1. Problem description

A person that comes to her physician may be *healthy*, or she may have a *cold* or a *flu*. The physician has only 1 type of observation, the measurement of body temperature: it can be *below 37*, *37-38*, *38-39*, or *over 39* degrees Celsius. The physician must decide whether the person needs *no cure* at all, or if she suggests *tea and sweat*, prescribes *antibiotics*, or sends the person to a *specialist*.

Task 1: Define the sets X, K, and D, and represent them as lists `X`, `K`, and `D` in Python.

Solution 1:

In [1]:
```
X = ['below_37', '37-38', '38-39', 'over_39']
K = ['healthy', 'cold', 'flu']
D = ['no', 'tea', 'anti', 'spec']
```

2. Bayesian formulation

Task 2: Formulate the task in the Bayesian framework. What other information do we need?

Solution 2: To solve the task in Bayesian framework, we need to know the joint probability distribution $p_{XK}: X \times K \to \langle 0, 1 \rangle$, and the cost matrix $W : K \times D \to R$. The task is to find a strategy $q: X \to D$ such that the risk of the strategy $R(q) = \sum_{x \in X} \sum_{k \in K} p_{XK}(x, k) \cdot W(k, q(x))$ is minimal.

Task 3: Make sure you understand the given joint probability p_{XK}.

The probability distribution is stored in file `exB_pXK.csv` in a DB table format. Let's look at the contents:

In [2]:
```
# This is not pure Python; this is the IPython way of executing OS commands
!cat exB_pXK.csv
```

```
temp, state, rate
below_37, healthy, 0.05
below_37, cold, 0.03
below_37, flu, 0.01
37-38, healthy, 0.05
37-38, cold, 0.15
37-38, flu, 0.03
38-39, healthy, 0.02
38-39, cold, 0.25
38-39, flu, 0.10
over_39, healthy, 0.01
over_39, cold, 0.10
over_39, flu, 0.20
```

Task 4: Implement a helper function `load_data(filename)` to load the data from file (now for the joint distribution, later usable for the penalty matrix).

The function shall return a dictionary, where the first 2 columns of the CSV file contain the keys, and the third column contains the values.

Solution 4:

In [3]:
```
def load_data(filename):
    data = {}
    first_line = True
    with open(filename, 'rt', encoding='utf-8') as f:
        for line in f:
            if first_line:
                first_line = False
            else:
                parts = [x.strip() for x in line.split(',')] # List comprehe
nsion, see DIP3
                key = tuple(parts[:-1])
                value = float(parts[-1])
                data[key] = value
    return data
```

In [4]:
```
pXK = load_data('exB_pXK.csv')
pXK
```

Out[4]:
```
{('37-38', 'cold'): 0.15,
 ('37-38', 'flu'): 0.03,
 ('38-39', 'healthy'): 0.02,
 ('38-39', 'flu'): 0.1,
 ('below_37', 'flu'): 0.01,
 ('37-38', 'healthy'): 0.05,
 ('over_39', 'flu'): 0.2,
 ('over_39', 'healthy'): 0.01,
 ('below_37', 'healthy'): 0.05,
 ('below_37', 'cold'): 0.03,
 ('38-39', 'cold'): 0.25,
 ('over_39', 'cold'): 0.1}
```

Task 5: Find in the documentation what the `zip()` function does and how it can be used when constructing a dictionary. Try out the following code. It may come handy in the next tasks.

Solution 5:

In [5]:
```
keys = ['k1','k2','k3']
values = ['v1','v2','v3']
list(zip(keys, values))
```

Out[5]:
```
[('k1', 'v1'), ('k2', 'v2'), ('k3', 'v3')]
```

In [6]:
```
d = dict(zip(keys, values))
d
```

Out[6]:
```
{'k2': 'v2', 'k1': 'v1', 'k3': 'v3'}
```

If you have two lists, one containing keys and the other containing values, using the `zip()` function in the `dict()` constructor allows you to easilly construct a dictionary from keys and values.

Task 6: Implement a set of functions allowing you to easilly compute the marginal and conditional distributions based on p_{XK}.

In the following, we will have to iteratively add several values to individual dictionary items. In such situation an error often occurs, that when adding the first value, the dictionary item does not exist at all. Python offers several solutions to this:

- you can pre-initialize the dictionary with correct starting values (0.0 in our case),
- you can use the `dict.get(key, default_value)` method, which returns the value for the key, if the key exists in the dictionary, otherwise it returns the `default_value`, or
- you can use the `collections.DefaultDict` class which allows you to specify how it should initialize the nonexisting items.

Solution 6: In our solution, we use the `dict.get()` method.

In [7]:
```
def get_pX(pXK):
    pX = {}
    for x,k in pXK:
        pX[x] = pX.get(x, 0.0) + pXK[x,k]
    return pX
```

In [8]:
```
pX = get_pX(pXK)
pX
```

Out[8]:
```
{'37-38': 0.22999999999999998,
 'below_37': 0.09,
 'over_39': 0.31000000000000005,
 '38-39': 0.37}
```

In [9]:
```
sum(pX.values())
```

Out[9]:
```
1.0
```

In [10]:
```
def get_pK(pXK):
    pK = {}
    for x,k in pXK:
        pK[k] = pK.get(k, 0.0) + pXK[x,k]
    return pK
```

In [11]:
```
pK = get_pK(pXK)
pK
```

Out[11]:
```
{'cold': 0.53, 'flu': 0.34, 'healthy': 0.13}
```

In [12]:
```
def get_pXgK(pXK):
    pXgK = {}
    pK = get_pK(pXK)
    for x,k in pXK:
        pXgK[x,k] = pXK[x,k] / pK[k]
    return pXgK
```

In [13]:
```
pXgK = get_pXgK(pXK)
pXgK
```

Out[13]:
```
{('37-38', 'cold'): 0.2830188679245283,
 ('37-38', 'flu'): 0.08823529411764705,
 ('38-39', 'healthy'): 0.15384615384615385,
 ('38-39', 'flu'): 0.29411764705882354,
 ('below_37', 'flu'): 0.029411764705882353,
 ('37-38', 'healthy'): 0.38461538461538464,
 ('over_39', 'flu'): 0.5882352941176471,
 ('over_39', 'healthy'): 0.07692307692307693,
 ('below_37', 'healthy'): 0.38461538461538464,
 ('below_37', 'cold'): 0.056603773584905655,
 ('38-39', 'cold'): 0.4716981132075471,
 ('over_39', 'cold'): 0.18867924528301888}
```

In [14]:
```
def get_pKgX(pXK):
    pKgX = {}
    pX = get_pX(pXK)
    for x,k in pXK:
        pKgX[k,x] = pXK[x,k] / pX[x]
    return pKgX
```

In [15]:
```
pKgX = get_pKgX(pXK)
pKgX
```

Out[15]:
```
{('flu', 'below_37'): 0.11111111111111112,
 ('cold', 'below_37'): 0.3333333333333333,
 ('cold', '38-39'): 0.6756756756756757,
 ('flu', 'over_39'): 0.6451612903225805,
 ('flu', '37-38'): 0.13043478260869565,
 ('flu', '38-39'): 0.2702702702702703,
 ('healthy', '38-39'): 0.05405405405405406,
 ('healthy', 'over_39'): 0.032258064516129024,
 ('healthy', '37-38'): 0.2173913043478261,
 ('cold', '37-38'): 0.6521739130434783,
 ('cold', 'over_39'): 0.32258064516129026,
 ('healthy', 'below_37'): 0.5555555555555556}
```

Task 7: Design your own penalty matrix $W: K \times D \to R$ and store it in `exB_W.csv`. Load the data into variable `W`.

Solution 7:

In [18]:
```
W = load_data('exB_W.csv')
W
```

Out[18]:
```
{('healthy', 'anti'): 200.0,
 ('cold', 'no'): 500.0,
 ('cold', 'spec'): 200.0,
 ('cold', 'anti'): 100.0,
 ('flu', 'no'): 500.0,
 ('healthy', 'tea'): 0.0,
 ('flu', 'tea'): 100.0,
 ('cold', 'tea'): 0.0,
 ('healthy', 'no'): 0.0,
 ('healthy', 'spec'): 800.0,
 ('flu', 'anti'): 0.0,
 ('flu', 'spec'): 100.0}
```

Task 8: Make sure you understand what a *strategy* is in the Bayesian formulation. For us a strategy shall be represented by a dictionary, so that we can ask `q[x]` (What is the decision for observation x?). Create function `make_strategy()` which takes a list of possible observations and a list of corresponding decisions, and creates a strategy, i.e. a dictionary.

Solution 8:

In [19]:
```
def make_strategy(obs, dec):
    return dict(zip(obs, dec))
```

Task 9: How many different strategies $q: X \to D$ are there?

Solution 9: A strategy q is a function which assigns one of the possible decisions to each possible observation. The number of strategies is thus $|Q| = |D|^{|X|}$, i.e.

In [20]:
```
n_strategies = len(D)**len(X)
n_strategies
```

Out[20]:
```
256
```

Task 10: Given all the strategies will have the same keys, we can represent them (at first) just as a tuple of decisions. All the possible strategies can be generated using the `itertools.product()` function. Generate a list of all possible 4-tuples of decisions.

Solution 10:

In [21]:
```
from itertools import product
strategies_as_tuples = list(product(D,D,D,D))
strategies_as_tuples[:10]  # Just look at the first 10 strategies
```

Out[21]:
```
[('no', 'no', 'no', 'no'),
 ('no', 'no', 'no', 'tea'),
 ('no', 'no', 'no', 'anti'),
 ('no', 'no', 'no', 'spec'),
 ('no', 'no', 'tea', 'no'),
 ('no', 'no', 'tea', 'tea'),
 ('no', 'no', 'tea', 'anti'),
 ('no', 'no', 'tea', 'spec'),
 ('no', 'no', 'anti', 'no'),
 ('no', 'no', 'anti', 'tea')]
```

In [22]:
```
len(strategies_as_tuples)
```

Out[22]:
```
256
```

Task 11: Create function `make_list_of_strategies()`, which takes the list of possible observations `X`, and the list of possible decisions `D`, and produces a list of all possible strategies, i.e. list of dictionaries.

Solution 11:

In [23]:
```
def make_list_of_strategies(X, D):
    decision_sets = [D for x in X]    # This is a list comprehension, find i
t in Dive into Python
    strategies_as_tuples = list(product(*decision_sets))
    return [make_strategy(X, tup) for tup in strategies_as_tuples]  # Anothe
r list comprehension
```

In [24]:
```
strategies = make_list_of_strategies(X, D)
strategies[:10]    # Look at the first 10 strategies
```

Out[24]:
```
[{'below_37': 'no', '37-38': 'no', 'over_39': 'no', '38-39': 'no'},
 {'below_37': 'no', '37-38': 'no', 'over_39': 'tea', '38-39': 'no'},
 {'below_37': 'no', '37-38': 'no', 'over_39': 'anti', '38-39': 'no'},
 {'below_37': 'no', '37-38': 'no', 'over_39': 'spec', '38-39': 'no'},
 {'below_37': 'no', '37-38': 'no', 'over_39': 'no', '38-39': 'tea'},
 {'below_37': 'no', '37-38': 'no', 'over_39': 'tea', '38-39': 'tea'},
 {'below_37': 'no', '37-38': 'no', 'over_39': 'anti', '38-39': 'tea'},
 {'below_37': 'no', '37-38': 'no', 'over_39': 'spec', '38-39': 'tea'},
 {'below_37': 'no', '37-38': 'no', 'over_39': 'no', '38-39': 'anti'},
 {'below_37': 'no', '37-38': 'no', 'over_39': 'tea', '38-39': 'anti'}]
```

Task 12: Create function `print_strategy()` which takes a strategy (dictionary) `q` and a list of observations (keys) `X`, and pretty-prints the strategy so that the order of keys is such as specified in `X`.

Solution 12:

In [25]:
```
def print_strategy(q, X):
    for key in X:
        print('{:10s} {:s}'.format(key, q[key]))
```

In [26]:
```
print_strategy(strategies[9], X)
```

```
below_37   no
37-38      no
38-39      anti
over_39    tea
```

2.1. Bayesian strategy via complete search

Task 13: Create function `risk()` which returns the risk for a particular strategy q: $R(q) = \sum_{x \in X} \sum_{k \in K} p_{XK}(x,k) \cdot W(k, q(x))$ What other inputs does the function need?

Solution 13:

In [27]:
```
def risk(q, pXK, W):
    risk = 0
    for x,k in pXK:
        risk += pXK[x,k] * W[k,q[x]]
    return risk
```

In [28]:
```
risk(strategies[9], pXK, W)
```

Out[28]:
```
159.0
```

In [29]:
```
strategies[9]
```

Out[29]:
```
{'below_37': 'no', '37-38': 'no', 'over_39': 'tea', '38-39': 'anti'}
```

The resulting risk may be of course different, depending on your choice of W.

Task 14: Create function `find_bayesian_strategy()`. What inputs does the function need? Go through all possible strategies, compute their risks, find the strategy with the minimal risk. Return a 2-tuple: the Bayesian strategy and its risk.

You may find functions `numpy.argmax()` and `numpy.argmin()` useful.

Solution 14:

In [30]:
```
import numpy as np

def find_bayesian_strategy(X, K, D, pXK, W):
    risks = []
    strategies = make_list_of_strategies(X, D)
    # Compute risks for all strategies
    for q in strategies:
        r = risk(q, pXK, W)
        risks.append(r)
    # Choose the strategy with minimal risk
    imin = np.argmin(risks)
    return strategies[imin], risks[imin]
```

In [31]:
```
q_bs, risk_bs = find_bayesian_strategy(X, K, D, pXK, W)

print_strategy(q_bs, X)
```

```
below_37   tea
37-38      tea
38-39      tea
over_39    anti
```

In [32]:
```
risk_bs
```

Out[32]: 26.0

2.2. Bayesian strategy via partial risks

We do not need to search the whole space of possible strategies. If we use partial risks, we can construct the whole Bayesian strategy by choosing the optimal decision for each observation one by one.

Task 15: Create function `partial_risk()`, which returns the partial risk for a particular decision d and observation x: $R(d, x) = \sum_{k \in K} p_{K|X}(k|x) \cdot W(k, d)$. What other inputs are needed?

Solution 15:

In [33]:
```
def partial_risk(d, x, K, pKgX, W):
    pr = 0
    for k in K:
        pr += pKgX[k,x] * W[k,d]
    return pr
```

Task 16: Create function `find_bayesian_strategy_via_partial_risks()`. For each observation, compute the partial risk of all decisions, and assign the decision with the minimal partial risk. Return a 2-tuple: the optimal strategy and its risk.

Solution 16:

First, let's create a function `find_optimal_decision_for_observation()` which will produce the optimal decision for an obervation and the partial risk related to it.

In [34]:
```
def find_optimal_decision_for_observation(x, D, K, pKgX, W):
    prisks = []
    # Given the observation x, find the partial risk of all decisions
    for d in D:
        pr = partial_risk(d, x, K, pKgX, W)
        prisks.append(pr)
    # Find the optimal decision for the given observation
    imin = np.argmin(prisks)
    return D[imin], prisks[imin]
```

Now, we can construct the optimal strategy finding optimal decision for each observation, one by one.

In [35]:
```
def find_bayesian_strategy_via_partial_risks(X, K, D, pXK, W):

    pX = get_pX(pXK)
    pKgX = get_pKgX(pXK)
    q = {}
    risk = 0
    # For each observation, find the optimal decision separately
    for x in X:
        d_opt, prisk_opt = find_optimal_decision_for_observation(x, D, K, pK
gX, W)
        # Make the optimal decision part of the strategy
        q[x] = d_opt
        risk += pX[x] * prisk_opt
    return q, risk
```

In [36]:
```
q_bs2, risk2 = find_bayesian_strategy_via_partial_risks(X, K, D, pXK, W)
print_strategy(q_bs2, X)
```

```
below_37   tea
37-38      tea
38-39      tea
over_39    anti
```

In [37]:
```
risk2
```

Out[37]: 26.0

Unsuprisingly, the result is the same as in the previous case.

3. Estimating the hidden state

Let's move to a different task - estimating the hidden state K, i.e. $D = K$.

Task 17: Can the physician say anything about the hidden state of a patient **before she actually sees the patient?**

Solution 17: Well, yes, using the prior probabilities of states K the physician can identify the most probable state:

In [38]:
```
pK
```

Out[38]: {'cold': 0.53, 'flu': 0.34, 'healthy': 0.13}

The most probable state of the patient is *cold*.

Task 18: If the physician learns a new information about the patient – the body temperature X, she should update her beliefs and maybe change her estimate. Make sure you understand what a strategy is in this case. How many different strategies q: $X \to K$ are there? Can you create their list?

Solution 18:

In [39]:
```
n_strategies = len(K)**len(X)
n_strategies
```

Out[39]: 81

In [40]:
```
strategies = make_list_of_strategies(X, K)
strategies[:5]  # Just look at the first 5 strategies
```

Out[40]:
```
[{'below_37': 'healthy',
  '37-38': 'healthy',
  'over_39': 'healthy',
  '38-39': 'healthy'},
 {'below_37': 'healthy',
  '37-38': 'healthy',
  'over_39': 'cold',
  '38-39': 'healthy'},
 {'below_37': 'healthy',
  '37-38': 'healthy',
  'over_39': 'flu',
  '38-39': 'healthy'},
 {'below_37': 'healthy',
  '37-38': 'healthy',
  'over_39': 'healthy',
  '38-39': 'cold'},
 {'below_37': 'healthy',
  '37-38': 'healthy',
  'over_39': 'cold',
  '38-39': 'cold'}]
```

In [41]:
```
len(strategies)
```

Out[41]: 81

3.1. MAP estimation

We are still in the field of Bayesian formulation, but with $D = K$, and with square matrix W containing all ones and zeros only on its diagonal.

Task 19: Implement function `find_MAP_strategy()` which returns a strategy that will provide for each observation x an estimate of the hidden state k with minimal probability of error.

Solution 19: The optimal strategy which estimates the hidden state k with a minimal probability of error is the same as strategy that classifies the observation into the class with maximal posterior probability, i.e. it is given by $q(x) = \arg\max_{k \in K} p_{K|X}(k|x)$.

In [42]:
```
def find_MAP_strategy(X, K, pKgX):
    q = {}
    for x in X:
        pK_for_x = [pKgX[k,x] for k in K]
        imax = np.argmax(pK_for_x)
        q[x] = K[imax]
    return q
```

In [43]:
```
q = find_MAP_strategy(X, K, pKgX)
print_strategy(q,X)
```

```
below_37   healthy
37-38      cold
38-39      cold
over_39    flu
```

3.2. Minimax formulation

Now, we leave the world of Bayesian formulation of the decision task.

We still want to estimate the object state K based on the observation X. The strategy should assign a state to each observation with the aim to minimize the maximal probabilities of wrong decision across all true states. We will need only the conditional probabilities $p_{X|K}$; p_K and W are not required.

Task 20: Implement function `find_minimax_strategy()` which returns such a strategy that minimizes the maximal probability of wrong decisions across all possible states.

Solution 20:

First, let's define a function `compute_max_wd_prob_for_strategy()`. The result of this function is used as a score for a strategy.

In [44]:
```
def compute_max_wd_prob_for_strategy(q, X, K, pXgK):
    wd_probs = []
    for k in K:
        wd_prob_for_k = sum(pXgK[x,k] for x in X if q[x] != k)
        wd_probs.append(wd_prob_for_k)
    # Return the maximum probability of a wrong decision
    return np.max(wd_probs)
```

Now, we are able to compute the score for all strategies, and return the one with minimal score.

In [47]:
```
def find_minimax_strategy(X, K, pXgK):
    max_wd_probs = []
    strategies = make_list_of_strategies(X, K)
    for q in strategies:
        # Compute probability of wrong decision for all states k
        max_wd_prob = compute_max_wd_prob_for_strategy(q, X, K, pXgK)
        max_wd_probs.append(max_wd_prob)
    # Among all strategies, find the one with minimal max_wd_prob
    imin = np.argmin(max_wd_probs)
    return strategies[imin]
```

In [48]:
```
q_mm = find_minimax_strategy(X, K, pXgK)
print_strategy(q_mm, X)
```

```
below_37   healthy
37-38      healthy
38-39      cold
over_39    flu
```

Note that in this case the minimax strategy is different from the MAP strategy.

Summary

We have demonstrated the principles of Bayesian decision making on a very simple task. We have shown two alternative ways how to come up with the Bayesian strategy. We have also shown how different formulations (and thus different objectives) result in different strategies.

We did quite a lot of programming in Python. But imagine what amount of work would have to be carried out in your favourite programming language to reach the same results...