Tanhim Islam
Simon Michau
Til Mohr

Exercise 1

We can define the *edit distance* $d_{\text{edit}}(w, w') : \Sigma^2 \to \mathbb{R}$ 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

Tanhim Islam
Simon Michau
Til Mohr

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

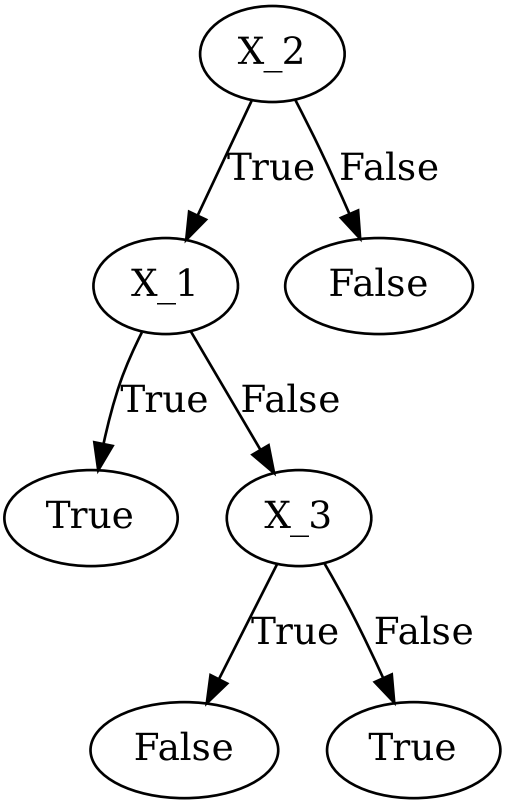
Exercise 3

(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

Exercise 6

Appendix

Code for Exercise 2

```python
from math import sqrt
#from statistics import mode

training_set = [
    ((-4, -2.1, -1), -1),
    ((-3.6, -1.4, 0.2), 1),
    ((1, -0.2, -0.3), 1),
    ((0.3, -0.5, -0.5), 1),
    ((-2, -3.5, -1), -1),
    ((-4.2, -4, 0.2), 1),
    ((-1.3, -0.1, -3), 1),
    ((-0.7, 0.9, -0.7), 1),
    ((1, 2, 1.4), 1),
    ((2.6, -1.5, 0.2), 1),
    ((2, 4.3, -0.7), -1),
    ((0.6, 0.4, 0.2), -1),
    ((2.9, -1.7, 3.6), -1),
    ((3.6, 0.4, -2.5), -1),
    ((1.2, 4, 1.2), -1),
    ((-1, 0.5, 0.5), -1),
    ((3, 2.7, 2.3), -1),
    ((4, -3, 2.2), -1),
    ((0.1, 0.1, 3.5), -1),
    ((2.8, 1.2, 2.4), -1)
]

test_set = [
    (1, -2, 0),
    (4, -0.5, 2),
    (1, 1.5, -2.5),
    (-2, -1, -2),
    (-4, -1, -1)
]


def manhattan_distance(x, y):
    assert len(x) == len(y)
    sum = 0
    for i in range(len(x)):
        sum += abs(x[i] - y[i])
    return sum


def euclidean_distance(x, y):
    assert len(x) == len(y)
    sum = 0
    for i in range(len(x)):
        sum += pow(x[i] - y[i], 2)
    sum = sqrt(sum)
    return sum
```

```python


def get_k_nearest_neighbors(training_set, test, k, distance_func):
    assert callable(distance_func)
    distances = list()
    for data in training_set:
        distance = distance_func(data[0], test)
        distances.append((data, distance))
    distances.sort(key=lambda x: x[1]) # sort by distance
    neighbors = list()
    for i in range(k): # get data of k nearest neighbors
        neighbors.append(distances[i][0])
    return neighbors


def predict_classificataion(training_set, test, k, distance_func):
    assert callable(distance_func)
    neighbors = get_k_nearest_neighbors(training_set, test, k, distance_func)
    classifications = [neighbor[1] for neighbor in neighbors] # list of all classifications
    #prediction = mode(classifications) # get classification most often in k nearest neighbors
    #return prediction
    count_neg = classifications.count(-1)
    count_pos = classifications.count(1)
    assert count_neg + count_pos == k
    if count_pos > count_neg:
        return 1
    if count_pos < count_neg:
        return -1
    if count_pos == count_neg:
        return 0


if __name__ == '__main__':
    print('Classification: k=2 Manhattan Distance')
    for test in test_set:
        prediction = predict_classificataion(
            training_set, test, 2, manhattan_distance)
        print(f'Test {test}: Prediction {prediction}')
    print('\n')
    print('Classification: k=3 Manhattan Distance')
    for test in test_set:
        prediction = predict_classificataion(
            training_set, test, 3, manhattan_distance)
        print(f'Test {test}: Prediction {prediction}')
    print('\n')
    print('Classification: k=2 Euclidean Distance')
    for test in test_set:
        prediction = predict_classificataion(
            training_set, test, 2, euclidean_distance)
        print(f'Test {test}: Prediction {prediction}')
    print('\n')
    print('Classification: k=3 Euclidean Distance')
    for test in test_set:
        prediction = predict_classificataion(
            training_set, test, 3, euclidean_distance)
        print(f'Test {test}: Prediction {prediction}')
```

Code for Exercise 4

```python
from math import log2
from graphviz import Digraph
import queue

feature_set = [0, 1, 2]

example_set = [
    ((False, False, False), False),
    ((False, False, True), False),
    ((False, True, False), True),
    ((False, True, True), False),
    ((True, False, False), False),
    ((True, False, True), False),
    ((True, True, False), True),
    ((True, True, True), True),
]


class Node:
    def __init__(self, value):
        self.left = None
        self.left_label = True
        self.value = value
        self.right = None
        self.right_label = False


def count_pos(example_set) -> int:
    classifications = [example[1] for example in example_set]
    return classifications.count(True)


def count_neg(example_set) -> int:
    classifications = [example[1] for example in example_set]
    return classifications.count(False)


def partition_example_set(feature, example_set) -> tuple:
    list_true = list()
    list_false = list()
    for example in example_set:
        if example[0][feature]:
            list_true.append(example)
        else:
            list_false.append(example)
    return (list_true, list_false)


def entropy(example_set) -> float:
    pos = count_pos(example_set)
    neg = count_neg(example_set)
    assert pos + neg == len(example_set)
    pos_frac = pos / float(pos + neg)
    neg_frac = 1 - pos_frac
    if pos_frac == 0:
        return -neg_frac * log2(neg_frac)
    if neg_frac == 0:
        return -pos_frac * log2(pos_frac)
```

```
    result = -(pos_frac * log2(pos_frac) + neg_frac * log2(neg_frac))
    assert 0 <= result and result <= 1
    return result


def remainder(feature, example_set) -> float:
    (list_true, list_false) = partition_example_set(feature, example_set)
    true_frac = len(list_true) / float(len(example_set))
    false_frac = 1 - true_frac
    result = true_frac * entropy(list_true) + false_frac * entropy(list_false)
    return result


def gain(feature, example_set) -> float:
    result = entropy(example_set) - remainder(feature, example_set)
    return result


def decision_tree(feature_set, example_set) -> Node:
    if len(example_set) == 0:
        return Node(False)  # arbitrary value

    if count_pos(example_set) == len(example_set):
        return Node(True)
    if count_neg(example_set) == len(example_set):
        return Node(False)

    gains = [(feature, gain(feature, example_set)) for feature in feature_set]
    gains.sort(key=lambda x: x[1], reverse=True)
    max_gain = gains[0]
    partition_feature = max_gain[0]
    (true_frac, false_frac) = partition_example_set(
        partition_feature, example_set)
    new_feature_set = list(feature_set)
    new_feature_set.remove(partition_feature)

    print(f'Feature Set: {list(map(lambda feature: f"X_{feature+1}", feature_set))}')
    print(f'Gains for each feature {gains}')
    print(f'Splitting using feature X_{partition_feature + 1}')

    node = Node(f'X_{partition_feature + 1}')
    node.left = decision_tree(new_feature_set, true_frac)
    node.left_label = True
    node.right = decision_tree(new_feature_set, false_frac)
    node.right_label = False

    return node


def draw_decision_tree(tree):
    dot = Digraph('Decision-Tree')
    q = queue.Queue()
    q.put(('root', tree))
    while not q.empty():
        (node_name, node) = q.get()
        dot.node(name=node_name, label=str(node.value))
        if node.left:
```

```
            child_name = node_name+str(node.value) + \
                str(node.left_label)+str(node.left.value)
            dot.node(name=child_name, label=str(node.left.value))
            dot.edge(node_name, child_name, label=str(node.left_label))
            q.put((child_name, node.left))

        if node.right:
            child_name = node_name + \
                str(node.value)+str(node.right_label)+str(node.right.value)
            dot.node(name=child_name, label=str(node.right.value))
            dot.edge(node_name, child_name, label=str(node.right_label))
            q.put((child_name, node.right))
    dot.render()


if __name__ == '__main__':
    tree = decision_tree(feature_set, example_set)
    draw_decision_tree(tree)
```