Information Theory

http://www.inf.ed.ac.uk/teaching/courses/it/

Week 1
Introduction to Information Theory

Iain Murray, 2010

School of Informatics, University of Edinburgh

Course structure

Constituents:

— ~17 lectures
— Tutorials starting in week 3
— 1 assignment (20% marks)

Website:

`http://tinyurl.com/itmsc`
`http://www.inf.ed.ac.uk/teaching/courses/it/`

Notes, assignments, tutorial material, news (optional RSS feed)

Prerequisites: some maths, some programming ability

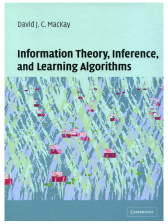
Maths background: This is a theoretical course so some general mathematical ability is essential. Be very familiar with logarithms, mathematical notation (such as sums) and some calculus.

Probabilities are used extensively: Random variables; expectation; Bernoulli, Binomial and Gaussian distributions; joint and conditional probabilities. There will be some review, but expect to work hard if you don't have the background.

Programming background: by the end of the course you are expected to be able to implement algorithms involving probability distributions over many variables. However, I am not going to teach you a programming language. I can discuss programming issues in the tutorials. I won't mark code, only its output, so you are free to pick a language. Pick one that's quick and easy to use.

The scope of this course is to understand the applicability and properties of methods. Programming will be exploratory: slow, high-level but clear code is fine. We will not be writing the final optimized code to sit on a hard-disk controller!

Resources / Acknowledgements



Recommended course text book

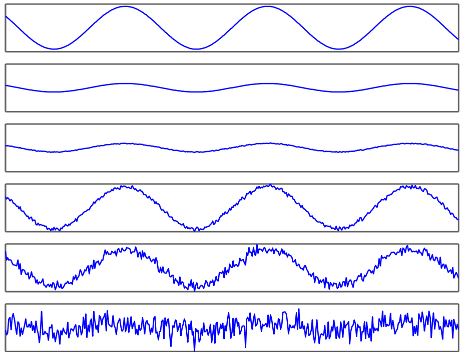
Inexpensive for a hardback textbook
(Stocked in Blackwells, Amazon currently cheaper)

Also free online:
http://www.inference.phy.cam.ac.uk/mackay/itila/

Those preferring a theorem-lemma style book could check out:
Elements of information theory, Cover and Thomas

I made use of course notes by MacKay and from CSC310 at the University of Toronto (Radford Neal, 2004; Sam Roweis, 2006)

Communicating with noise



Signal
Attenuate
Add noise
Boost
5 cycles
100 cycles

Consider sending an audio signal by *amplitude modulation*: the desired speaker-cone position is the height of the signal. The figure shows an encoding of a pure tone.

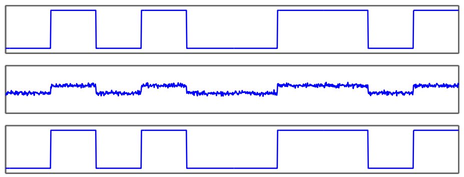
A classical problem with this type of communication channel is attenuation: the amplitude of the signal decays over time. (The details of this in a real system could be messy.) Assuming we could regularly boost the signal, we would also amplify any noise that has been added to the signal. After several cycles of attenuation, noise addition and amplification, corruption can be severe.

A variety of analogue encodings are possible, but whatever is used, no 'boosting' process can ever return a corrupted signal exactly to its original form. In digital communication the sent message comes from a discrete set. If the message is corrupted we can 'round' to the nearest discrete message. It is possible, but not guaranteed, we'll restore the message to exactly the one sent.

Digital communication

Encoding: amplitude modulation not only choice.
Can re-represent messages to improve signal-to-noise ratio

Digital encodings: signal takes on discrete values



Signal
Corrupted
Recovered

Communication channels

modem → phone line → modem

Galileo → radio waves → Earth

parent cell → daughter cells

computer memory → disk drive → computer memory

Real channels are error prone.

Physical solutions:
change system to reduce probability of error

System solution

message
↓
encoded message
↓
corrupted encoding
↓
decoded message

Rather than cooling a system, or increasing power,
we send more robust encodings over the existing channel

But how is reliable communication possible at all?

Repetition codes

Repetition code R_3:

`hello there` $\rightarrow$ `hhheeelllllllooo`   `ttthhheeerrreee`

$\downarrow$ (noise added)

`hello thfr?` $\leftarrow$ `hhkgeeslllllqooc m qttzhhfferrrBme`

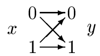
Possible results of errors:

— Corrected
— Detected
— Undetected

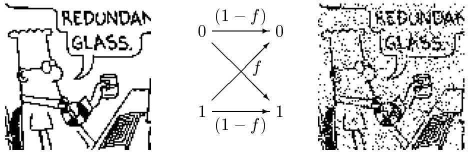
Binary symmetric channel

Binary messages: 0010100111001...

Each 0 or 1 is flipped with probability $f = 0.1$



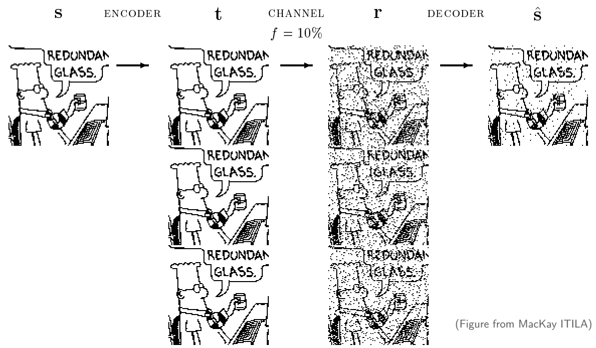
$$P(y=0\,|\,x=0) = 1-f; \quad P(y=0\,|\,x=1) = f;$$
$$P(y=1\,|\,x=0) = f; \quad P(y=1\,|\,x=1) = 1-f.$$



(Figure from MacKay ITILA)

Can repetition codes give reliable communication?

Repetition code performance



(Figure from MacKay ITILA)

Probability of error per bit ≈ 0.03. What's good enough?

Consider a single 0 transmitted using R_3 as 000

Eight possible messages could be received:
000 100 010 001 110 101 011 111

Majority vote decodes the first four correctly but the next four result in errors. Fortunately the first four are more probable than the rest!

Probability of 111 is small: $f^3 = 0.1^3 = 10^{-3}$
Probability of two bit errors is $3f^2(1-f) = 0.03 \times 0.9$
Total probability of error is a bit less than 3%

How to reduce probability of error further? Repeat more! (N times)

Probability of bit error = Probability $>$ half of bits are flipped:

$$p_b = \sum_{r=\frac{N+1}{2}}^{N} \binom{N}{r} f^r (1-f)^{N-r}$$

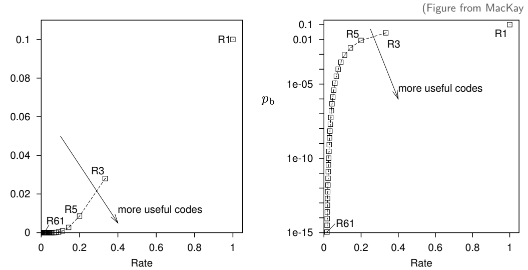
But transmit symbols N times slower! *Rate* is $1/N$.

Repetition code performance

Binary messages: 0010100111001...

Each 0 or 1 is flipped with probability $f = 0.1$

(Figure from MacKay ITILA)



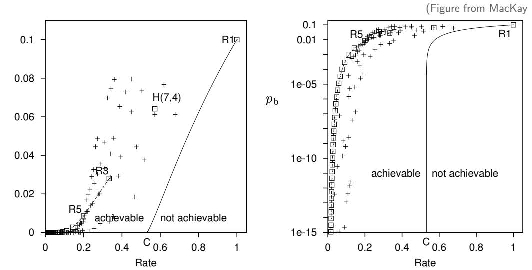
p_b = probability of error at each bit of message

What is achievable?

Binary messages: 0010100111001...

Each 0 or 1 is flipped with probability $f = 0.1$

(Figure from MacKay ITILA)



p_b = probability of error at each bit of message

Course content

Theoretical content

— Shannon's noisy channel and source coding theorems
— Much of the theory is non-constructive
— However bounds are useful and approachable

Practical coding algorithms

— Reliable communication
— Compression

Tools and related material

— Probabilistic modelling and machine learning

Storage capacity

3 decimal digits allow $10^3 = 1{,}000$ numbers: 000–999

3 **binary digits or bits** allow $2^3 = 8$ numbers:
000, 001, 010, 011, 100, 101, 110, 111

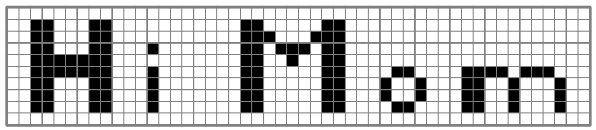
8 bits, a 'byte', can store one of $2^8 = 256$ characters

Indexing I items requires at least
$\log_{10} I$ decimal digits or $\log_2 I$ bits

Reminder: $b = \log_2 I \;\Rightarrow\; 2^b = I \;\Rightarrow\; b \log 2 = \log I \;\Rightarrow\; b = \frac{\log I}{\log 2}$

Representing data / coding

Example: a 10×50 binary image



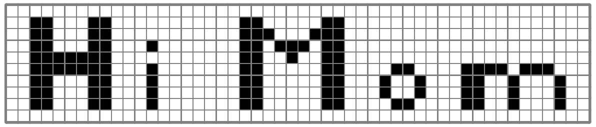
Assume image dimensions are known

Pixels could be represented with 1s and 0s

This encoding takes **500 bits** (binary digits)

2^{500} images can be encoded. The universe is $\approx 2^{98}$ picoseconds old.

Exploit sparseness



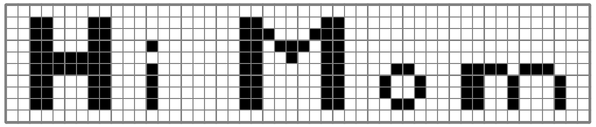
As there are fewer black pixels we send just them.
Encode row + start/end column for each run in binary.

Requires $(4{+}6{+}6){=}16$ bits per run (can you see why?)
There are 54 black runs ⇒ 54×16 = **864 bits**

That's worse than the 500 bit encoding we started with!

Scan columns instead: 33 runs, $(6{+}4{+}4){=}14$ bits each. **462 bits**.

Run-length encoding

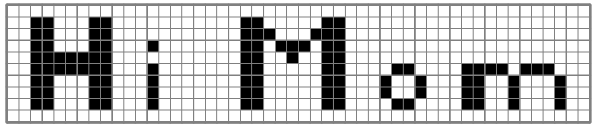


Common idea: store lengths of runs of pixels

Longest possible run = 500 pixels, need 9 bits for run length
Use 1 bit to store colour of first run (should we?)

Scanning along rows: 109 runs ⇒ **982 bits**(!)
Scanning along cols: 67 runs ⇒ **604 bits**

Adapting run-length encoding

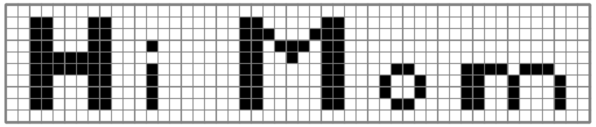


Store number of bits actually needed for runs in a header.
$4{+}4{=}8$ bits give sizes needed for black and white runs.

Scanning along rows: **501 bits** (includes 8+1=9 header bits)
55 white runs up to 52 long, 55×6 = 330 bits
54 black runs up to 7 long, 54×3 = 162 bits

Scanning along cols: **249 bits**
34 white runs up to 72 long, 24×7 = 168 bits
33 black runs up to 8 long, 24×3 = 72 bits (3 bits/run if no zero-length runs; we did need the first-run-colour header bit!)

Rectangles

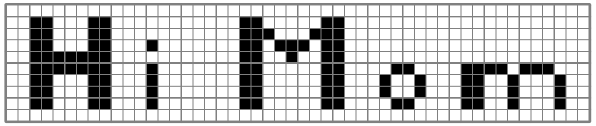


Exploit spatial structure: represent image as 20 rectangles

Version 1:
Each rectangle: (x_1, y_1, x_2, y_2), 4+6+4+6 = 20 bits
Total size: 20×20 = **400 bits**

Version 2:
Header for max rectangle size: 2+3 = 5 bits
Each rectangle: (x_1, y_1, w, h), 4+6+3+3 = 16 bits
Total size: 20×16 + 5 = **325 bits**

Off-the-shelf solutions?



Established image compressors:
Use PNG: 128 bytes = **1024 bits**
Use GIF: 98 bytes = **784 bits**

Unfair: image is tiny, file format overhead: headers, image dims

Smallest possible GIF file is about 35 bytes. Smallest possible PNG file is about 67 bytes.
Not strictly meaningful, but: (98-35)×8 = 504 bits. (128-67)×8 = 488 bits

"Overfitting"

We can compress the '`Hi Mom`' image down to 1 bit:

Represent '`Hi Mom`' image with a single '1'

All other files encoded with '0' and a naive encoding of the image.

. . . the actual message is one selected from a set of possible messages. The system must be designed to operate for each possible selection, not just the one which will actually be chosen since this is unknown at the time of design.

— Shannon, 1948

Summary of lecture 1 (slide 1/2)

Digital communication can work reliably over a noisy channel. We add *redundancy* to a message, so that we can infer what corruption occured and undo it.

Repetition codes simply repeat each message symbol N times. A majority vote at the receiving end removes errors unless more than half of the repetitions were corrupted. Increasing N reduces the error rate, but the *rate* of the code is $1/N$: transmission is slower, or more storage space is used. For the Binary Symmetric Channel the error probability is: $\sum_{r=(N+1)/2}^{N} \binom{N}{r} f^r (1-f)^{N-r}$

Amazing claim: it is possible to get arbitrarily small errors at a fixed rate known as the *capacity* of the channel. *Aside:* codes that reach the capacity send a more complicated message than simple repetitions. Inferring what corruptions must have occurred (occurred with overwhelmingly high probability) is more complex than a majority vote. The algorithms are related to how some groups perform inference in machine learning.

Summary of lecture 1 (slide 2/2)

First task: represent data optimally when there is no noise

Representing files as (binary) numbers:

C bits (binary digits) can index $I = 2^C$ objects.

$\log I = C \log 2$, $C = \frac{\log I}{\log 2}$ for logs of any base, $C = \log_2 I$

In information theory textbooks "log" often means "$\log_2$".

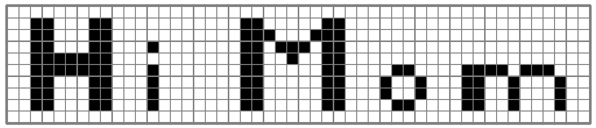
Experiences with the `Hi Mom` image:

Unless we're careful we can expand the file dramatically
When developing a fancy method always have simple baselines in mind
We'd also like some more principled ways to proceed.

Summarizing groups of bits (rectangles, runs, etc.) can lead to fewer objects to index. Structure in the image allows compression.

Cheating: add whole image as a "word" in our dictionary.
Schemes should work on future data that the receiver hasn't seen.

Where now



What are the fundamental limits to compression?
Can we avoid all the hackery?
Or at least make it clearer how to proceed?

This course: Shannon's information theory relates compression to *probabilistic modelling*

A simple probabilistic model (predict from three previous neighbouring pixels) and an *arithmetic coder* can compress to about **220 bits**.

Why is compression possible?

Try to compress *all* b bit files to $<b$ bits

There are 2^b possible files but only (2^b-1) codewords

Theorem: if we compress some files we must expand others
(or fail to represent some files unambiguously)

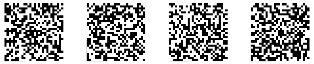
Search for the `comp.compression` FAQ currently available at:
`http://www.faqs.org/faqs/compression-faq/`

Which files to compress?

We choose to compress the **more probable** files

Example: compress 28×28 binary images like this:

At the expense of longer encodings for files like this:



There are 2^{784} binary images. I think $<2^{125}$ are like the digits

Sparse file model

Long binary vector $\mathbf{x}$, mainly zeros

Assume bits drawn independently

Bernoulli distribution, a single "bent coin" flip

$$P(x_i\,|\,p) = \begin{cases} p & \text{if } x_i\!=\!1 \\ (1-p)\equiv p_0 & \text{if } x_i\!=\!0 \end{cases}$$

How would we compress a large file for $p\!=\!0.1$?

Idea: encode blocks of N bits at a time

Intuitions:

'Blocks' of lengths $N\!=\!1$ give naive encoding: 1 bit / symbol
Blocks of lengths $N\!=\!2$ aren't going to help
…*maybe* we want long blocks

For large N, some blocks won't appear in the file, e.g. `11111111111`…
The receiver won't know exactly which blocks will be used
Don't want a header listing blocks: expensive for large N.

Instead we use our probabilistic model of the source to guide which blocks will be useful. For $N\!=\!5$ the 6 most probable blocks are:

`00000 00001 00010 00100 01000 10000`

3 bits can encode these as 0–5 in binary: `000 001 010 011 100 101`

Use spare codewords (`110 111`) followed by 4 more bits to encode remaining blocks. Expected length of this code $= 3 + 4\,P(\text{need 4 more})$
$= 3 + 4(1-(1-p)^5 - 5p(1-p)^4) \approx 3.3$ bits $\Rightarrow 3.3/5 \approx 0.67$ bits/symbol

Quick quiz

Q1. Toss a fair coin 20 times. (Block of $N\!=\!20$, $p\!=\!0.5$)
What's the probability of all heads?

Q2. What's the probability of '`TTHTTHHTTTHTTHTHHTTT`'?

Q3. What's the probability of 7 heads and 13 tails?

you'll be waiting forever	**A**	$\approx 10^{-100}$
about one in a million	**B**	$\approx 10^{-6}$
about one in ten	**C**	$\approx 10^{-1}$
about a half	**D**	≈ 0.5
very probable	**E**	$\approx 1-10^{-6}$
don't know	**Z**	???

Binomial distribution

How many 1's will be in our block?

Binomial distribution, the sum of N Bernoulli outcomes

$$k = \textstyle\sum_{n=1}^N x_n, \quad x_n \sim \text{Bernoulli}(p)$$

$$\Rightarrow\;\; k \sim \text{Binomial}(\text{N}, \text{p})$$

$$P(k\,|\,N,p) = \binom{N}{k} p^k (1-p)^{N-k}$$

$$= \frac{N!}{(N-k)!\,k!}\, p^k (1-p)^{N-k}$$

Reviewed by MacKay, p1

Evaluating the numbers

$$\binom{N}{k} = \frac{N!}{(N\!-\!k)!\,k!},$$

what happens for $N\!=\!1000$, $k\!=\!500$?
(or $N\!=\!10{,}000$, $k\!=\!5{,}000$)

Knee-jerk reaction: try taking logs

Explicit summation: $\log x! = \sum_{n=2}^{x} \log n$

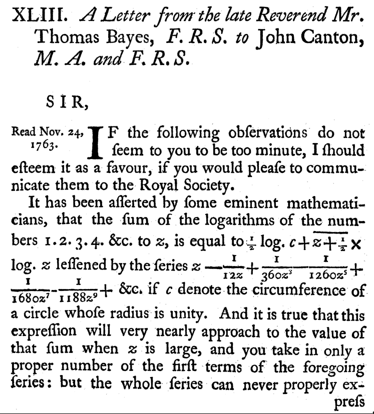
Library routines: $\ln x! = \ln\Gamma(x+1)$, e.g. `gammaln`

Stirling's approx: $\ln x! \approx x\ln x - x + \frac{1}{2}\ln 2\pi x \;\; \ldots$

Care: Stirling's series gets *less* accurate if you add lots terms(!),
but it is pretty good for large x with just the terms shown.

See also: more specialist routines. Matlab/Octave: `binopdf`, `nchoosek`

Philosophical Transactions (1683-1775) Vol. 53, (1763), pp. 269–271.
The Royal Society. `http://www.jstor.org/stable/105732`



XLIII. *A Letter from the late Reverend Mr.* Thomas Bayes, *F. R. S. to* John Canton, *M. A. and F. R. S.*

SIR,

Read Nov. 24, 1763.

IF the following obſervations do not ſeem to you to be too minute, I ſhould eſteem it as a favour, if you would pleaſe to communicate them to the Royal Society.

It has been aſſerted by ſome eminent mathematicians, that the ſum of the logarithms of the numbers 1.2.3.4. &c. to z, is equal to $\frac{1}{2}$ log. $c + \overline{z+\frac{1}{2}}\times$ log. z leſſened by the ſeries $z - \frac{1}{12z} + \frac{1}{360z^3} - \frac{1}{1260z^5} + \frac{1}{1680z^7} - \frac{1}{1188z^9} +$ &c. if c denote the circumference of a circle whoſe radius is unity. And it is true that this expreſſion will very nearly approach to the value of that ſum when z is large, and you take in only a proper number of the firſt terms of the foregoing ſeries: but the whole ſeries can never properly expreſs …

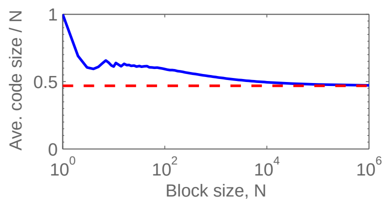
Compression for N-bit blocks

Strategy:

— Encode N-bit blocks with $\leq t$ ones with $C_1(t)$ bits.
— Use remaining codewords followed by $C_2(t)$ bits for other blocks.

Set $C_1(t)$ and $C_2(t)$ to minimum values required.

Set t to minimize average length: $C_1(t) + P\big(t < \sum_{n=1}^N x_n\big)\, C_2(t)$



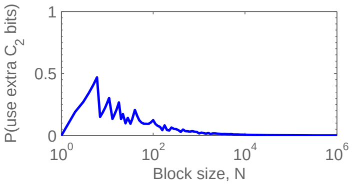
Can we do better?

We took a simple, greedy strategy:

Assume one code-length C_1, add another C_2 bits if that doesn't work.

First observation for large N:

The first C_1 bits index almost every block we will see.



With high probability we can compress a large-N block into a fixed number of bits. Empirically $\approx 0.47\,N$ for $p=0.1$.

Can we do better?

We took a simple, greedy strategy:

Assume one code-length C_1, add another C_2 bits if that doesn't work.

Second observation for large N:

Trying to use $< C_1$ bits means we *always* use more bits

At $N = 10^6$, trying to use 0.95 the optimal C_1 initial bits $\Rightarrow P(\text{need more bits}) \approx 1 - 10^{-100}$

It is very unlikely a file can be compressed into fewer bits.

Summary of lecture 2 (slide 1/2)

If some files are shrunk others must grow:

\# files length b bits $= 2^b$
\# files $< b$ bits $= \sum_{c=0}^{b-1} 2^c = 1 + 2 + 4 + 8 + \cdots + 2^{b-1} = 2^b - 1$

(We'll see that things are even worse for encoding blocks in a stream.
Consider using bit strings up to length 2 to index symbols:
`A=0, B=1, C=00, D=01, E=11`
If you receive `111`, what was sent? `BBB`, `BE`, `EB`?)

We temporarily focus on sparse binary files:

Encode blocks of N bits, $\mathbf{x}$ = `00010000001000...000`

Assume model: $P(\mathbf{x}) = p^k\,(1-p)^{N-k}$, where $k = \sum_i x_i =$ "# 1's"

Key idea: give short encoding to most probable blocks:
Most probable block has $k=0$. Next N most probable blocks have $k=1$
Let's encode all blocks with $k \leq t$, for some threshold t.
This set has $I_1 = \sum_{k=0}^{t} \binom{N}{k}$ items. Can index with $C_1 = \lceil \log_2 I_1 \rceil$ bits.

Summary of lecture 2 (slide 2/2)

Can make a lossless compression scheme:
Actually transmit $C_1 = \lceil \log_2 (I_1 + 1) \rceil$ bits
Spare code word(s) are used to signal C_2 more bits should be read, where $C_2 \leq N$ can index the other blocks with $k > t$.
Expected/average code length $= C_1 + P(k > t)\,C_2$

Empirical results for large block-lengths N
— The best codes (best t, C_1, C_2) had code length $\approx 0.47N$
— these had tiny $P(k > t)$; it doesn't matter how we encode $k > t$
— Setting $C_1 = 0.95 \times 0.47N$ made $P(k > t) \approx 1$

$\approx 0.47N$ bits are sufficient and necessary to encode long blocks (with our model, $p=0.1$) almost all the time and on average

No scheme can compress binary variables with $p=0.1$ into less than 0.47 bits on average, or we could condradict the above result.

Other schemes will be more practical (they'd better be!) and will be closer to the $0.47N$ limit for small N.

H/W: a weighing problem

Find 1 odd ball out of 12

You have a two-pan balance with three outputs:
"left-pan heavier", "right-pan heavier", or "pans equal"

How many weighings do you need to find the odd ball *and* decide whether it is heavier or lighter?

Unclear? See p66 of MacKay's book, but do not look at his answer until you have had a serious attempt to solve it.

Are you sure your answer is right? Can you prove it?
Can you prove it without an extensive search of the solution space?