

Exercise 1

We can define the *edit distance* $d_{\text{edit}}(w, w') : \Sigma^2 \rightarrow \mathbb{R}$ as follows. (Let $w = w_1 \dots w_n$ and $w' = w'_1 \dots w'_m$)

$$d_{\text{edit}}(w, w') \mapsto \begin{cases} |w| & \text{if } |w'| = 0 \\ |w'| & \text{if } |w| = 0 \\ d_{\text{edit}}(w_2 \dots w_n, w'_2 \dots w'_m) & \text{if } w_1 = w'_1 \\ 1 + \min \begin{cases} d_{\text{edit}}(w_2 \dots w_n, w') \\ d_{\text{edit}}(w, w'_2 \dots w'_m) \\ d_{\text{edit}}(w_2 \dots w_n, w'_2 \dots w'_m) \end{cases} & \text{otherwise} \end{cases}$$

As this definition of d_{edit} works by removing at most the first character of each word, we can proof by induction the length of $x, y, z \in \Sigma$, that d_{edit} is a metric on Σ :

- Let $|x| = |y| = |z| = 0$. Therefore, also $x = y = z$.
Then $0 \leq d_{\text{edit}} = 0$. Thus, Nonnegativity is given.
Since $x = y$, also $d_{\text{edit}}(x, y) = d_{\text{edit}}(y, x)$. Thus, Symmetry is given.
Since $x = y = z$, the Triangle Inequality $d_{\text{edit}}(x, z) \leq d_{\text{edit}}(x, y) + d_{\text{edit}}(y, z) \Leftrightarrow 0 \leq 0 + 0$ is given.
- Let $x = x_1 \dots x_n$, $y = y_1 \dots y_m$, and $z = z_1 \dots z_o$, $n, m, o \geq 1$. For $x' = x_2 \dots x_n$, $y' = y_2 \dots y_m$, and $z' = z_2 \dots z_o$ Nonnegativity, Symmetry, and the Triangle Inequality of d_{edit} is given.
- Since $n, m \geq 1$, the second rule of Nonnegativity, namely $d_{\text{edit}}(x, y) \Leftrightarrow x = y$ does not apply here.
Since all $d_{\text{edit}}(x', y')$, $d_{\text{edit}}(x', y)$, $d_{\text{edit}}(x, y')$ are non-negative, by definition of d_{edit} , $d_{\text{edit}}(x, y)$ must be non-negative as well. Therefore, the Nonnegativity of d_{edit} is proven.
- If $x_1 = y_1$, then $d_{\text{edit}}(x, y) = d_{\text{edit}}(x', y') = d_{\text{edit}}(y', x') = d_{\text{edit}}(y, x)$
If $x_1 \neq y_1$, then
- Triangle Inequality

Exercise 2

Result (see Appendix for code):

Classification: k=2 Manhattan Distance

Test (1, -2, 0): Prediction 1

Test (4, -0.5, 2): Prediction -1

Test (1, 1.5, -2.5): Prediction 0

Test (-2, -1, -2): Prediction 0

Test (-4, -1, -1): Prediction 0

Classification: k=3 Manhattan Distance

Test (1, -2, 0): Prediction 1

Test (4, -0.5, 2): Prediction -1

Test (1, 1.5, -2.5): Prediction 1

Test (-2, -1, -2): Prediction -1

Test (-4, -1, -1): Prediction 1

Classification: k=2 Euclidean Distance

Test (1, -2, 0): Prediction 1

Test (4, -0.5, 2): Prediction -1

Test (1, 1.5, -2.5): Prediction 1

Test (-2, -1, -2): Prediction 0

Test (-4, -1, -1): Prediction 0

Classification: k=3 Euclidean Distance

Test (1, -2, 0): Prediction 1

Test (4, -0.5, 2): Prediction -1

Test (1, 1.5, -2.5): Prediction 1

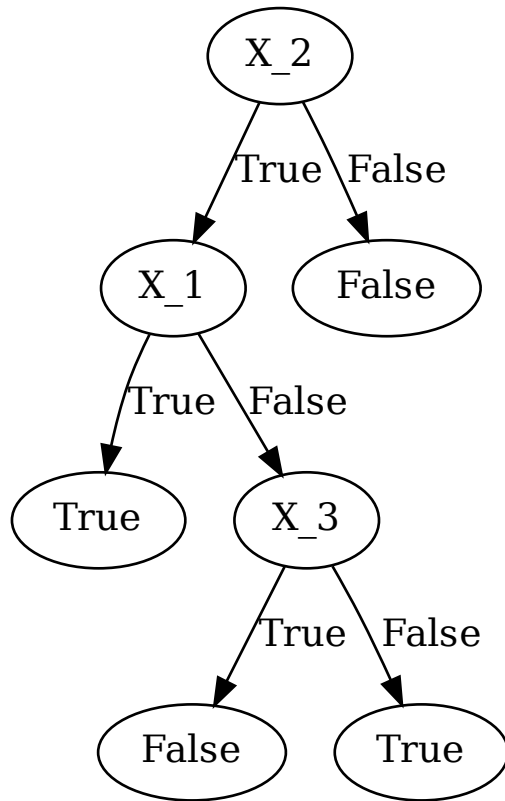
Test (-2, -1, -2): Prediction 1

Test (-4, -1, -1): Prediction -1

Exercise 3

- (a) $X_1 \wedge X_2 \wedge X_3$ is obviously in 1-CNF, thus also in 2-CNF. There exists no satisfiability equivalent formula in 2-DNF. For this, there would have to be at least two disjunctions of conjunctions of at most 2 literals. Thus, such a formula would be true even if one literal would be false.
- (b) $X_1 \vee X_2 \vee X_3$ is obviously in 1-DNF, thus also in 2-DNF. There exists no satisfiability equivalent formula in 2-CNF. Since such a formula can only have 2 literals in each disjunction, there is no possibility of validating, that the third literal might be true. Therefore, such a formula is not true, when just one literal is true, the others false.
- (c)

Exercise 4



Reasoning (see Appendix for code):

Feature Set: $[X_1, X_2, X_3]$

Gains for each feature $[(1, 0.5487949406953987), (0, 0.04879494069539858), (2, 0.04879494069539858)]$

Splitting using feature X_2

Feature Set: $[X_1, X_3]$

Gains for each feature $[(0, 0.31127812445913283), (2, 0.31127812445913283)]$

Splitting using feature X_1

Feature Set: $[X_3]$

Gains for each feature $[(2, 1.0)]$

Splitting using feature X_3

Exercise 5

(a) $x'_0 = [0, 0, 0, 1, 0]^\top$, since $\langle a, x \rangle = \frac{1}{3} \cdot (3 \cdot 0 + 1 \cdot 0 + 2 \cdot 0 + -2 \cdot 1 + 3 \cdot 0) = -\frac{2}{3}$

(b) $\langle a, x_1 \rangle + \frac{2}{3} = \frac{1}{3} \cdot (3 + 1 + 2 - 2 + 3) + \frac{2}{3} = \frac{7+2}{3} = 3 \geq 0$
 $\langle a, x'_1 \rangle + \frac{2}{3} = \frac{1}{3} \cdot (3 - 1 - 2 + 2 - 3) + \frac{2}{3} = \frac{-1+2}{3} = \frac{1}{3} \geq 0$

Therefore, x_1, x'_1 are contained in the same halfspace.

- (c) Let $X \in P$ be a point on the hyperplane. Then the length of the projection of $(x_1 - X)$ is $\frac{|\langle a, x_0 - X \rangle|}{\|a\|} = \frac{|\langle a, x_0 \rangle - \langle a, X \rangle|}{\|a\|} = \frac{\langle a, x_1 \rangle + b}{\|a\|} = \frac{3}{26 + \frac{1}{9}} = \frac{27}{235} \simeq 0.11489$

Exercise 6

Since we are in \mathbb{R}^3 we can divide the grape by all 3 dimensions twice, thus splitting the grape with 3 strikes into $2^3 = 8$ pieces. The last strike cannot split the grape by another dimension. We can only split at most 7 pieces in half, thus a maximum of 15 pieces. The swordmasters claims are invalid.

Appendix

Code for Exercise 2

```

1 from math import sqrt
2 #from statistics import mode
3
4 training_set = [
5     ((-4, -2.1, -1), -1),
6     ((-3.6, -1.4, 0.2), 1),
7     ((1, -0.2, -0.3), 1),
8     ((0.3, -0.5, -0.5), 1),
9     ((-2, -3.5, -1), -1),
10    ((-4.2, -4, 0.2), 1),
11    ((-1.3, -0.1, -3), 1),
12    ((-0.7, 0.9, -0.7), 1),
13    ((1, 2, 1.4), 1),
14    ((2.6, -1.5, 0.2), 1),
15    ((2, 4.3, -0.7), -1),
16    ((0.6, 0.4, 0.2), -1),
17    ((2.9, -1.7, 3.6), -1),
18    ((3.6, 0.4, -2.5), -1),
19    ((1.2, 4, 1.2), -1),
20    ((-1, 0.5, 0.5), -1),
21    ((3, 2.7, 2.3), -1),
22    ((4, -3, 2.2), -1),
23    ((0.1, 0.1, 3.5), -1),
24    ((2.8, 1.2, 2.4), -1)
25 ]
26
27 test_set = [
28     (1, -2, 0),
29     (4, -0.5, 2),
30     (1, 1.5, -2.5),
31     (-2, -1, -2),
32     (-4, -1, -1)
33 ]
34
35
36 def manhattan_distance(x, y):
37     assert len(x) == len(y)
38     sum = 0
39     for i in range(len(x)):
40         sum += abs(x[i] - y[i])
41     return sum
42

```

```
43
44 def euclidean_distance(x, y):
45     assert len(x) == len(y)
46     sum = 0
47     for i in range(len(x)):
48         sum += pow(x[i] - y[i], 2)
49     sum = sqrt(sum)
50     return sum
51
52
53 def get_k_nearest_neighbors(training_set, test, k, distance_func):
54     assert callable(distance_func)
55     distances = list()
56     for data in training_set:
57         distance = distance_func(data[0], test)
58         distances.append((data, distance))
59     distances.sort(key=lambda x: x[1]) # sort by distance
60     neighbors = list()
61     for i in range(k): # get data of k nearest neighbors
62         neighbors.append(distances[i][0])
63     return neighbors
64
65
66 def predict_classification(training_set, test, k, distance_func):
67     assert callable(distance_func)
68     neighbors = get_k_nearest_neighbors(training_set, test, k, distance_func)
69     classifications = [neighbor[1] for neighbor in neighbors] # list of all
70     # predictions
71     # prediction = mode(classifications) # get classification most often in k
72     # nearest neighbors
73     # return prediction
74     count_neg = classifications.count(-1)
75     count_pos = classifications.count(1)
76     assert count_neg + count_pos == k
77     if count_pos > count_neg:
78         return 1
79     if count_pos < count_neg:
80         return -1
81     if count_pos == count_neg:
82         return 0
83
84 if __name__ == '__main__':
85     print('Classification: k=2 Manhattan Distance')
86     for test in test_set:
87         prediction = predict_classification(
88             training_set, test, 2, manhattan_distance)
89         print(f'Test {test}: Prediction {prediction}')
90     print('\n')
91     print('Classification: k=3 Manhattan Distance')
92     for test in test_set:
93         prediction = predict_classification(
94             training_set, test, 3, manhattan_distance)
95         print(f'Test {test}: Prediction {prediction}')
96     print('\n')
97     print('Classification: k=2 Euclidean Distance')
98     for test in test_set:
99         prediction = predict_classification(
100             training_set, test, 2, euclidean_distance)
```

```
100     print(f'Test {test}: Prediction {prediction}')
101     print('\n')
102     print('Classification: k=3 Euclidean Distance')
103     for test in test_set:
104         prediction = predict_classification(
105             training_set, test, 3, euclidean_distance)
106         print(f'Test {test}: Prediction {prediction}')
```

Code for Exercise 4

```
1 from math import log2
2 from graphviz import Digraph
3 import queue
4
5 feature_set = [0, 1, 2]
6
7 example_set = [
8     ((False, False, False), False),
9     ((False, False, True), False),
10    ((False, True, False), True),
11    ((False, True, True), False),
12    ((True, False, False), False),
13    ((True, False, True), False),
14    ((True, True, False), True),
15    ((True, True, True), True),
16 ]
17
18
19 class Node:
20     def __init__(self, value):
21         self.left = None
22         self.left_label = True
23         self.value = value
24         self.right = None
25         self.right_label = False
26
27
28 def count_pos(example_set) -> int:
29     classifications = [example[1] for example in example_set]
30     return classifications.count(True)
31
32
33 def count_neg(example_set) -> int:
34     classifications = [example[1] for example in example_set]
35     return classifications.count(False)
36
37
38 def partition_example_set(feature, example_set) -> tuple:
39     list_true = list()
40     list_false = list()
41     for example in example_set:
42         if example[0][feature]:
43             list_true.append(example)
44         else:
45             list_false.append(example)
46     return (list_true, list_false)
47
48
49 def entropy(example_set) -> float:
50     pos = count_pos(example_set)
51     neg = count_neg(example_set)
```

```
52     assert pos + neg == len(example_set)
53     pos_frac = pos / float(pos + neg)
54     neg_frac = 1 - pos_frac
55     if pos_frac == 0:
56         return -neg_frac * log2(neg_frac)
57     if neg_frac == 0:
58         return -pos_frac * log2(pos_frac)
59     result = -(pos_frac * log2(pos_frac) + neg_frac * log2(neg_frac))
60     assert 0 <= result and result <= 1
61     return result
62
63
64 def remainder(feature, example_set) -> float:
65     (list_true, list_false) = partition_example_set(feature, example_set)
66     true_frac = len(list_true) / float(len(example_set))
67     false_frac = 1 - true_frac
68     result = true_frac * entropy(list_true) + false_frac * entropy(
69         list_false)
70     return result
71
72 def gain(feature, example_set) -> float:
73     result = entropy(example_set) - remainder(feature, example_set)
74     return result
75
76
77 def decision_tree(feature_set, example_set) -> Node:
78     if len(example_set) == 0:
79         return Node(False) # arbitrary value
80
81     if count_pos(example_set) == len(example_set):
82         return Node(True)
83     if count_neg(example_set) == len(example_set):
84         return Node(False)
85
86     gains = [(feature, gain(feature, example_set)) for feature in
87         feature_set]
88     gains.sort(key=lambda x: x[1], reverse=True)
89     max_gain = gains[0]
90     partition_feature = max_gain[0]
91     (true_frac, false_frac) = partition_example_set(
92         partition_feature, example_set)
93     new_feature_set = list(feature_set)
94     new_feature_set.remove(partition_feature)
95
96     print(f'Feature Set: {list(map(lambda feature: f"X_{feature+1}",
97         feature_set))}')
98     print(f'Gains for each feature {gains}')
99     print(f'Splitting using feature X_{partition_feature + 1}')
100
101     node = Node(f'X_{partition_feature + 1}')
102     node.left = decision_tree(new_feature_set, true_frac)
103     node.left_label = True
104     node.right = decision_tree(new_feature_set, false_frac)
105     node.right_label = False
106
107     return node
108
109 def draw_decision_tree(tree):
```

```
109 dot = Digraph('Decision-Tree')
110 q = queue.Queue()
111 q.put(('root', tree))
112 while not q.empty():
113     (node_name, node) = q.get()
114     dot.node(name=node_name, label=str(node.value))
115     if node.left:
116         child_name = node_name+str(node.value) + \
117             str(node.left_label)+str(node.left.value)
118         dot.node(name=child_name, label=str(node.left.value))
119         dot.edge(node_name, child_name, label=str(node.left_label))
120         q.put((child_name, node.left))
121
122     if node.right:
123         child_name = node_name + \
124             str(node.value)+str(node.right_label)+str(node.right.value)
125         dot.node(name=child_name, label=str(node.right.value))
126         dot.edge(node_name, child_name, label=str(node.right_label))
127         q.put((child_name, node.right))
128 dot.render()
129
130
131 if __name__ == '__main__':
132     tree = decision_tree(feature_set, example_set)
133     draw_decision_tree(tree)
```