Graphical Models Lab Report

Structure Learning in Linear Gaussian Bayesian Networks

Farin Lippmann - 190538

Due 21.03.2023

1 Problem Setting

For this report, we were asked to fit a linear gaussian bayesian network model to a set of data. The model we design should have a good fit on the data with as few parameters as possible. We want to find a model that maximizes the likelihood while minimizing the number of parameters. Additionally, we find the model that allows the best predictions of one of the variables in the dataset.
The data are measurements of $n = 12$ attributes of $m = 1199$ different wines. Thus, each datapoint $x^{(j)}$ for $j = \{1, .., 1199\}$ is a vector with entries $x_i^{(j)}$ for $i = \{1, .., 12\}$.
The dataset $X \in \mathbb{R}^{1199 \times 12}$ is complete, with no missing values. All attributes except *quality* are real-valued. *quality* lies on an integral ordinal scale, but will also be interpreted as real valued in our model, to better fit into the gaussian architecture. *quality* will also be the attribute that we find the best possible predictor for.

Attribute	Range
fixed acidity	[4.7, 15.9]
volatile acidity	[0.12, 1.58]
citric acid content	[0, 1]
residual sugar	[0.9, 15.5]
chlorides	[0.012, 0.611]
free sulfur dioxide	[1, 72]
total sulfur dioxide	[6, 289]
density	[0.99007, 1.00369]
pH	[2.74, 4.01]
sulphates	[0.37, 2.00]
alcohol	[8.4, 14.9]
quality	$\{3, 4, 5, 6, 7, 8\}$

Table 1: The attributes present in the data along with the range of values they take.

2 Linear Gaussian Bayesian Networks

2.1 Definition

Gaussian bayesian networks are a type of generative probabilistic model. They specify a full distribution over the data , $p(x)$, in the form of a normal distribution. Internally, this normal distribution is a product of more normal distributions, one for each attribute, represented by the

random variable x_i. These individual distributions are each conditional on some other attributes. Which of the other attributes one attribute is dependent on is specified by a graph, making gaussian bayesian networks graphical models.
Each attribute is represented by a node in a directed acyclic graph $G = (V, E)$ with nodes $V = \{1, .., n\}$ and edges E. The edges in said graph signal dependencies between variables. If $(k, l) \in E$, then attribute l is dependent on attribute k. Thus, we can write down the probability density function of the distributions specified by a gaussian bayesian network as follows:

$$p(x) = \prod_{j=1}^{n} p(x_i|\mathbf{pa}(i))$$

where $\mathbf{pa}(i)$ returns the random variables that correspond to the nodes in G that have edges going to the node i, also called the parents of x_i.
In linear gaussian bayesian networks, the dependency of x_i on its parents is given by a linear releationship. The mean of x_i is assumed to be a linear combination of the values of $\mathbf{pa}(i)$. This lets us define the individual distributions for each of the attributes:

$$p(x_i|\mathbf{pa}(i)) = \mathcal{N}\left(\beta_{i0} + \sum_{j \in \mathbf{pa}(i)} \beta_{ij} x_j, \sigma_i\right)$$

with parameters:

- β_{ij}: coefficients describing the way x_j influences x_i
- β_{i0}: offset
- σ_i: variance of x_i after conditioning on $\mathbf{pa}(i)$ [1]

2.2 Fitting

When the graph specifying the dependencies is given, we can find a unique linear gaussian bayesian network (LGBN) that maximizes the likelihood $L(X) = \prod_{j=1}^{m} p(x^{(j)})$. For each attribute x_i, we fit its conditional probability distribution $p(x_i|\mathbf{pa}(i))$ by finding values for β_{ij}, β_{i0} and σ_i (for j as indices of the parents of x_i). The β-parameters can be found by fitting a linear regression on x_i, using $\mathbf{pa}(i)$ as predictors. σ_i is the conditional variance of x_i, or the variance that is left after conditioning on $\mathbf{pa}(i)$. The formula for the conditional variance is [2]:

$$\mathrm{Var}[Y|X] = E\left[(Y - E[Y|X])^2\right]$$

And we can estimate the conditional variance using the mean squared error of predicting one variable using the linear regression:

$$\mathrm{sampleVar}(x_i|\mathbf{pa}(i)) = \frac{1}{m-1} \sum_{i=1}^{m} (x_i^{(j)} - \hat{x}_i^{(j)})^2$$

where:

- $x_i^{(j)}$ is the true value of this attribute for datapoint j
- $\hat{x}_i^{(j)}$ value of x_i for datapoint j as predicted by the linear regression given the values of the parents

In conclusion, we fit a gaussian bayesian network in the following way.

For each attribute x_i, do the following:
If x_i has at least one parent in the dependency graph, we fit a linear regression on x_i with $\mathbf{pa}(i)$ as the predictor variables. We store the fitted parameters β_{i0} and β_{ij} (for parent-indices j). We also store $\sigma_i = \text{sampleVar}(x_i|\mathbf{pa}(i))$.
If x_i has no parents, we can parameterize it more simply, by computing the static sample mean $\mu_i = \sum_{j=1}^{m} x_i^{(j)}$ and sample variance $\sigma_i = \frac{1}{m-1}\sum_{i=1}^{m}(x_i^{(j)} - \mu_i)^2$.

2.3 Computing Probabilities

A fit network defines a probability distribution over the data. We can use it to compute the probability density of datapoints using the definition given in section 2.1, as a product over the individual attributes:

$$p(x) = \prod_{j=1}^{n} p(x_i|\mathbf{pa}(i))$$

If x_i has no parents, $p(x_i|\mathbf{pa}(i))$ simplifies to $\mathcal{N}(\mu_i, \sigma_i)$.
If x_i has at least one parent, we compute $\mathcal{N}\left(\beta_{i0} + \sum_{j\in\mathbf{pa}(i)} \beta_{ij}x_j, \sigma_i\right)$.
With the ability to compute the probability density of one datapoint, we are also able to compute the likelihood of a whole dataset X, given a dependency graph represented by $\mathbf{pa}$:

$$L(X, \mathbf{pa}) = p(X|\mathbf{pa}) = \prod_{j=1}^{m}\prod_{i=1}^{n} p(x_i^{(j)}|\mathbf{pa}(x_i)^{(j)})$$

3 Structure Learning

Having established how an LGBN can be uniquely fit once a dependency graph is given, we now explore the core of this report, how a suitable dependency graph can be found.

3.1 Prediction

For our first assignment, we find the dependency graph that allows the LGBN to most accurately predict the *quality* of a wine from its other attributes. Because of the structure of LGBNs, only the direct parents of *quality* have an influence on the prediction. This means we can ignore all possible edges that do not lead to *quality*, leaving us with only 11 edges to consider, one for each other attribute. That makes $2^{11} = 2048$ possible dependency graphs, which we can simply iterate through and compute the likelihood for.
Further details on the best dependency graph and how it was chosen can be found in the results section.

3.2 Scores

Now we focus on the second assignment, finding a dependency graph that maximizes the overall likelihood of the data while minimizing the number of parameters. We employ score-based search algorithms, which require a metric that determines how fit a dependency graph is. We implement two very similar scores, the Bayesian information criterion (BIC) and the Akaike information criterion (AIC). Both of these take lower values for better fitting models. They reward higher likelihoods and penalize higher numbers of parameters, although the Bayesian information criterion takes the number of datapoints into account as well. They are derived using slightly different statistical assumptions, but function very similarly.

$$\text{BIC}(k, m, L) := \ln(m)k - \ln(L) \qquad \text{AIC}(k, L) := 2k - \ln(L)$$

where

- k is the number of parameters
- m is the number of datapoints
- L is the likelihood with the given dependency graph

For datasets with 8 or more datapoints, the BIC will be more conservative than the AIC, as it puts more weight on the number of parameters.

3.3 Legal Edges

To implement any of the algorithms we present, one has to determine wether an edge of the dependency graph is legal or not. Since dependency graphs are directed acyclic graphs, an edge is legal if it does not cause a cycle to be created in the graph.
Cycles are paths in the graph that start and end with the same node. For an edge (a, b) to complete a cycle in a graph, there needs to already have been a path from b to a, because only then would a path from a to a exists after adding the edge (a, b).
Thus, to check if an edge (a, b) is legal, we need only check that there is no path from b to a. We implement this using a depth-first-search starting in b that terminates when it reaches a or has explored all nodes it can reach from b.
With this subroutine, we can define our search algorithms.

3.4 Local Search

In local additive search, we start with an empty graph and in each step add the edge to the graph that brings the highest increase to the score metric. Once no more score increase can be achieved, the algorithm terminates.

Algorithm 1 Local Additive Search (*data*, metric, n)

```
A ← graph with n nodes and no edges
ŝ ← score of the LGBN fit with no dependencies          ▷ best score found so far
while found a score-increasing edge in last iteration do
    for each possible legal edge e do
        add e to A
        s ← score of LGBN fit with A
        if s < ŝ then                                    ▷ found a better edge
            ŝ ← s
        end if
        remove e from A
    end for
    if found a score increasing edge during for loop then
        add that edge to A
    end if
end while
return A
```

This algorithm is a greedy search algorithm and a form of hill climbing, always taking the best short-term step and never backtracking. This makes it highly susceptible to getting stuck in a local minimum of the score function. It is also restricted in that it can only add edges to the

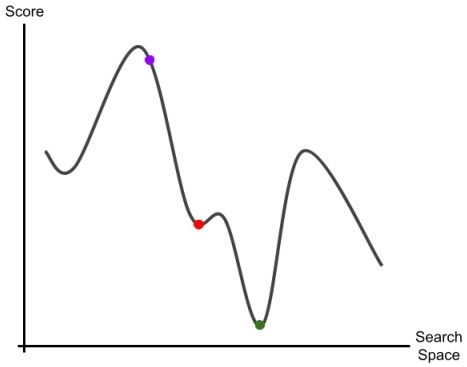

Figure 1: Visualization of a non-convex search space with multiple local minima. Where a local search starting at the purple point would find the local minimum in red, simulated annealing has the chance to make the jump across to the better minimum in green.

graph, never remove them. By also allowing the algorithm to see if removing any of the edges in the graph improves the score, we get a **local combined search**.
We address the issue of local minima using the next algorithm, simulated annealing.

3.5 Simulated Annealing

Simulated annealing is an extension of a local search, in which randomness is exploited to reduce the chance of getting stuck in a local minimum.
The basic idea of simulated annealing is to sometimes allow actions that do not immediately improve the score metric, in hopes of moving out of the influence of a bad local minimum. A visualization of this can be seen in Figure 1.
To ensure that the algorithm still terminates in the peak of a local minimum, this chance to allow locally bad moves should be lowered over time. Simulated annealing uses a series of monotonically decreasing numbers, called temperature values, as an input parameter to control this chance.
In each step, a random legal move is proposed (we sample this move randomly from all removable edges and all legal addable edges). If this move improves the score metric, it is always accepted. If it does not, it is still accepted with a probability proportional to the temperature, and inversely proportional to how much this move worsens the score:

$$p(\text{accept}|\bar{s}, s, t_i) = \exp\left(-\frac{(\bar{s} - s)}{t_i}\right)$$

where:

- we assume the score metric is to be minimized
- s is the score before the proposed move is applied
- $\bar{s}$ is the score after the proposed move is applied
- t_i is the temperature value in this step[3]

Over all iterations, we keep track of the best solution found so far, and return it after all temperature values were used.

Algorithm 2 Simulated Annealing (*data*, *score*, n, t)

$A \leftarrow$ graph with n nodes and no edges
$s \leftarrow$ score of the LGBN fit with no dependencies ▷ current score
$\hat{s} \leftarrow$ s ▷ best score
$\hat{A} \leftarrow$ A ▷ best graph
for each t_i in t **do**
  $a \leftarrow$ random action (addable legal edge or removable edge)
  perform a on A
  $\bar{s} \leftarrow$ score of LGBN fit with A
  revert a on A
  if $\bar{s} < s$ **then** ▷ always accept if score improves
    perform a on A
    $s \leftarrow \bar{s}$
  else ▷ sometimes accept if score worsens
    perform a on A with probability $p(\text{accept}|\bar{s}, s, t_i)$
    $s \leftarrow \bar{s}$
  end if
  if $s < \hat{s}$ **then** ▷ update best graph found
    $\hat{s} \leftarrow s$
    $\hat{A} \leftarrow A$
  end if
end for
return $\hat{A}$

4 Results

4.1 Prediction

Using the exhaustive search described earlier, we can find the dependency graph for the problem that maximizes the accuracy when predicting *quality*. Since, however, the number of parameters should also be considered, we find the accuracy-maximizing graph for each number of edges in the dependency graph. We can take a look at the highest accuracy each number of parameters was able to achieve in Figure 2.

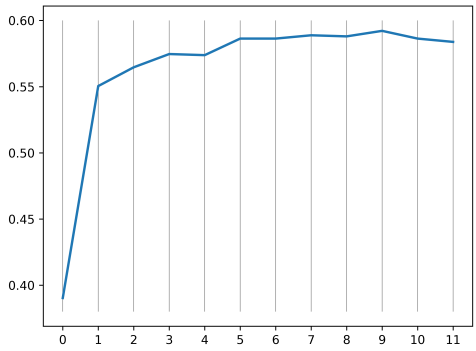

Figure 2: Progression of highest-achieved accuracies with rising numbers of dependencies for *quality*.

As one would expect, the accuracy increases steadily with the number of influencing attributes, until it however reaches a maximum at 9, then decreases for 10 and 11. The unusual decrease in accuracy for more parameters may be caused by the fact that the predicted value for *quality* gets rounded when calculating the accuracy. Interestingly, this gives us a natural choice for the best prediction model, the one with 9 influencing attributes. This model incorporates all other attributes except *density* and *pH* (and *quality*, of course), and is able to achieve an accuracy of 0.5922 on the training set.

4.2 Local Search

Using local additive search, we find one dependency graph with each score metric. The graph found with BIC achieves a log-likelihood of -5699.43 on the training set using 65 parameters, while the graph found using AIC achieves a log-likelihood of -5672.74 using 75 parameters. As expected, the BIC causes our algorithm to be more cautious and prefer fewer parameters to higher likelihood-values. Using a combined search that allows our algorithm to remove edges leads to the same optima.

4.3 Simulated Annealing

With explorational testing and trial and error, we find some general rules for the hyperparameters of simulated annealing. Using these general rules, we conduct a grid search over possible hyperparameter-values to find one with a good score.

4.3.1 Number of Parameters

We use the number of adjacency matrices that the local searches needed to look at to find the optimum as a baseline for the number of iterations in simulated annealing. In our case, this is about 4000.

4.3.2 Transforms

Because the higher temperatures are not as productive as the lower temperatures, since the algorithm mostly jumps around wildly, we transform the temperature values in different ways. From the results of some tests, visible in Figure 3, we notice the most success using linear transforms, with $t(x) = \frac{x}{4}$ achieving the best BIC-score.

4.3.3 Adding a Final Slide

Even though the transforms already put more weight on the lower temperature values, we found that the algorithm often did not find the very bottom of the local minimum it ended up in. To combat this, we add 300 0.001-values at the end of all temperature series, which almost turns the algorithm into a local search for the last part of its run.

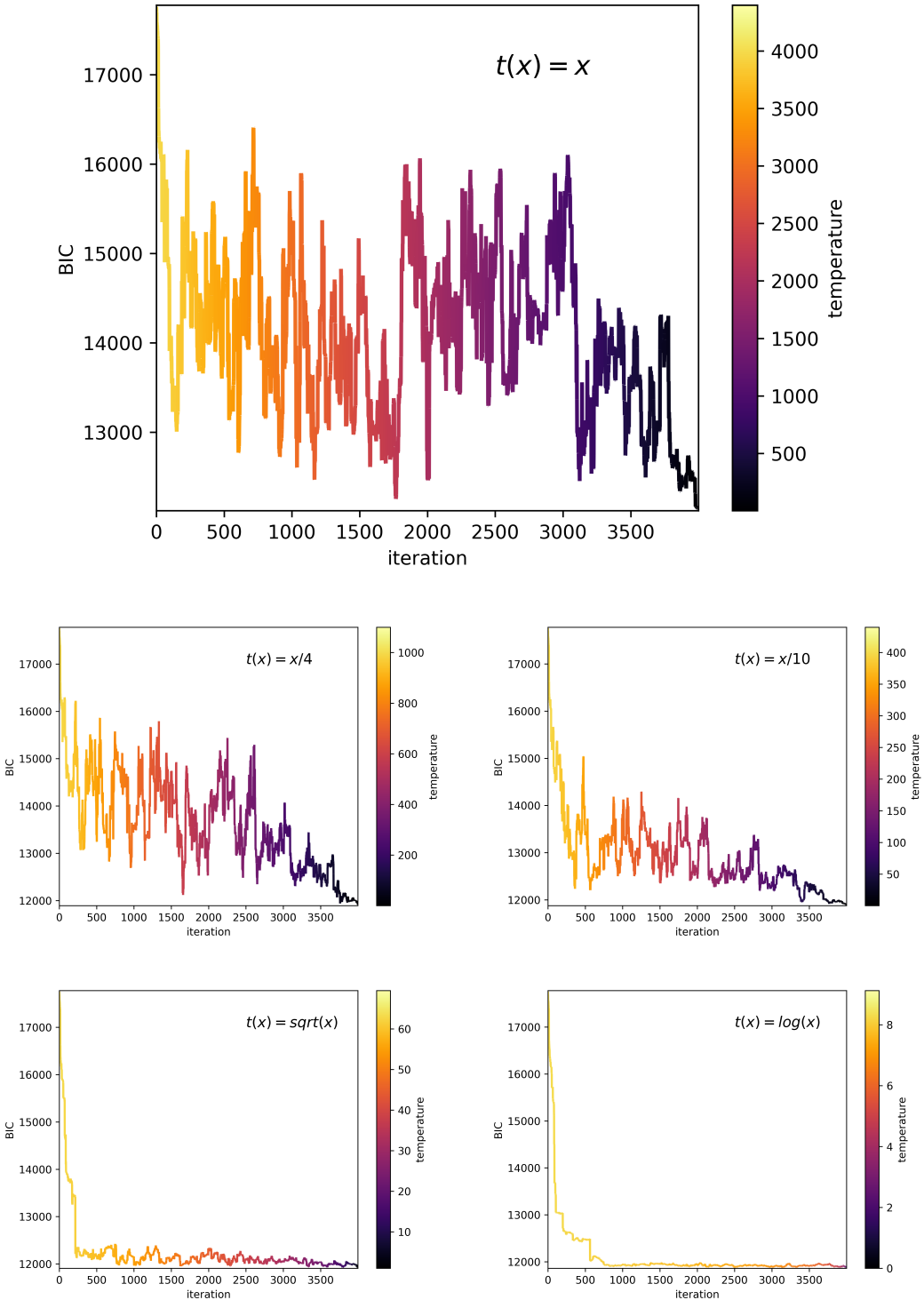

Figure 3: Runs of simulated annealing on our dataset using different temperature values.

4.3.4 Grid Search

To find the dependency graph that has the best general fit on the data, we conduct a grid search over possible temperature transformations for simulated annealing.
Each run of simulated annealing uses 4000 temperature values $\{4000, ..., 1\}$, with a different transformation applied to them in each run. We also add the final slide of 300 low-temperature values to each series of transformed temperatures. We focus mostly on linear transformations, but also include combinations of linear transformations with powers between 0.5 and 1. A full list of the used transformations is visible in Table 2.

$x/2$	$\log(x)$	$(x/3)^{0.7}$	$(x/6)^{0.9}$	$(x/12)^{0.5}$
$x/3$	$\sqrt{x}$	$(x/3)^{0.9}$	$(x/7)^{0.5}$	$(x/12)^{0.7}$
$x/4$	$x^{0.6}$	$(x/4)^{0.5}$	$(x/7)^{0.7}$	$(x/12)^{0.9}$
$x/5$	$x^{0.7}$	$(x/4)^{0.7}$	$(x/7)^{0.9}$	$(x/15)^{0.5}$
$x/6$	$x^{0.8}$	$(x/4)^{0.9}$	$(x/8)^{0.5}$	$(x/15)^{0.7}$
$x/7$	$x^{0.9}$	$(x/5)^{0.5}$	$(x/8)^{0.7}$	$(x/15)^{0.9}$
$x/8$	$(x/2)^{0.5}$	$(x/5)^{0.7}$	$(x/8)^{0.9}$	
$x/10$	$(x/2)^{0.7}$	$(x/5)^{0.9}$	$(x/10)^{0.5}$	
$x/12$	$(x/2)^{0.9}$	$(x/6)^{0.5}$	$(x/10)^{0.7}$	
$x/15$	$(x/3)^{0.5}$	$(x/6)^{0.7}$	$(x/10)^{0.9}$	

Table 2: All temperature transformations checked during grid search.

Each run of simulated annealing was done twice, one time with the BIC and one time with the AIC as score metric. The best BIC-score was achieved using the transform $(x/6)^{0.7}$, while the best AIC-score was achieved using $(x/5)^{0.7}$.
The best dependency graph found using BIC and AIC can be seen in Figure 4. The best BIC-graph achieves a log-likelihood of -5695.34 on the training set with 64 parameters, while the best AIC-graph achieves a log-likelihood of -5673.41 with 72 parameters.

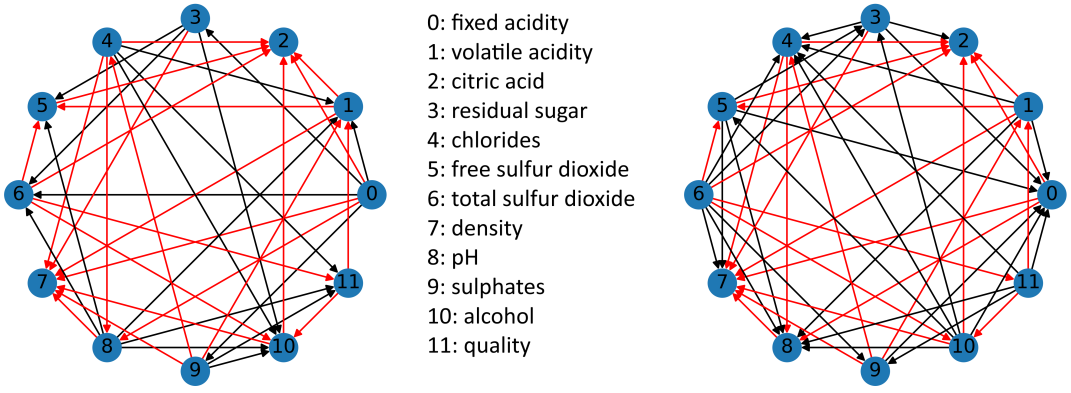

Figure 4: Best dependency graphs found using simulated annealing with BIC (left) and AIC (right). Red edges exist in both graphs.

Qualitatively, the two graphs look very different, with the only a few major trends being the same

between the two, mainly that *density* and *citric acid* are influenced by many other parameters. We can also analyze some quantitative measures:

- The graphs agree in existence and direction of 39 out of 66 edges (excluding self-edges).
- The graphs have 15 edges exactly flipped.
- The BIC-graph includes 2 edges that the AIC-graph does not.
- The AIC-graph includes 10 edges that the BIC-graph does not.

In all, because these graphs are so different, we would refrain from trying to make causal inferences from these findings.

5 Possible Improvements

The following work could be done to build on our findings:

- With help from experts (such as food scientists), one could do causal inference to find out the true relationships between attributes.
- A larger and finer grid could be used during hyperparameter-tuning to find even better models.
- The choice of score metric could be refined and tuned to the task at hand.
- A more sophisticated algorithm, like adaptive simulated annealing or simulated quenching could be used.

References

[1] David Heckerman. "A Tutorial on Learning With Bayesian Networks". In: *Learning in Graphical Models* (1996).

[2] *Conditional Variance*. `https://en.wikipedia.org/wiki/Conditional_variance`. visited: 09.02.2023, 11:47.

[3] *Simulated Annealing*. `https://de.wikipedia.org/wiki/Simulated_Annealing`. visited: 11.02.2023, 11:14.