Modelling and Analysis of Data

Datalogisk institut, Copenhagen University (DIKU)
Exam 2024

Exam. No.
`KU-ID: ?`

November 27, 2023.

Contents

1 Statistics **1**

2 Principal component analysis **4**

3 kNN **9**

4 Classification **11**

5 Clustering **20**

6 Appendix **31**

6.1 Full Code for Question 2 31

6.2 Full Code for Question 4 31

6.3 Full Output for Question 4 35

6.4 Full Code for Question 6 35

6.5 Full Output for Question 6 39

6.6 Full Code for Question 7 41

1 Statistics

Question 1

We are given the following continous random variable X with the following probability density function:

$$f(x;\theta) = \begin{cases} \frac{\theta}{x} \cdot \exp\left(-\frac{\theta}{x^2}\right) & \text{for } x \in \mathbb{R}_+, \\ 0 & \text{otherwise.} \end{cases} \tag{1.1}$$

We are asked to prove the maximum likelihood estimator for $\hat{\theta}$ is given by:

$$\hat{\theta} = \frac{N}{\frac{1}{x_1^2} + \frac{1}{x_2^2} + \cdots + \frac{1}{x_N^2}}. \tag{1.2}$$

We start by writing the likelihood function:

$$\mathcal{L}(\theta) = \prod_{i=1}^{N} \frac{\theta}{x_i} \cdot \exp\left(-\frac{\theta}{x_i^2}\right).$$

The goal of maximum likelihood estimation is to find the value of the model parameters that maximizes the likelihood function over the parameter space:

$$\hat{\theta} = \operatorname{argmax}_\theta \mathcal{L}(\theta) = \operatorname{argmax}_\theta \prod_{i=1}^{N} \frac{\theta}{x_i} \cdot \exp\left(-\frac{\theta}{x_i^2}\right). \tag{1.3}$$

We then take the logarithm of the likelihood function:

$$\begin{aligned} \ell(\theta; y) &= \ln \mathcal{L}(\theta) \\ &= \sum_{i=1}^{N} \ln\left(\frac{\theta}{x_i} \cdot \exp\left(-\frac{\theta}{x_i^2}\right)\right) \\ &= \sum_{i=1}^{N} \ln\left(\frac{\theta}{x_i}\right) \cdot \ln\left(\exp\left(-\frac{\theta}{x_i^2}\right)\right) \\ &= \sum_{i=1}^{N} \ln\left(\frac{\theta}{x_i}\right) - \sum_{i=1}^{N} \frac{\theta}{x_i^2} \end{aligned}$$

Apply the logarithmic property $\ln(a/b) = \ln(a) - \ln(b)$:

$$= \sum_{i=1}^{N} \ln\theta - \ln x_i - \sum_{i=1}^{N} \frac{\theta}{x_i^2}.$$

So, the simplified logarithm of the likelihood function is:

$$\ln \mathcal{L}(\theta) = \sum_{i=1}^{N} \ln\theta - \ln x_i - \frac{\theta}{x_i^2}.$$

We then take the derivative of the logarithm of the likelihood function with respect to θ, $\frac{\partial}{\partial\theta}\ln(\theta) = \frac{1}{\theta}$, $\frac{\partial}{\partial\theta}\ln(x_i) = 0$, and $\frac{\partial}{\partial\theta}\frac{\theta}{x_i^2} = \frac{1}{x_i^2}$, so:

$$\frac{\partial}{\partial\theta}\ln\mathcal{L}(\theta) = \sum_{i=1}^{N}\frac{1}{\theta} - \frac{1}{x_i^2}.$$

We then set the derivative of the logarithm of the likelihood function with respect to θ equal to zero:

$$\sum_{i=1}^{N}\frac{1}{\theta} - \frac{1}{x_i^2} = 0.$$

We then solve for θ:

$$\begin{aligned}\sum_{i=1}^{N}\frac{1}{\theta} &= \sum_{i=1}^{N}\frac{1}{x_i^2}\\ \frac{N}{\theta} &= \sum_{i=1}^{N}\frac{1}{x_i^2}\\ N &= \left(\sum_{i=1}^{N}\frac{1}{x_i^2}\right)\theta\\ \theta &= \frac{N}{\sum_{i=1}^{N}\frac{1}{x_i^2}} \qquad (1.4)\end{aligned}$$

We know that $\sum_{i=1}^{n}\frac{1}{x_i^2} = \frac{1}{x_1^2} + \frac{1}{x_2^2} + \cdots + \frac{1}{x_N^2}$ and from (1.3) we know $\hat{\theta} = \operatorname{argmax}_\theta \mathcal{L}(\theta)$, meaning that $\hat{\theta}$ is the value of θ that maximizes the likelihood function, thus we can rewrite equation 1.4 as:

$$\hat{\theta} = \frac{N}{\frac{1}{x_1^2} + \frac{1}{x_2^2} + \cdots + \frac{1}{x_N^2}}.$$

Now we have proven the maximum likelihood estimator for $\hat{\theta}$ is given by:

$$\hat{\theta} = \frac{N}{\frac{1}{x_1^2} + \frac{1}{x_2^2} + \cdots + \frac{1}{x_N^2}}. \qquad (1.5)$$

.

Question 2

We're given a normal distributed random variable X with mean μ and variance $\sigma^2 = 0.1$, i.e., $X \sim \mathcal{N}(\mu, \sigma^2 = 0.1)$ and mu is unknown. We're given a sample from X of $n = 11$ observations as follows:

$$\{8.1, 7.5, 8.7, 8.3, 8.5, 8.2, 8.9, 8.2, 8.7, 7.6, 8.5\}.$$

I am to perform a hypothesis test given the following:

1. **Null hypothesis**: $H_0 \ : \ \mu \geq 8.5$
2. **Alternative hypothesis**: $H_A \ : \ \mu < 8.5$
3. **Significance level**: 5%
4. **Population variance**: $\sigma^2 = 0.1$
5. **Sample data** $X\{8.1, 7.5, 8.7, 8.3, 8.5, 8.2, 8.9, 8.2, 8.7, 7.6, 8.5\}$

I will use the 6 steps of hypothesis testing from the lecture slides *L6_ advanced_ stats.pdf*. The sample mean $\bar{X}$ and sample standard derivation S are defined as:

$$\begin{aligned}\bar{X} &= \frac{1}{N}\sum_{i=1}^{n}(X_i) \\ &= \frac{8.1+7.5+8.7+8.3+8.5+8.2+8.9+8.2+8.7+7.5+8.5}{11} \\ &= 8.290909\end{aligned}$$

$$S = \sqrt{\frac{1}{n-1}\sum_{x=1}^{n}(X_i - \bar{X})^2}$$

Calculating the sum $\sum_{i=1}^{n}(X_i - \bar{X})^2 = (-0.19)^2 + (-0.79)^2 + 2(0.41)^2 + (0.009)^2 + (0.21)^2 + 2(-0.09)^2 + (0.61)^2 + (-0.69)^2 + (0.21)^2 = 1.949081$

$$\begin{aligned}&= \sqrt{\frac{0.1949081}{10}} \\ &= 0.441484\end{aligned}$$

Now that we have our sample mean $\bar{X}$ and sample standard derivation S, we can calculate the z-test defined as:

$$z = \frac{\bar{X} - \mu_0}{\frac{\sigma}{\sqrt{n}}}$$

where μ_0 is the hypothesized population mean under the null hypothesis, σ^2 is the population variance, and n is the sample size. The hypothesized population mean under the null hypothesis will be $\mu_0 = 8.5$, since the null hypothesis is given by $H_0 \ : \ \mu \geq 8.5$. We can now calculate the z-test:

$$z = \frac{8.290909 - 8.5}{\frac{\sqrt{0.1}}{\sqrt{11}}} = -2.19296$$

We can now calculate the p-value using the z-test and the cumulative distribution function (CDF) of the standard normal distribution $\Phi(z)$, for this we use the Python function `scipy.stats.norm.cdf`:

```
p = stats.norm.cdf(-2.19296)
```

And we get the following p-value:

$$p = \Phi(z) = \Phi(-2.19296) = 0.014155130578803 67$$

We can now compare the p-value to the significance level $\alpha = 0.05$:

1. If $p \leq \alpha$, then we reject the null hypothesis.
2. If $p > \alpha$, then we fail to reject the null hypothesis.

Since $p = 0.01415513057880367$ and $\alpha = 0.05$, then $p \leq \alpha$, so we reject the null hypothesis $H_0 : \mu \geq 8.5$ in favor of the alternative hypothesis $H_A : \mu \leq 8.5$. This suggests that the true mean of the population from which the sample was drawn is likely less than 8.5.

2 Principal component analysis

Question 3

We have been given a 2-dimensional points set $x_1, x_2, \ldots, x_N$ where $N = 6$. We are first asked to compute the y-values of the points set: The second dimension y is defined as $y = \dfrac{2x}{4-x}$. Mathematically, we can write the y-values of the points set as:

$$y_i = \frac{2x_i}{4 - x_i}. \tag{2.1}$$

We can then compute the y-values of the points set:

$$\begin{aligned}
y_1 &= \frac{2 \cdot 0.5}{4 - 0.5} = 0.3, \\
y_2 &= \frac{2 \cdot 1.1}{4 - 1.1} = 0.75, \\
y_3 &= \frac{2 \cdot -0.7}{4 - (-0.7)} = -0.29, \\
y_4 &= \frac{2 \cdot 1.5}{4 - 1.5} = 1.2, \\
y_5 &= \frac{2 \cdot -1.2}{4 - (-1.2)} = -0.46, \\
y_6 &= \frac{2 \cdot 0.9}{4 - 0.9} = 0.58.
\end{aligned}$$

So, the x-values and y-values of the points set are given in the following table (rounded to two decimals) where the x-values are in the first row and the y-values are in the second row:
We are asked to find the principal components of the points set. We start by finding the mean of the points set:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i = \frac{1}{6} \cdot (0.5 + 1.1 - 0.7 + 1.5 - 1.2 + 0.9) = 0.35. \tag{2.2}$$

Table 1: Points

coordinate x	0.5	1.1	-0.7	1.5	-1.2	0.9
coordinate y	0.3	0.75	-0.29	1.2	-0.46	0.58

We then center the points set by subtracting the mean from each point for x (i.e. $x_i - \bar{x}$):

$$\begin{aligned} x_1 &= 0.5 - 0.35 = 0.15, \\ x_2 &= 1.1 - 0.35 = 0.75, \\ x_3 &= -0.7 - 0.35 = -1.05, \\ x_4 &= 1.5 - 0.35 = 1.15, \\ x_5 &= -1.2 - 0.35 = -1.55, \\ x_6 &= 0.9 - 0.35 = 0.55. \end{aligned}$$

Subsequentially, we center the points set by subtracting the mean from each point for y (i.e. $y_i - \bar{y}$): To start we compute the mean of the y-values:

$$\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i = \frac{1}{6} \cdot (0.3 + 0.75 - 0.29 + 1.2 - 0.46 + 0.58) \approx 0.35. \tag{2.3}$$

$$\begin{aligned} y_1 &= 0.3 - 0.35 = -0.05, \\ y_2 &= 0.75 - 0.35 = 0.4, \\ y_3 &= -0.29 - 0.35 = -0.64, \\ y_4 &= 1.2 - 0.35 = 0.85, \\ y_5 &= -0.46 - 0.35 = -0.81, \\ y_6 &= 0.58 - 0.35 = 0.23. \end{aligned}$$

The covariance $Cov(x, y)$ is then given by:

$$\Sigma = Cov(x, y) = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y}). \tag{2.4}$$

We then compute the covariance:

$$\begin{aligned} \Sigma &= \frac{1}{5} \cdot ((0.15 \cdot -0.05) + (0.75 \cdot 0.4) + (-1.05 \cdot -0.64) + (1.15 \cdot 0.85) + (-1.55 \cdot -0.81) + (0.55 \cdot 0.23)) \\ &= \frac{1}{5} \cdot (-0.0075 + 0.3 + 0.672 + 0.9775 + 1.2555 + 0.1265) \\ &= \frac{1}{5} \cdot 3.3245 \\ &= 0.6649 \end{aligned}$$

We then compute the variance for x:

$$
\begin{aligned}
Var(x) = Cov(x,x) &= \frac{1}{N-1}\sum_{i=1}^{N}(x_i-\bar{x})(x_i-\bar{x}) = \frac{1}{N-1}\sum_{i=1}^{N}(x_i-\bar{x})^2 \\
&= \frac{1}{5}\cdot 0.15^2 + 0.75^2 + (-1.05)^2 + 1.15^2 + (-1.55)^2 + 0.55^2 \\
&= \frac{1}{5}\cdot 0.0225 + 0.5625 + 1.1025 + 1.3225 + 2.4025 + 0.3025 \\
&= \frac{1}{5}\cdot 5.715 \\
&= 1.143.
\end{aligned}
$$

We then compute the variance for y:

$$
\begin{aligned}
Var(y) = Cov(y,y) &= \frac{1}{N-1}\sum_{i=1}^{N}(y_i-\bar{y})(y_i-\bar{y}) = \frac{1}{N-1}\sum_{i=1}^{N}(y_i-\bar{y})^2 \\
&= \frac{1}{5}\cdot(-0.05)^2 + 0.4^2 + (-0.64)^2 + 0.85^2 + (-0.81)^2 + 0.23^2 \\
&= \frac{1}{5}\cdot 0.0025 + 0.16 + 0.4096 + 0.7225 + 0.6561 + 0.0529 \\
&= \frac{1}{5}\cdot 2.0036 \\
&= 0.40072.
\end{aligned}
$$

The formula for the covariance is matrix is $\Sigma = cov(r) = \frac{1}{N-1}\sum_{i=1}^{N} pp^T$, but in our case of 2 dimensional points we can use the following:

$$
\Sigma = \begin{bmatrix} Var(x) & Cov(x,y) \\ Cov(y,x) & Var(y) \end{bmatrix} \tag{2.5}
$$

$$
= \begin{bmatrix} 1.143 & 0.6649 \\ 0.6649 & 0.40072 \end{bmatrix} \tag{2.6}
$$

To find eigenvectors and eigenvalues, we first must solve $\det(\Sigma - \lambda I) = 0$:

$$
\begin{aligned}
\det(\Sigma-\lambda I) &= \det\begin{bmatrix} 1.143-\lambda & 0.6649 \\ 0.6649 & 0.40072-\lambda \end{bmatrix} \\
&= (1.143-\lambda)(0.40072-\lambda) - 0.6649^2 \\
&= 1.143(0.40072) - 1.143\lambda - \lambda 0.40072 + \lambda^2 - 0.44209201 \\
&= 0.45802296 - 1.143\lambda - \lambda 0.40072 + \lambda^2 - 0.44209201 \\
&= \lambda^2 - 1.54372\lambda + 0.01593095
\end{aligned}
$$

To solve this polynomial equation, we use the quadratic formula, by first computing the discriminant d:

$$
d = b^2 - 4ac = (-1.54372)^2 - 4\cdot 1\cdot 0.01593095 = 2.3193476384
$$

Then we can solve the quadratic equation:

$$\begin{aligned}\lambda &= \frac{-b \pm \sqrt{d}}{2a} \\ &= \frac{1.54372 \pm \sqrt{2.3193476384}}{2} \\ &= \frac{1.54372 \pm 1.5229404579}{2} \\ &= \begin{cases} 1.53333023 & \text{or} \\ 0.01038977 \end{cases}\end{aligned}$$

So, the following two eigenvalues are found to be:

$$\lambda_1 = 1.53333023 \text{ and } \lambda_2 = 0.01038977. \tag{2.7}$$

Now I can proceed to find the eigenvectors. To find the eigenvectors, we find $\Sigma - \lambda_1 I$ for each eigenvalue, starting with $\lambda_1 = 1.53333023$.

$$\begin{aligned}\Sigma - \lambda_1 I &= \begin{bmatrix} 1.143 - \lambda_1 & 0.6649 \\ 0.6649 & 0.40072 - \lambda_1 \end{bmatrix} \\ &= \begin{bmatrix} 1.143 - 1.53333023 & 0.6649 \\ 0.6649 & 0.40072 - 1.53333023 \end{bmatrix} \\ &= \begin{bmatrix} -0.390330 & 0.6649 \\ 0.6649 & -1.132610 \end{bmatrix}\end{aligned}$$

Now we can compute the equation $(\Sigma - \lambda_1 I)x = 0$:

$$\begin{bmatrix} -0.390330 & 0.6649 \\ 0.6649 & -1.132610 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 0$$

This is a system of equations can can be solved by firstly isolating x_1 from the second equation and then substitute into the first equation to isolate for x_2:

$$\begin{aligned}-0.390330x_1 + 0.6649x_2 &= 0 \\ 0.6649x_1 - 1.132610x_2 &= 0\end{aligned}$$

Isolating for x_1 in second equation:

$$\begin{aligned}0.6649x_1 &= 1.132610x_2 \\ x_1 &= \frac{1.132610x_2}{0.6649} \\ x_1 &= 1.703429x_2\end{aligned}$$

I insert x_1 in first equation:

$$\begin{aligned}-0.390330 \cdot 1.703429x_2 + 0.6649x_2 &= 0 \\ -0.6649x_2 + 0.6649x_2 &= 0 \\ -0.6649x_2 &= -0.6649x_2\end{aligned}$$

This means, that there are infinite solutions, so x_2 can be any value. Thus, $x_1 = 1.703429x_2$ and $x_2 = x_2$. I choose x_2 to be 1, so we end up with the eigenvector, which we have to normalize:

$$\begin{aligned} v_{\lambda_1} &= \text{norm}(\begin{bmatrix} 1.703429 \\ 1 \end{bmatrix}) \\ &= \frac{1}{\sqrt{1.703429^2 + 1^2}} \cdot \begin{bmatrix} 1.703429 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0.86238 \\ 0.506261 \end{bmatrix} \end{aligned}$$

The same procedure will then be done to compute the eigenvectors for $\lambda_2 = 0.01038977$. So, we start by computing $\Sigma - \lambda_2 I$ where $\lambda_2 = 0.01038977$.

$$\begin{aligned} \Sigma - \lambda_1 I &= \begin{bmatrix} 1.143 - \lambda_1 & 0.6649 \\ 0.6649 & 0.40072 - \lambda_1 \end{bmatrix} \\ &= \begin{bmatrix} 1.143 - 0.01038977 & 0.6649 \\ 0.6649 & 0.40072 - 0.01038977 \end{bmatrix} \\ &= \begin{bmatrix} 1.132610 & 0.6649 \\ 0.6649 & 0.390330 \end{bmatrix} \end{aligned}$$

Now we can compute the equation $(\Sigma - \lambda_1 I)x = 0$:

$$\begin{bmatrix} 1.132610 & 0.6649 \\ 0.6649 & 0.390330 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 0$$

System of equations:

$$\begin{aligned} 1.132610x_1 + 0.6649x_2 &= 0 \\ 0.6649x_1 + 0.390330x_2 &= 0 \end{aligned}$$

Isolating for x_1 in second equation:

$$\begin{aligned} 0.6649x_1 &= -0.390330x_2 \\ x_1 &= \frac{-0.390330x_2}{0.6649} \\ x_1 &= -0.587051x_2 \end{aligned}$$

Insert x_1 in first equation:

$$\begin{aligned} 1.132610 \cdot -0.587051x_2 + 0.6649x_2 &= 0 \\ -0.6649x_2 + 0.6649x_2 &= 0 \\ -0.6649x_2 &= -0.6649x_2 \end{aligned}$$

This is the same case before, so we follow procedure, where we choose x_2 to be

1 and $x_1 = -0.587051x_2$ and normalize:

$$\begin{aligned} v_{\lambda_2} &= \text{norm}(\begin{bmatrix} -0.587051 \\ 1 \end{bmatrix}) \\ &= \frac{1}{\sqrt{1.587051^2 + 1^2}} \cdot \begin{bmatrix} -0.587051 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} -0.506261 \\ 0.86238 \end{bmatrix} \end{aligned}$$

Thus, we have found the two eigenvectors and their respective eigenvalues to be:

$$\begin{aligned} v_{\lambda_1} &= \begin{bmatrix} 0.86238 \\ 0.506261 \end{bmatrix} \text{ and } \lambda_1 = 1.53333023 \\ v_{\lambda_2} &= \begin{bmatrix} -0.506261 \\ 0.86238 \end{bmatrix} \text{ and } \lambda_2 = 0.01038977 \end{aligned}$$

3 kNN

Question 4

For this task we are to test our understanding of methods for regression. We're provided with a database of patients with and without a heart disease, which is seperated into a training set and a test set. The training set is used to train a kNN classifier, and the test set is used to test the accuracy of the classifier.

The dataset consists of 6 features, four of which are numerical and two of which are categorical. The last column of the dataset is the target value, which is either 0 or 1 and represents the HeartDisease attribute.

The task at hand is to predict the predict manifestation of the heart disease using patient features, which can be solved using neighbor classifier. For a vector x, the prediction is given via

$$f(x) = round(\frac{1}{k} \sum_{x_n \in N_K(x)} t_n) \tag{3.1}$$

where $N_k(x)$ denotes the set of the $k \geq 1$ nearest neighbors of x in the set $\{x_1, ..., x_N\}$ of training points.

1) RMSE and accuracy value for the optimal k and scatter plot with the dependency between the accuracy and k.

I have implemented nearest neighbor classification for $k \geq 1$ in the Python file `question4.py` and the results from running the program, i.e., the output are shown in the table below (Table 2) for clarity. For the actual output refer to the Appendix 6.3.

k	RMSE	Accuracy
1	0.6082762530298219	0.63
3	0.5830951894845301	0.66
5	0.5477225575051661	0.7
7	0.5099019513592785	0.74
9	0.5385164807134504	0.71

Table 2: RMSE and accuracy for different values of k

It is quite clear that the optimal value of k is 7, as this gives the highest accuracy and the lowest RMSE. The scatter plot in Figure 1 shows the dependency between the accuracy and k.

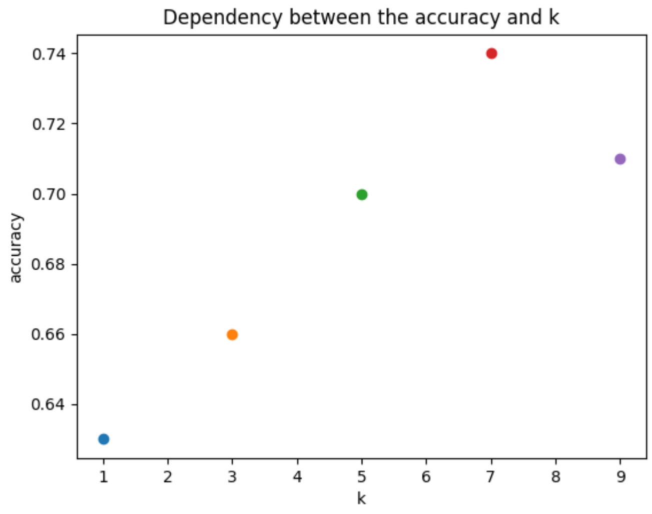

Figure 1: Accuracy for different values of k

2) RMSE and accuracy value for the w_{cat} values of interest mentioned in the task.

For the second task, I am to further extend the Python implementation to handle both numerical and categorical features. The combined distance will in this case be calculated as:

$$D(p,q) = w_{num} \sum_{I \in I} d_{num}(p_i, q_i) + w_{cat} \sum_{J \in J} d_{cat}(p_j, q_j) \tag{3.2}$$

where w_{num} and w_{cat} are the weight factors multiplied to numerical and categorical features. The results or output of our newly implementation for handling categorical and numerical features is shown in the table below (Table 3) with different values for w_{cat}, and the actual full output is still shown in Appendix 6.3.

w_{cat}	RMSE	Accuracy
0.025	0.5477225575051661	0.7
0.05	0.5099019513592785	0.74
0.1	0.458257569495584	0.79

Table 3: RMSE and accuracy for different values of w_{cat}

It is quite clear that the optimal value of w_{cat} is 0.1, as this gives the highest accuracy and the lowest RMSE.

4 Classification

Question 5

Combination of a), b) and c).

Our goal is to compute the optimal threshold for separating the following data into two subsets using information gain as a metric. We have been given a training set T on the following form $T = \{(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), \ldots, (\mathbf{x_n}, y_n)\}$; where $\mathbf{x} \in \{10, 20, 30, \ldots, 110\}$ is the feature, and $\mathbf{y} \in \{0, 1, 2\}$; is the label. The data is given in the following table:

Table 4: Training set

feature	10	20	30	40	50	60	60	70	80	90	100	110
label	1	0	0	2	1	1	0	0	0	0	1	0

This next section answers a) The formula for entropy is given by:

$$H(T) = -\sum_{i=1}^{n} p_i \log_2 p_i. \tag{4.1}$$

We'll compute the entropy for the three classes on the entire training set T:

$$H(T) = -\frac{7}{12}\log_2\frac{7}{12} - \frac{4}{12}\log_2\frac{4}{12} - \frac{1}{12}\log_2\frac{1}{12} \approx 1.281.$$

This next section answers b) The formula for split points is given by:

$$\text{split points} = x_i + \frac{x_{i+1} - x_i}{2} : y_{i+1} \neq x_i. \tag{4.2}$$

We start by computing the split points for the training set T:

$$\begin{aligned}
10 + \frac{20-10}{2} &= 15, \\
20 + \frac{30-20}{2} &= 25, \\
30 + \frac{40-30}{2} &= 35, \\
40 + \frac{50-40}{2} &= 45, \\
50 + \frac{60-50}{2} &= 55, \\
60 + \frac{70-60}{2} &= 65, \\
70 + \frac{80-70}{2} &= 75, \\
80 + \frac{90-80}{2} &= 85, \\
90 + \frac{100-90}{2} &= 95, \\
100 + \frac{110-100}{2} &= 105.
\end{aligned}$$

These are all the split points for the training set T, now we will correctly identify the potential thresholds. The potential thresholds are usually the midpoints between consecutive values where the label changes, hence why we are only interested in the midpoints where the label changes. This is because only at these points does a change in the threshold potentially impact the split of the data in a way that might increase information gain. The potential thresholds are then:

$$\text{potential thresholds} = \{15, 35, 45, 95, 105\}. \tag{4.3}$$

This next section answers c) Now we will split the data into two groups. One group should contain all the samples with feature values less than the threshold. The other group should contain the rest. And then we will compute the entropy for each group.
First split point is 15:

1. **Group 1**: (10, 1)
2. **Group 2**: (20, 0), (30, 0), (40, 2), (50, 1), (60, 1), (60, 0), (70, 0), (80, 0), (90, 0), (100, 1), (110, 0)

We'll compute the entropy for the two groups:

$$\begin{aligned}
H(T_1) &= -\frac{1}{1}\log_2\frac{1}{1} = 0, \\
H(T_2) &= -\frac{7}{11}\log_2\frac{7}{11} - \frac{3}{11}\log_2\frac{3}{11} - \frac{1}{11}\log_2\frac{1}{11} \approx 1.241.
\end{aligned}$$

We'll compute the information gain for the split point 15:

$$\begin{aligned}IG(T,15) &= H(T) - \frac{|T_1|}{|T|}H(T_1) - \frac{|T_2|}{|T|}H(T_2)\\ &= 1.281 - \frac{1}{12}\cdot 0 - \frac{11}{12}\cdot 1.241\\ &\approx 0.143.\end{aligned}$$

Second split point is 35:

1. **Group 1**: (10, 1), (20, 0), (30, 0)
2. **Group 2**: (40, 2), (50, 1), (60, 1), (60, 0), (70, 0), (80, 0), (90, 0), (100, 1), (110, 0)

We'll compute the entropy for the two groups:

$$\begin{aligned}H(T_1) &= -\frac{1}{3}\log_2\frac{1}{3} - \frac{2}{3}\log_2\frac{2}{3} = 0.918,\\ H(T_2) &= -\frac{5}{9}\log_2\frac{5}{9} - \frac{3}{9}\log_2\frac{3}{9} - \frac{1}{9}\log_2\frac{1}{9} \approx 1.351.\end{aligned}$$

We'll compute the information gain for the split point 35:

$$\begin{aligned}IG(T,35) &= H(T) - \frac{|T_1|}{|T|}H(T_1) - \frac{|T_2|}{|T|}H(T_2)\\ &= 1.281 - \frac{3}{12}\cdot 0.918 - \frac{9}{12}\cdot 1.351\\ &\approx 0.03825.\end{aligned}$$

Third split point is 45:

1. **Group 1**: (10, 1), (20, 0), (30, 0), (40, 2)
2. **Group 2**: (50, 1), (60, 1), (60, 0), (70, 0), (80, 0), (90, 0), (100, 1), (110, 0)

We'll compute the entropy for the two groups:

$$\begin{aligned}H(T_1) &= -\frac{2}{4}\log_2\frac{2}{4} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{4}\log_2\frac{1}{4} = 1.5,\\ H(T_2) &= -\frac{5}{8}\log_2\frac{5}{8} - \frac{3}{8}\log_2\frac{3}{8} = 0.954.\end{aligned}$$

We'll compute the information gain for the split point 45:

$$\begin{aligned}IG(T,45) &= H(T) - \frac{|T_1|}{|T|}H(T_1) - \frac{|T_2|}{|T|}H(T_2)\\ &= 1.281 - \frac{4}{12}\cdot 1.5 - \frac{8}{12}\cdot 0.954\\ &\approx 0.145.\end{aligned}$$

Fourth split point is 95:

1. **Group 1**: (10, 1), (20, 0), (30, 0), (40, 2), (50, 1), (60, 1), (60, 0), (70, 0), (80, 0), (90, 0)
2. **Group 2**: (100, 1), (110, 0)

We'll compute the entropy for the two groups:

$$\begin{aligned}
H(T_1) &= -\frac{6}{10}\log_2\frac{6}{10} - \frac{3}{10}\log_2\frac{3}{10} - \frac{1}{10}\log_2\frac{1}{10} \approx 1.295,\\
H(T_2) &= -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1.
\end{aligned}$$

We'll compute the information gain for the split point 95:

$$\begin{aligned}
IG(T, 95) &= H(T) - \frac{|T_1|}{|T|}H(T_1) - \frac{|T_2|}{|T|}H(T_2)\\
&= 1.281 - \frac{10}{12}\cdot 1.295 - \frac{2}{12}\cdot 1\\
&\approx 0.0351.
\end{aligned}$$

Final split point is 105:

1. **Group 1**: (10, 1), (20, 0), (30, 0), (40, 2), (50, 1), (60, 1), (60, 0), (70, 0), (80, 0), (90, 0), (100, 1)
2. **Group 2**: (110, 0)

We'll compute the entropy for the two groups:

$$\begin{aligned}
H(T_1) &= -\frac{6}{11}\log_2\frac{6}{11} - \frac{4}{11}\log_2\frac{4}{11} - \frac{1}{11}\log_2\frac{1}{11} \approx 1.322,\\
H(T_2) &= -\frac{1}{1}\log_2\frac{1}{1} = 0.
\end{aligned}$$

We'll compute the information gain for the split point 105:

$$\begin{aligned}
IG(T, 105) &= H(T) - \frac{|T_1|}{|T|}H(T_1) - \frac{|T_2|}{|T|}H(T_2)\\
&= 1.281 - \frac{11}{12}\cdot 1.322 - \frac{1}{12}\cdot 0\\
&\approx 0.069.
\end{aligned}$$

Now we have computed the information gain for all the potential thresholds. The threshold with the highest information gain is 45. So, the optimal threshold for separating the data into two subsets is 45. A table below shows the information gain for all the potential thresholds:
We can see that the threshold 15 and 45 offer nearly equivalent information gain, but 45 is the optimal threshold.

Table 5: Information gain

threshold	15	35	45	95	105
IG	0.143	0.038	0.145	0.035	0.069

Question 6

a) Convert the categorical features into numerical features.

The code below (see Code §4.1, p. 15) converts the categorical features into numerical features. The code is also available in the file `utils.py`.

Code §4.1: Conversion: Categorical Features to Numerical Features

```python
 1 def from_categorical_to_numerical(X):
 2     X[:, 5] = np.where(X[:, 5] == 'M', Gender.MALE.value, Gender.
          FEMALE.value)
 3     chest_pain_categories = np.unique(X[:, 6])
 4     last_column = X[:, -1].copy()
 5     X = np.delete(X, -1, axis=1)
 6     for category in chest_pain_categories:
 7         one_hot_column = (X[:, 6] == category).astype(int)
 8         X = np.hstack((X, one_hot_column.reshape(-1, 1)))
 9     X = np.delete(X, 6, axis=1)
10     X = np.hstack((X, last_column.reshape(-1, 1)))
11     return X
```

The second line converts the categorical feature `'Sex'` into a numerical binary feature ($x \in \{0, 1\}$), where `'M'` and `'F'` are represented as 1 and 0, respectively. The third line uses `np.unique` function to identify the unique values in the `'ChestPainType'` column.
The fourth line saves the last column `'HeartDisease'` and the fifth line removes it temporarily from the dataset.
Lines six through eight iterate through the unique categories identified in the `'ChestPainType'` column. For each category, the line

```
(X[:, 6] == category).astype(int)
```

generates a boolean array indicating if each entry matches the current category. This array is then converted to integers with `astype(int)`, changing True values to 1 and False values to 0, and the resulting One-Hot Encoded column is added to `X`.
Line 9 deletes the original categorical `'ChestPainType'` column.
Line 10 adds back the `'HeartDisease'` column, ensuring that the target variable remains correctly positioned at the end of the dataset.
Below is a table displaying the dataset before and after the One-Hot Encoding of the `'ChestPainType'` column, not in the right order, but for the sake of clarity.
An accurate representation of the data seen in the table above is visualized below in the form of an array.

```
[id  0.59 0.65 0.34 0.25 1 1 0 0 0 0]
```

Before Conversion							After Conversion			
Age	BP	Chol	HR	Sex	CP Type	HD	CP_ASY	CP_ATA	CP_NAP	CP_TA
0.59	0.65	0.34	0.25	M	ASY	0	1	0	0	0

b) Implement random forest classifier.

The code below (see Code §4.2, p. 16) implements a random forest classifier. The code between lines 1 and 3 defines a function that calculates the accuracy of a predictor given features X from a dataset and the target variables t. The code between lines 4 and 8 defines a function that instantiates a random forest classifier class given features X from a dataset and the target variables t, for later use it also takes arguments such as, the number of estimators, the criterion, the maximum features, and the maximum depth. If not specified, the default values are used. We then fit the model on the features X and the target variables t provided. The code between lines 9 and 11 utilizes both our newly created methods by first initializing the random forest classifier with the default values and then then prints the accuracy of the model.

Code §4.2: Random Forest:

```
1  def accuracy(predictor, X, t):
2      predictions = predictor.predict(X)
3      return np.sum(predictions == t)/len(t)
4  def random_forests_init(X, t, n_estimators=100, criterion="gini",
       max_features="sqrt", max_depth=None):
5      predictor = RandomForestClassifier(n_estimators=n_estimators,
           criterion=criterion, max_depth=max_depth,
6                                       max_features=max_features)
7      predictor.fit(X, t)
8      return predictor
9  predictor = random_forests_init(X_train, t_train)
10 precision = accuracy(predictor, X_train, t_train)
11 print("Accuracy on training data: ", precision)
```

The output of the code above is:

```
Accuracy on training data:  1.0
```

This makes sense since we are using the same data to train and test the model which the exercise explicitly states to use the train csv file and test the accuracy on the training data. So, it's no big surprise that the accuracy is 1.0, i.e., 100%, since the model has already seen the data and is able to predict it perfectly.

c) Optimal set of random forest classifier parameters.

There are given the three following parameters to test:

1. **Criterion**: Entropy or gini.
2. **Maximal Tree Depth**: 2, 5, 7, 10, 15
3. **The Number of Features**: `sqrt`, `log2` of the total number of features.

To find the optimal set of parameters, we shall try 15 randomly generated options for criterion, tree depth and number of features split rule. Each option we will calculate and print two metrics:

1. Number of correctly classified validation samples
2. The average probability assigned to the correct class for all validation samples

Let's start by looking at our `compute_metrics` function (see Code §4.3, p. 18). We start by predicting the classes of the validation set using the `predict` method of the predictor object. We then calculate the probabilities of the predicted classes using the `predict_proba` method of the predictor object. Then we calculate the number of correctly classified validation samples using the `correct_classifications` variable by summing the number of predictions that are equal to the target variables.

Then we create an empty list called `correct_predictions_probabilities` and we then iterate through the predictions and calculate the probability of the predicted class for each prediction.

The way this works is that the `probabilities` is a 2D array with the shape `(n_samples, n_classes)`. Where each column represents the probability of the sample belonging to the class. Therefore, I can index using i to index the sample and `predicted_class` to index the class. This will give me the probability of the predicted class for the current sample. We then append this probability to the `correct_predictions_probabilities` list.

In the end we calculate the mean probability of the correct predictions by taking the mean of the `correct_predictions_probabilities` list. We then calculate the accuracy by dividing the number of correctly classified validation samples by the total number of validation samples. We then return the accuracy, the number of correctly classified validation samples and the average probability assigned to the correct class for all validation samples.

Code §4.3: Compute Metrics

```python
def compute_metrics(predictor, X, t):
    predictions = predictor.predict(X)
    probabilities = predictor.predict_proba(X)
    correct_classifications = np.sum(predictions == t)
    correct_predictions_probabilities = []
    for i, predicted_class in enumerate(predictions):
        probability_of_predicted_class = probabilities[
            i,
            predicted_class
        ]
        correct_predictions_probabilities.append(
            probability_of_predicted_class
        )
    mean_probability = np.mean(correct_predictions_probabilities)
    accuracy = correct_classifications/len(t)
    return accuracy, correct_classifications, mean_probability
```

Now to the optimal parameter search in our Random Forest Classifier (see Code §4.4, p. 19). We start by initializing the four variables:

1. `best_params`: A dictionary containing the best parameters found so far.
2. `best_correct_classifications`: The number of correctly classified validation samples for the best parameters found so far.
3. `best_avg_probability`: The average probability assigned to the correct class for all validation samples for the best parameters found so far.
4. `final_accuracy_on_validation_data`: The accuracy on the validation data for the best parameters found so far.

We then iterate through 15 randomly generated options for criterion, tree depth and number of features split rule. For each iteration we calculate the metrics using the `compute_metrics` function. We then check if the current parameters are better than the best parameters found so far based on the `avg_probability` and if the `avg_probability` is equal to the best average probability we check if the amount of correct classifications is higher. If they are, we update the best parameters, the number of correctly classified validation samples, the average probability assigned to the correct class for all validation samples and the accuracy on the validation data.

Code §4.4: Random Forest Classifier

```python
optimal_params = None
best_correct_classifications = 0
best_avg_probability = 0
final_accuracy_on_validation_data = 0
for i in range(15):
    n_estimators = Parameters.estimators.value
    criterion = random.choice(Parameters.criteria.value)
    max_depth = random.choice(Parameters.max_depths.value)
    max_features = random.choice(Parameters.n_features.value)
    model = random_forests_init(X_train, t_train, criterion=
         criterion,
                                max_depth=max_depth, max_features=
                                     max_features)
    precision, correct_classifications, avg_probability =
         compute_metrics(
        model, X_validation, t_validation
    )
    if avg_probability > best_avg_probability or (avg_probability
         == best_avg_probability and correct_classifications >
         best_correct_classifications):
        best_params = {
            'criterion': criterion,
            'max_depth': max_depth,
            'max_features': max_features
        }
        best_correct_classifications = correct_classifications
        best_avg_probability = avg_probability
        final_accuracy_on_validation_data = precision
```

I shall note I removed print statements from the code above to make it more readable, for the full code with prints see `exercise6.py`.

d) The accuracy improvements during the optimal parameter search.

Recall in task c), we were instructed to print two specific metrics. Subsequently, in task d), we are now additionally required to display a detailed string formatting that includes parameters and performance metrics: 'criterion = ? ; max depth = ? ; max features = ? ; accuracy on validation data = ? ; number of correctly classified validation samples = ?'.

The full output of running the actual Python program is shown in Appendix 6.5, but for the sake of clarity, I decided that it was better to show the output in a table. Hence, to improve readability, I have created a table below (Table 6) that shows the parameters and performance metrics for each iteration, and the best parameters and performance metrics found so far by highlighting it in bold.

Iter	Correct Cls	Avg Prob	Crit	Max D	Max F	Acc Val
1	64	0.722951	gini	10	log2	0.64
2	64	0.722780	-	-	-	-
3	66	0.723766	entropy	15	sqrt	0.66
4	66	0.706307	-	-	-	-
5	64	0.711746	-	-	-	-
6	65	0.666356	-	-	-	-
7	66	0.718409	-	-	-	-
8	**69**	**0.731324**	**entropy**	**10**	**sqrt**	**0.69**
9	69	0.662814	-	-	-	-
10	66	0.723841	-	-	-	-
11	68	0.655351	-	-	-	-
12	67	0.654395	-	-	-	-
13	64	0.657618	-	-	-	-
14	66	0.724132	-	-	-	-
15	65	0.718798	-	-	-	-

Table 6: Results across iterations with different parameters

To check how effective the different parameters are we used two metrics. The first metric was the number of correctly classified validation samples. The second was the average probability assigned to the correct class across all validation samples. These metrics provided insight into not only the classifier's accuracy but also its confidence in the predictions made. From the results seen in (Table 6), we can see:

1. Iteration 3, which used the 'entropy' criterion with a maximum tree depth of 15 and 'sqrt' for the maximum features, resulted in 66 correctly classified instances and an average probability of 0.723766.

2. Iteration 9 showed a slight improvement in the number of correctly classified instances (69) but a lower average probability of 0.662814.

3. The best performance was seen in iteration 8, which used the 'entropy' criterion, a maximum depth of 10, and 'sqrt' for the maximum features parameter. It achieved the highest accuracy on validation data of 0.69, as well as 69 correctly classified instances and an average probability of 0.71324.

5 Clustering

Question 7

a)

For the first task, we are asked to read and normalize the data from the comma-separated file `housing.csv`. The code below (see Code §5.1, p. 21) reads the data from the file and normalizes it.

Code §5.1: Read and Normalize Data

```python
1 X_data = pd.read_csv("housing.csv").values.astype('float64')
2 X_min = X_data.min(axis=0)
3 X_max = X_data.max(axis=0)
4 normalized_X = (X_data - X_min) / (X_max - X_min)
```

First, we read the data from the file using the `read_csv` function from the `pandas` library. We then convert it to a `numpy` array using the `.values` attribute, followed by a conversion to a `float` type using the `astype` function. To normalize the data using minimum and maximum values computed for each feature, we can use min-max feature scaling as defined in (5.1).

$$X^i_{norm} = \frac{X^i - X_{min}}{X_{max} - X_{min}}. \tag{5.1}$$

where X^i_{norm} is the normalized value of the i'th sample, X^i is the original value for the i'th sample, X_{min} is the minimum value of the feature across all samples in the dataset, and X_{max} is the maximum value of the feature across all samples in the dataset. This is implemented in the code above between lines 2 and 4.

b)

For the second task, we are asked to implement implement hierarchical K-means clustering without using any existing implementation. I took a object-oriented approach to this task, and the code might be a bit more verbose than necessary, but it is easier to read and understand. I will introduce the classes and methods used in small sections, as the full code is quite long and doesn't fit on a single page. So, let's start with the class declaration and the initialization method, `__init__`. The `__init__` method takes the number of clusters k as an argument, and `epsilon` which by default is set to `1e-9` which is equivalent to 10^{-9} and will serve as a threshold for the convergence of the algorithm. It then initializes the `k` attribute with the value of k and the `epsilon` attribute with the value of `epsilon`. The code below (see Code §5.2, p. 21) declares the class and the `__init__` method.

Code §5.2: K-means Clustering: Initialization

```python
class KMeans:
    def __init__(self, k: int, epsilon=1e-9):
        self.k = k
        self.epsilon = epsilon
```

Then we have the `__euclidian_distance` method which takes two points as arguments, and calculates the Euclidian distance between those two points and returns the Euclidian distance between them. More formally, the Euclidian distance between two points is as defined in (5.2).

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}. \tag{5.2}$$

The code below (see Code §5.3, p. 22) implements the `__euclidian_distance` method.

Code §5.3: K-means Clustering: Euclidian Distance

```python
def __euclidian_distance(self, p, q):
    return np.sqrt(np.sum((p - q)**2))
```

The `__pick_random_initial_centroid` method takes the dataset as an argument and the amount of samples to pick from the dataset. It then picks `amount` of random points from the dataset to serve as the initial centroids. The code below (see Code §5.4, p. 22) implements the `__pick_random_initial_centroid` method.

Code §5.4: K-means Clustering: Pick Random Initial Centroid

```python
def __pick_random_initial_centroid(self, data, amount=1):
    return data[np.random.choice(data.shape[0], amount, replace=
        False), :]
```

The `initialize_centroids` method takes the dataset X as argument and initializes the centroids by picking k random points from the dataset by utilizing the `__pick_random_initial_centroid` method helper function to pick the random centroid samples and then inserting them into the `centroids` array. The code below (see Code §5.5, p. 22) implements the `initialize_centroids` method.

Code §5.5: K-means Clustering: Initialize Centroids

```python
def initialize_centroids(self, X):
    centroids = np.zeros((self.k, X.shape[1]))
    for i in range(self.k):
        centroid = self.__pick_random_initial_centroid(X)
        centroids[i] = centroid
    return centroids
```

The `closest_centroid_indices` method takes the dataset and the centroids as arguments. We iterate through all points x_i in the dataset X and compute the distance from the point x_i to all respective centroids, c_j. We then find the index of the centroid with the smallest distance to the point x_i which mathematically can be formulated as:

$$\text{argmin}_j\, D(x_i, c_j). \tag{5.3}$$

And luckily Python numpy has a function for that, `np.argmin`. We then append the index of the closest centroid to the `closest_centroid_indices` array and we end up with an array of indices of the closest centroids for each point in the dataset X. The code below (see Code §5.6, p. 23) implements the `closest_centroid_indices` method.

Code §5.6: K-means Clustering: Closest Centroid Indices

```python
def closest_centroid_indices(self, X, centroids):
    closest_centroids_indices = np.array([])

    for point in X:
        distances_from_point_to_centroids = [
            self.__euclidian_distance(point, centroid) for centroid
                 in centroids
        ]
        closest_centroid_indice = np.argmin(
            distances_from_point_to_centroids
        )
        closest_centroids_indices = np.append(
            closest_centroids_indices, closest_centroid_indice
        )
    return closest_centroids_indices
```

The `update_centroids` method takes the dataset, the indices of the closest centroids, and the centroids as arguments. It then iterates through all centroids and for each centroid c_j it starts by creating an array of all the points x_i in the cluster associated with the centroid c_j by filtering out all points in the dataset X that does not fit the criteria of being closest to that centroid c_j. Then we ensure that the cluster is not empty, and if it is, we just skip the centroid c_j and move on to the next centroid. Otherwise, we calculate the mean of all points in the cluster associated with the centroid c_j, this process can also mathematically be formulated as:

$$c_j = \frac{1}{n_j} \sum_{x_i \text{ closest to } c_j} x_i. \tag{5.4}$$

where n_j is the number of points in the cluster associated with centroid c_j then in the end we update the centroid at index j in our `centroids` array with the new centroid $c_{j_{\text{new}}}$ that is the the mean of the cluster, `points_in_cluster`, and return the updated centroids. The code below (see Code §5.7, p. 23) implements the `update_centroids` method.

Code §5.7: K-means Clustering: Update Centroids

```python
def update_centroids(self, X, closest_centroids_indices,
     centroids):
    for centroid_indice, _ in enumerate(centroids):
        points_in_cluster = np.array([
            X[i] for i in range(len(X))
            if (closest_centroids_indices[i] == centroid_indice)
        ])

        if (len(points_in_cluster) == 0):
            continue

        centroid = points_in_cluster.mean(axis=0)
        centroids[centroid_indice] = centroid
    return centroids
```

The `fit` takes the data X as an argument and another optional argument `max_iter` which by default is set to 10k (which is very high, but because it is a small dataset, it doesn't take long to run and all times I have tested it has converged way before reaching even 40-50 iterations). This is where our object-oriented approach gets really useful in terms of code readability and understandability. We start by initializing the centroids using the `initialize_centroids` method. Then we enter a for loop that runs until the `fit` method is converged or max iterations has been reached. We then calculate the indices of the closest centroids for each point in the dataset X using the `closest_centroid_indices` method. We then save a copy of the `centroids` and store in `previous_centroids` before updating the centroids using the `update_centroids` method. Then we check if the centroids have converged by calculating the difference between the old centroids and the new centroids and if the difference is less than the threshold `epsilon` and for that we just use a numpy method called `allclose`. If the difference is less than the threshold `epsilon` we break out of the for loop and return the centroids and their respective closest centroids indices array. The code below (see Code §5.8, p. 24) implements the `fit` method.

Code §5.8: K-means Clustering: Fit

```python
def fit(self, X, max_iter=10000):
    centroids = self.initialize_centroids(X)

    for _ in range(max_iter):
        closest_centroids = self.closest_centroid_indices(X,
            centroids)
        previous_centroids = np.copy(centroids)
        centroids = self.update_centroids(X, closest_centroids,
            centroids)

        if np.allclose(previous_centroids, centroids, atol=self.
            epsilon):
            break
    return centroids, closest_centroids
```

The `__init__` in the `HierarchicalTopDownKMeans` class takes a list of k values as an argument and initializes the `k_sequence`.

Code §5.9: Hierarchical Top-Down K-means Clustering: Initialization

```python
class HierarchicalTopDownKMeans:
    def __init__(self, k_sequence: list):
        self.k_sequence = k_sequence
        self.centroids = []
        self.labels = []
```

The `__compute_intra_cluster_distance` method takes the dataset X, the centroids, and the indices of the closest centroids as arguments. It then iterates through all the centroids and for each centroid c_j it starts by creating an array of all the points x_i in the cluster associated with the centroid c_j by filtering out all points in the dataset X that does not fit the criteria of being closest to that centroid c_j. Then we ensure that the cluster is not empty, and if it is, we just skip the centroid c_j and move on to the next centroid. Otherwise, we calculate the sum of differences between all points in the cluster associated with the centroid c_j and the centroid c_j itself.

Code §5.10: Hierarchical Top-Down K-means Clustering: Compute Intra Cluster Distance

```python
def __compute_intra_cluster_distance(self, X, centroids,
     centroid_indices):
    intra_cluster_distance = 0
    for centroid_idx, centroid in enumerate(centroids):
        points_in_cluster = np.array([
            X[i] for i in range(len(X))
            if (centroid_indices[i] == centroid_idx)
        ])

        if (len(points_in_cluster) == 0):
            continue

        intra_cluster_distance += np.sum(
            [np.linalg.norm(point - centroid) for point in
                 points_in_cluster]
        )

    return intra_cluster_distance
```

The `fit` method takes the dataset X as an argument and starts by initializing the clusters with a single cluster containing all the data points and their respective indices. Then we initialize the final labels array with zeros and the label counter with zero. Then we iterate through the k values in the `k_sequence` and reset the clusters and centroids for this level/depth. Then we iterate through all the clusters and apply our custom k-means algorithm to each cluster. Then we add the new centroids to the `centroids_depth` array. Then we create new clusters and update the labels. Then we update the clusters for the next iteration. Then we update the centroids with the new ones found at this level. Then we reset the label counter for the next level of hierarchy. Then we assign the final labels to the class attribute `self.labels` and return the centroids, the labels, and the intra cluster distance. The code below (see Code §5.11, p. 26) implements the `fit` method.

Code §5.11: Hierarchical Top-Down K-means Clustering: Fit

```python
def fit(self, X):
    clusters = [(X, np.arange(X.shape[0]))]
    final_labels = np.zeros(X.shape[0], dtype=int)
    label_counter = 0
    for k in self.k_sequence:
        clusters_depth = []
        centroids_depth = []
        for cluster_data, indices in clusters:
            kmeans = KMeans(k)
            centroids, cluster_labels = kmeans.fit(cluster_data)
            centroids_depth.extend(centroids)
            for i in range(k):
                sub_cluster_mask = (cluster_labels == i)
                if np.any(sub_cluster_mask):
                    final_labels[indices[sub_cluster_mask]] =
                        label_counter
                    next_cluster_data = cluster_data[
                        sub_cluster_mask]
                    next_cluster_indices = indices[sub_cluster_mask
                        ]
                    next_cluster = (next_cluster_data,
                                    next_cluster_indices)
                    clusters_depth.append(next_cluster)
                    label_counter += 1
        clusters = clusters_depth
        self.centroids = centroids_depth
        label_counter = 0
    self.labels = final_labels

    return self.centroids, self.labels,
        self.__compute_intra_cluster_distance(X, self.centroids,
            self.labels)
```

Now that our implementation of hierarchical K-means clustering is done, we can finally test it. We are asked to test our implementation using the normalized data from a) and set number of clusters to $K_0 = 3$ on the first level and $K_1 = 2$ on the second level and $K_2 = 2$ on the third level. Furthermore, we are asked to run K-means 5 times using random samples from the dataset as the initial cluster centers (which our `KMeans` implementation handles internally). Then we have to select the solution with the smallest intra-cluster distance and compute and print the number of samples in each cluster. The code below (see Code §5.12, p. 27) tests our implementation of hierarchical K-means clustering.

Code §5.12: Get Optimized Hierarchical K-means

```python
def get_optimized_hierarchical_kmeans(X, iterations=5, k_sequence
    =[3, 2, 2]):
    best = (np.zeros(0), np.zeros(0), -1)
    worst = (np.zeros(0), np.zeros(0), -1)
    for _ in range(iterations):
        predictor = HierarchicalTopDownKMeans(k_sequence)
        centroids, label_indices, intra_cluster = predictor.fit(X)
        if best[2] == -1 or intra_cluster < best[2]:
            best = (centroids, label_indices, intra_cluster)
        if worst[2] == -1 or intra_cluster > worst[2]:
            worst = (centroids, label_indices, intra_cluster)
    return best, worst
best, worst = get_optimized_hierarchical_kmeans(normalized_X)
cluster_labels = best[1]
cluster_counts = np.zeros(12, dtype=int)
for label in cluster_labels:
    cluster_counts[label] += 1
for cluster, count in enumerate(cluster_counts):
    print(f"Cluster {cluster} has {count} samples.")
print("Total number of samples: ", np.sum(cluster_counts))
```

The `get_optimized_hierarchical_kmeans` function takes the dataset X as an argument, and the amount of iterations which by default is set to 5, and takes a `k_sequence` as an argument which by default is set to `[3, 2, 2]`. Then we create a 3-tuple that has the first element as the centroids, and the second element as the labels or centroid indices, and the last third and last element as the intra-cluster distance for the given H-K-means clustering solution. Then we iterate through the amount of iterations and for each iteration we fit the `HierarchicalTopDownKMeans` class with the dataset X and the `k_sequence` and then we check if the intra-cluster distance is one of the following: 1. the default value: -1, which means that it is the first iteration and we just want to set the best to the first initial test. The 2nd is if the intra-cluster distance is less than the previous intra-cluster distance set to the current best one, we update the `best` with the new solution. We do the exact same, just with a flipped sign for the intra-cluster distance, to find the worst solution. Then we return the best and worst solution as a 2-tuple. The output of the code above is:

```
$ python .\question7.py
Cluster 0 has 170 samples.
Cluster 1 has 55 samples.
Cluster 2 has 146 samples.
Cluster 3 has 129 samples.
Cluster 4 has 72 samples.
Cluster 5 has 116 samples.
Cluster 6 has 59 samples.
Cluster 7 has 105 samples.
Cluster 8 has 33 samples.
Cluster 9 has 32 samples.
Cluster 10 has 48 samples.
Cluster 11 has 34 samples.
Total number of samples: 999
```

c)

For the third task, we are asked to compute the principal components of the data and we are allowed to use existing implementations, so I will be using the `PCA` class from the `sklearn.decomposition` module. The code below (see Code §5.13, p. 28) computes the principal components of the data.

Code §5.13: Compute Principal Components

```
pca = PCA(n_components=2)
pca.fit(normalized_X)
```

The deliverables said to include the number of samples in each cluster in the report, but in the task exercises it said to print it in the previous exercise, so for the output of number of samples in each cluster, please refer to the previous exercise.

d)

For the fourth task, we have transformed the normalized data X using the two largest components (the first two principal components) i.e., eigenvectors with the largest eigenvalues by using the `transform` method from the `PCA` class. The code below (see Code §5.14, p. 29) transforms the normalized data X using the two largest components and plots the transformed data.

Code §5.14: Transform Data and Plot

```
1  X_pca = pca.transform(normalized_X)
2  def visualize_clusters(X, centroids, closest_centroids):
3      plt.figure(figsize=(15, 10), dpi=90)
4      unique_labels = np.unique(closest_centroids)
5      for i, label in enumerate(unique_labels):
6          color = colors[i % len(colors)]
7          plt.scatter(X[closest_centroids == label, 0], X[
               closest_centroids == label, 1],
8                      color=color, label=f'Cluster {label + 1}',
                           alpha=0.3, s=50, edgecolor='k')
9          centroid_transformed = pca.transform(centroids[i].reshape
               (1, -1))
10         plt.scatter(centroid_transformed[:, 0],
               centroid_transformed[:, 1],
11                     color=color, edgecolor='black', alpha=1, s=90,
                           marker='o', lw=2)
12     plt.xlabel('Principal Component 1')
13     plt.ylabel('Principal Component 2')
14     plt.title('Clustering results with PCA')
15     plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left',)
16     plt.grid(True)
17     plt.tight_layout()
18     plt.show()
19 centroids_best, centroids_indices_best, dist = best
20 visualize_clusters(X_pca, centroids_best, centroids_indices_best)
```

The first line of the code transforms the normalized dataset, denoted as X, into its first two principal components, PC1 and PC2. Which is crucial for being able to visualize the data in a two-dimensional space.
The second line introduces the function `visualize_clusters`. This function takes two arguments: the transformed dataset X and the `closest_centroids` labels, which represent the indices of the nearest centroids for each data point.
From the third to the eleventh line, the code plots the transformed data in the two-dimensional space defined by PC1 and PC2. This is achieved by iterating through each cluster. Ideally, the number of unique clusters should match the length of the unique labels array, which is expected to be 12 in this case. For each cluster, the code selects the corresponding data points using `X[closest_centroids == label]` and plots them using a scatter plot.
In the scatter plot, each cluster's data points are colored to match their respective cluster. Additionally, the centroids are also plotted in the same color as their cluster but distinguished by a black ring. This distinction helps in easily differentiating between the centroids and the data points. To enhance visibility and clarity for the centroids and clusters, the data points are also rendered to be slightly transparent.
It is also important to note that since the centroids are originally in the same space as the dataset, they must also be transformed to the two-dimensional space of PC1 and PC2. We again use the `transform` method just before the centroids are scattered on the plot.

Furthermore, as described in the previous exercise I have both the best and worst solution they can be seen in Figures 2 and 3, respectively.

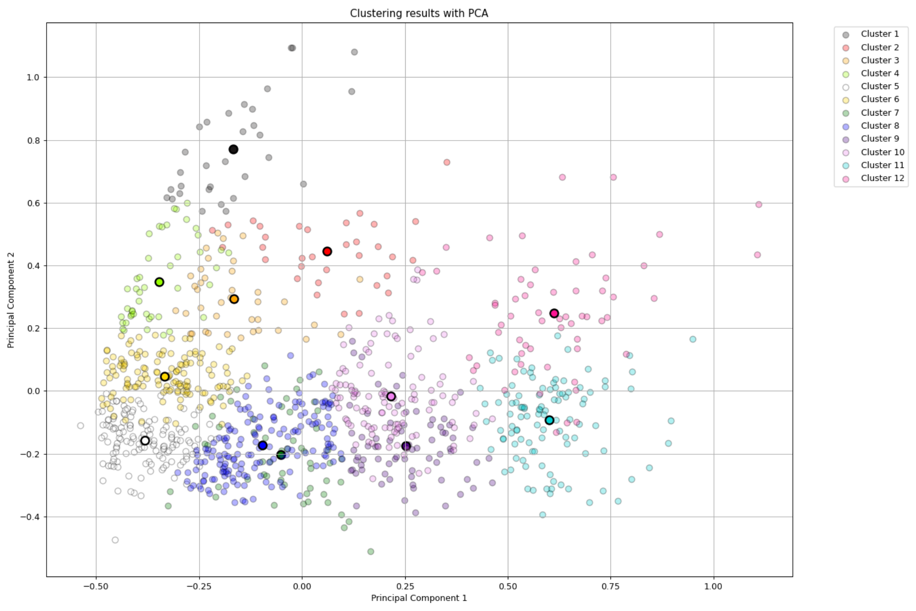

Figure 2: Best solution

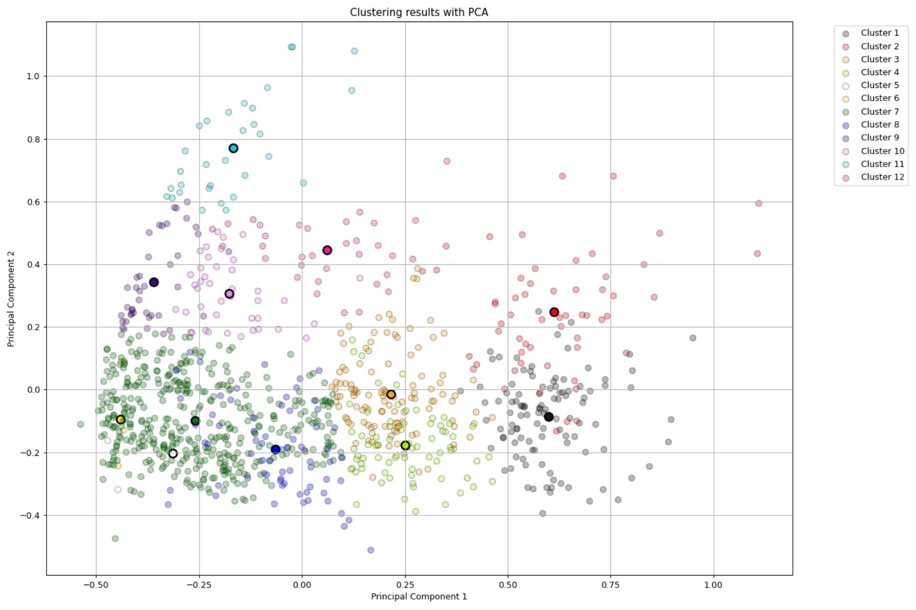

Figure 3: Worst solution

It is quite clear if we look at worst Figure 3 that the clusters are not as well separated and the centroids are not very well placed. If we look at the bottom left at the yellow cluster (Cluster 6) we see that it is very poorly placed among the green cluster (Cluster 7).

The best solution Figure 2 on the other hand, has a better separation of clusters and the centroids are very well placed especially the yellow cluster (Cluster 6) is placed much more reasonable.

This is also reflected in the intra-cluster distance, where the best solution has an intra-cluster distance of 227.38 and the worst solution has an intra-cluster distance of 250.67.

6 Appendix

6.1 Full Code for Question 2

```
import numpy.linalg as linalg
import scipy.stats

z_score = -2.19296
p_value = scipy.stats.norm.cdf(z_score)

print(p_value)
```

6.2 Full Code for Question 4

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

np.random.seed(2025)


def data_info(dataset, samples, num, combined):
    print("Number of samples in %s data: %i" % (dataset, samples)
        )
    print("Number of numerical features in %s data: %i" % (
        dataset, num))
    print("Number of combined features in %s data: %i" % (dataset
        , combined))


train_data = pd.read_csv("./datafiles/heart_simplified_train_2023
    .csv")
test_data = pd.read_csv("./datafiles/heart_simplified_test_2023.
    csv")
```

```
train_data_numerical = train_data[[
    'Age', 'RestingBP', 'Cholesterol', 'MaxHR']].values.astype('
        float64')
train_data_combined = train_data[[
    'Age', 'RestingBP', 'Cholesterol', 'MaxHR', 'Sex', '
        ChestPainType']].values
train_data_labels = train_data[['HeartDisease']].values.astype('
    int')

test_data_numerical = test_data[[
    'Age', 'RestingBP', 'Cholesterol', 'MaxHR']].values.astype('
        float64')
test_data_combined = test_data[[
    'Age', 'RestingBP', 'Cholesterol', 'MaxHR', 'Sex', '
        ChestPainType']].values
test_data_labels = test_data[['HeartDisease']].values.astype('int
    ')


data_info("training", train_data_numerical.shape[0],
           train_data_numerical.shape[1], train_data_combined.
               shape[1])
data_info("test", test_data_numerical.shape[0],
    test_data_numerical.shape[1],
           test_data_combined.shape[1])


class NearestNeighborRegressor:
    def __init__(self, n_neighbors=3, wnum=1, wcat=0.025):
        """
        Initializes the nearest neighbor regression model.
        """
        self.n_neighbors = n_neighbors
        self.wnum = wnum
        self.wcat = wcat

    def fit(self, X, t, numerical_indices=None,
        categorical_indices=None):
        """
        Fits the model to the provided data.
        """
        self.X_train = X
        self.t_train = t
        self.numerical_indices = numerical_indices
        self.categorical_indices = categorical_indices

    def predict(self, data):
        """
        Makes predictions on the provided data.
        :return: the predictions
        """
        predictions = []
        for x in data:
            distances = [self.__combinedDistance(x, p) for p in
                self.X_train]
            indices = np.argsort(distances)[:self.n_neighbors]
            neighbors_labels = self.t_train[indices]
            prediction = round(np.mean(neighbors_labels))
            predictions.append(prediction)
        return np.array(predictions)

    def __combinedDistance(self, p, q):
        """
        Computes the combined distance between two data points.
        :return: the combined distance
        """
        numerical_distance = self.wnum * np.sum(np.abs(p[self.
            numerical_indices] -
                                                      q[self.
                                                          numerical_indices
                                                          ]))
        categorical_distance = self.wcat * np.sum(
            p[self.categorical_indices] != q[self.
                categorical_indices]
        )
        return numerical_distance + categorical_distance

    def rmse(self, t, tp):
        """
        Computes the RMSE for two input arrays.
        :return: the RMSE
        """
        mse = np.mean((t - tp)**2)
        rmse_value = np.sqrt(mse)
        return rmse_value

    def accuracy(self, t, tp):
        """
        Computes the accuracy for two input arrays.
        :return: the accuracy
        """
        correct_predictions = np.sum(t == tp)
        total_samples = len(t)
        accuracy_value = correct_predictions / total_samples
        return accuracy_value
```

```
# ==QUESTION A==
k_values = [1, 3, 5, 7, 9]

print("-" * 30)
for k in k_values:
    model = NearestNeighborRegressor(n_neighbors=k)
    model.fit(train_data_numerical, train_data_labels.flatten())

    predictions = model.predict(test_data_numerical)

    rmse_value = model.rmse(test_data_labels.flatten(),
        predictions)
    accuracy_value = model.accuracy(test_data_labels.flatten(),
        predictions)

    print(f"Results for k={k}:")
    print(f"RMSE: {rmse_value}")
    print(f"Accuracy: {accuracy_value}")
    print("-" * 30)
    plt.scatter(k, accuracy_value)
    plt.xlabel("k")
    plt.ylabel("accuracy")
    plt.title("Dependency between the accuracy and k")

plt.show()

# ==QUESTION B==
numerical_indices = [0, 1, 2, 3]
categorical_indices = [4, 5]

wcat_values = [0.025, 0.05, 0.1]

for wcat in wcat_values:
    model = NearestNeighborRegressor(n_neighbors=5, wcat=wcat)
    model.fit(train_data_combined, train_data_labels.flatten(),
              numerical_indices, categorical_indices)

    predictions = model.predict(test_data_combined)

    rmse_value = model.rmse(test_data_labels.flatten(),
        predictions)
    accuracy_value = model.accuracy(test_data_labels.flatten(),
        predictions)

    print(f"Results for wcat={wcat}:")
    print(f"RMSE: {rmse_value}")
    print(f"Accuracy: {accuracy_value}")
    print("-" * 30)
```

6.3 Full Output for Question 4

```
$ python .\question4.py
Number of samples in training data: 500
Number of numerical features in training data: 4
Number of combined features in training data: 6
Number of samples in test data: 100
Number of numerical features in test data: 4
Number of combined features in test data: 6
------------------------------
Results for k=1:
RMSE: 0.6082762530298219
Accuracy: 0.63
------------------------------
Results for k=3:
RMSE: 0.5830951894845301
Accuracy: 0.66
------------------------------
Results for k=5:
RMSE: 0.5477225575051661
Accuracy: 0.7
------------------------------
Results for k=7:
RMSE: 0.5099019513592785
Accuracy: 0.74
------------------------------
Results for k=9:
RMSE: 0.5385164807134504
Accuracy: 0.71
------------------------------
Results for wcat=0.025:
RMSE: 0.5477225575051661
Accuracy: 0.7
------------------------------
Results for wcat=0.05:
RMSE: 0.5099019513592785
Accuracy: 0.74
------------------------------
Results for wcat=0.1:
RMSE: 0.458257569495584
Accuracy: 0.79
------------------------------
```

6.4 Full Code for Question 6

```
from enum import Enum
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import random

train_data = pd.read_csv("./datafiles/heart_simplified_train_2023
    .csv")
test_data = pd.read_csv("./datafiles/heart_simplified_test_2023.
    csv")
validation_data = pd.read_csv(
    "./datafiles/heart_simplified_validation_2023.csv")

np.random.seed(2025)


class Parameters(Enum):
    estimators = 100
    criteria = ['gini', 'entropy']
    max_depths = [2, 5, 7, 10, 15]
    n_features = ['sqrt', 'log2']


class Gender(Enum):
    MALE = 1
    FEMALE = 0

# ==QUESTION A==


def from_categorical_to_numerical(X):
    """
    Converts the categorical data to numerical
    :return: the data with the categorical data converted to
        numerical
    """
    X[:, 5] = np.where(X[:, 5] == 'M', Gender.MALE.value, Gender.
        FEMALE.value)

    chest_pain_categories = np.unique(X[:, 6])

    last_column = X[:, -1].copy()
    X = np.delete(X, -1, axis=1)

    for category in chest_pain_categories:
        one_hot_column = (X[:, 6] == category).astype(int)
        X = np.hstack((X, one_hot_column.reshape(-1, 1)))
    X = np.delete(X, 6, axis=1)
    X = np.hstack((X, last_column.reshape(-1, 1)))
    return X


train_data = from_categorical_to_numerical(train_data.values)
X_train = train_data[:, 1:-1]
t_train = train_data[:, -1].astype('int')

test_data = from_categorical_to_numerical(test_data.values)
X_test = test_data[:, 1:-1]
t_test = test_data[:, -1].astype('int')

validation_data = from_categorical_to_numerical(validation_data.
    values)
X_validation = validation_data[:, 1:-1]
t_validation = validation_data[:, -1].astype('int')


# ==QUESTION B==

def accuracy(predictor, X, t):
    """
    Computes the accuracy of a predictor
    :return: the accuracy
    """
    predictions = predictor.predict(X)
    return np.sum(predictions == t)/len(t)


def random_forests_init(X, t, n_estimators=100, criterion="gini",
     max_features="sqrt", max_depth=None):
    """
    Creates a random forest classifier
    :return: the predictor
    """
    predictor = RandomForestClassifier(n_estimators=n_estimators,
         criterion=criterion, max_depth=max_depth,
                                    max_features=max_features)
    predictor.fit(X, t)
    return predictor


predictor = random_forests_init(X_train, t_train)
precision = accuracy(predictor, X_train, t_train)
print("Accuracy on training data: ", precision)
```

```
# ==QUESTION C==

def compute_metrics(predictor, X, t):
    """
    Computes the precision, correct classifications and mean
        probability of a predictor
    :return: precision, correct classifications, mean probability
    """
    predictions = predictor.predict(X)
    probabilities = predictor.predict_proba(X)
    correct_classifications = np.sum(predictions == t)
    correct_predictions_probabilities = []
    for i, predicted_class in enumerate(predictions):
        probability_of_predicted_class = probabilities[
            i,
            predicted_class
        ]
        correct_predictions_probabilities.append(
            probability_of_predicted_class
        )
    mean_probability = np.mean(correct_predictions_probabilities)
    accuracy = correct_classifications/len(t)
    return accuracy, correct_classifications, mean_probability


optimal_params = None
best_correct_classifications = 0
best_avg_probability = 0
final_accuracy_on_validation_data = 0

for i in range(15):
    print("--------------------Iteration %i--------------------"
        % (i + 1))
    n_estimators = Parameters.estimators.value
    criterion = np.random.choice(Parameters.criteria.value)
    max_depth = np.random.choice(Parameters.max_depths.value)
    max_features = np.random.choice(Parameters.n_features.value)

    model = random_forests_init(X_train, t_train, criterion=
        criterion,
                              max_depth=max_depth, max_features=
                                  max_features)

    precision, correct_classifications, avg_probability =
        compute_metrics(
        model, X_validation, t_validation
    )
    print("Correct classifications: ", correct_classifications)
    print("Average probability: ", avg_probability)

    if avg_probability > best_avg_probability or (avg_probability
         == best_avg_probability and correct_classifications >
        best_correct_classifications):
        print("New best parameters found!")
        best_params = {
            'criterion': criterion,
            'max_depth': max_depth,
            'max_features': max_features
        }
        best_correct_classifications = correct_classifications
        best_avg_probability = avg_probability
        final_accuracy_on_validation_data = precision
        print("criterion = %s; max depth = %i; max features = %s;
            accuracy on validation data = %f; number of correctly
            classified validation samples = %i" % (
            criterion, max_depth, max_features, precision,
                correct_classifications))
    print(" ")

print("--------------------Best Parameters--------------------")
print("Best Parameters:", best_params)
print("Best Correct Classifications:",
    best_correct_classifications)
print("Best Average Probability:", best_avg_probability)
print("Best Accuracy on Validation Data:",
    final_accuracy_on_validation_data)
```

6.5 Full Output for Question 6

```
$ python .\question6.py
Accuracy on training data: 1.0
--------------------Iteration 1--------------------
Correct classifications: 64
Average probability: 0.722951187429673
New best parameters found!
criterion = gini; max depth = 10; max features = log2; accuracy
    on validation data = 0.640000; number of correctly classified
     validation samples = 64

--------------------Iteration 2--------------------
Correct classifications: 64
Average probability: 0.7227801924987305

--------------------Iteration 3--------------------
```

```
Correct classifications: 66
Average probability: 0.7237660086482145
New best parameters found!
criterion = entropy; max depth = 15; max features = sqrt;
    accuracy on validation data = 0.660000; number of correctly
    classified validation samples = 66

--------------------Iteration 4--------------------
Correct classifications: 66
Average probability: 0.7063067751716661

--------------------Iteration 5--------------------
Correct classifications: 64
Average probability: 0.711746380261937

--------------------Iteration 6--------------------
Correct classifications: 65
Average probability: 0.6663555089276468

--------------------Iteration 7--------------------
Correct classifications: 66
Average probability: 0.7184091935067772

--------------------Iteration 8--------------------
Correct classifications: 69
Average probability: 0.7313240614268254
New best parameters found!
criterion = entropy; max depth = 10; max features = sqrt;
    accuracy on validation data = 0.690000; number of correctly
    classified validation samples = 69

--------------------Iteration 9--------------------
Correct classifications: 69
Average probability: 0.662814018788145

--------------------Iteration 10--------------------
Correct classifications: 66
Average probability: 0.7238408937673952

--------------------Iteration 11--------------------
Correct classifications: 68
Average probability: 0.6553506140207151

--------------------Iteration 12--------------------
Correct classifications: 67
Average probability: 0.6543949115305117

--------------------Iteration 13--------------------
```

```
Correct classifications: 64
Average probability: 0.6576182098806165

--------------------Iteration 14--------------------
Correct classifications: 66
Average probability: 0.7241316758241757

--------------------Iteration 15--------------------
Correct classifications: 65
Average probability: 0.7187982984461092

--------------------Best Parameters--------------------
Best Parameters: {'criterion': 'entropy', 'max_depth': 10, '
    max_features': 'sqrt'}
Best Correct Classifications: 69
Best Average Probability: 0.7313240614268254
Best Accuracy on Validation Data: 0.69
```

6.6 Full Code for Question 7

```
    from matplotlib.colors import ListedColormap
from sklearn.decomposition import PCA
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import random

colors = [
    "#191616",
    "#FF0000",
    "#FFA500",
    "#99FF00",
    "#FFFFFF",
    "#FFD700",
    "#008000",
    "#0000FF",
    "#4B0082",
    "#EE82EE",
    "#00CED1",
    "#FF1493",
]

# ==QUESTION A==

X_data = pd.read_csv("./datafiles/housing.csv").values.astype('
    float64')
X_min = X_data.min(axis=0)
```

```
X_max = X_data.max(axis=0)
normalized_X = (X_data - X_min) / (X_max - X_min)

# ==QUESTION B==


class KMeans:
    def __init__(self, k: int, epsilon=1e-9):
        self.k = k
        self.epsilon = epsilon

    def __euclidian_distance(self, p, q):
        """
        Calculates the euclidian distance between p and q
        """
        return np.sqrt(np.sum((p - q)**2))

    def __pick_random_initial_centroid(self, data, amount=1):
        """
        Picks a random initial centroid
        :return: the random initial centroid
        """
        return data[np.random.choice(data.shape[0], amount,
            replace=False), :]

    def initialize_centroids(self, X):
        """
        Initializes k random centroids samples from the data
        :return: the centroids
        """
        centroids = np.zeros((self.k, X.shape[1]))
        for i in range(self.k):
            centroid = self.__pick_random_initial_centroid(X)
            centroids[i] = centroid
        return centroids

    def closest_centroid_indices(self, X, centroids):
        """
        Returns the closest centroid indices for each point in X
        :return: the closest centroid indices
        """
        closest_centroids_indices = np.array([])

        for point in X:
            distances_from_point_to_centroids = [
                self.__euclidian_distance(point, centroid) for
                    centroid in centroids
            ]
            closest_centroid_indice = np.argmin(
                distances_from_point_to_centroids
            )
            closest_centroids_indices = np.append(
                closest_centroids_indices, closest_centroid_indice
            )
        return closest_centroids_indices

    def update_centroids(self, X, closest_centroids_indices,
        centroids):
        """
        Updates the centroids
        :return: the updated centroids
        """
        for centroid_indice, _ in enumerate(centroids):
            points_in_cluster = np.array([
                X[i] for i in range(len(X))
                if (closest_centroids_indices[i] ==
                    centroid_indice)
            ])

            if (len(points_in_cluster) == 0):
                continue

            centroid = points_in_cluster.mean(axis=0)
            centroids[centroid_indice] = centroid
        return centroids

    def fit(self, X, max_iter=10000):
        """
        Fits the data to the model
        :return: the centroids and the closest centroid indices
        """
        centroids = self.initialize_centroids(X)

        for _ in range(max_iter):
            closest_centroids = self.closest_centroid_indices(X,
                centroids)
            previous_centroids = np.copy(centroids)
            centroids = self.update_centroids(X, closest_centroids
                , centroids)

            if np.allclose(previous_centroids, centroids, atol=
                self.epsilon):
                break
        return centroids, closest_centroids
```

```
class HierarchicalTopDownKMeans:
    def __init__(self, k_sequence: list):
        """
        Initializes the model
        """
        self.k_sequence = k_sequence
        self.centroids = []
        self.labels = []

    def __compute_intra_cluster_distance(self, X, centroids,
        centroid_indices):
        """
        Computes the intra cluster distance
        :return: the intra cluster distance
        """
        intra_cluster_distance = 0
        for centroid_idx, centroid in enumerate(centroids):
            points_in_cluster = np.array([
                X[i] for i in range(len(X))
                if (centroid_indices[i] == centroid_idx)
            ])

            if (len(points_in_cluster) == 0):
                continue

            intra_cluster_distance += np.sum(
                [np.linalg.norm(point - centroid) for point in
                    points_in_cluster])

        return intra_cluster_distance

    def fit(self, X):
        """
        Fits the data to the model
        :return: the centroids, the labels and the intra cluster
            distance
        """
        clusters = [(X, np.arange(X.shape[0]))]
        final_labels = np.zeros(X.shape[0], dtype=int)
        label_counter = 0

        for k in self.k_sequence:
            clusters_depth = []
            centroids_depth = []

            for cluster_data, indices in clusters:
                kmeans = KMeans(k)
                centroids, cluster_labels = kmeans.fit(
                    cluster_data)
                centroids_depth.extend(centroids)
                for i in range(k):
                    sub_cluster_mask = (cluster_labels == i)
                    if np.any(sub_cluster_mask):
                        final_labels[indices[sub_cluster_mask]] =
                            label_counter
                        next_cluster_data = cluster_data[
                            sub_cluster_mask]
                        next_cluster_indices = indices[
                            sub_cluster_mask]
                        next_cluster = (next_cluster_data,
                                        next_cluster_indices)
                        clusters_depth.append(next_cluster)
                        label_counter += 1

            clusters = clusters_depth
            self.centroids = centroids_depth
            label_counter = 0
        self.labels = final_labels
        return self.centroids, self.labels, self.
            __compute_intra_cluster_distance(X, self.centroids,
            self.labels)


def get_optimized_hierarchical_kmeans(X, iterations=5, k_sequence
    =[3, 2, 2]):
    """
    Runs the model multiple times and returns the best and worst
        results
    :return: the best and worst results
    """
    best = (np.zeros(0), np.zeros(0), -1)
    worst = (np.zeros(0), np.zeros(0), -1)
    for _ in range(iterations):
        predictor = HierarchicalTopDownKMeans(k_sequence)
        centroids, label_indices, intra_cluster = predictor.fit(X)
        if best[2] == -1 or intra_cluster < best[2]:
            best = (centroids, label_indices, intra_cluster)
        if worst[2] == -1 or intra_cluster > worst[2]:
            worst = (centroids, label_indices, intra_cluster)
    return best, worst


best, worst = get_optimized_hierarchical_kmeans(normalized_X)

cluster_labels = best[1]
cluster_counts = np.zeros(12, dtype=int)
```

```
for label in cluster_labels:
    cluster_counts[label] += 1

for cluster, count in enumerate(cluster_counts):
    print(f"Cluster {cluster} has {count} samples.")
print("Total number of samples: ", np.sum(cluster_counts))


# ==QUESTION C==
pca = PCA(n_components=2)
pca.fit(normalized_X)

# ==QUESTION D==
X_pca = pca.transform(normalized_X)


def visualize_clusters(X, centroids, closest_centroids):
    plt.figure(figsize=(15, 10), dpi=90)

    unique_labels = np.unique(closest_centroids)

    for i, label in enumerate(unique_labels):
        color = colors[i % len(colors)]
        plt.scatter(X[closest_centroids == label, 0], X[
            closest_centroids == label, 1],
                    color=color, label=f'Cluster {label + 1}',
                        alpha=0.3, s=50, edgecolor='k')
        centroid_transformed = pca.transform(centroids[i].reshape
            (1, -1))
        plt.scatter(centroid_transformed[:, 0],
            centroid_transformed[:, 1],
                    color=color, edgecolor='black', alpha=1, s=90,
                        marker='o', lw=2)

    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.title('Clustering results with PCA')
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left',)
    plt.grid(True)
    plt.tight_layout()

    plt.show()


centroids_best, centroids_indices_best, dist = best
visualize_clusters(X_pca, centroids_best, centroids_indices_best)
```

```
# for my report to show worst case, not a part of d)
# centroids_worst, centroids_indices_worst, dist = worst
# visualize_clusters(X_pca, centroids_worst,
    centroids_indices_worst)
```