{M^2}$, which corresponds to a length of

$$-\log \frac{2^n}{M^2} = 2\log M - n\log 2. \tag{5}$$

Of course equations 3, 4 and 5 were found in a way that demands an investigation of discretization constants. Whatever discretization constant we choose, it will be added m times to both equation 3 and equation 4. As a consequence, it does not influence comparison of the two. Equation 5 is an entirely different case. It is not hard to see that a discretization constant should be added twice to this quantity — once for the point consisting of the lower endpoints and once for the point consisting of the upper endpoints. As m increases, we would like to be more demanding towards the detail in the model, so ideally equation 5 should increase with m when the discretization constant is taken into account. When the discretization is too fine, the sample becomes sparse with regard to the discretization and the cost of the added hyperinterval can hardly be made up by compression. A very elegant choice for the discretization constant is $-\log \frac{M}{m^n}$, transforming equation 5 into

$$2\log M - n\log 2 - 2\log \frac{M}{m^n} = n\left(2\log m - \log 2\right) = n\log \frac{m^2}{2}. \tag{6}$$

This can be interpreted as the cost of specifying two elements from the sample for each attribute to choose that attribute of the two elements as lower, respectively upper endpoint of the attribute in the hyperinterval; an interpretation that certainly is more than acceptable.

On a final note, we consider the model class of arbitrary many hyperintervals and their (dis)ability to compress a sample. Elements of such a model can be approximated very well by iterative modeling with only one hyperinterval of interest. This makes it unnecessary to explore such a model class in great detail here.

6 A simple model class

The theory of data mining practice that the previous section developed is extended with a model class. Formulas are derived to compare the model that was introduced in the previous section and other models in the model class to each other for a given database.

A model class that is interesting for data mining practice is one that holds information on relevance of dimensions. Oftentimes data is gathered without knowing what attributes play a part in what patterns. A contrived example of such a situation is presented in Section 10.

Instead of inducing a model class that forces us to investigate our model for all 2^n possible dimension selections of the sample, we will define a simple model class based on a variation of our model. For this purpose, we note that the dimensionality of the codomain of the random variable mainly comes into play through equation 5. Arguably it plays a role in the volume M as well, but as we will discuss in a bit more detail in Section 8, the relation between the volume and the number of dimensions is such a delicate one, that it is not easy to develop a simple, general theory about it for data mining purposes. Thus we turn to equations 5 and 6 for finding a measure of the informativeness of each dimension. We derived a length for the description of the detail in the model, with discretization constant c, of $2 \log M - n \log 2 + 2c$, which straight away suggests a per-dimension length of

$$\frac{2}{n} \log M - \log 2 + \frac{2}{n} c.$$

For the sake of notational clarity, we will let Δ_c denote this quantity. In case of $c = -\log \frac{M}{m^n}$, as in equation 6, we have:

$$\Delta_c = \log \frac{m^2}{2}.$$

The interpretation of Δ_c is that it represents the information held in the specification of a single dimension of the hyperinterval of the model.

Now for a selection of $\bar{n} \leq n$ dimensions, we amended the model by altering the complexities expressed by equations 4 and 5. The complexity of the model (equation 5) is redefined to

$$\bar{n}\Delta_c = \frac{\bar{n}}{n} 2 \log M - \bar{n} \log 2 + \frac{\bar{n}}{n} 2c. \tag{7}$$

The form of equation 4 remains intact, our adjustments will lie in the definitions of m_{in} and M_{in}. On the $n - \bar{n}$ non-selected dimensions, we think of hyperinterval as spanning the full range of the random variable on that dimension. This means m_{in} will be the number of points that are covered on the selected dimensions (coverage on the non-selected dimensions is implied) and M_{in} will be calculated from a hyperinterval that is 'full' (for the respective ranges of the random variable) in the non-selected dimensions. Note that the original equations 4 and 5 are special cases of these newly defined complexities for $\bar{n} = n$.

The remaining step in the construction of our model class is the codification of the selected dimensions, so that resulting complexities with regard to particular models can be compared within the class. This is really where we choose to keep the model class simple. By taking a uniform prior over the 2^n models in the class, we obtain a constant complexity for each model, irrespective of the model. In doing so, we make comparison of models in the class solely dependent on the lengths defined in those models. Put bluntly a dimension is thus relevant when the reduction in description length obtained by specifying boundaries for the hyperinterval in that dimension is bigger than Δ_c.

7 Feasibility

In practice, one would not manually try to find best compressing hyperintervals, that is, hyperintervals that can not be extended or shrunk into hyperintervals that compress better. Instead, one would use a computer. We explore the efficiency and effectiveness of a method to do so.

7.1 Setting a goal

In real-world scenarios, jargon differs somewhat from what we used until now. From here on, we will follow a more data-centric approach and call the realization of a statistical sample a *database*. However common a name that might be in data mining, it is a bit of a misnomer since the corresponding concept in practice is usually a 'table'. For the variables of the previous section we now get alternative views. The number of attributes (in our case each of which is real-valued) of the database is indicated by n, the number of rows by m. The volume of the smallest hyperinterval wherein the entire database resides is indicated by M. Again, for a hyperinterval inside that of volume M, we designate by M_{in} its volume and by m_{in} the number of rows of the database that fall inside this hyperinterval. The numbers M_{out} and m_{out} are once more defined as $M - M_{\text{in}}$ and $m - m_{\text{in}}$, respectively.

We are now interested in hyperintervals for which equation 3, the complexity of the database, is bigger than the sum of equations 4 and 5 plus two times a discretization constant c:

$$\Big[m_{\text{in}} \log \frac{M_{\text{in}}}{m_{\text{in}}} + m_{\text{out}} \log \frac{M_{\text{out}}}{m_{\text{out}}} + m \log m\Big] + \Big[2 \log M - n \log 2\Big] + \Big[2c\Big]. \tag{8}$$

When this is the case, the hyperinterval parameters provide enough 'information' to justify the cost of their specification and we have learned a property of the database.

Now let us set c to the intuition-stimulating $-\log \frac{M}{m^n}$ of the previous section, subtract $m \log m$ from both equation 3 and equation 8, and rearrange the terms somewhat. We find that we are equally interested in hyperintervals for which

$$m \log \frac{M}{m} - n \log \frac{m^2}{2} \tag{9}$$

is bigger than

$$m_{\text{in}} \log \frac{M_{\text{in}}}{m_{\text{in}}} + m_{\text{out}} \log \frac{M_{\text{out}}}{m_{\text{out}}}. \tag{10}$$

Here, we have gathered everything that does not depend on the choice of the hyperinterval in equation 9 and everything that does in equation 10. The result is a fairly elegant pair of formulas. If we want to broaden our scope to the slightly more general model definition of Section 6, equation 9 becomes:

$$m \log \frac{M}{m} - \bar{n} \log \frac{m^2}{2}, \tag{9$'$}$$

where $\bar{n}$ is the number of chosen dimensions.

It turns out to be surprisingly easy to find interesting hyperintervals under this criterion. This is due to the fact that very many sub- and supersets of interesting hyperintervals are interesting too. In data mining, this is a common phenomenon. It is identified in [15] as the *explosion* of the number of found patterns, which is deemed a major defeat in the usefulness of pattern mining. The way out for Krimp is to define a compression scheme based on a pattern dictionary and focus on the dictionary that best compresses a database. Our model is simpler than Krimp, but we too can confine our attention to those interesting hyperintervals (patterns) that maximize (locally, at least) the difference between equations 9′ and 10. By locally, we mean that no hyperinterval that is either an extension or a restriction of the hyperinterval under inspection yields a better compression. Thus our search is one for locally best compressing hyperintervals. It is good to note that the method for finding such hyperintervals outlined below is not of a pruning/post-processing nature. Instead, it directly searches for such best compressing hyperintervals, obliterating the need to keep track of an enormous amount of hyperintervals.

A further difficulty pointed out in [15] concerns the computational time complexity of mining algorithms as a function of the size, m, of the database. Especially for non-nominal data, it is hard to come up with an algorithm that scales well. In case our database consists of points in $\mathbb{R}^n$ generated by a process with an atomless underlying measure, with probability one, no two rows in the database are the same. We are tempted to believe this means every row is relevant to whatever method we devise. Under some mild conditions, we can, however, get control over the run-time complexity of the mining procedure.

7.2 A method outline

We propose the outline below for a locally best compressing interesting hyperinterval mining algorithm. In this algorithm, we use complexity calculations based on the simple model of Section 5. The more general model class from Section 6 is heuristically covered in a post-processing phase.

1. Sample the database.
2. Calculate all distances between rows in the sample[*].
3. Pick two neighbouring (in distance) rows in the sample.
4. Extend a hyperinterval covering these rows based on other rows in the sample in a compression (calculated on the entire database) increasing direction.
5. Report the hyperinterval if it is interesting.

We can post-process the hyperinterval after step 4 in the following way. Complexity calculations are now based on the model class of Section 6 and initially take $\bar{n} = n$.

4.1. Calculate the coverage of each dimension of the hyperinterval.

4.2. In order of decreasing coverage, determine if compression improves when leaving out a dimension and leave it out if it does.

5. Report the pruned hyperinterval if it is interesting.

As we will see in Section 10, this heuristic finds reasonable models in the model class. We will sequentially go over the steps in more detail.

The first step means randomly selecting a fixed number s, say, of rows from the database. This step is where control is gained over the run-time of the rest of the algorithm. In a way, this step makes the algorithm a Monte Carlo method, for a random sample of the database is identically distributed to a random sample of outcomes in the outcome space of the random variable underlying

[*] Take care not to confuse this sample with the database.

the database. Depending on the smoothness of the distribution of the random variable underlying the database, we can expect the sample to represent all the details of the distribution — i.e., be a faithful representation of the distribution — for sample sizes bigger than a certain number.

We expect the two sampled rows with minimum distance between them to be in the region of the database where the density function of the random variable assumes its maximum. Because this is a good place to look for deviations (positive ones) from the average density of the database, the second step consists of calculating the distance between every two distinct rows in the sample. A list of all pairs is generated ($\frac{1}{2}s(s-1) \in \mathcal{O}(s^2)$ pairs in total) and sorted by the distance between their two members.

In step three it is determined where a hyperinterval will be 'grown'. Based on earlier runs of the algorithm we may want to exclude certain starting points. This is for instance the case when we want to start in a region not covered by any hyperinterval we have found earlier. From the acceptable pairs of the sample we pick the first one, that is, the one which is the pair of smallest relative distance. This step is the entry point for consecutive runs of the algorithm with the same sample, skipping the expensive previous step.

Step four can be considered the heart of the algorithm. One way to think about it is as a way to (locally) execute a maximum likelihood estimate of the parameter, the hyperinterval, of our model. Another is as a minimization, again local, of the complexity as a function on the space of all possible hyperintervals through a greedy hill climber approach. To this end, given a hyperinterval H, consider the row x in the sample that is closest to it.* For the smallest extension H' of H such that H' covers x, we calculate the complexity of the entire database given H'. If this complexity is smaller than the complexity given H, in other words: if better compression is obtained, we adopt H' as our new hyperinterval of interest. If it does not yield better compression, we would at first not expect that the same actions for a different x would, because the sampling scheme makes that we expect the chosen x to be outside H in the direction of highest data density. However, with positive probability the highest data density is not in the direction of x, thus we should try a few, say, at most p, different rows of the sample in order of closeness to H before we conclude that we cannot improve on the compression performance of H. This same approach also makes that we go over small complexity bumps in the hyperinterval space. Indeed, if p is very large (with regard to s) we can hope to find an approximation of a globally best compressing hyperinterval. The effect of such perseverance is illustrated by Figure 1. It is important that complexity assessments happen on the entire database, because that is the best source we have for the presumed 'true' distribution of the random variable from which the database was generated. The sample is inherently too coarse for such calculations: the resolution of the changes to H makes that the sample is not a good indication of the changes in covered probability density. As long as we shift our focus from H to a bigger H', we repeat this step, initially setting $H := H'$.

The fifth and final step in the simplest form of the proposed outline concerns the comparison of the complexities expressed by equations 9 and 10, where the values of the variables in the latter are determined by the hyperinterval H

* 'Closest to it' is of course ill-defined, but for most cases it turns out the exact definition is not crucial or comes naturally with the database. Section 9 puts forth a rather general choice of a definition.

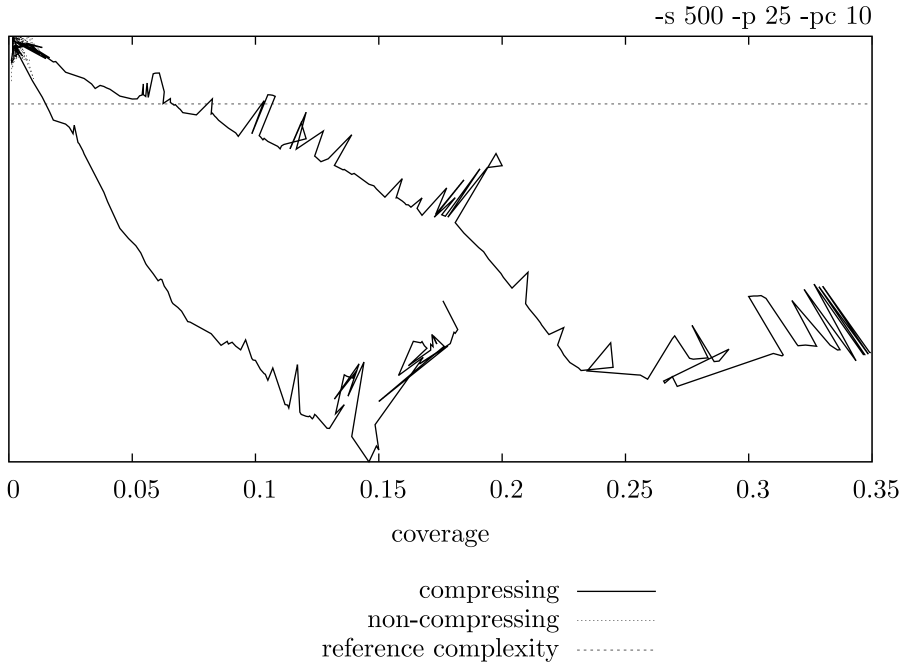

Figure 1: Candidate complexity related to database coverage for the exemplary database explored in Section 10. Depicted are six runs of the procedure, of which two yielded a compressing hyperinterval (the other four runs are short and stuck in the upper-left corner of the plot). Changes in the discretization constant translate the complexity scale (cf. equation 2) and changes in the base of the involved logarithms act as a scaling transformation. Therefore, the complexity scale is not labeled. The horizontal line represents the outcome of equation 9: complexity minima below this line correspond to best compressing hyperintervals. The method proceeds in coverage increasing direction. The bumps in the complexity lines correspond to local minima of the complexity in the hyperinterval space or unfortunate extensions of a hyperinterval under investigation.

resulting from the previous step. If H actually gives rise to compression, we have learnt something about our database (and couldn't easily 'learn more') and can proudly report the hyperinterval. If we were not able to compress, we can be in a number of situations. The easiest reason we can imagine is that the database does not harbor an 'exceptional model' (in our simple one-or-the-other setting) and the data is best described as-is. We could also have sampled too sparingly and need to increase the sample size. Incidentally, increasing p in step four could help too in this case, as the grandest features of the database are likely to be expressed rightly by the sample early on, thus more global features can potentially still be found with a sample that is too small given the peculiarities of the distribution underlying the database. Lastly, we could just have been out of luck in step three and need to go back to that step, picking a new starting point outside the non-compressing H.[*] Just like in step four, after restarting a few times, each time finding a non-compressing hyperinterval, we can conclude there really is nothing to be found.

Estimating the model in the simple model class of Section 6 can be done by a minor alteration of the above procedure. As mentioned in the last paragraph of said section, we will not mind the cost of specifying a model in the model class, but focus entirely on the complexity of the database for specific models and estimated parameters (hyperintervals) for those models.

At step 4.1, we determine per attribute (column) of the database how much of it is covered by the hyperinterval. This can be done based on the database, but under the hypothesis that the database is uniformly distributed over a certain range (this is the model we find exceptions on) this equals the fraction of the range that is covered by the hyperinterval. Using this coverage information is a heuristic that makes the next step of the algorithm greedy. Doing so prevents having to examine all possible 2^n models in the class. Instead, at most n are considered, making the number of complexity calculations done in the post-processing an element of the $\mathcal{O}(n)$ complexity class. As mentioned earlier, the run-time complexity of the complexity calculations is linear or worse in the size, m, of the database[†].

If a dimension of the database holds no information, we expect the mined hyperinterval to cover it for as much as the sample allows. Thus it is sane to check highly covered dimensions first in step 4.2. We take the result, say, C, of evaluating expression 10 for the hyperinterval resulting from step 4 and repeat the following. In decreasing order of coverage, we take an attribute of the database and alter H so that its altered variant H' covers all values of the attribute, that is, coveres the entire corresponding dimension of the range of the random variable. Next we check whether the result, C', of evaluating expression 10 for H' improves on $C + \Delta_c$, where Δ_c is as discussed in Section 6 and $c = -\log \frac{M}{m^n}$, and if it does we set $H := H'$ and $C := C'$. If C' is in fact bigger than $C+\Delta_c$, we can induce (the philosophical form, not the mathematical form) that no dimension can be removed from H to obtain a better compressing hyperinterval. However, as before this could be due to chance, so once more we go on trying to prune other dimensions and only after being unsuccessful a fixed

[*] This is a bit tricky as there might be interesting hyperintervals completely inside the discarded H, but the risk of restarting inside H is that we end up with more or less the same non-compressing hyperinterval.

[†] For huge databases, one could take another sample, some orders of magnitude larger than the sample generated in step 1, to use for complexity calculations.

few times in a row, we conclude that the not left out $\bar{n}$ dimensions are relevant dimensions of the hyperinterval.

In case we started the previous step with a compressing hyperinterval, the result of that step is guaranteed to be an interesting (pruned) hyperinterval. It is, however, possible that a non-interesting hyperinterval gives rise to an interesting pruned hyperinterval. The final step of the algorithm with post-processing only differs slightly from the final step of the algorithm without post-processing. The difference is that we no longer compare the complexities expressed by equations 9 and 10, but those of equation $9'$, with $\bar{n}$ resulting from step 4.2, and equation 10, where the hyperinterval is the one resulting from step 4.2. Equivalently, we compare the complexity expressed by equation 9 with $C + (n - \bar{n})\Delta_c$, which is known from the previous step.

7.3 Complexity analysis

It is worth noting that the method just described is a consistent estimator for the hyperintervals it finds in that as the sample size (and thus the database size too) tends to infinity, the reported hyperintervals are indeed locally best compressing. We would like the method to be efficient too, so we take a closer look at the computational time complexity of the various steps.

The first two steps of our method are only executed once for every mining session. As more mining results are produced by a single mining session, the complexity of these steps thus becomes less important. During the experimentation treated in Part III, the time needed to load the database into memory often was not insignificant with regard to the time needed to execute the first two steps of the procedure, which, in a way, is a good thing.

When done cleverly, the sampling of the database in step 1 has a complexity about linear in the requested sample size. So, if s rows from the database are needed, the step could suffice with generating s random row-numbers within the range of the database, for which the Knuth shuffle is an algorithm in the $\mathcal{O}(s)$ class. This operation involves only integers and is really trivial on almost any computer.

For step 2, the order of the number of distance calculations is $\mathcal{O}(s^2 \log s^2) = \mathcal{O}(s^2 \log s)$, since the number of pairs is of the $\mathcal{O}(s^2)$ order. The complexity of this step is important in the overall the run-time complexity of the algorithm. It is important to note that distance calculations are unlikely to be atomic. In general, the complexity of a distance calculation is a function of the number, n, of attributes of the database. In the most common case, where the complexity of a distance calculation is linear in n, the run-time complexity of this step is in the $\mathcal{O}(ns^2 \log s)$ class.

The first step that is potentially visited multiple times in a mining session is step 3. Because of the sorting in the previous step, this step initially has $\mathcal{O}(1)$ complexity: the two rows in the sample that are closest together make up the first pair in the sorted list created by step 2. As the algorithm proceeds, pairs may have to be excluded, which is done by coverage checks. If the mining session exhausts the sample, all pairs are excluded, which cumulatively would have taken $\mathcal{O}(ns^2)$ operations. It is, however, uncommon that a session exhausts the sample when the right parameters are used.

The fourth step is the most important to analyse, but also the hardest. It is of little use to investigate worst case scenarios, because they correspond to highly unlikely databases and samplings. There are a few things we can say about more usual runs. If a hyperinterval is initiated within a compressing hyperinterval, the number of extensions the hyperinterval will undergo is a function of the relative density in the eventually resulting hyperinterval: the higher the relative density, the larger the part of the sample that is involved. So the best mining results are likely to keep us waiting the longest. If a hyperinterval is initiated in completely the wrong region of the database, we would detect so in about $\mathcal{O}(p)$ attempts to extend the hyperinterval, where p is the perseverance parameter mentioned in the previous section. It is important to know the complexity of an extension attempt. Such an attempt involves three parts:

1. Selecting an extension candidate.
2. Generating the extended hyperinterval.
3. Calculating the complexity given the extended hyperinterval.

The first part entails about s distance calculations, so is in the $\mathcal{O}(ns)$ class for common distance functions. The second part is very simple and in the $\mathcal{O}(n)$ class. The third part involves coverage checks for the entire database and a volume calculation for the hyperinterval. Assuming both coverage checks and volume calculations are linear in n, this puts this part in the $\mathcal{O}(nm)$ class. In general, it looks reasonable to put the entire fourth step in the $\mathcal{O}(ns^2 + nms)$ class, although this might be a bit biased to a worst case analysis. In any case it is seemingly outside the $\mathcal{O}(nsp + nmp)$ class for compressing hyperintervals.

The last step is of course $\mathcal{O}(1)$, so the algorithm is supposedly well within the $\mathcal{O}([s]+[ns^2 \log s]+[1]+[ns^2+nsm]) = \mathcal{O}(ns^2 \log s+nsm)$ class for mining a single hyperinterval. Additional hyperintervals can then be mined in $\mathcal{O}([ns^2] + [ns^2 + nsm]) = \mathcal{O}(ns^2 + nsm)$ operations.

Post-processing adds at most n complexity calculations in step 4.2. This means that for common distance functions, the added time complexity is of the order $\mathcal{O}(n^2 m)$. Altogether, the run-time complexity of the algorithm is clearly polynomial in n, s, m, where it is good to know that substantial influence over the run-time can be exercised through the choice of s.

8 Shortcomings

This section is where we stop developing new theory and step back for a moment. A few difficulties that are either specific to our method or to data mining in the context of our method are discussed.

An immediate difficulty related to the very first step of the proposed algorithm lies in the choice of the sample size. Without prior knowledge of the data, it is unclear what the best sample size is. When chosen too small, lots of, potentially interesting, detail in the data is lost. When chosen too large, the algorithm will take an unnecessary amount of time to execute. Unfortunately, there appears to be little we can do about this and it will be part of the data miner's task to find a suitable sample size.

Less immediate, but still of a practical nature is the following. Until now, our descriptions of methods have been rather data agnostic. There is, however, a way to make use of any possible prior information about the data. This is done through the specification of a measure, with which the volumes M, M_{in} and M_{out} are calculated. Indeed, the choice of a measure is a freedom in the application of the methods outlined in this thesis. Even if all attributes of the database are readily encoded in real numbers, it is in general not wise to use the standard Lebesgue measure. Often, the ranges of different attributes differ, causing large-scaled attributes to be of higher influence in calculations. The straight-forward way of dealing with this, is by normalizing the attributes, or, equivalently, scale the measure in each dimension. In practice it is easy to scale the measure so that the distance between the minimum and the maximum value of an attribute equals 1, but this is not robust to outliers or marginal distributions with tails that behave differently for different attributes. One possible solution is to scale the measure so that the standard error of each attribute is normalized, but again, cases can be thought of where this is either cumbersome or not satisfactory. Thus the freedom of choosing a measure can at times, when very little is known about the data, be a drawback.

One further difficulty that is inherent to most measures is that a subspace of a dimension lower than the number of attributes of the database can fail to have a strictly positive volume. In other words: usually M_{in} is of zero volume inside M when $\dim M_{\text{in}} < \dim M$. This would lead to equation 4 being undefined. We note that, in theory, having the same value for an attribute in more than one row of a database consisting of measurements[*] occurs with probability 0. Nevertheless, due to rounding[†] and abuse of data types[‡] such situations do occur. A not too ugly way out is to define a resolution ε_a of measurement for each attribute a, which can be thought of as the rounding error. Using this resolution, we can equip a hyperinterval with collapsed dimensions of a thickness ε_a in each collapsed dimension a to be able to calculate a positive volume for it. Oftentimes, it is possible to pick every ε_a small enough for hyperintervals with collapsed dimensions to be of lesser volume than any hyperinterval without collapsed dimensions, so that this adequately deals with the problem of reduced dimensionality.

Next to these practical difficulties, the method proposed in Section 7.2 suffers two shortcomings of a more fundamental nature. As mentioned in the paragraph accompanying step 3 of the outline in Section 7.2, the method is designed to mine 'positive exceptions' on the uniform model. It is true that hyperintervals of below average density can be compressing and, indeed, interesting too, but the method simply is not tailored to finding such hyperintervals. At the base of this lies a technical reason. For the method to mine negative deviations from the average density of the database, it should initiate its hyperinterval around two sampled rows with a maximum distance between them conditioned on the fact that no other sampled rows are covered by this initial hyperinterval. In case of our positive deviations, the maximum becomes a minimum and the condition is redundant. Thus it is this condition that makes it infeasible to start in low density regions. Certainly, initiating the hyperinterval as the hyperinterval bounding the two rows in the sample furthest apart is plain wrong.

The second shortcoming we want to discuss in this light is one about pre-

[*] More particular: of samples from random variables with atomless density functions.

[†] For instance: in the Abalone dataset, available in the UCI Machine Learning Repository, the age (a continuous value) of the abalone sea snail is recorded as an integer number of years.

[‡] Not seldom, the supposedly continuous domain of measurement is actually discrete, but with many levels.

cision. Of a mined hyperinterval, the most valuable is its boundary, not its interior. The bounds on attribute values are what is of interest to the data miner. Unfortunately, those bounds are unlikely to lie in high density regions of the database, since these regions are in general preferably inside the hyperinterval. Thus we should not hope for an abundant number of sampled rows near the boundaries of any locally best compressing hyperinterval. As a consequence, since the method will only consider attribute values that appear in the sampled rows, the boundary values are not determined with the best possible precision. If we visualize the consecutive runs through step 4 of the method, we see that only when the steps with which we extend the hyperinterval get larger, we potentially approach (locally) best compression. Informally this means many of the steps are used to establish the uninteresting interior of a hyperinterval instead of ascertaining the valuable boundary.

Lastly, we mention a shortcoming in comparison to Krimp, that is of a different nature than the previous shortcomings. The explosion of the number of found patterns is something Krimp specifically tries to tackle. Therefore it caters for interactions between the patterns and mines pattern sets instead of individual patterns. To our method, patterns are hyperintervals, which are a bit more general than the frequent itemsets Krimp considers as patterns. Although more general in our take on patterns, our method hardly pays attention to interactions between them. The only way a hyperinterval influences another is in that it excludes starting points for the search of follow-up hyperintervals. However, combining this with the MDL-principle-inspired stance that only (locally) best compressing hyperintervals should be regarded, we are confident (Part III) that our method will be proficient in eliminating the bulk of the patterns that make up the explosion in the number of results.

9 Implementation details

In the application of any theory, choices have to be made and (unexpected) hurdles have to be taken. This section touches on some of the choices and difficulties that were faced when implementing our method.

A sample implementation of the algorithm outlined in Section 7.2 was written in Python and is supplied here as Appendix A, and on the internet at `https://github.com/joukewitteveen/hint`. In this implementation, the in-memory representation of a database is a two-dimensional array. If the database is sparse, meaning the dimension, n, of the range, $\mathbb{R}^n$, of the random variable underlying it is large compared to number, m, of rows of the database, this data structure comes with great storage overhead. As a consequence, the program favors certain databases. This is not a trait of the model, but rather an indication of something related to the discrepancy between human and machine complexity as discussed in Section 3.

Not only in memory complexity are databases that are sparse in the above sense problematic, for these databases it can also be hard to define a meaningful metric. This phenomenon, known as the curse of dimensionality, holds that in a high-dimensional hypercube most points are far away from the center for Euclidean-like distance functions. Those distance functions are thus rendered impractical for data mining in high-dimensional databases. Nevertheless, distances and measures are of vital importance to our algorithm and the implementation bundles defaults for both.

For practical reasons, the default measure is based on the Lebesgue measure, where each dimension is scaled so that the absolute difference between the minimal and maximal value for the attribute encoded in that dimension equals 1. Benefits of this measure are that it is easy to calculate the scaling factors and the smallest hyperinterval covering the entire database is a hypercube with sides of length 1. Also, the measure can be evaluated fast, which is desirable because the algorithm does a lot of measure calculations.

The default distance function is one that suits the model well. Recall that the distance function is used to find points close to a hyperinterval and that a hyperinterval is axis-aligned. Therefore, the default distance function is implemented as a rectilinear distance, with the same scaling applied as in the measure.

Next, we want the distance between a point and a hyperinterval to be order-preserving with regard to the shortest distance between the given point and any point in the hyperinterval. Now given a hyperinterval H, let L be the corner of H consisting of the per-dimension minima of H and U be the corner of H consisting of the per-dimension maxima of H. Note that with a rectilinear distance function d, for all points x on the boundary of H, the expression $d(L_H, x) + d(U_H, x)$ evaluates to the same value. From this we see that points with the same shortest distance to any point in H define ellipses*, with foci L_H and U_H (or, for that matter, any two opposing corners of H). This is a characteristic of ellipses based on rectilinear distance functions. Based on these findings, the distance between a point x and a hyperinterval H is implemented as the sum of the distance between x and L_H and the distance between x and U_H, which behaves as required.

To conclude, we will observe the success of using complexity measures instead of probability densities in the implementation. In Section 3 the idea was put forth that complexities are more manageable quantities than probabilities. As it turns out, many of the calculations that are done in the implementation would be outside the mathematical domain of Python when executed on probabilities. For instance, the probability $\frac{M}{m^n}$ associated with discretization constant $-\log \frac{M}{m^n} = n \log m - \log M$ would in general be too small to keep track of, while the last expression is a very tangible one.

* More accurately, they define n-dimensional analogues of an ellipse ($n = 2$) and a prolate spheroid ($n = 3$).

Used	Estimated	error (%)
500	487.17	1.426
600	601.98	0.221
-1	-0.99226	0.774
0	-0.00106	0.106
400	403.98	0.442
750	755.22	0.581
-0.9	-0.90437	0.437
-0.8	-0.79722	0.278

Table 1: Values used in generating the data versus values estimated in a mining run (see: Figure 2). The errors are given as a percentage of the database span in the dimension corresponding to the estimated value.

Part III
Experimentation

In an attempt to identify the strengths and weaknesses of our method, we ran our algorithm on a few databases. Both synthetic and real-world databases were used. This part does not strive to mine the data in those databases, but tries to portray the behaviour of our method.

10 Two-dimensional beams

To demonstrate our data mining procedure, we will mine a contrived two dimensional database. Our database consists of 1500 points (x, y), generated uniformly such that $100 \leq x < 1000$ and $-1 \leq y < 0$ on top of which we put two more sets of points. The first set consists of 300 points, uniformly generated such that $500 \leq x < 600$ and $-1 \leq y < 0$, the second of another 200 with the conditions $400 \leq x < 750$ and $-0.9 \leq y < -0.8$.

We mine this database, using a sample of size 500, tolerating up to 25 consecutive failing attempts to extend a hyperinterval and rerunning until more than 10 non-compressing hyperintervals have been mined in a row. This is the same configuration as used to generate Figure 1. The result of a mining attempt (using the implementation of Appendix A) is visualized in Figure 2.[*] Before the 11 non-compressing hyperintervals that lead to the termination of this mining session, 2 compressing and 6 non-compressing hyperintervals were generated. As is visible in the figure, the non-compressing hyperintervals were deemed infeasible pretty soon after their initiation.

In order to do a qualitative assessment of the outcomes of this particular mining run, we gather the estimated boundary values in Table 1. Observe that the largest error occurs at the estimation of a boundary value of the hyperinterval with the smallest difference in data density between the inside and the outside of the hyperinterval.

[*] This is *not* the same run as used to generate Figure 1, although this run would yield a very similar picture. The runs were executed and studied separately and independently.

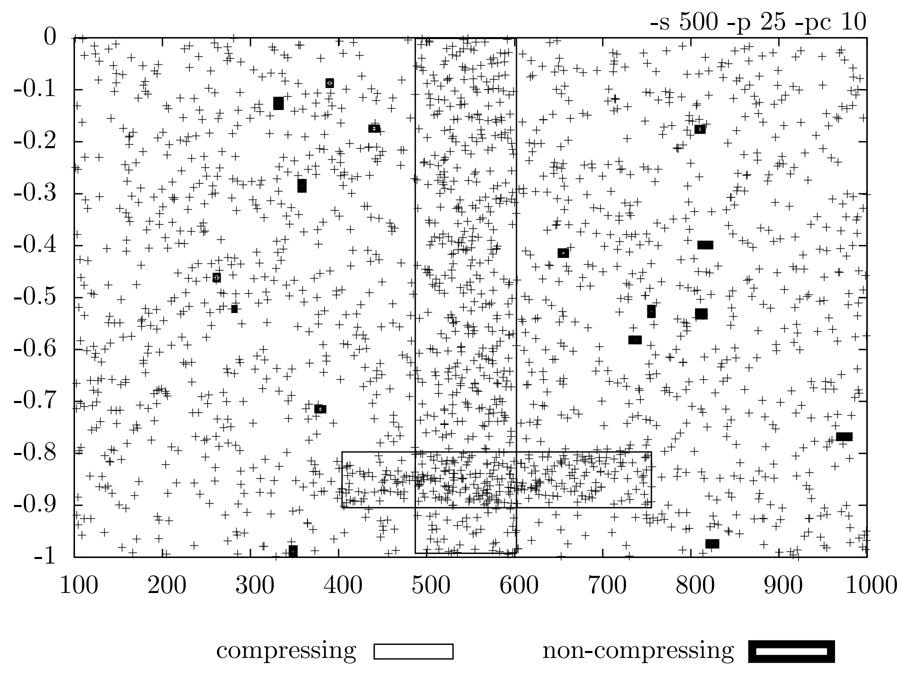

Figure 2: Mined hyperintervals in a synthetic database.

If we continue to post-process the reported hyperintervals, we will find that removing the vertical dimension as depicted in Figure 2 for the tall hyperinterval in that picture improves compression. For that hyperinterval, this is the first dimension considered for removal in the discussed procedure, demonstrating the usefulness of the heuristic.

We now turn to the impact of the choice of the sample size. On our small database, we ran the program of Appendix A using four different sample sizes, one hundred times for each sample size. As a measure of the accuracy of the outcome, we summed the error in estimating the horizontal boundaries of the tall hyperinterval in Figure 2. For the run detailed in Table 1, this combined error in estimating 500 and 600 would have been 1.647%. The result of the experiment is visualized in Figure 3. In the beginning of Section 7.3 it was noted that our method is a consistent estimator, so we might expect the errors to vanish. The figure shows, however, that they do not. Instead, they seem to settle on some fixed non-zero value. We were not in error and the figure is not wrong. Rather, the database is too small to accommodate really large sample sizes and because of this limitation the parameters of the locally best compressing hyperinterval are in fact different from the parameters that were used to generate the database. Running the program with a sample size equal to the database size (the procedure is deterministic in this case) we obtained the 'error' of the locally best compressing hyperinterval: 1.295%. The best boundary values were 489.12 and 599.22. In Figure 3, we see that it is very plausible that these values are estimated with increasing precision and likelihood as the sample size increases, since the errors could very well be converging to

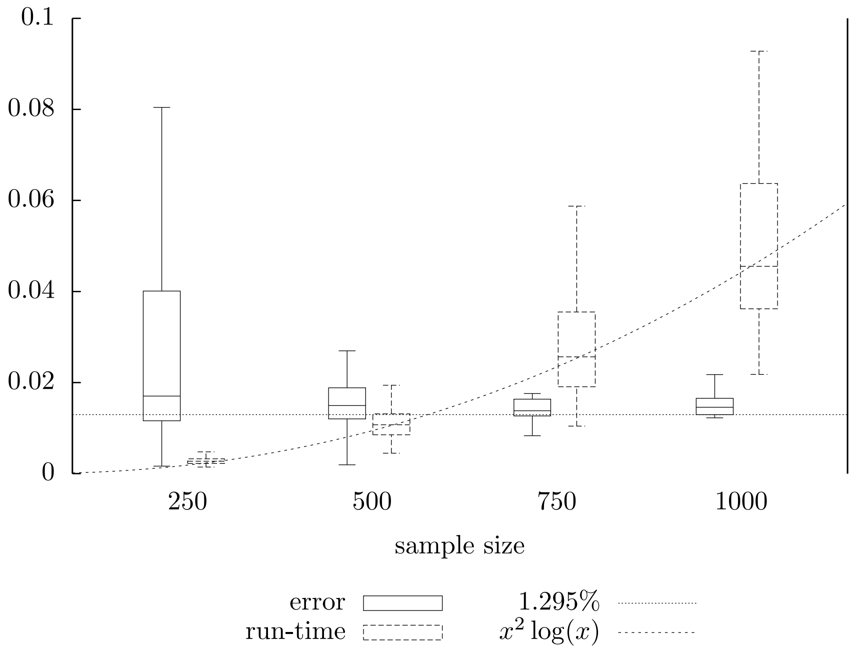

Figure 3: Total error of estimation of width parameters of the tall hyperinterval in Figure 2 and the run-time (on a linear, but unlabeled, time scale) for several sample sizes. The chosen perseverance (admissible consecutive non-improving extensions) was one fifth of the sample size. Suggestively, the big O class function of the run-time complexity is fitted to the median run-time. Outliers are not plotted.

the error of this locally best compressing hyperinterval.

Also shown in Figure 3 are boxplots of the time the program took to find the two hyperintervals. Since the exact time is hardware dependent, the scale of the run-times is unlabeled. As expected, the run-time increases with the sample size. The curve of the defining function of the run-time complexity class of the algorithm ($s^2 \log s$, see: Section 7.3) suggests that the run-time complexity bound might be tight. Of course, the figure presents too little data to justify such a claim and it would be more correct to place the curve so that the lengthiest runs would fall below it. However, this arrangement provides evidence that the average run-time of the algorithm might belong to the $\Theta(s^2 \log s)$ class of functions, which implies the worst case class the algorithm belongs to is not smaller than $\mathcal{O}(s^2 \log s)$.

11 Not easily bounded correlations

As the database of the previous section was devised specifically with our algorithm in mind, it is no surprise that the algorithm did well on it. We will now look at a database that features a regularity that the algorithm is not looking

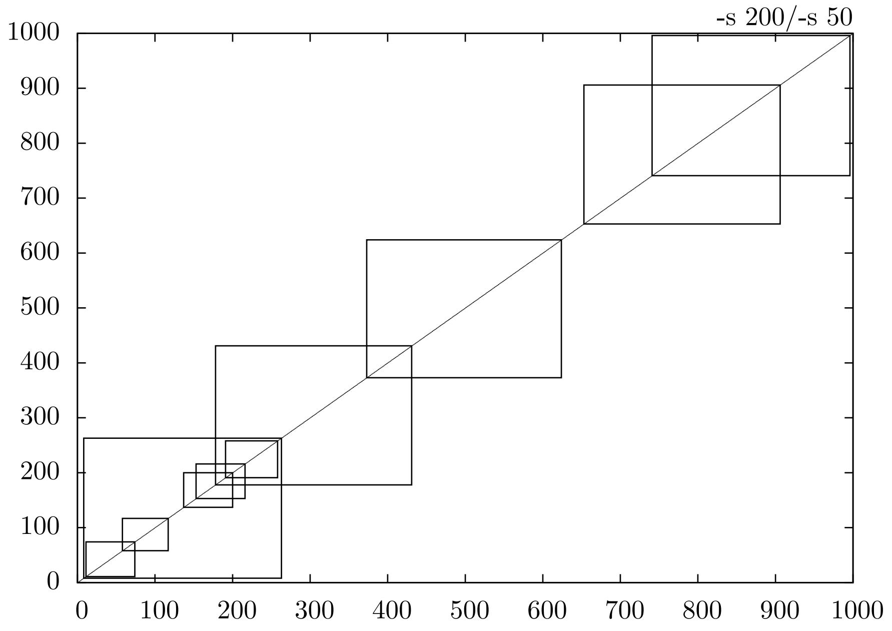

Figure 4: Mined hyperintervals in a synthetic database and inside a mined hyperinterval.

for directly.

For ease of display, our new database is again two dimensional. It consists of 1000 points (x, x), where x ranges over the integers from 0 to 999. It is clear that this database does not contain axis-aligned characteristics, so we cannot expect the parameters estimated by our method to make sense as easily as they did for our previous database. Since the database contains no noise, we do not have to worry about unlucky extensions of hyperintervals and the only parameter that matters is the sample size. The larger rectangles in Figure 4 depict the hyperintervals resulting from a mining run on our database. All of them were compressing.

The hyperintervals are all of approximately the same size (while the data is evenly spaced, the sample need not be) and lie more or less randomly along the diagonal/data. In Figure 4 the data is not covered by the hyperintervals. Note that the expected proportion of this database that is not covered by the set of mined hyperintervals is a monotonically decreasing function of the sample size. Because all data in the database contributes to the regularity underlying the database, the sample needs to be rather large with regard to the size of the database before we can expect all of the pattern to be included in the mining results.

One could argue that the hyperintervals resulting from the mining attempt do not provide the data miner with a meaningful description of the regularity in this database. In such cases it can, as this database demonstrates, be beneficial to recurse and mine for locally best compressing hyperintervals inside a reported hyperinterval. A possible outcome of such an action is also depicted in Figure 4. Here, we ran the algorithm on the portion of the database that was covered by the lower left mined hyperrectangle that came about in the first mining run. The figure shows the collection of mined sub-hyperintervals within the surrounding hyperinterval is very similar in structure to the collection of the large hyperintervals in the database. This should not come as a surprise, since the database restricted to any of the (larger) hyperintervals behaves exactly as a scaled copy of the entire database, be it with less points. If we would continue this recursion procedure, we would obtain an increasingly fine-grained match of the data. In fact, when the sample size would be big enough to make sure the data was covered and the database held enough data to allow for high levels of recursion, we would obtain a fit of the data through our continued recursion. This illustrates how hyperintervals can be used to model even patterns that are quite unlike hyperintervals. However, it is up to the data miner to recognize the pattern that is modeled.

12 Biogeography of European land mammals

Let us now turn our attention to a real-world database. We will look at a database consisting of presence/absence records of European land mammal species collected by the Societas Europaea Mammalogica. For each of 2183 grid cells covering 50×50 km of Europe, this database contains the following attributes:

- Latitude and longitude of the center of the cell.
- Mean temperature, maximum temperature, minimum temperature and precipitation per month for the cell.
- Nineteen bioclimatic variables for the cell which are considered biologically meaningful.
- Presence/absence in the cell for 101 species of land mammals.

The last class of attributes is binary data, the rest is numeric.

In order to cope with the binary data, we extend the rectilinear distance function introduced in Section 9, which we use for the numeric data, with an extra component. Given two sets of present species, the distance we assign to be between them is determined by the species in the symmetric difference between those sets. For each species in the symmetric difference, we add an amount to the distance that is based on the binary entropy of that species in the database. More specifically, if a species occurs k times in the database, it adds

$$-\frac{k}{m}\log_2\frac{k}{m}-\frac{m-k}{m}\log_2\frac{m-k}{m}=\log_2 m-\frac{1}{m}\left(k\log_2 k+(m-k)\log_2(m-k)\right)$$

to the distance between two sets of species if it is in the symmetric difference of those sets.

We will mine a part of this database for hyperintervals and discuss the proceedings. The part of the database we will mine consists of the bioclimatic attributes plus the presence/absence data. This forces the outcomes to be defined in terms of the 'biologically meaningful' parameters, and saves some information for the assessment of the mining results.

Our initial run is based on a sample of size 1000. This is just short of half the database, so most relevant patterns will likely be represented. There is no need to go with a smaller sample, as results were generated within an acceptable amount of time. Furthermore we will tolerate up to 100 consecutive failed attempts to enlarge a hyperinterval, before considering it to be optimal. It turned out no non-compressing hyperintervals were found before the entire sample was covered. This is, in a way, undesirable and will be discussed a few paragraphs down. As a consequence, we do not need to worry about tolerating non-compressing hyperintervals. In pruning dimensions too, it was not necessary to bother with failed pruning attempts. Many binary dimensions spanned all possibilities, meaning the hyperinterval went from '0' to '1'. For these dimensions, there was no chance of a loss in compression when pruning them, for pruning improved the compression by exactly Δ_c (which equals $\log \frac{m^2}{2}$ in equation $9'$). The real-valued attributes were almost all relevant in all mined hyperintervals.

The one case in which a real-valued attribute could be removed to improve compression, was also special in that it was the only hyperinterval that required a positive presence of certain species. All other hyperintervals were either indifferent to the presence of specific species, or modeled their absence. This phenomenon is also observed in the experiment of Section 13. The bioclimatic variable that was disregarded in this particular hyperinterval was the one that holds the mean difference between monthly maximum and minimum temperatures, the mean diurnal range. The species that were necessarily present in the hyperinterval were the *Vulpes vulpes* (Red fox), the *Capreolus capreolus* (European roe deer) and the *Lepus europaeus* (European hare). All three are in the top ten of most widespread (not meaning abundant) species in the database.

Existing studies of this database, for instance [9], use apparent coherence of clusters to fortify their claim of meaningful results. Although this method is recognized as a questionable one, we can at least show that our results are no worse than what was presented earlier. Figure 5 shows the elements of the database that are covered by the hyperinterval of the previous paragraph, according to their geographical position.

As noted, mining runs on the mammals database quickly cover the entire sample, showing that most, if not all, data points are a member of some pattern. A similar situation occurred in the previous section and once again, we can try to approximate more complex patterns by repeatedly mining hyperintervals in a recursive fashion. A sample recursion is also shown in Figure 5. As we are no experts on the biogeography of European land mammals, we will refrain from interpreting mining results or linking the results to known facts.

Although the geographical boundaries of these and other mined hyperintervals show similarities to the boundaries depicted in [9], we will not immediately claim that the boundaries are of actual interest to someone investigating this database. In our mining results too, geographical features such as the Alps can be distinguished, but instead of jumping to conclusions we will also assess the quality of the results in another way.

In the same way we recombined our results with the positional data that was concealed from the algorithm (Figure 5), we will now recombine the results

-s 1000/-s 500

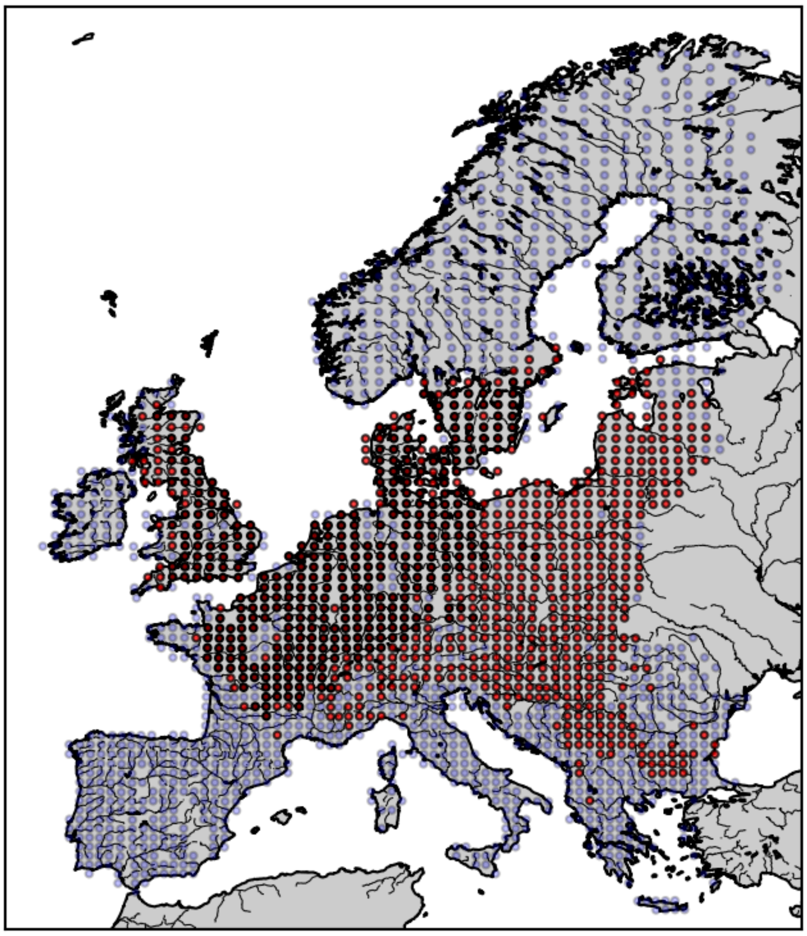

Figure 5: Spatial distribution of database elements inside (red) and outside (blue) a mined hyperinterval. Within the hyperinterval, the dark red elements designate a sub-hyperinterval.

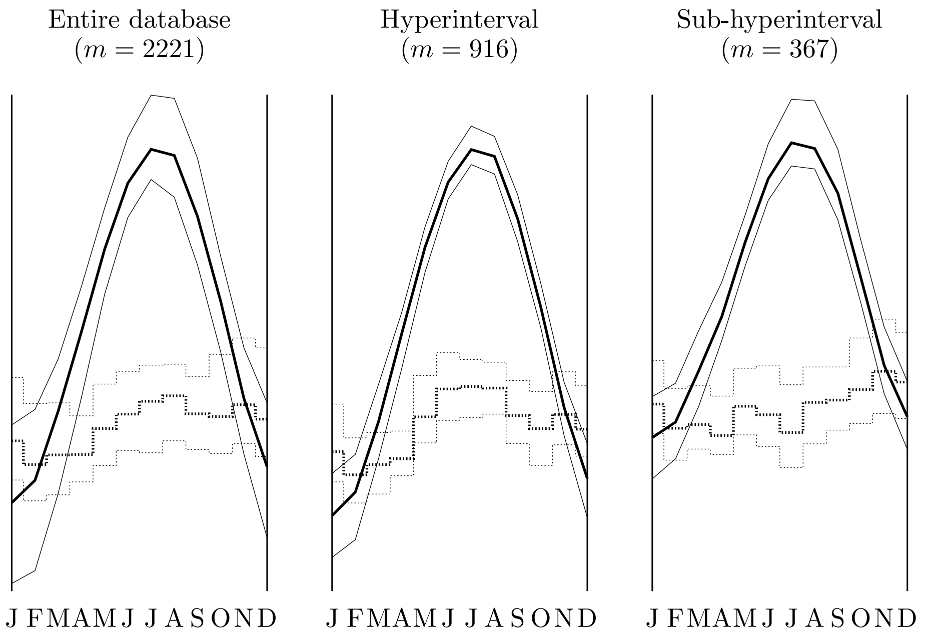

Figure 6: Climate data for portions of the database as shown in Figure 5. The solid lines represent the first, second (median) and third quartile of the mean temperatures of each entry in the database per month. The dashed lines represent the same quartiles for the recorded precipitation. The temperature scale ranges from $-5°$C to $20°$C and the precipitation scale ranges from 0mm to 175mm. The ranges are omitted from the plot for legibility.

with some of the climate data that was kept hidden to the algorithm. Idealy, the graphs in Figure 6 would demonstrate that the hyperintervals correlate with climate zones. With our small experiment it is, unfortunately, hard to tell whether this is the case. We do note that the spread of the course of both the temperature and the precipitation data decreases from the entire database to the hyperinterval. Additionally, the precipitation within the hyperinterval shows a hint of structural variation throughout the year that is absent in the graph for the entire database. The graph for the sub-hyperinterval shows a winter that is substantially wetter and warmer than both the surrounding hyperinterval and the entire database. In terms of the Köppen climate classification, the entire area covered by the sub-hyperinterval lacks a dry period, whereas this cannot be stated with equal confidence for either the entire surrounding hyperinterval or the all of the database.

13 Comparative performance

Whilst our method is inspired by Krimp, comparison with Krimp is hard. One of the reasons for this is that an obvious way to compare the performance of our method to that of Krimp is by comparing outcomes in a context where the program would function as a classifier. However, we did not investigate using our method in cases of a classifier nature, rendering this approach infeasible. Instead, we will look at itemset mining.

Before looking at any particular database, we note that we have generalized some notions common to Krimp. Firstly, because Krimp is designed for nominal data, complexity calculations in Krimp are absolute, unlike in our case, where we have to define a discretization constant for every mining instance. Secondly, to Krimp and others an itemset is defined in terms of positive occurrences, as was already noted in Section 5. The same holds for what is called a *tile* in [4, 10]. In [15] the same generalization of 'itemsets' that we have made is made when a method dubbed Less (Low-Entropy Set Selection) is introduced. The generalization holds that every attribute/item is given one of three predicates in every hyperinterval/generalized itemset, namely it is either necessarily 1/present, necessarily 0/absent or the hyperinterval/generalized itemset is indifferent to the value/presence of the attribute/item. This makes that we can replace equation 5 with $n \log 3$ and not add a discretization constant to it.

In principle, these observations are enough to start mining nominal datasets. As a test case, we will look at the database devised in [10] based on papers' abstracts from ICDM up to 2007. Stop word removal and stemming was applied. Every abstract is considered a database entry and every stemmed word an attribute. The resulting database has $m = 859$ and $n = 3933$. Because of the high dimensionality of this database, it is unwise to use a Euclidean distance or a rectilinear distance (which in this case is related to the symmetric difference of two itemsets) since both can behave rather erratic in spaces of high dimensionality. This is part of the curse of dimensionality that was mentioned in Section 9 and will be discussed in greater detail further down this section. We will use the entropy based correlation distance introduced in Section 12 for mining experiments on our present database.

A typical mining run on the abstracts database gives rise to a picture much like Figure 7. Immediately we note that all four runs almost entirely coincide,

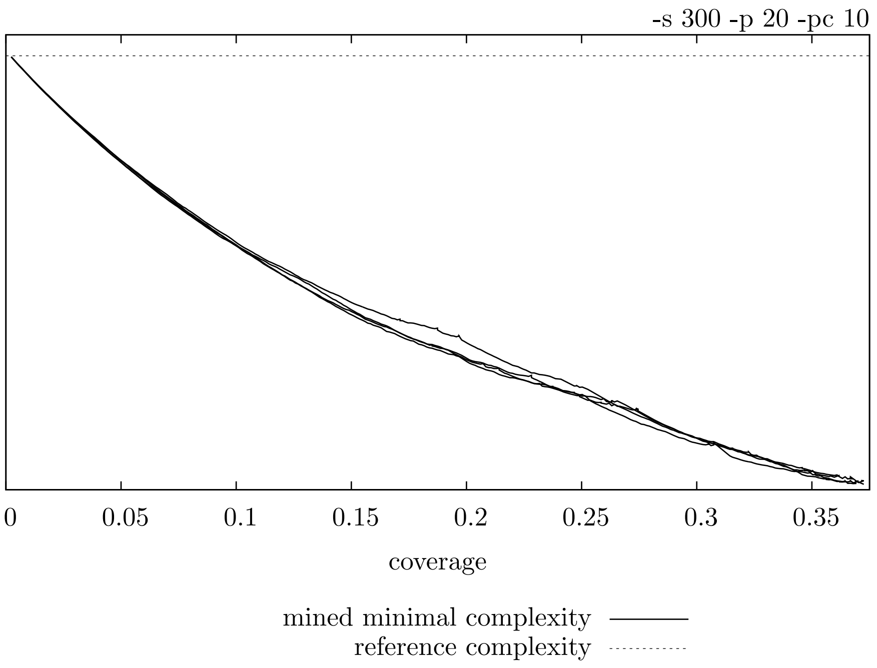

Figure 7: The equivalent of Figure 1 for the abstracts database. All four depicted runs were compressing from the start and little local minima were encountered. The sample was exhausted each run before a locally *best* compressing hyperinterval was found. After the fourth run the mined hyperintervals covered the entire sample.

in contrast to what was seen in Figure 1. Indeed, when we look at the similarities between the mining results, we see that every two mined hyperintervals are identical on approximately 94 percent of the attributes. Apparently, no matter where we initialize a hyperinterval, we will grow it into more or less the same pattern. A visual interpretation can be derived from Figure 4: the mined hyperintervals would overlap almost entirely.

If we take a closer look at the hyperintervals that emerge from the mining, we see that no attribute is deemed necessarily present.[*] This is understandable as our method only extends a hyperinterval and in this database every row consists of mainly (98.8% on average) zeros. To make matters worse, for this database we have $n > m$ and as part of the curse of dimensionality it follows that the database is sparse.[†] Indeed, about half of the attributes occur in only one element of the database and less than 5% occur in more than 45 elements of the database.

Now if we interpret a mining outcome as the result of selecting a subset

[*] This is no theoretical must, but a persistent observation.

[†] The number of possible database elements grows superpolynomially in n. Because of the nature of the database elements with many ones are unlikely, thus the number of possible database elements need not grow exponentially in n. Nevertheless it follows that m should increase superpolynomially in n for the database not to become sparse.

of the database and reporting `1` for all attributes that are present in all of the subset and `None` for all attributes that are present in some (but not all) of the subset, we can associate with an outcome an itemset consisting of all attributes for which `1` was reported. Now given our previous observations, it comes as no surprise that such an itemset is empty for (nearly) all mined hyperintervals. Luckily, though, interesting frequent itemsets should be of the form just described. Given an itemset, one could look at the subset of the database consisting of all database elements that have this itemset (i.e., are '1' on all attributes contained in the itemset) and if more attributes are present in all of this subset it would be interesting to extend the itemset with them. The notion 'frequent' would be formalized by requiring the hyperinterval determined by the subset to be compressing in the sense of Part II. However, this requisite is not enough, as it also happens to formalize infrequent itemsets. Thus a frequent itemset should be of the above form and the average database density within the corresponding hyperinterval should exceed the average database density outsde that hyperinterval.

From this we can at least conclude that our method could find interesting frequent itemsets. One way to obtain non-empty itemsets — typical runs such as the one corresponding to Figure 7 result in the empty frequent itemset being mined over and over again — is by recursion, such as is done in Section 11 and Section 12. Experiments have shown, however, that the first few recursions continue to produce the empty itemset.

Another possibility is by enforcing the hyperinterval to cover the point consisting of 1's for all attributes of the database, as laid out in Section 5. This can be done by initializing each hyperinterval so that the upper bound in every dimension is 1, thus altering step 3 of our procedure from Section 7.2. Unfortunately, the initial hyperintervals that this approach gives are *in*frequent itemsets and our algorithm concludes that compression cannot be improved by an extension based on a single database element (out of a fixed sample). In part, this is caused by our choice of a constant complexity 'cost' for the specification of a hyperinterval ($n \log 3$), while we expect small itemsets (of much less than $n = 3933$ items). This is a discrepancy between the miner's prior knowledge and the model. Updating the model for itemset mining has the consequence that the derived equation 9 would no longer be independent of the hyperinterval and the complexity calculations would have to be altered.

Lastly, because of the sparseness properties we observed for the abstracts database, our correlation distance might still be too uninformative. This means the distance function should potentially take more complex interactions between attributes into account or depend on the hyperinterval concerned, for our greedy method to work. In the end, the success of the greediness of our method depends largely on the quality of our distance function, which in this case acts as a heuristic for precisely the kind of interactions we are after.

We are left in a hopeful, but unsatisfying situation. Although frequent itemsets such as those mined by Krimp are likely to correspond to locally best compressing hyperintervals in our simple model, they are not found easily by our algorithm. Whether this is inherent to the algorithm's design, or this can be solved by a more clever metric or by making the model more complex is something further research has to show.

14 Conclusion

In this thesis, we have shown that the MDL principle can be used for mining data from uncountable sets too, if there is a σ-finite, strictly positive reference measure on that set. This is proven by the construction of an equivalent of the Kraft–McMillan theorem for Radon spaces.

An attempt at using this theorem through a simple model and a mining heuristic based on a metric on the uncountable set was fruitful on synthetic databases. Usage on real-world databases indicated that the applicability of the model depends on the database. This suggests the need for a more complex model, a broader model class or a more advanced metric.

A Source code

hint.py

```
#! /usr/bin/env python
"""Hyperinterval finder

Finds hyperintervals in a (numerical) database that have high density.

(c) 2011-2012 Jouke Witteveen
"""

import argparse, random
import hint_tools
# Customize the database measure by
#   import <custom_db_measure> as db_measure
# before importing this file
try:
  from __main__ import db_measure
except ImportError:
  import db_measure

# logarithmic function
log = hint_tools.log

db = None
sample = None
debug = None
params = {}


### ARGUMENT PARSING AND VALIDATION ###

def cli_args( *argv ):
  """Command line interface arguments"""
  global db, sample, debug
  parser = argparse.ArgumentParser( description = "Hyperinterval finder." )
  parser.add_argument( '-s', '--sample', metavar = 'SIZE',
    type = int, required = True,
    help = "sample size to determine hyperinterval boundaries" )
  parser.add_argument( 'database', type = argparse.FileType( 'r' ),
    help = "database file containing one line per entry" )
  parser.add_argument( '-p', '--perseverance', type = int, default = 0,
    help = "eagerness to not end up in local minima" )
  parser.add_argument( '-pc', '--thoroughness', type = int, default = 0,
    help = "tolerated consecutive non-compressing hyperintervals" )
  parser.add_argument( '-pd', '--dim-thoroughness', metavar = 'DIM_THOROUGH',
    type = int, default = -1,
    help = "tolerated consecutive non-discardable dimensions" )
  parser.add_argument( '-l', '--log', metavar = 'FILE',
    type = argparse.FileType( 'w' ), required = False,
    help = "log file to record all considered hyperintervals" )
  args = parser.parse_args( *argv )

  db = tuple( tuple( map( float, row.split() ) ) for row in args.database )
  if not ( 3 <= args.sample <= len( db ) ):
    parser.error( "SIZE should be between 3 and the size of the database." )
  if not ( 0 <= args.perseverance <= args.sample - 3 ):
    parser.error( "PERSEVERANCE should be between 0 and SIZE-3." )
  if args.thoroughness < 0:
    parser.error( "THOROUGHNESS should not be negative." )
  if not ( -1 <= args.dim_thoroughness < len( db[0] ) ):
    parser.error( "DIM_THOROUGH should be between -1 and dimensionality-1." )
  sample = tuple( random.sample( range( len( db ) ), args.sample ) )
  debug = args.log
  params['perseverance'] = args.perseverance
  params['thoroughness'] = args.thoroughness
  params['dim_thorough'] = args.dim_thoroughness
  if params['dim_thorough'] >= 0: hints.postproc = prune


### COMPLEXITY RELATED MACHINERY ###

def comp_hint_comp( hint ):
  """Comparative hint complexity calculation."""
  inside_count = hint_tools.covered( hint, db )
  outside_count = len( db ) - inside_count
  hint_volume = db_measure.volume( *hint )
  complexity = inside_count * ( log( hint_volume ) - log( inside_count ) )
  if outside_count != 0:
    complexity += outside_count \
                  * ( log( db_volume - hint_volume ) - log( outside_count ) )
  if debug:
    debug.write( "{}\t{}\t{}\n".format(
      hint_volume, inside_count / len( db ), complexity ) )
  return complexity


# SHORTCOMING: The boundaries are often found in low density regions.
#              The sample is not the most accurate source of coordinates in
#              those situations.
def grow_hint( hint, sample ):
  """Grow the hyperinterval to its maximal informativeness."""
  complexity = comp_hint_comp( hint )
  sample_out = [ i for i in sample
                   if not hint_tools.is_covered( db[i], hint ) ]
  perseverance = params['perseverance']
  while sample_out:
    candidate = db[sample_out.pop( min( range( len( sample_out ) ), key = \
      lambda i: db_measure.distance( db[sample_out[i]], hint[0] ) \
                + db_measure.distance( db[sample_out[i]], hint[1] ) ) )]
    candidate_hint = tuple( map( min, candidate, hint[0] ) ), \
                     tuple( map( max, candidate, hint[1] ) )
    candidate_comp = comp_hint_comp( candidate_hint )
    if candidate_comp < complexity:
      hint, complexity = candidate_hint, candidate_comp
      perseverance = params['perseverance']
    else:
      perseverance -= 1
      if perseverance < 0: break
  else:
    print( "Sample exhausted. "
           "Try a larger sample, or lower your perseverance." )
  return hint, complexity


### ENTRANCE HOOKS ###

def hints():
  """Generate all compressing hyperintervals."""
  global db_bound, db_volume, db_base_comp, model_comp
  db_bound = db_measure.measure_init( db )
  db_volume = db_measure.volume( *db_bound )
  db_base_comp = len( db ) * ( log( db_volume ) - log( len( db ) ) )
  model_comp = 2 * log( db_volume ) - len( db[0] ) * log( 2 ) \
               + 2 * db_measure.discretization_const
  hint_tools.queue_init( db, sample, db_measure.distance )
  if debug: debug.write( "#size\tcoverage\tcomplexity\n" )
  thoroughness = params['thoroughness']
  hint = hint_tools.next_hint()
  while hint:
    hint, complexity = grow_hint( hint, sample )
    _hint, complexity, keep = hints.postproc( hint, complexity )
    yield _hint, complexity, keep
    if keep:
      thoroughness = params['thoroughness']
    else:
      thoroughness -= 1
      if thoroughness < 0: break
    if debug: debug.write( "\n\n" )
    hint = hint_tools.next_hint( hint )

hints.postproc = lambda hint, complexity: \
  ( hint, complexity, complexity < db_base_comp - model_comp )


def prune( hint, complexity ):
  """Post-process a hyperinterval by pruning superfluous (full) dimensions."""
  dimensions = len( db[0] )
  dim_comp = model_comp / dimensions
  zbound = list( zip( *db_bound ) )
  zhint = list( zip( *hint ) )
  fullness = db_measure.fullness( hint )
  thoroughness = params['dim_thorough']
  for i in sorted( range( len( hint ) ), key = fullness.__getitem__,
                   reverse = True ):
    zcandidate = zhint[:i] + zbound[i:i + 1] + zhint[i + 1:]
    candidate_comp = comp_hint_comp( tuple( zip( *zcandidate ) ) )
    if candidate_comp <= complexity + dim_comp:
      zhint, complexity = zcandidate, candidate_comp
      dimensions -= 1
      thoroughness = params['dim_thorough']
    else:
      thoroughness -= 1
      if thoroughness < 0: break
  zhint = [ zhint[i] if zhint[i] != zbound[i] else ( None, None )
            for i in range( len( zhint ) ) ]
  return tuple( zip( *zhint ) ), complexity, \
         complexity < db_base_comp - dimensions * dim_comp


if __name__ == "__main__":
  cli_args()
  try:
    for run, ( hint, complexity, keep ) in enumerate( hints() ):
      print( "Hyperinterval {}:".format( run ), hint, complexity,
             "KEPT" if keep else "DISCARDED" )
  except KeyboardInterrupt:
    print( "Interrupted" )
  # If the volume is normalized to 1, the following prints 0.
  # This is not a bug.
  print( "Single uniform data complexity:             ",
         len( db ) * log( db_volume ) )
  print( "Comparative single uniform data complexity: ", db_base_comp )
  print( "Discretized double uniform model complexity:", model_comp )
```

hint_tools.py

```
"""Generic database measurement

General purpose hyperinterval tools.
This file is part of Hint, the hyperinterval finder.

(c) 2011-2012 Jouke Witteveen
"""

# Units are nats when using natural logarithms
from math import log


def bounding_hint( *a ):
  """The smallest hyperinterval covering all arguments."""
  return tuple( map( min, *a ) ), tuple( map( max, *a ) )


def is_covered( a, hint ):
  """Whether record a is covered by the hyperinterval."""
  return all( hint[0][i] <= x <= hint[1][i] for i, x in enumerate( a )
                                            if hint[0][i] is not None )


def covered( hint, db ):
  """The number of records covered by the hyperinterval."""
  count = 0
  for row in db:
    if is_covered( row, hint ): count += 1
  return count


def queue_init( db, sample, key ):
  """Initialize the internal queue of next_hint."""
  # Runtime is in O( len( sample ) ** 2 * log( len( sample ) ) ). That is slow.
  def _key( ij ): return key( db[ij[0]], db[ij[1]] )
  next_hint.db = db
  next_hint.queue = sorted( [ ( sample[i], sample[j] )
                              for i in range( len( sample ) )
                              for j in range( i ) ], key = _key )


def next_hint( exclude = None ):
  """The next hyperinterval to expand.

     Points in an excluded interval are not considered."""
  if exclude:
    next_hint.queue = [ ( i, j )
                        for i, j in next_hint.queue
                        if not is_covered( next_hint.db[i], exclude )
                           and not is_covered( next_hint.db[j], exclude ) ]
  if len( next_hint.queue ) == 0:
    print( "Sample exhausted." )
    return None
  return bounding_hint( next_hint.db[next_hint.queue[0][0]],
                        next_hint.db[next_hint.queue[0][1]] )
```

db_measure.py

```
"""Generic database measurement

General purpose measures on numerical databases of arbitrary dimension.
This file is part of Hint, the hyperinterval finder.

(c) 2011-2012 Jouke Witteveen
"""

import hint_tools
from sys import float_info

discretization_const = None


def measure_init( db ):
  """Establish a bounding box around the database and normalizing factors for
     all columns, so that distances become comparable."""
  global db_scale, discretization_const, distance1d
  def distance1d( x, y, z ): return abs( ( x - y ) / z )
  db_lb, db_ub = hint_tools.bounding_hint( *db )
  db_scale = tuple( y - x for x, y in zip( db_lb, db_ub ) )
  discretization_const = len( db_scale ) * hint_tools.log( len( db ) )
  volume.epsilon = float_info.epsilon ** ( 1 / len( db_scale ) )
  return db_lb, db_ub


def distance( a, b ):
  """The distance between two database records."""
  # A rectilinear distance suits the model
  return sum( map( distance1d, a, b, db_scale ) )


def volume( a, b ):
  """The volume of the hyperinterval between two database records.

     Subspaces are given a 'thickness' of epsilon.
     A record with itself yields a hyperinterval of volume 0.
     Two identical records yield a hyperinterval of strictly positive volume.
     The volume of the bounding box around the database is normalized to 1."""
  if a is b: return 0
  V = 1
  for side in map( distance1d, a, b, db_scale ):
    V *= ( side or volume.epsilon )
  return max( V, float_info.min )


def fullness( hint ):
  """Coverage per dimension."""
  return tuple( map( distance1d, hint[0], hint[1], db_scale ) )
```

Acknowledgement

The idea to look into data mining for data taken from uncountable sets came from Arno Knobbe. He has been very helpful in starting this project, for which I am grateful. I would also like to thank Peter Grünwald very much for his willingness to help in this project and for his amazing capability to quickly understand my thoughts, even when I struggled to formulate them. Special thanks goes to Wouter Duivesteijn, who was the most reliable guidance I had in this project. Parts of this thesis, among which Figure 6, are based on his ideas and suggestions. His joy in science is equally inspirational as Peter Grünwald's understanding of mathematics. Lastly I would like to thank Lennart Bal, for every time he said: "Hup, sommen maken!" by which he probably meant I had to get on with this thesis.

References

[1] Heinz Bauer. *Measure and Integration Theory.* Walter de Gruyter, 2001.

[2] Roger Brown. Reference in memorial tribute to Eric Lenneberg. *Cognition*, 4, 1976.

[3] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory.* Wiley, second edition, 2006.

[4] Tijl de Bie, Kleanthis-Nikolaos Kontonasios, and Eirini Spyropoulou. A framework for mining interesting pattern sets. *SIGKDD Explorations*, 12, 2010.

[5] David H. Fremlin. *Measure Theory*, volume 2. Torres Fremlin, second edition, 2010.

[6] Bert Fristedt and Lawrence Gray. *A Modern Approach to Probability Theory.* Birkhäuser, 1997.

[7] Liqiang Geng and Howard J. Hamilton. Interestingness measures for data mining: A survey. *ACM Computing Surveys*, 38, 2006.

[8] Peter D. Grünwald. *The Minimum Description Length Principle.* The MIT Press, 2007.

[9] Hannes Heikinheimo, Mikael Fortelius, Jussi Eronen, and Heikki Mannila. Biogeography of european land mammals shows environmentally distinct and spatially coherent clusters. *Journal of Biogeography*, 34, 2007.

[10] Kleanthis-Nikolaos Kontonasios and Tijl de Bie. An information-theoretic approach to finding informative noisy tiles in binary databases. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, 2010.

[11] Kleanthis-Nikolaos Kontonasios, Jilles Vreeken, and Tiel de Bie. Maximum entropy modelling for assessing results on real-valued data. In *Proceedings of the IEEE International Conference on Data Mining*, 2011.

[12] Dennis Leman, Ad Feelders, and Arno Knobbe. Exceptional model mining. In *Proceedings of ECML PKDD*, volume II, 2008.

[13] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications.* Springer, third edition, 2008.

[14] Jim Thomas. A treatment of data mining technologies. White Paper, MetaTech Consulting, Inc., 2003.

[15] Jilles Vreeken. *Making Pattern Mining Useful.* PhD thesis, Universiteit Utrecht, 2009.

[16] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining, Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, third edition, 2011.