Meta-Heuristic Driven Loss Ratio Optimization

Master Thesis Business Analytics

Published by:	P.A. de Jonge
For:	Business Analytics – Master Thesis
Date:	12 - 01 - 2015
Contact:	peter.de.jonge@friss.eu

Abstract

The material described in this thesis is the result of an internship of the master study Business Analytics. It has been conducted at FRISS from April 5th, 2015 until December 4th, 2015. FRISS is a consultancy and software company specialized in fraud- and risk management for the insurance sector and is located in Utrecht, the Netherlands. The main research problem described in this thesis is how to find high quality solutions to constrained Portfolio Value Optimization problems by using modern metaheuristic search procedures. A number of distinct search procedures were implemented and extensively tested. The experiments described in this work indicate that in practice high quality solutions can be found in short timespans and under a wide range of conditions.

FRISS

Contents

1 Introduction

The work described in this thesis takes place at FRISS. FRISS is a consultancy and software company specialized in fraud- and risk management for the insurance sector. FRISS focusses on three key areas: underwriting, claims and compliancy. In this thesis we will focus on the *underwriting* process. *Underwriting* refers to the process in which an insurance company assesses the risk to accept an individual as a client, i.e. a policy holder, into an existing *portfolio*, i.e. the total set of accepted policies.

By definition *insurance* refers to the transfer of a risk, from one entity to another, in exchange for money. From the insurer's perspective, accepting a client has two main effects. Firstly, the insurance company becomes liable for a predefined set of claims. Secondly, the insurance company receives systematic payments for the duration of the policy. The rejection of a policy application induces a loss of income for the insurer, however, it also frees them from having to pay potentially high claim amounts.

The *value* of the insurer's portfolio is related to the total premium income (P) taken over all policy holders in the portfolio and the total claim amount (S) that must be paid to the insured parties. The portfolio value is often taken as $W = P - S$ or as the *loss ratio* i.e. $L = S/P$.

A key question in this context is the following:

Can we systematically predict which clients will likely file claims after acceptance, based on an a priori known set of measurable features and can we use this information to increase the total value of a portfolio?"

In essence, we seek to find a *risk score*, based on a limited set of measurable features, that can be used to determine which clients to accept and which clients to reject.

A feature, also called *risk indicator*, here answers a particular question on a set of characteristics of a client e.g. *"Does the client have a history of bad credit debt?"*, *"Does the client have a history of previous claims?", "Is the client a young person?" etc.* In the context in which FRISS operates all these types of questions can be given a 'yes' or 'no' answer. If the answer to the question a feature relates to is 'yes' we say the client, from here on called a case, has a *hit* on the particular feature. The number of features used within a specific configuration varies between 100 and 500.

FRISS specializes in constructing risk scores that inform insurers on the risk of accepting policy applications. Typically, the higher the score, the higher the risk. Furthermore, FRISS provides the insurers a technical platform that, for a given case, can *automatically* gather all the necessary raw data from various internal and external data sources to measure the required features. Measuring features is done using decision tables, predictive models and profiles. Finally, the platform combines the features into a single risk score, known as the *FRISS score*. In this process, each feature is associated with a weight. For a given case, the FRISS score is the sum of all the feature weights for which a case has a hit.

The exact weight of a feature is determined in a *portfolio value analysis (PVA)*, which is the topic of the next section. Briefly, a PVA is a retrospective analysis on a historic portfolio in which all feature hits for all cases are known a priori, as well as the received premium and claim amounts.

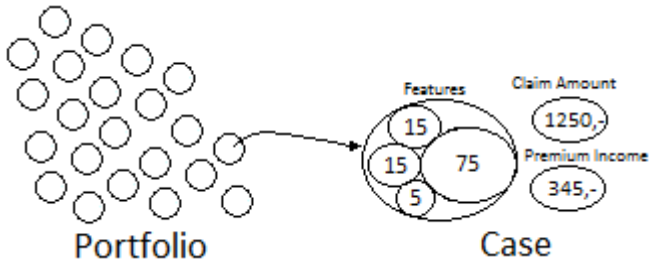


Figure 1: A portfolio with cases and the main components of an individual case. Each individual case is associated with a 3 attributes i.e. (i) a claim amount, (ii) a premium income and (iii) a set of feature measurements.

Figure 1 shows a toy example of an insurer's portfolio in which each individual case is depicted as a circle. Each of the cases is associated with a claim amount, a premium income and a set of feature hits with corresponding weights.

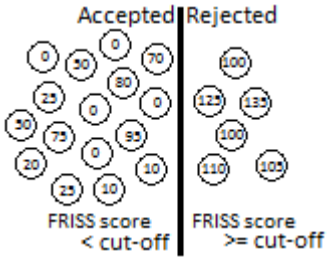


Figure 2: The cases divided into the accepted (A) and rejected (R) sets, based on the FRISS score and the cut-off. The cut-off value is taken as 100.

The FRISS score is used to divide the portfolio into cases with an acceptable risk and an unacceptable risk, based on a specific cut-off value. When the FRISS score is higher or equal the cut-off value, the case is rejected, else it is accepted into the portfolio. Figure 2 shows a possible division of the portfolio into an accepted and a rejected set.

Note that by changing the feature weights, we can change which cases end up in the accepted set, which in turn changes the accepted total premium income and the total claim amount. Ideally, we would like to optimize the feature weights in such a way that the value of the accepted set is maximized. However, in practice most insurers impose volume constraints on their portfolio by indicating that in a PVA at most $x\%$ of the cases can be rejected. With this in mind the research question of this thesis is:

"How to select an optimal weight vector that maximizes the value of the accepted set of cases, while still respecting the volume constraint?"

To find a solution to this combinatorial optimization problem, we propose to use meta-heuristic search algorithms. Meta-heuristic algorithms attempt to find feasible solutions for difficult optimization problems and problems with large search spaces. They combine a lower level search algorithm with a *meta search* strategy. A lower level search algorithm can easily get stuck into a specific part of the search space, being a local optimum, see Figure 3, or a large area with an almost equal objective value. A meta strategy is a higher level search strategy that helps the search algorithm to escape such local optima, or to lead the search algorithm to more fruitful areas in the solution space. Prime examples of meta-heuristic algorithms are Tabu Search [12] and Simulated Annealing [20].

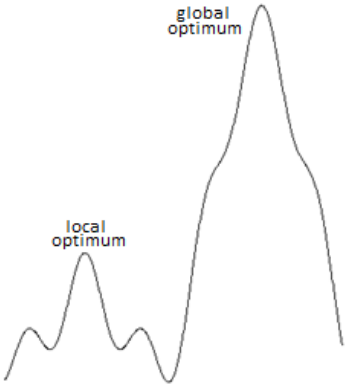


Figure 3: The optima of a part of the search space. If an optimum has the best objective value of the whole search space, it is called a global optimum. If it is the optimum of a certain search space area, but not the global optimum, then it is called a local optimum.

The basics of a meta-heuristic algorithm are shown in Figure 4. For the problem described above we start with an *initial solution* for the feature weights. Then at every iteration the algorithm generates new solutions from the current solution by systematically manipulating specific parts i.e. the feature weights, while keeping the rest unchanged. This in essence creates a neighbourhood of solutions in relation to the current solution. The solutions in the neighbourhood are 'close' to the current solution. Subsequently the algorithm selects the best solution in the neighbourhood and takes this as its new current solution. This process continues until a convergence criterion is reached, after which the search stops.

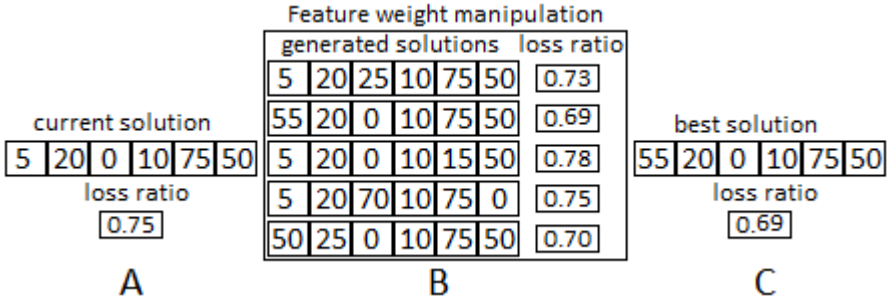


Figure 4: A) assume we have 6 features, a portfolio and a start solution of weights. For this weight vector the SPR = 0.75. B) at each iteration the feature weights of the current solution are manipulated, leading to 5 new solutions C) The best generated solution is selected and replaces the current solution, after which the process repeats.

In the upcoming sections we will dive deeper into the specifics of meta-heuristic algorithms and investigate their utility on generating high quality solution for portfolio value analysis as encountered at FRISS. Furthermore, to make proper comparisons we also generate artificial datasets for which specific characteristics e.g. portfolio size, loss ratio, etc., can be fully controlled.

Finally, the results in this thesis are accompanied by an interactive web application build with the R/Shiny framework [05,22] making all results fully reproducible and in line with the guides of reproducible research proposed by Knuth (1984) [21].

Ch.	Title	Information
2	Problem Description	Formal definition of the research problem
3	Meta-Heuristics	Description of the meta-heuristic research field and algorithms
4	Materials and Methods	The available data, objective function and implemented algorithms
5	Shiny Application	Description of the interactive shiny application
6	Results and Analysis	The performed experiments, results and analysis of the results
7	Discussion	Discussion and future research
8	Conclusion	A short overview of the most important findings and results
9	Appendix	The dataset and experiment parameters
10	References	A list of related literature

Table 1: A brief overview of the chapters and their contents.

2 Formal Problem Description

In the current situation, the FRISS score is a based on weighted features, with the weights based on the knowledge of experts and previous portfolio value analysis. We expect that the effectiveness of the FRISS score can be greatly improved by basing the weights directly on historical, i.e. empirical, data. A challenging aspect of such data is that the data can be quite noisy and is certain to include relatively large outliers. In such scenarios it is important to develop proper evaluation schemes which warrant against overfitting and allow for robust estimation of parameters.

In practice, the FRISS score is used to divide the portfolio into cases with an acceptable risk and an unacceptable risk. More formally, let A denote the set of accepted cases, the cases with acceptable risk. Assume all of these cases have a FRISS score lower than a specific cut-off value C. Furthermore, let R denote the set of rejected cases, the cases with unacceptable risk. Similarly, these cases all have a FRISS score equal to or higher than C.

The goal of the *portfolio value optimization* is to find a set of feature weights and a cut-off C such that the value of the set of accepted clients is maximized, in a sense to be made precise. For instance, two alternatives to determine the value of the set of accepted clients are the *loss ratio* and the portfolio *value*. Note that rejecting a client has two main effects i.e. we lose the premium income associated with this client, but any associated claims will not have to be paid out.

To determine the value of the accepted set, we will use the *loss ratio* of this set. The loss ratio is the total claim amount of the cases divided by the total premium income of the cases, where both the total claim amount and the total premium amount are over the cases in the accepted set. The loss ratio states how much claim amount the insurer must pay-out for every euro of premium income. Hence we can maximize the portfolio value by minimizing the loss ratio.

More formally, assume we have a set of *K* features. Let w_k denote the weight of feature *k* and **w** the weight vector. Furthermore, let p_i and s_i denote the *total premium income* and *total claim amount* associated with case i, and let f_i denote the associated FRISS score. Finally, let $\mathbb{1}_{i,k}$ represent an indicator variable which equals 1 if case i has a hit on feature *k* and 0 otherwise. The FRISS score of case i then equals:

$$f_i = \sum_{k \in K} w_k \cdot \mathbb{1}_{i,k}$$

Next, let $P_A = \sum_{i \in A} p_i$ denote the total premium income associated with the accepted set and let $S_A = \sum_{i \in A} s_i$ denote the total claim amount associated with the accepted set. The loss ratio of the portfolio is then defined as $L = S_A / P_A$.

As in practice most insurers impose volume constraints on their portfolio. For instance, the set of rejected cases R, can be at most $x\%$ of the total portfolio.

The main research question thus can be reformulated as:

*Which weight vector **w** minimizes the loss ratio, while at most x% of the portfolio is rejected?*

Mathematically, the full weight optimization problem can be formulated as the following constrained optimization problem:

Minimize:

$$(1)\ L = S_A / P_A$$

Under:

$$(2)\ P_A = \sum_{i \epsilon I} a_i * p_i$$

$$(3)\ S_A = \sum_{i \epsilon I} a_i * s_i$$

$$(4)\ a_i = \begin{cases} 1 & \sum_{k \epsilon K} w_k * \mathbb{1}_{i,k} < C \\ 0 & else \end{cases}$$

$$(5) \sum_{i \epsilon I} a_i \geq x * I$$

Formula (1) shows the loss ratio optimization formula for the portfolio. Constraints (2) and (3) define the premium income P_A and the claim amount S_A of the accepted cases, while constraint (4) defines when case i is accepted into the portfolio. Finally, constraint (5), defines the only real constraint, namely that we can only reject $x\%$ of the portfolio. Table 2 shows an overview of the variables used in (1) to (5).

Variable	*Description*
L	*The loss ratio of the portfolio associated with the accepted set*
S_A	*The total claim amount associated with the accepted set*
P_A	*The total premium income associated with the accepted set*
$i \epsilon I$	*i = 0,…,#cases*
$k \epsilon K$	*k = 0,…,#features*
C	*The cut-off value.* $C \epsilon R_+$ *and integer.*
a_i	*Is case i accepted into the portfolio?* $a_i \epsilon \{0, 1\}$.
p_i	*Total premium income associated with case i*
s_i	*Total claim amount associated with case i*
w_k	*The weight of feature k.* $0 \leq w_k \leq C$.
x	*The maximum percentage of rejected portfolio*
$\mathbb{1}_{i,k}$	*Indicator variable which equals 1 if case i has a hit on feature k and 0 otherwise*

Table 2: The variables of formula (1) and constraints (2) to (5).

3 Meta-Heuristics

We approach the portfolio value optimization problem described in the previous sections as a *combinatorial optimization problem* with constraints and use the advanced meta-heuristic search procedures to identify high quality solutions as described by Aarts et al. (2003) [01].

Meta-heuristic algorithms represent a class of search algorithms that attempt to find feasible solutions to difficult optimization problems with large search spaces. Typically the size of the solution space prohibits an exhaustive search of the whole space. Meta-heuristics provide a way to sample solutions from the solution space in order to find sufficiently good solutions. They combine a lower level heuristic search algorithm with a *meta search strategy* or, using the words of Bianchi et al. (2008) [H], meta-heuristic algorithms combine heuristics in a more general framework by using higher level concepts and search strategies.

Heuristic versus meta-heuristic algorithms
The important difference between a heuristic algorithm and a meta-heuristic algorithm lies in the higher level search strategy. To show a difference, let us take a look at the heuristic Hill Climbing algorithm. This algorithm uses a simple heuristic, namely "Having a current solution, search the neighbouring solutions for a better solution. When such a solution exists, move to this found solution and search again. When no better solution can be found, then terminate the algorithm".

When we use this algorithm on a search space as shown in Figure 3 of chapter 1, or Figure 12 of section 3.1.4, and would our initial solution be on the left of the given local optimum, then the Hill Climbing algorithm might end up in this local optimum and then terminate. However, we are searching for the global optimum more to the right. The heuristic algorithm is not able to escape from local optima and actively search for the global optimum.

However, the meta-heuristic algorithm combines a heuristic algorithm, like the Hill Climbing algorithm, with a strategy to escape the local optima of a search space, or a large area with almost equal or low objective values, and allows the algorithm to find solutions closer to the global optima. An example of such strategy can be found in the meta-heuristic algorithm Tabu Search [15]. In the Tabu Search algorithm, the higher level strategy forces the algorithm to accept solutions with worse objective values when solutions with better values cannot be found. Also the strategy blocks the algorithm to move back to the previous position in the search space.

Meta-heuristic algorithms
Classic meta-heuristic approaches include e.g. Evolutionary Algorithms [09] and Simulated Annealing [20]. More recent approaches are based on evolutionary computing methods such as Particle Swarm Optimization [19] and Covariance Matrix Adaptation Evolutionary Strategies [16]. Furthermore, additional constraints and/or augmented objective functions may be developed in order to combat overfitting and to reduce the impact of calamities i.e. data outliers.

In the upcoming sections we provide an overview of the various concepts used in modern search algorithms and provide a list of strategies that will be implemented to tackle the portfolio analysis problems as described in this thesis. Many of these algorithms share concepts and ideas which can be combined into a set of algorithms known as *hybrid-meta-heuristics*.

The next sections offer a more in-depth coverage of the various concepts related to search algorithms. Section 3.1 describes the different ingredients found in many meta-heuristic algorithms, while section 3.2 provides an overview of the meta-heuristic algorithms that we will use in this research. Finally, section 3.3 shows some practical considerations about generating solutions.

3.1 Search Concepts

In order to clarify the different ingredients found in modern meta-heuristics, we consider the following toy search problem. Given a set of features, which take integer values 0 to 9, and an objective function that maps feature vectors to objective values, find the best four feature combination, i.e. the feature vector of four features with the highest objective value.

In this case the complete *solution space* consists of all possible combinations of the four features. As each feature has one out of 10 different values, there are 10^4possible combinations in the search space, ranging from 0000 to 9999.

3.1.1 Initial Solution

The *initial solution*, or solutions, is the starting location of the algorithm within the search space. The algorithm can generate this solution itself, e.g. by using a construction algorithm, or it can be given an existing solution. Depending on the problem at hand generating a feasible initial solution to optimization problems with constraints can already be a difficult problem by itself. For the toy example we arbitrarily take 0000 as our initial solution.

3.1.2 Neighbourhood, Solution Space, Move and Path

The *neighbourhood* of the current solution contains all solutions that can be reached from the current solution by manipulating the current solution in a pre-defined systematic way. This gives us a finite set of new solutions which can be searched in its entirety. For instance, for the toy example we may create a neighbourhood by setting each of the four features to another integer value (see Figure 5).

Figure 5 shows the current solution (black circle) is 0000, while the neighbourhood (red circles) consists of solutions which have 1 added to one of the four features of the current solution.

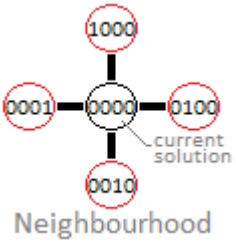


Figure 5: The neighbourhood of the current solution. Each, red circled, solution in the neighbourhood has *1* added to one of its four features.

When we chose one of the red solutions as our next solution, we call this a *move*. Hence, the search algorithm moves from one solution to the next in the search space. As we change solutions, the neighbourhood also changes. This can be seen in Figure 6. We have chosen solution 0100 as our new current solution and moved to that solution. Note that our previous solution from Figure 5 now is in the neighbourhood of our new current solution.

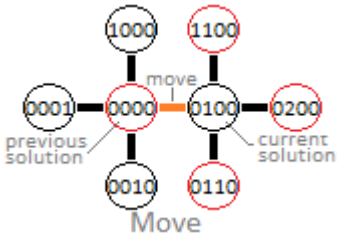


Figure 6: When we chose a solution from our neighbourhood as our next current solution. We conduct a move. Also the neighbourhood changes.

At every iteration of the search algorithm we select a move. A chain of moves is called a *path*, because in essence the algorithm walks a path in the solution space, shown as the orange lines in Figure 7. In this case the first solution of the path is the initial solution, while the final solution, 0210, is the current solution after a path consisting of three moves.

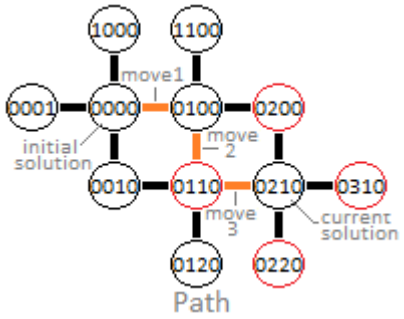


Figure 7: A path is a sequence of moves. The initial solution is the starting location of the path. The current solution is the final location of the path consisting of three moves.

Considering Figure 5, we wrote that the solution neighbourhood is generated by systematic changing a solution's features. This changing of the features can incorporate two different strategies: intensification and diversification strategies. As Glover et al. (2007) [20] describes, intensification strategies are based on changing features to encourage feature combinations which historically were found good. Also they may initiate a return to interesting, or attractive, search space areas to search them more thoroughly. On the other hand, diversification strategies seeks to incorporate new feature values and feature combinations within solutions, allowing for broader search.

3.1.3 Trajectory- and Population-based Algorithms

Meta-heuristic algorithms can be divided into two main categories: trajectory-based and population-based algorithms. Trajectory-based algorithms manipulate a single solution at each iteration. Each iteration a neighbourhood is generated around the current solution and best solution in the neighbourhood is chosen to replace the current solution. Doing this several times creates a path as shown in Figure 7.

In contrast, population-based algorithms handle multiple solutions at once. This set is often referred to as the population. A neighbourhood is generated for each solution, sometimes by combining the features of multiple solutions. Subsequently, the current population, or a part of the current population, is replaced by solutions from the neighbourhood. Figure 8 shows an example in which one current solution survives and the two others are replace by solutions from the neighbourhood.

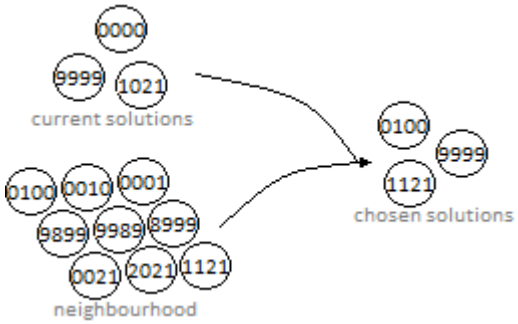


Figure 8: Select which solutions of the current iteration will be used in the next iteration. Both the solutions of the neighbourhood and the current solutions can be considered.

3.1.4 **Local and Global Optima**

The objective values of solutions in neighbouring areas are often related. Furthermore, a solution can be optimal for a given search space area. In this context we consider local and global optima. If an optimum has the best objective value of the whole search space, it is called a *global optimum*. If it is only the optimum of a small area in the search space, it is called a *local optimum*. Figure 9 shows a 2D search space with a local and a global optimum.

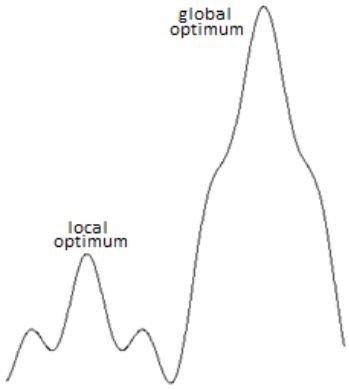


Figure 9: The optima of a part of the search space. If an optimum gives the best objective value of the whole search space, it is called a global optimum. If it is the optimum of a small search space area, it is called a local optimum.

3.1.5 **Cycling**

An important aspect of search algorithms is how they escape from local optima. For instance, from figure 9 it is clear that in order to escape the local optimum we must accept moves that will temporarily decrease the objective value of the current solution. However, when we allow this, the new solution typically has the previous local optimum as one of its neighbouring solutions. Note that a move to this neighbour, the local optimum, would increase the objective value again. Would we do this for a deterministic search strategy we are trapped and the algorithm will cycle between those two moves, see Figure 10 A) for a small example. However, cycles of more moves are also possible, see Figure 10 B).

Thus, an essential part of search algorithms is how to escape local optima while preventing cycling, also long-term cycling. For instance, Tabu Search, see section 3.2.2, does this by maintaining a tabu list which prevents such moves from happening, while another approach is to allow for randomization which is a key factor in Evolutionary Algorithms, see section 3.2.1, as accepting random moves will typically decrease the possibility of cycling.

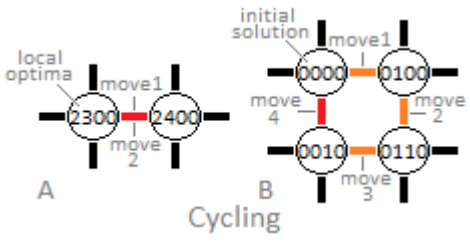


Figure 10: A) Cycling between two solutions: We are in the local optima 2300. When we move to 2400 (move 1), we leave the local optima. If we then choose move to 2300 again (move 2), we enter the same local optima again. B) Cycling between multiple solutions: We start at the initial solution, 0000, then walk a path consisting of moves 1, 2 and 3. Would we make move 4, we end up at our initial solution after a cycle consisting of moves 1 to 4.

3.1.6 Granularity

The granularity determines the detail used in selecting the feature weights. When the granularity is low, more distinct feature weights are available. For instance, a granularity of 1 allows the use of all integer weights between 0 and the cut-off value C. While a granularity of 10 only allows the integer values 0, 10, 20, etc. till the cut-off value to be used. Granularity is closely related to intensification, as a lower granularity leads to a higher degree of intensification, as the algorithm can stay longer in an area when there are more possible solutions.

Granularity	Possible Feature Weights
0 (continuous)	All decimal weights between 0 and C
1	All integer weights between 0 and C
5	0, 5, 10, …, C
10	0, 10, 20, …, C

Table 3: The different feature weights for different granularities. C is the cut-off value.

Note that some algorithms use continuous feature weights and, when the granularity is not continuous, the objective function rounds the feature weights before the objective value is calculated.

3.2 Algorithms

In the sections above we have described various aspects related to modern meta-heuristic search algorithms. In the upcoming sections we describe four meta-heuristic search algorithms from recent literature that incorporate many of the above described ingredients. The algorithms have been chosen to represent very different algorithms within the meta-heuristic research field and have proven to be able to provide high quality solutions to many challenging optimization problems. The selected algorithms include (1) Evolutionary Algorithms (simple variant), (2) Tabu Search, (3) Particle Swarm Optimization and (4) Covariance Matrix Adaptation Evolutionary Strategy (complex Evolutionary Algorithm variant).

3.2.1 Evolutionary Algorithms

The population-based evolutionary algorithms are inspired by natural evolution and originate from the 1960's. Evolutionary algorithms are one of the first meta-heuristic algorithms. Evolutionary Algorithms are population-based algorithms.

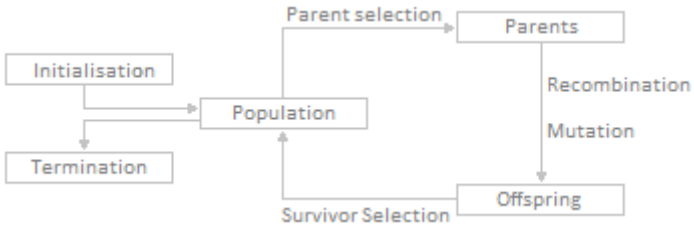


Figure 11: The general scheme of an Evolutionary Algorithm. First initialize the population, then choose the parents that may reproduce. Next, create offspring from the parents by using recombination and mutation. Finally, select the survivors from the population and the new offspring as the next generation of solutions. When you hit a stop criteria, terminate the algorithm. ([13] - Figure 3.2)

```
BEGIN
  INITIALIZE the population with candidate solutions
  EVALUATE each candidate
  REPEAT UNTIL (TERMINATION CONDITION is satisfied) DO
    1 SELECT parents
    2 RECOMBINE pairs of parents
    3 MUTATE the offspring
    4 EVALUATE the offspring
    5 SELECT individuals for the next generation
  OD
END
```

Pseudo-code 1: The pseudo-code for the general scheme of the Evolutionary Algorithm as shown in Figure 11. ([13] - Figure 3.1)

Figure 11 shows the general scheme of the algorithm and Pseudo-code 1 shows the algorithm's pseudo-code of the general scheme. In the initialisation phase the initial population is generated. The population is the group of solutions which are used in the algorithm. This initial population is mostly generated randomly, but can for example also consists of already known good solutions. Each solution has a certain fitness value based on their features, their *genes*. The higher the fitness of a solution, the better the solution is.

Next is the parent selection. The parents, solutions chosen from the current population, will reproduce and create new solutions, the offspring. A stochastic process is used to determine which solutions become the parents. In this process even the solution with the worst fitness has a probability of becoming a parent.

The reproduction of the parents is done in the recombination phase. The parent's genes are merged and each gene of the offspring is chosen randomly from one of the parents. Figure 12 shows this process on a small scale with only 4 genes. The first and fourth gene of the first parent end up in the first offspring and the second and third gene come from the second parent. The other genes end up in the second offspring.

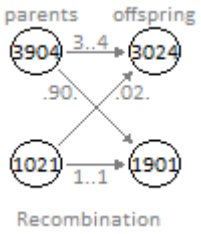


Figure 12: Recombination of two parents to create two offspring. Two genes of each parent end up in each offspring.

Immediately after the recombination phase comes the mutation phase. In the mutation phase small changes are performed to the genes of the offspring, e.g. changing a gene from 3 to 4.

Then the survivors from the current generation are chosen. Both the solutions from the population and the offspring can survive and be in the population in the next generation. Each solution has a probability to survive, based on, for example, their fitness. The higher the fitness of a solution, the higher the chance to survive.

Finally, the termination phase determines when the algorithm stops and the solution with the highest fitness is returned. The stop criterion for the termination can be i.e. the algorithm reached the maximum number of generations or the algorithm has not found an improvement for a number of generations. [13]

3.2.2 Tabu Search

Tabu Search is a trajectory-based algorithm designed to actively escape local optima and to prevent cycling. Cycling is prevented by declaring the move *tabu*. The algorithm is not allowed to choose a tabu-declared move for the next couple of iterations, keeping the algorithm from cycling into the same solutions or local optima, after just escaping those the previous iteration. Glover (1986) [14] introduced the algorithm and explains it more thoroughly in "Tabu Search – Part 1", Glover (1989) [16], and "Tabu Search – Part 2", Glover (1990) [17]. Tabu Search algorithms often use of *adaptive memory* and *responsive exploration* based on the stored memory. Adaptive memory refers to the fact that the algorithm stores data about the search in memory which it subsequently uses to improve the search process by using this data for creating more specific solutions or, for instance, to move to another part of the search space. This is called responsive exploration. Data that can be stored includes: timing, solution quality and the impact of a move [16,19].

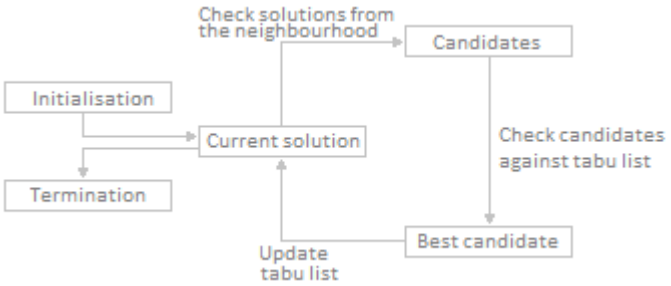


Figure 13: The tabu search algorithm cycle. Initialize the solution, then select the candidate solutions from the neighbourhood. Continue by checking the candidates against the tabu list and selecting the best candidate. Finally, update the tabu list and replace the current solution by the best candidate solution.

```
BEGIN
  INITIALIZE the current solution, the tabu list and the candidate list
  REPEAT UNTIL (TERMINATION CONDITION is satisfied) DO
    1 GET the neighbourhood for the current solution
    2 FOR EACH candidate from the neighbourhood DO
       a CHECK if features of the candidate are on the tabu list
       b ADD candidate to candidate list if it passed 'a'
      OD
    3 SELECT the best candidate from the candidate list
    4 UPDATE the current solution with the best candidate
    5 UPDATE the tabu list
  OD
END
```

Pseudo-code 2: The pseudo-code for the general scheme of the Tabu Search algorithm as shown in Figure 13 (Adapted from [06] – Algorithm 2.10.1)

The single solution is randomly generated in the initialisation phase. The objective value is determined by the objective function and a higher objective value means a better solution.

Each iteration the neighbourhood of the current solution is generated. All candidate solutions in the neighbourhood can be reached by changing the features of the current solution in a predefined manner. Each candidate solution in the neighbourhood is checked against the *tabu list*. The tabu list contains previous feature changes. These feature changes are 'tabu'. The algorithm is not allowed to redo these stored feature changes, generating solutions found in previous iterations. Adding feature changes to the tabu list reduces the probability of cycling, because for a number of iterations making specific changes to specific features are not allowed.

When a candidate solution passes the tabu list check, the candidate is added to the candidate list as possible solution for the next iteration of the algorithm. After all candidates have been checked, the candidate solution with the highest objective value is taken from the candidate list. This candidate solution replaces the current solution and the feature differences between the two solutions are added to the tabu list. Finally, feature changes

are removed from the feature list, when the number of feature changes is larger than the maximum size of the tabu list.

The algorithm is finished when the termination condition is satisfied. This condition can i.e. be reaching the maximum number of iterations or not having found an improvement of the objective value for a number of iterations. [06]

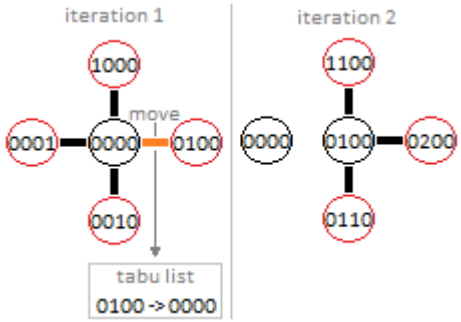


Figure 14: When at iteration 1 the move *0000 → 0100* is chosen from the solutions in the neighbourhood, the move *0100 → 0000* is placed on the tabu list. In iteration 2 the solution *0000* is not considered as a valid solution in the neighbourhood, because the move to reach the solution is on the tabu list.

3.2.3 Particle Swarm Optimization

The Particle Swarm Optimization algorithm is a population-based algorithm, which is based on bird flocking simulations. The algorithm was introduced by Kennedy & Eberhart (1995) [25]. Each solution, or particle, has a velocity which directs the movement of the particle through the search space.

In the initialisation phase the particles from the swarm are generated randomly scattered throughout the search space and get a random velocity. Each particle contains the following information: its position in the search space, the fitness value of this position, the velocity at which it moves. Also it stores, in memory, its best known position and the fitness value of this position.

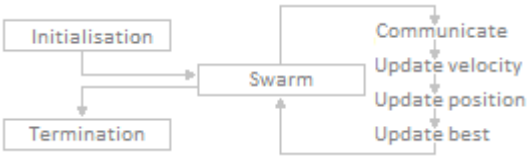


Figure 15: The algorithm cycle. Initialize the swarm with random position and velocity. Communicate with other particles and update your velocity, position and best known position based on that information. (Based on [07] – Algorithm 6.2.1)

```
BEGIN
  INITIALIZE the particle swarm
  REPEAT UNTIL (TERMINATION CONDITION is satisfied) DO
    1 FOR EACH particle in the swarm DO
       a COMMUNICATE with up to K other particles
       b UPDATE velocity of particles
       c UPDATE position of particles
       d UPDATE previous best and fitness value of particles
    OD
  OD
END
```

Pseudo-code 3: The pseudo-code for the general scheme of the Particle Swarm Optimization algorithm as shown in Figure 15 (Adapted from [07] – Algorithm 6.2.1)

Each time step each particle communicates with up to *K* other particles and exchanges information about their previous best location. The particles communicated with are called *informants*. Each time step, if the best solution has not been improved, the informants of each particle are randomly reassigned. [09,10]

Following M. Clerc (2010) [09], the new velocity of particle *i* is calculated with formula $v(t+1)_i = w * v(t)_i + C_1 * (p(t)_i - x(t)_i) + C_2 * (l(t)_i - x(t)_i)$, where $v(t)_i$ is the velocity of particle *i* at time step *t*, $p(t)_i$ is the previous best location of particle *i*, $x(t)_i$ is the current location of particle *i* and $l(t)_i$ is the previous best location of all the informants. C_1 and C_2 are drawn from a $uniform(0, 0.5 + \ln(2))$ distribution and $w = 1/(2 * \ln(2))$. Then the new position of the particle is calculated with the formula $x(t+1)_i = x(t)_i + v(t+1)_i$. When the velocity and position of all particles have been updated, then the previous best location and fitness value of each particle is updated.

Figure 16 shows an example of the updating of the velocity and position of a particle. Part A shows the information that particle *i*, currently has. Part B shows the calculation of the new velocity. Note that C_1, C_2 and C_3 have already been drawn. Finally, part C shows the new position for particle *i.*

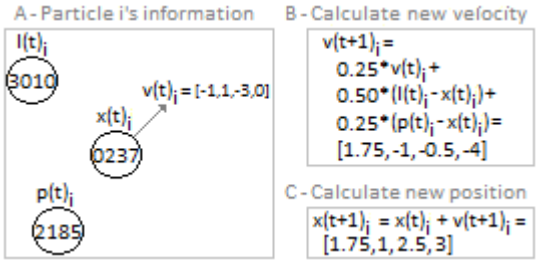


Figure 16:
A) The current information of particle i.
B) Calculating the new velocity based on the current information. Note that to simplify this example $w = C_2 = 0.25$ and $C_1 = 0.50$.
C) Calculating the new position of particle i, using the new velocity.

The algorithm terminates and returns the best found position in the search space when the stop criterion is satisfied. This stop criterion can i.e. be that the maximum time step has been reached or that the swarm has flocked together in a small area in the search space.

3.2.4 Covariance Matrix Adaptation Evolutionary Strategy

Hansen et al. (2001) [22] propose the trajectory-based Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) algorithm as an improvement of standard Evolution Strategy algorithm for local optimization, as well as global optimization [17]. The algorithm is designed to solve problems that cannot easily be solved by naïve random search, i.e. brute-force search and deterministic search. Based on the chosen population size, CMA-ES can be a trajectory-based or population-based algorithm. The generation cycle of the algorithm is shown in Figure 17.

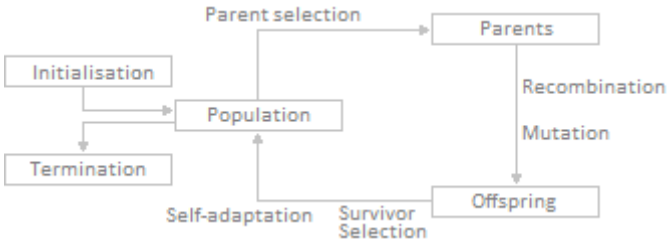


Figure 17: The generation cycle of the CMA-ES algorithm. Start with initialisation of the population. Choose the parents that my generate offspring by recombination and mutation. Select the solutions from the offspring and current population that survive. Then conduct self-adaptation on the recombination, mutation and selection parameters. Finally, when you hit a stop criterion, terminate the algorithm. (Based on [13] – Figure 3.1)

```
BEGIN
  INITIALIZE the initial populations and the covariance matrix
  REPEAT UNTIL (TERMINATION CONDITION is satisfied) DO
    1 SELECT the parents
    2 SAMPLE the offspring
    3 SELF-ADAPTATION of the search parameters
       a UPDATE the mean
       b UPDATE the covariance matrix of the normal distribution
       c UPDATE parameter step size σ
  OD
END
```

Pseudo-code 4: The pseudo-code for the general scheme of the CMA-ES algorithm as shown in Figure 17 (adapted from [02], slide 16)

The initialisation phase generates the initial population with population size μ, which are generated randomly, and the self-adaptation parameters of this solution.

Following Auger et al. (2013) [02], λ offspring solutions are generated each generation g. The offspring for generation g is generated using the formula $x_i^g \sim m^{g-1} + \sigma^{g-1} * N(0,\ C^{g-1})$ for each of the $i = 1..\lambda$ offspring. In this formula m^{g-1} is the weighted average of the population of generation *g-1*, σ^{g-1} is the mutation step size of generation *g-1* and is based on the length of the evolution path of *m* over the previous generations. The evolution path is the distance in the search space that *m* covered in the previous generations. When the evolution path is long then make $\sigma^g > \sigma^{g-1}$ and when the evolution path is short $\sigma^g < \sigma^{g-1}$. Last is C^{g-1}, which is the covariance matrix used by the multi-variate normal distribution in generation *g-1*.

As most evolution strategy algorithms CMA-ES uses parent selection, recombination and mutation. These are hidden in the update steps of the self-adaptation parameters. The parent selection and recombination steps can be found in calculating m^{g-1}. m^{g-1} is the weighted average over the poulation of generation *g-1*. [21] When a weight is 0 then that specific solution is not selected to become a parent and all solutions that become parents recombine their features to create m^{g-1}. Mutation is included in the generation of the offspring for generation g. We add values drawn from the multi-variate normal distribution $N(0,\ C^{g-1})$ to m^{g-1}.

The most important part of the self-adaptation of the CMA-ES algorithm is updating the covariance matrix. The covariance matrix keeps track of the pairwise dependencies between the solution variables. By keeping track of these dependencies the algorithm aims to increase the likelihood of repeatedly producing succesful mutation steps.

The algorithm terminates and returns the best found solution when the stop criterion is satisfied. This stop criterion can i.e. be that the maximum number of generations has been reached or that the fitness value has not been improved for a number of generations.

3.3 Practical considerations of solutions

In practical applications three criteria come into play in accepting solutions, i.e. the solution feasibility, the solution generalization and the solution complexity.

3.3.1 Feasibility

In the context of the portfolio value analysis problem described in this work, a solution is considered feasible when it satisfies the rejection constraint, see formula (5) in chapter 2. This constraint states that the rejected set R can contain a maximum of $x\%$ of the portfolio. This fact makes it generally easy to construct feasible solutions.

Furthermore, when solutions violate this constraint, they are easy to make feasible. Note that by changing enough feature weights to zero, we will always end up with a feasible solution. In addition, the 0-weight vector is always feasible as you do not reject a single case.

Figure 18 shows a small example of two solutions. The first solution has a higher loss ratio and acceptance percentage, "% accepted", and the second solution has a slightly lower loss ratio and reasonably lower acceptance percentage. When taking the minimum acceptance percentage as 0.90, or 90%, the first solution is feasible, while the second is not.

solutions						loss ratio	% accepted	feasible
5	20	0	10	75	50	0.70	0.93	yes
5	20	70	10	75	0	0.69	0.88	no

Figure 18: Two solutions, the second solution has a slightly better loss ratio, but rejects more cases. Having the maximum reject percentage set to 10% makes the first solution feasible, while the second is unfeasible.

3.3.2 Overfitting

A concern in applying meta-heuristics is that they can be too aggressive. When an algorithm generates solutions which work well on training data, but generalize poorly on unseen data, it is said to *overfit* the data. The overfitting can happen, for instance, due to the presence of outliers in the data, for instance, the algorithm tries only to remove cases with a high claim amount, instead of removing groups of cases which together have a high claim amount. Without proper care the meta-heuristics may focus on regions in the solution space that will lead to the rejection of outliers. This can lead to solutions that reject only a few outliers, which leads to a strong reduction in loss ratio, with minimum volume loss. However, the outliers in the portfolio data are typically random and do not represent systematic behaviour. Hence, focusing on non-representative data points may lead to overfitted solutions. In section 4.1.2 the outliers will be explained in-depth.

3.3.3 Solution Complexity

Meta-heuristic algorithms have another problem besides overfitting, although there are similarities. The solutions that meta-heuristic algorithms find, can be very complex, because they have just found one special pattern that provides a very good solution. This means that a very complex solution can often be an overfitting solution.

Also most complex solutions are hard to understand. Would we use the weight vector found by an algorithm, we must be able to understand the solution, or at least be able to explain what it does, based on the training data. We would like to be able to explain the weight vectors as FRISS provides its services to insurers. Insurers do not like to choose for black box solutions, because they must be able to explain to their clients why certain policies are rejected.

To reduce the solution complexity we can use Occam's razor [28]. This principle states, that when two solutions, one simple and one complex, predict equally well, then the least complex one should be chosen. When there is not enough information to truly determine the best solution, we can better choose the solution that makes the least assumptions or is easier to understand. There are voices against using Occam's razor in certain situations [11]. However, for our problem simple and understandable solutions are the key. An additional advantage of using Occam's razor is that the chance of a solution overfitting on the data is reduced by choosing simpler solutions.

4 Methods

This chapter contains descriptions about the data and the different methods used in this thesis. First, section 4.1 describes the data that was available for the thesis and which data we created ourselves. Section 4.2 shows the development of the objective function. Finally, section 4.3 shows the implementations of the algorithms, described in section 3.2, we will use to solve the problem.

4.1 Data Description

4.1.1 Data

The data of this thesis is divided into three different kinds: real data, randomized real data and artificial data. The real data consists of anonymized portfolio value analysis data, while the artificial data is generated by a dataset generator with user-specified parameters.

Representation
Each dataset contains cases and each case contains the total premium income, the total claim amount and the feature hits of the case. The premium income and the claim amount are decimal numbers, while the hit, or no hit, on a feature is binary. Figure 19 shows an example of some cases for a given dataset.

ind5	ind6	ind7	ind8	ind9	ind10	ind11	ind12	ind13	ind14	ind15	ind16		ind114	ind115	ind116	claim	premium
0	0	0	1	0	0	0	0	0	1	0	0		0	0	0	3359.32	1290.10
0	0	0	1	0	0	0	0	0	0	1	0		1	0	0	0.00	216.28
0	0	0	1	0	0	0	0	1	0	0	0		1	0	0	0.00	14.79
0	0	0	1	1	0	0	0	0	0	0	0		0	0	0	0.00	861.71

Figure 19: Four cases. This dataset has 116 features (ind1…ind116), a claim amount and a premium income. "ind" stands for "risk indicator".

Case Characteristics
We can roughly divide the cases of a dataset based on two characteristics i.e. the number of feature hits and the height of the claim amount. For the first characteristic, we consider two major differences i.e. cases with zero feature hits and cases with at least one hit. For the second characteristic, we consider differences in cases with a claim amount of zero, a normal claim amount and those with exceptionally high claim amounts i.e. the outliers in the dataset. The latter group is described more in-depth in section 4.1.2. Note that in a portfolio value analysis the cases with a FRISS score of zero will always be in the accepted group of policies as their score is always below the cut-off value.

Compound Cases
Datasets at FRISS typically contain many thousands of cases. The number of unique feature vectors i.e. hit patterns, however, is often much lower. For instance, all cases with a FRISS score of zero have the same hit pattern, i.e. none of them has any hits. The claim amount and premium income of cases with the same hit pattern can be summed up, combining multiple cases to a single case. This single case is called a *compound case*, as the single case is a composition of multiple cases with the same hit pattern. Combining cases can greatly reduce the size of most datasets and adds an extra item to the dataset, *compound*, which is the number of cases combined into the compound case. Figure 20 shows five compound cases. Generally, the size reduction allows for faster calculation of the objective value of a solution as there are fewer cases to compute the FRISS score for.

ind1	ind2	ind3	ind4	ind5	ind6	ind7	ind8	ind9	ind10	ind35	ind36	ind37	ind38	ind39	compound	claim	premium
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	17226	5056137.21	7357673.18
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	164	66367.55	111316.19
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	2	868.11	1789.99
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	3	0.00	2096.99
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1672	291036.09	617817.38

Figure 20: Five compound cases of a compressed dataset. They contain a total of 19067 cases, see the "compound" item. The claim amount and premium income are the sum of the claim amount and premium income of the compound cases. "ind" stands for "risk indicator".

4.1.2 Calamities

The outliers in the data are often referred to as calamities. Calamities form an important part of the optimization problem. Calamities are cases with extremely high claim amounts. High claim amounts can occur e.g. when a house burns down or when an expensive vehicle is involved in an accident. These cases have a very large influence on loss ratio computations. When not properly addressed, the optimization process may bias towards solutions that only remove a few calamities. However, we cannot simply remove the calamities from the data, as they represent genuine claims. Also calamities cannot be easily modelled, because they occur mostly by chance and the number of calamites is typically small, around 0.15% of all cases with at least one claim.

Which cases will be referred to as calamities are mostly determined by determining a maximum claim amount for a *normal* case and any cases with a larger claim amount are considered calamities. This maximum normal claim amount is typically the value of the 99.99th percentile of the claim amount distribution. At FRISS for cars the calamity value is taken as 25.000, but for housing this value is much higher.

ind1	ind2	ind3	ind4	ind5	ind6	ind7	ind8	ind9	ind10	ind35	ind36	ind37	ind38	ind39	compound	claim	premium
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	26094.28	726.43
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	25562.14	1168.90
0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	35216.40	568.03
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	36652.95	967.55
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	66443.33	885.40

Figure 21: Five calamity cases. The claim amount is much higher than the claim amount of any normal claim. "ind" stands for "risk indicator".

Spreading Calamities

We might deal with calamities by spreading them out over other claims. Spreading is a technique used by FRISS for portfolio value analysis. The claim amount of each calamity is capped at the maximum normal claim amount for a case, as explained in the last paragraph. We then sum the claim amount that is above this value, which we refer to as E, the *excess amount*. Then we divide the premium of each case, p_i, by the total premium amount over all cases, P, i.e. $p_{i\%} = p_i/P$. $p_{i\%}$ is the percentage of the total premium amount that is associated with case i. Finally, the claim amount of case i is increased by $p_{i\%}$ of the excess amount E. Note that the calamities also receive their share of the excess amount.

This leads to a re-distribution of the claim amount in which the total loss ratio and premium amount remains unaltered, while the excess E is spread out over the cases in such way that cases with low premium amounts receive small corrections, while others receive larger corrections. Other approaches to deal with calamities are possible, but in this thesis only this approach is considered. Figure22 shows five spreaded cases of which the first two were calamity cases.

ind1	ind2	ind3	ind4	ind5	ind6	ind7	ind8	ind9	ind10		ind35	ind36	ind37	ind38	ind39	compound	claim	premium
0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	1	25034.81	726.43
0	0	0	0	0	1	1	0	0	0		0	0	0	0	0	1	25055.97	1168.10
0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	1	9.55	199.41
0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	1	37.39	780.33
0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	1	2.32	48.37

Figure 22: Five cases of a spread-out dataset. The first two cases were calamities, the last three cases were normal cases. The claim amount has first been topped-off at 25.000. Then the topped-off claim amount of the calamities has been added to the claim amount of all cases based on the premium amount of each case. "ind" stands for "risk indicator".

Note that spreading might reduce the overfitting of the meta-heuristic algorithms on the data as otherwise the algorithm may reduce the loss ratio only by removing only a few calamity cases. Such a move would indeed reduce the loss ratio, but will likely lead to a solution that generalizes poorly, because by far most calamities are related to chance effects. These chance effects are not possible to model e.g. someone had a car accident with bodily harm

4.1.3 Randomized Real Dataset

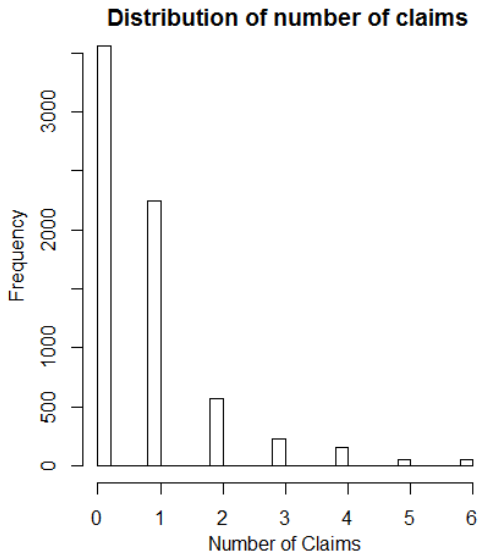
The randomized real datasets are created from the normal real datasets. First, the dataset is split into two parts. The feature hits and the pair of the claim amount and premium income. The feature hits and the pairs are then put together randomly, so that each combination of feature hits is joined with a new claim amount and premium income. This randomized real dataset will be used in the experiments of chapter 6 to determine if an algorithm can find a pattern where no pattern should be present i.e. to detect the ability of algorithms to overfit on the data.

4.1.4 Artificial Data

Comparing the effectiveness of the meta-heuristics is challenging as the true relation of the measured feature set and the claim amounts is not known in the real datasets. Therefore, we will create a set of artificial datasets to test specific scenarios in which the ground truth, i.e. the relation of the features and the premium and claim amounts, is known. These sets will be generated by an artificial dataset generator that is specially build for this thesis and forms an important ingredient in the experiments described in chapter 6. Knowing which features are correlated to the claim amount also helps to see if the algorithms pick the correct feature weights.

The dataset generator generates datasets based on user input and the expert knowledge of FRISS. First, the internal parameters are calculated based on user input or on an existing dataset. These internal parameters are then used to stochastically generate the artificial dataset.

The expert knowledge is knowledge that was obtained from historical data and previous portfolio value analysis. For instance, when we have a case, the number of claims of that case is Poisson distributed, see Figure 23, and the claim amount per claim is Lognormal distributed, see Figure 24. Finally, the premium income per case follows an Exponential distribution, see Figure 25.



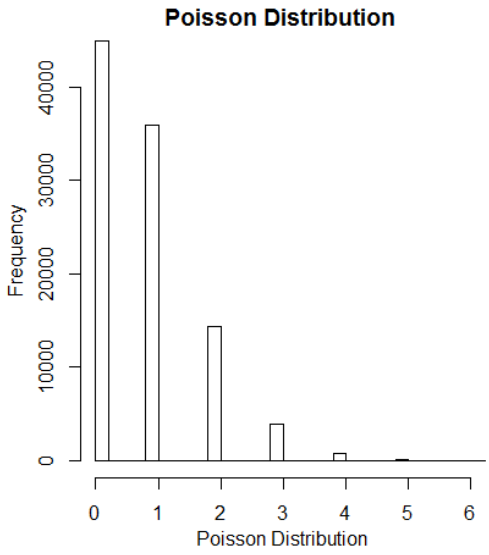
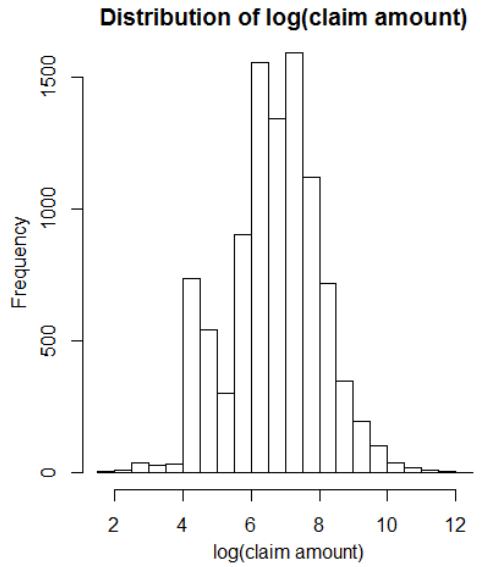


Figure 23: The distribution of the calculated number of claims for one of the real datasets and 100.000 instances of the Poisson distribution. We can conclude that the number of claims could be Poisson distributed.



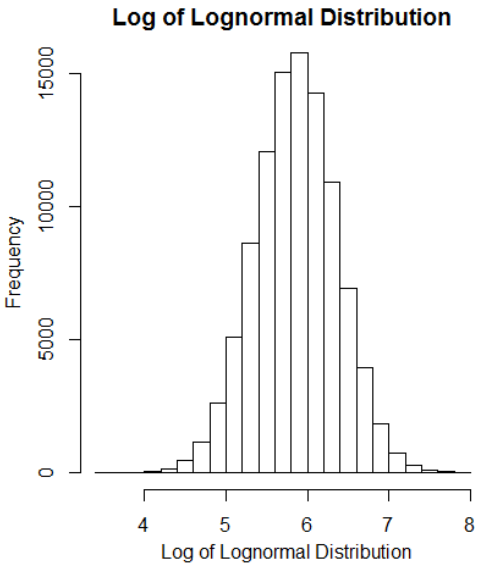
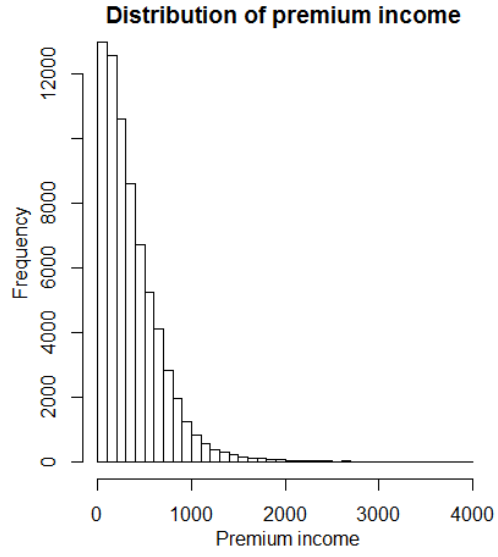


Figure 24: The log of the claim amount of a real dataset and the log of 100.000 instances of the lognormal distribution. We can conclude that the claim amount could be lognormal distributed.



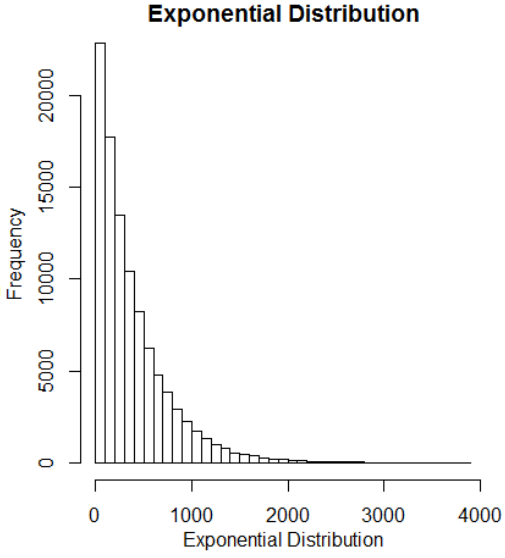


Figure 25: The premium income and 100.000 instances from the exponential distribution. We can conclude that the premium income could be exponentially distributed

We make two assumptions when generating an artificial dataset. The first assumption is that there is a connection between the number of feature hits and the chance to have one or more claims. In practise, for each feature hit of a case the mean of the Poisson distribution, used for that specific case, is increased by a small amount. The second assumption is that every case with at least one claim has the same chance of being a calamity case.

An artificial dataset is divided into up to four different risk groups, risk groups 0 to 4. Each risk group can have different parameters. Each risk group has a specific set of features for which it can hit. However, risk group 0 contains no features as this group contains all cases without feature hits. Cases in the other three groups can hit on their own features and the features of the lower risk groups, i.e. cases of risk group 1 can only hits on his own set of features, while cases of risk group 2 can hits on both their own set of features, as well as the features of risk group 1. Finally, each case of a risk group must have at least a single feature hit in his own group.

Section 5.1 shows the artificial dataset generator as implemented in the application and the generation of data is explained more in-depth.

4.2 Objective Function

The objective function is used to determine how *good* a solution is. The solution with the best objective value is the solution returned by the algorithms after termination. Different options are available for measuring the objective value of a solution. For this thesis we focus on the loss ratio as the objective value of a solution. The basic objective function therefore is:

$$(1)\ Objective\ value = LossRatio = {}^{S_A}/_{P_A}$$

where S_A is the total claim amount over all claims in the accepted set A and P_A is the total premium income over all claims in the accepted set A.

Taking the loss ratio as the objective value of a solution, immediately means that the best solution has the lowest objective value. However, our problem has a restriction on the number of cases that can be rejected, in other words the solution with the lowest objective value can be infeasible. Objective function (1) does not check if a solution is infeasible nor pushes the algorithm towards producing feasible solutions. It will only search for the solution that minimizes the loss ratio.

By adding a penalty to the objective function, algorithms will be pushed towards producing feasible solutions, because a solution that is closer to being feasible has a lower penalty and thus can have a better objective value. This gives us objective function (2).

$$(2)\ O_{value} = {}^{S_A}/_{P_A} + \left(1 - \mathbb{1}_{a_{\%} \geq x}\right) * (1 - x - a_{\%})$$
$$\text{with } a_{\%} = \frac{1}{I}\sum_{i \in I} a_i$$

Objective function (2) incorporates a penalty in the form of $\left(1 - \mathbb{1}_{a_{\%} \geq x}\right) * (1 - x - a_{\%})$. When a solution is feasible, having $\mathbb{1}_{a_{\%} \geq x} = 1$, then no penalty is given. However, when the solution is infeasible, a penalty of $(1 - x - a_{\%})$ is given. The size of the penalty is the distance to feasibility, e.g. when a solution accepts only 80% of the cases and a solution is feasible when it accepts 90%, then 0.1 is added to the solution's objective value.

Objective function (2) is used for the CMA-ES and PSO algorithms. For the EA and TS algorithms a slightly different objective function is used. As we programmed the EA and TS algorithms ourselves this allows for more freedom in designing the objective function for the algorithms. CMA-ES and PSO on the other hand are pre-programmed and have limited possibilities for the objective function.

The objective function used for the EA and TS algorithms is (3).

$$(3)\ O_{value} = {}^{S_A}/_{P_A} + \left(1 - \mathbb{1}_{a_{\%} \geq x}\right) * (1 - x - a_{\%}) * \left(1 + 9 * {}^{iter}/_{maxiter}\right)$$
$$\text{with } a_{\%} = \frac{1}{I}\sum_{i \in I} a_i$$

This objective function increases the penalty given to an infeasible solution each generation, for EA, and iteration, for TS. The first generations the penalty is only slightly larger than in objective function (2), but for the last iteration the penalty is 10 times as high. This forces the algorithms even stronger to search for feasible solutions.

4.3 Implemented Algorithms

In this thesis different one variant of each type of the meta-heuristic algorithms, described in section 3.2, have been implemented. This section shows for each of the algorithms the best performing variant. These variants are further addressed in the experiments described in section 6.1.

4.3.1 Evolutionary Algorithm

Our Evolutionary Algorithm uses a population of 10 individuals which are randomly generated in the initialisation phase. The fitness of each individual solution, generated by the algorithm, is determined by evaluation function (3) of section 4.2. The algorithm terminates after 1000 generations. The granularity of the feature weights is 10. This helps keeping the solutions simpler and more understandable for the user.

Each generation, all solutions in the population become parents and are assigned randomly in pairs of two. Each pair will generate two offspring through recombination and mutation. The probability for each gene to be assigned to offspring one or two is $p_r = 0.1 * (1.01 - \frac{g}{maxg})$. The result is that each generation the probability for each gene to end up in the other offspring is decreased. In the mutation phase each gene mutates with probability $p_m = 0.1 * (1.01 - \frac{g}{maxg})$. When a gene mutates, its new value is chosen randomly from the full range of feature weights. Also, each gene has a probability $p_0 = 0{,}05 * (1.01 - \frac{g}{maxg})$ to set its weight to 0. This second mutation increases the probability that the created offspring is feasible, while also keeping the solutions as simple as possible. In this case 'simple' means using only the features that are really needed for a good fitness value.

At the end of each generation the best solutions from the population and the offspring survive and become the population for the next generation.

The algorithm focusses on finding feasible solutions as fast as possible, while trying to keep the solution fitness low, and then improving the fitness. As the recombination and mutation probability decreases slowly each generation, fewer changes are made to the feature weights the longer the algorithm runs. This reduces jumping away from the current optimal solution. However, this also means that solutions can get stuck in local optima easier.

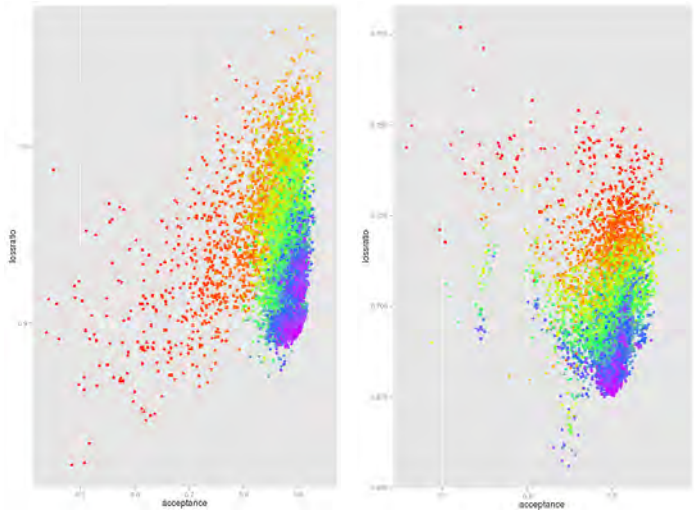


Figure 26: The loss ratio and acceptance percentage of all solutions generated by two runs of the Evolutionary Algorithm. The colour of each solution is linked to a specific iteration: Red (0), yellow (250), green (500), blue (750) and purple (1000).

Parameter	Value	Abbrevation (when used in a formula)
Population Size	10	
Number of Generations	1000	maxg
Current Generation	1..maxg	g
Number of Offspring	10	
Recombination Probability (for each gene)	0.1*(1.01 – g/maxg)	p_r
Mutation Probability to random weight (for each gene)	0.1*(1.01 – g/maxg)	p_m
Mutation Probability to 0 weight (for each gene)	0.05*(1.01 – g/maxg)	p_0
Granularity	10	
Cut-off Value	100	C
Initial Solution Feature Weight Range	1..C	

Table 4: The parameters used in the Evolutionary Algorithm. The second column contains the abbreviation of the parameter (when the parameter is used in a formula) and the value is the initial value or value range of the parameter.

4.3.2 Tabu Search

The feature weights of the initial solution are chosen randomly. The algorithm runs for 1000 iterations and then terminates. Each iteration the algorithm selects 10 candidates from the neighbourhood of the current solution. The neighbourhood consists of all solutions that differ in up to 5 features from the current solution. The difference of these features is chosen from $m = \{-10, -5, -1, 1, 5, 10\}$.
To select the candidate solutions, the algorithm first selects 5 features that are not on the tabu list and then randomly selects 10 solutions from the neighbourhood which differ from the current solution in one or more features and has no differences in any other feature. So, for example, when the 5 chosen features are features 1 to 5, a solution with only changes in features 1 and 2 can be a candidate, but a solution that has changes in features 1, 2 and 6 is not. From the 10 candidate solutions the best solution is chosen to replace the current solution and the chosen 5 features are placed on the tabu list. The size of the tabu list is 25% of the total number of features. When the size exceeds the 25% the oldest features are removed from the tabu list.

The 5 features are selected randomly, but the algorithm uses adaptive memory programming [18] to assigns a probability to each feature. This probability is based on the previous times the feature was selected by the algorithm. When the last time the feature was selected ended up with a candidate solution that had a higher objective value than the current solution, the probability of choosing the feature is higher. Also the number of times the algorithm found a candidate solution that was better or worse than the current solution is taken into

account. The probability of choosing a feature increases when more candidate solutions with better object values were found in previous iterations.

The objective value is obtained by using evaluation function (3) of section 4.2.

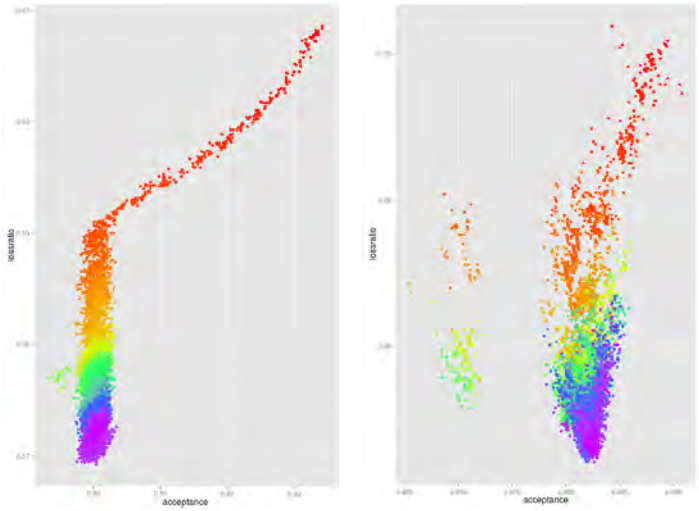


Figure 27: The loss ratio and acceptance percentage of all solutions generated by two runs of the Tabu Search algorithm. The colour of each solution is linked to a specific iteration: Red (0), yellow (250), green (500), blue (750) and purple (1000).

Parameter	Value	Abbrevation (when used in a formula)
Population Size	1	
Number of Iterations	1000	maxg
Number of Candidate Solutions	10	
Weight Mutation Range	{-10, -5, -1, 1, 5, 10}	m
Number of Changed Features	5	
Tabu List Size	25% of #features	
Cut-off Value	100	C
Initial Solution Feature Weight Range	1..C	

Table 5: The parameters used in the Tabu Search algorithm. The second column contains the abbreviation of the parameter (when the parameter is used in a formula) and the value is the initial value or value range of the parameter.

4.3.3 Particle Swarm Optimization

In R the package “pso” [04] contains two variants of the Particle Swarm Optimization algorithm, namely the Standard PSO 2007 and 2011 variants described by Clerc (2012) [10]. We will use the Standard PSO 2007 variant. The algorithm will run for 400 time steps and then terminates.

The particle swarm consists of 25 particles of which 24 are initialized randomly scattered through the search space by the algorithm and one is defined as the 0-weight vector. This might increase the chance that the algorithm generates feasible solutions earlier, because at least one initial solution is feasible. All particles start with a random velocity.

The algorithm uses evaluation function (2) of section 4.2. The evaluation function calculates the fitness value using a granularity of 10. However, as the PSO algorithm itself uses continuous weights, the granularity of the weights only affects the evaluation function. The evaluation function therefore rounds the weights before calculating the fitness value.

We use 3 informants, although we also tried different numbers, based on research from Garcia-Nieto et al. (2012) [14]. They recommend using around six informants. However, our results when using six informants were worse than our chosen three informants.

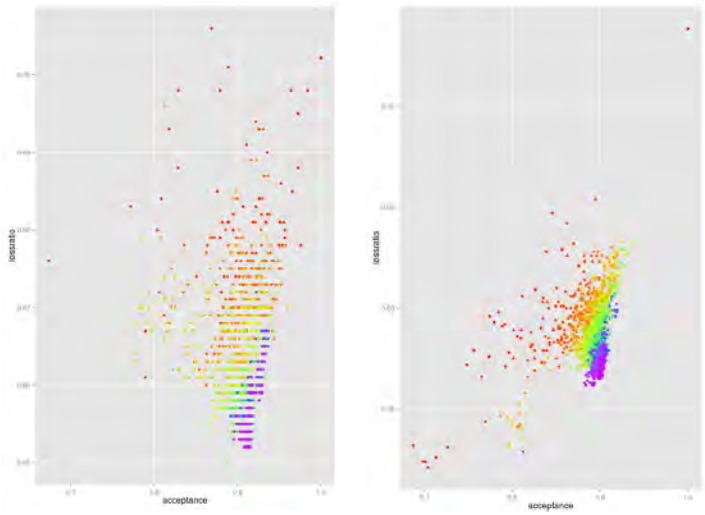


Figure 28: The loss ratio and acceptance percentage of all solutions generated by two runs of the Particle Swarm Optimization algorithm. The colour of each solution is linked to a specific iteration: Red (0), yellow (100), green (200), blue (300) and purple (400).

Parameter	Value	Abbrevation (when used in a formula)
Population Size	25	
Number of Time Steps	400	maxg
Granularity	10	
Number of Informants	3	K
Cut-off Value	100	C
Initial Solution Feature Weight Range	$[0, {}^{C}/_{2}]$	

Table 6: The parameters used in the Particle Swarm Optimization algorithm. The second column contains the abbreviation of the parameter (when the parameter is used in a formula) and the value is the initial value or value range of the parameter.

4.3.4 Covariance Matrix Adaptation Evolutionary Strategy

In R the package "cmaes" [31] contains the CMA-ES algorithm. The package allows for changing the initial parameters, but changing parameters, other than the population size and number of generations, is discouraged. The explanation by Auger and Hansen (2013) [02] is that the algorithm choses near optimal parameters itself, based on previous research.

Our CMA-ES algorithm uses a population size of 1 with 10 offspring created each generation. The feature weights of the initial solution are chosen randomly from the range $[{}^{C}/_{5}, {}^{C}/_{2}]$ where C is the cut-off value. This range seems to give a good initial solutions.

The algorithm uses evaluation function (2) of section 4.2 and uses a granularity of 1. This slightly simplifies the solutions the algorithm returns without increasing the loss ratio, because integer weighted solutions are easier to read than continuous weighted solutions. However, using higher granularities than 1, makes the algorithm find solutions that do have an increased loss ratio. Also more runs end up in bad local optima. Note that CMA-ES uses continuous weights. As we are using a granularity of 1, the fitness function rounds the weights to integers before calculating the loss ratio.

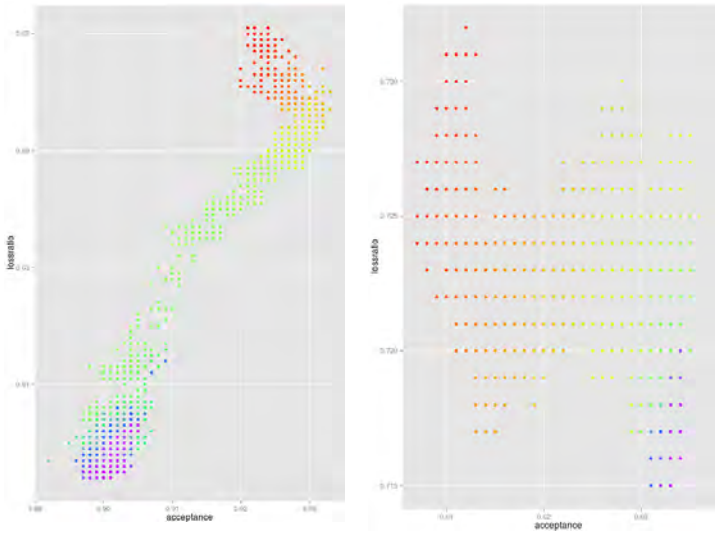


Figure 29: The loss ratio and acceptance percentage of all solutions generated by two runs of the Covariance Matrix Adaptation Evolutionary Strategy algorithm. The colour of each solution is linked to a specific iteration: Red (0), yellow (250), green (500), blue (750) and purple (1000).

Parameter	Value	Abbrevation (when used in a formula)
Population Size	1	μ
Number of Generations	1000	maxg
Granularity	1	
Number of Offspring	10	λ
Cut-off Value	100	C
Initial Solution Feature Weight Range	$[C/5, C/2]$	

Table 7: The parameters used in the Covariance Matrix Adaptation Evolutionary Strategy algorithm. The second column contains the abbreviation of the parameter (when the parameter is used in a formula) and the value is the initial value or value range of the parameter.

5 Shiny Application

"shiny" [08] is a package for the statistical program R [29] that allows for easy to build and user-friendly web applications. This chapter shows the shiny application that we specifically built for this thesis. The application is mostly written in R, but a small part is written in C++ for speed. [03,12] The application is available and free to use on "https://padejonge.shinyapps.io/WeightOptimisation". Note that the datasets loaded into and created in the free to use application cannot be saved. The sections of this chapter give an overview of the three tabs in the application. These tabs are the "Artificial Dataset Generator", the "Algorithm" and the "Analysis" tabs.

5.1 Artificial Dataset Generator

The first tab of the Shiny Application contains the Artificial Dataset Generator and options for loading and saving datasets into and from the application. In this section we will show the user interface of the Dataset Generator.

5.1.1 Overview

User input

The left side of the first tab, see Figure 30, contains the user input for the dataset generator's parameters. Here we see multiple options grouped together. These groups are *Risk Group Parameters*, *Mean feature hits per rank* and *Calamity Options*. Also there are some single parameters, namely *Remove 0 hit features*, *Seed* and *Input the probability to hit on features*, of which the latter is optional.

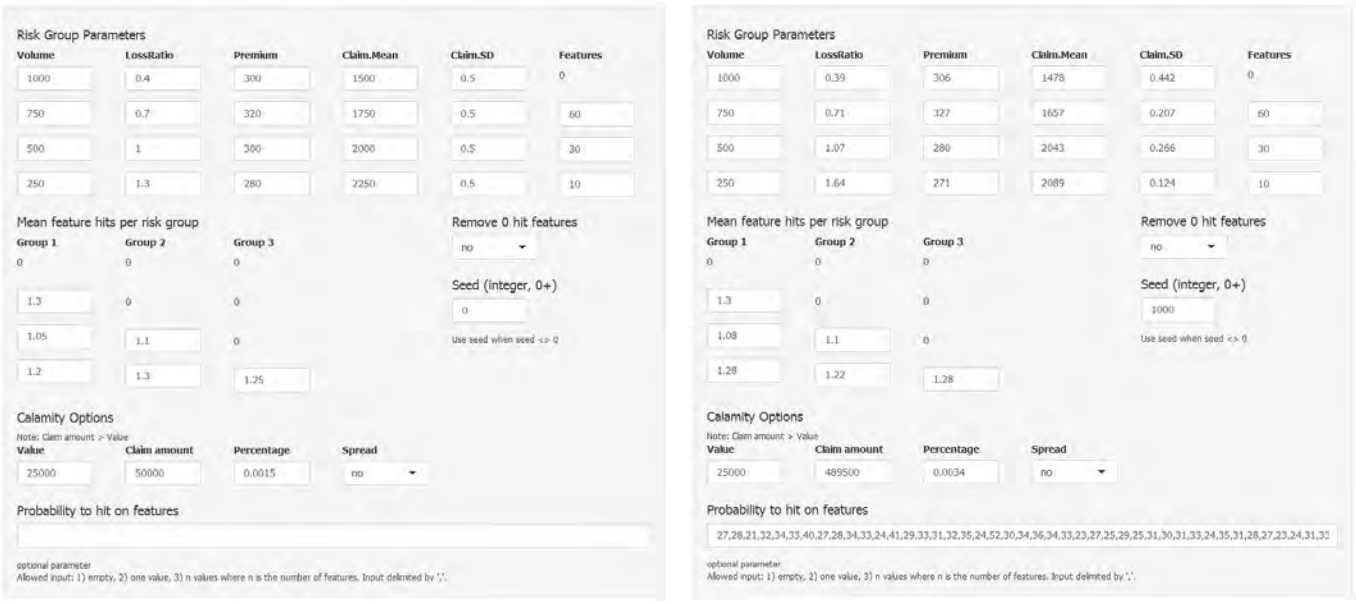


Figure 30: The left part of the Artificial Dataset Generator Tab. To the left are the pre-set parameters for the user input. To the right are the calculated parameters from the dataset created with the pre-set parameters and seed 1000.

Risk Group Parameters

The dataset generator generates cases based on minimum two and maximum 4 *risk groups*, groups 0 to 4. Each group is a set of cases, which combined form the artificial dataset. Each group is generated with its own set of parameters and features. The parameters that can be set for each risk groups are explained in Table 8.

Parameter	Explanation
Volume	The number of cases
Loss Ratio	The loss ratio
Premium amount	The premium income per case
Claim.Mean	The mean claim amount per case
Claim.SD	The standard deviation of the mean claim amount
Features	The number of features. The first risk group has no features.

Table 8: The parameters of each risk group, which are used by the dataset generator.

Mean feature hits per risk group

This group sets the mean number of feature hits for the risk groups. For example, take a look at the parameters in Figure 0. A case of risk group 2 it has on average 1.05 and 1.1 hits on features of risk groups 1 and 2 respectively. A case must always have at least one hit on the features of its own risk group.

Calamity Options

The parameter subgroup *Calamity Options* contains the options for calamity claims. These parameters can be used to add outliers, calamities, to the dataset. Table 9 gives a short overview of the parameters.
The first parameter is *Value* which is the minimum claim amount of any calamity case. The *Claim Amount* is the mean claim amount for the calamity cases. *Percentage* determines the percentage of cases with at least one claim to become a calamity case. Finally, the *Spread* parameter can be used to spread-out the claim amount of the calamity cases as is explained at section 4.1.2.

Parameters / Options	Explanation
Value	When a case has a claim amount is higher than this value, it is considered a calamity.
Claim Amount	The mean claim amount of generated calamity cases.
Percentage	The percentage of the cases with at least one claim that is a calamity case.
Spread	Spread the calamity claim amount over all other cases.

Table 9: The calamity case parameters and extra options for the dataset generator.

Single Parameters

Besides the standard parameters explained above, there are a couple of extra options, shown in Table 10. The first is *Remove 0 hit features*. When set to yes then if any feature is created without a case with a hit on it, then the feature is removed from the dataset. *Seed* sets the random number generator seed. This can be used create exactly the same dataset multiple times. *Probability to hit on features* allows to put in probabilities for cases to hit on specific features. A feature with probability 0.1 will hit, on average, twice as much than a feature with probability 0.05. Note that the probabilities do not need to add up to 1. The application takes care of normalizing the probabilities. When the parameter is empty, when no probabilities are given, every feature of a specific risk group is assigned the same probability.

Options	Explanation
Remove 0 hit features	Any features that have no case with a hit are removed from the dataset.
Seed	Use a seed to generate the dataset
Probability to hit on feature	Define the probability to hit on a specific feature. The probabilities are allowed not to add up to 1. (optional)

Table 10: The extra options for the dataset generator.

Creating, loading and saving datasets
The right side of the *Artificial Dataset Generator* tab, see Figure 31, contains options to generate a new artificial dataset, to load in an existing dataset, to obtain approximations of the parameters of the current dataset in the application and to save the current dataset. Also some statistics, like the total premium income and claim amount, of the current datasets in the application are shown.

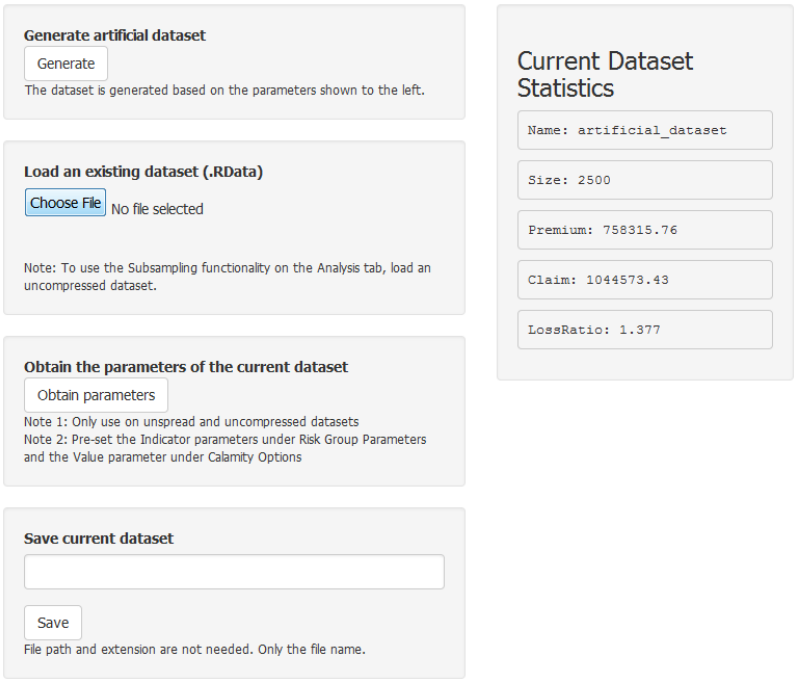


Figure 31: The right part of the Artificial Dataset Generator Tab. This side contains various options for creating and saving datasets, as well as some statistics of the current dataset in the application.

Generate artificial dataset generates a new artificial dataset based on the user input. *Load an existing dataset* allows for loading an existing dataset, with extension *.RData*, into the application. With *Obtain the parameters of the current dataset* an approximation of the parameters of the current dataset are returned. With these parameters we can create a similar dataset. However, the number of features at the parameter *Indicator* of the *Risk group parameters* in the user input must be filled out. The result of obtaining the parameters of an existing dataset is shown in the right figure of
Figure 1. Last, at *Save current dataset* you can enter a name or file path to save the dataset currently in the application.

5.1.2 Generating the parameters for the artificial dataset

Feature hits
When generating a new dataset the user input parameters are obtained from the application. First, the probabilities to hit on specific features are calculated. When the input parameter Probability to hit a knowledge rule is empty all features of a risk group are assigned the same chance, else each hit gets assigned a probability based on the input parameter. Also mean number of feature hits for each risk group is obtained from the user input. The probability is calculated based on the binomial distribution as each feature has its own probability to hit. The probability parameter for the binomial distribution is calculated by finding the root of the polynomial containing the coefficients of the binomial distribution. The found binomial probability outputs the average selected in the user input.

Number of claims
First, a *base value* for the lambda parameter of the Poisson distribution is calculated. This is done based on the average premium income and average claim amount per case. Then, taking into account the probability of the

features to hit, the *extra lambda value* for the Poisson distribution is calculated. For each feature hit the base value of the lambda parameter is increased by the extra lambda value. In this way the assumption about the relation between the number of feature hits and the number of claims is made. The more feature hits the higher the chance to have more claims.

Calamities
Finally, the lambda parameter for the exponential distribution for the claim amount of the calamities is calculated. To enforce that the calamity claim value is always high enough to create a calamity case, a lower bound of the calamity value is set used while generating the parameter. This lower bound is also used when generating the calamities in the artificial dataset.

5.1.3 Generating the artificial dataset

Feature hits
The feature hits are generated with the binomial distribution. Each feature has the probability to hit that was calculated as explained in section 9.1.1. As each risk group can have different probabilities to hit on a feature, the hits are generated for each group separately. As each case must have at least one feature hit on the risk group specific features of its risk group, each case without such a hit is randomly assigned a hit.

Premium income
The premium income is generated with the exponential distribution. The parameter used for the exponential distribution is $1/premium\ income$, where the premium income is taken from the user input. The premium income can be different for each risk group.

Number of claims and claim amount
The number of claims for each case is determined based on the Poisson distribution, as shown in section 9.1.1. First, for each feature hit the lambda parameter of the Poisson distribution is increased by the extra lambda value, to increase the chance of more claims.
For each claim that is generated for a case the claim amount will be drawn from a lognormal distribution.

Calamities
First, based on a binomial distribution the cases are determined that become calamity cases. Only the cases with at least one claim are considered. The claim amount of each chosen case is then increased by the calamity claim value drawn from the exponential distribution, with a minimum value of the calamity value.

5.2 Algorithm

On this tab the different meta-heuristic algorithms can be run and the best solution will be shown afterwards. Several options are available for running the algorithms. Figure 32 shows the options for the algorithms within the application.

To run an algorithm set the different options to your liking. First is the option *Algorithm*. Choose here one of the 5 algorithms available for the application, the 4 meta-heuristic algorithms and an algorithm that randomly draws solutions, section 6.1 covers this last algorithm. Then the *#Iterations* option to set the number of iterations, and *#Solutions*, the population size. The *#Offspring per solution* defines the number of offspring that each solution generates per iteration. *#Runs* is the number of times the algorithm is run. The 3 means the algorithm is run 3 times with the same basic parameters. *Maximum reject %* is the maximum percentage of the portfolio that is allowed to be rejected. *Cut-off value* is the FRISS score at which a case is rejected. *Manipulation chance* is the probability for each feature to be manipulated. *Step size* determines the granularity of the feature weights as

defined in section 3.1.6. The *Initial lower bound* and *Initial upper bound* determine the minimum and maximum feature weights for the initial solution(s). Last is the *Seed* which sets the seed of the random number generator when the chosen seed is not 0. You can select one or more seeds and the number of seeds does not need to equal the number of runs. Note that the CMA-ES algorithm used in the application is a slightly different version of the algorithm described in sections 3.2.4 and 4.3.4. The changes are due to added code for returning extra information about the generated solutions.

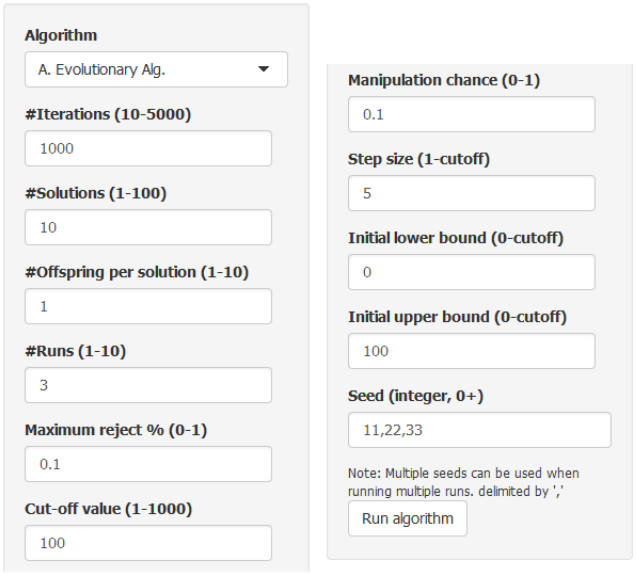


Figure 32: The left part of the Algorithm Tab. Here the various options for the algorithms can be chosen. In the application this is one column of options. Note that the "Step size" parameter is the granularity of the feature weights as described in section 3.1.6.

After an algorithm has been run the right side of the Algorithm Tab, see Figure 33, shows the best solution found under *Best generated solution* and the parameters of the last run under *Last run*. The lower part shows several graphs about the generated solutions and the objective value improvement of the generated solutions.

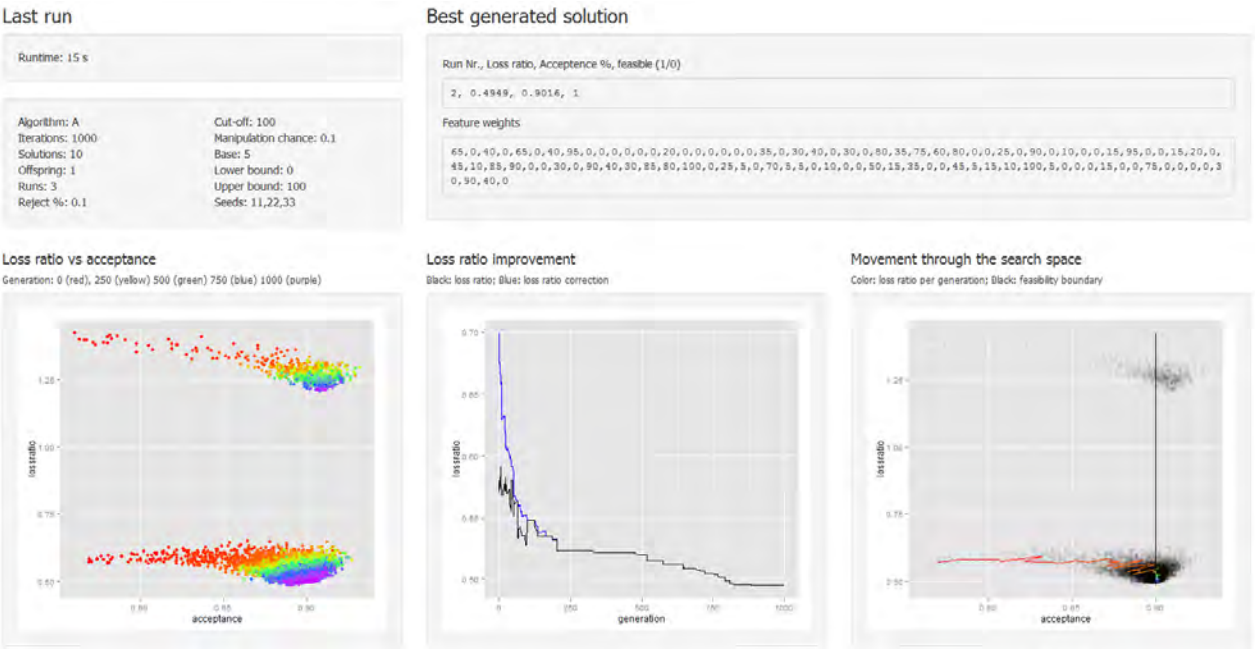


Figure 33: The right part of the Algorithm Tab. The last run of the algorithm can be viewed, as well as the best generated solution of the last run. The plots shows, from left to right, the solutions created, the improvement of the loss ratio (black) and the objective value of the solution (blue), and the movement of the best solution. In this last plot, the colour of the line shows the iteration in which the solution was generated.

5.3 Analysis

On the Analysis Tab the previous algorithm run on the Algorithm tab can be viewed. In the top right of the tab you can choose to which run to view or to view all runs together, to better compare the solutions to each other, regarding objective function improvement and movement.

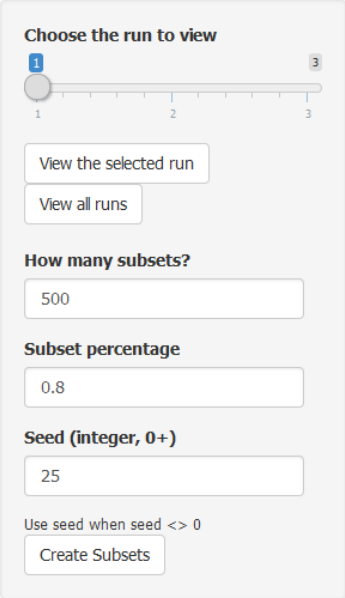


Figure 34: The left side of the Analysis Tab. Here the best solutions of each run can be viewed, as well as the best solutions of all runs together. Also you can see the objective values of the best solutions of the different runs by subsampling against the training set.

When you choose *View the selected run*, see Figure 34, the tab shows the best solution found in that specific run and also shows the plots about generated solutions, objective value and loss ratio improvement and movement through the search space, which are basically the same as the plots shown in Figure 31, but of the selected run.

When you select *View all runs* the plots change to what you can see in Figure 33. The first plot will be empty at first, but the second plot shows the fitness improvement of all runs, while the third plot shows the movement for all runs. The first plot can be filled by pressing the “Subset” button. With this button the application starts creating subsets of the data and calculates the fitness of the best weight vector on the subsets. With user input, see Figure 34, the number of subsets and the percentage of the dataset that must be in each set. Also the seed of the random number generator can be set. The working of subsetting the dataset is shown more in-depth at section 6.3.3.

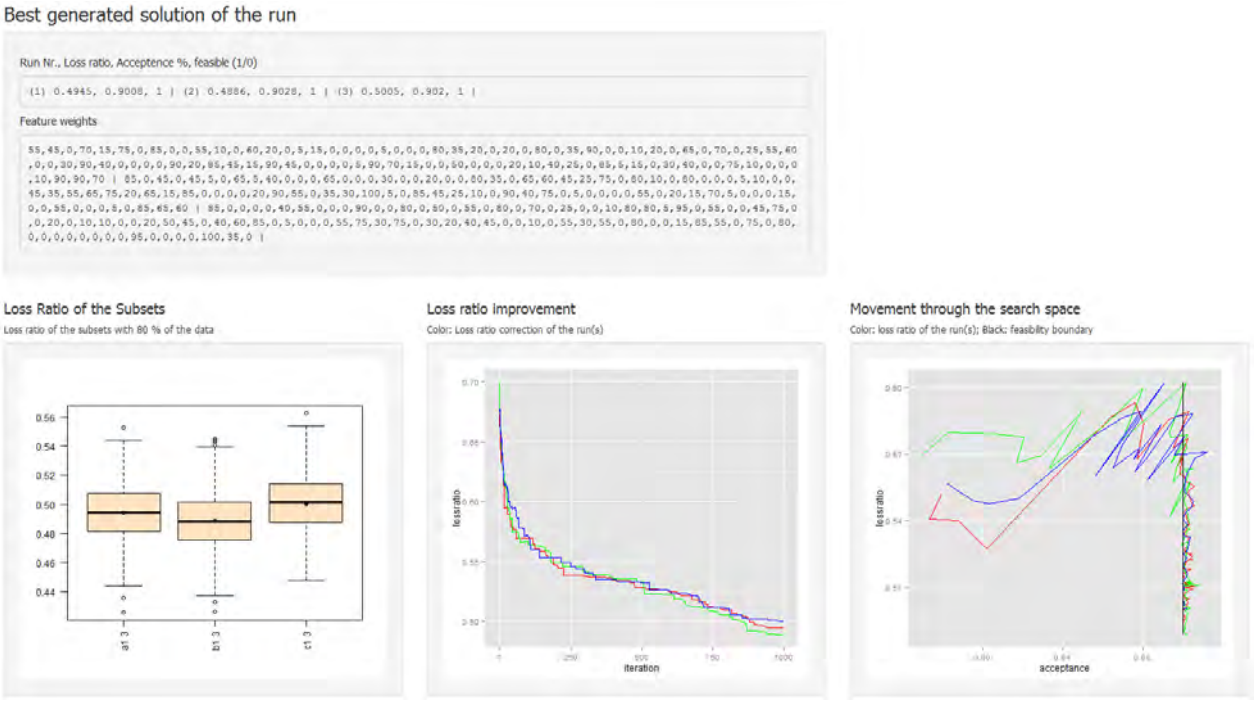


Figure 35: When both the "View all runs"- and "Create Subsets"-buttons shown in Figure 34 are pressed, then the first plot shows a boxplot containing the loss ratio over the subsets. The second plot shows the objective value improvement of the runs and the third plot shows the change in the loss ratio and acceptance percentage of the best solution of each run.

6 Results

6.1 Experiment Protocol

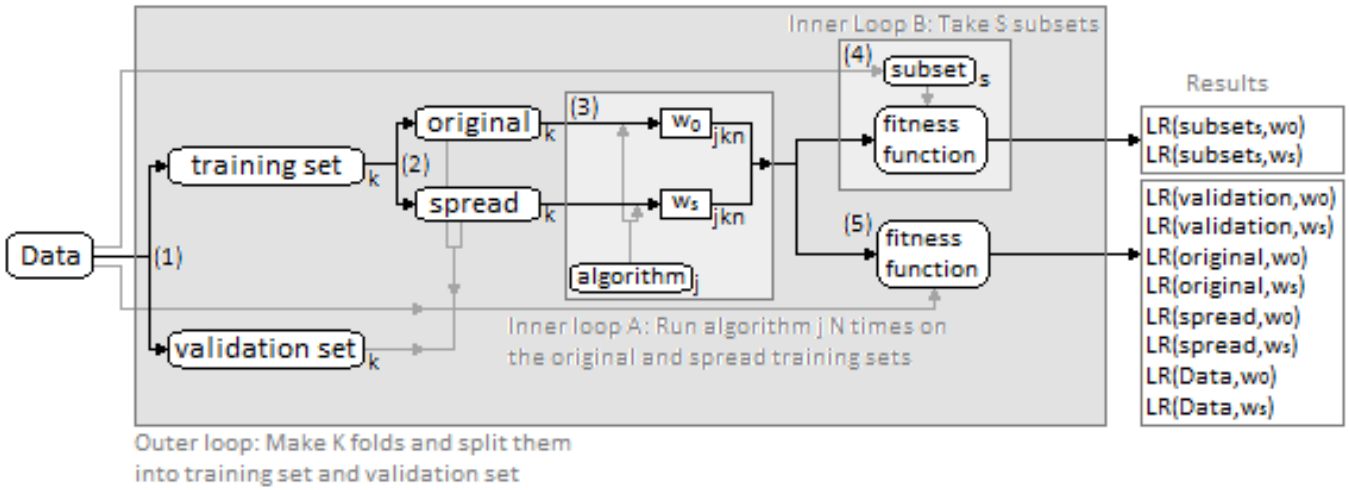


Figure 36: Experiment Protocol: (1) In the outer loop the data is divided K times into a training set and a validation set. (2) From the training set, the original set, a second training set, the spread set, is created by spreading the calamities in the training set. (3) In the first inner loop the J algorithms are run N times on the two training sets, generating weight vectors. (4) The weight vectors are tested against S subsets of the full data. (5) The weight vectors are tested against four datasets: the full dataset, the two training sets and the validation set.
In total K*2*J*N weight vectors and K*2*J*N*(S+4) results are generated for one dataset.

In Figure 36, the Experiment Protocol is shown. This is the schema we use to produce all results in this thesis. We will use a master script to produce all results. This script is not included in this thesis, but comparable results can be generated with the application shown in chapter 5 and the parameter values shown in chapter 9. For each part of the Experiment Protocol pre-defined seeds are used.

In the *Outer Loop* each dataset is processed and is divided into K folds. Each fold is used $K-1$ times in the *training set* and once as *validation set*, (1). From the training set, basically the *original* training set, a second set is generated, (2). This set, the *spread* training set, is the original training set with the calamities spread out, as explained in section 4.1.2. As calamities are considered all cases with a claim amount higher than 25.000 euros.

Then, in the first *Inner Loop*, (3), each of the J algorithms is run N times on both the original and spread training sets created by (2). Each algorithm run generates one weight vector for each training set.

Then in the second *Inner Loop*, (4), the generated weight vectors will be tested against *S* subsets, each containing 80% of the full dataset, to determine how well the weight vector performs when tested against a dataset consisting of part training data and part unknown data which is in the validation set.

The weight vectors are then put against four datasets, namely the original and spread training sets, the validation set and the full dataset, determining the objective value of the weight vector, on each dataset, (5). The results of the weight vectors can be paired together based on fold, algorithm and training set. In this manner a clean performance comparison can be made to determine de effects of the training set and algorithm used.

6.1.1 Algorithms, datasets and parameters

Five algorithms, $J = 5$, will compete, these are the four meta-heuristic algorithms shown in chapter 4.3, and an algorithm that basically draws random solutions. This last algorithm draws random feature weights from the search space with a weight granularity of 5. For this random algorithm, we also tested with granularities of 1 and 10, but a granularity of 5 generally returned the best solutions. The data of this test is not shown in this thesis.

For each algorithm 10.000 objective value evaluations are processed, except for the random algorithm which will do 100.000 objective value evaluations. The idea is that when fast drawing of random solutions will easily yield a good solution, than meta-heuristic algorithms are unnecessary to solve the problem.

We will use eight different datasets, namely two real datasets, the same real datasets randomized, see section 4.1.3, and four artificial datasets, see section 4.1.4. In section 6.1.2 the scenario's behind the different artificial datasets is explained and section 9.1 contains the generation parameters of these datasets.

The Outer Loop will divide each dataset into $K = 5$ folds and each algorithm in the first Inner Loop will do $N = 5$ runs. In total $K * 2 * J * N = 5 * 2 * 5 * 5 = 250$ weight vectors will be generated for each dataset. The second Inner Loop will take $S = 100$ subsets of the data. In total $8 * 250 * 4 = 8.000$ weight vectors and $8 * 250 * 100 = 200.000$ subset results will be generated.

In section 9.2 the specific parameters of each run are shown. Though all meta-heuristic algorithms will do the same number of objective value evaluations, not all algorithms will have the same number of iterations, solutions and solutions in the neighbourhood.

6.1.2 Artificial datasets

Increasing Risk
Each risk group, as described in section 4.1.4, has a slightly higher risk than the previous risk group. All risk groups have the same mean claim amount per claim and premium income, as well as the same average number of feature hits on the risk group specific features of each risk group. However, each risk group has a higher loss ratio, which immediately means a higher risk for the insurer. To make up for the higher risk, the risk groups have a decreasing number of cases and risk group specific features. This dataset is created as somewhat ideal portfolio. You have a large amount of low risk cases and a small amount of high risk cases, where the risk increases slowly with each risk group.

Small Calamities
The second dataset almost has the same parameters as the first dataset. However, the dataset contains 15 times more calamity cases, 19 and 291 respectively, but with a lower average calamity claim amount, 42.000 versus 30.000 euros. This dataset represents data with a large amount of small local optima, equivalent to a portfolio with a large number of cases with higher than average claim amounts.

High Risk
The third dataset is a high risk dataset. $2/3^{\text{rd}}$ of the cases are in risk groups with high loss ratio and claim amount, while the average premium income is decreases each risk group. This can be seen as people filing high value claims, but may default on paying their insurance premium.

Medium Risk
The last dataset has medium risk with half of the cases are in the second risk group. However, despite having a low number of calamities, the calamities have a high calamity claim amount. This dataset follows a scenario where we have low risk cases, but when a calamity occurs there are extensive costs. Think of a car accident with bodily injury.

The exact generation parameters and the statistics of the artificial datasets are shown in section 9.1. Also the artificial dataset generator parameters, that can be used to obtain an approximation of the client datasets and randomized client datasets, are available there.

6.2 Results

The results obtained from the experiments of section 6.1 are shown in Tables 11, 12 and 13. In Table 11 the average results of the different algorithms on the datasets are shown. On the rows of the table the results of the algorithms are shown and the results are divided between the results on the training set (*T*), the validation set (*V*), the whole dataset (*D*) and the whole dataset with spread-out calamities (*Ds*). These results are the average loss ratio (*Lr*) and average acceptance percentage (*Acc*) over 25 runs, namely 5 folds of each dataset and 5 algorithm runs per fold. Each column contains the average results of the found weight vectors on the datasets. These results are divided into two groups, based on if the calamities of the training set were spread-out (*yes*) or not (*no*).

We can see that generating feasible solutions for the High Risk dataset is harder than for most other datasets. Also the random draw algorithm has troubles generating feasible solution on some datasets.

		A	Real A		Real B		Incr. Risk		Small Calamities		High Risk		Med. Risk		Real A Random.		Real B Random.	
B	C	D\E	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*
EA	*Lr*	*T*	0.56	0.62	0.66	0.67	0.58	0.58	0.69	0.70	0.95	0.98	0.89	0.99	0.67	0.69	0.68	0.68
		V	0.63	0.65	0.69	0.69	0.59	0.59	0.72	0.72	1.01	1.01	1.03	1.01	0.74	0.74	0.70	0.70
		D	0.58	0.61	0.66	0.68	0.58	0.58	0.70	0.70	0.96	0.98	0.92	0.98	0.69	0.69	0.68	0.68
		Ds	0.63	0.65	0.68	0.69	0.58	0.58	0.70	0.70	0.99	0.99	1.00	0.99	0.70	0.70	0.69	0.69
	Acc	*T*	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90
		V	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90
		D/Ds	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90
TS	*Lr*	*T*	0.58	0.63	0.67	0.68	0.57	0.57	0.68	0.69	0.95	0.97	0.95	0.98	0.71	0.72	0.68	0.69
		V	0.62	0.66	0.69	0.69	0.59	0.59	0.72	0.72	0.99	0.99	1.01	1.01	0.73	0.73	0.71	0.70
		D	0.59	0.63	0.67	0.68	0.58	0.58	0.69	0.69	0.96	0.96	0.96	0.97	0.72	0.72	0.69	0.69
		Ds	0.63	0.66	0.68	0.69	0.58	0.58	0.70	0.70	0.98	0.98	0.99	0.99	0.72	0.72	0.69	0.69
	Acc	*T*	0.90	0.90	0.93	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.98	0.97	0.95	0.94
		V	0.90	0.90	0.93	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.98	0.97	0.95	0.94
		D/Ds	0.90	0.90	0.93	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.98	0.97	0.95	0.94
CMAES	*Lr*	*T*	0.62	0.63	0.68	0.69	0.57	0.57	0.69	0.69	0.71	0.71	0.92	0.98	0.72	0.72	0.69	0.69
		V	0.64	0.65	0.69	0.69	0.59	0.59	0.71	0.71	0.74	0.73	1.01	1.02	0.74	0.74	0.71	0.70
		D	0.63	0.63	0.68	0.68	0.58	0.58	0.69	0.69	0.72	0.71	0.94	0.98	0.72	0.72	0.69	0.69
		Ds	0.66	0.67	0.69	0.69	0.58	0.58	0.70	0.70	0.73	0.72	0.99	0.99	0.72	0.73	0.70	0.70
	Acc	*T*	0.90	0.90	0.93	0.90	0.90	0.90	0.90	0.90	0.64	0.62	0.90	0.90	0.95	0.94	0.93	0.93
		V	0.90	0.90	0.93	0.90	0.90	0.90	0.90	0.90	0.64	0.62	0.90	0.90	0.95	0.94	0.93	0.93
		D/Ds	0.90	0.90	0.93	0.90	0.90	0.90	0.90	0.90	0.64	0.62	0.90	0.90	0.95	0.94	0.93	0.93
PSO	*Lr*	*T*	0.56	0.62	0.65	0.67	0.57	0.58	0.65	0.70	0.54	0.54	0.89	0.98	0.68	0.70	0.68	0.68
		V	0.62	0.64	0.68	0.69	0.59	0.59	0.71	0.72	0.60	0.59	1.02	1.01	0.74	0.73	0.70	0.70
		D	0.57	0.60	0.66	0.67	0.58	0.58	0.69	0.70	0.55	0.53	0.92	0.97	0.69	0.70	0.68	0.68
		Ds	0.63	0.65	0.68	0.69	0.58	0.58	0.70	0.70	0.58	0.55	1.00	0.99	0.71	0.71	0.69	0.69
	Acc	*T*	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.49	0.46	0.90	0.90	0.91	0.91	0.91	0.91
		V	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.49	0.46	0.90	0.90	0.91	0.91	0.91	0.91
		D/Ds	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.90	0.49	0.46	0.90	0.90	0.91	0.91	0.91	0.91
Rand.	*Lr*	*T*	0.60	0.64	0.69	0.70	0.56	0.56	0.68	0.68	0.58	0.59	0.81	0.91	0.72	0.73	0.70	0.71
		V	0.61	0.67	0.69	0.69	0.56	0.56	0.69	0.68	0.61	0.60	0.91	0.88	0.73	0.73	0.70	0.70
		D	0.60	0.66	0.69	0.69	0.56	0.56	0.68	0.68	0.59	0.59	0.83	0.87	0.72	0.73	0.70	0.70
		Ds	0.64	0.68	0.70	0.70	0.56	0.56	0.68	0.68	0.61	0.59	0.90	0.91	0.73	0.73	0.71	0.71
	Acc	*T*	0.83	0.87	0.93	0.90	0.82	0.82	0.82	0.82	0.46	0.45	0.54	0.58	0.88	0.88	0.93	0.93
		V	0.83	0.87	0.93	0.90	0.82	0.82	0.82	0.82	0.46	0.45	0.54	0.58	0.88	0.88	0.93	0.93
		D/Ds	0.83	0.87	0.93	0.90	0.82	0.82	0.82	0.82	0.46	0.45	0.54	0.58	0.88	0.88	0.93	0.93

Table 11: The mean loss ratio and acceptance percentage of each algorithm of the training, validation and full datasets for the weight vectors generated on the original and spread training sets. (A) the dataset that was used. (B) The algorithm used for training. (C) The results are split in: *Lr = loss ratio* and *Acc = acceptance* percentage. (D) The dataset where the result was obtained from: *T = training set*, *V = validation set*, *D = full dataset and Ds = full dataset with calamities spread-out*. (E) Was the weight vector trained on the spread training set (yes) or not (no)?

Table 12 shows the average results of subsetting the datasets and then calculating the objective value of the found weight vectors. The rows of the table show the average results of the algorithms. These results are divided into results over the whole dataset (*D*) and the whole dataset with spread out calamities (*Ds*), which are divided again into the average loss ratio (*Lr*) and the standard deviation of the loss ratios (*Std*). Also included is the average feasibility of the weight vectors over the datasets. The results are taken over 100 subsets of both datasets.
The columns each contains the results on a specific dataset and if the calamities of the training set that produced the weight vector were spread-out (*yes*) or not (*no*).

		A	Real A		Real B		Incr. Risk		Small Calamities		High Risk		Med. Risk		Real A Random.		Real C Random.	
B	C	D\E	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*
EA	*D*	*Lr*	0,58	0,61	0,66	0,68	0,58	0,58	0,70	0,70	0,96	0,98	0,92	0,98	0,69	0,69	0,68	0,68
		Std	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,02	0,03	0,01	0,01	0,01	0,01
	Ds	*Lr*	0,63	0,65	0,68	0,69	0,58	0,58	0,70	0,70	0,99	0,90	1,00	0,99	0,70	0,70	0,69	0,69
		Std	0,01	0,01	0,01	0,01	0,00	0,00	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01
	D/Ds	*Feas*	0,67	0,59	0,73	0,43	0,56	0,61	0,63	0,58	0,59	0,53	0,68	0,75	0,93	0,82	0,87	0,91
TS	*D*	*Lr*	0,59	0,63	0,67	0,68	0,57	0,58	0,69	0,69	0,96	0,96	0,96	0,97	0,72	0,72	0,69	0,69
		Std	0,01	0,01	0,01	0,01	0,01	0,00	0,01	0,01	0,01	0,01	0,03	0,03	0,01	0,01	0,01	0,01
	Ds	*Lr*	0,63	0,66	0,68	0,69	0,58	0,58	0,70	0,70	0,98	0,98	0,99	0,99	0,72	0,72	0,69	0,69
		Std	0,01	0,01	0,01	0,01	0,00	0,00	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01
	D/Ds	*Feas*	0,74	0,62	0,97	0,57	0,67	0,65	0,57	0,59	0,80	0,68	0,80	0,73	1,00	1,00	1,00	0,99
CMAES	*D*	*Lr*	0,63	0,63	0,68	0,68	0,58	0,57	0,69	0,69	0,72	0,71	0,94	0,98	0,72	0,72	0,69	0,69
		Std	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,02	0,02	0,03	0,03	0,01	0,01	0,01	0,01
	Ds	*Lr*	0,66	0,67	0,69	0,69	0,58	0,58	0,70	0,70	0,73	0,72	0,99	0,99	0,72	0,73	0,70	0,70
		Std	0,01	0,01	0,01	0,01	0,00	0,00	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01
	D/Ds	*Feas*	0,87	0,62	1,00	0,55	0,62	0,59	0,66	0,61	0,00	0,00	0,74	0,59	1,00	1,00	1,00	1,00
PSO	*D*	*Lr*	0,58	0,60	0,66	0,67	0,58	0,58	0,69	0,70	0,55	0,53	0,92	0,97	0,69	0,70	0,68	0,68
		Std	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,02	0,02	0,02	0,03	0,01	0,01	0,01	0,01
	Ds	*Lr*	0,63	0,65	0,68	0,69	0,58	0,58	0,70	0,70	0,58	0,55	0,10	0,99	0,71	0,71	0,69	0,69
		Std	0,01	0,01	0,01	0,01	0,00	0,00	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01
	D/Ds	*Feas*	0,74	0,58	0,89	0,47	0,65	0,56	0,59	0,58	0,00	0,00	0,50	0,53	1,00	0,95	0,95	0,99
Rand.	*D*	*Lr*	0,60	0,66	0,69	0,69	0,56	0,56	0,68	0,68	0,59	0,58	0,84	0,97	0,72	0,73	0,70	0,70
		Std	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,02	0,02	0,03	0,03	0,01	0,01	0,01	0,01
	Ds	*Lr*	0,64	0,68	0,70	0,70	0,56	0,56	0,69	0,68	0,61	0,59	0,90	0,91	0,73	0,73	0,71	0,70
		Std	0,01	0,01	0,01	0,01	0,00	0,00	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01
	D/Ds	*Feas*	0,00	0,00	1,00	0,60	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,96	1,00

Table 12: The mean of the results of testing the weight vectors against 100 subsets of the data. (A) The dataset that was used. (B) The algorithms that produced the weight vectors. (C) Were the subsets made from *D* = the full dataset or from *Ds* = the full dataset with calamities spread-out. (D) The type of result: *Lr* = the mean loss ratio on the subsets. *Std* = the standard deviation of the loss ratios of the subsets. *Feas* = the percentage of subsets on which the weight vector returned a feasible solution. (E) Was the weight vector trained on the spread training set (yes) or not (no)?

Table 13 shows the average runtime in seconds of a specific algorithm over a specific dataset. The average is taken over all runs of the algorithm over the specific dataset, regardless if the training set was spread-out and the current fold of the dataset. This makes an average of 50 runs for each time result.

We can immediately see that of the meta-heuristic algorithms, our implementation of the Tabu Search algorithm is the fastest, while the implementation of the Particle Swarm Optimization algorithm is very slow.

	Time (in seconds)				
Dataset	**EA**	**TS**	**CMAES**	**PSO**	**Random**
Real A	16	12	28	108	39
Real B	4	3	6	20	12
Increasing Risk	48	45	75	411	140
Small Calamities	47	44	75	393	142
High Risk	72	69	121	610	264
Medium Risk	92	89	154	789	291
Real A Randomized	15	12	27	104	38
Real B Randomized	4	3	6	19	11

Table 13: The average runtime per run of a specific algorithm on a specific dataset. The runtime is in seconds. The average runtime is taken over 50 runs.

6.3 Evaluation Options

6.3.1 Loss Ratio Reduction

The most important evaluation option is the reduction of the loss ratio. Can the algorithms find solutions that greatly improve, read decrease, the loss ratio by rejecting cases? For example, we have two algorithms. The first algorithm reduces the loss ratio by 0.03 and the second algorithm reduces the loss ratio by 0.10. In this case the second algorithm clearly performs better.

6.3.2 Objective Value Improvement

Figure 37 shows three different objective value improvement curves from different algorithms. All curves have the same initial and final objective value. However, clear difference can be seen in the speed the objective value is improved. Curve A shows the curve of an algorithm that improves the objective value by much in the first few iterations and then slows down. The algorithm of curve B improves the objective value in large steps after a couple of generations. Finally, the algorithm that created curve C shows a slow objective value improvement in the first and last few generations.

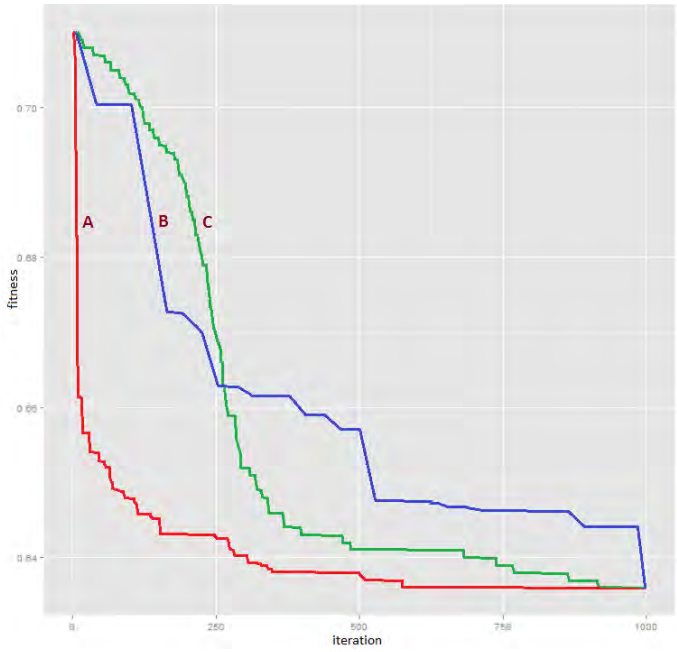


Figure 37: Three different objective value curves. All curves have the same initial and final fitness. However, curve A improves the objective value by much in the first few generations and then slows down. Curve B improves the objective value in large steps when it finds a better loss ratio. Curve C improves slowly the first few generations, then speeds up and slows down again.

All three curves have advantages and disadvantages. Curve A's advantage is the rapid improvement of the objective value, finding a good solution in a short time. However, the algorithm might get stuck in a local optimum when the algorithm focusses on a too small search area. Curve B improvement by large steps is nice, but the algorithm seems to have trouble to continuously find solutions with better objective value. Curve C is steadily improving the objective value by small steps at the first few iterations and possibly exploring large portions of the search space. However, when the steps are to small it might not find the best solution before it reaches the final iteration.

6.3.3 Time and Number of Iterations

The time and number of iterations before a feasible and/or acceptable solution is generated is important. As long as the algorithm has not found feasible solutions, the objective function restrains the improvement of the loss ratio by the algorithm, because of the penalty given to infeasible solutions, see section 4.2. Algorithms are forced to accept solutions with higher loss ratio's in order to find feasible solutions.

For this reason alone, we would like to see an algorithm that already starts generating feasible solutions in the first few iterations, because then the algorithm has more iterations to improve the loss ratio of the feasible solutions.

Also, when the algorithm generates feasible and good solutions faster, fewer iterations are needed to obtain an acceptable solution, which directly reduces the total runtime. However, comparing the runtime of different algorithms is difficult, as the algorithms can have e.g. a not-optimal implementation and a different number of calculations that must be performed by the algorithm.

6.3.4 Subset Results

In the Experiment Protocol subsets of the data are generated to test the fit of a specific weight vector on the data. The data of the subset is drawn from the full dataset, see Figure 38 for an example, and that means that each subset contains cases from the original training set and the validation set. When a weight vector has a highly variable loss ratio and/or acceptance percentage over the different subsets, this might be a sign that the

algorithm generated a weight vector that has an overfit or a generally bad fit on the training set. However, when the variability is low the weight vector could provide a *good* indication of the dataset.

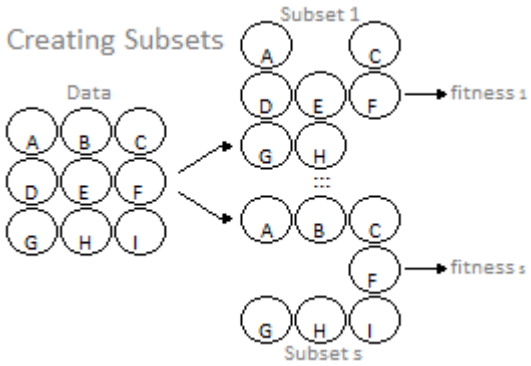


Figure 38: We create subsets from the Data by leaving certain cases out of the Data for each subset. E.g. subset 1 has left out cases B and I, while subset s left out D and E. For each sample you calculate the objective value against the weight vectors.

6.3.5 Distribution of Solutions

The manner the loss ratio and acceptance percentage of the generated solutions are distributed, can give important information about the algorithm's performance. Figure 39 shows three different distributions of solutions. First, there is (A), which shows the Random Draw algorithm. The solutions are spread over a large area and the solutions aren't focussed on one particular area. Then (B), the Evolutionary Algorithm. The solutions of the first few iterations are, just like (A), distributed over a large area. However, the algorithm generates solutions with more similar loss ratio and acceptance percentage after more iterations have been done. Finally, (C), the CMA-ES algorithm. This algorithm generates for each iteration solutions with similar loss ratio keeping the solutions close together and mostly creating a narrow line of solutions.

The unfocussed manner of generating solutions of (A) might find good solutions when the generated solutions are feasible. However, when this is not the case then none of the generated solutions will be any good. As (B) has a wide solution distribution in the early iterations it might cover a large portion of the search space. After which it starts focussing on the most promising solutions. (C) has a good chance of finding better solutions in the current neighbourhood of the current solution, but has a higher chance of getting stuck in a local optimum as it only searches close by.

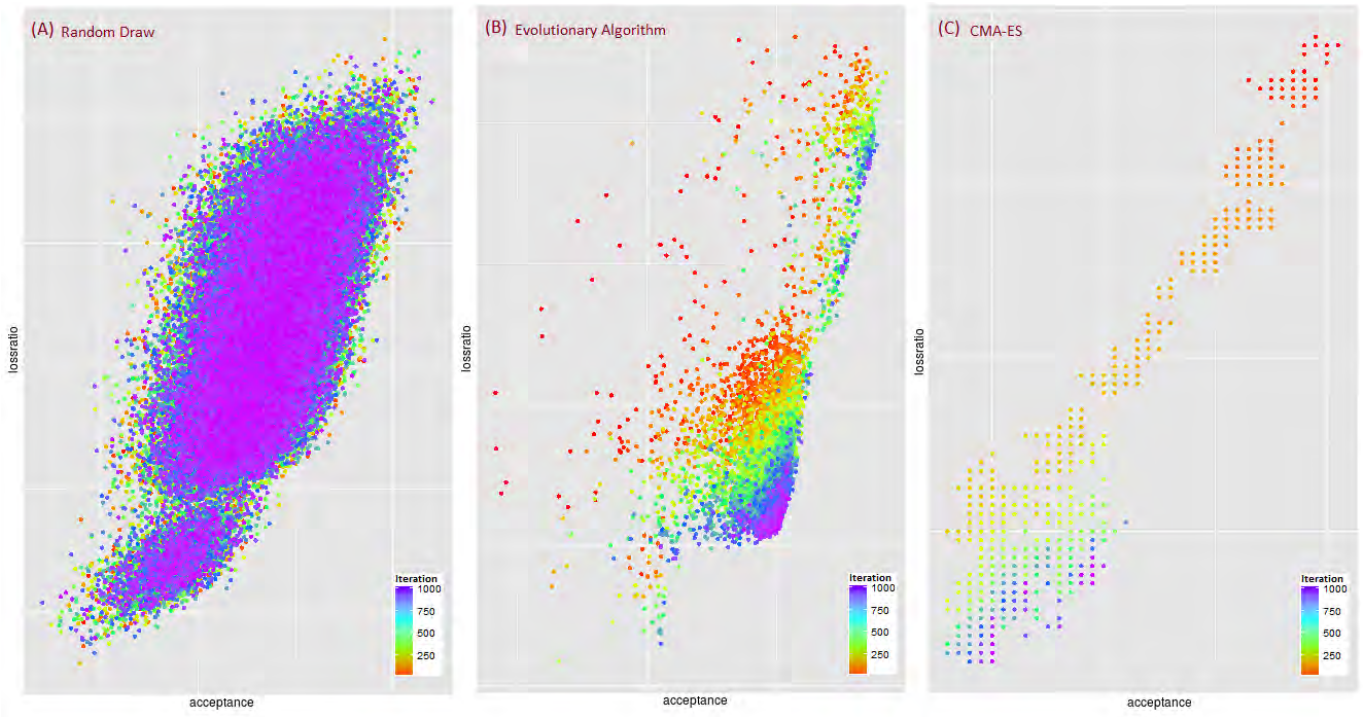


Figure 39: The generated solutions of (A) Random Draw, (B) Evolutionary Algorithm and (C) CMA-ES on the same dataset. Each algorithm shows a very different distribution of points. (A)'s distribution of points is unfocussed and spread over a large area, while (B)'s distribution has the solutions closer and closer together, slowly focussing onto one small area as more iterations are executed. However, (C) distributes its solutions very close together, creating a narrow line.
Note that the x- and y-axis are different. However, this does not have any impact on the shown distributions. Also (B) has the initial solutions located to the left of the feasibility boundary, while (C) starts on the right. Again, this does not change the specific manner the solutions are distributed.

6.3.6 Stability of Solutions

The stability of the solutions is about a single question *"Do the final solutions of multiple runs of the same algorithm look similar?"*, i.e. that specific features have similar weights in weight vectors obtained from running an algorithm multiple times on the same dataset. An algorithm that returns similar weight vectors every run might be more reliable to find good solutions, than an algorithm that returns a very different weight vector each run. Table 14 shows 4 weight vectors, A to D, generated by two different algorithms. Algorithm 1 finds similar weight vectors, A and B, while algorithm 2 finds different weight vectors, C and D.
However, the stability of the found weight vectors might say less about the algorithm than about the problem itself. The search space could have multiple local and global optima with basically the same loss ratio and acceptance percentage. Therefore, it is mostly based on preference, if we wish to use an algorithm which returns similar weight vectors or very different ones.

Algorithm 1	**A**	30	40	0	90	100	30	10
	B	20	40	5	100	85	30	15

Algorithm 2	**C**	35	30	0	90	95	30	20
	D	0	70	40	30	100	75	0

Table 14: Algorithm 1 generated weight vectors A and B, which have more similar feature weights than weight vectors C and D, which were generated by algorithm 2. When the generated weight vectors of an algorithm are stable the algorithm returns similar weight vectors every run, while an unstable algorithm returns different weight vectors. Note that the loss ratio and acceptance percentage can be exactly the same.

6.4 Analysis

6.4.1 Training and Validation Results

The average loss ratio's in Table 11, see section 6.2, show that the weight vectors trained by the meta-heuristic algorithms on the training set with spread-out calamities have never a lower average loss ratio. Also, the acceptance percentage is in almost every case the same. However, the difference between the loss ratio on the training set and the validation set is generally larger for the weight vectors trained on the original training set, see Table 15. This could mean that although the weight vectors trained on datasets with spread-out calamities usually have a higher loss ratio on the training set, the loss ratio is closer to the loss ratio found on the validation set.

A	**Real A**		**Real B**		**Increasing Risk**		**Small Calamities**		**High Risk**		**Medium Risk**		**Real A random.**		**Real B random.**	
B \ C	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*
EA	0.07	0.03	0.03	0.02	0.01	0.01	0.03	0.02	0.06	0.03	0.14	0.02	0.07	0.05	0.02	0.02
TS	0.04	0.03	0.02	0.01	0.02	0.02	0.04	0.03	0.04	0.02	0.06	0.03	0.02	0.01	0.03	0.01
CMAES	0.02	0.02	0.01	0.00	0.02	0.02	0.02	0.02	0.03	0.02	0.09	0.04	0.02	0.02	0.02	0.01
PSO	0.06	0.02	0.03	0.02	0.02	0.01	0.06	0.02	0.06	0.05	0.13	0.03	0.06	0.03	0.02	0.02
Random	0.01	0.03	0.00	-0.01	0.00	0.00	0.01	0.00	0.03	0.01	0.10	-0.03	0.01	0.00	0.00	-0.01

Table 15: The differences between the average validation set loss ratio and the average training set loss ratio ($Lr_V - Lr_T$).We can see that, except on the Real B dataset, the difference of the average loss ratio's is smaller for weight vectors trained on a spread training set than on the original training set. (A) the dataset that was used. (B) The algorithm used for training. (C) Was the weight vector trained on the spread training set (yes) or not (no)?

Table 16 shows the standard deviation of the loss ratio of the 25 generated weight vectors on the different datasets. The validation set generally has a larger standard deviation than the training set. The influence of a single case, e.g. a calamity, in the validation set is higher than that in the training set, because of the 80/20 split of the training and validation sets. The standard deviation of the weight vectors trained on the original training set and spread-out training set can be considered equal.

B	A / C \ D	Real A		Real B		Increasing Risk		Small Calamities		High Risk		Medium Risk		Real A random.		Real B random.	
		no	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*
T	**EA**	0,01	0,01	0,01	0,01	0,00	0,00	0,01	0,01	0,01	0,00	0,02	0,03	0,01	0,01	0,01	0,01
	TS	0,01	0,01	0,01	0,01	0,00	0,00	0,01	0,01	0,01	0,00	0,03	0,02	0,01	0,01	0,01	0,01
	CMAES	0,02	0,02	0,01	0,01	0,00	0,00	0,01	0,01	0,02	0,01	0,03	0,02	0,01	0,01	0,01	0,01
	PSO	0,01	0,01	0,01	0,01	0,00	0,00	0,01	0,01	0,03	0,03	0,03	0,03	0,01	0,01	0,01	0,01
	Random	0,03	0,02	0,01	0,01	0,01	0,01	0,01	0,01	0,05	0,03	0,05	0,03	0,02	0,01	0,01	0,01
V	**EA**	0,04	0,03	0,04	0,05	0,02	0,02	0,03	0,03	0,03	0,02	0,11	0,12	0,05	0,05	0,04	0,04
	TS	0,03	0,03	0,05	0,05	0,02	0,02	0,03	0,03	0,03	0,03	0,11	0,12	0,04	0,05	0,05	0,04
	CMAES	0,04	0,04	0,04	0,04	0,02	0,02	0,03	0,03	0,04	0,04	0,11	0,11	0,04	0,05	0,04	0,04
	PSO	0,03	0,03	0,04	0,05	0,02	0,02	0,03	0,03	0,05	0,06	0,12	0,11	0,05	0,05	0,04	0,04
	Random	0,05	0,05	0,05	0,05	0,02	0,02	0,04	0,04	0,08	0,08	0,16	0,14	0,04	0,05	0,04	0,04

Table 16: The standard deviation of the loss ratio's found over the different folds and runs. Each standard deviation is done over 25 runs. On most training sets and validation sets the standard deviation of the loss ratio's found on the spread-out training set is equal or lower than the standard deviation of the loss ratio's found on the original training set. (A) The dataset that was used. (B) The training set (*T*) or validation set (*V*). (C) The algorithm used for training. (D) Was the weight vector trained on the spread training set (*yes*) or not (*no*)?

Figure 40 shows the loss ratio and acceptance percentage of all runs on four of the datasets, namely "Real A", "Increasing Risk", "Medium Risk" and "Real A Randomized". The runs are divided by algorithm and if the training sets were spread-out or not. As we could see at Table 15, the loss ratio of the weight vectors trained on the spread-out training set are higher. Also the difference of the acceptance percentages, shown in Table 16, are clearly visible. The loss ratios of the training sets are much closer together than the validation set's loss ratios. Although the loss ratio values differ greatly, the acceptance percentages on the training and validation sets are almost equal.

We can also see that the algorithms might overfit on the Medium Risk dataset, as the variance of the validation loss ratios is so much higher than the variance of the training loss ratios. On the other hand, the weight vectors generated on the Increasing Risk dataset seem to have a very good fit.

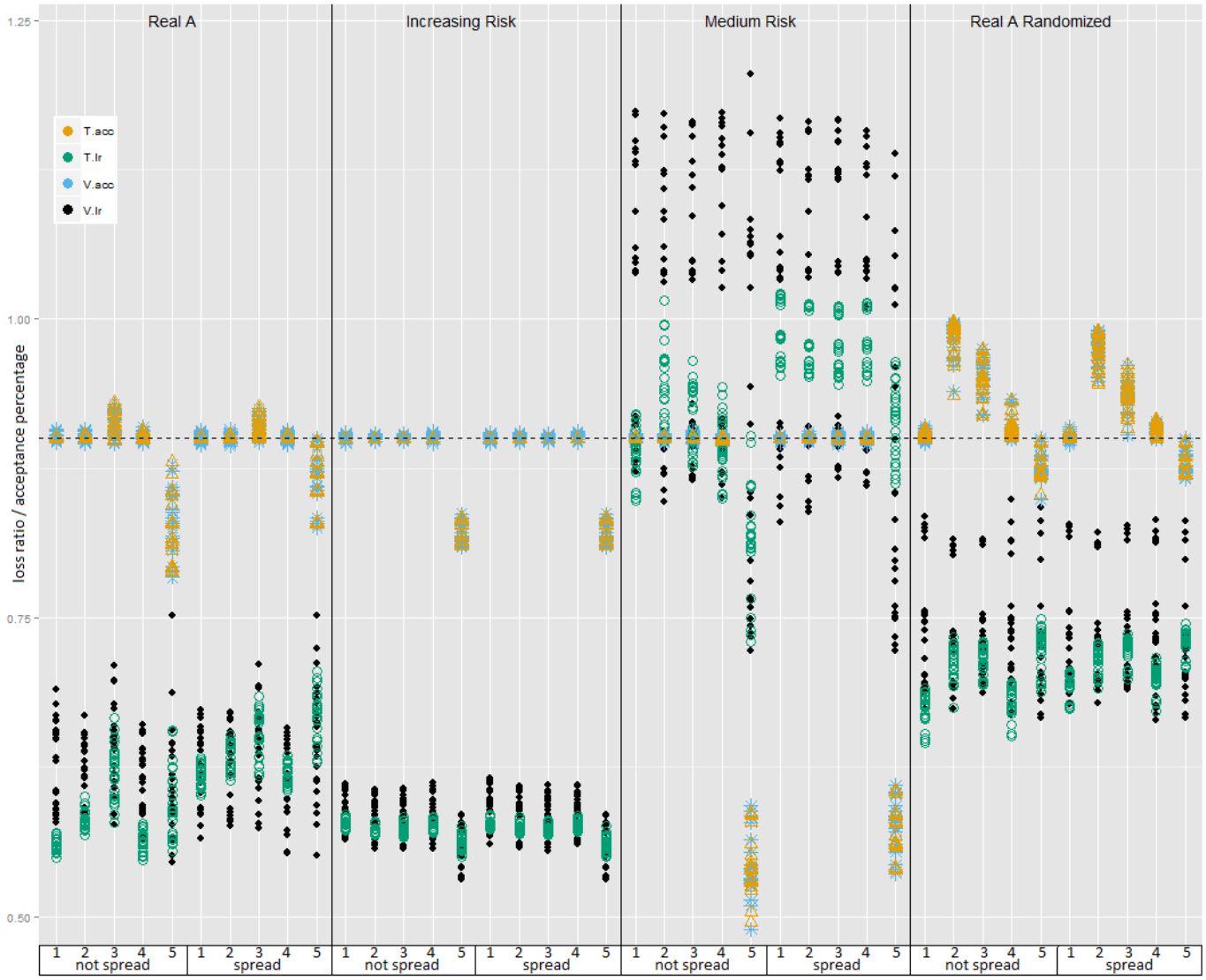


Figure 40: The loss ratio and acceptance percentage on the training sets and validation sets of four different datasets. The algorithms are numbered 1 to 5 (EA, TS, CMAES, PSO, Random). "not spread" means that the original training set was used, while "spread" means the spread-out training set was used. In the legend: *T.acc* = training set acceptance percentage. *T.lr* = training set loss ratio. *V.acc* = validation set acceptance percentage. *V.lr* = validation set loss ratio.

Table 17 shows the average improvement of the loss ratio over the used datasets. When a value is red, the average acceptance percentage of the solutions on which the improvement is found, is lower than 90%. In other words, too many cases have been rejected and the solutions are infeasible.
When we look at the average improvements on the validation sets, we can clearly see that the algorithms found good improvements on the Real A, Increasing Risk, Small Calamities and High Risk datasets. However, the Real B and Medium Risk datasets are improved less. When inspecting the results on the subsets, the Medium Risk dataset also shows good improvements.
For both the validation sets and the subsets, the results of the Real A randomized and Real B randomized datasets show no real improvement. This shows that the real datasets indeed have specific patterns in the feature hits, claim amount and premium income and also that these patterns disappear when we randomize the datasets.

	B	Real A		Real B		Increasing Risk		Small Calamities		High Risk		Medium Risk		Real A random.		Real B random.	
	C	*0.73*		*0.71*		*0.66*		*0.79*		*1.09*		*1.06*		*0.73*		*0.71*	
A	D \ E	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*	*yes*
V	**EA**	0.10	0.08	0.02	0.02	0.07	0.07	0.07	0.07	0.08	0.08	0.03	0.05	-0.01	-0.01	0.01	0.01
	TS	0.11	0.07	0.02	0.02	0.07	0.07	0.07	0.07	0.10	0.10	0.05	0.05	0.00	0.00	0.00	0.01
	CMAES	0.09	0.08	0.02	0.02	0.07	0.07	0.08	0.08	0.35	0.36	0.05	0.04	-0.01	-0.01	0.00	0.01
	PSO	0.11	0.09	0.03	0.02	0.07	0.07	0.08	0.07	0.49	0.50	0.04	0.05	-0.01	0.00	0.01	0.01
	Random	0.12	0.06	0.02	0.02	0.10	0.10	0.10	0.11	0.48	0.49	0.15	0.18	0.00	0.00	0.01	0.01
S	**EA**	0.15	0.12	0.05	0.03	0.08	0.08	0.09	0.09	0.13	0.11	0.14	0.08	0.04	0.04	0.03	0.03
	TS	0.14	0.10	0.04	0.03	0.09	0.08	0.10	0.10	0.13	0.13	0.10	0.09	0.01	0.01	0.02	0.02
	CMAES	0.10	0.10	0.03	0.03	0.08	0.09	0.10	0.10	0.37	0.38	0.12	0.08	0.01	0.01	0.02	0.02
	PSO	0.15	0.13	0.05	0.04	0.08	0.08	0.10	0.09	0.54	0.56	0.14	0.09	0.04	0.03	0.03	0.03
	Random	0.13	0.07	0.02	0.02	0.10	0.10	0.11	0.11	0.50	0.51	0.22	0.09	0.01	0.00	0.01	0.01

Table 17: The improvement, read decrease, made to the loss ratio of the datasets by rejecting specific cases. The red values are obtained from runs of which the average acceptance percentage was "<90%". In other words the found solutions are mostly infeasible solutions. (A) the average loss ratio improvement on the V=validation set and on the S=subset datasets. (B) The datasets on which the improvement was measured. (C) The initial loss ratio of the datasets. (D) The algorithm used for training. (E) Was the weight vector trained on the spread training set (yes) or not (no)?

6.4.2 Calamity Influence

Calamities can have a large influence on the search algorithms. The more calamities in the dataset and the higher their claim amount, the larger this influence is. In Figure 41 three different meta-heuristic algorithm runs are shown. Each point is a generated solution and the colour is based on which iteration it was generated, from red to purple. . On a relatively normal dataset the generated solutions can be seen distributed like the first. These are the generated solutions of a Tabu Search run. Two (local) optima are clearly visible by the two areas with a lot of purple coloured solutions. The second figure shows the same algorithm and the same dataset, but in this case the calamities are spread-out. Now there are no visible local optima, there only is a global optimum. On an extraordinary dataset something like the third figure might happen. For every couple of calamities in the dataset, the algorithm found a local optimum. Although this figure is from an early Evolutionary Algorithm variant, it illustrates a worst-case scenario when looking at the influence of calamities. Therefore, we state that calamities can have a serious influence on the performance of a meta-heuristic algorithm.

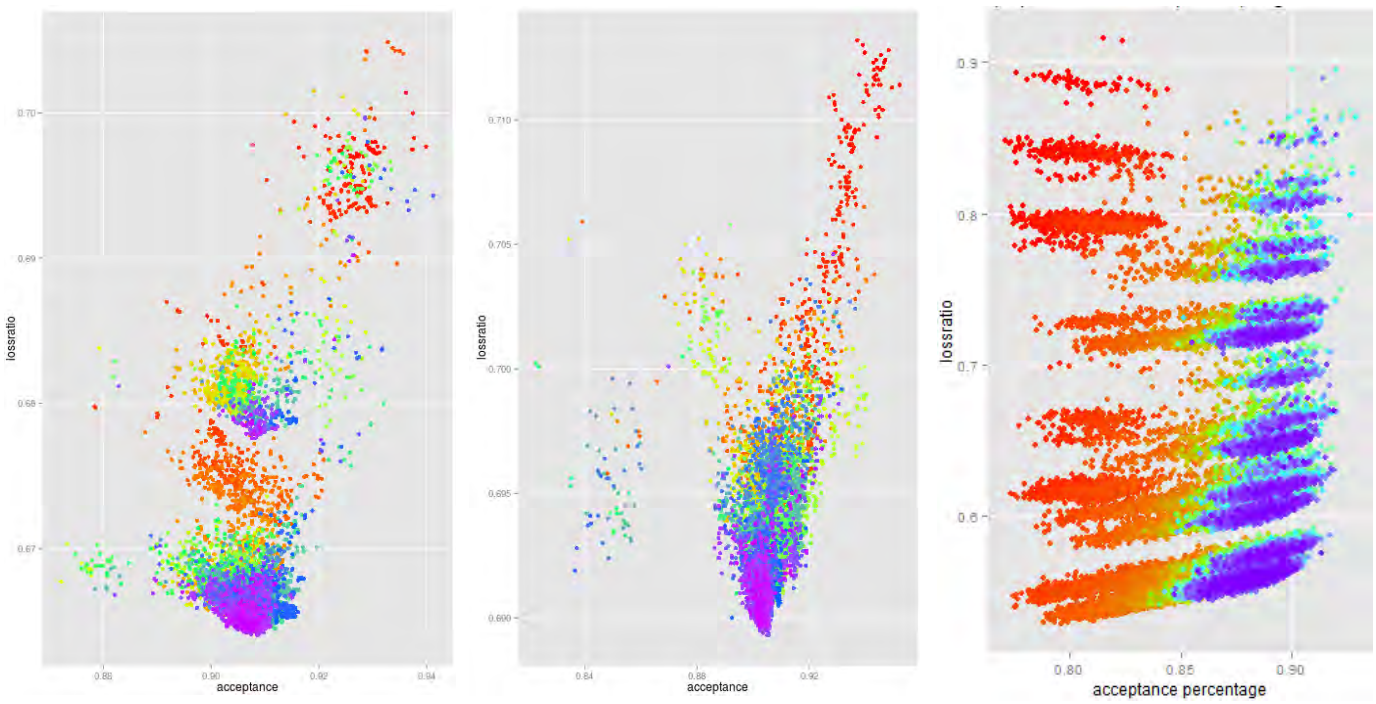


Figure 41: Three different runs of meta-heuristic algorithms. Each point is a generated solution. The colour is based on the iteration it was generated (from red to purple). The first figure shows the generated solutions of a Tabu Search run. Two (local) optima are clearly visible by the purple solutions. The second figure shows the generated solutions of the same algorithm on the same dataset, but in this case the dataset has spread-out calamities. There is no visible local optimum. The third figure shows what may happen with a problematic dataset. For each couple of calamities a local optima was found. Although the search pattern is of an early Evolutionary Algorithm variant (therefore not included in the thesis), it was run on one of the used datasets and shows a worst case scenario.

6.4.3 Time and Number of Iterations

EA and TS are very fast, finishing on the largest datasets in around 90 seconds, and also CMAES has reasonable speed with around 2.5 minutes. However, the PSO algorithm is, compared to the other algorithms, very slow. On a medium-sized dataset, it already takes 6.5 minutes to finish one run. Table 18 and Figure 42 show that the size of a dataset is almost linearly related to the runtime of the algorithms. The size of a dataset is the number of compound cases times the number of features.
Note that the runtime is over training sets containing 80% of each dataset, because the results were obtained with 5-fold cross validation. However, we can safely assume that, in most instances, at least a single case of each compound case is in each dataset created by the cross validation.

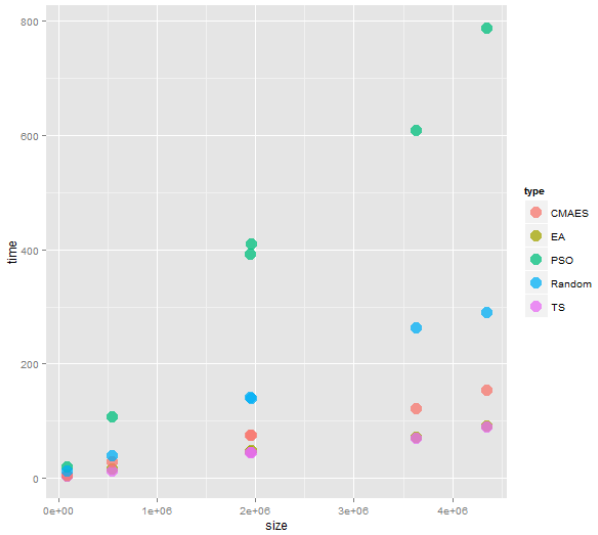


Figure 42: The size of a dataset and the average time it took each algorithm to finish one run. The size and The runtime have an almost linear relation.

Dataset	#Comp. Cases	#Features	Size
Real B (random.)	2206	39	86.034
Real A (random.)	4971	109	541.839
Small Calamities	19432	100	1.943.200
Increasing Risk	19509	100	1.950.900
High Risk	45426	80	3.634.080
Medium Risk	43418	100	4.341.800

Table 18: The datasets ordered by size. The size is the number of compound cases time the number of features.

6.4.4 Initial Solutions

We can already state that for some algorithms the initial solution is very important. Mostly the CMAES and PSO algorithms need a profitable initial solution, i.e. both algorithms generate better solutions when the initial solution(s), which are randomly generated, are close to, or are, feasible. More importantly the closer the acceptance percentage of the initial solution(s) is to the minimum allowed acceptance percentage, the better the solutions that the algorithms find.

However, the EA and TS algorithms have little trouble regarding the initial solution, mostly because of their manner of feature manipulation. The EA algorithm draws random weights, so multiple weights can be changed drastically in a single iteration, allowing it to leave the location of the initial solution very fast. While the TS algorithm can leave any area by adding moves to the tabu list.

Figure 43 shows the best loss ratio and acceptance percentage of five CMAES runs (left) and five PSO runs (right). Of the CMAES runs, four runs went fine, those are on the left. However, the fifth run had an initial solution to far away from the minimum acceptance percentage and got stuck in a local optimum. Of the PSO runs, all five runs got stuck in the same very bad local optimum, even though one of the initial solutions was the, always feasible, 0-weight vector.

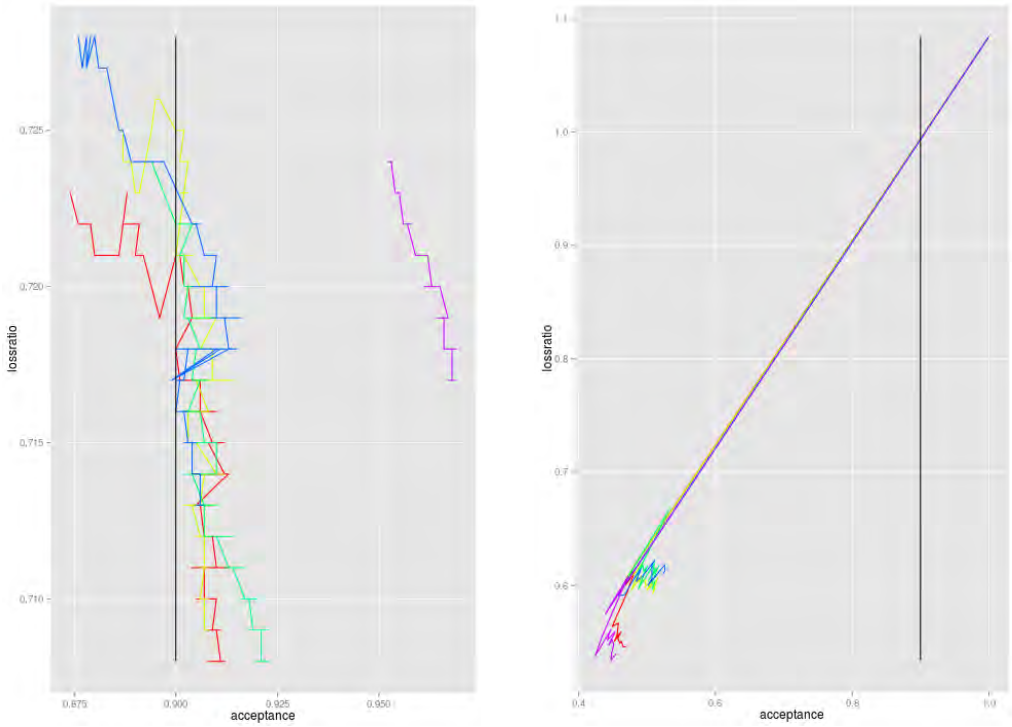


Figure 43: The initial solutions of these five CMAES (left) and PSO (right) runs do matter. On this dataset, the purple, most right, CMAES run has an initial solution with a loss ratio away from the feasibility boundary of 0.9 and is not able to improve its loss ratio and find a good solution. While the PSO runs cannot even find a feasible solution (not taking into account the initial solution with the 0-weight vector).

6.4.5 Generated Solutions

Table 19 shows some parts of different weight vectors. These parts show some of the characteristics of the different algorithms. As coded, EA sets feature weights randomly to 0 and this improves the loss ratio quite well, which is also shown by the weight vector shown as 75% of the weights is 0! Seen over all weight vectors generated, this percentage lies around 40%. TS has quite some weights close to 0. However, because of the tabu list the algorithm hardly sets them to 0. CMAES spreads out the weights almost equally over the features and only a few weights are close to 0 and 100. Then PSO, this algorithm has its weights distributed closely like EA, but with fewer weights on 0. Finally, random draw shows some equally spread out weights, which is more or less what you would expect.

Feature:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
EA	0	10	40	70	90	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	60
TS	0	44	56	64	95	36	0	25	2	6	30	0	14	87	1	0	41	1	18	0	50
CMAES	1	24	5	43	28	39	10	51	31	66	61	21	38	43	18	33	5	34	50	33	35
PSO	10	10	30	100	100	100	30	10	40	20	30	100	0	40	0	0	30	0	50	70	80
Random	15	85	35	20	100	60	80	30	10	70	25	85	25	65	20	75	40	55	20	75	35

Table 19: For each algorithm a part of a weight vector (trained on the same dataset) is shown. These parts show some characteristics of the algorithms. EA sets feature weights to 0 if it means an immediate improvement of the loss ratio. In this case 75% of the weights is 0! TS has also quite a few features close to 0 weight, but because of the tabu list it hardly sets them to 0. CMAES spreads out the weights almost equally over the features. While PSO seems to combine EA and CMAES. Some weights at 0 or 100 while the rest has quite equal weights. Finally, Random shows some equally spread out weights, like CMAES, but does not really lower weights to 0.

6.4.6 Stability of Solutions

Then the stability of the algorithms, see section 6.3.5. Table 20 shows for the different algorithms a part of three different weight vectors. These are weight vectors trained on the Real C dataset. The color-coding tells how close the weights are together. Dark green weights have at most a 5 point difference, light green weights differ at most a 10 points and orange weights differ at most 20 points.
We can clearly see that the Evolutionary algorithm is the most stable, while the random draw is the least stable. Furthermore, the other three algorithms are generating equally stable solutions. This behaviour is also visible with weight vectors generated on the other datasets.

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
EA	0	80	20	100	100	100	70	0	50	0	10	0	0	60	0	0	60	0	60	0	80
	10	10	40	30	100	100	40	0	0	0	10	100	0	0	30	0	70	100	50	0	100
	0	10	40	70	90	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	60
TS	4	38	0	29	18	0	44	21	59	12	50	100	53	22	17	0	19	6	0	0	49
	4	27	28	86	88	31	1	31	32	1	50	51	48	54	0	10	1	34	1	88	51
	0	44	56	64	95	36	0	25	2	6	30	0	14	87	1	0	41	1	18	0	50
CMAES	1	24	5	43	28	39	10	51	31	66	61	21	38	43	18	33	5	34	50	33	35
	1	46	19	31	31	31	34	29	48	10	45	33	53	42	20	17	15	46	44	1	31
	33	32	26	56	41	35	16	14	27	36	45	60	33	42	38	50	32	26	25	9	22
PSO	10	10	30	100	100	100	30	10	40	20	30	100	0	40	0	0	30	0	50	70	80
	0	10	30	70	100	100	10	10	20	10	30	100	50	10	10	0	0	0	0	10	10
	0	40	60	50	0	100	0	0	10	0	10	10	0	0	0	10	30	0	0	0	70
Random	25	75	0	10	75	95	0	55	10	100	0	100	50	25	90	10	10	25	25	45	45
	35	90	0	70	5	25	75	0	20	95	40	55	0	25	40	10	40	10	10	80	25
	15	85	35	20	100	60	80	30	10	70	25	85	25	65	20	75	40	55	20	75	35

Table 20: Some weights of weight vectors generated by the different algorithms trained on the real B dataset. The color-coding shows how stable the weights are. Dark green means that the weights have a maximum point difference of 5, light green means that the weights differ at most 10 points, while orange means that the weights differ at most 20 points. You can clearly see that the EA algorithm produces more stable solutions than the random algorithm.

However, the weight vectors trained on the same training set, but with spread out calamities, show a different pattern, which is visible in Table 21. Fewer weights are coloured dark green and the Evolutionary Algorithm is not producing very stable solutions anymore. The stability of the weight vectors of the other algorithms is comparable to the stability shown in Table 20.

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
EA	30	90	10	90	80	60	40	0	50	0	0	20	100	80	20	10	0	0	0	0	0
	0	20	40	100	40	100	0	10	0	30	20	100	60	20	0	0	0	0	0	0	10
	10	70	10	50	70	100	0	40	50	0	20	100	50	0	0	0	10	100	0	0	30
TS	0	10	10	14	80	47	22	16	49	38	47	48	46	29	0	39	0	0	26	0	38
	0	30	33	58	93	5	22	1	47	41	39	9	28	20	0	26	0	71	30	78	70
	1	0	18	19	100	1	39	6	79	33	2	19	21	23	25	0	34	10	2	0	26
CMAES	24	25	7	33	27	31	18	43	29	55	66	21	31	26	22	11	22	27	14	37	47
	1	29	22	27	31	42	31	28	48	36	43	39	52	45	23	20	17	46	42	12	33
	52	36	22	48	51	25	19	26	32	13	50	49	28	18	32	13	34	29	18	19	32
PSO	10	10	30	10	80	100	40	10	40	0	20	100	0	10	0	0	30	0	70	0	90
	0	40	0	30	100	70	50	20	70	20	20	100	10	20	0	20	50	100	20	10	0
	0	50	10	30	20	80	30	10	70	50	50	100	40	0	0	0	10	20	10	0	60
Random	25	75	0	10	75	95	0	55	10	100	0	100	50	25	90	10	10	25	25	45	45
	30	20	0	40	70	65	30	15	20	50	65	45	15	75	75	50	35	10	10	80	5
	15	85	35	20	100	60	80	30	10	70	25	85	25	65	20	75	40	55	20	75	35

Table 21: Some weights of weight vectors generated by the different algorithms trained on the real B dataset with spread-out calamities. The color-coding shows how stable the weights are. Dark green means that the weights have a maximum point difference of 5, light green means that the weights differ at most 10 points, while orange means that the weights differ at most 20 points.

7 Discussion

7.1 Meta-heuristic search is useful

We chose for meta-heuristic algorithms based on the properties of the problem and methods proposed by various papers, e.g. “Tabu Search – Part II” (1989) of F. Glover [17], wherein meta-heuristic algorithms are used to solve combinatorial optimization problems. Our analysis shows that on the real and artificial datasets, the meta-heuristic algorithms do perform better than just randomly drawing weight vectors.

Also, to compare randomly drawing weight vectors with the meta-heuristic search when no pattern should be in the data, we ran the algorithms on the randomized real datasets. On these datasets the random draw works equally well compared to the meta-heuristic search. Therefore, we can conclude that, when the data contains patterns, systematic search is useful regarding our problem. As the meta-heuristic algorithms can find and exploit the patterns in the data.

Although search is better than random draw, the exact solution does not matter as most algorithms are perfectly able to generate solutions with equal loss ratio and acceptance percentage. Probably, there are no real differences between the algorithms, because the problem was easier than expected, there was only one constraint and solution feasibility was easy to achieve.

Besides meta-heuristic algorithms, some other methods could be considered for solving the combinatorial optimization problems. These methods are *Complete Enumeration*, a *Mixed Integer Linear Problem solver* and other meta-heuristic algorithms, for instance *Generalized Simulated Annealing.*

Complete Enumeration
Complete Enumeration basically is just testing all possible solutions and is sure to find the optimal solution. However, this is undoable. For instance, we take K features and cut-off value C the number of unique solutions is K^{C+1}. This is an exponentially growing amount of solutions. I.e. when we consider $K = 25$ and $C = 100$, then the number of possible solutions is $1.55 * 10^{141}$, which is more than the estimated number of particles in the universe. The FRISS software usually contains between 100 and 400 features.

Increasing the granularity of the feature weights helps, but taking our small example above with a weight granularity of 5 or 10, this still leaves us with $2.27 * 10^{29}$ and $2.38 * 10^{15}$ solutions respectively. When taking the smaller amount and 1 microsecond, $1 * 10^{-6}$ seconds, to calculate one solution, we need little over 31 years to calculate the objective value of every solution. This means that complete enumeration is not feasible as method for solving our problem.

Mixed integer linear problem solver
The second method is using a Mixed Integer Linear Problem solver (MILP solver). The optimization problem can be rewritten as a mixed integer problem, for which a solver can, theoretically, find the exact optimal objective value and corresponding solution. However, there are downsides of using a solver. Firstly, using a solver is expensive. Would we use one of the best solvers available at the time of writing, Cplex. This solver costs $10.000 on an annual basis. Secondly, there are problems concerning the properties of our problem. The data mostly available for this problem are crude approximations of the actual insurance portfolios. For instance, when an insurer has a portfolio of 1 million cases and provides FRISS with a set of 100.000 cases that is only 10% of the portfolio. Therefore, the available data is only a crude approximation of the real data.

Both the meta-heuristic algorithms and the MILP solver face this same problem. However, the solver will give an exact solution to this crude approximation of the portfolio. While a meta-heuristic algorithm, which does not guarantee an optimal solution, searches for a solution which lies close to the optimal solution. Because the dataset is an approximation of the full portfolio, it is doubtful if using a solver to exactly solve a problem with an approximation of the dataset is useful. Also the precise solution might not generalize well to the whole portfolio, while a sub-optimal solution might.

However, in this specific case the question can be raised if the combinatorial optimization problem is sufficiently well formulated. To be precise, every solution with a better objective value than the current solution must be accepted. When this better solution does not generalize, then the data is not a good approximation of the whole portfolio, or a constraint or objective must be added to the current objective function.

Other meta-heuristic algorithms
Besides the four meta-heuristic algorithms used in this thesis, there is an extensive list of other possible algorithms. For instance, hybrid meta-heuristic algorithms can be used. These are meta-heuristic algorithms which, e.g. combine search strategies of two different algorithms or switch the neighbourhood generation of the solutions halfway through the search. There are almost infinite variations of hybrid algorithms that can be considered. However, our results show that the knowledge of meta-heuristic algorithms and the programmer's skill seem at least equally important to finding a good solution than choosing the right algorithm

Generated Simulated Annealing
Of the other meta-heuristic algorithms possible, we did also try the Generalized Simulated Annealing algorithm. Although no data is included in this thesis. Generalized Simulated Annealing is a global search algorithm based on the original Simulated Annealing algorithm proposed by Kirkpatrick at all. (1983) [26]. Tsallis & Stariolo (1996) [32] proposed a global search distribution, which is a generalized version of the local search distribution used with Simulated Annealing.

The Generalized Simulated Annealing algorithm seems to generate good solutions. However, the algorithm needs much more objective value evaluations, and run time, than the other meta-heuristic algorithms that were used. The jumping through the search space and accepting worse solutions allows for escaping local optima, but also increases the number of iterations needed to settle down into the currently found local or global optima.

7.2 Comparing Algorithm Traits

We will not declare one algorithm the best algorithm to solve our combinatorial optimization problem. As described in section 7.1 and substantiated by the analysis in section 6.4, the meta-heuristic search algorithms perform equally well when the seed of the random number generator is *good*. Although when the seed is *bad* than there is a difference between the CMAES and PSO algorithms on one hand and EA and TS on the other, with the latter performing more reliably as the former often gets stuck into local optima.

Run time
The run times of the used algorithms differ a lot, as shown in section 6.2 and noted in section 6.4. However, we cannot fully compare the run times of the different algorithms, as the specific implementation of each algorithm decides much of the total run time. Currently we might only compare the EA, TS and Random Draw algorithms, because the CMAES and PSO implementations have been written by respectively Trautmann et al. (2011) [31] and Bendtsen (2012) [04].

The run time is influenced by the skill of the programmer of the algorithm and the type of algorithm, i.e. comparing two algorithms that use very different computations is like comparing apples and oranges. Therefore, we cannot prove that one algorithm truly is faster than another. For instance, an older version of our EA algorithm took more than twice the current run time for the same number of iterations with the only reason being a worse implementation.

Solution Complexity
We can make a slight difference between algorithms based on the complexity of the generated solutions. Two algorithms, EA and PSO, generate solutions with a weight granularity of 10, which generally returns solutions that are simpler as well as easier to understand. Also these two algorithms generate solutions with low loss ratio, so we can state that they have a slight advantage over the other algorithms based on Occam's razor, see section 3.3. Starting with feasible or unfeasible weight vectors does not matter most of the time. The search algorithms have the tendency to always reject the maximum amount of cases that are allowed to be rejected. In other words, they try to abuse the feasibility constraint as much as possible.

Overfitting on Calamities
The relation between overfitting on the calamities in the data seems relatively small. Only large calamities are shown to be rejected. Also, sections 6.4.1 and 6.4.2 show that the standard deviation of the loss ratio of the generated weight vectors, over the original and spread-out training sets, is equal. So, it does not make any real difference on which training set the weight vector was trained. However, the loss ratio of the weight vectors trained on the spread-out training sets are closer to the loss ratio of the validation sets.

All in all, we can state that on our used datasets the difference between using the original and spread-out training sets is neglectable. There might only be a slight advantage for the weight vectors trained on the spread-out training set as their loss ratio is closer to the loss ratio of the validation set.

Increased number of objective value evaluations
Besides the showed results in chapter 6, we also ran the same experiments with 20.000 objective value evaluations for each algorithm instead of the used 10.000 evaluations. In practice, this means doubling of the number of iterations for each algorithm. In most cases the extra evaluations did not significantly improve the objective value of the best solution. Figure 44 shows a Tabu Search algorithm run that returns no significant improvement of the objective value.

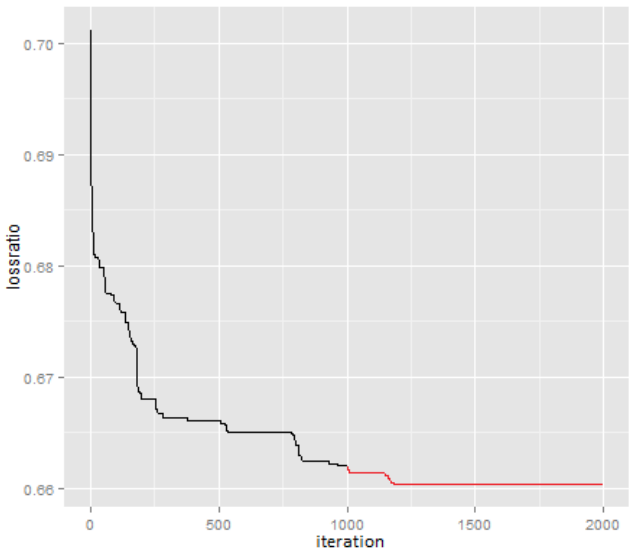


Figure 44: The improvement of the objective value of a Tabu Search algorithm after iteration 1000 is coloured red. In this case the extra iterations only improves the objective value by 0.0016.

7.3 Future Research

Initial solutions

Currently the initial solutions of the algorithms are generated randomly. A method that searches for *better than random* initial solutions might give the search algorithms a boost in the first few iteration. For example, we can choose a couple of features and run a few iterations on the dataset and then use the resulting weight vectors for the initial solutions of the real run.

Operational and feature costs

The problem definition, introduced in chapter 2, can be extended. As mentioned in the introduction, FRISS' technical platform consults various external data sources. For most of these data sources a fee must be paid for each case. Therefore, in some situations the benefit of adding a specific data source, and thus using specific features, can be less than the cost associated with its use. Also insurers have operational costs, e.g. personal costs and housing costs. These can be modelled as a fixed percentage of their total premium income. For most insurers this value lies around 35%. Both types of costs are subtracted from the bruto premium income. Formula (1) of section 2.1 would then become:

$$L = {}^{S_A}\Big/\left((1 - c_{\%}) * P_A - I * \sum_{k \epsilon K} c_k * \mathbb{1}_{w_k > 0}\right)$$

Variable	*Description*
L	*The loss ratio of the portfolio associated with the accepted cases*
S_A	*The total claim amount associated with the accepted cases*
P_A	*The total premium income associated with the accepted cases*
$i \epsilon I$	*i = 0,…,#cases*
$k \epsilon K$	*k = 0,…,#features*
$c_{\%}$	*The percentage of the premium income used for the insurer's operational costs*
c_k	*The costs of using feature k.* $c_k \geq 0$.
$\mathbb{1}_{w_k > 0}$	*Indicator variable which equals 1 if weight k is larger than 0 and 0 otherwise*

Table 22: The variables of formula (1) and constraints (2) to (4).

Workload

The chosen problem definition makes a split between the immediately accepted cases, the set A, and the immediately rejected cases, the set R. However, FRISS usually works with three layers, namely *green*, *orange* and *red*, where commonly the green cases are in the set A and the red cases are in the set R. The orange cases are cases that the insurer has to investigate, say a set I, as their FRISS score is not low enough to be accepted nor high enough to be rejected. This can be modelled as a workload constraint for the insurer. Which basically implies that we cannot reject more than $x\%$ of the cases immediately, but we also may not have more than $y\%$ of the cases as workload.

8 Conclusion

The key question of this thesis was: C*an we systematically predict which clients will likely file claims after acceptance, based on an a priori known set of measurable features and can we use this information to increase the total value of a portfolio?* This key question gave us the following research question: *How to select an optimal weight vector that maximizes the value of the accepted set of cases, while still respecting the volume constraint?*

Based on our research, we can conclude that we can successfully select a set of feature weights, assigning each potential client a risk score. This risk score allows us to reject clients associated with a higher risk. Rejecting high risk clients improves the portfolio, because fewer claims will have to be paid out. This allows the loss ratio to decrease. A lower loss ratio generally means that the total value of the underlying portfolio is higher. Secondly, a low loss ratio is often associated with low risk.

To find the specific sets of weights which optimize the loss ratio of the portfolio a diverse set of meta-heuristic algorithms can be used. These algorithms are able to produce solutions in a relatively short time. We found that systematic search is better than randomly generating sets of feature weights, but that the algorithm choice is not decisive for the loss ratio improvement that can be made, as each tested meta-heuristic algorithm is able to find near optimal sets of feature weights. However, the underlying portfolio data is decisive for the loss ratio improvement. Also, the algorithms generally respect the volume constraint, but they do attempt to get maximum value out of the constraint by rejecting just the allowed number of cases.

Finally, we can conclude that large outliers in the data do not have a major impact on the search. When we compare the loss ratio of the weight vectors trained on the original and spread-out training sets, the variance is equal. However, a noticeable difference is that, when training on the spread-out training sets, the loss ratio of the weight vectors found, is closer to the loss ratios found on the validation sets. This might indicate that a weight vector trained on a spread-out training set generalizes better towards the whole portfolio than a weight vector trained on an original training set.

As the algorithms do not differ much in performance, any preference for which algorithm to use must be based on other features. Two of these features are the time needed to find a satisfying solution and the simplicity of this solution. Based on these two features, we recommend using our Evolutionary Algorithm.

An important contribution of this work is a free to use application, which is shown and explained in chapter 5, containing all algorithms and options described. The application is implemented using modern web standards with R as the statistical engine.

9 Appendix

9.1 Datasets

In this section the different datasets are mentioned shortly. Also the parameters for the artificial dataset generator are given, so that the artificial datasets can be recreated, while an approximation of the real datasets can be generated. Table 23 shows an overview of the datasets used in this thesis. Shown are the number of cases a dataset has. How many calamities there are in the dataset. The starting loss ratio of the dataset, which we want to improve by changing the feature weights. The number of features and finally, the number of compound cases, which are created when cases with the same pattern of feature hits are summed.

Dataset	#Cases	#Calamities	Loss Ratio	#Features	#Comp. cases
Real A	71.092	133	0.729	109	4.971
Real B	56.779	52	0.706	39	2.206
Real A Randomized	71.092	133	0.729	109	4.971
Real B Randomized	56.779	52	0.706	39	2.206
Increasing Risk	100.000	19	0.658	100	19.509
High Risk	100.000	32	1.087	80	45.426
Medium Risk	100.000	19	1.060	100	43.418
Small Calamities	100.000	291	0.791	100	19.432

Table 23: A short overview of the eight datasets that are used for this thesis. First, there are two real datasets. Then the same two datasets randomized and finally four artificial datasets.

9.1.1 Real Datasets

Tables 24 and 25 show the parameters for the artificial dataset generator to create an approximation of the two real datasets used in this thesis. The risk group specific features are chosen based on the original feature weights in the FRISS software.

Dataset: Real A

Volume	Loss Ratio	Premium Income	Claim Amount	Claim SD	Features
22962	0.37	383	835	1.198	0
27477	0.55	341	1172	1.254	26
11868	0.51	368	1224	1.286	17
8785	0.66	369	1401	1.309	66

Risk 1 Hits	Risk 2 Hits	Risk 3 Hits
1.31	0	0
1	1.1	0
1.3	0.23	1.08

Value	Claim Mean	Percentage
25000	44000	0.0124

Number of hits per feature
4063,7,4,2574,625,5737,156,6338,3529,11711,5416,1013,1206,2203,129,54,3477,273,185,283,6,3,2295,223,158,7556,2525,266,305,873,1646,60,860,1442,4718,4,1,286,331,374,835,207,314,36,86,390,45,190,441,360,7,682,752,294,528,1,226,492,89,274,358,124,50,33,8,4,1696,193,610,191,86,14,20,2,315,26,69,8,16,6,2,1,237,13,48,6,13,18,6,13,2,23,17,2,28,87,12,6,4,12,59,2,5,13,10,30,48,14,44

Table 24: The first real dataset, Real A. From top to bottom: the Risk Group parameters, the Feature Hits parameters, the Calamity Case parameters and the number of hits per feature.

Dataset: Real B

Volume	Loss Ratio	Premium Income	Claim Amount	Claim SD	Features
17226	0.6	427	1448	1.135	0
23362	0.58	499	1508	1.198	23
14809	0.67	366	1634	1.189	11
1382	0.64	347	1543	1.21	5

Risk 1 Hits	Risk 2 Hits	Risk 3 Hits
1.55	0	0
1.07	1.2	0
1.17	0.55	1.04

Value	Claim Mean	Percentage
25000	49800	0.005

Number of hits per feature
206,9,741,26,151,8285,1826,210,7161,296,131,1,4,14707,390,206,228,359,2450,1098,14757,117,348,365,2358,2169,3489,250,5649,168,3868,1,149,74,88,422,314,424,195

Table 25: The second real dataset, Real B. From top to bottom: the Risk Group parameters, the Feature Hits parameters, the Calamity Case parameters and the number of hits per feature.

9.1.2 Randomized Real Datasets

The Randomized Real datasets are created from the Real A and B datasets by randomizing the features and the combination of premium income and claim amount pairs, see section 4.1.3. The R-code used for the randomization is shown at Code 1.

```
K <- dim(dataset)[2] – 3 #the number of features
set.seed(1234)
randomized <- cbind(dataset[,1:K],dataset[sample(1:nrow(dataset)),K+(1:3)])
```

Code 1: The R-code used for randomizing the real datasets. The randomization is over the premium income and claim amount pairs. The third column selected at the 3rd code row, at K+(1:3), is a row reserved for the size of a compound case.

When using the shiny application from the shown url in chapter 5, the randomization using the code cannot be done. Therefore, approximations of the datasets can be created using the parameters shown in Tables 26 and 27, in combination with the parameters of Tables 24 and 25.

Dataset: Real A Randomized

Volume	Loss Ratio	Premium Income	Claim Amount	Claim SD	Features
22962	0.49	363	1170	1.263	0
27477	0.49	365	1073	1.261	26
11868	0.53	359	1158	1.302	17
8785	0.48	360	1042	1.212	66

Table 26: the Risk Group parameters of the randomized version of dataset Real A.

Dataset: Real B Randomized

Volume	Loss Ratio	Premium Income	Claim Amount	Claim SD	Features
17226	0.58	465	1507	1.177	0
23362	0.63	466	1552	1.183	23
14809	0.6	462	1483	1.161	11
1382	0.71	462	1770	1.307	5

Table 27: the Risk Group parameters of the randomized version of dataset Real B.

9.1.3 Artificial Datasets

Tables 28, 29, 30 and 31 show, for the four artificial datasets, the parameters of the artificial dataset generator that were used to generate the datasets.

Dataset: Increasing Risk

Volume	Loss Ratio	Premium Income	Claim Amount	Claim SD	Features
40000	0.45	550	2000	0.5	0
32500	0.6	550	2000	0.5	60
22500	0.85	550	2000	0.5	30
5000	1.5	550	2000	0.5	10

Risk 1 Hits	Risk 2 Hits	Risk 3 Hits
1.25	0	0
1.25	1.25	0
1.25	1.25	1.25

Value	Claim Mean	Percentage	Seed
25000	50000	0.0015	32000

Table 28: The first artificial dataset, Increasing Risk. From top to bottom: the Risk Group parameters, the Feature Hits parameters and the Calamity Case parameters.

Dataset: High Risk

Volume	Loss Ratio	Premium Income	Claim Amount	Claim SD	Features
25500	0.3	350	1500	0.5	0
8500	0.5	320	1750	0.5	40
33000	1.2	300	2250	0.5	15
33000	1.5	280	2500	0.5	25

Risk 1 Hits	Risk 2 Hits	Risk 3 Hits
2	0	0
1.25	1.5	0
1.2	1.25	1.5

Value	Claim Mean	Percentage	Seed
25000	100000	0.002	100

Table 29: The second artificial dataset, High Risk. From top to bottom: the Risk Group parameters, the Feature Hits parameters and the Calamity Case parameters.

Dataset: Medium Risk with high calamity claim amount

Volume	Loss Ratio	Premium Income	Claim Amount	Claim SD	Features
10000	0.5	350	1500	0.5	0
50000	0.7	320	1750	0.5	60
13250	1.1	280	2000	0.5	15
26750	1.3	320	2500	0.5	25

Risk 1 Hits	Risk 2 Hits	Risk 3 Hits
2	0	0
1.5	1.25	0
1.25	1.1	1.25

Value	Claim Mean	Percentage	Seed
25000	250000	0.0015	20

Table 30: The third artificial dataset, Medium Risk. From top to bottom: the Risk Group parameters, the Feature Hits parameters and the Calamity Case parameters.

Dataset: Small Calamities

Volume	Loss Ratio	Premium Income	Claim Amount	Claim SD	Features
40000	0.45	550	2000	0.5	0
32500	0.6	550	2000	0.5	60
22500	0.85	550	2000	0.5	30
5000	1.5	550	2000	0.5	10

Risk 1 Hits	Risk 2 Hits	Risk 3 Hits
1.25	0	0
1.25	1.25	0
1.25	1.25	1.25

Value	Claim Mean	Percentage	Seed
25000	30000	0.02	9876

Table 31: The fourth artificial dataset, Small Calamities. From top to bottom: the Risk Group parameters, the Feature Hits parameters and the Calamity Case parameters.

9.2 Experiment Parameters

Table 32 shows the algorithm parameters that are used in the experiments of section 6.1. All parameter, except for the *Cross-Val. Seed* parameter, can also be used in the artificial dataset generator, to replicate the results of this thesis. The *Cross-Val. Seed* parameter is the seed used to split up the datasets in folds for the 5-fold cross-validation. See also section 6.1 Figure 36 for information on the use of this parameter.

Parameter \ Algorithm	CMAES	EA	PSO	TS	Rand. Draw
#Iterations	1000	1000	400	1000	1000
#Solutions	1	10	25	1	100
#Offspring per Sol.	10	1	1	10	1
#Runs	5				
Maximum Reject %	0.1				
Cut-off value	100				
Manipulation chance	-	0.1	-	-	-
Step size	1	10	10	-	5
Initial lowerbound	20	0	0	0	-
Initial upperbound	50	50	0	50	-
Run Seed	1234,2345,3456,4567,5678				
Subset Seed	1111				
Cross-Val. Seed	9999				

Table 32: The algorithm parameters for the runs on the datasets. The '-' means that the specific parameter is not used by that specific algorithm. When replicating the results with the artificial dataset generator, note that it does not contain the *Cross-Val. Seed* parameter. This parameter is used outside the application to generate the fold for the 5-fold cross-validation. Note that the step size parameter is the weight granularity show in section 3.1.6.

10 References

[01] **Aarts, E. and Lenstra, J.K.** | *Local search in combinatorial optimization* | Princeton University Press | 2003 | http://press.princeton.edu/titles/7564.html

[02] **Auger, A. and Hansen, N.** | *Tutorial CMA-ES - Evolution Strategies and Covariance Matrix Adaptation* | Genetic and Evolutionary Computation Conference '13 (GECCO'13) | ACM New York | 2013 | https://www.lri.fr/~hansen/gecco2013-CMA-ES-tutorial.pdf

[03] **Bates, D. and Eddelbuettel, D.** | *Fast and Elegant Numerical Linear Algebra Using the RcppEigen Package* | Journal of Statistical Software, Vol. 52(5) | Foundation for Open Access Statistics | 2013 | http://www.jstatsoft.org/v52/i05

[04] **Bendtsen, C.** | *pso: Particle Swarm Optimization* | R package version 1.0.3 | 2012 | http://CRAN.R-project.org/package=pso

[05] **Bianchi, L., Dorigo, M., Gambardella, L.M. and Gutjahr, W.J.** | *A survey on metaheuristics for stochastic combinatorial optimization* | Natural Computing: an international journal, Vol.8(2), p239-287 | Kluwer Academic Publishers | 2008 | http://code.ulb.ac.be/dbfiles/BiaDorGamGut2009natcomp.pdf

[06] **Brownlee, J.** | *Clever Algorithms: Nature-Inspired Programming Recipes* | ch. 2.10 Tabu Search, p76-81 | 2012 | https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.190/Mitarbeiter/voelkel/CleverAlgorithms.pdf

[07] **Brownlee, J.** | *Clever Algorithms: Nature-Inspired Programming Recipes* | ch. 6.2 Particle Swarm Optimization, p241-246 | 2012 | https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.190/Mitarbeiter/voelkel/CleverAlgorithms.pdf

[08] **Chang, W., Cheng, J., Allaire, J., Xie, Y. and McPherson, J.** | *shiny: Web Application Framework for R* | R package version 0.12.2 | 2015 | http://CRAN.R-project.org/package=shiny

[09] **Clerc, M.** | *Beyond Standard Particle Swarm Optimisation* | International Journal of Swarm Intelligence Research (IJSIR), Vol. 1(4), p46-61 | IGI Global | 2010 | http://www.irma-international.org/viewtitle/50311/

[10] **Clerc, M.** | *Standard Particle Swarm Optimisation - From 2006 to 2011* | HAL, archives-ouvertes.fr | 2012 | https://hal.archives-ouvertes.fr/hal-00764996/document

[11] **Domingos, P.** | *The Role of Occam's Razor in Knowledge Discovery* | Data Mining and Knowledge Discovery, Vol. 3(4), p409-425 | Kluwer Academic Publishers | 1999 | http://homes.cs.washington.edu/~pedrod/papers/dmkd99.pdf

[12] **Eddelbuettel, D. and Francois, R.** | *Rcpp: Seamless R and C++ Integration* | Journal of Statistical Software, Vol. 40(8) | Foundation for Open Access Statistics | 2001 | http://www.jstatsoft.org/v40/i08

[13] **Eiben, A.E. and Smith, J.E.** | *Introduction to Evolutionary Computing* | Springer | 2015 | http://file.allitebooks.com/20150722/Introduction%20to%20Evolutionary%20Computing,%202nd%20edition.pdf

[14] **Garcia-Nieto, J. and Alba, E.** | *Why six informants is optimal in PSO* | Genetic and Evolutionary Computation Conference '12 (GECCO'12), p25-32 | ACM New York | 2012 | https://www.lri.fr/~hansen/proceedings/2012/GECCO/proceedings/p25.pdf

[15] **Glover, F.** | *Future Paths for Integer Programming and links to Artificial Intelligence* | Computers & Operations Research, Vol. 13(5), p533-549 | Pergamon Journals Ltd. | 1986 | http://leeds-faculty.colorado.edu/glover/TS%20-%20Future%20Paths%20for%20Integer%20Programming.pdf

[16] **Glover, F.** | *Tabu Search - part I* | ORSA Journal on Computing, Vol. 1(3), p190-206 | Operations Research Society of America | 1989 | http://leeds-faculty.colorado.edu/glover/TS%20-%20Part%20I-ORSA-aw.pdf

[17] **Glover, F.** | *Tabu Search - part II* | ORSA Journal on Computing, Vol. 2(1), p4-32 | Operations Research Society of America | 1990 | http://leeds-faculty.colorado.edu/glover/fred%20pubs/203%20-%20TS%20-%20Part%20II-ORSA-aw.pdf

[18] **Glover, F.** | *Tabu Search and Adaptive Memory Programming - Advances, Applications and Challenges* | Interfaces in Computer Science and Operations Research Vol. 7, p1-76 | Kluwer Academic Publishers | 1996 | http://leeds-faculty.colorado.edu/glover/TS%20%20-%20AdaptiveMemory.pdf

[19] **Glover, F. and Laguna, M.** | *Tabu Search* | Kluwer Academic Publishers | 1997 | http://www.amazon.com/Tabu-Search-Fred-W-Glover/dp/079239965X

[20] **Glover, F., Laguna, M. and Martí, R.** | *Principles of Tabu Search* | Approximation Algorithms and Metaheuristics 23, p1-12 | 2007 | http://www.uv.es/rmarti/paper/docs/ts1.pdf

[21] **Hansen, N.** | *The CMA Evolution Strategy: A Comparing Review* | Towards a New Evolutionary Computation - Studies in Fuzziness and Soft Computing, Vol. 192, p75-102 | Springer-Verlag | 2006 | https://www.lri.fr/~hansen/hansenedacomparing.pdf

[22] **Hansen, N. and Ostenmeier, A.** | *Completely Derandomized Self-Adaptation in Evolution Strategies* | Evolutionary Computation, Vol. 9(2), p159-195 | ed. Lozano, J.A., Larrañaga, P., Inza, I. and Bengoetxea E. | Massachusetts Institute of Technology | 2001 | https://www.lri.fr/~hansen/cmaartic.pdf

[23] **Hansen, N. and Kern, S.** | *Evaluating the CMA Evolution Strategy on Multimodal Test Functions* | Parallel Problem Solving from Nature (PPSN), Vol. VIII, p282-291 | ed. Yao, X., et al. | Springer-Verlag | 2004 | https://www.lri.fr/~hansen/ppsn2004hansenkern.pdf

[24] **Hansen, N., Arnold, D.V. and Auger, A.** | *Evolution Strategies* | Handbook of Computational Intelligence, p871-898 | ed. Kacprzyk, J. and Pedrycz, W. | Springer | 2013 | https://www.lri.fr/~hansen/es-overview-2014.pdf

[25] **Kennedy, J. and Eberhart, R.** | *Particle Swarm optimization* | IEEE International Conference on Neural Networks '95 (proceedings), Vol. 4, p1942-1948 | 1995 | http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.248.4138&rep=rep1&type=pdf

[26] **Kirkpatrick, S., Gelatt, C.D. and Vecchi, M.P.** | *Optimization by Simulated Annealing* | Science, Vol. 220, No. 4598, p671-680 | American Association for the Advancement of Science | 1983 | https://stat.duke.edu/~scs/Courses/Stat376/Papers/TemperAnneal/KirkpatrickAnnealScience1983.pdf

[27] **Knuth, D.E.** | *Literate programming* | The Computer Journal, Vol. 27(2), p97-111 | Oxford University Press | 1984 | http://www.literateprogramming.com/knuthweb.pdf

[28] **Michell, T.M.** | *Machine Learning* | McGraw-Hill Science/Engineering/Math | 1997 | http://personal.disco.unimib.it/Vanneschi/McGrawHill_-_Machine_Learning_-Tom_Mitchell.pdf

[29] **R Core Team** | *R: A language and environment for statistical computing* | R Foundation for Statistical Computing | 2015 | http://www.R-project.org

[30] **Szu, H. and Hartley, R.** | *Fast Simulated Annealing* | Physics Letters A, Vol. 122(3-4), p157-162 | American Institute of Physics | 1987 | http://www.researchgate.net/publication/234014515_Fast_simulated_annealing

[31] **Trautmann, H., Mersmann, O. and Arnu, D.** | *cmaes: Covariance Matrix Adapting Evolutionary Strategy* | R package version 1.0-11 | 2011 | http://CRAN.R-project.org/package=cmaes

[32] **Tsallis, C. and Stariolo, D.A.** | *Generalized Simulated Annealing* | Physica A, Vol. 233, p395-406 | Elsevier | 1996 | http://www.if.ufrgs.br/~stariolo/publications/TsSt96_PhysA233_395_1996.pdf