Algorithmic Information Theory

prof. dr Arno Siebes

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

Preface

If your interest is piqued by this lecture, there are two good books you could read

- An Introduction to Kolmogorov Complexity and Its Applications, 3rd edition. Ming Li and Paul Vitányi
- The Minimum Description Length Principle. Peter Grünwald

Less comprehensive, but very well written, is the article

- A Philosophical Treatise of Universal Induction, by Samuel Rathmanner and Marcus Hutter in Entropy, 2011

And, of course, there is plenty of material on the web

- use your Google-Fu

An Embarrassing Problem

At the end of the previous lecture we noticed

- if you mine with a high minimal support, you'll get a few – often well-known – itemsets
- if you mine with low support

  you may end up with more itemsets than you have transactions!

The reason is simple:

- one transaction can/will support multiple itemsets

Somehow this doesn't feel like a solution to Big Data

- one of the goals of data analysis is to get insight in the data
  - the data is usually too big to give you that insight by inspecting the data directly
- an important aspect of that is that you can inspect the models and/or patterns you mine
  - and that is not going to happen if you end up with more rather than less.

A Pattern Set

Given that we end up with more itemsets than transactions, the problem is clear

the resulting set of itemsets is highly redundant

That is

- many itemsets have more or less the same support set

So, we shouldn't look at individual itemsets

- although there are many approaches that do exactly that
- as some of you will see in the course Pattern Set Mining

but at sets of itemsets

- rather than finding all (frequent) itemsets
- we want to find *the best set of frequent itemsets*

What Makes a Set of Itemsets Good?

If we want to find the best set of itemsets,

- we first should determine what makes one set of itemsets better than another
- in other words: what makes a set of itemsets good?
  - what is the quality of a set of itemsets on a dataset D?

Clearly, the redundancy in the set should play an important role

- but, how do you define that exactly?
- and how do choose among itemsets?

There is no single answer

- see Pattern Set Mining

This week we look at one approach

- the Utrecht approach

I am not only shamelessly plugging my next course

- but also my own research

But, then again, it is the best approach, of course.

Key Insight 1

The first observation is that

- this is very much an *unsupervised learning* problem

The individual transactions contain no information whatsoever

- to let you pit one set of itemsets against another

Only the complete dataset can help.

- that is, there is no relation with classification whatsoever.

The fact that we are doing unsupervised learning

- with no relationship to classification

means that we cannot use the theory of induction we have learned to love in this course

- PAC learning is useless for this problem

We need an alternative Theory of Induction

Key Insight 2

If we prune the complete set of frequent itemsets to remove redundancy

- we shouldn't prune too much

The resulting set should still

- cover the data

But just covering the data is not enough

- you can do that with just the singletons

The set of itemsets should be *descriptive*

- the itemsets should be *characteristic* for the data

The set of itemsets should collectively describe the data

- or, even better, the underlying data distribution
- the distribution the data has been sampled from

We are looking for

A small, descriptive, set of characteristic itemsets

Come Together

Together the two key insights tell us that we are looking for

> *an inductive theory that rates a set of itemsets on how descriptive the set is of the underlying distribution.*

And, fortunately, such a theory exists, it is known as

- Algorithmic Information Theory

It is a very computational view on induction

- which is a nice bonus for computer scientists

Algorithmic Information Theory

Founded independently by

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

In the words[1] 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*

[1] according to Wikipedia

The Data

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

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

An algorithmic information theorist sees only finite strings of symbols

The Model

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 and then halts*

Turing Machine (Wikipedia)

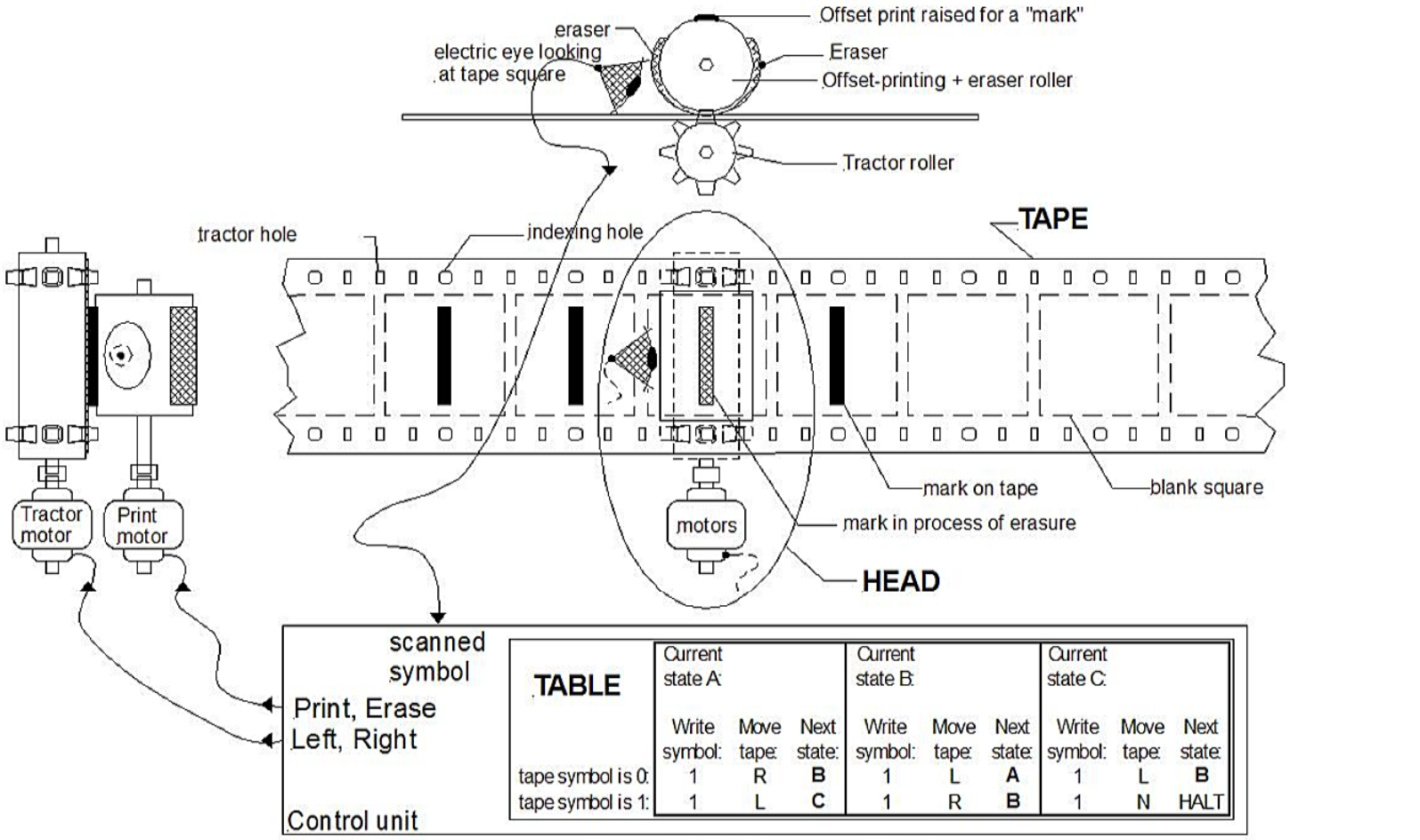
A Turing machine is defined as a 7-tuple:

$$M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$$

- Q is a finite, non-empty set of states
- Γ is a finite, non-empty set of tape alphabet symbols
- $b \in \Gamma$ is the blank symbol, the only one that is allowed to occur infinitely often on the tape
- $\Sigma \subseteq \Gamma \setminus \{b\}$, is the set of input symbols
- $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a partial function called the transition function, where L is left shift, R is right shift.
  - δ is simply a look-up table
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final or accepting states.

The initial tape contents is said to be accepted by M if it (at the "end of the tape") halts in a state from F

Turing Machine: The Picture (thanks you Wikipedia)



A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.

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*

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

What is *The* Model

Given our discussion so far, it shouldn't come as a surprise that

x^* *is the model of* x

That x^* is a model of x is something we already discussed

- but why do we select the shortest one?

There are at least three ways to motivate this

- by Ockham's razor
- by compression
- by Universal Induction

We'll briefly discuss each of these

Note that there may be more programs with the same length, if so, we simply choose the first in some arbitrary order.

By Ockham's Razor

We already met William of Occam (1287 - 1347) and his principle of simplicity:

- Numquam ponenda est pluralitas sine necessitate
- Plurality must never be posited without necessity

Among all the bitstrings (programs) that compute x

- the shortest is surely the simplest

So, Ockham's razor

- which is not completely uncontroversial, but very often applied
- see, e.g., Ockham's Razors – A User's Manual by Elliott Sober

we are completely justified to choose the shortest program.

By Compression

We already noted that if x is non-random – i.e., $K(x) \ll |x|$; x^* compresses x

- and lossless compression since you can reconstruct x from x^*

Compression is based on regularity

- the more regularity you detect, the better you can compress

Clearly,

- x^* is the ultimate compression of x

That is

- x^* exploits all regularity in x

Which is a different way of stating

- x^* is the best possible model of x

By 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

So, slightly different from before

- your model produces x and keeps producing
- it produces $x*$ (notice the difference with x^*)

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

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

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 that we are using Kraft's inequality again

- 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 not easy

- if you reparametrize 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 Introduction, 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

From Solomonoff back to Kolmogorov

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

Hence, also from this point of view

> *Kolmogorov's choice for the shortest program is the most sensible choice.*

Our New Induction Principle

In the beginning we observed

- that we needed a new induction principle
- to select the best set of frequent itemsets

We now have that principle

- by Kolmogorov complexity
- choose the program that compresses D best
  - the smallest program that computes D and halts

The remaining question seems to be

- how do we compute $K(D)$

The somewhat disappointing answer is

- we can't!

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 out putting 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 briefly look at two other examples

- Chatin's Ω
  - an application in theory
- the normalised information distance
  - an application in practice

Mind Boggling

Fix again a universal (prefix) Turing machine U and define

$$\Omega_U = \sum_{p \text{ halts}} 2^{-l(p)}$$

Thanks to Kraft, $\Omega_U \in [0, 1]$

- it is uncomputable
  - otherwise the halting problem would be decidable
- it is uncompressible
  - the first n bits of Ω_U let you decide halting for programs of upto n bits. That cannot be computed with fewer than n bits
- it is a normal number in any base b
  - for all b^k possible k-digit sequences the limiting frequency of the b-ary expansion on Ω_U is $1/b^k$
  - why? otherwise Ω_U would be compressible

Meta Math!: The Quest for Omega, Gregory Chaitin (popscience, but good)

Chaitin's Incompleteness Result

A Formal Axiomatic System can be seen as a program

- generating all its theorems

Since Ω_U is incompressible

- (upto a constant) no FAS can determine more bits of Ω_U
- than its own complexity $K(FAS)$

That is

- essentially, the only way to prove theorems of the form

  $$\textit{the 23574 bit of } \Omega_U \textit{ is 0}$$

  is to add them as an axiom

One could say (and Chaitin does)

- that math is random
- since the bits of Ω_U are.

Math Random?

Remember Goldbach's Conjecture

- every even integer > 2 is the sum of two prime numbers
  - Uncle Petros and Goldbach's Conjecture, Apostolos Doxiadis.
  - should you simply be a good mathematician or go for glory and possibly fail?
  - Doxiadis also co-wrote: Logicomix
  - together with Christos Papadimitriou (complexity theory)
  - who also wrote a novel: Turing (a Novel about Computation)

It is easy to write a verifier for that conjecture

- which halts iff Goldbach is false

Ω_U is an Oracle that can tell us

- whether or not this program halts
- that is: whether or not Goldbach is true

And Ω_U is random

The Normalised Information Distance

The normalised information distance between two strings x and y is defined by:

$$d(x,y) = \frac{\max\{K(x \mid y^*), K(y \mid x^*)\}}{\max\{K(x), K(y)\}}$$

where $K(x \mid y^*)$

- is the shortest program that computes x given y^*
  - not just y but (also) the shortest program that computes it

Since $K(x)$ is uncomputable

- so is $d(x,y)$

So, why bother?

- it inspires a measure we can compute
- using your favourite compression algorithm

The Normalised Compression Distance

The normalised compression distance between two strings x and y is defined by:

$$d(x,y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}$$

where

- xy is simply the concatenation of x and y
- $C(x)$ is the compressed size of x
  - with C your favourite compression algorithm

It can be proved that for compression algorithms

- with suitable properties

this is actually a distance measure

Experimentally, it works pretty well. Cilibrasi and Vitányi, Clustering by Compression, IEEE transactions on information theory

Clustering Animals

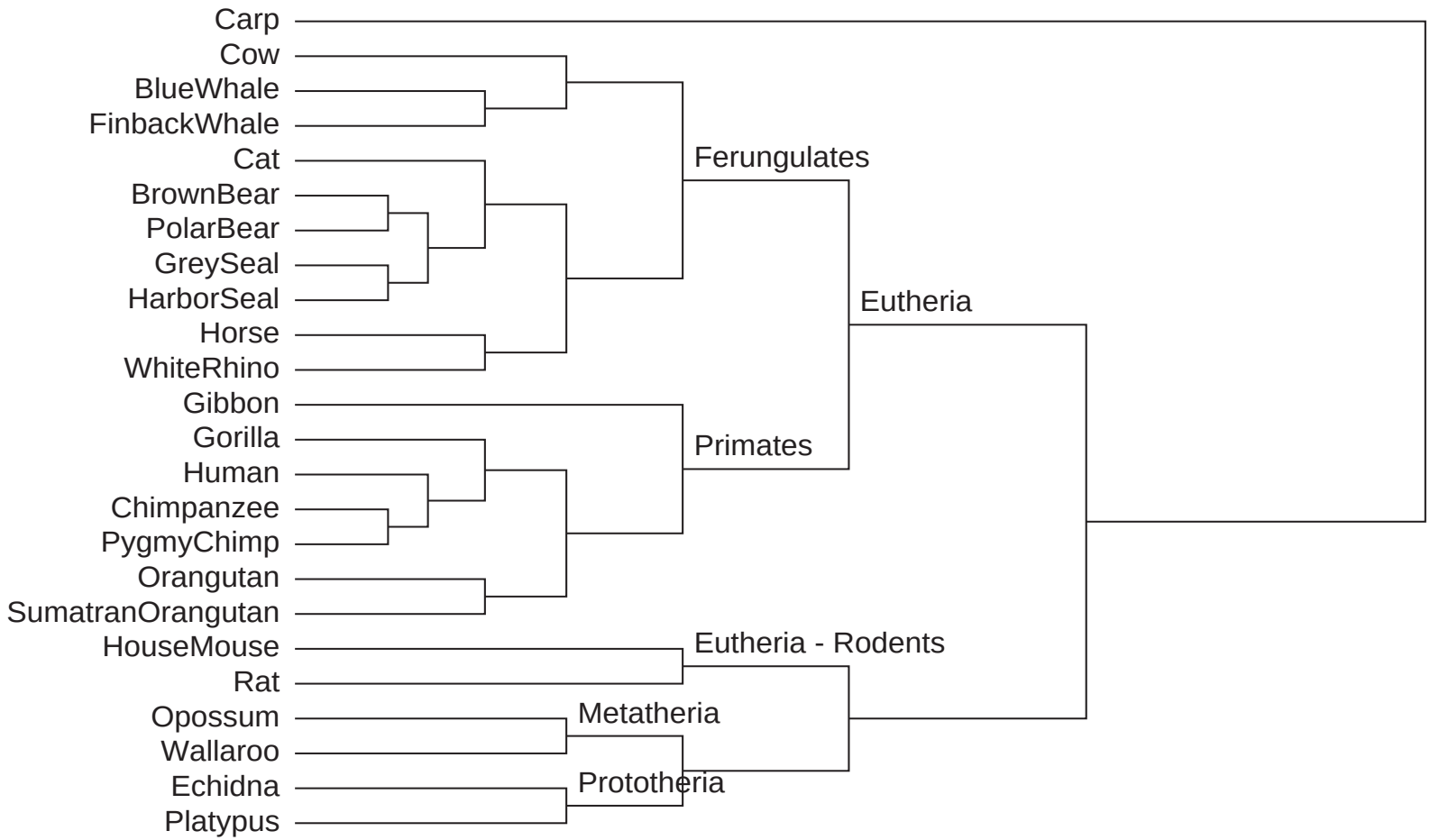


Fig. 7. The evolutionary tree built from complete mammalian mtDNA sequences of 24 species, using the NCD matrix of Figure 9. We have redrawn

That is Nice, But

(Did I already tell you not to use such expressions?)

I hope I have convinced you so far that

- AIT is a good induction principle
- at least in theory
- and sometimes in practice

The $64000 question is, of course,

- does it help us to find the best set of itemsets

And then the answer is

- not based on the theory we have so far
- but with some further work, we'll have a match

Programs as Models

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.

Again, we choose the minimum which can be motivated

- from Bayes
- and from AIT (although, this is very much part of AIT)

From Bayes to MDL

Bayes tells us that

$$\mathbb{P}(H \mid D) = \frac{\mathbb{P}(D \mid H) \times \mathbb{P}(H)}{\mathbb{P}(D)}$$

Clearly, we want the H that maximises $\mathbb{P}(H \mid D)$ and since $\mathbb{P}(D)$ is the same for all models, we have to maximize

$$\mathbb{P}(D \mid H) \times \mathbb{P}(H)$$

Or, equivalently, *minimize*

$$-\log\left(\mathbb{P}(H)\right) - \log\left(\mathbb{P}(D \mid H)\right)$$

Shannon tells us that the $-\log$ transform takes us from probabilities to *codes*, i.e., we minimize

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

for some encoding – probability distribution – for H and $D \mid H$

Solomonoff, Of Course! Not

If we want to do MDL in the suggested way – i.e., being a Bayesian – we need to specify, probability distributions – or coding schemes. This is very much the problem we already discussed

- almost the reason why Solomonoff came up with his distribution

But, the answer is

- No, we know that that distribution is uncomputable
- and we are going to be practical now

That is, we are going to make

- an ad-hoc choice
  - life isn't a bed of roses
  - we can't have nice theorems all the time

So, What Distribution to Use?

Again, if we want to do MDL in the suggested way – i.e., being a Bayesian – we need to specify

- both a prior probability $\mathbb{P}(H)$ on the models
- and $\mathbb{P}(D \mid H)$

What are reasonable choices? We could, e.g., use

- some uninformative prior on $\mathcal{H}$ and an maximum entropy measure for $\mathbb{P}(D \mid H)$

But these are not necessary easy to compute and/or query and ad hoc

If we are going to be ad hoc,

- we will explicitly encode both the models and data given a model

The Second Motivation

The second motivation is

- to a large extent a formalisation of our original derivation
- using a (generating) program part and a tape of random numbers part

Showing that MDL

- is in the core of AIT

The complete story is, however, rather technical

- we'll keep it intuitive here.

Objects and Sets

Recall that in Algorithmic Information Theory we are looking for (optimal) descriptions of objects.

One way to describe an object is

- describe a set of which it is a member
- point out which of the members it is.

In fact, it is what we do all the time:

- (the) beach (i.e., the set of all beaches)
- over there (pointing out a specific one)

Algorithmic Statistics

We have, a set S

- which we call a *model*
- and has complexity $K(S)$

And an object $x \in S$

- S is a model of x
- and the complexity of pointing x out in S is
- the complexity of x given S, i.e., $K(x \mid S)$

And obviously:

$$K(x) \leq K(S) + K(x \mid S)$$

So?

A premise of of Algorithmic Information Theory is that

- a program P that outputs x and halts encodes the *information* in x
- the/a smallest such program encodes *exactly* the information in x

If x is a *data set*, i.e., a *random sample* from some distribution, we expect that x has

- "true" (epistemic) structure; captured by S
- accidental (aleatoric) structure; captured by $x \mid S$

Hence, we minimize

$$K(S) + K(x \mid S)$$

which is akin to two-part MDL

- $K(S)$ is the model code
- $K(x \mid S)$ is the data to model code

More on the Complexities

We can actually say a bit more about these two complexities:

For $K(S)$

- as usual in AIT, this is simply the length of the shortest program that outputs S and halts
- i.e., it is a *generative* model of x

For $K(x \mid S)$:

- if x is a typical element of S
- the only way to find x in S is using an index, i.e.,

$$K(x \mid S) \approx \log(|S|)$$

That is, we have to minimize

$$K(S) + \log(|S|)$$

Kolmogorov's Structure Function

This formulation of MDL suggest a way to discover the best model.

Define Kolmogorov's structure function as

$$h_x(i) = \min_S \{\log(|S|) \mid x \in S, K(S) \leq i\}$$

That is, we start with very simple – in terms of complexity – models and gradually work our way up.

Alternatively we can define the MDL function

$$\lambda_x(i) = \min_S \{K(S) + \log(|S|) \mid x \in S, K(S) \leq i\}$$

Again we try to find the minimum by using ever more complex models.

Caveat Emptor

There are two critical remarks to make here.

Firstly, it is not a given that x a typical element of S is, i.e., that

$$K(x \mid S) \approx \log(|S|)$$

holds. It is almost the other way around,

- one should check that this is right to know that one has a good model!

It is tempting to say that one should simply look for

- the best model by every increasing complexity

but

> *although it may be true that the maximal compression yields the best solution, it may still not be true that every incremental compression brings us closer to the solution*

Approximation of the Two-Part MDL Code, Pieter Adriaans and Paul Vitányi, IEEE transactions on information theory

In Conclusion

Today we discovered

- that we need to find the best set of itemsets

And that for that

- we needed an alternative theory of induction

We found that in MDL

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

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

And discussed why this is a reasonable theory of Induction

Next time we'll show how that principle can be used

- to approximate our goal

With the most beautiful and elegant KRIMP algorithm