Report

Problem 2.3

$$median = L_1 + \frac{\left(\frac{n}{2} - (\sum freq)_l\right)}{freq_{median}} * width$$

1. = 20 + (3194/2 - 950) / 1500 * (50 - 20)

= 32.94

Problem 2.6

Code: 2.py

(a) Euclidean distance = $\sqrt{\sum(x_i - y_i)^2}$

= sqrt ((22-20)^2 + (1-0)^2 + (42-36)^2 + (10-8)^2)

= 6.7082

(b) Manhattan distance = $\sum |x_i - y_i|$

= |22-20|+ |1-0| + |42-36|+ |10-8|

= 11

(c) Minkowski distance (q = 3) = $\sqrt[3]{\sum(x_i - y_i)^3}$

= pow (((22-20)^3 + (1-0)^3+ (42-36)^3 + (10-8)^3)), 1/3)

= 6.1534

(d) Supremum distance = $\max(|x_i - y_i|)$

= max (|22-20|, |1-0|, |42-36|, |10-8|)

= 6

Problem 2.7

Firstly, and most simply, we can sort all data and find the median in the middle of ordered data values. This is the most time and space consuming method. If we use heap sort, the average time complexity and space complexity would be O(nlogn) and O(nlogn).

Secondly, we can group our data into intervals to avoid ordering the whole dataset. The following method described in the textbook would be a good one:

$$median = L_1 + \frac{\left(\frac{n}{2} - (\sum freq)_l\right)}{freq_{median}} * width$$

However, there is a trade-off between how many intervals we split the dataset and the approximation accuracy. Which means, if we choose a smaller number of intervals, the time we need to separate data into groups would be fewer. But as it is indicated in the formula above, the median resolution will drop accordingly.

So, lastly, we can apply a smarter method, which is split the data into few intervals and find which one the median is in, and iteratively split this interval until we get to the needed accuracy. By doing so, we avoid looping over the whole dataset and arrive wanted resolution at the same time.

Problem 2.8

Code: 2_6_8.py

Part a

Euclidean distance = $\sqrt{\sum(x_i - y_i)^2}$

Manhattan distance = $\sum |x_i - y_i|$

Supremum distance = $\max(|x_i - y_i|)$

Cosine similarity = $\frac{x^t \cdot y}{||x||||y||}$

Using the formulas above, calculation results are as follows:

	Euclidean distance	Manhattan distance	Supremum distance	Cosine similarity
x1	.14142136	.2	.1	. 99999139
x2	. 67082039	.9	.6	. 99575226
x3	. 28284271	.4	.2	. 99996948
x4	. 2236068	.3	.2	. 99902823
x5	. 60827625	.7	.6	. 96536339

So the rank is:

Euclidean distance	Manhattan distance	Supremum distance	Cosine similarity

x2	x2	x2	x1
x5	x5	x5	x3
x3	x3	x3	x4
x4	x4	x4	x2
x1	x1	x1	x5

Part b

Codes: 2.py

The problem described a L2-norm process. After applying L2-norm to the data and calculate the Euclidean distance, the result is as follows:

	Normalized vector	Euclidean distance
X	[0.65850461 0.75257669]	-
X1	[0.66162164 0.74983786]	[0.00414935]
X2	[0.72499943 0.68874946]	[0.09217091]
X3	[0.66436384 0.74740932]	[0.00781232]
X4	[0.62469505 0.78086881]	[0.04408549]
X5	[0.83205029 0.5547002]	[0.26319805]

Then sort the data points according to the Euclidean distance, the result is:

X5
X2
X4
X3
X1

Problem 3.1

Accuracy: The intended use of data affects the assessment of data accuracy. For example, the accuracy of weights. Consider two situations, one you want to mine the relationship between body weights and diabetes rates, the other one you want to explore how the weight proportion of carbon affects the strength of steel. Above or below 1% of the accrual weight can be considered accurate in the first situation, but apparently not acceptable in the latter one.

Completeness: If the missing data is not the key attribute to the determination process, it may be a difference in consideration of its completeness. For example, when you explore the relationship between IQ and GPA, you may have incomplete data of student age. It is not what you intend to consider, but it can be added as an affecter. Under this circumstance, the incomplete of the data is not a big problem.

Consistency: when doing a time series analysis like macroeconomics trend prediction, consistency of the data is extremely important. But if you are dealing with cross-section data, the consistency is not as crucial as the first one.

There are other dimensions of data quality, one is timeliness. Some types of data are only useful for decision making in a particular frame of time. For example, satellite clouds pictures.
And, believability also affects data quality. Data from unauthorized sources or not collected in a proper way may not be considered trustworthy.

Problem 3.3

Code: 3.py

Part a

Step 1: Separate sorted values into bins with bin depth 2. Results are as follows:
bin1: [13, 15, 16]
bin2: [16, 19, 20]
bin3: [20, 21, 22]
bin4: [22, 25, 25]
bin5: [25, 25, 30]
bin6: [33, 33, 35]
bin7: [35, 35, 35]
bin8: [36, 40, 45]
bin9: [46, 52, 70]
Step 2: Use mean, median, or boundaries to represent the data in each bin. The results are:
Smoothing by bin means:
[[14.0, 14.0, 14.0], [17.0, 17.0, 17.0], [20.333333333333332, 20.333333333333332, 20.333333333333332], [23.0, 23.0, 23.0], [25.0, 25.0, 25.0], [32.0, 32.0, 32.0], [35.0, 35.0, 35.0], [37.0, 37.0, 37.0], [47.666666666666664, 47.666666666666664, 47.666666666666664]]
Smoothing by bin medians:
[[14.0, 14.0, 14.0], [16.0, 16.0, 16.0], [20.0, 20.0, 20.0], [22.0, 22.0, 22.0], [25.0, 25.0, 25.0], [33.0, 33.0, 33.0], [35.0, 35.0, 35.0], [36.0, 36.0, 36.0], [46.0, 46.0, 46.0]]
Smoothing by bin boundaries:
[[13, 13, 16], [16, 16, 20], [20, 20, 22], [22, 22, 25], [25, 25, 30], [33, 33, 35], [35, 35, 35], [36, 36, 45], [46, 46, 70]]
After smoothing, the data became centered within each bin.

Part b

By looking at difference of the first and last bins. If they are too far away from other bins, there might be outliners.

Part c

Concept hierarchies. A concept hierarchy defines a sequence of mappings from a set of low-level concepts to higher-level, more general concepts, which smooths the data differences in the low-level dimentions.

Problem 3.5

(a) min-max normalization: [new_min, new_max]
(b) z-score normalization: [(old_min – mean)/std, (old_max – mean)/std]
(c) z-score normalization using the mean absolute deviation instead of standard deviation: [(old_min – mean)/mad, (old_max – mean)/ mad]
(d) normalization by decimal scaling: (–1.0, 1.0)

Problem 3.7

Code: 3.py
(a) age_minmax = (age-min(age))/(max(age)-min(age))
35 is nomadized to 0.38596491
(b) age_zscore = (age - mean(age)) / 19.24
35 is nomadized to 0.2618002618002619
(c) j = round(np.log10(np.max(age)))
age_decimal = age / 10^j
35 is nomadized to 0.35
(d) I would choose z_score normalization over the other two methods. Because the dataset we have is small, which means it is probably biased, and given the standard deviation, method two normalized the data more properly as the only one considered whole data distribution.

Problem 3.11

(a)

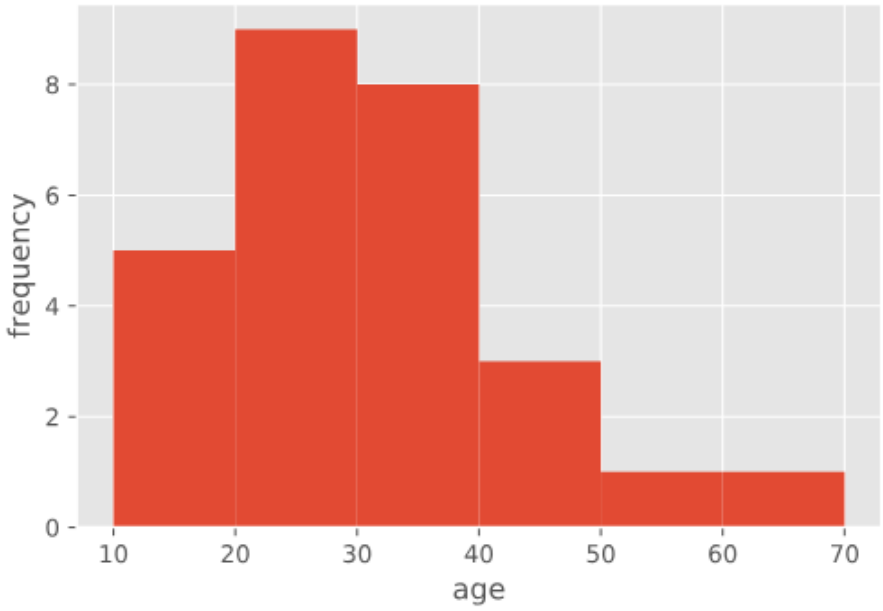

Figure 1 Histogram of width 10

(b) Simple random sample without replacement (SRSWOR):
15(T2), 22(T9), 35(T17), 35(T17), 45(T23)

Simple random sample with replacement (SRSWR):

15(T2), 22(T9), 35(T17), 35(T18), 45(T23)

Cluster Sampling:
Step 1 clustered in two groups:
Cluster #1: 13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25, 25, 25, 30, 33, 33
Cluster #2: 35, 35, 35, 35, 36, 40, 45, 46, 52, 70
Step 2 sample from each cluster
From cluster1: 15(T2), 22(T9), 33(T16)
From cluster 2:35(T18), 45(T23)

Stratified Sampling:
Step 1 separating dataset according to the strata:
Youth: 13, 15, 16, 16, 19, 20, 20, 21, 22, 22, 25, 25, 25, 25, 30, 33, 33
Middle-aged: 35, 35, 35, 35, 36, 40, 45, 46
Senior: 70
Step 2: sample from different strata
Youth: 20(T6), 25(T14)
Middle-aged: 35(T20, 40(T24)
Senior: 70(T27)

Problem 3.13

Code: 3.py
Pseudocode as follows:
(a) Function conceptHierarchyDistinctValues(Attributes set A):
Init result set R;
for each attribute A_i in attributes set A:
count distinct value d_i;
sort A according to d_i;
from the smallest A_i to the largest in sorted A:
generate hierarchy from (A_i), add into R;
Exam the generated hierarchy;
Return R;

(b) Function conceptHierarchyEqualWidth(Attribute, Width):
Result set R = [];
Init fisrt bin b;
Sort attribute values;
for each value i in sorted attribute value set:
if i – b[0]>= width:
append b to R;
Set b to initial state;
Else:
Append i to b;
return R;

(c) Function conceptHierarchyEqualFrequency(attribute, frequency):

```
Init result set R, bin b to empty;
Set count = 0;
Sort attribute = sort;
for each value i in sort:
        if count == frequency:
                append b to R;
                set b to empty;
                set count to 0;
        else:
                append i to b;
Return R;
```