MAD Exam

Exam number: 12

January 19, 2025

Question 1

1)

The likelihood of each random variable X_n is described by the given exponential function: $f(x;\lambda) = \lambda e^{-\lambda x}$ From 2.30 in A First Course In Machine Learning we see that the joint density is simply the product of the likelihood function of each random variable (as they are i.i.d from the same distribution). This means the likelihood function for all random variables can be derived in the following way:

$$\begin{aligned} L(\lambda) &= \prod_{n=1}^{N} \lambda e^{-\lambda x_n} \\ &= \lambda^N \prod_{n=1}^{N} e^{-\lambda x_n} \\ &= \lambda^N (e^{-\lambda x_1} e^{-\lambda x_2} ... e^{-\lambda x_n}) \\ &= \lambda^N (e^{-\lambda \sum_{n=1}^{N} x_n}) \end{aligned}$$

We have now derived the likelihood function for a set for i.i.d random variables.

2)

The maximum likelihood estimator $\hat{\lambda}$ of λ can be found by first taking the natural logarithm (denoted log as in the book A First Course In Machine Learning) on the likelihood function:

$$\begin{aligned} log(L(\lambda)) &= log(\lambda^N \cdot (e^{-\lambda(\sum_{n=1}^{N} x_n)})) \\ &= log(\lambda^N) + log(e^{-\lambda \sum_{n=1}^{N} x_n}) \\ &= Nlog(\lambda) + log(e^{-\lambda \sum_{n=1}^{N} x_n}) \\ &= Nlog(\lambda) - \lambda \sum_{n=1}^{N} x_n \end{aligned}$$

Next we need to take the derivative of the above and set it equal to 0 to find

the MLE:

$$\begin{aligned}\hat{\lambda} &= \frac{d}{d\lambda}(Nlog(\lambda) - \lambda \sum_{n=1}^{N} x_n) \\ &= \frac{N}{\lambda} - \frac{d}{d\lambda}(\lambda \sum_{n=1}^{N} x_n)) \\ &= \frac{N}{\lambda} - \sum_{n=1}^{N} x_n\end{aligned}$$

Setting it equal to 0 and isolating λ:

$$\begin{aligned}0 = \frac{N}{\lambda} - \sum_{n=1}^{N} x_n \quad &\rightarrow \\ \sum_{n=1}^{N} x_n = \frac{N}{\lambda} \quad &\rightarrow \\ \lambda \sum_{n=1}^{N} x_n = N \quad &\rightarrow \\ \lambda = \frac{N}{\sum_{n=1}^{N} x_n}\end{aligned}$$

3 & 4)

We can estimate/calculate the rate parameter $\hat{\lambda}$ for the dataset provided in sub task 3 by using the formula found in sub task 2, inserting the values we get:

$$\begin{aligned}\hat{\lambda} &= \frac{10}{(2.1 + 3.4 + 1.8 + 4.2 + 2.9 + 3.1 + 5.0 + 1.5 + 4.3 + 2.7)} \\ &= \frac{10}{31}\end{aligned}$$

5)

To calculate the mean time between arrivals, we need to calculate $\frac{1}{\lambda}$ this is because this will tell us how many minutes before one lambda (our rate), or in other words how many minutes between arrivals.

$$\frac{1}{\hat{\lambda}} = \frac{1}{\frac{10}{31}} = \frac{31}{10}$$

This means the mean time between arrivals is 3.1 minutes using the estimated $\hat{\lambda}$

Question 2

To test whether the mean of the dataset is different from the known population mean, we need to formulate a null hypothesis and an alternate hypothesis. As we want to find out whether the sample mean is different we chose:

$$H_0 : \mu = \mu_0$$
$$H_1 : \mu \neq \mu_0$$

We can assume that the data is normally distributed without a known population standard deviation. We should be making a two-tailed test as we want to check whether it's different (both ways). We should use a t-test as the population standard deviation is unknown, even if it was known we should still chose a t-test as the sample size is smaller than 30 (CLT). We will be using a significance level of 0.05.

We now need to calculate the sample mean and sample standard deviation to perform the t-test:

$$\bar{X} = \frac{\sum_{i=1}^{n} x_i}{n} \approx 27.48$$

$$S = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{X})^2} \approx 4.6894207176011$$

We can now perform the two-tailed t-test assuming $\mu_0 = 30$:

$$T = \frac{\bar{X} - \mu_0}{S/\sqrt{n}}$$

$$\begin{aligned} T &= \frac{\bar{X} - \mu_0}{S/\sqrt{n}} \\ &= \frac{27.48 - 30}{4.6894207176011/\sqrt{10}} \\ &= \frac{-2.52}{4.6894207176011/\sqrt{10}} \\ &= \frac{-2.52}{4.6894207176011/\sqrt{10}} \\ &= -1.69934415859 \end{aligned}$$

Using `scipy.stats.t.ppf` with 0.975, as we have $\alpha = 0.05$ and it's two-tailed, and saying it's 9 degrees of freedom (as $df = n - 1 = 10 - 1 = 9$). We

get 2.2621571628540993 and -2.2621571628540993 and as $-2.262 < -1.699 < 2.262$, the test statistics lies within the acceptance range.

This means we accept the null hypothesis, and there isn't sufficient evidence at the significance level $\alpha = 0.05$ to say that the sample mean is different from the population mean assuming $\mu_0 = 30$.

Question 3

To ease the calculation step we can start by creating a table containing the provided information.

	Drug A	Drug B	Survival time
Patient 1	1	0	0.6
Patient 2	0	1	1.1
Patient 3	1	0	3
Patient 4	0	1	1
Patient 5	1	1	0.2

This table is structured in the same way as our feature matrix for the principal component analysis. I have chosen to only look at the drugs as features, as in a real use case scenario we would likely want to predict survival time and would use PCA to check if perhaps one of the drugs actually covers most or all of the variance.

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

The first step of PCA is to center the data, this is done by calculating the mean of each feature (column in the matrix) and subtracting it from the feature matrix in the respective column.

$$\mu_A = \frac{1+0+1+0+1}{5} = 0.6$$
$$\mu_B = \frac{0+1+0+1+1}{5} = 0.6$$

$$X_{cent} = \begin{bmatrix} 0.4 & -0.6 \\ -0.6 & 0.4 \\ 0.4 & -0.6 \\ -0.6 & 0.4 \\ 0.4 & 0.4 \end{bmatrix}$$

Next we need to compute the covariance matrix, this is done by multiplying the feature matrix with it's transpose and dividing by the amount of samples (as

on page 245 in A First Course in Machine Learning).

$$\begin{aligned}
\Sigma &= \frac{1}{N} X_{cent}^T X_{cent} \\
X_{cent}^T &= \begin{bmatrix} 0.4 & -0.6 & 0.4 & -0.6 & 0.4 \\ -0.6 & 0.4 & -0.6 & 0.4 & 0.4 \end{bmatrix} \\
\frac{1}{N} X_{cent}^T X_{cent} &= \frac{1}{5} \begin{bmatrix} 1.2 & -0.8 \\ -0.8 & 1.2 \end{bmatrix} \\
&= \begin{bmatrix} 0.24 & -0.16 \\ -0.16 & 0.24 \end{bmatrix}
\end{aligned}$$

Now we can calculate the eigenvalues for the covariance matrix, which can be done by solving $det(\Sigma - \lambda I) = 0$ where Σ is the calculated covariance matrix. Meaning we need to solve:

$$\begin{aligned}
0 &= det\left(\begin{bmatrix} 0.24 - \lambda & -0.16 \\ -0.16 & 0.24 - \lambda \end{bmatrix} \right) \\
&= (0.24 - \lambda)^2 - (-0.16)^2 \\
&= ((0.24 - \lambda) \cdot (0.24 - \lambda)) - 0.0256 \\
&= 0.0576 - 0.24\lambda - 0.24\lambda + \lambda^2 - 0.0256 \\
&= 0.0576 - 0.48\lambda + \lambda^2 - 0.0256 \\
&= 0.032 - 0.48\lambda + \lambda^2
\end{aligned}$$

Simply plotting the equation we can see the roots are $\lambda_1 = 0.08$ and $\lambda_2 = 0.4$. Now we can calculate the eigenvectors by solving $\Sigma e = \lambda e$ for all eigenvalues, as there are infinitely many solutions we simply use the proportions to create a eigenvector and then reduce it to have unit vector length:

$$\begin{aligned}
&\begin{bmatrix} 0.24 & -0.16 \\ -0.16 & 0.24 \end{bmatrix} \begin{bmatrix} e_{1,1} \\ e_{1,2} \end{bmatrix} = 0.08 \begin{bmatrix} e_{1,1} \\ e_{1,2} \end{bmatrix} \rightarrow \\
&\quad 0.24e_{1,1} - 0.16e_{1,2} = 0.08e_{1,1} \\
&-0.16e_{1,1} + 0.24e_{1,2} = 0.08e_{1,2} \rightarrow \\
&\quad -0.16e_{1,2} = 0.08e_{1,1} - 0.24e_{1,1} = -0.16e_{1,1} \Leftrightarrow e_{1,2} = e_{1,1}
\end{aligned}$$

Meaning $e_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ is a eigenvector for the eigenvalue $\lambda_1 = 0.08$, now we do the same for λ_2:

$$\begin{aligned}
&\begin{bmatrix} 0.24 & -0.16 \\ -0.16 & 0.24 \end{bmatrix} \begin{bmatrix} e_{1,1} \\ e_{1,2} \end{bmatrix} = 0.4 \begin{bmatrix} e_{1,1} \\ e_{1,2} \end{bmatrix} \rightarrow \\
&\quad 0.24e_{1,1} - 0.16e_{1,2} = 0.4e_{1,1} \\
&-0.16e_{1,1} + 0.24e_{1,2} = 0.4e_{1,2} \rightarrow \\
&\quad -0.16e_{1,2} = 0.4e_{1,1} - 0.24e_{1,1} = 0.16e_{1,1} \rightarrow \\
&\quad -e_{1,2} = e_{1,1} \Leftrightarrow e_{1,2} = -e_{1,1}
\end{aligned}$$

We now have the second eigenvector $e_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$, but they need to be unit vector length so we divide by their current euclidean distance.

$$||e_1|| = \sqrt{1^2 + 1^2} = \sqrt{2}$$
$$||e_2|| = \sqrt{(-1)^2 + 1^2} = \sqrt{2}$$

$$e_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$
$$e_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

Now using the eigenvalues we can calculate how much of the variance is explained by each component and in turn order them. $total = \lambda_1 + \lambda_2 = 0.08 + 0.4 = 0.48$ As λ_2 has the larger value we will swap so that e_1 is now the old e_2 and the same applies for the eigenvalues. The variance explained by each component is then:

$$\lambda_1 = \frac{0.4}{0.48} = 83.33\%$$
$$\lambda_2 = \frac{0.08}{0.48} = 16.67\%$$

To transform the data to the principal components we just need to multiply the centered data by a matrix containing the unit length eigenvectors in order on the columns. $W = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$

$$\begin{aligned} Y_{PC} &= X_{cent} W \\ &= \begin{bmatrix} -\frac{0.2}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{0.2}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{0.2}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ -\frac{0.2}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{0.8}{\sqrt{2}} & 0 \end{bmatrix} \end{aligned}$$

If patient 5 was only treated with drug A all patients would be treated with drug B if and only if they were not treated with drug A, this would mean that the PCA would find one eigenvalue of 0 and one other. This other non-zero eigenvalue would describe 100% of the variance in the dataset, meaning we only need that one feature (PC1) to describe the complete dataset.

Question 4

I assume that by predicting survival it's meant as either you die from cancer or don't by beating it (if they have it of course). One of the reasons for making it binary is that an old person dying from old age with cancer could skew the result a bit towards a shorter survival time prediction. And in the other case a person who beat cancer and lives very long will skew the result heavily in the other direction.

To solve this problem we could use a decision tree, however as there is many features and we don't know the amount of training data it's prone to overfitting. Therefore I would use random forest to classify the data, as this deals better with more features and reduces overfitting. Random forest also makes it so we don't have to do any dimensionality reduction or scaling of the variables like we would have to for other models.

Another approach I thought about was using kNN, however to use kNN it would be easiest to just remove Diagnosis as it's a categorical variable and it seems like AFP describes the exact same thing. To speed up kNN we could try to use PCA on the training data to figure out whether more of the dimensions are redundant. But before doing this we have to normalize the data as AFP has very large difference in values compared to the others. For kNN we would also have to try different k-values and use validation data to make sure we picked a good/the best one.

As kNN and random forest does mostly the same thing, creating decision boundaries, the extra work to use kNN is only worth it if the dataset is very large. Because for very large datasets it's better if we reduce the dimensionality (less features to calculate on), it also helps for large datasets that we can change kNN's distance metric from euclidean to manhattan.

To sum it up I would use random forest classification because it works well if the training data is not too large, which I'm assuming it's not. However if the training data set was very large I would instead do dimension reduction using PCA and use kNN perhaps with manhattan distance metric instead of euclidean.

Question 5

As we were not allowed to use numpy's built in functions for calculating mean, unique values and bin_count, I had to implement my own helper function that counts how many occurrences there is of each outcome.

```
    def count_occurences(data):
        count_array = {}

        for occurence in data:
            if occurence in count_array:
                count_array[occurence] += 1
            else:
                count_array[occurence] = 1
        return count_array
```

The only two packages I'm using is pandas, for reading the csv file into a dictionary, and math as I need to calculate log2. The math package is however a built-in package, and as the exam specified machine-learning packages I assumed math and pandas data reading/parsing is not such.

a)

```
    def entropy(data):
        count = count_occurences(data["HeartDisease"])
        if len(count) < 2: # If there is only one class the leaf is
     pure
            return 0

        sum_count = 0
        for key in count:
            sum_count += count[key]
        pos = count[1] / sum_count # Heart disease
        neg = count[0] / sum_count # No heart disease

        entropy = (-pos) * math.log2(pos) - neg * math.log2(neg)

        return entropy
```

To calculate the entropy we count the occurrences of positive and negative, if there is only positive or negative we return 0 as the leaf is pure. Otherwise we calculate the entropy and return it.

```
    def quality_of_threshold(data, threshold, column_name):
        data_left = data[data[column_name] <= threshold]
        data_right = data[data[column_name] > threshold]
        information_gain = entropy(data) - (len(data_left) / len(
    data)) * entropy(data_left) - (len(data_right) / len(data)) *
    entropy(data_right)
        print("    For feature", column_name, "and the threshold",
    threshold, "(entropy) quality is:", information_gain)
```

```
        return information_gain
```

The function to evaluate the quality of a threshold on a selected feature works by splitting the data on the given threshold and then calculating the information gain. It both prints the found result to the console and returns it. The console output can be seen by running the python script provided in the appendix.

b)

```
    def find_threshold(data, column_name):
        thresholds = []
        for value in data[column_name]:
            if value not in thresholds:
                thresholds.append(value)

        best_threshold = None
        best_quality = 0
        for threshold in thresholds:
            quality = quality_of_threshold(data, threshold,
    column_name)
            if quality > best_quality:
                best_quality = quality
                best_threshold = threshold
        print("  The best threshold for feature", column_name, "is"
    , best_threshold, "with quality", best_quality)
        return best_threshold, best_quality
```

For thresholds I chose to check all unique values in the dataset, I thought about using the mean and check from mean - std to mean + std, however it's hard to figure out how much to increment by when the data is not normalized (as it can differ from feature to feature). If the data had been normalized before processing it could potentially find a better value as it would also consider values between the ages present in the dataset. The full console output can be seen by running the python script provided in the appendix, the function found that 0.53 was the best threshold for age for the ones I checked with an information gain of 0.124.

c)

```
    def find_feature(data, features):
        best_feature = None
        best_quality = 0
        best_threshold = None
        for column_name in data.columns:
            if column_name not in features:
                continue
            threshold, quality = find_threshold(data, column_name)
            print
```

```
            if quality > best_quality:
                best_quality = quality
                best_threshold = threshold
                best_feature = column_name
        print("The best feature is", best_feature, "with threshold"
    , best_threshold, "and quality", best_quality)
        return best_feature, best_threshold, best_quality
```

To find the best feature to do separation on we check all relevant features with the relevant thresholds, as discussed in sub task b, and pick the one with the best information gain. Once all thresholds on all features has been checked the best feature, it's threshold and information gain is printed to the console. The full console output can be seen by running the python script provided in the appendix, the function found that RestingBP was the best feature with the threshold 0.65 which had an information gain of 0.244.

Question 6

As it's not stated for question 6 (as it was for question 5), I chose to interpret it as numpy being allowed for calculating mean and standard deviations.

a)

We can normalize the data using the provided formula, it can be done in python as seen in the following code snippet:

```
    normalized_data = (data - np.mean(data, axis=0)) / np.std(data,
     axis=0)
```

Where data is the data loaded from the csv with only the features stated included.

b)

We can use the implementation from sklearn to do agglomerative clustering for the different linkages, as seen in the following code snippet:

```
complete = AgglomerativeClustering(linkage="complete", n_clusters=
    amount_of_clusters).fit(normalized_data)

average = AgglomerativeClustering(linkage="average", n_clusters=
    amount_of_clusters).fit(normalized_data)

single = AgglomerativeClustering(linkage="single", n_clusters=
    amount_of_clusters).fit(normalized_data)
```

To compare the different linkage types we can compare the average intra and inter cluster distance, I also included the normalized intra cluster distance, the variability scaled according to the mean and the standard deviation. The calculated values can be seen in the following table:

	Complete	Average	Single
Avg intra	4366	5180	7686
Avg normalized intra	1.1795	1.0566	0.3724
Avg inter	3.6513	4.1853	4.8083
$\frac{\sigma^2}{\mu}(Variability)$	0.3442472923849638	0.7164673347072618	1.9392853729077746
σ (STD)	3477	5016.973430266498	8254.0

To compare the cluster sizes we can use the normalized variability, to quickly compare between them. However this only tells us which one of the three has

more balanced cluster sizes. To check how "big" clusters each algorithm makes we can use the standard deviation.

When looking at the normalized variability we can see that Complete is much more balanced than both Average and Single, with Single being the worst. Looking at the standard deviation we can see Complete is pretty balanced as there is 20 thousand points and it "only" deviates by 3477 points. This is also confirmed when looking at the cluster sizes with it having 2 big clusters 1 medium and 2 smaller. Average is also balanced to some extent with a deviation of 5016 it's more compacted into bigger clusters, with it having 2 big clusters and 3 smaller clusters. Lastly Single has a deviation of 8254 indicating that it is not very balanced as it deviates by almost half of the data points. When looking at the specific cluster sizes it has 1 big cluster and 4 clusters consisting of only 1 data point.

This 1 mega cluster for Single makes sense as the data points are grouped closely together, and it merges on closest pair. This means that on the second merge it will check all pairs against the two points inside the first cluster, and it's more likely that a point is close to one of two than to only one. This effect multiplies quickly and this is the reason Single linkage found one big cluster in this dataset.

We would expect Complete and Average linkage to have more tightly packed clusters and Single to have more loosely packed clusters, looking at the normalized intra cluster distance it would seem the points in Single is closer, however this is mostly due to it clustering almost all points into 1 cluster (dividing by amount of points in cluster). If we don't normalize we can see Single has a much bigger intra cluster distance, which is expected when it clusters like it does. Both Complete and Average are pretty close in terms of intra cluster distance.

Looking at the inter cluster distance we can see that Average is close to the average of Complete and Single, this is expected as it tries to minimize the average distance between all points in each cluster. And as expected the inter cluster distance of Single is larger, as it calculates the distance between centroids and Single has one large cluster.

c)

To plot the clustering for each linkage, we can create a helper function and use it as seen in the following code snippet:

```
1 def plot_clusters(data, labels, centroids, title):
2     plt.scatter(data[:,1], data[:,2], c=labels)
3     plt.scatter(centroids[:,1], centroids[:,2], c='red', marker='x'
      , s=100)
4     plt.xlabel("Latitude")
5     plt.ylabel("Longitude")
```

```
    plt.title(title)
    plt.savefig(title + ".png")

plot_clusters(normalized_data, complete.labels_, centroids_complete
    , "Complete linkage")
plot_clusters(normalized_data, average.labels_, centroids_average,
    "Average linkage")
plot_clusters(normalized_data, single.labels_, centroids_single, "
    Single linkage")
```

Looking at the plots it seems that Average and Complete both cluster data pretty well (keeping in mind there is two more dimensions we cannot see on this plot). Whereas it seems that Single struggles to cluster it, this happens because two points are merged and then the next point with shortest distance keeps being closest to a point in the large cluster.

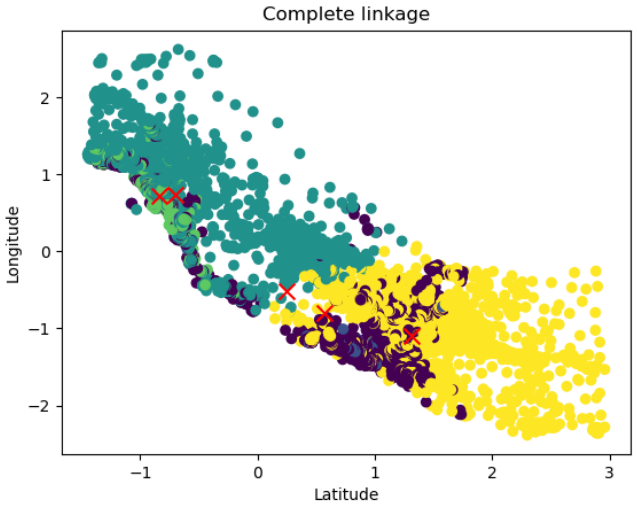

Figure 1: Plot showing the clustering with 5 clusters using complete linkage against two dimensions

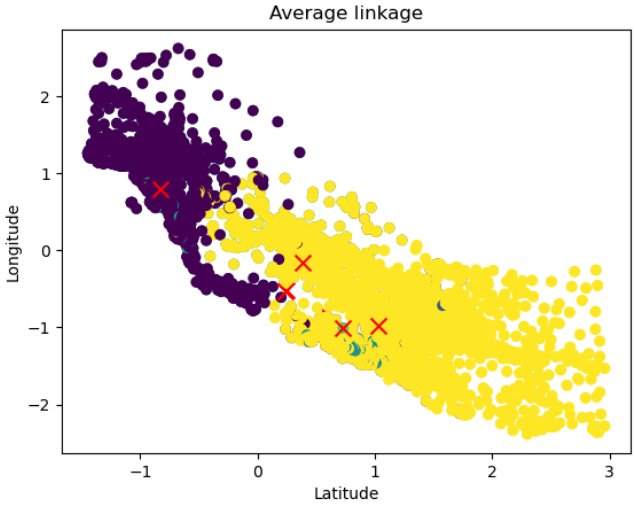

Figure 2: Plot showing the clustering with 5 clusters using average linkage against two dimensions

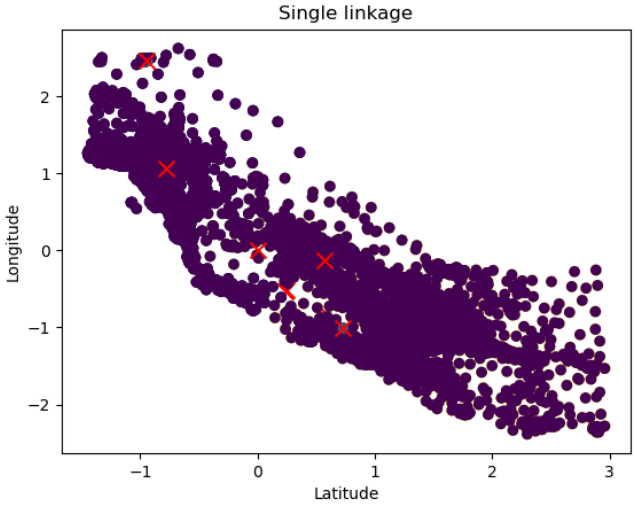

Figure 3: Plot showing the clustering with 5 clusters using single linkage against two dimensions

Appendix

2.py

```
import numpy as np
import scipy.stats

print(scipy.stats.t.ppf(0.975, 9))
```

5.py

```
import pandas as pd
import math

data = pd.read_csv('../heart_simplified_RandomForest.csv')

# Helper functions
def count_occurences(data):
    count_array = {}

    for occurence in data:
        if occurence in count_array:
            count_array[occurence] += 1
        else:
            count_array[occurence] = 1
    return count_array

# Subtask a
def entropy(data):
    count = count_occurences(data["HeartDisease"])
    if len(count) < 2: # If there is only one class the leaf is
    pure
        return 0

    sum_count = 0
    for key in count:
        sum_count += count[key]
    pos = count[1] / sum_count # Heart disease
    neg = count[0] / sum_count # No heart disease

    entropy = (-pos) * math.log2(pos) - neg * math.log2(neg)

    return entropy

def quality_of_threshold(data, threshold, column_name):
    # Split the data into two parts
    data_left = data[data[column_name] <= threshold]
    data_right = data[data[column_name] > threshold]
    information_gain = entropy(data) - (len(data_left) / len(data))
     * entropy(data_left) - (len(data_right) / len(data)) * entropy
    (data_right)
    print("    For feature", column_name, "and the threshold",
    threshold, "(entropy) quality is:", information_gain)
    return information_gain
```

```
40
41 print("=======================================")
42 print("Subtask a")
43 print("=======================================")
44 quality_of_threshold(data, sum(data["Age"]) / len(data["Age"]), "
    Age")
45 quality_of_threshold(data, sum(data["RestingBP"])  / len(data["
    RestingBP"]), "RestingBP")
46 quality_of_threshold(data, sum(data["Cholesterol"])  / len(data["
    Cholesterol"]), "Cholesterol")
47 quality_of_threshold(data, sum(data["MaxHR"])  / len(data["MaxHR"])
    , "MaxHR")
48
49 # Subtask b
50 def find_threshold(data, column_name):
51     thresholds = []
52     for value in data[column_name]:
53         if value not in thresholds:
54             thresholds.append(value)
55
56     best_threshold = None
57     best_quality = 0
58     for threshold in thresholds:
59         quality = quality_of_threshold(data, threshold, column_name
    )
60         if quality > best_quality:
61             best_quality = quality
62             best_threshold = threshold
63     print("  The best threshold for feature", column_name, "is",
    best_threshold, "with quality", best_quality)
64     return best_threshold, best_quality
65
66 print("=======================================")
67 print("Subtask b")
68 print("=======================================")
69 find_threshold(data, "Age")
70
71 # Subtask c
72 def find_feature(data, features):
73     best_feature = None
74     best_quality = 0
75     best_threshold = None
76     for column_name in data.columns:
77         if column_name not in features:
78             continue
79         threshold, quality = find_threshold(data, column_name)
80         print
81         if quality > best_quality:
82             best_quality = quality
83             best_threshold = threshold
84             best_feature = column_name
85     print("The best feature is", best_feature, "with threshold",
    best_threshold, "and quality", best_quality)
86     return best_feature, best_threshold, best_quality
87
88 print("=======================================")
89 print("Subtask c")
```

```
print("=======================================")
find_feature(data, ["Age", "RestingBP", "Cholesterol", "MaxHR"])
```

6.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering


data = pd.read_csv('../housing.csv') # Load the data

unprocessed_data = data.to_numpy() # Convert the data to a numpy array

data = unprocessed_data[:,[0,6,7,8]] # Select MedInc, Latitude, Longitude, MedHouseVal columns

normalized_data = (data - np.mean(data, axis=0)) / np.std(data, axis=0) # Normalize the data
amount_of_clusters = 5

complete = AgglomerativeClustering(linkage="complete", n_clusters=amount_of_clusters).fit(normalized_data)
average = AgglomerativeClustering(linkage="average", n_clusters=amount_of_clusters).fit(normalized_data)
single = AgglomerativeClustering(linkage="single", n_clusters=amount_of_clusters).fit(normalized_data)

def calculate_intra_dist(centroids, data, labels, n_clusters, normalize=True):
    intra_cluster_distance = []
    for i in range(n_clusters):
        points_in_cluster = data[labels == i]
        centroid = centroids[i]

        dist = np.linalg.norm(points_in_cluster - centroid, axis=1)
        norm_dist = dist / len(points_in_cluster) # normalize the distance by number of points in cluster

        if normalize:
            to_append = norm_dist
        else:
            to_append = dist

        intra_cluster_distance.append(np.sum(to_append))
    return intra_cluster_distance

def calculate_inter_dist(centroids, n_clusters):
    inter_cluster_distance = []
    for i in range(n_clusters):
        for j in range(i+1, n_clusters): # i+1 to avoid calculating the same distance twice (i,j) and (j,i)
            inter_cluster_distance.append(np.linalg.norm(centroids[i] - centroids[j]))
```

```
    return inter_cluster_distance

def calculate_var_cluster_size(labels):
    return np.var(np.bincount(labels))

def calculate_std_cluster_size(labels):
    return np.std(np.bincount(labels))

centroids_complete = np.array([normalized_data[complete.labels_ ==
    i].mean(axis=0) for i in range(amount_of_clusters)])
centroids_average = np.array([normalized_data[average.labels_ == i
    ].mean(axis=0) for i in range(amount_of_clusters)])
centroids_single = np.array([normalized_data[single.labels_ == i].
    mean(axis=0) for i in range(amount_of_clusters)])

norm_intra_dist_complete = calculate_intra_dist(centroids_complete,
     normalized_data, complete.labels_, amount_of_clusters)
norm_intra_dist_average = calculate_intra_dist(centroids_average,
    normalized_data, average.labels_, amount_of_clusters)
norm_intra_dist_single = calculate_intra_dist(centroids_single,
    normalized_data, single.labels_, amount_of_clusters)

intra_dist_complete = calculate_intra_dist(centroids_complete,
    normalized_data, complete.labels_, amount_of_clusters,
    normalize=False)
intra_dist_average = calculate_intra_dist(centroids_average,
    normalized_data, average.labels_, amount_of_clusters, normalize
    =False)
intra_dist_single = calculate_intra_dist(centroids_single,
    normalized_data, single.labels_, amount_of_clusters, normalize=
    False)

inter_dist_complete = calculate_inter_dist(centroids_complete,
    amount_of_clusters)
inter_dist_average = calculate_inter_dist(centroids_average,
    amount_of_clusters)
inter_dist_single = calculate_inter_dist(centroids_single,
    amount_of_clusters)

var_cluster_size_complete = calculate_var_cluster_size(complete.
    labels_)
var_cluster_size_average = calculate_var_cluster_size(average.
    labels_)
var_cluster_size_single = calculate_var_cluster_size(single.labels_
    )

std_cluster_size_complete = calculate_std_cluster_size(complete.
    labels_)
std_cluster_size_average = calculate_std_cluster_size(average.
    labels_)
std_cluster_size_single = calculate_std_cluster_size(single.labels_
    )

# Normalize the variability in cluster size by the mean
var_mean = np.mean([var_cluster_size_complete,
    var_cluster_size_average, var_cluster_size_single])
var_cluster_size_complete /= var_mean
```

```
var_cluster_size_average /= var_mean
var_cluster_size_single /= var_mean

print("==================================================")
print("Complete linkage:")
print("==================================================")
print("Avg intra cluster distance:", np.mean(intra_dist_complete))
print("Avg normalised intra cluster distance:", np.mean(
    norm_intra_dist_complete))
print("Inter cluster distance:", np.mean(inter_dist_complete))
print("Variability in cluster size divided by the mean:",
    var_cluster_size_complete )
print("Standard deviation in cluster size:",
    std_cluster_size_complete)
print("==================================================")
print("Average linkage:")
print("==================================================")
print("Avg intra cluster distance:", np.mean(intra_dist_average))
print("Avg normalised intra cluster distance:", np.mean(
    norm_intra_dist_average))
print("Inter cluster distance:", np.mean(inter_dist_average))
print("Variability in cluster size divided by the mean:",
    var_cluster_size_average)
print("Standard deviation in cluster size:",
    std_cluster_size_average)
print("==================================================")
print("Single linkage:")
print("==================================================")
print("Avg intra cluster distance:", np.mean(intra_dist_single))
print("Avg normalised intra cluster distance:", np.mean(
    norm_intra_dist_single))
print("Inter cluster distance:", np.mean(inter_dist_single))
print("Variability in cluster size divided by the mean:",
    var_cluster_size_single)
print("Standard deviation in cluster size:",
    std_cluster_size_single)
mean = np.mean([std_cluster_size_complete, std_cluster_size_average
    , std_cluster_size_single])

def plot_clusters(data, labels, centroids, title):
    plt.scatter(data[:,1], data[:,2], c=labels)
    plt.scatter(centroids[:,1], centroids[:,2], c='red', marker='x'
    , s=100)
    plt.xlabel("Latitude")
    plt.ylabel("Longitude")
    plt.title(title)
    plt.savefig(title + ".png")

plot_clusters(normalized_data, complete.labels_, centroids_complete
    , "Complete linkage")
plot_clusters(normalized_data, average.labels_, centroids_average,
    "Average linkage")
plot_clusters(normalized_data, single.labels_, centroids_single, "
    Single linkage")
```