Statistical machine learning - Exercise 1

Dominik Marino - 2468378, Pertami Kunz - 2380210
8. August 2020

Inhaltsverzeichnis

1 Task 1: Linear Algebra Refresher

1.1 1a)

- Associative rule:
  - Let **A, B, C** be matrices and * an operator
    - Rule: (A*B)*C = A*(B*C)
- Commutative rule:
  - Let **A, B, C** be matrices and * an operator
    - Rule: A*B*C = B*A*C = B*C*A = C*B*A = C*A*B
- Distributive rule:
  - Let **A, B, C** be matrices and *, △ operators
    - left distributive: A*(B△C) = (A*B) △(A*C)
    - right distributive: (A△B)*C = (A*C) △(B*C)

Let $\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}$, $\boldsymbol{B} = \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix}$, $\boldsymbol{C} = \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix}$ with a_{ij}, b_{ij}, c_{ij} scalars $\forall i, j$.

Addition:
Commutative, distributive, and associative rules apply. Proofs:

- Commutative

$$A + B \overset{?}{=} B + A$$
$$a_{ij} + b_{ij} \overset{\checkmark}{=} b_{ij} + a_{ij}$$

  Proven since for scalar addition, commutative rule applies.

- Left Distributive

$$C\,(A + B) \overset{?}{=} CA + CB$$
$$\begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix} \begin{bmatrix} a_{11}+b_{11} & \cdots & a_{1n}+b_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1}+b_{n1} & \cdots & a_{nn}+b_{nn} \end{bmatrix} \overset{?}{=} \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix} \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} + \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix}$$
$$[C\,(A + B)]_{ij} = \sum_k c_{ik}\,(a_{kj} + b_{kj}) \overset{\checkmark}{=} \sum_k c_{ik}a_{kj} + \sum_k c_{ik}b_{kj}$$

- Right distributive

$$(A + B)\,C \overset{?}{=} AC + BC$$
$$\begin{bmatrix} a_{11}+b_{11} & \cdots & a_{1n}+b_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1}+b_{n1} & \cdots & a_{nn}+b_{nn} \end{bmatrix} \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix} \overset{?}{=} \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix} \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{bmatrix}$$
$$[(A + B)\,C]_{ij} = \sum_k (a_{ik} + b_{ik})\,c_{kj} \overset{\checkmark}{=} \sum_k a_{ik}c_{kj} + \sum_k b_{ik}c_{kj}$$

- Associative

$$(A+B)+C \overset{?}{=} A+(B+C)$$
$$(a_{ij}+b_{ij})+c_{ij} \overset{\checkmark}{=} a_{ij}+(b_{ij}+c_{ij})\,.$$

  Proven since for scalar addition, associative rule applies.

Multiplication:

- Commutative

$$A*B \overset{?}{=} B*A$$
$$\begin{bmatrix} \sum_{k=1}^{n} a_{1k}b_{k1} & \cdots \\ \vdots & \ddots & \vdots \end{bmatrix} \overset{\times}{=} \begin{bmatrix} \sum_{k=1}^{n} b_{1k}a_{k1} & \cdots \\ \vdots & \ddots & \vdots \end{bmatrix}$$

  Not proven since in general $\sum_k a_{ik}b_{kj} \neq \sum_k b_{ik}a_{kj}$.

- Distributive property of multiplication over addition (by generalization, also subtraction) was proven above. Distributive property of multiplication over multiplication does not exist.
- Counterexample for commutative property:

$$A=\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}, B=\begin{pmatrix}0 & 1\\ 3 & 2\end{pmatrix}$$

- Associative:

$$(A*B)*C \overset{?}{=} A*(B*C)$$
$$\sum_{l=1}^{n}\left(\sum_{k=1}^{n} a_{ik}b_{kl}\right)c_{lj} \overset{?}{=} \sum_{k=1}^{n} a_{ik}\left(\sum_{l=1}^{n} b_{kl}c_{lj}\right)$$
$$\sum_{l=1}^{n}\sum_{k=1}^{n} a_{ik}b_{kl}c_{lj} \overset{\checkmark}{=} \sum_{k=1}^{n}\sum_{l=1}^{n} a_{ik}b_{kl}c_{lj}$$

  Proven.

1.2 1b)

- The inverse of matrix $A=\begin{bmatrix}1 & 2 & 3\\ 1 & 4 & 6\\ 0 & 0 & 1\end{bmatrix}$ using **linear row reduction**,

$$\left[\begin{array}{ccc|ccc}1 & 2 & 3 & 1 & 0 & 0\\ 1 & 4 & 6 & 0 & 1 & 0\\ 0 & 0 & 1 & 0 & 0 & 1\end{array}\right]$$
$$A'_{1\bullet}=(A_{1\bullet}-0.5A_{2\bullet})\times 2\left[\begin{array}{ccc|ccc}1 & 0 & 0 & 2 & -1 & 0\\ 1 & 4 & 6 & 0 & 1 & 0\\ 0 & 0 & 1 & 0 & 0 & 1\end{array}\right]$$
$$A'_{2\bullet}=(A_{2\bullet}-A_{1\bullet}-6A_{3\bullet})/4\left[\begin{array}{ccc|ccc}1 & 0 & 0 & 2 & -1 & 0\\ 0 & 1 & 0 & -0.5 & 0.5 & -1.5\\ 0 & 0 & 1 & 0 & 0 & 1\end{array}\right].$$

  The inverse is then $A^{-1}=\begin{bmatrix}2 & -1 & 0\\ -0.5 & 0.5 & -1.5\\ 0 & 0 & 1\end{bmatrix}$.

- The matrix $A=\begin{bmatrix}1 & 2 & 3\\ 1 & 4 & 6\\ 1 & 0 & 0\end{bmatrix}$ is not invertible because it is rank deficient; columns 2 and 3 are linear combination of each other. You can check it with the determinante of the Matix. Is det(A) $=0$, all the vectors are linear dependent. If det(A) $\neq 0$ however, linear independent. Its an easy way to check that.

1.3 1c)

- Left Moore-Penrose pseudoinverse of the matrix $A \in \mathbb{R}^{n\times m}$, $A^\dagger$, is where

$$A^\dagger A = I$$
$$A^\dagger = \left(A^T A\right)^{-1} A^T$$

- Right

$$AA^\dagger = I$$
$$A^\dagger = A^T \left(AA^T\right)^{-1}$$

- For $A \in \mathbb{R}^{2\times 3}$,
  - Left: $\left(A^T_{3\times 2} A_{2\times 3}\right)_{3\times 3} \Rightarrow \left[\left(A^T A\right)^{-1}\right]_{3\times 3} A^T_{3\times 2} \Rightarrow A^\dagger_{3\times 2}$
    $\left(A^T A\right)$ is rank deficient since $\operatorname{rank}\left(A^T A\right) \le \min\left\{\operatorname{rank}\left(A^T\right), \operatorname{rank}\left(A\right)\right\} \le 2$, therefore it cannot be inverted so the pseudoinverse does not exist.
  - Right: $\left(A_{2\times 3} A^T_{3\times 2}\right)_{2\times 2} \Rightarrow A^T_{3\times 2} \left[\left(AA^T\right)^{-1}\right]_{2\times 2} \Rightarrow A^\dagger_{3\times 2}$
    pseudoinverse exists if A has full row rank.
  - Solution of Tutors: If the number of rows is less than the number of columns, only the right pseudoinverse exists. The reasen is that $A^T A$ will not be full rank because there will be some linear dependence between the columns. Therefore $A^T A$ is not invertible. Furthermore, if the matrix is not full rank, the left and right Moore penrose pseudoinverse do not exist, as both $A^T A$ and AA^T are not invertible. In this case, we can comute the pseudoinverse using SVD deomposition.

1.4 1d)

Given a matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ (can also be complex) the eigenvalue λ and eigenvector $\boldsymbol{v}$ of $\boldsymbol{A}$ associated with λ are the entities such that

$$\boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v}. \tag{1}$$

If $\lambda_1, \ldots, \lambda_n$ are the n eigenvalues of $\boldsymbol{A}$, $\boldsymbol{v}_i$ is the eigenvector of $\boldsymbol{A}$ associated with λ_i. The n eigenvectors are also the n roots of the characteristic polynomial $p\left(\lambda\right) = \det\left(\boldsymbol{A} - \lambda \boldsymbol{I}\right)$.

- Definitions:
  - Eigenvalue: An eigenvalue is a scalar parameter (to transform/stretch an eigenvector), in an equation, which is considered to be a specific value or a set of value for the equation to have a solution.
  - Eigenvector: is a vector whose direction remain unchanged when linear transformation is applied to it.
- Why they are important in ML:
  - Eigenvalues quantify the importance of an information along the line of Eigenvectors. With Eigenvalues and Eigenvectors we know what part of information can be ignored and eventually compress information (SVD = Singular value dimension, dimensionality reduction, and PCA = Principal Component Analysis). It also helps us to extract features in developing machine learning models. Also it makes it easier to train our model, because of the reduction of some tangled information (Problem: curse of dimensionality). It also serves the purpose to visualize tangled raw data. So we can use eigenvector to understand the correlation among data.
- Solution of the Tutors:

  An eigenvector v of a matrix A is a vector that does not change its direction when the linear transformation defined by A is applied to it. Formally,

$$Av = \lambda v$$

  where λ is the eigenvalue associated to v.

  Eigenvectors are imprtant in ML because they are the smalles set of basis function to describe data variability. They also represent the coordinate system in which teh data covariance matrix becomes diagonal. Variables referenced to this coordinate system are therefore uncorrelated. For this reason, they are largely used in dimensinality reduction, e.g.PCA, SVD, LDA, ...

2 Task 2: Statistics Refresher

2.1 2a)

1)
With Ω a finite set and $P : \Omega \to \mathbb{R}$ a probability measure: $P(\omega) \geq 0, \sum_{\omega\in\Omega} P(\omega) = 1\ \forall\omega \in \Omega$ and let $f : \Omega \to \mathbb{R}$ an arbitrary function on Ω,

- Expectation of f or its mean is the measure of central tendency of f, mathematically the probability-weighted of the values of f:

  $$\mu_f = E[f(\omega)] = \sum_{\omega\in\Omega} P(\omega) f(\omega).$$

  Linearity of expectation is the property that the expected value of the sum of random variables is equal to the sum of their individual expected values, regardless of whether they are independent. The expected value of a random variable is essentially a weighted average of possible outcomes[1].

  Proof of the Expectation is linearity:

  $$\begin{aligned} E[X+Y] &= E[X] + E[Y] \\ E[X+Y] &= \sum_x \sum_y [(x+y)P(X=x, Y=y)] \\ &= \sum_x \sum_y [x \cdot P(X=x,Y=y)] + \sum_x \sum_y [y \cdot P(X=x,Y=y)] \\ &= \sum_x x \sum_y P(X=x,Y=y) + \sum_y y \sum_x P(X=x,Y=y) \\ &= \sum_x x(X=x) + \sum_y y(Y=y) \\ &= E[X] + E[Y] \end{aligned}$$

  More generally, for random variables $X_1, X_2, \ldots, X_n$ and constants $c_1, c_2, \ldots, c_n$

  $$E\left[\sum_i c_i X_i\right] = \sum_i (c_i E[X_i]).$$

  Expectation is a linear operator.

- Variance of f is the measure of spread of the values of f around its expected value:

  $$\begin{aligned} \sigma_f^2 = E\left[(f(\omega) - \mu_f)^2\right] &= E\left[(f(\omega))^2\right] - \mu_f^2 \\ &= \sum_{\omega\in\Omega} P(\omega)(f(\omega))^2 - \mu_f^2. \end{aligned}$$

  The Variance of X is the average value of (X - μ_x)^2 with (X - μ_x)^2 $\geq$ 0. So the variance is always larger than or equal zero. A larger variance means that the distribution is very spread out. A low variance means that the distribution is around the average.[2]

  In particular we have the following formula:

  $$\text{Var}(aX+b) = a^2\text{Var}(X).$$

  Proof that the variance is not a linear operator. If Y = aX + b, EY = aEX + b. Thus

  $$\begin{aligned} \text{Var}(Y) &= E\left[(Y - EY)^2\right] \\ &= E\left[(aX + b - aEX - b)^2\right] \\ &= E\left[a^2(X - \mu_X)^2\right] \\ &= a^2 E\left[(X - \mu_X)^2\right] \\ &= a^2\text{Var}(X) \end{aligned}$$

  Variance is not a linear operator.

[1] More information expectation: `https://www.probabilitycourse.com/chapter3/3_2_2_expectation.php`
[2] More information variance: `https://www.probabilitycourse.com/chapter3/3_2_4_variance.php`

2)
Given the following outcomes of 18 throws for each die A, B, and C:

		outcome					
		1	2	3	4	5	6
occurences	A	4	1	6	2	1	4
	B	5	6	1	1	4	1
	C	3	3	4	2	3	3

The unbiased estimate of the expected value of each die is the sample average,

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{2}$$

And the unbiased estimate of the population variance is the sample variance with degree of freedom n-1=18-1=17.

$$s^2 = \sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x}) = \frac{1}{n-1}(\sum_{i=1}^{n} x_i^2 - n * \overline{x}^2) \tag{3}$$

Class A:

$$\overline{x}_A = \frac{1}{18}(4*1+1*2+6*3+2*4+1*5+4*6) = \frac{1}{18}*61 = \frac{61}{18}$$

$$\sigma_A^2 = \frac{1}{17}*(263 - 18*\frac{61}{18}) = 3.3105$$

Class B:

$$\overline{x}_B = \frac{1}{18}(5*1+6*2+1*3+1*4+4*5+1*6) = \frac{1}{18}*50 = \frac{25}{9}$$

$$\sigma_B^2 = \frac{1}{17}*(190 - 18*\frac{50}{18}) = 3.0065$$

Class C:

$$\overline{x}_C = \frac{1}{18}(3*1+3*2+4*3+2*4+3*5+3*6) = \frac{1}{18}*62 = \frac{31}{9}$$

$$\sigma_C^2 = \frac{1}{17}*(266 - 18*\frac{62}{18}) = 3.0850$$

3)
The Kullback-Leibler divergence (in bits) from a uniform distribution $Q(x) = \frac{1}{6},\ x = 1, \ldots, 6$, is $\sum_i P(x_i) \cdot \log_2\left(\frac{P(x_i)}{Q(x_i)}\right)$ [2]

x	$P_A(x)$	$P_B(x)$	$P_C(x)$	$P(x) \cdot \log_2\left(\frac{P(x)}{Q(x)}\right)$		
				A	B	C
1	4/18	5/18	3/18	0.0922	0.2047	0
2	1/18	6/18	3/18	-0.0881	0.3333	0
3	6/18	1/18	4/18	0.3333	-0.0881	0.0922
4	2/18	1/18	2/18	-0.0650	-0.0881	-0.0650
5	1/18	4/18	3/18	-0.0881	0.0922	0
6	4/18	1/18	3/18	0.0922	-0.0881	0
$\sum$				$D_{\mathrm{KL}}(P_A\|Q))$ =0.2765	$D_{\mathrm{KL}}(P_B\|Q))$ =0.3659	$D_{\mathrm{KL}}(P_C\|Q))$ =0.0272

Answer:
The Kullback-Leibler Divergence (short KL Divergence) comparing two probability distributions. So the KL Divergence helps us to measure just how much information we lose when we choose an approximation[3]. According to the results, Class C is closest to a fair uniform die.

Note that the assignment requires you to compute **unbiased** estimators. Therefore, for the variance, you have to divide the sum by N - 1

[3]More information KL-Divergence: https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained

2.2 2b)

1) B: Random variable describing the occurence of back pain

   C: Random variable describing the occurence of cold

2) Domain: either happens/ doesn't happen (or suffering with/ not suffering with), i.e. binary:
   $B \in \{b, \bar{b}\}$, where the events b: has back pain, $\bar{b}$: doesn't have back pain
   $C \in \{c, \bar{c}\}$, where the events c: has cold, $\bar{c}$: doesn't have cold
   Tutor solution Domain of both variables is {YES, NO}

3) (Using short-hand notation $P(B = b|C = c) \Rightarrow P(b|c)$),
   $P(b|c) = 0.3$
   $P(c) = 0.03$
   $P(b|\bar{c}) = 0.1$

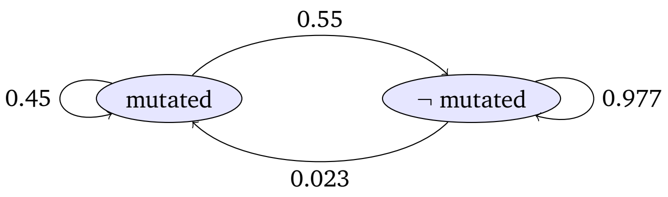
4) $P(C = c|B = b)$?

$$
\begin{aligned}
P(\bar{c}) &= 1 - P(c) = 1 - 0.03 = 0.97 \\
P(c|b) &= \frac{P(b,c)}{P(b)} = \frac{P(b|c)P(c)}{P(b,c) + P(b,\bar{c})} = \frac{P(b|c)P(c)}{P(b|c)P(c) + P(b|\bar{c})P(\bar{c})} \\
&= \frac{0.3 \times 0.03}{0.3 \times 0.03 + 0.1 \times 0.97} \\
&= 0.0849
\end{aligned}
$$

2.3 2c)

1)

$$
\begin{aligned}
P(\text{mutation}) &= 0.45 = m \\
P(\neg\text{mutation}) &= 0.55 = \tilde{m} \\
P &= \begin{pmatrix} 0.45 & 0.023 \\ 0.55 & 0.977 \end{pmatrix}
\end{aligned} \tag{4}
$$



2) The Markov transition probability matrix $P = \begin{bmatrix} 0.45 & 0.023 \\ 0.55 & 0.977 \end{bmatrix}$, with start-vector $x^{(0)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ Predicting mutation with following formula:

$$x^{(n)} = P^n \cdot x^{(0)}$$

   After 20 generations, the probability matrix is $P^{20} = \begin{bmatrix} 0.0401 & 0.0401 \\ 0.9599 & 0.9599 \end{bmatrix}$, i.e. 0.0401 probability any bacterium ends up with the mutation, and 0.9599 without.

```python
import numpy as np
import matplotlib.pyplot as plt

def mutation (n = 20):
  P = np.array([[0.45, 0.023], [0.55, 0.977]])
  start_vector = np.array ([1,  0]).T
  mut = []
  no_mut = []
  y = []
  for i in range(1, n):
    p = np.dot(np.linalg.matrix_power(P, i), start_vector)
    mut.append(p[0])
    no_mut.append(p[1])
    y.append(i)

  plt.plot(y, mut, label='mutation')
  plt.xlabel("iteration")
  plt.ylabel( 'Probability')

  plt.plot(y, no_mut , label='no mutation' )
  plt.legend()
  plt.show()
  print(np.linalg.matrix_power(P, 20))
  print(mut[-1], no_mut[-1])


if __name__ == "__main__":
  mutation()
```

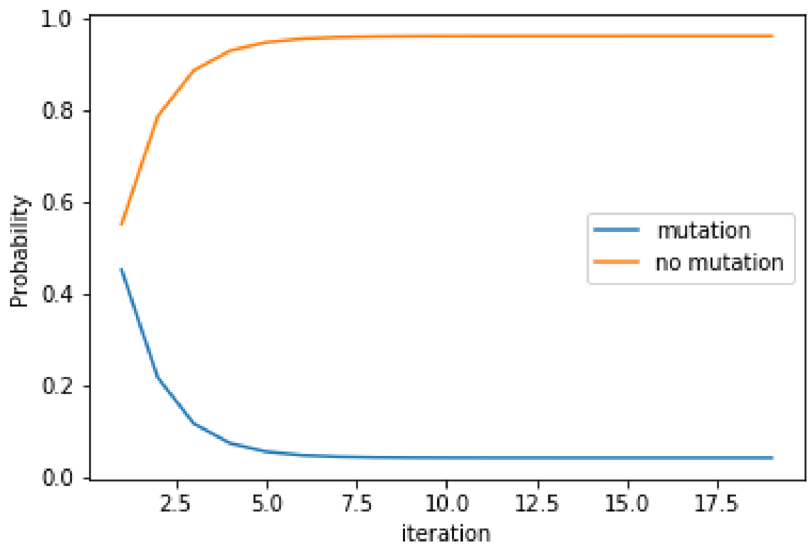


Abbildung 1: Probability of Mutation after 20 iterations

3) When $P^N = P^{N+1} = \ldots$, we achieve stationarity. We checked that after $N = 14$ iterations we achieve stationarity/stability with $P^{14} = \begin{bmatrix} 0.0401 & 0.0401 \\ 0.9599 & 0.9599 \end{bmatrix}$, so there's a chance of 0.0401 that mutation is present.

Tutor solution: Every ergodic Markov chain has a stable distribution, which it approaches as $t \to \inf$. The Markov chain in this exercise is ergodic, because every state is reachable from every other state with 0 < p < 1. This stable distribution is the fixed point of the iteration $x_{t+1} = A * x_t$ and is therefore related to the Eigenvectors λ_i of the transition matrix A. The limiting distribution will be the Eigenvector for the Eigenvalue of 1 or, more general, a linear combination of these if there are several.

3 Task 3: Optimization and Information Theory

3.1 3a)

Given the following transmit probabilities

	S_1	S_2	S_3	S_4
p_i	0.02	0.67	0.23	0.08

The information of each symbol S_i, $I(S_i) = -\log_2(p_i)$ [bits]:

	S_1	S_2	S_3	S_4
$I(S_i)$	5.64	0.58	2.12	3.64

The entropy of a source with the alphabet $S = \{S_1, S_2, S_3, S_4\}$ in bits is $H(S) = \sum_{i=1}^{4} P(S_i)\log_2\left(\frac{1}{P(S_i)}\right) = -0.02\log_2(0.02) - 0.67\log_2(0.67) - 0.23\log_2(0.23) - 0.08\log_2(0.08) = 1.2792$ bits per symbol.

In general, maximum information transmitted happens when the alphabets are uniformly distributed, then the entropy is $H'(S) = 4 \times \frac{1}{4}\log_2(4) = 2$ bits per symbol.

3.2 3b)

1) Constrained optimization problem [3]: (maximize$_{p_i}$ $\sum_{i=1}^{4} p_i \log_2\left(\frac{1}{p_i}\right)$ is converted to minimize$_{p_i}$ $\sum_{i=1}^{4} p_i \log_2(p_i)$)

$$\begin{aligned} \underset{\boldsymbol{p}=\{p_1,\dots,p_4\}}{\text{minimize}} \sum_{i=1}^{4} p_i \log_2(p_i) \\ \text{subject to} \sum_{i=1}^{4} 2p_i i = 6 \\ \sum_{i=1}^{4} p_i = 1. \end{aligned}$$

Note that it ist not necessary to explicit $p_i \leq 0$ because the constraint is already enforced by the entropy maximization (the log is defined only for non-negative p_i)

2) Its Lagrangian function:

$$L(\boldsymbol{p}, \lambda, \mu) = \sum_{i=1}^{4} p_i \log_2(p_i) + \lambda\left(\sum_{i=1}^{4} 2p_i i - 6\right) + \mu\left(\sum_{i=1}^{4} p_i - 1\right), \tag{5}$$
$$\boldsymbol{p} = \{p_1, \dots, p_4\} \text{ and } p_i \log_2(p_i) = \frac{p_i \ln(p_i)}{\ln(2)}$$

3) The partial derivatives of the Lagrangian $L(\boldsymbol{p}, \lambda, \mu)$

$$\begin{aligned} \nabla_{p_1} L &= \frac{\ln(p_1) + 1}{\ln(2)} + 2\lambda + \mu = 0 \\ \nabla_{p_2} L &= \frac{\ln(p_2) + 1}{\ln(2)} + 4\lambda + \mu = 0 \\ \nabla_{p_3} L &= \frac{\ln(p_3) + 1}{\ln(2)} + 6\lambda + \mu = 0 \\ \nabla_{p_4} L &= \frac{\ln(p_4) + 1}{\ln(2)} + 8\lambda + \mu = 0 \\ \nabla_{\lambda} L &= 2p_1 + 4p_2 + 6p_3 + 8p_4 - 4 = 0 \\ \nabla_{\mu} L &= p_1 + p_2 + p_3 + p_4 - 1 = 0 \end{aligned} \tag{6}$$

Since there are 6 unknowns, 4 of which are also enclosed in logarithmic function, the analytical solution is not straightforward.

Tutor solution:

$$\begin{aligned}\frac{\nabla L(\mathbf{p},\lambda,\mu)}{\nabla\lambda} &= 2\sum p_i i - 6\\ \frac{\nabla L(\mathbf{p},\lambda,\mu)}{\nabla\mu} &= \sum p_i - 1\\ \frac{\nabla L(\mathbf{p},\lambda,\mu)}{\nabla p_i} &= -(1+logp_i) + 2\lambda i + \mu\end{aligned}$$

4) The dual function

$$L^*(\lambda,\mu) = \underset{\boldsymbol{p}=\{p_1,\ldots,p_4\}}{\text{minimize}}\ L(\boldsymbol{p},\lambda,\mu)\,. \tag{7}$$

The dual problem

$$\begin{gathered}\underset{(\lambda,\mu)}{\text{maximize}}\ L^*(\lambda,\mu)\\ \text{subject to } \lambda,\mu \gtreqless 0.\end{gathered}$$

To solve analytically,

$$\begin{aligned}\nabla_{p_1} L \overset{!}{=} 0 \Rightarrow \frac{\ln(p_1)+1}{\ln(2)} + 2\lambda + \mu \overset{!}{=} 0&\\ \Rightarrow \log_2 p_1 &= -2\lambda - \mu - 1.44\\ p_1 &= 2^{(-2\lambda-\mu-1.44)}\\ \nabla_{p_2} L \overset{!}{=} 0 \Rightarrow \frac{\ln(p_2)+1}{\ln(2)} + 4\lambda + \mu \overset{!}{=} 0&\\ \Rightarrow \log_2 p_2 &= -4\lambda - \mu - 1.44\\ p_2 &= 2^{(-4\lambda-\mu-1.44)}\\ \nabla_{p_3} L \overset{!}{=} 0 \Rightarrow \frac{\ln(p_3)+1}{\ln(2)} + 6\lambda + \mu \overset{!}{=} 0&\\ \Rightarrow \log_2 p_3 &= -6\lambda - \mu - 1.44\\ p_3 &= 2^{(-6\lambda-\mu-1.44)}\\ \nabla_{p_4} L \overset{!}{=} 0 \Rightarrow \frac{\ln(p_4)+1}{\ln(2)} + 8\lambda + \mu \overset{!}{=} 0&\\ \Rightarrow \log_2 p_4 &= -8\lambda - \mu - 1.44\\ p_4 &= 2^{(-8\lambda-\mu-1.44)}.\end{aligned}$$

Then the Lagrangian dual

$$\begin{aligned}L^*(\lambda,\mu) &= \sum_{i=1}^{4} p_i \log_2(p_i) + \lambda\left(\sum_{i=1}^{4} 2p_i i - 6\right) + \mu\left(\sum_{i=1}^{4} p_i - 1\right)\\ &= \sum_{i=1}^{4}\left\{2^{(-2i\cdot\lambda-\mu-1.44)}\cdot(-2i\cdot\lambda-\mu-1.44)\right\}\\ &\quad+\lambda\left(\sum_{i=1}^{4} 2i\cdot 2^{(-2i\cdot\lambda-\mu-1.44)} - 6\right)\\ &\quad+\mu\left(\sum_{i=1}^{4} 2^{(-2i\cdot\lambda-\mu-1.44)} - 1\right).\end{aligned}$$

To find the optimal solutions $\lambda^*, \mu^*, p_1^*, p_2^*, p_3^*, p_4^*$ analytically, one must find the partial derivatives of the dual $L^*(\lambda,\mu)$ w.r.t. λ and μ, which is too tedious. Therefore, numerical solutions will be discussed in the next section.

Tutor solution:
By setting $\nabla L(\mathbf{p},\lambda,\mu)/\nabla p_i$ to zero

$$p_i^* = exp(2\lambda i + \mu - 1)$$

The dual is

$$\begin{aligned} g = L(\mathbf{p}^*, \lambda, \mu) &= \sum p_i^*(-2\lambda i - \mu + 1 + 2\lambda i + \mu) \\ &= \sum p_i^* - 6\lambda - \mu \end{aligned}$$

and since $\sum p_i = 1$

$$= 1 - 6\lambda - \mu$$

We can also rewrite the constraint on the distribution as

$$\begin{aligned} 1 = \sum p_i^* &= \sum exp(2\lambda i + \mu - 1) \\ &= \sum exp(2\lambda i) * exp(\mu - 1) \\ &= exp(\mu - 1) * \sum exp(2\lambda i) \end{aligned}$$

from which we get

$$\mu = 1 - log \sum exp(2\lambda i)$$

The dual can therefore be rewritten as

$$g = log \sum exp(2\lambda i) - 6\lambda$$

Solving for λ we obtain

$$\frac{\partial g}{\partial \lambda} = \frac{\sum 2iexp(2\lambda i)}{\sum exp(2\lambda i)} - 4 = 0$$

and finally (epanding the sum and simplifying)

$$\begin{aligned} \sum i * exp(2\lambda i) - \sum 2exp(2\lambda i) &= \sum (2i - 6)exp(2\lambda)^i = 0 \\ &= -4exp(2\lambda) - 2exp(2\lambda)^2 + 2exp(2\lambda)^4 = 0 \\ &\leftrightarrow -2exp(2\lambda) - exp(2\lambda)^2 + exp(2\lambda) \\ &\text{variable substitution } y = exp(2\lambda) \\ &= -2y - y^2 + y^4 = 0 \\ &\leftrightarrow -2 - y + y^3 \rightarrow y = 0.15214 \rightarrow \lambda = 0.20981 \end{aligned}$$

(Note that y = 0 is an invalid solution, because λ would not have a finite value)
Knowing λ, we can backtrack to find μ = -1.54276 and p = [0.11966, 0.18204, 0.27695, 0.42135].
Note that it would have been possible also to solve the problem by following.

$$g = \sum exp(2\lambda i + \mu - 1) - 6\lambda - \mu$$

and computing the partial derivatives with respect to λ and μ to have a system of two equations in two variables. However, this procedure is more complicate than the one above.

5) The Lagrangian Multiplier is a method in mathematical optimization and find a local maxima and minima of a function subject to equality constraints.

   Technique for numerically solving these problems:

   – Newtons Method is an iterative method for solving optimization problems:

     Since L is continuous and twice differentiable, we can use Newtons Method iteratively for finding the local minimum with variables update

$$\boldsymbol{p}_{n+1} = \boldsymbol{p}_n - \frac{L'(\boldsymbol{p}_n)}{L''(\boldsymbol{p}_n)}. \tag{8}$$

     For the computation, the Hessian (L"(p_n)) is required and it should be positive definite, because we converge to a saddle point not the minimum. L'(p_n) is the gradient. It also can be definite a stepsize α in the formula to modify the Newtons Method.

3.3 3c)

Using the Rosenbrock's function $f\left(\boldsymbol{x}\right) = \sum_{i=1}^{n-1}\left[100\left(x_{i+1} - x_i^2\right)^2 + \left(x_i - 1\right)^2\right]$, we want to find $\arg\min_{\boldsymbol{x}} f\left(\boldsymbol{x}\right)$ with $n = 20$.

The gradient of the function:

$$
\begin{aligned}
f\left(\boldsymbol{x}\right) = & \left[100\left(x_2 - x_1^2\right)^2 + \left(x_1 - 1\right)^2\right] + \left[100\left(x_3 - x_2^2\right)^2 + \left(x_2 - 1\right)^2\right] + \\
& \ldots + \left[100\left(x_n - x_{n-1}^2\right)^2 + \left(x_{n-1} - 1\right)^2\right] \\
\frac{\partial f}{\partial x_1} = & 200\left(x_2 - x_1^2\right)\left(-2x_1\right) + 2\left(x_1 - 1\right) \\
\frac{\partial f}{\partial x_2} = & 200\left(x_2 - x_1^2\right) + 200\left(x_3 - x_2^2\right)\left(-2x_2\right) + 2\left(x_2 - 1\right) \\
& \vdots \\
\frac{\partial f}{\partial x_{n-1}} = & 200\left(x_n - x_{n-1}^2\right) + 200\left(x_n - x_{n-1}^2\right)\left(-2x_{n-1}\right) + 2\left(x_{n-1} - 1\right) \\
\frac{\partial f}{\partial x_n} = & 200\left(x_n - x_{n-1}^2\right) \\
\Rightarrow \nabla \boldsymbol{f} = & \left[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right]^T
\end{aligned}
$$

```python
import numpy as np
import matplotlib.pyplot as plt

def rosen_function(X: np.ndarray) -> float:
    """
    #sum(i to i-1) [100 * (x_i+1 - x_i)^2 + (x_i - 1)^2]
    :param X: 1D array of points at which the Rosenbrock function is to be computed.
    :return: float: The value of the Rosenbrock function
    """
    rosen = 0
    for i in range (X.shape[0] - 1):
        rosen += 100 * (X[i+1] - X[i]**2.0) ** 2.0 + (X[i] - 1) ** 2.0
    return rosen


def rosen_derivative(X: np.ndarray) -> np.ndarray:
    """
    :param X: 1D array of points at which the derivative is to be computed
    :return: (N,) ndarray The gradient of the Rosenbrock function at ‘x‘.
    """
    df = [400 * X[0] * (X[0] ** 2 - X[1]) + 2 * (X[0] - 1)]
    for i in range(1, X.shape[0] - 1):
        dx = 400 * X[i] * (X[i]**2 - X[i+1]) + 2 * (X[i] - 1)
        dy = 200 * (X[i] - X[i-1] ** 2)
        df.append(dx + dy)

    df.append(200 * (X[X.shape[0] - 1] - X[X.shape[0] - 2] ** 2))
    return np.asarray(df)

def main():
    #initialization
    N = 10000
    n = 20
    learning_rate = 0.001
    X = np.linspace(0, 1, n)

    rosen = []
    y = []
    for i in range(N):
        r = rosen_function(X)
        rosen.append(r)
        df_rosen = rosen_derivative(X)
        X = X - learning_rate * df_rosen
        y.append(i)
    plt.plot(y, rosen, label='learning rate = 0.001')

    learning_rate = 0.002
    X = np.linspace(0, 1, n)
    rosen = []
    y = []
    for i in range(N):
        r = rosen_function(X)
        rosen.append(r)
        df_rosen = rosen_derivative(X)
        X = X - learning_rate * df_rosen
        y.append(i)

    plt.plot(y, rosen, label="learning rate = 0.002")

    learning_rate = 0.0001
    X = np.linspace(0, 1, n)
    rosen = []
    y = []
    for i in range(N):
        r = rosen_function(X)
        rosen.append(r)
        df_rosen = rosen_derivative(X)
        X = X - learning_rate * df_rosen
        y.append(i)

    plt.plot(y, rosen, label="learning rate = 0.0001")


    plt.title("Learning Curve of Rosenbrock function")
    plt.xlabel("Iteration")
    plt.ylabel("Learning rate")
    plt.legend()
    plt.show()

if __name__ == "__main__":
    main()
```

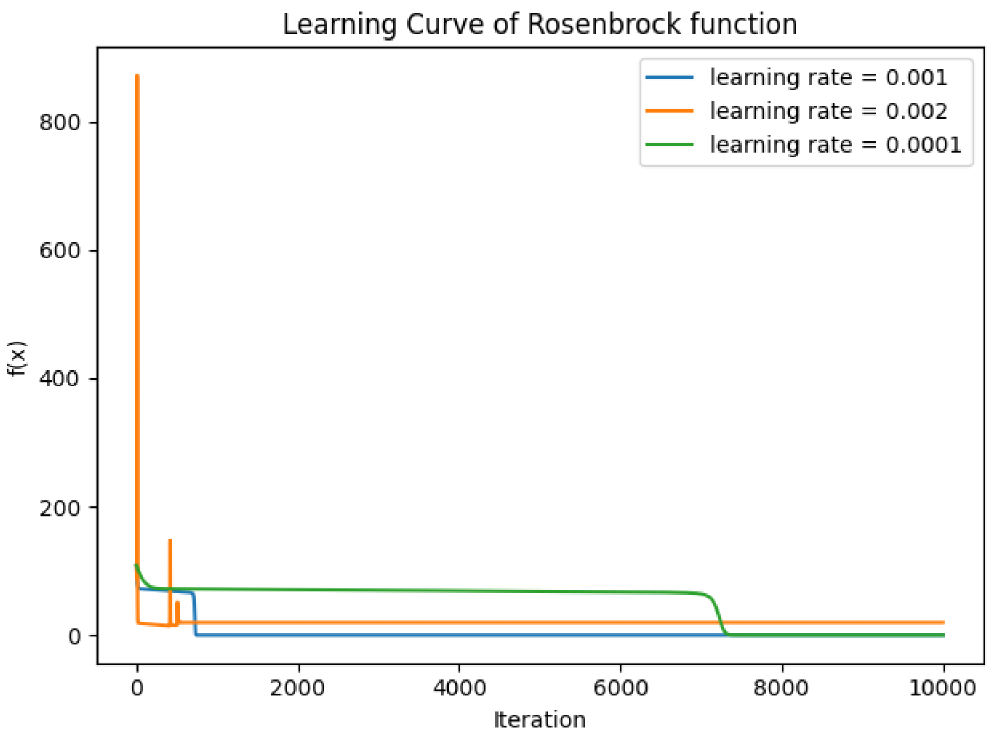


Abbildung 2: Rosenbrock function with different Learning Rate

3.4 3d)

1) **Batch Gradient descent:**

Batch gradient descent is a variation of the gradient descent algorithm that calculates the error for each example in the training dataset, but only updates the model after all training examples have been evaluated.

* **Contra:**
  - Problem with batch gradient descent in neural network is flat for every gradient descent update in the weights, you have to cycle trough every training sample, for example > 50000 training sample this can be prohibitive.
  - The more stable error gradient may result in premature convergence of the model to a less optimal set of parameters
* **Pro:**
  - Fewer updates to the model means the variant of gradient descent is more computationally efficient than stochastic gradient descent
  - Separation of the calculation of prediction errors and the model update lends the algorithm to parallel processing based implementations

Stochastic gradient descent:

Stochastic gradient descent updates the weight parameters after evaluation the cost function after each sample. That is rather than summing up the cost function results for all the sample then taking the mean, Stochastic gradient descent updates the weights after every training sample is analyzed.

* **Contra:**
  - Updating the model so frequently is more computationally expensive than other configurations of gradient descent, taking significantly longer to train models on large datasets
  - the frequent updates can result in a noisy gradient signal, which may cause the model parameters and in turn the model error to jump around (have a higher variance over training epochs)
* **Pro**
  - the frequent updates immediately give an insight into the performance of the model and the rate of improvement
  - the increased model update frequency can result in faster learning on some problems

Mini-Batch gradient descent:

Mini-Batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into mall batches that are used to calculate model error and update model coefficients.

* **Contra:**
  - Mini-Batch requires the configuration of an additional "mini-batch size"hyperparameter for the learning algorithm
  - error information must be accumulated across mini-batches of training examples like batch gradient descent
* **Pro:**
  - The model update frequency is higher than batch gradient descent which allows for a more robust convergence, avoiding local minima
  - the batched updates provide a computationally more efficient process than stochastic gradient descent

Tutor solution:[4]

* Batch: it uses all data available to learn.
  - high computational cost per iteration
  - not suitable for online learning
  + low number of epochs required to find the final parameters
  + less variance in the parameter updates
* Stochastic: It uses a single sample from the training set.
  - high variance in the parameter updates, update step may övershoot"
  - convergence is less straightforward
  - depending on the problem, the oder of the training data becomes important (random shuffing can alleviate this problem)
  + low computational cost per iteration
  + suitable for online learing

[4]Complete overview: https://ruder.io/optimizing-gradient-descent/

* Mini-batch: it uses small batches of data to learn.
  - size of the mini-batch is an addiational hyper-parameter
  - depending on the problem, the oder of the training data becomes important (random shuffing can alleviate this problem)
  + a good compromise between computational cost per iteration and variance
  + still suitable for online learning

2)

Momentum or Stochastic Gradient descent with momentum is a popular method which helps accelerate gradients vectors in the right directions, thus leading to faster converging. The Stochastic Gradient descent with momentum helps us to update equations of the algorithm before we jumping over the steepest descent. If the momentum and gradients points look at the same directions, than it reduces updates for dimensions. As result, it convergence faster and it reduces the oscillations to get to the local minimum.

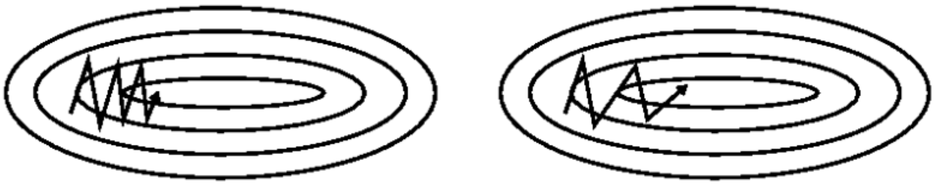


Abbildung 3: Left: SGD without momentum, right: SGD with momentum

Figure Source: https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d

- Tutor solution:

  The momentum is a term proportional of the "current velocity in the parameter space", added to the weight update in order to speed up the convergence and suppress oscillation in the weight updates. Formally,

$$v_i = \alpha v_{i-1} + \eta \nabla_x f(\vec{x})$$
$$\vec{x}_{i+1} = \vec{x}_i - v_i$$

  In practice, the momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscilation. The scalar scaling factor $\alpha \in [0, 1]$ determines how beg the impact of the previous changes in the parameters is w.r.t the current parameter update.

Literatur

[1] Statistical Machine Learning
Lecture course by Prof. Dr. Kristian Kersting

[2] Wikipedia - Kullback-Leibler divergence
https://en.wikipedia.org/wiki/Kullback-Leibler_divergence

[3] Convex Optimization for Signal Processing and Communications
Lecture course by Prof. Dr.-Ing. Marius Pesavento