KRR project 2

Daniel Sociu
Group:407

1 Exercise 1

For the first exercise I decided to use the following rules to represent the knowledge:

- If the student had dedication and is prepared then he will pass the exam.
- If the student studied more than 20 hours and understands the theory he is prepared for the exam.
- If the student participated on more than 5 courses then he understands the theory.
- If the student wakes up early everyday (before 8:00) then he is an early riser.
- If the student is an early riser and had dedication for the exam then he ambitious.

Given the rules rules we can build our KB which I have wrote in a .txt file (which is a Horn KB), which looks like this:

[¬dedication,¬prepared,pass]
[¬studied,¬understands,prepared]
[¬participation,understands]
[¬wakesup,early]
[¬early,¬dedication,ambitious]

Therefore the corresponding questions that the user will have to answer are:

- How many hours did the student study? (number of hours)
- How many courses did the student participate to? (number of courses)
- How early does the student wake up everyday? (xx:xx)
- Was the student dedicated for this exam? (yes/no)

Given the answers to the questions, we use those answers to build our atoms, which will be later on be passed to the two algorithms of backward chaining and forward chaining.

Therefore these algorithms will get as input the KB, the atoms and the question, and as the output both will print whether the question if logically entailed on not.

2 Exercise 2

For the second exercise I decided to implement a heat recommender system, which means that given the desired inside temperature and the outside temperature, it will predict from a scale from 0 to 10 how much the heating should the system be set to. I decided to use 0 to 10 since it can depend on the type of machine that sets the heat but it can be scaled easily from a 0-10 range, therefore I considered that the heating system has an intensity setting rather than a desired temperature (this could be the algorithm used inside the machine when the user sets the desired temperature).

The outside temperature can range between -20 and 40 degrees (scaled to 0-60) and the desired inside temperature ranges between 10-30 degrees (again scaled to 0-20). The mentioned scaling is only done to ease the implementation, the user will get the proper output.

These are the following rules that I considered to implement in my algorithm:

1. If the temperature outside is hot or the desired temperature is low then the heating is low
2. If the temperature outside is average and the desired temperature is medium then the heating is medium
3. If the temperature outside is cold and desired temperature is high then the heating is high

I decided to split the data in a simple manner of choosing the center of each case (this can be easily customized in code), therefore the curves can be observed in figures 1 for outside, 2 for desired, and finally heating: 3

As for the defuzzification algorithm I simply used the discrete values of the aggregated degree curve of the heating. I used an algorithm that would give me the centroid of the curve.

An example of how the aggregated values for the heating algorithm with a specific input can be seen in the figure 4. This specific case with input -15 and 24 gave an output of 6.56.

Therefore the user can input any values between the specified interval and get a recommendation of how to set the heater (which level to set it to), algorithm which was implemented using the reasoning procedure presented in course 8.

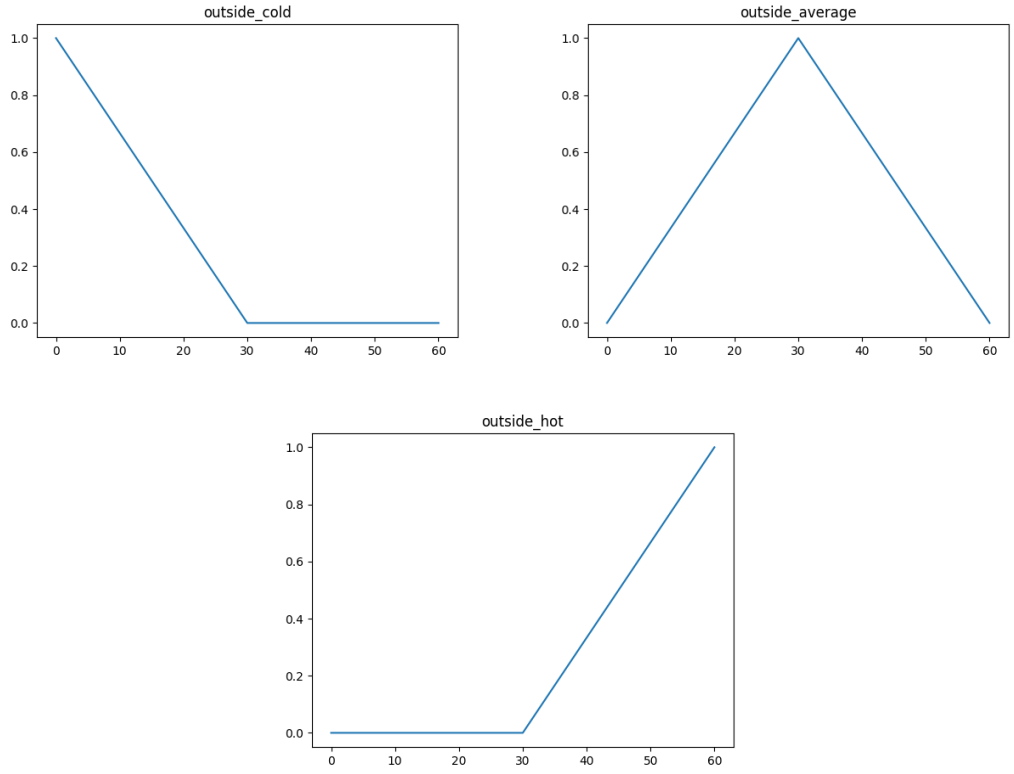

Figure 1: Outside temperature curves for each case

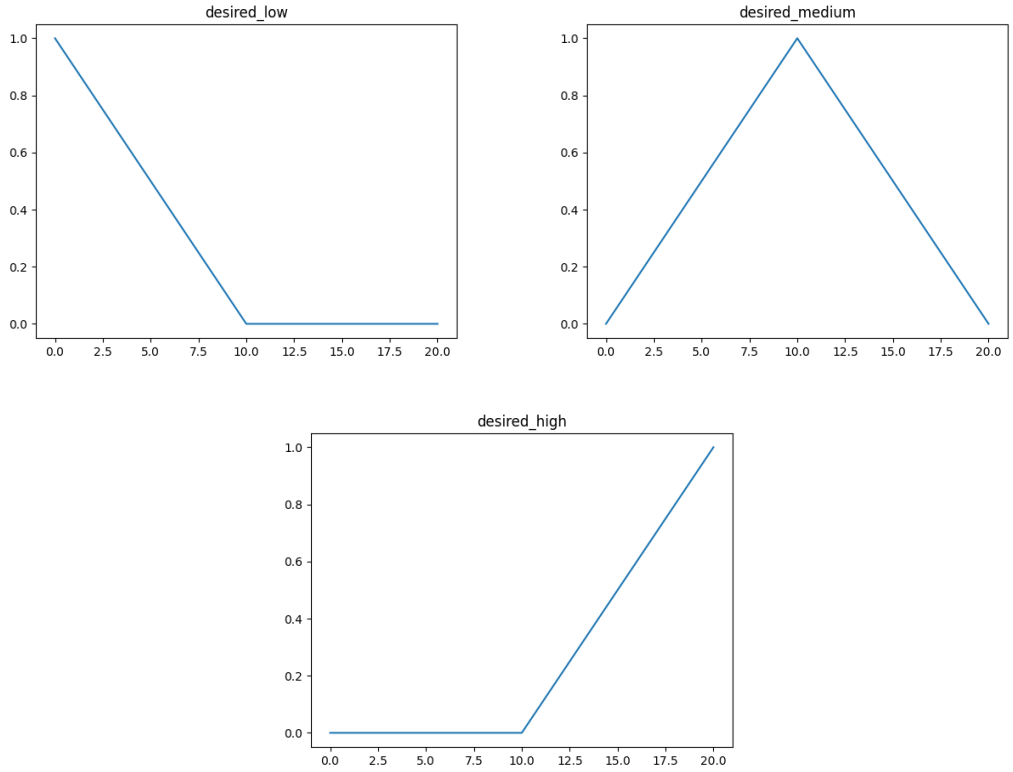

Figure 2: Desired temperature curves for each case

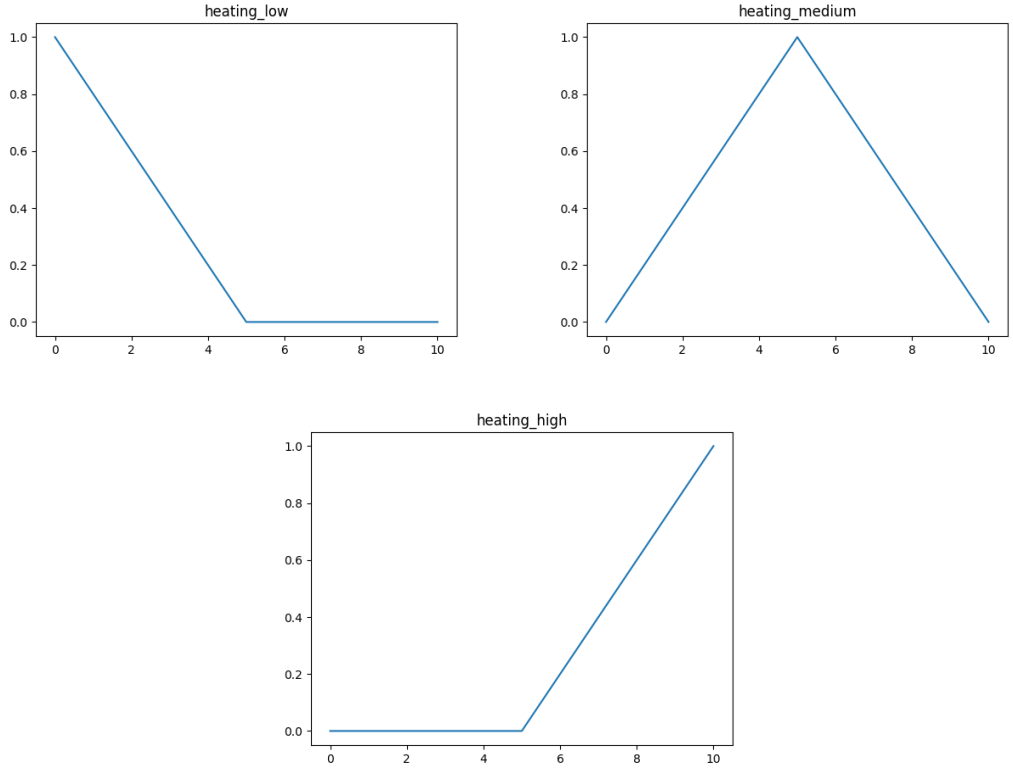

Figure 3: Heating level curves for each case

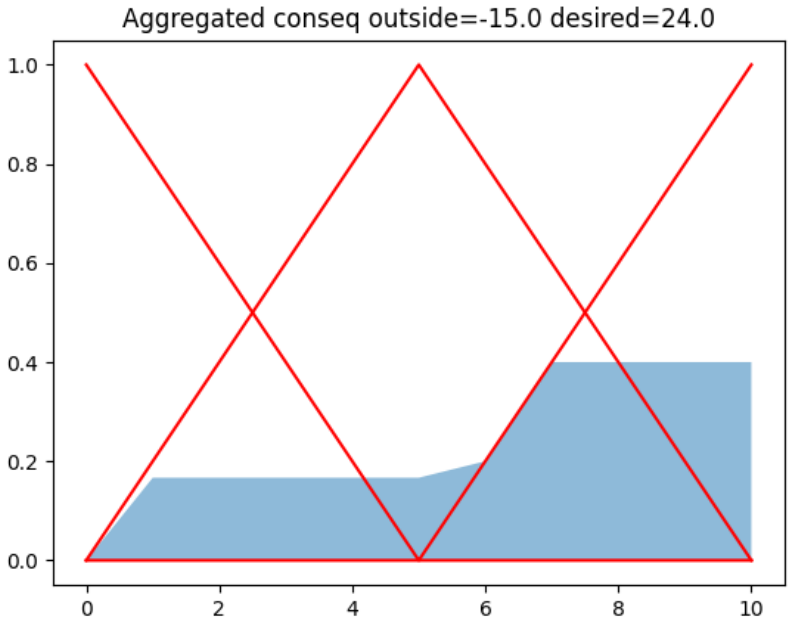

Figure 4: Aggregated output

3 Code

3.1 Exercise 1

```
from copy import deepcopy

FILE_NAME = 'ex1_input.txt'
DEBUG = True


def line_parser(line):
    clauses = []
    line = line[1:-1]
    length = len(line)
    index = 0
    cur_str = ''
    while index < length:
        if line[index] == '[':
            clauses.append([])
        elif line[index] == ']' or line[index] == ',':
            if cur_str != '':
                clauses[-1].append(cur_str)
            cur_str = ''
        else:
            cur_str = cur_str + line[index]
        index += 1

    return clauses


def negation(atom):
```

```
    if '¬' in atom:
        return atom[1:]
    else:
        return '¬' + atom


def is_negation(atom):
    return '¬' in atom


def check_inclusion(list1, list2):
    return all([item in list2 for item in list1])


def resolve(clause1, clause2, p):
    if p in clause1 and negation(p) in clause1:
        return None
    if p in clause2 and negation(p) in clause2:
        return None
    if p in clause1 and negation(p) in clause2:
        items = [x for x in clause1 if x != p]
        for x in clause2:
            if negation(p) != x:
                items.append(x)
        return items
    if p in clause2 and negation(p) in clause1:
        items = [x for x in clause2 if x != p]
        for x in clause1:
            if negation(p) != x:
                items.append(x)
        return items
    return None


def find_clauses(clauses, resolver):
    for clause in clauses:
        for x in resolver:
            resolve_attempt = resolve(resolver, clause, x)
            if resolve_attempt is not None and resolve_attempt not in clauses:
                return resolve_attempt
    return None


def resolution_backward(clauses, resolver):
    if resolver == []:
        return False

    new_clause = find_clauses(clauses, resolver)
    if DEBUG:
        print(new_clause)
    if new_clause is None:
        return True
    return resolution_backward(clauses, new_clause)


def find_positive_atom(atoms, kb):
```

```
    for clause in kb:
        found = True
        for x in clause:
            if is_negation(x):
                if [negation(x)] not in atoms:
                    found = False
        if found:
            for x in clause:
                if not is_negation(x) and [x] not in atoms:
                    return [x]


def resolution_forward(atoms, kb, question):
    if question in atoms:
        return False

    new_atom = find_positive_atom(atoms, kb)
    if DEBUG:
        print(new_atom)
    if new_atom is None:
        return True

    return resolution_forward([new_atom, *atoms], kb, question)


def execute_test(kb, atoms, question, algo_type="backward_chain"):
    print("====================")
    print(f'Test: {kb + atoms}')
    print(f'The question is: {question}')
    if algo_type == 'backward_chain':
        status = resolution_backward(
            kb + atoms, [negation(quest) for quest in question])
    else:
        status = resolution_forward(atoms, kb, question)
    print(f'Status: {status}')
    if status == False:
        print('Unsatisfiable')
        print("Therefore the answer to the question is logically entailed")
    else:
        print('Satisfiable')
        print("Therefore the answer to the question is NOT logically entailed")
    print("====================")


questions = {
    'studied': 'How many hours did the student study? (number of hours)',
    'participation': 'How many courses did the student participate to? (number of course
    'wakesup': 'How early does the student wake up everyday? (xx:xx)',
    'dedication': 'Was the student dedicated for this exam? (yes/no)',
}

processing_answers = {
    'studied': lambda x: int(x) >= 20,
    'participation': lambda x: int(x) >= 5,
    'wakesup': lambda x: int(x.split(':')[0]) < 8,
    'dedication': lambda x: (True if x == 'yes' else False),
```

```
}


def get_answer_atoms(answers):
    atoms = []
    for key, answer in answers.items():
        if answer:
            atoms.append([key])
        else:
            atoms.append([negation(key)])
    return atoms


def main():
    horn_kb = []
    with open(FILE_NAME, 'r') as f:
        for line in f.readlines():
            line = line.rstrip()
            horn_kb = line_parser(line)
    print('The horn KB is:')
    for rule in horn_kb:
        print(rule)

    while True:
        print('Do you want to continue (type exit otherwise)')
        response = input()
        if response == 'exit':
            break
        answers = {}
        for key, question in questions.items():
            print(question)
            response = input(key + ":")
            answers[key] = response
        processed_answers = {key: processing_answers[key](
            value) for key, value in answers.items()}
        question_atoms = get_answer_atoms(processed_answers)
        combined_kb = deepcopy(question_atoms + horn_kb)
        if DEBUG:
            print(combined_kb)

        print("Output using backward chain:")
        execute_test(
            deepcopy(horn_kb),
            deepcopy(question_atoms),
            ["pass"],
            algo_type='backward_chain'
        )
        print("Output using forward chain:")
        execute_test(
            deepcopy(horn_kb),
            deepcopy(question_atoms),
            ["pass"],
            algo_type='forward_chain'
        )
```

```
if __name__ == '__main__':
    main()
```

3.2 Exercise 2

```
import numpy as np
import matplotlib.pyplot as plt

FILE_NAME = 'ex2_input.txt'
PLOTTING = True
SHOW = True
DEBUG = False


def real_score(key, value):
    if key == "outside":
        return value - 20
    else:
        return value + 10


def line_parser(line):
    rules = []
    line = line[1:-1]
    length = len(line)
    index = 0
    cur_str = ''
    while index < length:
        if line[index] == '[':
            rules.append([])
        elif line[index] == ']' or line[index] == ',':
            if cur_str != '':
                if len(rules):
                    rules[-1].append(cur_str)
                else:
                    rules.append(cur_str)
            cur_str = ''
        else:
            cur_str = cur_str + line[index]
        index += 1

    return rules


def my_linspace(start, end, elements):
    if elements == 1:
        return np.array([max(start, end)])
    return np.linspace(start, end, elements)


def plot_all(predicate, show=False):
    for key, curve in predicate.items():
        curve.plot_curve(show)


class Antecedent():
```

```
def __init__(self, x, start_low, end_low, start_high, end_high, name='default'):
    self.start_low = start_low
    self.end_low = end_low
    self.start_high = start_high
    self.end_high = end_high
    self.name = name
    self.x = x
    self.curve = np.concatenate([
        np.zeros(start_low),
        my_linspace(0, 1, end_low - start_low + 1),
        np.ones(max(start_high - end_low - 1, 0)),
        my_linspace(1, 0, end_high - start_high + 1),
        np.zeros(max(len(x) - end_high - 1, 0))
    ])
    if (start_high - end_low) == 0:
        self.curve = np.delete(self.curve, [end_low + 1])

def get_curve_value(self, score):
    value = 0
    if int(score) != score:
        prior = int(score)
        next = prior + 1
        v1 = self.curve[prior]
        v2 = self.curve[next]
        if v1 < v2:
            x_dist = abs(prior - score)
        else:
            x_dist = abs(next - score)
        value = abs(v1 - v2) * x_dist + min(v1, v2)
    else:
        value = self.curve[int(score)]

    return value

def calculate_antecedent(self, score):
    value = self.get_curve_value(score)
    antecedent = np.zeros(len(self.curve))
    for i in range(len(self.curve)):
        antecedent[i] = min(self.curve[i], value)

    return np.array(antecedent)

def calculate_consequent(self, value):
    antecedent = np.zeros(len(self.curve))
    for i in range(len(self.curve)):
        antecedent[i] = min(self.curve[i], value)

    return antecedent

def plot_curve(self, show=False):
    plt.plot(self.x, self.curve, label=self.name)
    title = self.name.replace("/", "_")
    plt.title(title)
    plt.savefig('plots/' + title + '.png')
    if show:
        plt.show()
```

```
        plt.close()

    def plot_antecedent(self, score, show=False):
        antecedent = self.calculate_antecedent(score)
        base = np.zeros_like(antecedent)
        plt.fill_between(self.x, base, antecedent)
        plt.plot(self.x, self.curve, label=self.name)
        title = self.name.replace("/", "_") + "antecedent_" + str(score)
        plt.title(title)
        plt.savefig(
            'plots/' +
            title +
            '.png'
        )
        if show:
            plt.show()
        plt.close()


outside = {
    "cold": Antecedent(np.arange(0, 61, 1), 0, 0, 0, 30, 'outside/cold'),
    "average": Antecedent(np.arange(0, 61, 1), 0, 30, 30, 60, 'outside/average'),
    "hot": Antecedent(np.arange(0, 61, 1), 30, 60, 60, 60, 'outside/hot'),
}

desired = {
    "low": Antecedent(np.arange(0, 21, 1), 0, 0, 0, 10, 'desired/low'),
    "medium": Antecedent(np.arange(0, 21, 1), 0, 10, 10, 20, 'desired/medium'),
    "high": Antecedent(np.arange(0, 21, 1), 10, 20, 20, 20, 'desired/high'),
}

heating = {
    "low": Antecedent(np.arange(0, 11, 1), 0, 0, 0, 5, 'heating/low'),
    "medium": Antecedent(np.arange(0, 11, 1), 0, 5, 5, 10, 'heating/medium'),
    "high": Antecedent(np.arange(0, 11, 1), 5, 10, 10, 10, 'heating/high'),
}


def evaluate_antecedents(rule, scores, multiple_antecedents):
    antecedents = {}
    if multiple_antecedents:
        for ant_str in rule[1]:
            pred, pred_type = ant_str.split('/')
            pred_func = eval(pred)
            value = pred_func[pred_type].get_curve_value(scores[pred])
            antecedents[ant_str] = value
        if rule[0] == 'or':
            result = 0
            for key, antecedent in antecedents.items():
                result = max(result, antecedent)
        else:
            result = 1
            for key, antecedent in antecedents.items():
                result = min(result, antecedent)
    else:
        pred, pred_type = rule[0][0].split('/')
```

```
            pred_func = eval(pred)
            value = pred_func[pred_type].get_curve_value(scores[pred])
            antecedents[rule[0][0]] = value
            result = value
    return result, antecedents


def evaluate_rule(rule, scores):
    multiple_antecedents = type(rule[0]) is str

    antecedent_result, antecedents = evaluate_antecedents(
        rule, scores, multiple_antecedents)
    conseq_str, conseq_type = rule[-1][0].split('/')
    conseq_func = eval(conseq_str)
    consequent = conseq_func[conseq_type].calculate_consequent(
        antecedent_result)

    return {rule[-1][0]: consequent}, antecedents


def aggregate_consequents(consequents):
    aggregated_conseq = np.zeros(len(next(iter(consequents.values()))))
    for key, conseq in consequents.items():
        aggregated_conseq = np.fmax(aggregated_conseq, conseq)

    return aggregated_conseq


def plot_aggregated_conseq(aggregated_conseq, predicates, scores, show=False):
    for key, pred in predicates.items():
        plt.plot(pred.x, pred.curve, 'r', label=pred.name)

    zeros = np.zeros_like(pred.x)
    plt.fill_between(pred.x, zeros, aggregated_conseq, alpha=0.5)

    title = 'Aggregated conseq ' + \
        ' '.join(f'{key}={real_score(key, score)}' for key,
                 score in scores.items())
    plt.title(title)
    plt.savefig('plots/' + title.replace(" ", "_") + '.png')
    if show:
        plt.show()
    plt.close()


def defuse(aggregated_conseq, method='centroid'):
    numerator, denominator = 0, 0
    for x, f_x in enumerate(aggregated_conseq):
        numerator += x * f_x
        denominator += f_x
    return numerator / denominator


def main():
    if PLOTTING:
        plot_all(outside)
```

```
        plot_all(desired)
        plot_all(heating)

    rules = []
    with open(FILE_NAME, 'r') as f:
        for line in f.readlines():
            line = line.rstrip()
            rule = line_parser(line)
            rules.append(rule)

    print('rules are:')
    for rule in rules:
        print(rule)
    # service['good'].plot_antecedent(1.5, show=True)
    scores = {}
    while True:
        print('Insert scores or type exit to stop:')
        initial_input = input("Outside temp (-20 to 40):")
        if initial_input == 'exit':
            break
        scores['outside'] = float(initial_input) + 20.
        scores['desired'] = float(input("Desired temp (10 - 30):")) - 10
        consequents = {}
        for rule in rules:
            consequent, antecedents = evaluate_rule(rule, scores)
            consequents |= consequent
            if DEBUG:
                print(antecedents)
        aggregated_consequents = aggregate_consequents(consequents)
        if DEBUG:
            print(aggregated_consequents)
        if PLOTTING:
            plot_aggregated_conseq(
                aggregated_consequents, heating, scores, show=SHOW)
        defused_value = defuse(aggregated_consequents)
        print(f"The output is {defused_value}")


if __name__ == '__main__':
    main()
```