Information Theory for Data Science

Pouria Mohammadalipourahari s327015

Reihaneh Kharazmi s328016

Arezoo Parsian s327860

Armi Okshtuni s328780

Prof. Taricco

3 December 2023

1. Basic notions

- Entropy of the following distributions

1.1

Noramalisation

$P(n) \cdot e^{-\frac{1}{n}}$ n = 0,1,2,…

First of all, there is a need to normalise this function,To normalize a probability distribution the sum of all probabilities should equal 1, In this case we have a distribution with values $e^{-\lambda n}$ for n = 0,1,2,… the normalisation condition is given by the sum of all probabilities:

$$\sum_{n=0}^{\infty} e^{-\lambda n} = 1$$

This is a geometric series, and it converges to a finite value when $\sum_{n=0}^{\infty} e^{-\lambda n} = \frac{1}{1 - e^{-\lambda}}$. The sum of the series is given by:

$$\sum_{n=0}^{\infty} e^{-\lambda n} = \frac{1}{1 - e^{-\lambda}}$$

To normalize the distribution,we divide each term in the series by this sum:

$$P(n) = \frac{e^{-\lambda n}}{\sum_{n=0}^{\infty} e^{-\lambda n}}$$

Calculating Entropy:

As this distribution is a discrete distribution, the entropy of this type of distribution is:

$$P(n) = \frac{e^{-\lambda n}}{\sum_{n=0}^{\infty} e^{-\lambda n}}$$

And in this case, P(n) given by $H(x) = -\sum_{n=0}^{\infty} P(n) \cdot \log(P(n))$

Substitute the expression for P(n) into the entropy formula:

$$H(x) = -\sum_{k=0}^{\infty} \left(\frac{e^{-\lambda n}}{\sum_{k=0}^{\infty} e^{-\lambda k}} \right) \cdot \log_2 \left(\frac{e^{-\lambda n}}{\sum_{k=0}^{\infty} e^{-\lambda k}} \right)$$

Now Simplify the expression step by step:

The Logarithmic term $\log_2 \left(\frac{e^{-\lambda n}}{\sum_{k=0}^{\infty} e^{-\lambda k}} \right)$ can be expanded using properties of logarithms:

$$\log_2 \left(\frac{e^{-\lambda n}}{\sum_{k=0}^{\infty} e^{-\lambda k}} \right) = -\lambda n - \log_2 \left(\sum_{k=0}^{\infty} e^{-\lambda k} \right)$$

Now, substitute this back into the entropy formula:

$$H(x) = \sum_{k=0}^{\infty} \left(\frac{e^{-\lambda n}}{\sum_{k=0}^{\infty} e^{-\lambda n}} \right) (\lambda n) + \log_2 \left(\sum_{k=0}^{\infty} e^{-\lambda n} \right)$$

Since the sum in the denominator is a geometric series, it can be evaluated to a closed form:

$$\sum_{k=0}^{\infty} e^{-\lambda k} = \frac{1}{1 - e^{-\lambda}}$$

Now, Factor Out constants:

$$H(x) = \frac{\lambda}{1 - e^{-\lambda}} \sum_{n=0}^{\infty} n \cdot e^{-\lambda n} + \frac{1}{1 - e^{-\lambda}} \sum_{n=0}^{\infty} e^{-\lambda n} \cdot \log_2 \left(\frac{1}{1 - e^{-\lambda}} \right)$$

Both summations are standard results:

$$\sum_{n=0}^{\infty} n \cdot e^{-\lambda n} \cdot \frac{e^{-\lambda}}{(1 - e^{-\lambda})^2}$$

$$\sum_{n=0}^{\infty} e^{-\lambda n} = \frac{1}{(1 - e^{-\lambda})^2}$$

Now substitute these back into the expression:

$$H(x) = \frac{\lambda}{(1 - e^{-\lambda})^2} + \frac{1}{1 - e^{-\lambda}} \log_2 \left(\frac{1}{1 - e^{-\lambda}} \right)$$

Which is the Close Form
By using Logarithm Properties it could be simplified to:

$$H(x) = \frac{\lambda}{(1 - e^{-\lambda})^2} - \frac{1}{1 - e^{-\lambda}} \log_2 \left(1 - e^{-\lambda} \right)$$

1.2

Calculating the entropy of this Distribution by analytical method is not straightforward because of the logarithm unit in the normalizer, so it is preferable to use a numerical method to solve this question.

Normalisation

First of all, there is a need to normalise this function.To normalise a probability distribution the sum of all probabilities should equal 1. In this case we have a distribution with values e^{-n^2} for n = 0,1,2,...
The normalisation condition is given by:

$$Z = \sum_{n=0}^{\infty} e^{-n^2}$$

So Now, The Probability mass function after substituting normalizer is

$$P(n) = \frac{e^{-n^2}}{\sum_{n=0}^{\infty} e^{-n^2}}$$

For being sure about the Normalizer,We'll use specific N values of the PMF for checking that the sum is equal to 1.

Description about the code:

First of all we imported numpy which is the mathematical package, this is used for calculating exponential values and generating sequences.
After that we defined a function that calculated the pmf of the given distribution.

In this function, first we calculate the values of e^{-n^2}, Then calculate the value of the normalizer which is $Z = \sum_{n=0}^{\infty} e^{-n^2}$ and name it as Z_N. finally, calculate all of the equation by $e^{-n^2} / Z = \sum_{n=0}^{\infty} e^{-n^2}$.

After that we chose n = 1 to 10 , and N = 5 for being sure that the sum of this pmf equals one.

$$P(n) = \frac{e^{-n^2}}{\sum_{n=0}^{N} e^{-n^2}}$$

And it is shown that in output, the sum is equal to one, so we can be sure about the normalizer.

n_values: [0 1 2 3 4 5 6 7 8 9]

PMF: [0.7213349068959932, 0.26536428244635235, 0.013211709672545702, 8.901979954091776e-05, 8.11755497786411e-08, 1.0017858694810603e-11, 1.673152784796876e-16, 3.781876441848622e-22, 1.1568847794126817e-28, 4.789430934579018e-36]

Sum of PMF: 1.0

Calculating Entropy:

As this distribution is a discrete distribution. Entropy is calculated using the formula:

$$-\sum_{n=0}^{\infty} \left(\frac{e^{-n^2}}{\sum_{n=0}^{\infty} e^{-n^2}} \right) \cdot \log_2 \left(\frac{e^{-n^2}}{\sum_{n=0}^{\infty} e^{-n^2}} \right)$$

We've used a code for calculating entropy for N=5,10,20,1000000 and for n from 0 to 9.

Description about the code:

There is a function called calculate_entropy in 1.2 Block which has the input of pmf values that had been calculated from the previous part.
To avoid Division by zero we replaced any 0 values with small values, in this case we could use Numpy Library to add a float epsilon value, then we use the sum function to calculate entropy.

Output:

N = 5, Entropy: 0.9314981907484239

N = 10, Entropy: 0.9314981907484239

N = 20, Entropy: 0.9314981907484239

N = 50, Entropy: 0.9314981907484239

N = 100, Entropy: 0.9314981907484239

N = 200, Entropy: 0.9314981907484239

N = 500, Entropy: 0.9314981907484239

N = 1000, Entropy: 0.9314981907484239

N = 1000000, Entropy: 0.9314981907484239

As you can see, by increasing the value of N the entropy converges to a specific value.

Which is 0.931498.

1.3

Normalisation

Numerical Operation would be the same as operation that have done for 1.2
First of all, there is a need to normalise this function.To normalise a probability distribution the sum of all probabilities should equal 1. In this case we have a distribution with values $P(n) = n^{-4}$ for n = 1,2,...
The normalisation condition is given by:

$$Z = \sum_{n=0}^{\infty} n^{-4}$$

So Now, The Probability mass function after substituting normalizer is

$$P(n) = \frac{n^{-4}}{\sum_{n=0}^{\infty} n^{-4}}$$

For being sure about the Normalizer, we'll use specific N values of the PMF for checking that the sum is equal to 1.

Description about the code:

First of all we imported numpy which is the mathematical package.
After that we defined a function that calculated the pmf of the given distribution.

In this function, first we calculate the values of $P(n) = n^{-4}$, Then calculate the value of the normalizer which is $Z = \sum_{n=0}^{\infty} n^{-4}$ and name it as Z_N. Finally, calculate all of the equation by

$$P(n) = n^{-4} / Z = \sum_{n=0}^{\infty} n^{-4}$$

After that we chose n=1 to 14 , and N=5 for being sure that the sum of this pmf equals one.

$$P(n) = \frac{n^{-4}}{\sum_{n=0}^{N} n^{-4}}$$

And it is shown that in output, the sum is equal to one, so we can be sure about the normalizer.

n_values: [1 2 3 4 5 6 7 8 9 10]

PMF: [0.06757383339037096, 0.0803592834558376,
0.08893216608064863, 0.09556383164220253, 0.10104644940273703,
0.10575876465571102, 0.10991401357684863, 0.1136451885258294,
0.1170413126943168, 0.12016515657549734]

Sum of PMF: 1.0

Normalization Factor Z: 14.798627661435832

Calculating Entropy:

As this distribution is a discrete distribution. Entropy is calculated using the formula:

$$-\sum_{n=0}^{\infty} \left(\frac{n^{-4}}{\sum_{n=0}^{\infty} n^{-4}} \right) \cdot \log \left(\frac{n^{-4}}{\sum_{n=0}^{\infty} n^{-4}} \right)$$

We've used a code for calculating entropy for N=5,10,20,....,1000000 and for n from 1 to 14.

Description about the code:

There is a function called calculate_entropy in 1.3 Blocks which has the input of pmf values that had been calculated from the previous part.

To avoid Division by zero we replaced any 0 values with small values, in this case we could use Numpy Library to add a float epsilon value, then we use the sum function to calculate entropy.

Output:

N = 5, Entropy: 3.785321774475401

N = 10, Entropy: 3.785321774475401

N = 20, Entropy: 3.785321774475401

N = 50, Entropy: 3.785321774475401

N = 100, Entropy: 3.785321774475401

N = 200, Entropy: 3.785321774475401

N = 500, Entropy: 3.785321774475401

N = 1000, Entropy: 3.785321774475401

N = 100000, Entropy: 3.785321774475401

N = 10000000, Entropy: 3.785321774475401

As you can see, by increasing the value of N the entropy converges to a specific value.which is *3.7832*

1.4

Noramalisation

$P(n) = n^{-4}$ for $n = 1,2,\ldots,N$

First of all, there is a need to normalise this function,To normalize a probability distribution the sum of all probabilities should equal 1, In this case we have a distribution with values n^{-4} for n = 1,2,...,N
The normalisation condition is given by the sum of all probabilities:

$$\sum_{n=0}^{N} \alpha^n = 1$$

And by using this formula we can gain a normliser like this:

$$\sum_{n=0}^{N} \alpha^n = \frac{1}{\sum_{i=1}^{n} \alpha^i}$$

To normalize the distribution,we divide each term in the series by this sum:

$$P(n) = \frac{\alpha^n}{\sum_{i=1}^{n} \alpha^i}$$

Calculating Entropy:

As this distribution is a discrete distribution, the entropy of this type of distribution is:

$$P(n) = \frac{\alpha^n}{\sum_{i=1}^{N} \alpha^i}$$

And in this case P(n) is given by:

$$H(X) = -\sum_{n=0}^{\infty} P(n)\log(P(n))$$

Substitute the expression for P(n) into the entropy formula:

$$H(X) = \sum_{n=0}^{N} \frac{\alpha^n}{\sum_{i=1}^{N} \alpha^i} \log\left(\frac{\alpha^n}{\sum_{k=1}^{N} \alpha^k}\right)$$

Now Simplify the expression step by step:

The entropy for each n is:

$$H(X) = -\frac{\alpha^n}{\sum_{i=1}^{N} \alpha^i} \log\left(\frac{\alpha^n}{\sum_{i=1}^{N} \alpha^i}\right)$$

Let's simplify the logarithmic term:

$$\log\left(\frac{\alpha^n}{\sum_{i=1}^{N} \alpha^i}\right) = \log(\alpha^n) - \log\left(\sum_{i=1}^{N} \alpha^i\right)$$

Then, Using the property of Logarithms:

$$\log\left(\alpha^n\right) = n\log_2(\alpha)$$

$$\log\left(\frac{\alpha^n}{\sum_{i=1}^{N} \alpha^i}\right) = n\log_2(\alpha) - \log_2\left(\sum_{i=1}^{N} \alpha^i\right)$$

Simplify further

$$H(X) = -n\log(\alpha) + \sum \frac{\alpha^n}{\sum_{i=1}^{N} \alpha^i} \log\left(\sum_{i=1}^{N} \alpha^i\right)$$

Now, The Overall entropy of the distribution is the sum of there H(n) values for all n:

$$H(X) = \sum_{i=1}^{N} H(n)$$

And By Substituting This Expression in H(X):

$$H(X) = \sum_{i=1}^{N} \left(-n\log_2(\alpha)\right) + \frac{\alpha^n}{\sum_{k=1}^{N} \alpha^i} \log_2\left(\sum_{k=1}^{N} \alpha^i\right)$$

Simplify further:

$$H(X) = -\log_2(\alpha)\frac{N(N+1)}{2} + \log_2\left(\sum_{i=1}^{N} \alpha^i\right)\sum_{n=1}^{N} P(n)\cdot\alpha^n$$

After expressing $\sum_{i=1}^{N} \alpha^i = \alpha + \alpha^2 + \ldots + \alpha^{N-1} = \frac{\alpha(\alpha^N - 1)}{\alpha - 1}$ as a geometric series and simplifications:

$$H(X) = -\log_2(\alpha)\frac{N(N+1)}{2} + \log_2\left(\frac{\alpha(\alpha^N - 1)}{\alpha - 1}\right)\sum_{n=1}^{N} \frac{\alpha(\alpha^{2n-1})}{\alpha^{N-1}}$$

Let's simplify the $\sum_{n=1}^{N} \left(\frac{\alpha^{2n-1}}{\alpha^N - 1}\right)$ using geometric series:

Let $\sum_{n=1}^{N}\left(\frac{\alpha^{2n-1}}{\alpha^N-1}\right)=\frac{1-(\alpha^2)^N}{1-\alpha^2}$

Then we have:

$$H(X)=-\log_2(\alpha)\frac{N(N+1)}{2}+\log_2\left(\frac{\alpha(\alpha^N-1)}{\alpha-1}\right)\frac{\left(1-(\alpha^2)^N\right)}{1-\alpha^2}$$

1.5

Numerical Operation would be the same as operation that have done for 1.2 and 1.3.

First of all, there is a need to normalise this function.To normalise a probability distribution the sum of all probabilities should equal 1. In this case we have a distribution with values $P(n) = (1 + n^2)^{-k}$ for n = 1,2,...
The normalisation condition is given by:

$$Z = \sum_{n=1}^{\infty} \sum_{k=1}^{2} (1 + n^2)^{-k}$$

So Now, The Probability mass function after substituting normalizer is

$$P(n) = \frac{(1 + n^2)^{-k}}{\sum_{n=1}^{\infty} \sum_{k=1}^{2} (1 + n^2)^{-k}}$$

For being sure about the Normalizer,We'll use specific N values of the PMF for checking that the sum is equal to 1.
Description about the code:
First of all we imported numpy which is the mathematical package.
After that we defined a function that calculated the pmf of the given distribution.
In this function, first we calculate the values of $P(n) = \frac{(1 + n^2)^{-k}}{\sum_{n=1}^{\infty} \sum_{k=1}^{2} (1 + n^2)^{-k}}$, Then calculate the value of the normalizer which is $Z = \sum_{n=1}^{\infty} \sum_{k=1}^{2} (1 + n^2)^{-k}$ and name it as Z. finally, calculate all of the equation by

$$P(n) = \frac{(1 + n^2)^{-k}}{\sum_{n=1}^{\infty} \sum_{k=1}^{2} (1 + n^2)^{-k}} / Z = \sum_{n=1}^{\infty} \sum_{k=1}^{2} (1 + n^2)^{-k}$$

After that we chose n=1,2,...,5 , and k=1,2 for being sure that the sum of this pmf equals one.

$$P(n) = \frac{(1+n^2)^{-k}}{\sum_{n=1}^{\infty}\sum_{k=1}^{2}(1+n^2)^{-k}}$$

And it is shown that in output, the sum is equal to one, so we can be sure about the normalizer.

n_values: [1 2 3 4 5]

k_values: [1 2]

PMF:

P(n=1, k=1): 0.0263037851393766

P(n=1, k=2): 0.018599584842927133

P(n=2, k=1): 0.0657594628484415

P(n=2, k=2): 0.02940852581859743

P(n=3, k=1): 0.131518925696883

P(n=3, k=2): 0.04158993606205982

P(n=4, k=1): 0.22358217368470112

P(n=4, k=2): 0.054226642241600946

P(n=5, k=1): 0.34194920681189583

P(n=5, k=2): 0.06706175685351659

Sum of PMF: 1.0

Normalization Factor Z: 76.03468433925171

Calculating Entropy:

As this distribution is a discrete distribution. Entropy is calculated using the formula:

$$-\sum_{n=0}^{\infty}\left(\frac{(1+n^2)^{-k}}{\sum_{n=1}^{\infty}\sum_{k=1}^{2}(1+n^2)^{-k}}\right)\cdot\log_2\left(\frac{(1+n^2)^{-k}}{\sum_{n=1}^{\infty}\sum_{k=1}^{2}(1+n^2)^{-k}}\right)$$

We've used a code for calculating entropy for n=5,10,15,...,10000 and for k 1 or 2.

Description about the code:

There is a function called calculate_entropy in 1.5 Block which has the input of pmf values that had been calculated from the previous part.
To avoid Division by zero we replaced any 0 values with small values, in this case we could use Numpy Library to add a float epsilon value, then we use the sum function to calculate entropy.
Output(last 3 ones):

```
Entropy for n=990, k=1: 9.328853444379828
Entropy for n=990, k=2: 9.673391626891972
Entropy for n=995, k=1: 9.336117724407252
Entropy for n=995, k=2: 9.680655706020246
Entropy for n=1000, k=1: 9.343345610695637
Entropy for n=1000, k=2: 9.68788339397211
```

It seems that this series does not converge to a certain number, because the entropy value is increasing with the increase of the variable.

- The differential entropy of the following pdfs

1.

f(x) = u(x-a) - u(x-b) / b-a

The function f(x) is non-zero only in the interval [a,b], indicating that it can be treated as a uniform distribution over this interval.
The probability density function for a uniform distribution over [a,b] is given by: p(x) = 1/(b-a)
To calculate the differential entropy, we can use the formula:

H = -∫[a to b] p(x) * log(p(x)) dx

Substituting the PDF of the uniform distribution into the formula, we have:

H = -∫[a to b] (1/b-a) * log(1/(b-a)) dx

Simplifying further, we obtain:

H = -log(1/(b-a)) * ∫[a to b] (1/b-a)) dx
H = -log(1/(b-a)) * [x / (b-a)] | [a to b]
H = -log(1/(b-a)) * (b-a) / (b-a)
H = -log(1/(b-a))

2.

f(x) = λ * exp(-λ * x) * u(x)

To calculate the differential entropy of a continuous random variable defined by the probability density function (PDF) f(x), we can use the formula:

H(X) = -∫[f(x) * log(f(x))] dx
H(X) = -∫[(λ * exp(-λ * x) * u(x)) * log(λ * exp(-λ * x) * u(x))] dx

Since u(x) = 1 for x >= 1 otherwise, we can simplify the integral by considering two cases:

Case 1: x < 0
For x < 0, f(x) = 0, so the integral over this range has not any effect in entropy.

Case 2: x ≥ 0
For x ≥ 0, the PDF f(x) is non-zero, so we can calculate the entropy over this range:

H(X) = -∫[(λ * exp(-λ * x) * log(λ * exp(-λ * x))] dx
H(X) = -λ * ∫[exp(-λ * x) * log(λ * exp(-λ * x))] dx
H(X) = -λ * ∫[exp(-λ * x) * (log(λ) - λ * x)] dx
H(X) = -λ * [log(λ) * ∫[exp(-λ * x)] dx - λ * ∫[x * exp(-λ * x)] dx]
As ∫[exp(-λ * x)] dx = (-exp(-λ * x) / λ) - λ
And ∫[x * exp(-λ * x)] dx] = ∫[-exp(-λ * x)] dx - ∫[(-exp(-λ * x) / λ) dx]
H(X) = -λ * [log(λ) * (-exp(-λ * x) / λ) - λ * (∫[-exp(-λ * x)] dx - ∫[(-exp(-λ * x) / λ) dx])]

Simplify further:

$H(X) = -\lambda * [\log(\lambda) * (-\exp(-\lambda * x) / \lambda) - \lambda * (-\exp(-\lambda * x) / \lambda - \int[\exp(-\lambda * x)\, dx] / \lambda)]$
$H(X) = -\lambda * [\log(\lambda) * (-\exp(-\lambda * x) / \lambda) - \lambda * (-\exp(-\lambda * x) / \lambda + (-1/\lambda^2) * \exp(-\lambda * x))]$
$H(X) = -\lambda * [\log(\lambda) * (-\exp(-\lambda * x) / \lambda) + (1/\lambda) * \exp(-\lambda * x) - (1/\lambda^2) * \exp(-\lambda * x)]$
$H(X) = -\lambda * [(-\log(\lambda) + 1/\lambda - 1/\lambda^2) * \exp(-\lambda * x)]$
Now, for $x \geq 0$:
$H(X) = -\lambda * [(-\log(\lambda) + 1/\lambda - 1/\lambda^2) * [0, \infty)]$
$H(X) = \lambda * (\log(\lambda) - 1/\lambda + 1/\lambda^2)$

3.

f(x) = (1/Γ(n)) * (xn-1) * exp(-x) * u(x)**

Unfortunately, it is not possible to provide a closed-form expression for the differential entropy of this specific PDF without knowing the value of n.
The result will depend on the specific value of n and will involve the gamma function and its properties.
So we should calculate the numerical value of the differential entropy for the PDF using Python, we've used numerical integration techniques.by using the numerical integration function from the SciPy library
Description of code:
First, we've imported 2 Libraries Numpy and Scipy
Numpy for calculating Log and Exp
Scipy for calculating integration and importing special Values(in this case Γ(n))
Second, we've calculated PDF of a function by the given formula of a function
Third, we've defined a function for calculating differential entropy, which takes n and by using pdf function(output of the second step) and first prepare the function that should be integrated and then take the integration by using *'scipy.integrate.quad'* which returns to parameters, the second parameter is the estimate of the absolute error which has not any contribution and we don't use it ,so we put '_'.
And at the end this function returns the result of the integral.
Finally, we've set 10 values from 20 to 30 for n, and calculated the differential entropy for these values.

Output:

Differential entropy for n=20: 2.8999283285022166

Differential entropy for n=21: 2.9251366096789346

Differential entropy for n=22: 2.949135055027697

Differential entropy for n=23: 2.9720344683955506

Differential entropy for n=24: 2.9939310988820313

Differential entropy for n=25: 3.01490908292018

Differential entropy for n=26: 3.0350423948145835

Differential entropy for n=27: 3.0543964198632594

Differential entropy for n=28: 3.0730292344371

Differential entropy for n=29: 3.0909926561466516

Differential entropy for n=30: 3.1083331119513096

2. Markov Source

In this question we have a Markov source which is characterised by this conditional probabilities and alphabet:

$$P(X_n = x_n \mid X_{n-3:n-1} = x_{n-3:n-1}) \propto \max(x_{n-3:n-1})^{\frac{x_n}{5}}$$

Alphabet: {1,2,3,4,5}

A. Asymptotic State Distribution

To calculate it, at first we need to calculate the number of states and define the conditional probabilities in order to make the transition matrix.

- Number of States

It is the total number of states in the Markov chain (which is alphabet size raised to the power of 3, since the memory length is 3).¶

```
n_states = (alphabet_size)^(memory_length)
```

- Conditional Probabilities

The conditional probability is calculated using the given formula. At first we should find which item in the triple vector has the maximum amount, then we should raised it to the power of Xn/5. Therefore, it takes 2 parameters: triple vector which has 3 numbers and Xn.

```
conditional_probability(x_values, Xn)
```

- Transition Matrix

It iterates over all possible combinations of i, j, and k representing the current state, and for each combination, it calculates the conditional

probability for transitioning to each possible next state Xn. The index_d represents the departure state and it is calculated using a linearization of the three indices **i**, **j**, and **k** into a single index. The formula **i*np.power(alphabet_size,2) + j*alphabet_size + k** converts the three-dimensional indices into a one-dimensional index. The index_a represents the arrival state.
Similar to index_d, it is calculated by linearizing the indices **j**, **k**, and **Xn** into a single index. The formula **j*np.power(alphabet_size,2) + k*alphabet_size + Xn** converts the three-dimensional indices into a one-dimensional index.

```
vec = [i+1, j+1, k+1]
t_matrix[index_d, index_a] = conditional_probability(vec, Xn+1)
```

- Normalisation

After calculating the probabilities, each row of the transition matrix is normalized to ensure that the probabilities for each departure state sum to 1. We used **sum_row** which is the sum of probabilities for the departure state index_d. It is incremented by the current probability t_matrix[index_d, index_a].
This normalization step ensures that the rows of the transition matrix represent valid probability distributions.

- Eigenvalue Decomposition

To perform the eigenvalue decomposition we need to use the transpose of the transition matrix **t_matrix**.

```
eigenvalues, eigenvectors = np.linalg.eig(t_matrix.T)
```

- Identifying the Stationary Distribution

Then we should identify the index where the eigenvalue is close to 1. In a Markov chain, the stationary distribution corresponds to the eigenvector associated with the eigenvalue 1.

```
index = np.where(np.isclose(eigenvalues, 1))[0][0]
```

- Extracting the Eigenvector

Then we extract the eigenvector associated with the eigenvalue 1 from the eigenvectors array. The **.real** ensures that only the real part is considered, as eigenvectors may have complex components.

```
asymptotic_stationary_vector = eigenvectors[:, index].real
```

- Normalizing the Stationary Distribution

The extracted eigenvector is then normalized, ensuring that the elements of the stationary distribution sum to 1.

```
asymptotic_stationary_distribution = asymptotic_stationary_vector /
asymptotic_stationary_vector.sum()
```

- Result

Now we have a vector which has a size equal to 125 as our Asymptotic Stationary State Distribution:

```
[0.00132627 0.00162124 0.00200606 0.00250985 0.00317157
0.0016431
 0.00207275 0.00262861 0.00335089 0.00429325 0.00203539
0.00264435
 0.00344212 0.00448928 0.00586645 0.00251736 0.00336699
0.00450529
 0.00603103 0.00807702 0.00309341 0.00426807 0.00588878
0.00812492
 0.0112102  0.00167142 0.00210666 0.00266935 0.00339997
0.00435254
 0.00219678 0.00277579 0.00352596 0.00450201 0.00577707
0.00273798
 0.00355989 0.00463749 0.00605306 0.00791622 0.00340102
0.00455019
 0.00609027 0.00815518 0.01092504 0.00419239 0.00578436
0.00798086
 0.01101143 0.01519279 0.00207824 0.00269845 0.00351044
0.00457562
```

```
 0.00597564 0.00276443 0.00359332 0.00467976 0.00610655
0.00798395
 0.00368326 0.00479273 0.00624852 0.00816244 0.01068354
0.00460931
 0.00616828 0.00825812 0.01106086 0.01482147 0.00571399
0.00788376
 0.01087746 0.01500795 0.02070692 0.00254127 0.00339854
0.00454692
 0.00608596 0.0081495  0.00343189 0.00459093 0.00614403
0.00822611
 0.01101864 0.00463694 0.00620474 0.00830622 0.01112434
0.01490523
 0.00626822 0.00838999 0.01123487 0.01505108 0.02017262
0.00785283
 0.01083478 0.01494907 0.02062568 0.02845786 0.00301778
0.00416373
 0.00574482 0.00792629 0.01093614 0.00416373 0.00574482
0.00792629
 0.01093614 0.01508892 0.00574482 0.00792629 0.01093614
0.01508892
 0.02081863 0.00792629 0.01093614 0.01508892 0.02081863
0.02872408
 0.01093614 0.01508892 0.02081863 0.02872408 0.03963146]
```

B. The Entropy Rate

In the context of Markov chains, the entropy rate reflects the average uncertainty associated with predicting the next state given the current state.
The entropy rate thus quantifies the average uncertainty in the Markov process, providing insights into the inherent randomness and unpredictability of the system.

- Elementwise Multiplication

The multiplication of the asymptotic stationary distribution and the transition matrix produces a matrix where each element represents the joint probability of the departure state and the arrival state.

asymptotic_stationary_distribution * t_matrix

- Logarithmic Transformation

Taking the logarithm of the transition matrix elements helps quantify the information content associated with each transition. The logarithmic function is used with a small constant to avoid issues with zero probabilities.

```
np.log2(t_matrix + 1e-10)
```

- Elementwise Multiplication with Asymptotic Distribution

Multiplying the logarithmic transformation by the asymptotic stationary distribution weights the information content by the probability of each transition.

```
asymptotic_stationary_distribution * t_matrix * np.log2(t_matrix + 1e-10)
```

- Summation

Summing all the weighted information content across all transitions provides the overall entropy rate. The negative sign is applied to align with the convention in information theory.

```
-np.sum(asymptotic_stationary_distribution * t_matrix *
np.log2(t_matrix + 1e-10))
```

- Result

The result of those actions led to:

Entropy Rate = 2.3407333515602926

It indicates that, on average, there is a moderate level of uncertainty associated with predicting the next state. It also provides insights into the degree of randomness and complexity of the system, which can be valuable for understanding and modeling the underlying dynamics.

3. Huffman code

Question 3 asks to calculate **the entropy rate** and **the average number of bits per symbol** required by a Huffman code for the given probability function P(n) for three different cases (N = 10, 100, 1000) while P(n) is as follows:

$$P(n) \propto \frac{1}{1+n^3}, \quad n = 0,1,\ldots,N-1$$

To ensure that the sum of the probabilities for each member of the function P(n) equals one, a coefficient is determined for each value of N (10, 100, or 1000) that adjusts the probabilities of the function's members to ensure their sum equals one. The related part of the MATLAB code is as follows:

```
%% starting the loop for different N
for i = 1:numel(N)              %% N = [10,100,1000]
    p = zeros(1,N(i));
    symbols = 1:N(i);

%% Defining the probability for each N
    for n = 0:N(i)-1
        p(n+1) = 1/(1+(n^3));
    end
    probabilities = p/sum(p);
```

A. Entropy Rate

Entropy rate represents the average uncertainty or randomness of the distribution. In other words, it quantifies the "surprise" or "information content" associated with the distribution. A higher entropy value indicates greater uncertainty or randomness, while a lower entropy value indicates less uncertainty or randomness.

To calculate **the entropy rate,** we simply used the following formula:

$$H(p) = -\sum_{i} P(i) \cdot \log_2 P(i)$$

The related part of the MATLAB code is as follows:

```
%% Calculating the entropy rate
    for j = 1:N(i)
        entropy_rate(i) = (probabilities(j)*log2(1/probabilities(j))) +
entropy_rate(i);
    end
```

B. Average number of bits per symbol

The average number of bits per symbol is a measure of the efficiency of the Huffman code. A lower average number of bits per symbol indicates that the code is more efficient, and a higher one shows that the code is less efficient.

To calculate **the average number of bits per symbol,** we should create the Huffman tree for each three probability functions created based on N = 10, 100, 1000.

```
%% Creating the Huffman tree for each probability
    dict = huffmandict(symbols,probabilities);  %% symbols = 1:N(i)
    Huffman_tree = dict(:,2);
```

In the above-mentioned code, huffmandict is a MATLAB function to create a Huffman tree for a probability function (here probabilities) based on the symbols assigned to it.

After producing the Huffman tree, the average number of bits per symbol can be calculated based on the following formula:

$$\bar{\nu} = \sum_{i} P(i) \cdot \left(\text{number of digits in Huffman code to represent } P(i)\right)$$

The related part of the MATLAB code is:

```
%% Calculating the average bits per symbol
    for j = 1:N(i)
        avg_bits(i) = (probabilities(j)*numel(Huffman_tree{j})) +
avg_bits(i);
    end
end
```

4. Text File Encoding

For solving this question we wrote a code in python and put comments, but we explain it here too.

At first we need to build a dictionary, for it we defined a Node class and needed functions.

1. Dictionary of words and their occurrences:

We used "**Count_words Function**" to make the dictionary. It is shown in the code and it has printed. It is also saved in the "Ex4" folder in "Codes" folder as "Dictionary.txt".

```
{'the': 34722, 'project': 100, 'gutenberg': 30,
'ebook': 11, 'of': 15002, 'war': 301, 'and': 22278,
'peace': 115, 'by': 2404, 'leo': 4 …}
```

A. Node Class

This class represents a node in a binary tree used for Huffman coding. Each node has a value (word), frequency (count), and pointers to its left and right children. The *lt* method is implemented to allow nodes to be compared based on their frequencies, enabling the use of heapq to create a priority queue.

B. Count_words Function

Reads a text file and performs the following:

- Converts all text to lowercase for case-insensitive counting.
- Replaces double quotes with spaces and dashes between words with spaces.
- Removes punctuation and other non-alphanumeric characters.

• Splits the text into words.
• Counts the occurrences of each word and returns the word counts and the total number of words.

2. Entropy Rate by using the word frequencies:

We used "**Entropy_rate Function**" and as it is wanted in the question, it calculates the probabilities by using the word frequencies.

probabilities = [count / total_words for count in word_counts.values()]

And the result is: 9.61844

C. Entropy_rate Function

Calculates the entropy rate of a given set of word counts. Entropy is a measure of uncertainty, and the entropy rate is the average uncertainty per word in the given distribution.

3. Huffman encoding to encode the *N* words with the higher frequencies:

Huffman encoding is a variable-length prefix coding algorithm that assigns shorter codes to more frequent words. It involves constructing a binary tree (Huffman tree) based on word frequencies, where shorter codes correspond to words higher in frequency.
We applied the Huffman method for encoding the N words by using two functions, "**Build_huffman_tree Function**" and "**Build_huffman_codes Function**".

D. Build_huffman_tree Function

This function builds a Huffman tree using a priority queue (heap) based on the word counts.

• Heap Initialization: Creates a heap (priority queue) of nodes, each representing a word and its frequency.
• Building Huffman Tree: Repeatedly pops the two nodes with the lowest frequencies from the heap, creates a new node with their sum as the frequency, and inserts it back into the heap.
• Termination: Continues until only one node (the root) remains, representing the entire Huffman tree.
• Return: Returns the root of the Huffman tree.

E. Build_huffman_codes Function

Recursively builds Huffman codes for each word in the Huffman tree. The codes are generated based on the path from the root to each leaf node (word).
• Mapping Initialization: Initializes an empty dictionary (mapping) to store Huffman codes.
• Base Case: If the node represents a word (node.value is not None), assigns the Huffman code (code) to the word in the mapping and returns the updated mapping.
• Recursion: Updates the mapping by recursively traversing the left and right subtrees, appending '0' for the left branch and '1' for the right branch.
• Return: Returns the final mapping.

4. Encoding the remaining words by maximum-length coding:

Maximum Length Encoding is a fixed-length encoding algorithm. For each word, it uses a fixed number of bits to represent the word's length and the binary representation of each character in the word.
We used "**Mls_encode_word Function**" and "**Mls_encode_words Function**" to encode the remaining words.

F. Mls_encode_word Function

This function encodes a word using maximum length coding. It represents the word with a binary string consisting of:

- Length Encoding: Represents the length of the word using 5 bits.
- Character Encoding: Encodes each character of the word using 7 bits.
- Flag Bit: Appends a flag bit ('1') to indicate the start of the token.
- Return: Returns the encoded binary string.

G. Mls_encode_words Function

This function encodes a list of words using maximum length coding.

- Iterative Encoding: Iterates through a list of words starting from a specified position.
- Word Extraction: Extracts each word from the sorted list of words and its probabilities.
- Encoding: Calls mls_encode_word for each word and appends the result to the encoded_message list.
- Return: Returns a list of encoded binary strings.

5. Details of the encoder:

In conclusion:

- Huffman encoding focuses on assigning shorter codes to more frequent words.
- Maximum Length Encoding uses fixed-length codes for each word.
- Both methods aim to reduce the overall length of the encoded message and are used together in the provided code for optimal compression.

At the end by using the "**Save_to_csv Function**" we make a table of each word, its occurrence, the probability of it and the encoded format

	A	B	C	D	E
1	Word	Count	Probability	Huffman Code	Max Length Code
2	the	34722	611768%	1000	
3	and	22278	392517%	11100	
4	to	16753	295172%	1111	
5	of	15002	264321%	1000	
6	a	10569	186216%	110101	
7	he	9858	173688%	110000	
8	in	8962	157902%	101000	
9	his	7982	140635%	10111	
10	that	7902	139226%	10101	
11	was	7360	129676%	1110	

of that using Huffman codes and maximum length method. This table is saved as "Encoded.csv" file in the "Ex4" folder in "Codes" folder.

It is notable that words are sorted based on their occurrences and probabilities. For N = 10000 words, they are encoded by Huffman Code

10000	kinder	2	0.00035%	1110000000000000	
10001	rogues	2	0.00035%	1110000000000000	
10002	notwithstand	2	0.00035%		101110000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
10003	aunts	2	0.00035%		100101000000000000000000000000000000000000
10004	miniaturist	2	0.00035%		1010110000000000000000000000000000000000000000000000000000000000000000000000000000000
10005	reprehensible	2	0.00035%		1011010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
10006	incomplete	2	0.00035%		101010000000000000000000000000000000000000000000000000000000000000000000000000
10007	worthlessnes	2	0.00035%		1011010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
10008	frivolity	2	0.00035%		10100100000000000000000000000000000000000000000000000000000000000000000
10009	palpably	2	0.00035%		1010000000000000000000000000000000000000000000000000000000000000
10010	paralyzed	2	0.00035%		10100100000000000000000000000000000000000000000000000000000000000000000

method and for the remaining words the table looks like this:

H. Save_to_csv Function

This function takes word counts, calculates probabilities, builds Huffman codes for the top N words, and encodes the remaining words using maximum length coding. It then saves the results in a CSV file, including word statistics, Huffman codes, and maximum length codes.

Parameters:

- word_counts: Counter
- A Counter object containing word frequencies.

• total_words: int
• Total number of words in the text.
• csv_file_path: str
• The file path for the CSV file to be created.
• top_n: int, optional (default=10000)
• The number of top words to be considered for Huffman coding.

6. Evaluate the memory occupied by the dictionary:

Memory Occupied by the Dictionary:

• The memory occupation is influenced by the size of the dataset and the number of unique words present.
• This memory is mainly consumed by the Python objects, including keys (words) and values (frequencies).

We write these lines to evaluate the memory occupation by the dictionary:

First we used sys library to get the size of our dictionary.

word_counts_size = sys.getsizeof(word_counts)

Then printed it in bytes:

Memory occupied by the dictionary: 589936 bytes

7. Evaluate the memory occupied by the encoded text as a function of N:

Memory Occupied by Encoded Text as a Function of N:

• The encoded text's memory occupation as a function of N (when N = 10000) is 75,672 bytes.
• This value represents the storage requirements for the encoded text using both Huffman and Maximum Length Encoding.
• The size is influenced by the number of words encoded, with additional overhead for encoding information.

top_n is a parameter to indicate the value of N.
For instance we assumed: top_n = 10000.

top_n_encoded_size = sys.getsizeof(encoded_results[:top_n])

It is obvious that for smaller N we will need a lower amount of bytes.

Just like before we got the size by sys library and printed it in bytes:

Memory occupied by the dictionary: 68688 bytes

Evaluation:

• The memory usage is significantly reduced in the encoded text compared to the original word frequencies dictionary.
• Huffman encoding efficiently represents frequently occurring words with shorter codes, reducing the overall size.
• Maximum Length Encoding further contributes to compression, using fixed-length codes for each word after the top N.
• The memory efficiency is a trade-off between the encoding scheme and the original dataset's characteristics.

To conclude:

The code achieves effective compression, reducing the memory footprint, especially in the encoded text. The balance between Huffman and Maximum Length Encoding, along with careful data structures, results in an optimized memory utilization strategy.