Algorithmic Information Theory: A Brief Introduction Motivating KRIMP

prof. dr Arno Siebes

Algorithmic Data Analysis Group
Department of Information and Computing Sciences
Universiteit Utrecht
Motivation

**Krimp** relies on the Minimum Description Length Principle (MDL) to select a small set of itemsets, you may wonder, e.g.,

- where MDL comes from
- why it is a good idea in general
- and why it is a reasonable idea to use it for pattern set mining

This short lecture introduces you to Algorithmic Information Theory

- and by that hopefully answer some of your questions
Induction
How Reliable is Pattern Set Mining?

How trustworthy are our results?

I am sure that you are willing to bet a modest sum on
▶ the outcome of your supervised learning algorithm
▶ only modest because even if you have taken measures against all possible pitfalls you know that things may still go horribly wrong

But would you bet the life of your first-born on
▶ the outcome of your unsupervised learning algorithm?
▶ that your cluster algorithm has discovered all relevant structure present in the data?
▶ that you selected the “right” set of patterns?

The reason is *the problem of induction*
Induction

The OED defines induction (in the sense we use it) by

*the process of inferring a general law or principle from the observation of particular instances*

in contrast with deduction where we (may) apply general laws to specific instances.

For deduction, well, at least for, say, First Order Logic, we can prove that it is sound

- if the premisses are true, so will be the conclusion

The question is if there is a similarly good procedure for induction, i.e., (Stanford Encyclopaedia of Philosophy):

*can we justify induction; to show that the truth of the premise supported, if it did not entail, the truth of the conclusion?*

This is known as *The Problem of Induction*
David Hume

Mid 18th century the philosopher David Hume argued

No! There is no justification for induction

There is no procedure that will always, guaranteed,
▶ give you the true general rule

Hume was actually more concerned with the more general induction problem
▶ conformity betwixt the future and the past

how do we know that regularity we have observed in the past will also be shown in the future
▶ (before Newton): will the Sun also rise tomorrow?

According to Hume all justifications are circular:
▶ the inductive step was successful yesterday, so it will also work today
Data Science’s Limited Inductive Problem

Data science’s problem is that with a finite number of observations, many hypotheses are consistent, which one to choose?

Given a finite number of data points
  ▶ there are infinitely many functions that go through them

If your adversary gives you a number of data points
  ▶ and you guess the general rule, and predict the next data point

your adversary has enough leeway to think of another, consistent rule
  ▶ and generate a next data point that proves you wrong

no matter how many data points you have seen, and guesses you have made, you’ll always give a wrong answer

So, data science’s limited induction problem doesn’t have a solution either.
Bummer!

Philosophers thought about this since at least the ancient Greeks
➤ Epicurus (300 BC) had the principle of multiple explanations
    ➤ discard no hypotheses that is consistent with the observations
➤ William of Ockham (1287 - 1347) had the principle of simplicity
    ➤ Numquam ponenda est pluralitas sine necessitate (Plurality must never be posited without necessity)
    ➤ the clue: discard all hypotheses except the simplest one that is consistent.
    ➤ But, then? Which one is the simplest?

(Data) scientists have learned to be both
➤ pragmatic (and the very existence of computers shows that this pays off pretty well).
➤ and extremely careful
    ➤ hence the existence of the Scientific Method and (even more) Statistics
The Scientific Method

Induction is not done blindly, rather
▶ Formulate a Hypothesis
▶ Devise an Experiment that tests this Hypothesis
▶ Execute that Experiment
▶ Decide on the Hypothesis based on the outcomes of the Experiment

Presto;

New Knowledge Is Acquired

There is, however, a wee problem:

how do you decide based on the outcome of the experiment?

That is, the problem of Induction is still very much alive
▶ slaying dragons is never easy
Enter: Statistics

Formulate a hypothesis and an alternative
▶ usually $H_0$ and $H_1$

And choose the most likely one given the outcome of the experiment

Quite often
▶ formulate a hypothesis $H_0$ that is opposite to what you actually think
▶ reject that hypothesis if the outcome of the experiment would be unlikely if that hypothesis were true

Notoriously hard to do right
▶ given the many errors regarding p-values
▶ in published, peer-reviewed scientific literature
Why Rejection?

Why do scientists

- *reject* hypotheses rather than *confirm* them?

The reason is simple

- seeing your $573827125364384937236512^{th}$ white swan does not unequivocally prove that
  
  *all swans are white*

- spotting just one black swan

  *nullifies that hypothesis once and for all*

That is

- because of the problem of Induction

Slaying dragons really is hard
Models

Often the hypotheses we consider are actually models. More precisely

\[ D \sim f(\vec{x} | \Theta) \]

That is we have a \emph{parametrised} family of models

- we first estimate \( \Theta \) based on the data
  - e.g., by maximum likelihood
  - usually involving some penalty function to prevent overfitting
- and then test whether the model is a good fit
  - by analysing the residuals, testing the significance of parameters and so on and so on

This is probably the best known method

- itself known as \emph{frequentist} statistics

having an adjective there suggests there is an alternative, and there is, viz.

- \emph{Bayesian} statistics
Bayesian Statistics

Rather than choosing just one hypothesis
▶ i.e., a specific choice for $\Theta$

We could also define
▶ a probability distribution over all possible hypotheses
▶ i.e., on $\Theta$

And update that distribution
▶ whenever data becomes available
▶ every further experiment gives further insight by repeated updates

The updates are done using the celebrated theorem of Reverend Thomas Bayes

\[
P(H \mid D) = \frac{P(D \mid H) \times P(H)}{P(D)}
\]

If you want to make a prediction
▶ you either choose the (at that moment) most likely model
▶ or you average, i.e., compute the expectation
The Prior

To get this process started
  ▶ we need a probability distribution on $\Theta$ before we have seen any data

This has led to (almost literal) wars between
  ▶ objectivists (frequentists) and subjectivists (Bayesians)

Which was rather silly (according to me)
  ▶ because you choose $f$ a priori as well

If you want to know I am a subjectivist
  ▶ how surprised you are depends on what you already knew

Fortunately, this unsavoury debate died down
  ▶ and most people simply use the technique that is most appropriate to their problem

As far as the prior is concerned
  ▶ if you have no idea, use an uninformative one
  ▶ we’ll discuss an optimal one later
The Scientific Method Revisited

The scientific method works well

▶ in those cases were you can do ceteris paribus experiments

There are, however many cases were this is not possible

▶ because it is immoral
  ▶ fire half the population at random to test your hypothesis how people cope

▶ or downright impossible
  ▶ we only have one history, which is immutable

There are many scientific questions

▶ that actually involve such circumstances

And you could hardly say

▶ research in such problems is forbidden because the scientific method doesn’t condone it
There is, However, Data, Big Data

We may not be able to do experiments
  ▶ but we do have vast amounts of data
    *Observational Data*

You may be morally unable to fire people
  ▶ but many have been, are, or will be fired
  ▶ and we have detailed information on their behaviour and changes therein.

Can’t we use that data
  ▶ to test our hypotheses?

Or even, to *create* our hypotheses?
Supervised Learning

If we want to make predictions, it is easy:

You can split your data in half
   ▶ train your model on the first half
   ▶ and test it on the second half
This, and more sophisticated methods, are
   ▶ firmly based in Statistics
      ▶ i.e., mathematically proven correct

Hence, one can say

   Such use of Big Data is condoned by the Scientific Method

It is science
   ▶ as you know it
But not all science involves (testable) predictions
Unsupervised Learning

Historians don’t usually aim to predict

▶ with the notable exception of Hari Seldon\(^1\)

Rather, they try to organize, e.g., the evolution of ideas into a coherent picture

▶ your scientific forebears never declared themselves to be active in the scientific revolution

This used to be based on painstakingly meticulous

▶ research of source documents in archives, by hand and by eye

With the advent of large digitised archives

▶ such processes can be done much faster, with many more sources for many more topics

▶ topics that are not even known when the process starts

▶ automatically or largely semi-automatically

This involves no predictions

▶ the scientific method does not apply

\(^1\)The fictional originator of psychohistory in Asimov’s Foundation series
An Example from Biology

DNA sequencing machinery is based on
► reads that are 100 - 600 bp long
► longer than that, reliability goes down quickly
  ► there do exists long read methods for 10k bp’s, but they are still very expensive

Sequencing is thus a rather crude process
► you amplify (copy) the DNA string many times
► you blast those DNA strings to smithereens
► sequence those reads
► and stitch the results back to one complete genome

How do you stitch back?
► based on the principle of parsimony
  ► that is, Ockham’s razor by another name
► and, hopefully, protection by redundancy

You give 10 copies of War and Peace to a baby or a dog or ...
► and hope to reconstruct the text from the scraps.
Two Further Examples

It is hard to know which different species of bacteria and/or viruses live
  - in your gut, a patch of land, a given lake, ...

and many of these critters refuse to grown in the lab.
  - Enter: Metagenomics!
  - blast them all to smithereens, all in one go
  - and reconstruct all of their genomes from the debris

If you have many genomes of species sequenced
  - you can try to reconstruct the course of evolution
    \textit{construct the tree (web) of life}

by careful analysis of
  - the differences in their genomes

Both examples again rely on the principle of parsimony
The Scientific Method Does Not Apply

There are no predictions here
  ▶  to some limited extent their might be
    ▶  you might look for bacteria you hypothesize to exist
    ▶  fossils of extinct species you hypothesize
    ▶  undiscovered documents that strengthen your story
  ▶  but what if you don’t find them?

So, we cannot say that for such problems
  ▶  the problem of Induction is kept in check by the Scientific Method

Are we prone to errors then?

is there a well-founded approach?
Algorithmic Information Theory
Algorithmic Information Theory

Founded independently by
- Ray Solomonoff (1960)
- Andrey Kolmogorov (1965)
- Gregory Chaitin (1966)

In the words\(^2\) of Chaitin it is:

"the result of putting Shannon’s information theory and Turing’s computability theory into a cocktail shaker and shaking vigorously."

It encompasses areas such as
- Algorithmic – Kolmogorov – Complexity,
- Algorithmic Probability and Universal Induction,
- and Algorithmic Randomness

For our purposes it is simply

learning by compression

\(^2\) according to Wikipedia
The first basic premise is that all data can be encoded as a string over some finite alphabet, usually this alphabet is taken to be \( \{0, 1\} \). Hence

\[
\text{the data is a (finite) string } x \in \{0, 1\}^* 
\]

This is, perhaps, not a surprising observation for a computer scientist, but it is rather different from the usual view on data, e.g.,

- tables filled with numbers
- relational databases
- graphs
- libraries of texts

We will need some simple facts about this encoding – i.e., standard information theory – later on, which we briefly discuss now.
Coding Theory

We want to store or transmit sequences of elements of a finite set $A = \{a_1, \ldots, a_n\}$ by binary strings

- $A$ is known as the *alphabet*, if we describe our hypotheses in natural language, $A$ would simply be our own well-known alphabet

A *code* is a function

- $C : A \rightarrow \{0, 1\}^*$
- mapping each symbol in the alphabet to its code word

Coding is easily extended to strings of symbols to sequences by concatenation:

- $C : A^* \rightarrow \{0, 1\}^*$
- by $C(xy) = C(x)C(y)$

Note, we require a code ($C : A \rightarrow \{0, 1\}^*$) to be invertible
- otherwise you cannot decode, i.e., recover what the original sequence was
Codes and Trees

A code $C$ defines a binary tree in which each code word $C(a_i)$ denotes a path from the root of the tree to a leaf

- say 0 is branch to the left, 1 is branch to the right
- i.e., you label the edges with 0 and 1
- and the symbols from your alphabet $A$ in the node where their path ends

This tree makes it easy to decode a binary string

- at least when we know when a code word ends and the next one begins
- we could achieve this by a special symbol
  - a comma, added to our 0/1 alphabet or a reserved word
- but we can also simply stipulate that no code word is the prefix of another code word
  - all alphabet symbols are in a leaf node

This is known as a prefix code
Decoding Prefix Codes

If we have a prefix code \( C \)

- decoding a string \( C(x) \) with \( x \in A^* \)
is easy:

  - start at the root
  - if the first bit is 0 go to the left, otherwise go right
  - continue until you hit a leaf: output the symbol in that leaf node and return to the root
Kraft’s Inequality

For prefix codes there is an important inequality for the lengths of the code words $|C(a)|$, i.e., the number of bits used:

$$\sum_{a \in A} 2^{-|C(a)|} \leq 1$$

This inequality provides a link between probability distributions and coding, both in our finite setting and more general in the countable case. For $a \in A$, its probability is given by

$$P(a) = \frac{2^{-|C(a)|}}{\sum_{a \in A} 2^{-|C(a)|}}$$

This relationship also holds in the other direction:

★ but first we prove Kraft
Proving Kraft

If our code does not correspond to a complete binary tree
▶ a tree that splits in two at every internal node
▶ equivalently all leaves of the tree correspond to a symbol in $A$.
we can always extend it so that it is complete
▶ adding some bogus symbols to our alphabet

Using Induction:
▶ Kraft holds for the two leaf tree: both probabilities are $1/2$
▶ let $w$ be a path with length $w$ splitting the node gives us two paths $w_1$ and $w_2$ such that $2^{-|w_1|} + 2^{-|w_2|} = 2^{-|w|}$

In other words, for prefix codes corresponding with complete binary trees equality holds
▶ in all other cases we get an inequality since we remove the probabilities that correspond to the bogus symbols
Codes and Probabilities

We already saw that prefix code words for an alphabet \( A \) define a probability distribution on \( A \) by

\[
P(a) = \frac{2^{-|C(a)|}}{\sum_{a \in A} 2^{-|C(a)|}}
\]

This relation also holds in the other direction

- for every probability distribution on \( A \)
- there is a corresponding prefix code for \( A \)

To prove this we first show that if we have a set of integers \( \{n_1, \ldots, n_k\} \) such that

\[
\sum_{i=1}^{k} 2^{-n_i} \leq 1
\]

Then there is an alphabet \( A = \{a_1, \ldots, a_k\} \) such that

- there is a prefix encoding \( C \) for \( A \)
- such that \( |C(a_i)| = n_i \)
Constructing the Code

Assume that the $n_i$ are ordered by

\[ n_1 \leq n_2 \leq \cdots \leq n_k \]

Take the fully balanced binary tree of depth $n_k$.
   ▶ take the left most path 000..00 till length $n_1$, choose a symbol for that node
   ▶ and cut the rest of the tree below that node
For the other $n_i$ we do the same
   ▶ i.e., take the left-most path that does not end in a labelled leaf node and repeat.

Note that this gives us a relation between all (finite) probability distributions and codes by choosing the integers

\[ n(a) = \left\lfloor \log \left( \frac{1}{P(a)} \right) \right\rfloor \]
Shannon Fano Coding

This simple lemma gives us the promised translation from probability distributions on $A$ to coding $A$ by choosing the integers

$$n(a) = \left\lceil \log \left( \frac{1}{P(a)} \right) \right\rceil$$

This is known as a Shannon Fano coding of $A$. It is optimal in the following sense:

Let $C$ be a prefix code for $A$, with $|C(a_i)| = n_i$ and $P$ a probability distribution on $A$ with $P(a_i) = p_i$. Then

1. $E_P(l) = \sum p_i n_i \geq \sum p_i \log 1/p_i \overset{\text{def}}{=} H(P)
2. $E_P(l) = H(P) \iff \forall i : p_i = 2^{-n_i}$

this is known as Shannon’s noise free coding theorem
Proof

We have:

\[ \sum p_i \log 1/p_i - \sum p_i n_i = \sum p_i \log 1/p_i - \sum p_i \log 2^{n_i} \]

\[ = \sum p_i \log 1/p_i + \sum p_i \log 2^{-n_i} \]

\[ = \sum p_i \log \frac{2^{-n_i}}{p_i} \]

\[ = \log e \sum p_i \ln \frac{2^{-n_i}}{p_i} \]

\[ \leq \log e \left( \sum p_i \frac{2^{-n_i}}{p_i} - 1 \right) \text{ because } \ln x \leq x - 1 \]

\[ = \log e \left( \sum 2^{-n_i} - 1 \right) \leq 0 \]

Note that all our results also hold for countable \( A \).
The Model

Back to *algorithmic* information theory

The second basic premise is that a model is

▶ a program that outputs the data

This may seem surprising, but it isn’t.

▶ firstly notice that all models you have encountered upto now are essentially programs

The deeper motivation is

▶ you probably model the data because you want to do something with that model

▶ not necessarily prediction, perhaps just what-if style reasoning

▶ and since Turing we have the computable function as the embodiment of effective computing

▶ the ultimate collection of things one can do.

That is, in the language of computer science,

*A model of $x$ is a Turing Machine that outputs $x$*
Universal Turing Machines

One of the remarkable facts Turing proved in his 1936 paper is the existence of Universal Turing machines.

A machine $U$ that can emulate any other Turing machine

- given an input string that first specifies the intended $TM$ and then the intended input tape $T$ for $TM$
- it computes the result $TM$ would on $T$

The proof is easiest in the (equivalent) language of partial recursive functions:

Let $\{\phi_i\}_{i \in \mathbb{N}}$ be an enumeration of Gödel numbers of the partial recursive functions. Define $u : \mathbb{N} \times \mathbb{N}$ by

$$u(i, x) = \phi_i(x)$$

$u$ is a partial recursive function,
Too Abstract?

While all of this was very surprising and new in 1936

► it is all completely standard now

You can think of

► the universal Turing machine as your favourite programming language and a compiler for it
► identifying the designated Turing machines by a program in that language

And it is all completely equivalent

The important point is that

► fix some universal Turing machine $U$
► and model $x$ relative to $U$.

Note that

► each program for $U$ is a bitstring in $\{0, 1\}^*$

so, one could say that a model of a bitstring is a bitstring
Universal Induction

If you try to understand your environment
▶ you know you are well under way if you can predict what the environment is going to do next
▶ this is actually rather important for survival

If you make a mistake
▶ you update your current “model” of the environment

Broadly speaking
▶ this is what Solomonoff aimed to formalize

The environment is a (continuous) bitstring
▶ and at each time-point you aim to predict which bit comes next
Updating? That is Bayes

Updating a model? That is what we have Bayes theorem for.

- we have a probability distribution over all possible models
- get a new data point
- and update the distribution with this observation, using

$$P(H|O) = \frac{P(O|H) \times P(H)}{P(O)}$$

So, we have our observed string \(x\)

- and all programs that compute \(x\) (and more)
  - technically: all minimal programs, removing any bits from the end will cause it not to compute \(x\) any more
- a probability distribution on that set
- and we update that distribution with each new observation using Bayes law

But, what distribution do we have?

- that depends on our prior distribution!
A Non-Informative Prior

If we start this process with a given distribution

- Bayesian updating will return a distribution every time

So, if we specify our very first distribution on the models

- we are all set to go

The question is what distribution do we take?

- it is a completely new environment
- we have observed nothing
- so everything is still possible

Hence, we should use a non-informative prior

- a prior distribution that assumes nothing about the environment

An example of a non-informative prior

- is the principle of indifference
- in the finite case: use the uniform distribution, everything is equally likely
Solomonoff’s Universal Prior

Since we want to predict $x$ (or better, how it continues) we define the prior directly for all strings:

$$M(x) = \sum_{p: U(p) = x} 2^{-|p|}$$

Note Kraft’s inequality shows that this sum is $\leq 1$

▶ hence, it is a semi-measure
▶ we should perhaps normalise to sum to 1, but that is not important for us now.

Choosing a non-informative prior, is often not easy

▶ if you re-parametrisise your problem, the priors may suddenly change!

Solomonoff’s prior does not suffer from these problems

▶ it has all the nice properties one could hope for

Moreover, it majorizes all other possibilities

▶ you could say: it assumes the least about the environment of all.
Recall Epicurus?

In the first part, we noticed that Epicurus (300 BC) had the principle of multiple explanations

▷ discard no hypotheses that is consistent with the observations

That is exactly what we are doing here. For, all the $p$ in

$$M(x) = \sum_{p: U(p) = x^*} 2^{-|p|}$$

are still viable hypothesis of the environment

▷ they correctly compute the observations so far, $x$ on the environment

Moreover, after receiving the new bit $b_i$, we only consider the programs $p$ that compute $xb_i^*$

▷ the other half is discarded!

AIT embraces both Epicurus and Ockham (as we will see shortly)
If you look at the expression

\[ M(x) = \sum_{p: U(p) = x^*} 2^{-|p|} \]

you should note that the highest contribution is
► by the shortest program

In fact
► if \( p_1 \) is only 1 bit longer than \( p_2 \)
► its contribution is only half

In other words,
► the shortest program is the most important
Relative Complexity

Let \( x \in \{0, 1\}^* \) be some finite string and let \( U \) be some fixed universal Turing machine.

The *relative complexity* of \( x \) with regard to \( U \) is defined as

\[
K_U(x) = \min\{l(p) \mid U(p) \text{ halts and } U(p) = x\}
\]

That is, \( K_U(x) \) is the length of the *shortest* program that outputs \( x \) on \( U \) and halts.

A string \( x \) is random relative to \( U \) iff

\[
K_U(x) \geq |x|
\]
The Invariance Theorem

Theorem:
Let $U_1$ and $U_2$ be two universal Turing machines, there exists a constant $c_{U_1,U_2}$ such that for all finite $x \in \{0,1\}^*$:

$$K_{U_1}(x) \leq K_{U_2}(x) + c_{U_1,U_2}$$

Proof
Let $p_0$ be a shortest program that makes $U_1$ behave like $U_2$ (i.e., $p_0$ is a cross compiler), $c_{U_1,U_2} = l(p_0)$.

That is, for large $x$ the relative complexity doesn’t really depend on which universal Turing machine is chosen.
Kolmogorov Complexity

Because the (relative) unimportance of the chosen universal Turing machine, we simply write

$$K(x)$$

and talk about the complexity of $x$. A shortest program that computes $x$ is often denoted by $x^*$.

As before $x$ is random iff $K(x) \geq |x|$, but note that this is upto an additive constant – to be non-random $K(x)$ should be a lot smaller than $|x|$

Note, for many mathematical reasons it is often convenient to restrict oneself to so-called prefix or self-delimiting Turing machines.

- no accepted tape is a prefix of another accepted tape

We’ll skip over such niceties here.
Compression

Most strings will be random – there are a lot fewer strings of length \( n \) then there are of length \( 2n \);

- \( 2^{-n} \) to be precise.

Half of the strings do not even have a model

- that is 1 bit shorter

We live in a truly random world

If \( x \) is non-random – i.e., \( K(x) \ll |x| \); \( x^* \) compresses \( x \)

- lossless compression as you can reconstruct \( x \) from \( x^* \)
- simply run \( x^* \) on your reference universal Turing machine
Optimal but Uncomputable

Unfortunately, $K(x)$ is uncomputable

**Proof**

...output $x$ and halts ..., the halting problem is undecidable.

It is upper semi-computable, though

**Proof**

pick your favourite enumeration of Turing machines – programs for your reference universal Turing machine – and dovetail (like enumerating $\mathbb{N}^2$)

- the first step of the first machine
- the second step of the first machine and the first step of the second machine
- the third step of the first machine, the second of the second and the first of the third

Whenever a machine stops after outputting $x$, you can check whether you have a new lowest upper bound for $K(x)$
Uncomputable \neq Useless

Contrary to what you may think,

- Kolmogorov complexity is a very useful concept
- Optimality is a very powerful property
- making uncomputability a minor inconvenience
- Its power is probably best used in complexity theory,
- but an exploration of that would take us too far afield

We’ll discuss a computationally more amenable variant:

MDL
An input string for your favourite UTM $U$ consists – often – of two parts.

- first a part that selects a certain Turing machine
  - the program
- followed by a “random” part that lets that program generate $D$

In such a case, the complexity consists of two parts.

- firstly the complexity of the model (the program)
- secondly the complexity of the data given that model (the data encoded by the model)

This line of reasoning suggest another – related – induction principle.
The Minimum Description Length Principle

Given a set of models $\mathcal{H}$, the best model $H \in \mathcal{H}$ is the one that minimizes

$$L(H) + L(D \mid H)$$

in which

- $L(H)$ is the length, in bits, of the description of $H$, and
- $L(D \mid H)$ is the length, in bits, of the description of the data when encoded with $H$.

Note, this is two-part – or crude – MDL, refined MDL is beyond our scope.

This is my favourite weapon to slay the dragon of induction

- one of the reasons for the name of my chair

With KRIMP as a prototypical example
Krimp
One reason to mine a set of patterns is

- is to understand the data

Note that this is a different goal from

- trying to find *interesting patterns*
- the former is objective, the latter is subjective

To allow us to understand the data well

- the selected set of patterns should collectively describe the data well

In the language of AIT that is

- they should collectively compress the data well.
Central Idea: The Code Table

To use MDL we need models, *code tables* turn sets of item sets into models.

A code table $CT$ is

- a two-column table
  - lefthand-side: itemsets
  - righthand-side: codes from a prefix code
- item sets and codes occur at most once
- If $I$ and $J$ both occur in CT, and $I \subset J$, $J$ occurs before $I$
- All singleton item sets are present.
Coding and Decoding

- to *code* a transaction $t$ with $CT$
  - we *cover* $t$ with itemsets in $CT$
    - pick the first $I \in CT : I \subseteq t$
    - continue covering $t \setminus I$
  - replace the itemsets in the cover by their code
- to *decode* a coded transaction
  - simply replace the codes (prefix!) by their itemsets
- WLOG: codes in $CT$ are Shannon for $D$
## Cover in Action

<table>
<thead>
<tr>
<th>Code Table</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
</tr>
</tbody>
</table>

### Transaction t

- B
- C
- E

**Code Table**

- Itemset A
- Itemset B
- Itemset C
- Itemset D
- Itemset E

**Usage**

- 0
- 0
- 0
- 0
- 0
## Cover in Action

<table>
<thead>
<tr>
<th>Code Table</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itemset</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

Transaction $t$:

- B
- C
- E
### Cover in Action

#### Code Table

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Transaction t

- B
- C
- E

#### Cover of t

- C
- E
## Cover in Action

<table>
<thead>
<tr>
<th><strong>Code Table</strong></th>
<th><strong>Usage</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Itemset</strong></td>
<td><strong>A C</strong></td>
</tr>
<tr>
<td><strong>B D</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td><strong>C E</strong></td>
<td><strong>1</strong></td>
</tr>
<tr>
<td><strong>A</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td><strong>B</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td><strong>C</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td><strong>D</strong></td>
<td><strong>0</strong></td>
</tr>
<tr>
<td><strong>E</strong></td>
<td><strong>0</strong></td>
</tr>
</tbody>
</table>

**Transaction t**

- **Cover of t**
  - **C E B**
The Code Table

\[ I = \{ A, B, C, D, E \} \]

**Code Table**

<table>
<thead>
<tr>
<th>Itemset</th>
<th>Code</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A C</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>B D</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>C E</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Length and Size

- let $P(c)$ denote the probability that $c$ occurs in the cover of a random $t \in db$ (computed from usage statistics)
- The length of the code for $c$: $l(c) = -\log(P(c))$
- The left-hand side of $CT$ is encoded with the standard code table (singletons only)
- The size of a transaction is the sum of length of the codes in its cover
- The size of the encoded database is simply the sum of the sizes of the encoded tuples
Did You Notice a Bias?

In computing the size of the model

- we take the data into account

Some people don’t like this

- for the right-hand column it is a no-brainer – if want optimal compression of the data, this is the code to use
- i.e., if we didn’t force it before hand, it are the codes the algorithm would have to detect anyway.
- phrased differently: it are the optimal parameters

But what about the left-hand column?

- remember, coding is ad hoc
- with this choice we bias to larger – and hopefully fewer – patterns
The Total Size

- The total size of an encoding of $db$ by a code table $CT$ is given by:

$$\mathcal{L}(CT, db) = L_{db}(CT) + L_{CT}(db)$$

- Note, we disregard Cover as it is identical for all $CT$ and $db$ and thus would only add a constant.

- Notation: $CT(db) = \mathcal{L}(CT, db)$
The Optimal Code Table

**Task:** is to find the optimal code table

**Problem:**
- huge search space (exponential in the number of frequent item sets)
- without usable structure.

**Solution:** use a heuristic: the Krimp algorithm
To find good code tables, observe that

- If $I \subset J$ and $I$ is before $J$ in $CT$, then $\text{usage}(J) = 0$.
- If $I_1$ and $I_2$ have the same size, and $\text{supp}(I_1) > \text{supp}(I_2)$, $I_1$ covers a larger area than $I_2$.

Heuristic: order the elements of the code table, descendingly:

1. first on size
2. then on support
Many many patterns

**Krimp**

- Database
- Many many patterns

**MDL**

- **select pattern**
- **accept / reject**
- **add to code table**
- **compress database**

**Code table**
Order and Pruning

- **Krimp** starts with the simplest code table: containing singletons only.
- The order in which candidates are tested is code table order reversed.
- If we add a new itemset to $CT$, the usage of earlier elements may go down.
- That is, they do more harm than good:
  - prune such elements from $CT$ to see if the compression becomes better.
  - note: singletons stay in $CT$ (but are not counted if not used).
Wine

Minimum support

# patterns

Wine

Frequent itemsets

Picked by Krimp
Experiments
Reductions of upto $10^7$, only one in 10.000.000 is chosen!

*Are you impressed?*
Discussion

Reductions of upto $10^7$, only one in 10,000,000 is chosen!

Are you impressed?

I hope not!

- far simpler algorithms can do that
- are these results significant?

We’ll look at two ways to “prove” this.
Swap Randomisation

Doing, randomly, as many swaps as there are 1’s in the data set:

```
1         0
|         |
|         |
0         1
```

```
0         1
|         |
|         |
1         0
```

we generate a random data set with the same row and column sums. Generating a 1000 of such data sets allows us to do a significance test.
$p = 0$
Characteristic?

The reduction achieved by KRIMP is significant in the space of all data sets with the same row and column sums. Does this mean that the code table characterises the data distribution well?

- The patterns are characteristic if, e.g.,
  - Different distributions get different code tables
  - Different code tables mean different distributions
- This is what classification is all about!
Classification with **KRIMP**

- Database *(n classes)*
- Split per class
- **Apply** KRIMP
- Code table per class
- Encode unseen transactions
- **Shortest code wins!**

**Note with Laplace correction!**
Classification Example

CT1

CT2

Transaction 1

Transaction 2

CT2
## Classification Results

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Baseline</th>
<th>Krimp</th>
<th>NB</th>
<th>C4.5</th>
<th>CBA</th>
<th>HRM</th>
<th>iCAEP</th>
<th>SVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adult</td>
<td>76.1</td>
<td>84.3</td>
<td>80.2</td>
<td><strong>85.5</strong></td>
<td>84.2</td>
<td>81.9</td>
<td>80.9</td>
<td>84.7</td>
</tr>
<tr>
<td>Anneal</td>
<td>76.2</td>
<td>96.6</td>
<td>96.3</td>
<td><strong>97.8</strong></td>
<td>94.7</td>
<td>95.1</td>
<td>95.3</td>
<td></td>
</tr>
<tr>
<td>Breast</td>
<td>65.5</td>
<td>94.1</td>
<td>93.3</td>
<td>94.1</td>
<td>94.0</td>
<td></td>
<td><strong>97.4</strong></td>
<td>93.7</td>
</tr>
<tr>
<td>Chess (k–k)</td>
<td>52.2</td>
<td>90.0</td>
<td>87.6</td>
<td><strong>99.4</strong></td>
<td>72.8</td>
<td></td>
<td>94.6</td>
<td>93.9</td>
</tr>
<tr>
<td>Chess (kr–k)</td>
<td>16.2</td>
<td>57.9</td>
<td>35.9</td>
<td><strong>78.5</strong></td>
<td>25.8a</td>
<td>44.9</td>
<td></td>
<td>46.3</td>
</tr>
<tr>
<td>Connect–4</td>
<td>65.8</td>
<td>69.4</td>
<td>67.9</td>
<td><strong>80.1</strong></td>
<td>68.7</td>
<td>68.1</td>
<td>69.9</td>
<td>77.6</td>
</tr>
<tr>
<td>Heart</td>
<td>54.1</td>
<td>61.7</td>
<td>55.1</td>
<td>54.8</td>
<td>57.3</td>
<td></td>
<td><strong>80.3</strong></td>
<td>58.4</td>
</tr>
<tr>
<td>Ionosphere</td>
<td>64.1</td>
<td><strong>91.0</strong></td>
<td>90.9</td>
<td>90.9</td>
<td>87.2b</td>
<td></td>
<td>90.6</td>
<td>90.9</td>
</tr>
<tr>
<td>Iris</td>
<td>33.3</td>
<td><strong>96.0</strong></td>
<td>94.7</td>
<td>94.0</td>
<td>94.0</td>
<td>94.7</td>
<td>93.3</td>
<td><strong>94.0</strong></td>
</tr>
<tr>
<td>Led7</td>
<td>11.0</td>
<td>75.3</td>
<td>75.4</td>
<td>75.3</td>
<td>66.6</td>
<td>74.6</td>
<td></td>
<td><strong>75.8</strong></td>
</tr>
<tr>
<td>Letter</td>
<td>4.1</td>
<td>70.9</td>
<td>57.2</td>
<td><strong>77.5</strong></td>
<td>28.6</td>
<td>76.8</td>
<td></td>
<td>69.8</td>
</tr>
<tr>
<td>Mushroom</td>
<td>51.8</td>
<td><strong>100</strong></td>
<td>94.0</td>
<td><strong>100</strong></td>
<td>46.4</td>
<td>99.9</td>
<td>99.8</td>
<td>99.9</td>
</tr>
<tr>
<td>Nursery</td>
<td>33.3</td>
<td>92.3</td>
<td>92.2</td>
<td><strong>99.5</strong></td>
<td>90.1</td>
<td>92.8</td>
<td>84.7</td>
<td>97.6</td>
</tr>
<tr>
<td>Page blocks</td>
<td>89.8</td>
<td><strong>92.6</strong></td>
<td>91.8</td>
<td>92.5</td>
<td>90.9</td>
<td>91.6</td>
<td></td>
<td>92.2</td>
</tr>
<tr>
<td>Pen digits</td>
<td>10.4</td>
<td>95.0</td>
<td>84.2</td>
<td>95.6</td>
<td>87.4</td>
<td>96.2</td>
<td></td>
<td><strong>96.6</strong></td>
</tr>
<tr>
<td>Pima</td>
<td>65.1</td>
<td>72.7</td>
<td>73.8</td>
<td>72.7</td>
<td><strong>75.0</strong></td>
<td>73.0</td>
<td>72.3</td>
<td>74.0</td>
</tr>
<tr>
<td>Tic–tac–toe</td>
<td>65.3</td>
<td>88.7</td>
<td>68.8</td>
<td>93.3</td>
<td><strong>100</strong></td>
<td>81.0</td>
<td>92.1</td>
<td>87.9</td>
</tr>
<tr>
<td>Waveform</td>
<td>33.9</td>
<td>77.1</td>
<td>77.4</td>
<td>74.1</td>
<td>77.6</td>
<td>80.5</td>
<td><strong>81.7</strong></td>
<td>80.1</td>
</tr>
<tr>
<td>Wine</td>
<td>39.9</td>
<td><strong>100</strong></td>
<td>95.5</td>
<td>96.6</td>
<td>53.2</td>
<td>63.0</td>
<td>98.9</td>
<td>97.2</td>
</tr>
<tr>
<td>Average</td>
<td>47.8</td>
<td>84.5</td>
<td>79.6</td>
<td>87.0</td>
<td>75.4</td>
<td></td>
<td></td>
<td><strong>84.5</strong></td>
</tr>
</tbody>
</table>
Why Does Classification Work?

Our motivation was the following:

\[
    l_{CT}(t) = \sum_{c \in \text{cover}(t)} l_{CT}(c) = \sum_{c \in \text{cover}(t)} -\log (P(c | db))
    
    = -\log \left( \prod_{c \in \text{cover}(t)} P(c | db) \right) = -\log (P(t | db))
\]

Clearly, the last equation assumes a Naive Bayes like independence assumption, which is \textit{not} true. Disregarding this, we get the Bayes optimal procedure:

\[
    l_{CT_1}(t) \leq l_{CT_2}(t) \rightarrow P(t | db_1) \geq P(t | db_2)
\]
So, Why Does it Work

The occurrence of item sets in the code table is not independent because of the order in the table.

- however, when

\[ P(c_1 \mid db) \times P(c_2 \mid db) < P(c_1 \cup c_2 \mid db) \]

it becomes favourable to add \( c_1 \cup c_2 \) to the code table

- and if it is added, it will be above \( c_1 \) and \( c_2 \)

Hence, problems may only occur for those cases where:

\[ P(c_1 \mid db) \times P(c_2 \mid db) > P(c_1 \cup c_2 \mid db) \]

But this means that \( c_1 \cup c_2 \) doesn’t occur very often in \( db \), and thus presumably also not in transactions that need to be classified.

- Moreover, like with naive Bayes, we do not have to be right with our estimates of \( P(t \mid db_1) \) and \( P(t \mid db_2) \) as long as we rank them right.
Efficiency

**Krimp** is a rather wasteful algorithm
- we first compute all frequent item sets and then we throw almost all of them away
Moreover, it is wasteful in the sense that we consider each item set only once
- if $I$ doesn’t help compression straight away, that doesn’t mean that it would be a very good addition after $J$ has been added to the code table.

It turns out that one can remedy both problems at the same time. This is done in the **SLIM** algorithm (Smets and Vreeken, SDM, 2012)
The First Observation

If adding $I$ to the code table doesn’t help compression it is highly unlikely that adding $I \cup J$ to the same code table will do so.

Hence we can change our search as follows

- at each step try adding each combination of two elements of the code table
- select the combination that improves the compression most

As before code table elements that don’t contribute are removed.

Note that this remedies both problems.
The Second Observation

Compressing the database over and over again is rather costly.

- But, we can estimate the gain of adding an element to the code table.

Calculate the gain from the usage counts of code pairs

- disregarding the effect on other elements

Use branch-and-bound to find the pair with the highest estimated gain.

These two observations turn \textbf{Krimp} into \textbf{Slim}.
Better compression
comparing results after at most computing 1 day

- High difference → mine at lower minsup threshold
- Impossible to mine all of those, need only a few good ones

![Bar chart showing difference in relative compression (ΔL%) for various datasets.](image-url)