**Information Theory and Applications**

**Homework part 1**

**Report**

Daniele Genta, s276922

*Run the programs by using in matlab: appdesigner(‘filename.mlapp’).*

**Exercise – 1**

The first step of the exercise is the gather the user’s input when the “calculate” button is pressed.

Note that the pdf f is acquired using the **str2func** method that permits to define a function handler. The variable inserted as input is used to specify the rv’s that characterize the pdf.

a = app.aEditField.Value;

b = app.bEditField.Value;

try

f = str2func(strcat('@(',app.VariableEditField.Value,')',app.pdfEditField.Value));

catch

check\_ok = false;

warning('Not able to convert the specified PDF');

end

The sanity checks implemented are mainly present in the app design. a, b are in fact numeric textfields so that it’s mandatory to insert numerical values (-Inf, Inf included). In addition, a try/catch clause is inserted when trying to convert the pdf into Matlab function handler and a supplementary check makes sure that a <= b.

Moreover, the output textfields are not editable by the user in order to guide the usage of the program.

The second step of the program is the normalization of the pdf, as the assignment specifies this is done via computing the normalization factor and using it to divide the pdf. The normalization factor is the integral between a, b of the pdf.

% 2: normalization factor for the pdf

normalization\_factor = int(sym(app.pdfEditField.Value), [a,b]);

app.NormalizationfactorEditField.Value = char(normalization\_factor);

f\_normalized = f/normalization\_factor;

note: the sym function is used in order to symbolically interpret a function.

The next step is the [differential entropy](https://en.wikipedia.org/wiki/Differential_entropy) calculus. This is achieved implementing the following snippet of code. It’s important to point out that the entropy for continuous RV’s may be not positive.

expr = f\_normalized \* log2(f\_normalized);

hX = -int(sym(expr), [a,b]);

hX\_ = double(hX);

app.hXEditField.Value = hX\_;

drawnow;

Finally, the gaussian UB is calculated for the differential entropy.

From theoretical background we know:

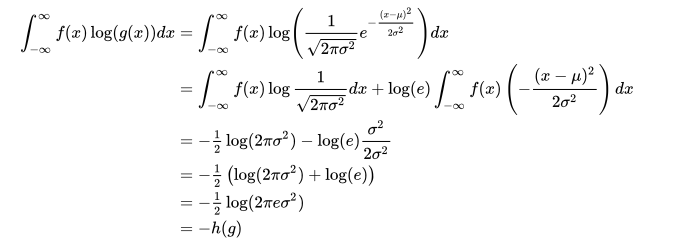
****

Figure 1: source wikipedia

We know that h(X) <= h(g) and we can compute the latter via the following formula:

**h(X) ≤ h(XG) = log2(2πe σX2 )**

Implemented in matlab using this piece of code:

% 4- Gaussian UB

% UB = h(g) = integral(f(x) \* log(g(x)) dx

% shortcut formula from slides

expr =@(x) x \* f\_normalized;

mu = int(sym(expr), [a,b]);

mu\_ = double(mu);

expr =@(x) x^2 \* f\_normalized;

mu2 = int(sym(expr), [a,b]);

mu2\_ = double(mu2);

varian = mu2\_ - (mu\_^2);

hg = 1/2 \* log2(2\*pi\*exp(1)\*varian);

app.UpperboundEditField.Value = hg;

**Exercise – 2**

The first step of the exercise is to gather the user’s input when the “Compute” button is pressed.

X = str2num(app.RangeXEditField.Value);

Y = str2num(app.RangeYEditField.Value);

Z = str2num(app.RangeZEditField.Value);

px = str2num(app.ProbDistrXEditField.Value);

py = str2num(app.ProbDistrYEditField.Value);

pz = str2num(app.ProbDistrZEditField.Value);

fxyz = str2func( ['@(x,y,z) ', app.fxyzEditField.Value ]);

As for the previous exercise the sanity checks are a mix of app design implementations and control via source code. In particular i check that the probability distribution for each discrete RV (X, Y, Z) is consistent, that means that the length of the RV vector must be equal the length of its probability distribution.

*Example for RV X*:

% 2-sanity checks

% a- probability distributions consistent with rv's

check\_ok = true;

if (numel(X) ~= numel(px))

setError(app, 'X range is not consistent with X probability distr');

drawnow;

check\_ok = false;

end

If the sanity checks are passed i normalize the probability distributions (i want their sum to be one). This is performed by the following snippet of code:

% normalization factors

normalization\_factor\_px = sum(px);

normalization\_factor\_py = sum(py);

normalization\_factor\_pz = sum(pz);

% 3-normalized probability vectors

px = px/normalization\_factor\_px;

py = py/normalization\_factor\_py;

pz = pz/normalization\_factor\_pz;

Now i compute the entropies for each RV’s and the function f(X,Y,Z).

% 4- calculate entropies

hx = H(app, px);

hy = H(app, py);

hz = H(app, pz);

In order to compute the entropy of the function f(X, Y, Z) firstly I calculate the “range” of the function f(X, Y, Z), in order to do that I explore all the possible combinations of X, Y, Z and I create a vector with every possible f(X, Y, Z):

syms f(x,y,z)

f(x,y,z) = fxyz;

range\_f\_xyz = zeros(1, numel(X)\*numel(Y)\*numel(Z));

cnt = 1;

for i = 1:numel(X)

for j = 1:numel(Y)

for w = 1:numel(Z)

aus = (f(X(i), Y(j), Z(w)));

range\_f\_xyz(cnt) = aus;

cnt = cnt+1;

end

end

end

Then I count the occurrences of each element of the vector:

uv = unique(range\_f\_xyz);

n = histc(range\_f\_xyz,uv); % frequency of each element

Finally I compute the probability vector for f(X, Y, Z) starting from the occurrences vector, using this I can calculate the entropy of the function:

pxyz = n/sum(n); % total probability of each unique element

hxyz = H(app,pxyz);

The simple function used to compute the entropies is the same developed at lessons that accepts as input a probability distribution.

function h = H(app,p)

p(p==0) = [];

h = -sum(p.\*log2(p));

end

The final step of the pipeline is to compute all relevant inequalities, these include the following:

* H(X) >= 0, where X is a discrete RV
* H(X) <= log2(N), where N is the cardinality of the alphabet of RV X
* H(X, Y, Z) <= H(f(X, Y, Z))

Each of the above inequalities is computed for each of the 3 RV’s and the result (“Verified” or not) is displayed in proper textfields, not editable by the user.

**Note**: Since the 3 discrete RV’s are independent i did not consider the following inequalities, which would have resulted in equalities.

* H(X1) <= H(X1, X2) = H(X2, X1), where X1, X2 are 2 discrete RV’s
* H(X1, X2) <= H(X1) + H(X2) , where X1, X2 are 2 discrete RV’s
* H(X1|X2) = H(X1) – I(X1; X2) <= H(X1)

H(X2)≥0

**Exercise – 3**

Theoretical source:

https://en.wikipedia.org/wiki/Arithmetic\_coding

Inputs of the program:

* Alphabet expressed as a character range (ex. “abcde”);
* Probability vector (p), maps the above defined alphabet to a probability distribution, that may be unnormalized. In order to normalize it i compute the sum of the probability vector (s) and i divide **p/s**;
* Arbitrary string to encode (ex. “abba”).

**Important notes:**

* i did consider the **whitespace** in my program that is not explicitly specified in the alphabet yet is assigned with the last probability in the probability vector (that must therefore be the same length of the alphabet +1)
* renormalization: the precision I set at each instance of the program is equal to the length of the string to encode
* when I choose the final interval to encode, the value I’m encoding is the mean of the interval

Methodology:

Firstly, i retrieve the user’s input, applying the routine sanity checks and normalizing the probability vector.

%1: retrieve user's input

% check is alphabet is unique

alphabet = lower(app.AlphabetEditField.Value);

alphabet = unique(alphabet);

try

p\_vec = str2num(app.ProbabilityvectorEditField.Value);

catch

setError(app,'ERROR: conversion of probability vector');

end

str\_to\_encode = lower(app.StringtoEncodeEditField.Value);

%2: check if probability vector is normalized, if not normalize it

if sum(p\_vec) ~= 1

s = sum(p\_vec);

p\_vec = p\_vec/s; %normalized pvec

end

Specifically, the sanity checks include:

* the check that the alphabet maps the string to encode (each char of the string to be encoded must be in the alphabet)
* the control that the length of the probability vector is equal to the alphabet length +1 (each char in the alphabet must be mapped into a probability, the whitespace is assigned as the last probability in the probability vector).
* making both the alphabet and the string to encode lowercase

%3: sanity checks

% 1-alphabet should map string to encode

check\_ok = true;

unique\_str\_to\_encode = unique(str\_to\_encode);

for i = 1:numel(unique\_str\_to\_encode)

if ~contains(alphabet, unique\_str\_to\_encode)

check\_ok = false;

setError(app,'ERROR: alphabet does not map string to encode');

end

end

% 2-probability vector size must be equal to alphabet size

if numel(p\_vec) ~= length(alphabet)+1

check\_ok = false;

setError(app,'ERROR: probability vector is not consistent with alphabet');

end

Secondly i apply the arithmetic encoding, which is a technique of entropy encoding used in lossless data compression. The key idea behind it is to encode an arbitrary string is to use an arbitrary precision function q (0 <= q <= 1) by exploiting the probability distribution of each character. (this overcomes the single input symbol encoding used in other algorithms).



Initially we have the space [0,1) and we want to map our alphabet over it using its probability distribution.

Suppose that we have an Alphabet = “ABC” with normalized probability distribution: P(A) = 0.2, P(B) = 0.5 and P(C) = 0.3. We will partition the interval into N = cardinality(Alphabet) sub-intervals that will be as follow: int\_A: [0 0.2), int\_B: [0.2 0.7), int\_C = (0.7 1).

In order to implement this first part of the problem in Matlab i’ve used a dictionary (Map.Containers) that has each char in the alphabet as key and the corresponding interval as value.

The second iterative step is to narrow our space [0, 1) into one of the sub-intervals corresponding to the char we’re encoding (suppose that we are encoding B, the space will become [0,2 0,7]) and calculate the new probabilities for each element in the alphabet as the original probability multiplied by the width of the chosen interval.

Repeating the above step until the sequence has been encoded we’ll have as a result a final interval [L,U). By definition in order to encode the entire sequence we can choose whatever element in the interval we like, in my case i used (U+L)/2. This floating number (that lives between 0 and 1) is our arithmetic coding and the last step is to binarize it.

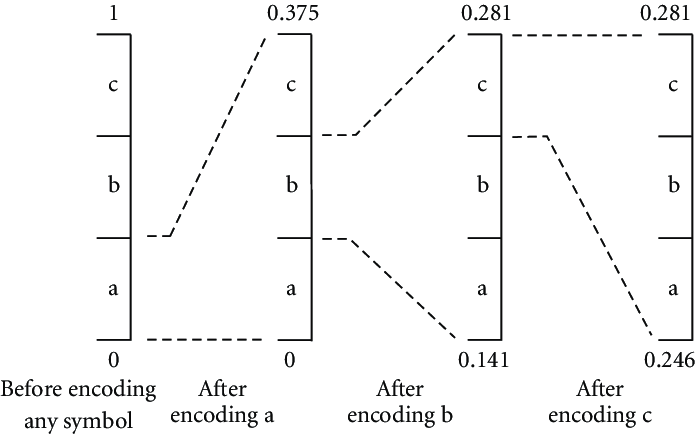


Figure 2: Example of arithmetic encoding

The routine used in order to binarize the resulting floating number is the following.

function Tag\_bits = float2bin(app,tag, l, digits1)

format long g;

% Embedding the Final Tag Value.

Tag\_bits=zeros(1,0);

bits=zeros(1,0);

if(2\*tag>1)

tag=2\*tag-1;

bits=[bits,'1'];

else

tag=2\*tag;

bits=[bits,'0'];

end

str = sprintf('text2int("%s",2)', bits);

s = (evalin(symengine, str));

digits(digits1)

my\_l = vpa(s/2^length(bits));

while my\_l < l

if(2\*tag>1)

tag=2\*tag-1;

bits=[bits,'1'];

else

tag=2\*tag;

bits=[bits,'0'];

end

%upd

str = sprintf('text2int("%s",2)', bits);

s = evalin(symengine, str);

digits(digits1)

my\_l = vpa(s/2^length(bits));

end

Tag\_bits=[Tag\_bits,bits];

end

**Note on renormalization:**

I tried to apply the renormalization as stated on: <https://en.wikipedia.org/wiki/Arithmetic_coding#Precision_and_renormalization>

But i was not able to make it fully workout (i attach the code in the resources folder), so in order to prevent underflow and reach a result i use a dynamical precision equal to the length of the string to encode, the sintax is the following:

digits(numel(str\_to\_encode))

my\_value\_precise = vpa(my\_value);