Reinforcement Learning and Artificial Intelligence in the context of revenue management

Freie wissenschaftliche Arbeit
zur Erlangung des akademischen Grades
"Master of Science"
Studiengang: Finanz- und Informationsmanagement

an der
Wirtschaftswissenschaftlichen Fakultät
der Universität Augsburg

– Lehrstuhl für Analytics & Optimization –

Eingereicht bei:	Prof. Dr. Robert Klein
Betreuer:	Dr. Sebastian Koch
Vorgelegt von:	Stefan Glogger
Adresse:	tbd
Matrikel-Nr.:	tbd
E-Mail:	stefan.glogger@student.uni-augsburg.de
Datum:	July 23, 2019

Contents

Chapter 1

Aufzuschreiben

This is a quick overview of my results, that I want to incorporate in the thesis.

1. Computing the Offer Set Dynamically as presented in Bront et Al 5.2 and reproduce example of Table 3 to proof correct implementation `C:\Users\Stefan\LRZ Sync+Share\Masterarbeit-Klein\Code\notebooks\0427 - Deep dive into DPD Offer Set Calculation.ipynb`
2. Most structure of all coding until May also in `C:\Users\Stefan\LRZ Sync+Share\Masterarbeit-`
3.

Inhaltlich

1. Einleitung mit
    a) Literatur
    b) Problembeschreibung
    c) Notation
2. Methoden mit nur mini Beispielen
    a) Dynamic Programming
    b) CDLP
    c) API mit Miranda Bront Heuristik
    d) Neural Network
3. Theorie zum Vergleich verschiedener Methoden (Statistik)
4. 1 Single Leg Beispiel
5. 1 Multi Leg Beispiel
6. Zusammenfassung

Chapter 2

Problem and Approaches to Solution

2.1 The problem

Here, we want to lay out the classical revenue management problem an the "brute force" approach to solve.

Consider a firm that produces products $j = 1, \dots, n$ with revenues $\boldsymbol{r} = (r_1, \dots, r_n)^T$. Resources $h = 1, \dots, m$ are used for production. In order to produce one unit of product j, resources $\boldsymbol{a}_j = (a_{1j}, \dots, a_{mj})^T$ are necessary, with $a_{hj} = 1$ if resource h is needed for production of product j and $a_{hj} = 0$ otherwise. Initially, the capacity is described by $\boldsymbol{c}^0 = (c_1^0, \dots, c_m^0)^T$.

The booking horizon is modelled by sufficiently small time periods $t = 0, \dots, T-1$, such that in each time period at most one customer arrives. This customer also purchases at most one product. If product j is purchased at time t, the capacity reduces to $\boldsymbol{c}^{t+1} = \boldsymbol{c}^t - \boldsymbol{a}_j$. Time moves forward, such that the last selling might occur at time $T-1$.

The firm wants to increase the value of the products sold and has flexibility in the sets offered. Thus, the decision variables at each time point t are given by $\boldsymbol{x}^t = (x_1^t, \dots, x_n^t)^T$ with $x_j^t = 1$ if product j is offered at time t and $x_j^t = 0$ otherwise. Thus, at each time point t and each capacity $\boldsymbol{c}$, the offer set $\boldsymbol{x}$ has to be determined. Note that in machine learning terms, this mapping is defined as a policy $\boldsymbol{\psi} = (\psi^1, \dots, \psi^T)^T$.

One popular method of describing the probabilities of purchases is to have each customer belonging to one customer segment $l = 1, \dots L$, each of which following a multinomial logit model (MNL). A customer of segment l arrives with probability λ_i. His preference weights are given by $\boldsymbol{u}_l = (u_{l1}, \dots, u_{ln})^T$ and no purchase preference of u_{l0}. Note: $u_{lj} > 0$ if consumer of segment l might purchase product j (the higher, the more interested) and $u_{lj} = 0$ if customer is not interested in product. The probability of purchasing product j

when set $\boldsymbol{x}$ is offered is given by $p_{lj}(\boldsymbol{x}) = \frac{u_{lj}x_j}{u_{l0}+\sum_{p\in[n]} u_{lp}x_p}$ and the no-purchase probability is given by $p_{l0}(\boldsymbol{x}) = 1 - \sum_{j\in[n]} p_{lj}$. Together with the uncertainty of which customer segment arrives (if any), we end up at a purchase probability for product j given $\boldsymbol{x}$ of $p_j(\boldsymbol{x}) = \sum_{l\in[L]} \lambda_l p_{lj}(\boldsymbol{x})$ and a no purchase probability of $p_0(\boldsymbol{x}) = 1 - \sum_{j\in[n]} p_j(\boldsymbol{x})$. Thus, $p_j(\boldsymbol{x})$ can be interpreted as the deterministic quantity of product j being sold, if set $\boldsymbol{x}$ is offered.

2.1.1 Dynamic Programming

The first solution approach might be to maximize the expected value of all revenues to gain given period t and capacity $\boldsymbol{c}$, denoted by the value function $V^t(\boldsymbol{c})$. This can then be computed recursively as

$$V^t(\boldsymbol{c}) = \max_{x^t\in\{0,1\}^n} \left\{ \sum_{j\in[n]} p_j(x_t)\left(r_j + V^{t+1}(\boldsymbol{c}-\boldsymbol{a}_j)\right) + p_0 V^{t+1}(\boldsymbol{c}) \right\} \tag{2.1}$$ {eq-Bell

$$= \max_{x^t\in\{0,1\}^n} \left\{ \sum_{j\in[n]} p_j(x_t)\left(r_j - \Delta_j V^{t+1}(\boldsymbol{c})\right) \right\} + V^{t+1}(\boldsymbol{c}) \quad \forall t, \boldsymbol{c} \geq 0 \tag{2.2}$$

with $\Delta_j V^{t+1}(\boldsymbol{c}) := V^{t+1}(\boldsymbol{c}) - V^{t+1}(\boldsymbol{c}-\boldsymbol{a}_j)$ and boundary conditions $V^{T+1}(\boldsymbol{c}) = 0$ if $\boldsymbol{c} \geq \boldsymbol{0}$ and $V^t(\boldsymbol{c}) = -\infty$ if $\boldsymbol{c} \ngeq \boldsymbol{0}$.

The optimal offerset ${}^*\boldsymbol{x}^t$ is then given by

$${}^*\boldsymbol{x}^t = \arg\max_{x^t\in\{0,1\}^n} \left\{ \sum_{j\in[n]} p_j(x_t)\left(r_j - \Delta_j V^{t+1}(\boldsymbol{c})\right) \right\} + V^{t+1}(\boldsymbol{c}) \quad \forall t, \boldsymbol{c} \geq 0 \tag{2.3}$$ {eq-DP-o

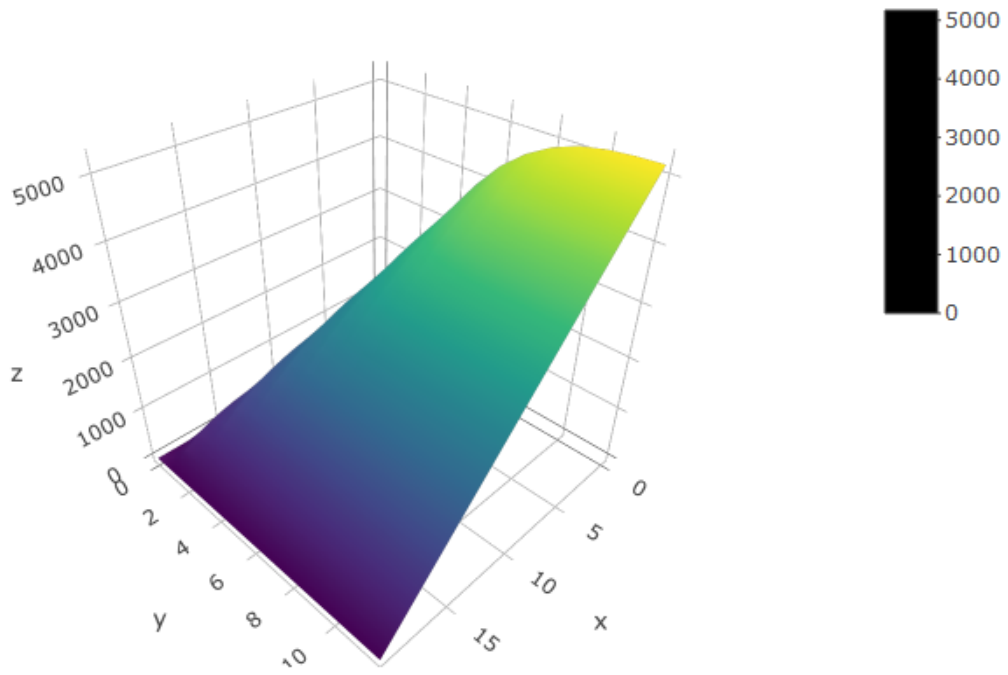
The resulting value function can be plotted in a three dimensional plot as done in Figure 2.1.

2.1.2 Dynamic Programming - Theory

As this whole thesis is based on the ideas of Dynamic Programming, we want to give a short overview of the underlying mathematical theory and directly combine it to our setting.

Dynamic Programming (DP) refers to a broad collection of algorithms used to compute

fig-valueFunc
Figure 2.1: Value function of single leg flight example with $t \in [20]$ on x-axis, $c \in [12]$ on y-axis.



optimal policies given a perfect model of the environment as a Markov Decision Process (MDP), as stated in Sutton and Barto (2018). A MDP is characterized by a state set $\mathcal{S}$ (the varying capacities $\mathbf{c}$ together with the current point in time t), an action set $\mathcal{A}$ (the offer sets $\mathbf{x}$) and reward sets $\mathcal{R}$ (the revenue r if a product is sold). A MDP evolves over time, but for the evolution from time t to $t+1$ only the current state s_t is relevant and the history of previous states $s_h, h < t$ can be ignored. The dynamic is given by a set of probabilities $p(s', r|s, a)$ (capacities reduce according to which random product the random customer has bought according to his preferences).

The goal of DP is to determine the optimal policy, i.e. which action to choose at each given state. Key to the solution is the usage of value functions as seen above. These fulfil the Bellman optimality equations as stated in Equation (2.1).

In our setting, uniqueness of the optimal value function is ensured, as the MDP is guaranteed to terminate under any policy because time is moving forward no matter which action we choose, compare Sutton and Barto (2018). Thus, while the problem can be solved, due to large scale decision problems and the curse of dimensionality, we are lack-

ing computing power to solve it exactly and have to come up with approximate solution methods.

One approximate solution method is the usage of Approximate Policy Iteration (API), which consists of two steps. The policy evaluation step (inner loop Line 4 to Line 15) evaluates a fixed policy over a set of sample paths $\omega_i, i = 1, \ldots, I$. The policy improvement step (outer loop Line 2, Line 3, Line 16) improves the policy over the iterations $k = 1, \ldots, K$.

2.1.3 Choice based linear programming

based on Bront et al and van Ryzin.

Taking an eagle-eye perspective, the company has to decide which sets to offer in every single time period, but as the probabilities don't change over time, the specific time point when to offer one set is indistinguishable, we can aggregate over time. Let us introduce a few more notation to get a hand on this. Let $R(\mathbf{x})$ represent the expected revenue given offer set $\boldsymbol{x}$, i.e.

$$R(\boldsymbol{x}) = \sum_{j \in [n]} r_j p_j(x) \quad . \tag{2.4}$$

Furthermore, let $\boldsymbol{Q}(\boldsymbol{x}) = (Q_1(\boldsymbol{x}), \ldots, Q_m(\boldsymbol{x}))^T$ denote the expected capacity being used, i.e.

$$Q_h(\boldsymbol{x}) = \sum_{j \in [n]} a_{hj} p_j(\boldsymbol{x}) \quad . \tag{2.5}$$

We can denote the total time set $\boldsymbol{x}$ is offered by $t(\boldsymbol{x})$ and the set of all possible offersets as $N = \{0, 1\}^n$. Thus, we can formulate the choice-based linear program (CDLP) as presented in van Ryzin and Liu. Here the optimization problem is solved by varying the decision variables $t(\boldsymbol{x})$.

$$V^{CDLP} = \max \sum_{\boldsymbol{x} \in N} R(\boldsymbol{x}) t(\boldsymbol{x}) \tag{2.6}$$

$$\text{s.t.} \sum_{\boldsymbol{x} \in N} Q_h(\boldsymbol{x}) t(\boldsymbol{x}) \leq c_h \quad , \forall h \tag{2.7}$$

$$\sum_{\boldsymbol{x} \in N} t(\boldsymbol{x}) \leq T \ , \tag{2.8}$$

$$t(\boldsymbol{x}) \geq 0 \quad \forall \boldsymbol{x} \in N \ . \tag{2.9}$$

Korrektheit getestet mit `C:\Users\Stefan\LRZ Sync+Share\Masterarbeit-Klein\Code\H_CDLP_si` - Example0 in settings Ergebnisse in `C:\Users\Stefan\LRZ Sync+Share\Masterarbeit-Klein\Code\`

2.2 Approximate Dynamic Programming

2.2.1 Approximate Policy Iteration

ADP builds upon Equation (2.3) and approximates opportunity costs additively using bid prices $\pi_h(t, c_h)$, i.e. $\Delta_j V^{t+1}(\boldsymbol{c}) = \sum_{h \in [m]} a_{hj} \pi_h(t+1, c_h)$. Thus, the policy is given as the solution of

$${}^{\pi}\boldsymbol{x}^t = \underset{x^t \in \{0,1\}^n}{\arg\max} \left\{ \sum_{j \in [n]} p_j(x_t) \left(r_j - \sum_{h \in [m]} a_{hj} \pi_h(t+1, c_h) \right) \right\} \tag{2.10}$$

Thus, the policy is completely described by the bid prices $\pi_h(t, c_h) \forall t$.

Let us discuss the algorithm in depth. In the beginning (Line 1), all parameters are set to zero which results in the greedy policy of offering the set resulting in the highest expected revenue (no consideration of costs).

In each policy iteration k, the current policy is evaluated first. To do so, we use the current policy to evaluate I random paths. Line 3 sets up the dataset for revenues to go $\hat{V}_t^i$ and remaining capacities $\hat{\mathbf{C}}_t^i$ for all times t and sample paths i.

For each sample path i, a storage unit for revenue $\hat{r}_t$ and capacities $\hat{\mathbf{c}}$ is created for all times t (Line 5). Furthermore, the capacity is set to the initial capacity in Line 6.

For each point in time, the current capacity $\hat{\mathbf{c}}_t$ is stored in Line 8, the current bid prices are calculated in Line 9 and used to determine the offerset $\mathbf{x}$ in Line 10. More details on

the determination of the offerset can be found in Section 2.2.2. With, this, a sales event j' is simulated and in case a product has been sold, the revenue is stored and capacity adjusted (note: the revenue thus belongs to t and capacity will affect $t+1$) (Line 11 to Line 13).

After simulating one sample path, the value function for this sample path $\hat{V}_t$ is updated for all times t in Line 14, and so are the capacities in line Line 15.

With all the information from the I sample paths the parameters (θ_t, π_t) are updated. Note that the old parameters are used as starting values and the current number of policy iterations k is also passed to potentially take care of exponential smoothing. Thus, we have $(1+h)*T$ parameters ($theta_t$ and π_t) and $2TI$ data points. More details on the update of the parameters can be found in Section 2.2.3.

Actions possible in $t = 0, \ldots, T-1$.

Algorithm 1 Approximate policy iteration

1: Set $\theta_t = 0$ and $\pi_t = \mathbf{0}$ $\forall t = 0, \ldots, T-1$
2: **for** `k = 1 to K` **do**
3: Set $\hat{V}_t^i = 0$ and $\hat{\mathbf{C}}_t^i = 0$ $\forall t = 0, \ldots, T-1, \forall i = 1, \ldots, I$
4: **for** `i = 1 to I` **do**
5: Set $\hat{r}_t = 0$ and $\hat{\mathbf{c}}_t = 0$ $\forall t = 0, \ldots, T-1$
6: Initialize $\mathbf{c} = \mathbf{c}^0$
7: **for** `t = 0 to T-1` **do**
8: $\hat{\mathbf{c}}_t \coloneqq \mathbf{c}$
9: Compute $\pi(t, \mathbf{c})$
10: Compute $\mathbf{x} = \text{determineOfferset}(\pi(t, \mathbf{c}), \epsilon_t)$
11: Simulate a sales event $j' \in \{0, 1, \ldots, n\}$
12: **if** $j' \in \{1, \ldots, n\}$ **then**
13: $\hat{r}_t = r_{j'}$ and $\mathbf{c} = \mathbf{c} - \mathbf{a}_{j'}$
14: Compute $\hat{V}_t^i = \sum_{\tau=t}^{T} \hat{r}_t \quad \forall t = 0, \ldots, T-1$
15: Assign $\hat{\mathbf{C}}_t^i = \hat{\mathbf{c}}_t \quad \forall t = 0, \ldots, T-1$
16: $(\theta_t, \pi_t) = \text{updateParameters}\left(\hat{V}_t^i, \hat{\mathbf{C}}_t^i, \theta_t, \pi_t, k\right) \quad \forall t = 0, \ldots, T-1, \forall i = 1, \ldots, I$
return $(\theta_t, \pi_t) \quad \forall t = 1, \ldots, T$

{alg-API}
{alg-API1
{alg-API-
{alg-API3
{alg-API-
{alg-API5
{alg-API6
{alg-API8
{alg-API9
{alg-API1
{alg-API1
{alg-API1
{alg-API1
{alg-API-

Overview of parameters:

1. θ_t optimization parameter (offset)
2. π_t optimization parameter (bid price for each resource)
3. $\hat{V}_t = 0$ all sample revenues to go for each sample for each time

4. $\hat{\mathbf{C}}_t = 0$ all sample available capacities for each sample for each resource for each time
5. r_t sample revenue generated at time t
6. $\mathbf{c}$ available capacities for each resource at current time
7. $\mathbf{c}^0$ starting capacities
8. $\mathbf{x}$ offerset at current time
9. ϵ_t epsilon used at current time

2.2.2 Greedy Heuristic to determine offerset

{sec-dete

The following algorithm is based on the ideas of the greedy heuristic for the column generation subproblem outlined in Bront et al. (2009).

Our goal is to determine a reasonable set of products to offer in a fast manner. Thus, we use a heuristic and cut down the amount of products to consider as fast as possible.

The function determineOfferset(π, ϵ) calculates the set to offer depending on the current bid prices π via the greedy algorithm layed out in Bront et al. (2009). To account for the exploration vs exploitation dilemma, an epsilon-greedy strategy is used. With a probability of $\epsilon/2$ either no product is offered at all or all products with positive contribution $r_j - \sum_{h\in[m]} a_{hj} \cdot \pi_h$ are offered. With a probability of $1-\epsilon$, the proper calculated set is offered.

Value Funktion gebuendelt dargestellt und ueberall mit λ. Vergleiche zu Bront et al. 4.2.2, wo in 3. ohne λ und in 4.a mit λ.

Algorithm 2 Greedy Heuristic {alg-Gree

1: $\text{Value}(X) \coloneqq \sum_{l=1}^{L} \lambda_l \frac{\sum_{i\in X}(r_i - A_i^T\pi)u_{li}}{\sum_{i\in X} u_{li}+u_{l0}}$
2: $S \coloneqq \emptyset, \quad S' \coloneqq \left\{j \in N : r_j - A_j^T\pi > 0\right\}$ {alg-L1}
3: $j^* \coloneqq \arg\max_{j\in S'} \text{Value}(\{j\})$
4: **repeat**
5: $\quad S \coloneqq S \cup \{j^*\}, \quad S' \coloneqq S'\backslash\{j^*\}$
6: $\quad j^* \coloneqq \arg\max_{j\in S'} \text{Value}(S \cup \{j\})$
7: **until** $\text{Value}(S \cup \{j^*\}) \leq \text{Value}(S)$
8: **return** S

Let S' be the set of products with positive reduced costs, i. Line 2

2.2.3 Update of parameters

{sec-upda

The function $(\theta_t, \pi_t) = \text{updateParameters}\left(\hat{V}_t, \hat{\mathbf{C}}_t, \theta_t, \pi_t, k\right)$ really optimizes the following least squares optimization problem for all parameters $(t = 1, \ldots, T)$ at the same time.

$$V_t(\theta_t, \pi_t, \mathbf{c}_t) \coloneqq \theta_t + \sum_{h=1}^{m} \sum_{s=1}^{S_h} \pi_{ths} f_{hs}(c_h) \tag{2.11}$$

$$f_{hs}(c_h) \coloneqq \begin{cases} 0 & \text{if } c_h \leq b_h^{s-1} \\ c_h - b_h^{s-1} & \text{if } b_h^{s-1} < c_h \leq b_h^s \\ b_h^s - b_h^{s-1} & \text{if } b_h^s < c_h \end{cases} \tag{2.12}$$

{{def-f}}

Equation (2.12) describes the occupied amount of capacity of interval $\left(b_h^{s-1}, b_h^s\right]$.

The following optimization problem depends on the old parameters $\theta_t = \theta_t^k$ and $\pi_t = \pi_t^k$ to determine the optimal parameter θ_t^{update} and π_t^{update}.

$$\min \sum_{i=1}^{I} \sum_{t=1}^{T} \left(\hat{V}_t^i - V_t(\theta_t, \pi_t, \mathbf{c}_t^i)\right)^2 \tag{2.13}$$

$$s.t. \tag{2.14}$$

$$\theta_t \geq 0 \qquad \forall t \tag{2.15}$$

$$\max_{j=1,\ldots,n} r_j \geq \pi_{ths} \geq 0 \qquad \forall t, h, s \tag{2.16}$$

$$\pi_{ths} \geq \pi_{th,s+1} \qquad \forall t, h, s = 1, \ldots, S_h - 1 \tag{2.17}$$

$$\theta_t \geq \theta_{t+1} \qquad \forall t = 1, \ldots, T-1 \tag{2.18}$$

$$\pi_{ths} \geq \pi_{t+1,hs} \qquad \forall t = 1, \ldots, T-1 \tag{2.19}$$

The final parameters θ_t^{K+1} and π_t^{K+1} can obtained via two possible equally possible ways. One is the so called exponential smoothing, where in each iteration k the parameter for the next iteration $k+1$ is calculated via:

$$\theta_t^{k+1} = \left(1 - \frac{1}{k}\right) \theta_t^k + \frac{1}{k} \theta_t^{update} \tag{2.20}$$

$$\pi_t^{k+1} = \left(1 - \frac{1}{k}\right) \pi_t^k + \frac{1}{k} \pi_t^{update} \tag{2.21}$$

The other one uses $\theta_t^{k+1} = \theta_t^{update}$ and $\pi_t^{k+1} = \pi_t^{update}$ and averages at the very end.

$$\theta_t^{K+1} = \frac{1}{K} \sum_{k=1}^{K} \theta_t^k \tag{2.22}$$

$$\pi_t^{K+1} = \frac{1}{K} \sum_{k=1}^{K} \pi_t^k \tag{2.23}$$

Proof.

Den Beweis sauber ausfuehren. Hier sind die Inhalte. Die Aussage stimmt, falls die gefundenen optimalen Loesungen in jeder Iteration stets dieselben sind. Dies ist der Fall, wenn es stets nur ein Minimum gibt. Wir haben eine quadratische Zielfunktion, die sozusagen eine mehrdimensionale, nach oben geoeffnete Parabel zeigt, die genau ein globales Minimum besitzt. Weitere noetige Punkte: eindeutiges globales Minimum ueber positiv definite Hesse-Matrix. Optimum auch zulaessig (schwierig?)

■

2.2.4 More stuff

Calculations: $\pi(t, \mathbf{c}) = \pi_h(t, c_h)$ for $h \in [m]$

$$\pi_h(t, c_h) = \begin{cases} \infty & \text{if } c_h = 0 \qquad (2.24) \\ \sum_{s=1}^{S_h} \pi_{ths} \mathbb{1}_{\left(b_h^{s-1}, b_h^s\right]}(c_h) & \text{otherwise.} \qquad (2.25) \end{cases}$$

Fuer Line 9 verwende Zeit **t** statt **t+1**. Grund: Kenne Informationen zur Zukunft nicht.

A sales event is simulated by first having one or zero customer arrive at random. In case a customer arrives, its preference function given the offer set determines the probability according to which one product is sold ($j' \in \{1, \ldots, n\}$) or no product is sold ($j' = 0$).

Chapter 3

Examples

3.1 Example0

One interesting result is that for exact reproduction of the results of Bront et al for their example0, the empty offerset has to be excluded from the optimization problem. Otherwise (including empty set) results in the same optimal value but via another combination of the decision variables.

Example 0 represents a small airline network. Three cities are interconnected by three flights (legs), with a capacity vector $\boldsymbol{c} = (10, 5, 5)^T$. The booking horizon consists of $T = 30$ periods and there are five customer segments with preferences as in Table 3.1

Table 3.1: Segment definition for Example 0 `tb-Example0-Customers`

Segment	λ_l	Consideration set	Preference vector	No purchase preference	Descripti
1	0.15	$\{1, 5\}$	$(5, 8)$	2	Price sensitive, non

Figure 3.1: Airline network for example 0. `fig-Example0`

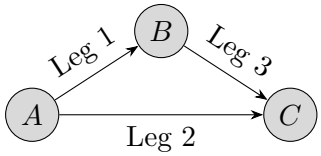


Table 3.2: Airline network for example 0 (products). `tb-Example0-Products`

Product	Origin-destination	Fare
1	$A \to C$	$1,200$
2	$A \to B \to C$	800
3	$A \to B$	500
4	$B \to C$	500
5	$A \to C$	800
6	$A \to B \to C$	500
7	$A \to B$	300
8	$B \to C$	300

Chapter 4

Main chapter

4.1 Notation

1. time index always at the top
2. index for products always at bottom

4.1.1 Examples

Single-leg flight example

For reasons of comparability, we use the same example as in Koch (2017). An airline offers four products with revenues $\boldsymbol{r} = (1000, 800, 600, 400)^T$ over $T = 400$ periods. Only one customer segment exists with arrival probability of $\lambda = 0.5$ and preference weights $\boldsymbol{u} = (0.4, 0.8, 1.2, 1.6)^T$. Different network loads can be analyzed by varying initial capacity $c^0 \in \{40, 60, \ldots, 120\}$ and varying no-purchase preference weights $u_0 \in \{1, 2, 3\}$.

4.2 Implementation

Here, we present the results of the exact calculation for the single leg flight example.

Storage folder:

`"C:/Users/Stefan/LRZ Sync+Share/Masterarbeit-Klein/Code/Results/singleLegFlight-Tru`

Log:

```
Time (starting): 2019-06-11 09:17:42.448064
Example :  singleLegFlight
use_var_capacities :  True
```

```
Total time needed:
 151.31108283996582 seconds =
 2.521851380666097 minutes
```

Results:

	capacity	no-purchase preference	DP-value	DP-optimal offer set
0	40	1	35952	(1, 1, 0, 0)
1	60	1	49977.1	(1, 1, 0, 0)
2	80	1	59969.8	(1, 1, 1, 0)
3	100	1	65956.4	(1, 1, 1, 0)
4	120	1	68217.2	(1, 1, 1, 1)
5	40	2	35952	(1, 1, 0, 0)
6	60	2	49977.1	(1, 1, 0, 0)
7	80	2	59969.8	(1, 1, 1, 0)
8	100	2	65956.4	(1, 1, 1, 0)
9	120	2	68217.2	(1, 1, 1, 1)
10	40	3	35952	(1, 1, 0, 0)
11	60	3	49977.1	(1, 1, 0, 0)
12	80	3	59969.8	(1, 1, 1, 0)
13	100	3	65956.4	(1, 1, 1, 0)
14	120	3	68217.2	(1, 1, 1, 1)

Furthermore, we want to visualize the value function and its approximations. We use the example of $c^0 = 60$ and $u_0 = 1$.

4.2.1 Network flight example

Chapter 5

Comparison of different policies

5.1 Two sample test

Our goal is to evaluate two different policies. One policy is considered better then another policy if it generally leads to higher total revenues. To put this vague phrasing into a scientifically sound comparison, we use an approximate two sample test, which we'll describe here in more detail. We base this overview and notation mainly on Ch. 14.7 of Bamberg et al. (2011) with extensions as in Ch. 11.3 of Fahrmeir et al. (2007).

We have a total of M sample paths each consisting of T time steps, i.e. $M \times T$ random numbers determining which customer arrives and $M \times T$ random numbers leading to which product is purchased. Those two numbers are called *exogenous information*. When applying policy A, for each sample path at each time step, the respective offerset is presented to the current customer and its preferences determine which product is purchased. All purchases of one sample path are aggregated and stored, such that policy A results in the vector of values $\mathbf{v}^A \in \mathbb{R}^M$. Accordingly, $\mathbf{v}^B$ is determined. As both policies rely on the same exogenous information, the samples are dependent and a paired samples t-test has to be used.

Thus, we end up with two observations of size M

$$v_1^A, \ldots, v_M^A \text{ respectively } v_1^B, \ldots, v_M^B \ .$$

Let $\bar{\mathbf{v}^A}$ and S_A^2 resp. $\bar{\mathbf{v}^B}$ and S_B^2 be the empirical mean and empirical variance. Furthermore, μ^A and σ_A^2 denote the expected value and variance of v_i^A, respectively μ^B and σ_B^2 denote the expected value and variance of v_i^B. The hypothesis that policy A is better then policy B results in the following null and alternative hypothesis:

$$H_0 : \mu^A \leq \mu^B, H_1 : \mu^A > \mu^B$$

As $\bar{\mathbf{v}}^A$ and $\bar{\mathbf{v}}^B$ are unbiased estimators of μ^A and μ^B, the difference $D = \bar{\mathbf{v}}^A - \bar{\mathbf{v}}^B$ can be used to test the hypothesis. A large value of D represents the data to be in favour of H_1.

As we don't know the distribution of v_i^A or v_i^B and the sample size $M > 30$, we know for the test statistic

$$T = \frac{\bar{\mathbf{v}}^A - \bar{\mathbf{v}}^B}{\sqrt{\frac{S_A^2 + S_B^2}{M}}} \sim N(0; 1) \ .$$

With this knowledge, we can compute the p-value according to $p = 1 - \Phi(T)$ with $\Phi(.)$ being the standard normal distribution function. Note: The p-value can be seen as evidence against H_0. A small p-value represents strong evidence against H_0 and the null hypothesis should be rejected if the p-value is below a significance threshold α.

Chapter 6

Thoughts on implementation

This shall be a short overview of my thoughts regarding the implementation:

1. I wanted to produce reproducible results, i.e. my code shall be usable on a larger scope. Logic and Settings shall be separated. Completed code shall be script based and after run, create a folder with its running configuration.

2. Be careful, when storing list or dataframes, as a deepcopy might have to be used.

3.

Chapter 7

Interesting findings

7.1 DP offer nothing at start

In the single leg flight scenario with no purchase preference of 1, we found that it makes no difference (due to rounding error), whether just the most expensive product is offered or no product at all. Thus, we want to quickly note this result down here.

We want to estimate the actions in time $t = 1$. After having the final results computed for all capacities in $t = 2$, we arrive at the following values:

Furthermore, we have the following offersets:

resulting in the purchase probabilities:

This results in the following expected values,

[**1000.**815.38461538890.90909091810.52631579955.55555556835.29411765893.33333333826.0869565**100**

i.e. offering nothing and offering just the most expensive product both result in an expected value of 1000.

As it almost never makes sense to offer no product, we switched the offer set with no products to offer at the very end. The python function `numpy.amax` returns the index of the first maximum (if there are more than one).

presentation_of_code

July 16, 2019

```
In [1]: from B_helper import *
        import inspect
```

1 get_offer_sets_all

1.1 Documentation

```
In [7]: print(inspect.getdoc(get_offer_sets_all))
```

```
Generates all possible offer sets, starting with offering nothing.

:param products: array of all products that can be offered.
:return: two dimensional array with rows containing all possible offer sets, starting with offering no product.
```

1.2 Source Code

```
In [8]: print(inspect.getsource(get_offer_sets_all))
```

```
def get_offer_sets_all(products):
    """
    Generates all possible offer sets, starting with offering nothing.

    :param products: array of all products that can be offered.
    :return: two dimensional array with rows containing all possible offer sets, starting with offering no product.
    """
    n = len(products)
    offer_sets_all = np.array(list(map(list, itertools.product([0, 1], repeat=n))))
    offer_sets_all = np.vstack((offer_sets_all, offer_sets_all[0]))[1:]  # move empty offer set to the end
    return offer_sets_all
```

1.3 Example

```
In [30]: products = np.arange(3)
         print("Products: \t", products)
         print("\nResult of get_offer_sets_all: \n", get_offer_sets_all(products))
```

```
Products:           [0 1 2]

Result of get_offer_sets_all:
 [[0 0 1]
 [0 1 0]
 [0 1 1]
 [1 0 0]
 [1 0 1]
```

```
[1 1 0]
[1 1 1]
[0 0 0]]
```

2 customer_choice_individual

2.1 Documentation

```
In [ ]: print(inspect.getdoc(customer_choice_individual))
```

2.2 Source Code

```
In [ ]: print(inspect.getsource(customer_choice_individual))
```

2.3 Example

```
In [ ]: offer_set_tuple = tuple(np.ones(3, dtype=np.int))
        preference_weights = np.array([1.0, 1.0, 2.0])
        preference_no_purchase = np.array(1.0)
        print("Offer set tuple: \t\t", offer_set_tuple)
        print("Preference Weights: \t\t", preference_weights)
        print("Preference no purchase: \t", preference_no_purchase)
        print("\nResult of customer_choice_individual: \n", customer_choice_individual(offer_set_tuple, preference_
```

3 customer_choice_vector

3.1 Documentation

```
In [52]: print(inspect.getdoc(customer_choice_vector))
```

```
From perspective of retailer: With which probability can he expect to sell each product (respectively non-purchase)

:param offer_set_tuple: tuple with offered products indicated by 1=product offered
:param preference_weights: preference weights of all customers
:param preference_no_purchase: preference for no purchase for all customers
:param arrival_probabilities: vector with arrival probabilities of all customer segments
:return: array with probabilities of purchase ending with no purchase

NOTE: probabilities don't have to sum up to one? BEACHTE: Unterschied zu (1) in Bront et all
```

3.2 Source Code

```
In [36]: print(inspect.getsource(customer_choice_vector))
```

```
def customer_choice_vector(offer_set_tuple, preference_weights, preference_no_purchase, arrival_probabilities):
    """
    From perspective of retailer: With which probability can he expect to sell each product (respectively non-purch

    :param offer_set_tuple: tuple with offered products indicated by 1=product offered
    :param preference_weights: preference weights of all customers
    :param preference_no_purchase: preference for no purchase for all customers
    :param arrival_probabilities: vector with arrival probabilities of all customer segments
    :return: array with probabilities of purchase for each product ending with no purchase

    NOTE: probabilities don't have to sum up to one? BEACHTE: Unterschied zu (1) in Bront et all
    """
```

```
    if sum(arrival_probabilities) > 1:
        raise ValueError("The sum of all arrival probabilities has to be <= 1.")

    probs = np.zeros(len(offer_set_tuple) + 1)
    for l in np.arange(len(preference_weights)):
        probs += arrival_probabilities[l] * customer_choice_individual(offer_set_tuple, preference_weights[l, :],
                                                                       preference_no_purchase[l])
    return probs
```

3.3 Example

```
In [41]: offer_set_tuple = tuple(np.ones(3, dtype=np.int))
         preference_weights = np.array([[1.0, 1.0, 2.0],
                                        [1.0, 2.0, 1.0],
                                         [2.0, 1.0, 1.0]])
         preference_no_purchase = np.array([1.0, 1.0, 1.0])
         arrival_probabilities = np.ones(3)/3
         print("Offer set tuple: \t\t", offer_set_tuple)
         print("Preference Weights: \n", preference_weights)
         print("Preference no purchase: \t", preference_no_purchase)
         print("Arrival probabilities: \t\t", arrival_probabilities)
         erg = customer_choice_vector(offer_set_tuple, preference_weights, preference_no_purchase, arrival_probabil
         print("\nResult of customer_choice_vector: \n", erg)
         print("Sum: ", sum(erg))
```

```
Offer set tuple:                  (1, 1, 1)
Preference Weights:
 [[1. 1. 2.]
 [1. 2. 1.]
 [2. 1. 1.]]
Preference no purchase:          [1. 1. 1.]
Arrival probabilities:                  [0.33333333 0.33333333 0.33333333]

Result of customer_choice_vector:
 [0.26666667 0.26666667 0.26666667 0.2       ]
Sum:  1.0
```

4 simulate_sales

4.1 Documentation

```
In [2]: print(inspect.getdoc(simulate_sales))
```

```
Simulates a sales event given two random numbers (customer, sale) and a offerset. Eventually, no customer arrives.
This would be the case if the random number random_customer > sum(arrival_probabilities).

:param offer_set: Products offered
:param random_customer: Determines the arriving customer (segment).
:param random_sales: Determines the product purchased by this customer.
:param arrival_probabilities: The arrival probabilities for each customer segment
:param preference_weights: The preference weights for each customer segment (for each product)
:param preferences_no_purchase: The no purchase preferences for each customer segment.
:return: The product that has been purchased. No purchase = len(products) = n   (products indexed from 0 to n-1)
```

4.2 Source Code

In [3]:
```
print(inspect.getsource(simulate_sales))
```

```
def simulate_sales(offer_set, random_customer, random_sales, arrival_probabilities, preference_weights, preferences
    """
    Simulates a sales event given two random numbers (customer, sale) and a offerset. Eventually, no customer arriv
    This would be the case if the random number random_customer > sum(arrival_probabilities).

    :param offer_set: Products offered
    :param random_customer: Determines the arriving customer (segment).
    :param random_sales: Determines the product purchased by this customer.
    :param arrival_probabilities: The arrival probabilities for each customer segment
    :param preference_weights: The preference weights for each customer segment (for each product)
    :param preferences_no_purchase: The no purchase preferences for each customer segment.
    :return: The product that has been purchased. No purchase = len(products) = n   (products indexed from 0 to n-1
    """
    customer = random_customer <= np.array([*np.cumsum(arrival_probabilities), 1.0])
    customer = min(np.array(range(0, len(customer)))[customer])

    if customer == len(arrival_probabilities):
        return len(preference_weights[0])  # no customer arrives => no product sold (product out of range)
    else:
        product = random_sales <= np.cumsum(customer_choice_individual(offer_set,
                                                                       preference_weights[customer],
                                                                       preferences_no_purchase[customer]))
        product = min(np.arange(len(preference_weights[0]) + 1)[product])
        return product
```

4.3 Example

In [26]:
```
offer_set_tuple = tuple(np.ones(3, dtype=np.int))
preference_weights = np.array([[1.0, 1.0, 2.0],
                               [1.0, 2.0, 1.0],
                                [2.0, 1.0, 1.0]])
preference_no_purchase = np.array([1.0, 1.0, 1.0])
arrival_probabilities = np.ones(3)/4

print("Offer set tuple: \t\t", offer_set_tuple)
print("Preference Weights: \n", preference_weights)
print("Preference no purchase: \t", preference_no_purchase)
print("Arrival probabilities: \t\t", arrival_probabilities)

print("\n   Customer preferences of customer 0: \t", customer_choice_individual(offer_set_tuple, preferenc
print("   Customer preferences of customer 1: \t", customer_choice_individual(offer_set_tuple, preference_

random_customer = 0.1
random_sales = 0.1
print("\n\n##########\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.1
random_sales = 0.3
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
```

```
print("Result of simulate_sales: \n", erg)

random_customer = 0.1
random_sales = 0.5
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.1
random_sales = 0.9
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.4
random_sales = 0.1
print("\n\n##########\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.4
random_sales = 0.3
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.4
random_sales = 0.5
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.4
random_sales = 0.9
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.8
random_sales = 0.1
print("\n\n##########\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.8
random_sales = 0.3
print("\n\nRandom customer: \t", random_customer)
print("Random sales: \t", random_sales)
erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
print("Result of simulate_sales: \n", erg)

random_customer = 0.8
```

```
        random_sales = 0.5
        print("\n\nRandom customer: \t", random_customer)
        print("Random sales: \t", random_sales)
        erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
        print("Result of simulate_sales: \n", erg)

        random_customer = 0.8
        random_sales = 0.9
        print("\n\nRandom customer: \t", random_customer)
        print("Random sales: \t", random_sales)
        erg = simulate_sales(offer_set_tuple, random_customer, random_sales, arrival_probabilities, preference_wei
        print("Result of simulate_sales: \n", erg)
```

```
Offer set tuple:                 (1, 1, 1)
Preference Weights:
 [[1. 1. 2.]
 [1. 2. 1.]
 [2. 1. 1.]]
Preference no purchase:          [1. 1. 1.]
Arrival probabilities:                  [0.25 0.25 0.25]

   Customer preferences of customer 0:          [0.2 0.2 0.4 0.2]
   Customer preferences of customer 1:          [0.2 0.4 0.2 0.2]


#########
Random customer:          0.1
Random sales:          0.1
Result of simulate_sales:
 0


Random customer:          0.1
Random sales:          0.3
Result of simulate_sales:
 1


Random customer:          0.1
Random sales:          0.5
Result of simulate_sales:
 2


Random customer:          0.1
Random sales:          0.9
Result of simulate_sales:
 3


#########
Random customer:          0.4
Random sales:          0.1
Result of simulate_sales:
 0


Random customer:          0.4
Random sales:          0.3
```

```
Result of simulate_sales:
 1


Random customer:          0.4
Random sales:          0.5
Result of simulate_sales:
 1


Random customer:          0.4
Random sales:          0.9
Result of simulate_sales:
 3


#########
Random customer:          0.8
Random sales:          0.1
Result of simulate_sales:
 3


Random customer:          0.8
Random sales:          0.3
Result of simulate_sales:
 3


Random customer:          0.8
Random sales:          0.5
Result of simulate_sales:
 3


Random customer:          0.8
Random sales:          0.9
Result of simulate_sales:
 3
```

5 calc_value_marginal

5.1 Documentation

```
In [2]: print(inspect.getdoc(calc_value_marginal))
```

```
Calculates the marginal value as indicated at Bront et al, 4.2.2 Greedy Heuristic -> step 4a

:param indices_inner_sum: C_l intersected with (S union with {j})
:param pi: vector of dual prices for each resource (np.inf := no capacity)
:param revenues: vector of revenue for each product
:param A: matrix with resource consumption of each product (one row = one resource)
:param arrival_probabilities: The arrival probabilities for each customer segment
:param preference_weights: The preference weights for each customer segment (for each product)
:param preferences_no_purchase: The no purchase preferences for each customer segment.
:return: The value inside the argmax (expected marginal value given one set of products to offer)
```

5.2 Source Code

In [2]:
```
print(inspect.getsource(calc_value_marginal))
```

```
def calc_value_marginal(indices_inner_sum, pi, revenues, A, arrival_probabilities, preference_weights, preferences_
    """
    Calculates the marginal value as indicated at Bront et al, 4.2.2 Greedy Heuristic -> step 4a

    :param indices_inner_sum: C_l intersected with (S union with {j})
    :param pi: vector of dual prices for each resource (np.inf := no capacity)
    :param revenues: vector of revenue for each product
    :param A: matrix with resource consumption of each product (one row = one resource)
    :param arrival_probabilities: The arrival probabilities for each customer segment
    :param preference_weights: The preference weights for each customer segment (for each product)
    :param preferences_no_purchase: The no purchase preferences for each customer segment.
    :return: The value inside the argmax (expected marginal value given one set of products to offer)
    """
    v_temp = 0
    for l in np.arange(len(preference_weights)):  # sum over all customer segments
        v_temp += arrival_probabilities[l] * \
                  sum(indices_inner_sum * (revenues - np.apply_along_axis(sum, 1, A.T * pi)) *
                      preference_weights[l, :]) / \
                  (sum(indices_inner_sum * preference_weights[l, :]) + preferences_no_purchase[l])
    return v_temp
```

5.3 Example

In [16]:
```
print("Example analogously to Bront et al\n\n")
revenues = np.array([100, 19, 19])
pi = np.array([0])
A = np.array([[1, 1, 1]])
arrival_probabilities = np.array([1, 1, 1])
preference_weights = np.array([[1,1,1],
                               [0,1,0],
                               [0,0,1]])
preferences_no_purchase = np.array([1,1,1])

print("Revenues: \t\t\t", revenues)
print("Pi: \t\t\t\t", pi)
print("A:\t\t\t\t", A)
print("Preference Weights: \n", preference_weights)
print("Preference no purchase: \t", preferences_no_purchase)
print("Arrival probabilities: \t\t", arrival_probabilities)

indices_inner_sum = np.array([1, 0, 0])
print("\n#####\nIndices inner sum: \t", indices_inner_sum)
erg = calc_value_marginal(indices_inner_sum, pi, revenues, A, arrival_probabilities, preference_weights, p
print("Result of simulate_sales: \n", erg)

indices_inner_sum = np.array([1, 1, 1])
print("\n#####\nIndices inner sum: \t", indices_inner_sum)
erg = calc_value_marginal(indices_inner_sum, pi, revenues, A, arrival_probabilities, preference_weights, p
print("Result of simulate_sales: \n", erg)
```

```
Example analogously to Bront et al


Revenues:                       [100  19  19]
```

```
Pi:                                  [0]
A:                                [[1 1 1]]
Preference Weights:
 [[1 1 1]
 [0 1 0]
 [0 0 1]]
Preference no purchase:          [1 1 1]
Arrival probabilities:                 [1 1 1]

#####
Indices inner sum:          [1 0 0]
Result of simulate_sales:
 50.0

#####
Indices inner sum:          [1 1 1]
Result of simulate_sales:
 53.5
```

6 determine_offer_tuple

6.1 Documentation

```
In [17]: print(inspect.getdoc(determine_offer_tuple))
```

```
Determines the offerset given the bid prices for each resource.

Implement the Greedy Heuristic from Bront et al: A Column Generation Algorithm ... 4.2.2
and extend it for the epsilon greedy strategy
:param pi: vector of dual prices for each resource (np.inf := no capacity)
:param eps: epsilon value for epsilon greedy strategy (eps = 0 := no greedy strategy to apply)
:param revenues: vector of revenue for each product
:param A: matrix with resource consumption of each product (one row = one resource)
:param arrival_probabilities: The arrival probabilities for each customer segment
:param preference_weights: The preference weights for each customer segment (for each product)
:param preferences_no_purchase: The no purchase preferences for each customer segment.
:return: the offer set to be offered
```

6.2 Source Code

```
In [24]: print(inspect.getsource(determine_offer_tuple))
```

```
def determine_offer_tuple(pi, eps, revenues, A, arrival_probabilities, preference_weights, preferences_no_purchase)
    """
    Determines the offerset given the bid prices for each resource.

    Implement the Greedy Heuristic from Bront et al: A Column Generation Algorithm ... 4.2.2
    and extend it for the epsilon greedy strategy
    :param pi: vector of dual prices for each resource (np.inf := no capacity)
    :param eps: epsilon value for epsilon greedy strategy (eps = 0 := no greedy strategy to apply)
    :param revenues: vector of revenue for each product
    :param A: matrix with resource consumption of each product (one row = one resource)
    :param arrival_probabilities: The arrival probabilities for each customer segment
    :param preference_weights: The preference weights for each customer segment (for each product)
    :param preferences_no_purchase: The no purchase preferences for each customer segment.
    :return: the offer set to be offered
    """
```

```
# no resources left => nothing to be sold
if all(pi == np.inf):
    return tuple(np.zeros_like(revenues))

# epsilon greedy strategy - offer no products
eps_prob = random.random()
if eps_prob < eps/2:
    return tuple(np.zeros_like(revenues))

# epsilon greedy strategy - offer all products
if eps_prob < eps:
    offer_tuple = np.ones_like(revenues)
    offer_tuple[np.sum(A[[pi == np.inf], :], axis=0) > 0] = 0  # one resource not available => don't offer prod
    return tuple(offer_tuple)

# setup
offer_tuple = np.zeros_like(revenues)

# line 1
s_prime = revenues - np.apply_along_axis(sum, 1, A.T * pi) > 0
if all(np.invert(s_prime)):
    return tuple(offer_tuple)

# line 2-3
# offer_sets_to_test has in each row an offer set, we want to test
offer_sets_to_test = np.zeros((sum(s_prime), len(revenues)))
offer_sets_to_test[np.arange(sum(s_prime)), np.where(s_prime)] = 1
offer_sets_to_test += offer_tuple
offer_sets_to_test = (offer_sets_to_test > 0)

value_marginal = np.apply_along_axis(calc_value_marginal, 1, offer_sets_to_test, pi, revenues,
                                     preference_weights, arrival_probabilities, A, preferences_no_purchase)

offer_tuple[np.argmax(value_marginal)] = 1
s_prime = s_prime & offer_tuple == 0
v_s = np.amax(value_marginal)

# line 4
while True:
    # 4a
    # offer_sets_to_test has in each row an offer set, we want to test
    offer_sets_to_test = np.zeros((sum(s_prime), len(revenues)))
    offer_sets_to_test[np.arange(sum(s_prime)), np.where(s_prime)] = 1
    offer_sets_to_test += offer_tuple
    offer_sets_to_test = (offer_sets_to_test > 0)

    # 4b
    value_marginal = np.apply_along_axis(calc_value_marginal, 1, offer_sets_to_test, pi, revenues,
                                         preference_weights, arrival_probabilities, A, preferences_no_purchase)

    if np.amax(value_marginal) > v_s:
        v_s = np.amax(value_marginal)
        offer_tuple[np.argmax(value_marginal)] = 1
        s_prime = s_prime & offer_tuple == 0
        if all(offer_tuple == 1):
            break
    else:
        break
```

```
    return tuple(offer_tuple)
```

6.3 Example

```
In [23]: print("Example analogously to Bront et al\n\n")
         revenues = np.array([100, 19, 19])
         pi = np.array([0])
         A = np.array([[1, 1, 1]])
         arrival_probabilities = np.array([1, 1, 1])
         preference_weights = np.array([[1,1,1],
                                        [0,1,0],
                                        [0,0,1]])
         preferences_no_purchase = np.array([1,1,1])
         eps = 0

         print("Revenues: \t\t\t", revenues)
         print("Pi: \t\t\t\t", pi)
         print("A:\t\t\t\t", A)
         print("Preference Weights: \n", preference_weights)
         print("Preference no purchase: \t", preferences_no_purchase)
         print("Arrival probabilities: \t\t", arrival_probabilities)
         print("Epsilon: \t\t\t", eps)

         erg = determine_offer_tuple(pi, eps, revenues, A, arrival_probabilities, preference_weights, preferences_n
         print("\nResult of determine_offer_tuple: \n", erg)
```

```
Example analogously to Bront et al


Revenues:                           [100  19  19]
Pi:                                   [0]
A:                                  [[1 1 1]]
Preference Weights:
 [[1 1 1]
 [0 1 0]
 [0 0 1]]
Preference no purchase:           [1 1 1]
Arrival probabilities:                [1 1 1]
Epsilon:                           0

Result of determine_offer_tuple:
 (1, 0, 0)
```

	value	offer_set_optimal	num_offer_set_optimal
0	0.0	0	0
1	1000.0	0	2
2	2000.0	0	2
3	3000.0	0	2
4	4000.0	0	2
5	5000.0	0	2
6	6000.0	0	2
7	7000.0	0	2
8	8000.0	0	2
9	9000.0	0	2
10	10000.0	0	2
11	11000.0	0	2
12	12000.0	0	2
13	13000.0	8	1
14	14000.0	8	1
15	15000.0	8	1
16	16000.0	8	1
17	17000.0	8	1
18	18000.0	8	1
19	19000.0	8	1
20	20000.0	8	1

	0	1	2	3
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

	0	1	2	3	4
0	0.000000	0.000000	0.000000	0.000000	1.000000
1	0.000000	0.000000	0.000000	0.307692	0.692308
2	0.000000	0.000000	0.272727	0.000000	0.727273
3	0.000000	0.000000	0.157895	0.210526	0.631579
4	0.000000	0.222222	0.000000	0.000000	0.777778
5	0.000000	0.117647	0.000000	0.235294	0.647059
6	0.000000	0.133333	0.200000	0.000000	0.666667
7	0.000000	0.086957	0.130435	0.173913	0.608696
8	0.142857	0.000000	0.000000	0.000000	0.857143
9	0.066667	0.000000	0.000000	0.266667	0.666667
10	0.076923	0.000000	0.230769	0.000000	0.692308
11	0.047619	0.000000	0.142857	0.190476	0.619048
12	0.090909	0.181818	0.000000	0.000000	0.727273
13	0.052632	0.105263	0.000000	0.210526	0.631579
14	0.058824	0.117647	0.176471	0.000000	0.647059
15	0.040000	0.080000	0.120000	0.160000	0.600000

Bibliography

Bamberg, G., Baur, F., and Krapp, M. (2011). *Statistik*. Lehr- und Handbücher der Wirtschafts- und Sozialwissenschaften. Oldenbourg, München, 16., überarb. aufl. edition.

Bront, J. J. M., Méndez-Díaz, I., and Vulcano, G. (2009). A column generation algorithm for choice-based network revenue management. *Operations Research*, 57(3):769–784.

Fahrmeir, L., Künstler, R., Pigeot, I., and Tutz, G. (2007). *Statistik: Der Weg zur Datenanalyse*. Springer-Lehrbuch. Springer, Berlin, 6., überarb. aufl. edition.

Koch, S. (2017). Least squares approximate policy iteration for learning bid prices in choice-based revenue management. *Computers & Operations Research*, 77:240–253.

Sutton, R. S. and Barto, A. (2018). *Reinforcement learning: An introduction*. Adaptive computation and machine learning. The MIT Press, Cambridge, MA and London, second edition edition.

Nomenclature

API approximate policy iteration

CDLP choice-based linear program

DP dynamic programming

MDP markov decision process

MNL multinomial logit model