Verisk - computer purchase problem

by Jolanta Śliwa

Problem Statement

Suppose you're trying to help a company determine which computers to purchase.

Data - utilization data by employee:

The company has been able to pull utilization data by employee that classifies users into 3 bins, depending on how much they use their computer in their work:

- Low usage - spends a lot of time in meetings, checking email, doing people management
- Average usage - requires some compute power, with balanced mix of heads down/technical work along with a good amount of meetings/email writing
- High usage - power user, relies heavily on computer performance

```
import pandas as pd
```

```
utilization = pd.read_csv(
    "https://raw.githubusercontent.com/shubhamkalra27/dsep-2020/main/datasets/util_b_emp.c
)
```

```
utilization.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 146 entries, 0 to 145
Data columns (total 2 columns):
 #   Column           Non-Null Count  Dtype 
---  ------           --------------  ----- 
 0   employee_id      146 non-null    int64 
 1   utilization_bin  146 non-null    object
dtypes: int64(1), object(1)
memory usage: 2.4+ KB
```

```
utilization.head()
```

	employee_id	utilization_bin
1	1752	high

Checking types of utilization

```
utilization["utilization_bin"].unique()
```

```
array(['high', 'medium', 'low'], dtype=object)
```

average usage is stored as medium

Data - survey

Additionally, they've surveyed employees to collect the relative importance of the following variables describing a computer's performance:

- Memory
- Processing
- Storage
- Price inverse - this metric was given to you by the company as you can see in the dataset, with the directive that price inverse being fixed at a 25% weight in the purchase decision

```
survey = pd.read_csv(
    "https://raw.githubusercontent.com/shubhamkalra27/dsep-2020/main/datasets/survey_emp.c
)
```

```
survey.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 146 entries, 0 to 145
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   employee_id    146 non-null    int64  
 1   memory         146 non-null    float64
 2   processing     146 non-null    float64
 3   storage        146 non-null    float64
 4   inverse_price  146 non-null    float64
dtypes: float64(4), int64(1)
memory usage: 5.8 KB
```

```
survey.head()
```

	employee_id	memory	processing	storage	inverse_price
0	1743	0.375	0.225	0.150	0.25

Checking whether we have 100% in every column

```
for i, row in survey.iterrows():
    if sum(row[1:]) != 1:
        print("problem")
        break
print("ok")
```

```
ok
```

Data - computers

Lastly, the company is looking to purchase a maximum of 3 different computer models, and have compiled the following list scoring their memory, processing, storage, and relative price. Each dimension is scored from 0-10, with 10 being the best.

```
computers = pd.read_csv(
    "https://raw.githubusercontent.com/shubhamkalra27/dsep-2020/main/datasets/vendor_optio
)
```

```
computers.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11 entries, 0 to 10
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   computer_id    11 non-null     int64
 1   memory         11 non-null     int64
 2   processing     11 non-null     int64
 3   storage        11 non-null     int64
 4   inverse_price  11 non-null     float64
dtypes: float64(1), int64(4)
memory usage: 568.0 bytes
```

```
computers.head()
```

	computer_id	memory	processing	storage	inverse_price
1	16	9	8	9	1.3

Checking "real" range of scores:

```
print(computers.max())
```

```
computer_id      20.0
memory            9.0
processing       10.0
storage          10.0
inverse_price     5.7
dtype: float64
```

```
print(computers.min())
```

```
computer_id      1.0
memory           5.0
processing       4.0
storage          4.0
inverse_price    1.0
dtype: float64
```

Task

Given this information, provide the company with a recommendation on which computers to purchase.

List of parameters:

```
parameters = computers.columns[1:]
print(parameters)
```

```
Index(['memory', 'processing', 'storage', 'inverse_price'], dtype='object')
```

It will be more convenient for me to store all employees related data in one DataFrame instead of two.

Merging survey and utilization into employees column:

```
employees = utilization.merge(survey, left_on="employee_id", right_on="employee_id")
```

```
employees.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 146 entries, 0 to 145
Data columns (total 6 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
```

```
 0   employee_id      146 non-null    int64
 1   utilization_bin  146 non-null    object
 2   memory           146 non-null    float64
 3   processing       146 non-null    float64
 4   storage          146 non-null    float64
 5   inverse_price    146 non-null    float64
dtypes: float64(4), int64(1), object(1)
memory usage: 8.0+ KB
```

```
employees.head()
```

	employee_id	utilization_bin	memory	processing	storage	inverse_price
0	1743	high	0.375	0.225	0.150	0.25
1	1752	high	0.450	0.225	0.075	0.25
2	1758	high	0.375	0.300	0.075	0.25
3	1825	high	0.300	0.300	0.150	0.25
4	1842	high	0.300	0.300	0.150	0.25

Normally (if we would like to make some predictions) it would be better to store "object" (nominal data - categorical data) using one-hot-encoding but in this case it is more convenient for me to leave it like that.

Under we will see outputs of different metrics. For each there will be solution provided by:

- Simulated Annealing
- Naive algorithm

Results given by algorithms: id of computer in DataFrame - not the one in colum computer_id

```
from problem import (
    ProblemNothing,
    ProblemMax,
    ProblemScale,
    ProblemMaxHalf,
    ProblemScaleHalf,
)
```

- ProblemNothing - metric without using utilization info
- ProblemMax - metric where every group have a different max score (3, 7, 10)
- ProblemMaxHalf - like above with different values (5, 7.5, 10)
- ProblemScale - metric where we scale computers scores by multiplying them (3/1, 3/2, 3/3)
- ProblemScaleHalf - like above with different values (4/2, 4/3, 4/4)

```
from simulated_annealing import SimulatedAnnealing, SimulatedAnnealingConfig
from naive_solution import Naive
```

No utilization value

```
prob_n = ProblemNothing(computers, employees)
```

```
annealing_n = SimulatedAnnealing(SimulatedAnnealingConfig(), prob_n)

annealing_n.solve()
```

```
SOLUTION:
 Best: [10, 1, 8]
[10, 1, 8]
```

```
naive_n = Naive(prob_n)

naive_n.solve()
```

```
SOLUTION:
 Best: [1, 2, 8]
[1, 2, 8]
```

As we can see simulated annealing sometimes returns solutions that aren't optimal Let's try one more time

```
annealing_n = SimulatedAnnealing(SimulatedAnnealingConfig(), prob_n)

annealing_n.solve()
```

```
SOLUTION:
 Best: [8, 2, 1]
[8, 2, 1]
```

This time we manage to get "the best" solution for this metric

Let's see how it looks like:

```
prob_n.calculate_state_cost([10, 1, 8])
```

```
1024.650000000001
```

```
prob_n.calculate_state_cost([8, 2, 1])
```

```
1024.650000000001
```

So actually there are at least 2 optimal solutions

Problem MAX

max: (3, 7, 10)

```
prob_max = ProblemMax(computers, employees)
```

```
annealing_max = SimulatedAnnealing(SimulatedAnnealingConfig(), prob_max)
```

```
result_max = annealing_max.solve()
```

```
SOLUTION:
 Best: [4, 5, 1]
```

```
top_max = prob_max.get_wanted_computers(result_max)
```

```
print(top_max.keys())
```

```
dict_keys([1, 5, 4])
```

```
naive_max = Naive(prob_max)
```

```
result_max_n = naive_max.solve()
```

```
SOLUTION:
 Best: [1, 4, 5]
```

```
print(prob_max.get_wanted_computers(result_max_n).keys())
```

```
dict_keys([1, 5, 4])
```

max: (5, 7.5, 10)

```
prob_max_half = ProblemMaxHalf(computers, employees)
```

```
annealing_max_half = SimulatedAnnealing(SimulatedAnnealingConfig(), prob_max_half)
```

```
result_max_half = annealing_max_half.solve()
```

```
SOLUTION:
 Best: [1, 3, 4]
```

```
print(prob_max_half.get_wanted_computers(result_max_half).keys())
```

```
dict_keys([1, 3, 4])
```

```
naive_max_half = Naive(prob_max_half)
```

```
result_max_half_n = naive_max_half.solve()
```

```
SOLUTION:
 Best: [1, 3, 4]
```

```
print(prob_max_half.get_wanted_computers(result_max_half_n).keys())
```

```
dict_keys([1, 3, 4])
```

Problem Scale

scale: (3/1, 3/2, 3/3)

```
prob_scale = ProblemScale(computers, employees)
```

```
annealing_scale = SimulatedAnnealing(SimulatedAnnealingConfig(), prob_scale)
```

```
result_scale = annealing_scale.solve()
```

```
SOLUTION:
 Best: [4, 5, 1]
```

```
print(prob_scale.get_wanted_computers(result_scale).keys())
```

```
dict_keys([1, 5, 4])
```

```
naive_scale = Naive(prob_scale)
```

```
result_scale_n = naive_scale.solve()
```

```
SOLUTION:
 Best: [1, 4, 5]
```

```
print(prob_scale.get_wanted_computers(result_scale_n).keys())
```

```
dict_keys([1, 5, 4])
```

scale: (4/2, 4/3, 4/4)

```
prob_scale_half = ProblemScaleHalf(computers, employees)
```

```
annealing_scale_half = SimulatedAnnealing(SimulatedAnnealingConfig(), prob_scale_half)
```

```
result_scale_half = annealing_scale_half.solve()
```

```
SOLUTION:
 Best: [5, 1, 3]
```

```
print(prob_scale_half.get_wanted_computers(result_scale_half).keys())
```

```
dict_keys([1, 3, 5])
```

```
naive_scale_half = Naive(prob_scale_half)

result_scale_half_n = naive_scale_half.solve()
```

```
SOLUTION:
 Best: [1, 3, 5]
```

```
print(prob_scale_half.get_wanted_computers(result_scale_half_n).keys())
```

```
dict_keys([1, 3, 5])
```

There is still a question which 3 computers are the best for that company?

We can see according to all presented metrics computer in row nr 1 is always in top three

Others can vary depending on metric we use. We can choose one of the above mathod but there are only 3 candidates for 2 positions. nr five wasn't choosen only by problem max

```
print(result_max_half)
```

```
[1, 3, 4]
```

```
print(prob_max_half.calculate_state_cost(result_max_half))
```

```
885.2749999999984
```

```
print(prob_max_half.calculate_state_cost([1, 3, 5]))
```

```
884.3749999999992
```

```
print(prob_max_half.calculate_state_cost([1, 5, 4]))
```

```
876.6750000000017
```

For that metric there is not a big difference between computer 5 and 4 nr 3 seems to better option to leave

On the other hand we see that nr 4 also was't choosen only by on metric: scale half So it seems that we have final three

```
print(result_scale_half)
```

```
[5, 1, 3]
```

```
print(prob_scale_half.calculate_state_cost(result_scale_half))
```

```
1120.6249999999986
```

```
print(prob_scale_half.calculate_state_cost([1, 5, 4]))
```

```
1112.025000000003
```

```
print(prob_scale_half.calculate_state_cost([1, 4, 3]))
```

```
1112.624999999997
```

Computer nr 4 seems to be less valuable and changing it with 3 or 5 give almost the same results

Just in case: Let's check nr 3 - wasn't choseen by 2 metrics

```
print(result_max)
```

```
[4, 5, 1]
```

```
print(prob_max.calculate_state_cost(result_max))
```

```
852.549999999999
```

```
print(prob_max.calculate_state_cost([3, 5, 1]))
```

```
840.5500000000009
```

```
print(prob_max.calculate_state_cost([3, 4, 1]))
```

```
834.1249999999976
```

```
print(result_scale)
```

```
[4, 5, 1]
```

```
print(prob_scale.calculate_state_cost(result_scale))
```

```
1151.0499999999988
```

```
print(prob_scale.calculate_state_cost([3, 5, 1]))
```

```
1139.0500000000004
```

```
print(prob_scale.calculate_state_cost([3, 4, 1]))
```

```
1132.624999999997
```

As we can see using above metrics and adding computer 3 instead of any other (4 or 5) results in a bigger i other cases derease of cost value

Under we I displayed top three computers:

```
computers.iloc[[1, 4, 5]]
```

	computer_id	memory	processing	storage	inverse_price
1	16	9	8	9	1.3
4	3	5	4	4	5.7
5	2	6	7	7	3.3

seems like we ended up with rather ballanced final state

```
computers.iloc[[1, 4, 5]]["computer_id"]
```

```
1    16
4     3
5     2
Name: computer_id, dtype: int64
```

And here we have computer 3 as an addiction

```
computers.iloc[[3]]
```

	computer_id	memory	processing	storage	inverse_price
3	1	8	8	9	1.7

it is simmilar to computer nr 3 but with a slightly better price but worse memory

```
# problem.py
from collections.abc import Generator
from copy import deepcopy
from time import time
import pandas as pd
import pandas.core.frame as pd_frame
import pandas.core.series
```

```python
from random import shuffle
from abc import ABC, abstractmethod
from typing import TypeVar

series_type = TypeVar("pandas.core.series.Series")


class Problem(ABC):
    def __init__(self, computers: pd_frame.DataFrame, employees: pd_frame.DataFrame):
        self.employees = employees
        self.computers = computers
        self.parameters = computers.columns[1:]

    @abstractmethod
    def get_value_by_needs(self, value: int, employee_type: str) -> int:
        pass

    def get_random_state(self):
        state = [c for c in range(self.computers.shape[0])]
        shuffle(state)
        return state[:3]

    def calculate_computer_value_for_employee(
        self, computer: series_type, employee: series_type
    ) -> float:
        result = sum(
            [
                employee[col]
                * self.get_value_by_needs(computer[col], employee["utilization_bin"])
                for col in self.parameters[:-1]
            ]
        )
        result += employee[self.parameters[-1]] * computer[self.parameters[-1]]
        return result

    def get_best_from_three_for_employee(
        self, computers_indexes: list[int], employee: series_type
    ) -> float:
        return max(
            [
                self.calculate_computer_value_for_employee(
                    self.computers.iloc[c], employee
                )
                for c in computers_indexes
            ]
        )

    def calculate_state_cost(self, state: list[int]) -> float:
        cost = 0
        for _, e in self.employees.iterrows():
            cost += self.get_best_from_three_for_employee(state, e)
        return cost

    def improvement(self, new_state: list[int], old_state: list[int]) -> float:
        return self.calculate_state_cost(new_state) - self.calculate_state_cost(
```

```python
            old_state
        )

    def get_random_neighbour(self, state: list[int]) -> Generator:
        neighbour_states = [
            (i, j)
            for i in range(len(state))
            for j in [x for x in range(self.computers.shape[0]) if x not in state]
        ]

        shuffle(neighbour_states)

        for i, j in neighbour_states:
            new_state = deepcopy(state)
            new_state[i] = j
            yield new_state

    def get_all_states(self):
        result = []
        for i in range(self.computers.shape[0]):
            result += [[k, j, i] for j in range(i) for k in range(j)]

        return result

    def get_wanted_computers(self, computers_indexes):
        return {
            max(
                [c for c in computers_indexes],
                key=lambda x: self.calculate_computer_value_for_employee(
                    self.computers.iloc[x], employee
                ),
            ): None
            for _, employee in self.employees.iterrows()
        }


class ProblemMax(Problem, ABC):
    max_values = {"high": 10, "medium": 7, "low": 3}

    def get_value_by_needs(self, value: int, employee_type: str) -> int:
        return min(value, self.max_values.get(employee_type))


class ProblemMaxHalf(Problem, ABC):
    max_values = {"high": 10, "medium": 7.5, "low": 5}

    def get_value_by_needs(self, value: int, employee_type: str) -> int:
        return min(value, self.max_values.get(employee_type))


class ProblemScaleHalf(Problem, ABC):
    scale_values = {"high": 1, "medium": 4 / 3, "low": 4 / 2}

    def get_value_by_needs(self, value: int, employee_type: str) -> int:
        return min(value * self.scale_values.get(employee_type), 10)
```

```python
class ProblemScale(Problem, ABC):
    scale_values = {"high": 1, "medium": 3 / 2, "low": 3 / 1}

    def get_value_by_needs(self, value: int, employee_type: str) -> int:
        return min(value * self.scale_values.get(employee_type), 10)


class ProblemNothing(Problem, ABC):
    def get_value_by_needs(self, value: int, employee_type: str) -> int:
        return value


class Timer:
    def __init__(self, time_limit: float):
        self._time_limit = time_limit
        self._terminated = False

    def start_timer(self):
        self.start_time = time()

    def wall_time(self) -> float:
        return time() - self.start_time

    def is_timeout(self):
        return self.wall_time() > self._time_limit

    def stop_timer(self):
        self.total_time = self.wall_time()



# simulated_annealing.py
from dataclasses import dataclass
from problem import Problem, Timer
from random import random, randint
from math import exp
from typing import Union


@dataclass
# just some parameters for this problem
class SimulatedAnnealingConfig:
    initial_temperature: int = 5  # temperature at the beginning
    cooling_step: float = 0.999  # used to calculate temperature in th next step: how much
    min_temperature: float = 1e-10  # temperature can't be lower
    escape_reheat_ratio: float = 0.1  # when we want ro reheat we decrease initial temp -
    local_optimum_moves_threshold: int = (
        10  # parameter to determine whether we are stuck
    )


class SimulatedAnnealing:
    """
```

```python
    implementation of the simulated annealing algorithm.
    """

    best_state: Union[list[int], None] = None  # place to store the best state
    steps_from_last_state_update: int = 0  # parameter to determine whether we are stuck
    timer = Timer(60)  # to control time

    def __init__(self, config: SimulatedAnnealingConfig, problem: Problem):
        self.config = config  # parameters
        self.temperature = self.config.initial_temperature  # current temperature
        self.problem = problem  # our problem
        self.cooling_time = 0  # nr of steps since beginning / reheat

    def solve(self):
        self.timer.start_timer()  # set timer

        solution_state = self.problem.get_random_state()  # current state
        self.best_state = (
            solution_state  # our current state is the best what we have right now
        )

        while not self.timer.is_timeout():  # time limit
            next_state = self.next_state(solution_state)  # get next state

            if next_state:  # if we have new state we can start searching from there
                solution_state = next_state
            else:
                solution_state = self.best_state  # going back to the best one

        self.timer.stop_timer()
        print("SOLUTION:\n", "Best:", self.best_state)  # print best state

        return self.best_state  # return the best state

    def next_state(self, state: list[int]):
        if self._is_stuck_in_local_optimum():  # if we "think" that we are stuck
            next_state = self.escape_local_optimum(
                state, self.best_state
            )  # we are trying to escape (we have 2 options)
        else:
            next_state = self.find_next_state(
                state
            )  # else we are just trying to find next state

        if next_state is not None:  # if we have our next state
            self._update_state(state, next_state)  # we can update our information

        return next_state

    def _update_state(self, state: list[int], new_state: list[int]):
        # if self.best_state is None:
        #     self.best_state = new_state

        if self.problem.improvement(new_state, state) > 0:  # if there is an improvement
            self.steps_from_last_state_update = 0  # update to 0
        else:
```

```
    else:
        self.steps_from_last_state_update += 1  # update += 1

    if self.problem.improvement(new_state, self.best_state) > 0:
        self.best_state = new_state  # we can update our best state

def find_next_state(self, state: list[int]) -> list[int]:
    # — find random neighbour:
    #   [1] create a generator of the random neighbors
    generator = self.problem.get_random_neighbour(state)
    #   [2] use `next` to read a single element from a generator, e.g. `next(generator
    neighbour = next(generator)
    # — if the neighbour is better than mark is as the next state:
    #   [1] check for improvement
    if self.problem.improvement(neighbour, state) > 0:
        return neighbour
    # — otherwise calculate the probability of transition
    prob = self.calculate_transition_probability(state, neighbour)
    #   [1] use random() to generate a random number from range [0,1];
    p = random()
    #   [2] compare it to the probability to check if algorithm should go to the new s
    if p > prob:
        # — update temperature using `update_temperature`
        self.update_temperature()
        # — return the new state
        return neighbour

def calculate_transition_probability(
    self, old_state: list[int], new_state: list[int]
) -> float:
    return exp(self.problem.improvement(new_state, old_state) / self.temperature)

def update_temperature(self):

    # — update self.temperature according to the exponential decrease function:
    #   `T_k = T * a^k`
    # - make sure, the temperature can't go below `self.config.min_temperature`!
    self.temperature = max(
        self.temperature * (self.config.cooling_step**self.cooling_time),
        self.config.min_temperature,
    )
    # - update self.cooling_time
    self.cooling_time += 1

def reheat(self, from_state: list[int]):
    # — restore the initial temperature based on config (escape_reheat_ratio * initial_
    #   [1] initial temperature is stored in `self.config.initial_temperature`
    #   [2] you should decrease it a bit (multiply by `self.config.escape_reheat_ratio
    self.temperature = (
        self.config.initial_temperature * self.config.escape_reheat_ratio
    )
    # — reset cooling schedule (`self.cooling_time`)
    self.cooling_time = 0
    # — reset counter looking for local minima (`self.steps_from_last_state_update`)
    self.steps_from_last_state_update = 0
    # — return the `from_state`
```

```python
        # - return the "from_state"
        return from_state

    def escape_local_optimum(
        self, state: list[int], best_state: list[int]
    ) -> list[int]:
        strategies = ["random", "reheat"]
        strategy = strategies[randint(0, len(strategies) - 1)]
        if strategy == "random":
            return self.problem.get_random_state()
        if strategy == "reheat":
            return self.reheat(state)

    def _is_stuck_in_local_optimum(self):
        return (
            self.steps_from_last_state_update
            >= self.config.local_optimum_moves_threshold
        )



# naive_solution.py
from problem import Problem


class Naive:
    def __init__(self, problem: Problem):
        self.problem = problem

    def solve(self):
        states = self.problem.get_all_states()
        best_state = states[0]

        for state in states[1:]:
            if self.problem.improvement(state, best_state) > 0:
                best_state = state

        print("SOLUTION:\n", "Best:", best_state)
        return best_state
```