Exercise 1 / Autumn 2022

1.1 Cows with numerical and categorical features

Learning goals: How to use distance/similarity measures when there are both numerical and categorical features in data; similarity graphs.

Look at the cow data in Table 1. The task is to evaluate distances between cows. Note that field 'name' is the cow identifier and not used in any distance calculations. You can calculate distances manually or make scripts, but **implement the distance measures yourself** (do not use ready-made library functions). You can use library functions for min, max, mean, and standard deviation, if you want. Remember to report intermediate steps and prepare to show your code and its outputs.

Table 1: Cow data: name, race, age (years), daily milk yield (litres/day), character and music taste.

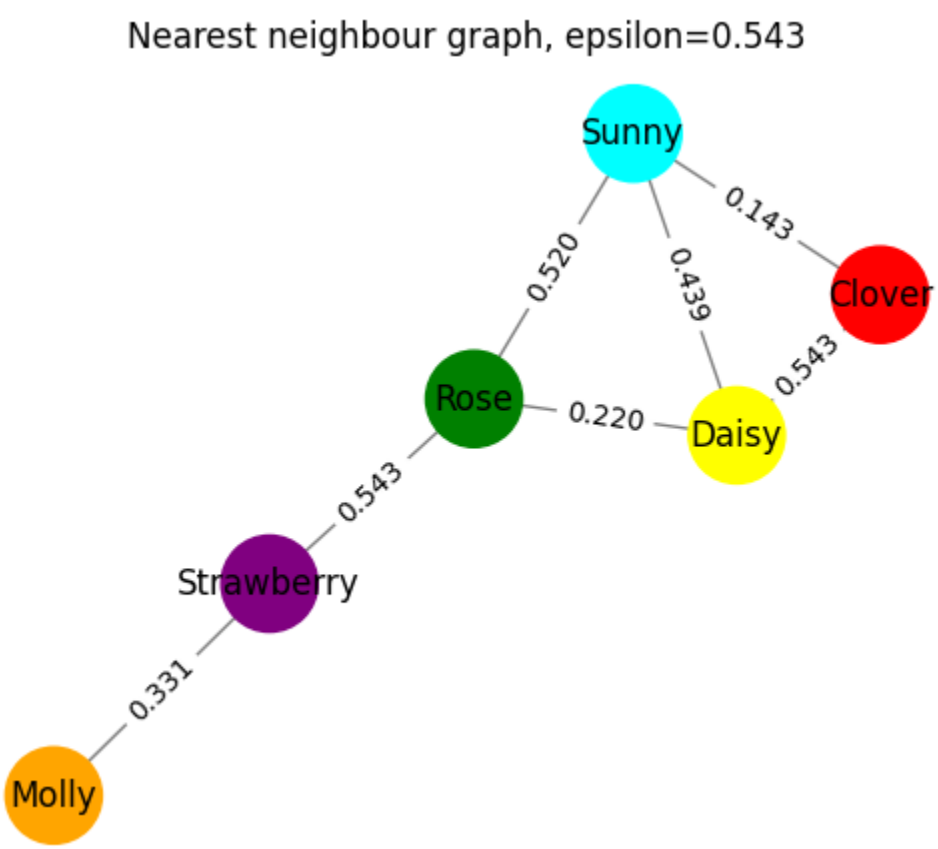
name	race	age	milk	character	music
Clover	Holstein	2	10	calm	rock
Sunny	Ayrshire	2	15	lively	country
Rose	Holstein	5	20	calm	classical
Daisy	Ayrshire	4	25	kind	rock
Strawberry	Finncattle	7	35	calm	classical
Molly	Ayrshire	8	45	kind	country

a) In this part, use only numerical features. Scale the features with the min-max scaling described in the book (Aggarwal section 2.3.3) and calculate pairwise Euclidean distances (L_2 norm) between cows. Present the results as a nearest neighbour graph (as described in Lecture 1 and Aggarwal Sec. 2.2.2.9). Select the threshold ϵ as small as possible still keeping the graph connected.

The code for Task (a)

```
import pandas as pd
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
# Read the data
df = pd.read_csv('cowdata.csv', header=None)
df.columns = ['name', 'race', 'age', 'milk', 'character', 'music']

names = df['name'].to_numpy()
races = df['race'].to_numpy()
ages = df['age'].to_numpy()
milks = df['milk'].to_numpy()
characters = df['character'].to_numpy()
musics = df['music'].to_numpy()

def min_max_scaling(data):
    min_val = np.min(data)
    max_val = np.max(data)
    scaled_arr = (data - min_val) / (max_val - min_val)
    return scaled_arr

scaled_ages = min_max_scaling(ages)
scaled_milks = min_max_scaling(milks)

def create_nearest_neighbor_graph(squared_distances, epsilon):
    n = squared_distances.shape[0]
    G = nx.Graph()
    for i in range(n):
        for j in range(i+1, n):
            if squared_distances[i, j] <= epsilon:
                G.add_edge(i, j, weight=squared_distances[i, j])
    return G

def pairwise_Euclidean(X, Y):
    n = len(X)
    square_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
```

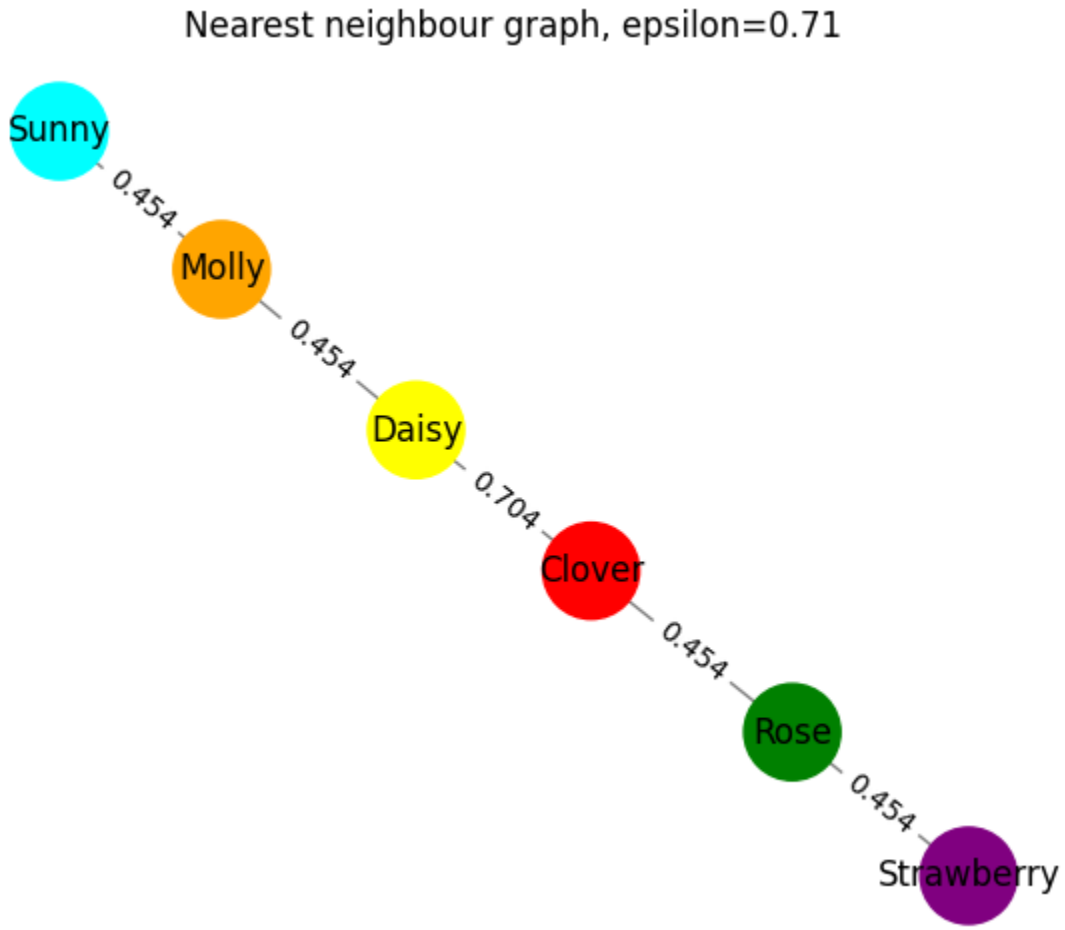
```
            square_matrix[i, j] = np.sqrt((X[i] - X[j])**2 + (Y[i] -
Y[j])**2)
    return square_matrix

Euclidean_distances = pairwise_Euclidean(scaled_ages, scaled_milks)
print(Euclidean_distances)
epsilon = 0.543
# Create the nearest neighbor graph
G = create_nearest_neighbor_graph(Euclidean_distances, epsilon)
# visualize the graph
label_dict = {i: name for i, name in enumerate(names)}
colors = ['red', 'cyan', 'green', 'yellow', 'purple', 'orange']
layout = nx.spring_layout(G, k=0.05)
plt.figure(figsize=(5, 4))
nx.draw(G, pos=layout, with_labels=True, nodelist=list(label_dict.keys()),
labels=label_dict, node_color=colors, edge_color='gray', node_size=1200)
edge_labels = {(i, j): f"{G[i][j]['weight']:.3f}" for i, j in G.edges()}
nx.draw_networkx_edge_labels(G, pos=layout, edge_labels=edge_labels)
plt.title(f"Nearest neighbour graph, epsilon={epsilon}")
plt.show()
```



b) In this part, use only categorical features. First, define Goodall distance measure d_G from the Goodall similarity measure G with $d_G = 1 - G$. The Goodall similarity measure is presented in Aggarwal sec. 3.2.2 and the slides of lecture 2 (use that version, since there are many alternative Goodall measures). Then calculate pairwise Goodall distances and present the results as a nearest neighbour graph, once again selecting minimal ϵ such that the graph remains connected.

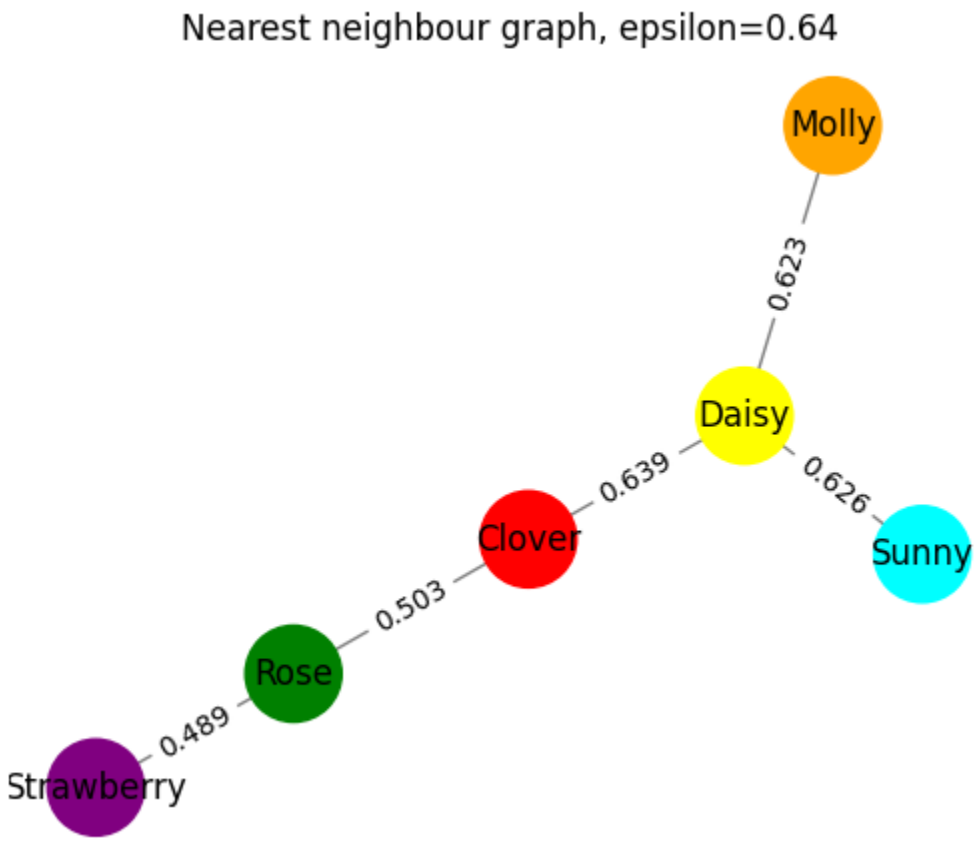
```
def goodall_similarity(dataList):
    n = len(dataList[0])
    total_similarities_score = np.zeros((n, n))
    for data in dataList:
        # Filter out data into categories
        unique_categories = np.unique(data)
        fraction_of_records = {category: np.sum(data == category)/n for
category in unique_categories}
        #print(fraction_of_records)
        similarity_matrix = np.zeros((n, n))
        for i in range(n):
            for j in range(n):
                if data[i] == data[j]:
                    fraction = fraction_of_records[data[i]]
                    similarity_matrix[i, j] = 1 - fraction**2
        #print(similarity_matrix)
        total_similarities_score += similarity_matrix
    average_similarity_matrix = total_similarities_score/len(dataList)
    return average_similarity_matrix

goodall_similarity_matrix = goodall_similarity([races, characters,
musics])
goodall_distances = 1 - goodall_similarity_matrix
print(goodall_distances)
epsilon = 0.71
# Create the nearest neighbor graph
G = create_nearest_neighbor_graph(goodall_distances, epsilon)
# visualize the graph
label_dict = {i: name for i, name in enumerate(names)}
colors = ['red', 'cyan', 'green', 'yellow', 'purple', 'orange']
layout = nx.spring_layout(G, k=0.05)
```

```
plt.figure(figsize=(5.5, 4.5))
nx.draw(G, pos=layout, with_labels=True, nodelist=list(label_dict.keys()),
labels=label_dict, node_color=colors, edge_color='gray', node_size=1200)
edge_labels = {(i, j): f"{G[i][j]['weight']:.3f}" for i, j in G.edges()}
nx.draw_networkx_edge_labels(G, pos=layout, edge_labels=edge_labels)
plt.title(f"Nearest neighbour graph, epsilon={epsilon}")
plt.show()
```



c) In this part, use both numerical and categorical features. Create a distance measure that combines the previous distance measures (L_2 and d_G) using Equation 3.9 in the book (Aggarwal sec. 3.2.3). (Note that Aggarwal gives similarity measure, but you can combine distance measures in the same manner.) Set λ as the proportion of numerical features. It is recommended to use the unbiased estimate of standard deviation. Create now a nearest neighbour graph using the combined measure and select minimal ϵ that keeps the graph connected.

```
lambda_value = 2/5
weighted_distances = lambda_value * Euclidean_distances + (1 -
lambda_value) * goodall_distances
print(weighted_distances)
epsilon = 0.64
# Create the nearest neighbor graph
G = create_nearest_neighbor_graph(weighted_distances, epsilon)
# visualize the graph
label_dict = {i: name for i, name in enumerate(names)}
colors = ['red', 'cyan', 'green', 'yellow', 'purple', 'orange']
layout = nx.spring_layout(G, k=0.05)
plt.figure(figsize=(5, 4))
nx.draw(G, pos=layout, with_labels=True, nodelist=list(label_dict.keys()),
labels=label_dict, node_color=colors, edge_color='gray', node_size=1200)
edge_labels = {(i, j): f"{G[i][j]['weight']:.3f}" for i, j in G.edges()}
nx.draw_networkx_edge_labels(G, pos=layout, edge_labels=edge_labels)
plt.title(f"Nearest neighbour graph, epsilon={epsilon}")
plt.show()
```



d) Compare the results. Is the combined measure graph (c) more similar to the numerical (a) or categorical (b) measure graph? Can you explain why?

The combined measure graph in (c) is much more similar to the categorical measure graph. Here are the few reasons:

- There are fewer numerical features than categorical features, so the influence of the numerical distances on the combined distance will be reduced.
- There are only a few categories in each categorical variable, which greatly influence the distance when two cows share the same categorical data in any dimension

1.2 Similarity in social media profiles

Learning goal: To study distance functions and metrics for set form data.

Consider a social network where each user is associated with a set of labels that best describe a set of properties of the user. We define the profile of the user to be the set of associated labels, i.e., given a set $P = \{p_1, \ldots, p_n\}$ of user profiles and a universe of labels $L = \{l_1, \ldots, l_m\}$, each profile $p_i \in P$ is a set of labels $L_i \subseteq L$. The task is to design functions to measure the distance between two labels and similarity between two user profiles.

a) Propose a distance measure between labels, more precisely, given any two labels $l_1, l_2 \in L$, present a distance function d such that $d(l_1, l_2)$ returns a distance measure between labels l_1 and l_2. The distance function should be (i) intuitive and (ii) satisfy the metric properties (see next parts).

Given that we know no semantic meanings of each label, we can only determine the distance between two labels based on their co-occurrence in user profiles. If two labels frequently appear together in user profiles, they are likely to be closely related and should have a small distance between them. Conversely, if they rarely or never appear together, they are likely unrelated and should have a larger distance.

For example, on Linkedin Profiles, the label "Data Science" and "Machine Learning" are likely to appear together, while the label "Data Science" and "Music" are unlikely to appear together. Therefore, the distance between "Data Science" and "Machine Learning" should be smaller than the distance between "Data Science" and "Music".

In other words, the distance function $d(l_1, l_2)$ should be inversely proportional to the number of profiles containing both l_1 and l_2 and directly proportional to the number of profiles containing either l_1 or l_2.

Let's define $P(L_i)$ as the set of profiles containing label L_i and N as the total number of profiles.

My distance function for two labels l_1 and l_2 is defined as follows:

$$d(L_1, L_2) = \begin{cases} |(P(L_1) \cup P(L_2)| - |P(L_1) \cap P(L_2)| + 1)/N & \text{if } L_1 \neq L_2 \\ |(P(L_1) \cup P(L_2)| - |P(L_1) \cap P(L_2)|)/N & \text{if } L_1 = L_2 \end{cases}$$

b) Discuss the intuition, strengths, and limitations of your measure.

$|P(L_1) \cup P(L_2)|$ is the number of profiles containing either L_1 or L_2, and $|P(L_1) \cap P(L_2)|$ is the number of profiles containing both L_1 and L_2. Therefore, $|P(L_1) \cup P(L_2)|$ - $|P(L_1) \cap P(L_2)|$ is the number of profiles containing either L_1 or L_2 but not both. This number is a good measure of how different L_1 and L_2 are. The larger the number, the more distant L_1 and L_2 becomes. The plus 1 term is added to avoid the case where L_1 and L_2 are the same label but $|P(L_1) \cup P(L_2)|$ - $|P(L_1) \cap P(L_2)|$ is 0, which violates the coincidence axiom of metric space. The division by N is to normalize the distance to be between 0 and 1. This helps to take into account the size of the dataset. For example, if we have a very large dataset, the number of profiles containing either L_1 or L_2 but not both will be large, and the distance between L_1 and L_2 will be large. However, if we have a very small dataset, the number of profiles containing either L_1 or L_2 but not both will be small, and the distance between L_1 and L_2 will be small. This is not desirable because the distance between L_1 and L_2 should be the same regardless of the size of the dataset.

Strength: The distance function is intuitive and easy to understand. It is also easy to compute.

Limitation: I haven't seen any obvious limitations

c) Prove that your distance function is a metric. Depending on your measure, this can be tricky, but study at least the easy properties!

1. Non-negativity $d(L_1, L_2) \geq 0$

This is true since the union $|P(L_1) \cup P(L_2)|$ is always greater than or equal the intersection $|P(L_1) \cap P(L_2)|$ according to set theories

2. Coincidence axiom:

$d(L_1, L_2) = 0$ if and only if $L_1 = L_2$

The union and intersection of the profiles are always equal if $L_1 = L_2$.

Question: when is the case when $L_1 \neq L_2$ but $|P(L_1) \cup P(L_2)|$ - $|P(L_1) \cap P(L_2)|$ = 0?

Answer: when both labels always co-occur together in the same profiles but never appear alone in any other profiles.

This is prevented by the plus 1 term in the distance function. If $L_1 = L_2$, then $|P(L_1) \cup P(L_2)|$ - $|P(L_1) \cap P(L_2)|$ is 0, but the plus 1 term makes the distance 1.

3. Symmetry: $d(L_1, L_2) = d(L_2, L_1)$

This is true since the union and intersection of the profiles are commutative.

4. Triangle Inequality:

$$d(L_1, L_3) + d(L_2, L_3) \geq d(L_1, L_2) \quad \forall L_1, L_2, L_3 \in L$$

Let's denote the size of the union as U and intersection as I. Then we need to prove that

$$U_{13} - I_{13} + U_{23} - I_{23} \geq U_{12} - I_{12}$$

Case of disjoint sets: If sets 1, 2, and 3 are all disjoint (i.e., they have no elements in common), then all intersections are empty, and the inequality simplifies to:

$$U_{13} + U_{23} \geq U_{12}$$

Case of Overlapping Sets: Let's consider the worst-case scenario where all three sets overlap. This means the union is equal to the intersection. Then the inequality simplifies to:

$$2U_{13} + 2U_{23} \geq 2U_{12}$$
$$U_{13} + U_{23} \geq U_{12}$$

This is true with the same reasoning as the case of disjoint sets.

d) Now we want to compare the similarity of two user profiles. Propose an appropriate function $s(p_1, p_2)$ to compute the similarity of any two profiles $p_1, p_2 \in P$ and discuss its intuition.

We can use the Jaccard similarity to measure the similarity between two user profiles. The Jaccard similarity is defined as the size of the intersection divided by the size of the union of the two sets of labels belonging to each user. In other words, it is the number of common labels divided by the total number of labels in both profiles.

$$s(p_1, p_2) = \frac{|p_1 \cap p_2|}{|p_1 \cup p_2|}$$

e) Be prepared to show code that implements d and s and demonstrate its behavior with a small set of toy data.

```
# Implementing the distance measure d for labels

def d(L1, L2, profiles):
    numProfilesContainingBothL1andL2 = 0
    numProfilesContainingL1OrL2 = 0
    numProfiles = len(profiles)
    for user in profiles:
        profileLabels = profiles[user]
        if L1 in profileLabels and L2 in profileLabels:
            numProfilesContainingBothL1andL2 += 1
        if L1 in profileLabels or L2 in profileLabels:
            numProfilesContainingL1OrL2 += 1

    print(f"numProfiles Containing {L1} or {L2}: ",
numProfilesContainingL1OrL2)
    print(f"numProfiles Containing Both {L1} and {L2}: ",
numProfilesContainingBothL1andL2)

    if L1 != L2:
        return (numProfilesContainingL1OrL2 -
numProfilesContainingBothL1andL2 + 1)/numProfiles
    else:
        return (numProfilesContainingL1OrL2 -
numProfilesContainingBothL1andL2)/numProfiles

# Similarity of two user profiles: Jaccard similarity

def s(p1, p2):
    intersection = len(p1.intersection(p2))
    union = len(p1.union(p2))
    return intersection / union

# Toy data
labels = {"A", "B", "C", "D", "E", "F", "G"}  # A universe of 5 labels

profiles = {
    'user1': {"A", "B", "C"},
```

```
    'user2': {"A", "B", "D"},
    'user3': {"A", "D", "E"},
    'user4': {"E"},
    'user5': {"C", "D", "E"},
    'user6': {"A", "B", "C", "D", "E"},
    'user7': {"F", "G", "D"},
    'user8': {"F", "G", "E"},
}

# Demonstrate behavior
print("Distance between labels A and B:", d("A", "B", profiles),"\n")
print("Distance between labels A and C:", d("A", "C", profiles),"\n")
print("Distance between labels A and A:", d("A", "A", profiles),"\n")
print("Distance between labels F and G:", d("F", "G", profiles),"\n")

print("Similarity between user1 and user2 profiles:", s(profiles['user1'],
profiles['user2']))
print("Similarity between user1 and user3 profiles:", s(profiles['user1'],
profiles['user3']))
print("Similarity between user2 and user4 profiles:", s(profiles['user2'],
profiles['user4']))
```

```
numProfiles Containing A or B:  4
numProfiles Containing Both A and B:  3
Distance between labels A and B: 0.25

numProfiles Containing A or C:  5
numProfiles Containing Both A and C:  2
Distance between labels A and C: 0.5

numProfiles Containing A or A:  4
numProfiles Containing Both A and A:  4
Distance between labels A and A: 0.0

numProfiles Containing F or G:  2
numProfiles Containing Both F and G:  2
Distance between labels F and G: 0.125

Similarity between user1 and user2 profiles: 0.5
Similarity between user1 and user3 profiles: 0.2
Similarity between user2 and user4 profiles: 0.0
```

1.3 Lower bounding a distance

Learning goal: To consider effective implementations of nearest neighbor search.

Consider the *nearest neighbor search problem*: Given a dataset of n objects $X = \{x_1, \ldots, x_n\}$ and a query object q, we want to find the object $x^* \in X$ that minimizes the distance $d(q, x)$, that is,

$$d(q, x^*) \leq d(q, x) \quad \forall x \in X \; . \tag{1}$$

Assume that computing the distance function d is *very expensive*. Assume now that we are able to define another distance function d_{LB}, which is a *lower bound* of distance d. This means that for all pairs of objects x and y it should be

$$d_{\text{LB}}(x, y) \leq d(x, y) \; . \tag{2}$$

Furthermore, assume that computing d_{LB} is *significantly more efficient* than computing d.

a) Write pseudocode of an algorithm for the nearest neighbor search using distance d.

```
def d(x, y):
    # expensive distance function
    return <d_value>

def NearestNeighborSearch(X, q):
    min_distance = inf
    nearest_neighbor = None

    for each object x_i in X:
        dist = d(x_i, q)
        ff dist < min_distance:
            min_distance = dist
            nearest_neighbor = x
    return nearest_neighbor
```

b) Explain how to use the lower-bound distance d_{LB} to speed up the search algorithm of the previous part. Write pseudocode for the modified search algorithm.

We can use the Lower Bound distance to enable Early Stopping

If the lower-bound distance $d_{LB}(x, q)$ for an object x is larger than d(y, q), where y is another object, then x cannot be nearer to q than y is due to the inequality

$$d(x, q) \geq d_{LB}(x, q) > d(y, q)$$

Therefore, we can skip computing d(x, q).

How to incorporate the lower-bound distance into the search algorithm?:

First, we compute the lower-bound distance for all objects in X. Then, we sort the objects in X according to their lower-bound distances, and call it X_sorted. Finally, we use X_sorted to perform the nearest neighbor search. If the lower-bound distance of an object x_i during the search is larger than the current true distance of the current nearest neighbor, then we can stop the search, since we know that all remaining objects in X_sorted are farther away from q than the current nearest neighbor due to the inequality above

The pseudocode:

```
def d(x, y):
    # expensive distance function
    return <d_value>

def d_LB(x, y):
    # cheap lower bound distance function
    return <d_LB_value>
```

```
def NearestNeighborSearchLB(X, q):
    """
    Arguments:
        X: Data set of N objects {x1,...,xn}
        q: Query object
    Return:
        x*: Object in X that is nearest to q based on distance d
    """

    # Initialize an array to store the lower-bound distances
    X_dLB = []

    # compute lower-bound distances
    for each object x_i in X:
        dist_LB = d_LB(x_i, q)
        X_dLB.append(dist_LB)
```

```
# sort the objects in X according to their lower-bound distances
X_zip_dLB = zip(X, X_dLB)
X_zip_dLB_sorted = sorted(X_zip_dLB, key=x[1]) # sort X by dLB

min_distance = inf
nearest_neighbor = None

# We perform the search on the sorted X based on dLB
for each tuple x_i, dLB in X_zip_dLB_sorted:
    if dLB > min_distance:
        break
        # early stopping, since all remaining objects are
        # farther away than the current nearest neighbor

    # Important: only calculate d if dLB <= min_distance
    dist = d(x_i, q)
    if dist < min_distance:
        min_distance = dist
        nearest_neighbor = x_i

return nearest_neighbor
```

c) What is a desirable property for the lower-bound distance d_{LB} to be as effective as possible for the modified algorithm? Explain why.

Three properties are desirable for the lower-bound distance dLB to be as effective as possible for the modified algorithm:

1. The lower-bound distance should be as close as possible to the true distance. Good d_{LB} will help as early as possible pruning data points that are far away from the query point.
2. The lower-bound distance should be as cheap as possible to compute. Otherwise, the cost of computing the lower-bound distance will be too high and will not help to speed up the search.
3. The lower-bound distance should be unique for each object as much as possible. Otherwise, the sorting step becomes trivial and we cannot carry out much pruning.