Logic Regression
in Association Mapping

Master’s Thesis
Department of Computer Science
Aarhus University
Ole Jørgensen
Student ID: 20012263

Supervisors:
Christian Nørgaard Storm Pedersen
Thomas Mailund

March 6, 2009
# Contents

1 Logic Regression: An Introduction 3

2 Methods 5
   2.1 Logic Expressions ........................................ 5
   2.2 Logic Trees ................................................ 6
   2.3 Evolutionary Algorithm ................................. 8
      2.3.1 Reproduction: Logic Regression .............. 9
      2.3.2 Evaluation: Scoring Trees ............. 11

3 Implementation 13
   3.1 Sampler .................................................. 14
   3.2 Data ....................................................... 15
   3.3 Tree Structure ....................................... 16
   3.4 Mutation Model ..................................... 18

4 Experimental Results and Discussion 21
   4.1 Simulation of Data ................................. 22
   4.2 Analysis of Results ................................. 22
   4.3 Experiments ........................................... 23
      4.3.1 Experiment: Score Function Modification (β) .. 24
      4.3.2 Experiment: Iterations .......................... 26
      4.3.3 Experiment: Recombination Rate (ρ) ........ 30
      4.3.4 Experiment: Mutation Risk Variable (m) ...... 31
      4.3.5 Experiment: Comparison to Other Methods. .... 32
   4.4 Investigation of Potential .......................... 32
      4.4.1 Experiment: Mutation Risk and Ranking ..... 34
      4.4.2 Experiment: β and Ranking ................. 34
   4.5 Discussion ............................................. 37

5 Conclusion and Future Work 39
   5.1 Conclusion .............................................. 39
   5.2 Future Work .......................................... 40

A Supplementary Experimental Figures 43
List of Figures

1.1 Example of an Association Matrix .......................... 3
2.1 General Logical Expressions BNF .......................... 6
2.2 Used Logical Expressions BNF .......................... 6
2.3 Tree Representing a Logic Expression ...................... 7
2.4 Possible Tree Mutations .................................. 10
2.5 Scoring Matrix ........................................... 11
3.1 Tree Equality ............................................. 17
4.1 Experiment: $\beta$ (High $\rho$) ............................ 25
4.2 Experiment: $\beta$ (Low $\rho$) ................................ 26
4.3 Experiment: Iterations (Limit) ............................ 28
4.4 Experiment: Iterations (Degeneration) ..................... 29
4.5 Experiment: Recombination Rate .......................... 30
4.6 Experiment: Mutation Risk ................................ 31
4.7 Experiment: Comparisons ................................ 33
4.8 Trait Marker Selection ..................................... 34
4.9 Experiment: Trait Marker Rank ............................ 35
4.10 Experiment: Rank and $\beta$, $\rho = 100$ .................. 36
4.11 Experiment: Rank and $\beta$, $\rho = 200$ .................. 36
A.1 Comparison: Mutation Risk 0.04 .......................... 43
A.2 Comparison: Mutation Risk 0.05 .......................... 43
A.3 Comparison: Mutation Risk 0.07 .......................... 44
A.4 Comparison: Mutation Risk 0.09 .......................... 44
Abstract

Logic Regression is a relatively new method used in association mapping. It attempts to build boolean expressions that can be used as predictors. In this thesis I, will use Logic Regression in combination with Evolutionary Programming to find associations between genes and diseases in case/control binary SNP data. I will present a model using these methods and their implementation. The implemented algorithm has been used to find the optimal configuration of the parameters for both data and algorithm.

Resumé

1 Introduction

Association mapping is a well known and heavily researched area of bioinformatics. In recent years, a great number of different methodologies has been developed and some of these have been summarized by Geert Heidema et al. [Heidema et al., 2006].

The idea in association mapping is to find associations between a phenotype and one or more genes. Common phenotypes that are influenced by the human genome is eye color, blood type, and height. While the first two are only dependent on the genes the last one is also influenced by environmental factors like diet. Another group of traits that are influenced by the genes are a large range of diseases like breast cancer, diabetes, and cystic fibrosis. Carriers of these genes will not necessarily be afflicted by the associated disease, but will have a higher risk.

To be able to find an association between a gene and a given disease, we need to consider a relatively large group of individuals where some are affected and some are unaffected by the disease. The individuals are genotyped, in other words, specific information about the individuals genes, and thereby DNA sequences, is obtained.

The data obtained can be seen as a matrix where the rows contain the genetic information of a single individual and the columns represent information on the status of a single nucleotide in the genome.

![Genetic Matrix Example](image)

Figure 1.1: In this example the individuals have been classified and genotyped. The sign - represents nucleotides that are identical for all individuals, while the A, T, C, and G show the nucleotides that are varying between the individuals.
The idea is to find one or more columns in this matrix that, in combination gives the best explanation of the phenotype. It is infeasible to use an exhaustive search method since there are $2^c$ possible combinations, where $c$ is the number of columns and when using the whole human genome the number comes close to 3 billions.

The genetic variation between two individuals is very small compared to the complete genome. This is exploited when selecting the nucleotides to use in the analysis. When looking at the human genome, the number of variations in single nucleotides is approximately 10 million nucleotides. These variations are called Single Nucleotide Polymorphism (SNP)\(^1\) and are used to identify a large fraction of the genetic diversity in the human species.

The number of datapoints is further reduced by exploiting the fact that the SNPs are correlated so when selecting a few of the SNPs we also get information on nucleotides around those. This correlation is called Linkage Disequilibrium (LD), and the effect of this is widely used in finding associations between selected nucleotides.

In this thesis, I will describe an implemented model that uses Logic Regression to find association between genes and diseases. This is done by building logic expressions that combine columns from the mentioned matrix by logic operators. The result from evaluating the expression is used to calculate the correlation between the phenotype and the expression.

Logic Regression have been used in combination with a Markov Chain Monte Carlo (MCMC) [Kooperberg and Ruczinski, 2005], and Simulated Annealing [Kooperberg et al., 2001] to identify interaction between SNPs.

The model I will present is using an Evolutionary Algorithm (EA) to evolve logic expressions over the case/control data. It is only used to locate a single trait marker that is associated with the disease, but can easily be extended to look for multiple mutations.

Section 2, will give an introduction to Logic Regression and the Evolutionary Algorithms used, followed by a description of the implementation of these in Section 3. In Section 4 the results of the experiments that have been performed to verify the usability of the algorithm will be presented and discussed. Section 5, gives some ideas for future work and a conclusion.

---

\(^1\)See [Consortium, 2005] for more information on SNP and SNP selection
2 Methods

Logic is a systematic method of coming to the wrong conclusion with confidence.

- Murphy's law

In this section, I will give an introduction to the terminology of the methodologies of Logic Regression and Evolutionary Algorithms.

First, I will give a short introduction to logic expressions, Section 2.1, followed by a description of how these expressions can be represented as trees, Section 2.2. Finally, I will describe a method which can be used to move around in the search space using the evolutionary algorithm introduced in Section 2.3. This algorithm is generating a set of expression-trees that are classified as the most fit trees found.

2.1 Logic Expressions

Logic takes care of itself; all we have to do is to look and see how it does it.

- Ludwig Wittgenstein

A logic expressions is built of logic operators, logic constants and logic variables with the value true or false. Since a logic expression can be built iteratively from smaller expressions when combined with an operator, it is easy to see that the building of logic expressions can be expressed as a context-free grammar. A grammar in Backus-Naur form\(^2\) (BNF) describing a general logic expression is shown in Figure 2.1.

The logic expressions described by the BNF in Figure 2.1 are very rich in their expression range. In the model proposed in this thesis, I have chosen to use only a subset of the expressions generated by the grammar in Figure 2.1, where the constants have been removed, only variables can be negated, and the operators xor, nand, and nor have been removed. The modified BNF can be seen in Figure 2.2.

The reason to remove the constants is that they are meaningless in this context, while the reason for not allowing the extra operators and negation of operators was a simplification of the expressions.

\(^2\)Named after John Backus and Peter Naur.
\[ <\text{exp}> ::= \text{'}(<\text{exp}> <\text{op}> <\text{exp}> \text{'}) \]
\[ \mid \text{var} \mid <\text{const}> \mid \text{neg} <\text{exp}> \]
\[ <\text{const}> ::= 1 \mid 0 \]
\[ <\text{op}> ::= \text{and} \mid \text{or} \mid \text{xor} \mid \text{nor} \mid \text{nand} \]

Figure 2.1: The BNF describing general logic expressions.

\[ <\text{exp}> ::= \text{'}(<\text{exp}> <\text{op}> <\text{exp}> \text{'}) \]
\[ \mid \text{var} \mid \text{neg} \text{var} \]
\[ <\text{op}> ::= \text{and} \mid \text{or} \]

Figure 2.2: The BNF describing the logic expression used in this thesis.

As an example of how this grammar can be used, the logic expression in Equation (2.1) has been built by combining the two expressions \( \neg X_1 \lor X_5 \) and \( (X_2 \land X_3) \land (\neg X_9 \lor X_1) \) with an \( \land \) operator. These two expressions can be seen as combinations of even smaller expression combined by operators until the expressions contain only variables or negated variables.

\[ (\neg X_1 \lor X_5) \land ((X_2 \land X_3) \land (\neg X_9 \lor X_1)) \] \hspace{1cm} (2.1)

\[ \]

2.2 Logic Trees: From Expression to Tree.

A tree is a tree - how many more do you need to look at?

- Ronald Reagan

An expression built from a BNF can be converted into a tree, and the grammar in Figure 2.2 can, in particular, be used to build a binary tree where the operators are inner nodes and the variables/negated variables are leaves. This would convert the expression shown in Equation (2.1) into the the tree shown in Figure 2.3, where the negation operator is shown as a node on its own. Logic trees are often more convenient compared to logic expressions when used in algorithms.
2.2 Logic Trees

![Logic Tree Diagram]

Figure 2.3: A logic tree equivalent to the logic expression in Equation (2.1)

When working with trees instead of expressions, I am using another terminology listed below:

**Inner node** An inner node represents the boolean operator that is combining its subtrees.

**Child** A child is a subnode of another node.

**Leaf node** A leaf node does not have any children and represents a variable.

**Parent** A parent node is a node that has either one\(^3\) or two children.

**Root** A root node is a node that does not have any parent and is representing the entrance into the tree. The result of evaluating a tree can be read directly from the root if the root node is a leaf. Otherwise, the result is found using the boolean operator on the results from evaluating its two children.

**Border node** A border node is a node that has one leaf node and one inner/border node as children.

**Sibling** A sibling is a node that has the same parent as the referring node meaning that all nodes except the root have a sibling.

---

\(^3\)The only node that can have one child is the negation node. A negation node together with its child is seen as a leaf node.
2.3 Evolutionary Algorithm

Evolution is not a force but a process. Not a cause but a law.

- John Morley

In this section I will give a short introduction to Evolutionary Programming (EP) which is the Evolutionary Algorithms (EA) used in this thesis. In Section 2.3.1, I will describe how Logic Regression is used as part of the EP, followed by Section 2.3.2 that describes a method used to evaluate trees which is also used by the EP.

The general idea in EP is maintaining a population of structures\(^4\) that undergo an evolutionary processes. These processes can be seen in Algorithm 2.1 as pseudo code.

\begin{algorithm}
\caption{Evolutionary Programming}
\begin{algorithmic}
\State population \(S = \text{initialize}(S)\)
\While {not done}
\State \(S = \text{reproduce}(S)\) \text{ // Sec. 2.3.1}
\State \(\text{evaluate}(S)\) \text{ // Sec. 2.3.2}
\State \(S = \text{selection}(S)\) \text{ // Sec. 2.3.2}
\EndWhile
\end{algorithmic}
\end{algorithm}

The first step is to generate an initial population that is the base for further evolution. Subsequently, the EP algorithm is entering a loop that is continued as long as needed. Inside the loop, three things are happening, namely a reproduction step, evaluation step, and a selection step. The reproduction step is using the current population in an asexual\(^5\) process where new structures are added to the population. Since it is an asexual process the only way to create variance between parent and offspring is by introducing mutations. In most other EA, the reproduction is performed by using a sexual process meaning that offspring is created based on two different parents. The sexual process makes it possible to also use \textit{crossover} for creating greater variance in the population. The individuals used in the reproduction step can either be selected based on a fitness score, chosen randomly, or all individuals can be used. In Section 2.3.1, a method for performing mutations on the logic trees is presented. The second step is to assign a score to the individuals in the population. The score can be calculated based on many

\(^4\)In this case a forest containing logic trees.
\(^5\)An asexual process is a process that only needs a single individual to produce an offspring.
different factors, but is used as a fitness score such that individuals with a high score either have a higher probability of being selected for further reproduction or have a higher probability of surviving. How the individuals are scored will be described in Section 2.3.2 together with a description on how the selection is performed.

2.3.1 Reproduction: Logic Regression

If you’re walking down the right path and you’re willing to keep walking, eventually you’ll make progress.

- Barack Hussein Obama

Now that the terminology is in place, I will introduce the model for which changes can be made to a given tree, representing a move in search space. A move or change can be interpreted as a mutation of the current tree and these terms will be used as one so forth.

Moving around in search space is done by making a mutation to an existing tree. There are six possible mutations that can be performed as shown in Figure 2.4. The moves used are based on the tree-growing process [Ruczinski et al., 2003].

**Alternate Leaf:** Alternate leaf substitutes an existing leaf with a new leaf or negates the existing leaf. In Figure 2.4 (b) the variable $X_9$ is alternated with the variable $X_1$.

**Alternate Operator:** Alternate operator substitutes an existing operator with another one. In Figure 2.4 (c) the $\lor$ operator is changed into an $\land$ operator.

**Delete Leaf:** Deleting a leaf causes the parent to become obsolete, and it is therefore replaced with the sibling of the deleted leaf. In Figure 2.4 (d) the leaf containing the variable $X_9$ is deleted together with its parent, and $X_3$ replaces its parent's role as child of the root.

**Split Leaf:** Split leaf is the opposite of delete leaf. The selected leaf is replaced with an inner node that is given the selected leaf and a new leaf as children. In Figure 2.4 (e), the variable $X_9$ is split so that it becomes a child of a new inner node with a $\lor$ together with a new leaf containing the variable $X_5$. 
(a) Initial Tree

X_2 \lor X_9 X_3

(b) Alternate Leaf

X_2 \lor X_1 X_3

(c) Alternate Operator

X_2 \lor X_9 X_3

(d) Delete Leaf

X_2 \lor X_3

X_5 X_9

(e) Split Leaf

X_2 \lor X_7

X_9 X_3

(f) Grow Branch

X_9 X_3

X_9 X_3

(g) Prune Branch

Figure 2.4: Possible mutations that can be performed on a tree. (a) shows the initial tree and the other figures show the result of the different operations.

Grow Branch: Grow branch replaces the selected inner node with a new inner node which has the selected and a new random leaf as children. In Figure 2.4 (f) the parent of X_9 and X_3 is replaced by a new \land operator that as children has the replaced inner node and a new leaf with the variable X_7.

Prune Branch: Prune branch is the opposite of a Grow Branch move. A border node is removed together with its leaf child. The sibling of the removed leaf replaces the removed border node. In Figure 2.4 (g) the inner node having the leaf node X_2 as child is removed together with the leaf X_2 and the right sub tree is becoming the root instead.
2.3.2 Evaluation: Scoring Trees

To be able to select trees for further processing I need a way to assign a score to the trees in the forest. The score can be seen as a fitness value which tells how fit the tree is. This fitness can be used to select trees from the forest based on either probability or an ordering of the trees. In the case where probability is used, a tree with a high fitness will have higher probability of being selected for further processing; and in the case where an ordering is used, the trees with the lowest score are removed until the forest has a certain size.

I use the ordering strategy such that only the best scoring trees are kept in the forest. The fitness of the trees is calculated by the use of a modified Pearson's chi-square test, also known as $\chi^2$ test. The modification is introduced into the calculation to avoid overfitting the data which could happen when the trees grow larger. Therefore the size of the tree is taken into account by punishing the score by a factor depending on the number of leaves in the tree. The resulting function used for calculating the score can be seen in Equation (2.2) where $\beta$ is a variable used to tune the function depending on the data.

$$f(\vec{x}) = \log(\chi^2(\vec{x})) - \beta \cdot \log(\#leaves) \tag{2.2}$$

The function takes $\vec{x}$ as input which represents a $2 \times 2$ matrix encompassing the results of evaluating the tree on each of the individuals in the case/control data set combined with the phenotype of the individual. The nature of the entries in the matrix can be seen in Figure 2.5 where $N$ is the total number of individuals in the data set, $k$ the number of affected and $l$ the number of unaffected individuals, $n$ the number of times the given tree evaluated true, and $m$ the number of times it evaluated to false.

<table>
<thead>
<tr>
<th></th>
<th>affected</th>
<th>unaffected</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>$x_0$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>false</td>
<td>$x_2$</td>
<td>$x_3$</td>
</tr>
<tr>
<td></td>
<td>$k$</td>
<td>$l$</td>
</tr>
</tbody>
</table>

Figure 2.5: Matrix containing the evaluation of the trees combined with the true phenotype.
3 Implementation Details

The proposed model and the evolutionary algorithm have been implemented in order to perform experiments and evaluate how well they perform. The implementation has been done in C++ and I will assume in the rest of this section that the reader is well versed in this language. The code can be downloaded from the following URL:


The tar ball contains, in addition to the code, an instruction on how to compile the code in the file README.txt. It also contains a few small programs that have been used to generate data and results. These programs are also described in the readme file.

In the following, I will go into details with the general idea of the algorithm and thereafter go into greater details with the more important and interesting implemented classes.

The general idea of the implemented program was to run the evolutionary algorithm described earlier and shown with pseudo code in Algorithm 3.1. The algorithm is implemented in a class called Sampler that will be described in Section 3.1. Thereafter, in Section 3.2, a description of how data is handled is presented followed by a description of the tree structure, Section 3.3, representing the individuals in the forest. Lastly, in Section 3.4, I give a description of the class that is used to introduce mutations to the individuals when expanding the forest.

Algorithm 3.1 Main algorithm

| Forest $S = \text{initialize(data)}$ |
| for $i = 0$ to numbers of iterations do |
| 
| $S = \text{expand}(S,\text{data})$ |
| $\text{evaluate}(S)$ |
| $S = \text{scythe}(S)$ |
| end for |
| analyse($S,\text{data}$) |
Before starting the algorithm, the data must be read and initialized, and various variables must be set. The variables are used to control termination and size of the forest. The variables that need to be initialized are described below:

**Forest size:** The size of the initial forest.

**Sampling size:** The size of the expanded forest.

**Iterations:** The number of iterations.

Since this is not directly a part of the evolutionary algorithm, it has not been included in Algorithm 3.1.

The Sampler is used by giving it the initialized data as input and it will thereafter calculate a resulting forest based on this data. After the algorithm has finished its calculations, the forest can be collected and analyzed. Originally, analysis of the resulting forest was part of the Sampler, but it has been removed and is now done by an external program.

### 3.1 Sampler

The algorithm consists of a simple initialization of the forest, $S$, followed by a calculation of the resulting forest which is subsequently analyzed. The algorithm is implemented in a class called Sampler that has the following signature:

```cpp
template<typename model_t, typename tree_t, typename data_t, typename eval_policy>
class Sampler
```

The forest, $S$, is initialized by creating a predefined number of trees consisting of a single leaf. These single leaf trees are required to be unique, meaning that two leaves cannot have the same value, except if one of them has been negated and the other one has not. Subsequently, the for loop runs a predefined number of iterations where three steps occur in each iteration.

The first step is to expand the forest to a predefined size. This expansion is based on the current forest, where one of the trees in the forest is used to create a copy modified by a single mutation. The mutation performed is one of the mutations described in Section 2.3.1. The tree used for this operation is selected iteratively, so when adding the $i$'th new tree to $S$ the $i$'th tree from $S$ is used. The newly created tree might be identical to an already existing tree. In this case, the created tree will be discarded and a replacement will be created instead. The method to compare two trees is described
in Section 3.3 together with the tree structure. To create a new tree I have implemented a separate class called \texttt{trModel} which is described in Section 3.4.

The second step is the evaluation of the forest giving a score for each tree. The score is calculated by filling out the $2 \times 2$ matrix shown in Figure 2.5, based on the phenotypes and the evaluation of the trees on the data. Then the $\chi^2$ function from Equation (2.2) is used on that matrix to get the resulting score. The implementation of the 	exttt{Sampler} makes it easy to change the calculation of the score. This is achieved by letting the evaluation be an \texttt{Evaluator\_policy} class which is one of the template parameters of the \texttt{Sampler} class. The reason for this is to able to use the more exact, but also more time consuming, function \textit{Fisher's exact test} in cases where needed, or even to use a completely different evaluation strategy. Another reason is to allow other ways to modify the result calculated by $\chi^2$, and a \texttt{policy\_class} would make this very convenient.

The third step is the removal of the unfit trees from the forest. The trees are sorted based on their calculated score from step two, and the trees with the lowest scores are removed until the forest is reduced to its original size. When sorting the trees, it may happen that two trees have the same score and in this case the tree with fewer nodes is chosen as the best scoring tree. If the size of the two trees cannot determine the ordering, I use a \texttt{toString()} function on the two trees and compare the two strings using a standard lexical ordering. This gives a strict weak ordering that is needed in the implementation.

\subsection*{3.2 Data}

I made the choice to make as much of the code independent of the data as possible, when starting out with the implementation of a data handler. This choice was made since I had no way of knowing what data I would be working with in the end of the experimental process. By making the rest of the code independent of the data, it was easy to implement a new data handler when the need arose and insert it into the existing code. For this to work out properly, I made the assumption that all the data I would be working with contained information on a set of individuals that each have a phenotype and a list of ordered, genotyped SNPs. The data also contains information about the position of each SNP. This is a prudent assumption, since if the data did not include this information, the whole algorithm would have to be reimplemented and the model redesigned.
The sampler is not the only part of the program that needs information from the data class. After generating the resulting forest, the analyzer must analyze it based on the forest and some extra information from the data. The information needed is dependent on the data type and the analysis wanted. The data used in the experiments must, apart from the basic information, also contain information on the trait marker position, which is used in calculating the distance between the trait marker and the candidate marker.

To handle the data in the performed experiments, I implemented a class called SNPData. This class not only implement methods to read in and initialize the data, it must also implement the implicit interface defined by the Sample class. The Sampler needs access to phenotypes, the SNPs for each individual, and information on the number of SNPs. The phenotype information is found by a simple look up while the SNPs are accessed through a wrapper that supports indexed look ups. Since the SNP data can be seen as a matrix where the SNPs for a single individual is corresponding to a matrix row, the wrapper I have implemented is called Matrix_row. This structure is a simple struct with a pointer to the SNP data and a row index.

## 3.3 Tree Structure

The tree structure is perhaps the most important structure since it is the core of the model described earlier. The tree structure should support the binary structure described in Section 2.2 and should be able to evaluate a single individual by storing indices in its leaves that point into the Matrix_row.

Flexibility is ensured by implementing the tree as a template class that takes two template arguments. The first argument is the type of data it should evaluate which in the final experiments was the Matrix_row, and the second parameter is a trait marker that is used to define how the data should be evaluated. The class has the following signature:

```
template<row_t, eval_trait> class Tree
```

The tree is implemented as a binary tree consisting of the three node types inner, leaf and negate where the values in the leaves are index variables that map uniquely into a row_t object. The inner node contains two subtrees and a logical operator which defines how the results from the subtrees should be evaluated. This logical operator can easily be exchanged by a new operator type and is implemented as a function class that is called with two boolean values and returns a boolean value. The negate node is a special node that has a single child and is used to propagate the negated
value of its child upward. I have chosen that the negate node can only be the parent of a leaf.

The class implements a range of functions where only one of these is directly used by the Sampler, namely bool evaluate(const row_t&). This function is used to calculate the result of the tree based on the given individual given by the row_t object. The function is implemented as a recursive function that works its way down the tree. The inner node is returning its operator’s evaluation of the result from the evaluation of its children and the negate is returning the negated result of the evaluation of its lonely child. The leaf node is using the eval_trait class together with its leaf value to return the correct value from the row_t object. The reason for the eval_trait class is that data in the row_t do not have to be directly convertible into a boolean value. As an example, if the row_t object consists of two genotyped vectors of SNPs, the trait class could be either implementing a recessive model or a dominant model and thereby return different results on the same data.

The trees are implemented so that they can be compared for equality, a property needed by the Sampler. This is achieved by sorting the children of the inner nodes such that the child containing the lowest leaf value is set as the left child. Performing this recursively down the tree makes the tree comparable in \(O(\#\text{leaves})\). It should be noted that this comparison of the trees only takes the topology of the tree into account meaning that two trees that evaluate to the same result on all inputs are not necessarily considered equal. An example of this can be seen in Figure 3.1.

\[\text{It supports only equal and not equal, but not less than and greater than.}\]

\[
\begin{array}{c}
  \lor \\
  X_1 \\
  \lor \\
  X_2 \\
  \lor \\
  X_3 \\
\end{array}
\quad
\begin{array}{c}
  \lor \\
  X_2 \\
  \lor \\
  X_1 \\
  \lor \\
  X_3 \\
\end{array}
\quad
\begin{array}{c}
  \lor \\
  X_3 \\
  \lor \\
  X_2 \\
  \lor \\
  X_1 \\
\end{array}
\]

(a) \quad (b) \quad (c)

Figure 3.1: Evaluating the three trees with the same values for \(X_1, X_2\) and \(X_3\) will result in the same result. But only (a) and (c) will be considered equal when using the implemented compare function. If \(X_1 < X_2 < X_3\) the tree in (c) will have all its children swapped in such a way that it will be structured in the same way as (a).
3.4 Mutation Model

The Sampler uses the mutation model to create a new tree based on an already existing tree to expand its forest. The mutation model \texttt{lrModel} is an implementation of the logic regression model described in Section 2.3.1. It is using a simple visitor pattern to recursively copy the input tree, but while doing so selecting a single node and mutating it. When copy and mutation has been performed, the resulting tree is returned to the Sampler. The \texttt{lrModel} has the following signature:

\begin{verbatim}
template<row_t, eval_trait, select_policy> lrModel
\end{verbatim}

Each node in the tree, except the negation nodes\(^7\), is chosen with equal probability \(\frac{1}{n}\), where \(n\) is the total number of inner and leaf nodes. The type of mutation introduced into the new tree is dependent on the selected node as mentioned in Section 2.3.1. When a node is selected, its type is classified. The possible node types are inner nodes, border nodes or leaf nodes. Based on the classification, a possible mutation is selected with equal probability for that node type. The possible mutations can be seen in Section 2.3.1 and are summarized below:

**Inner node:** Alternate operator and grow branch.

**Border node:** Alternate operator, grow branch and prune branch.

**Leaf Node:** Alternate value, split leaf and delete leaf.

The reason that the prune branch mutation is not allowed on the inner node is that this would cause the complete removal of a subtree. This mutation cannot be reversed in a single mutation and has been disallowed.

I have also chosen to enforce the following invariants when performing the mutations:

1. All leaves in a tree must be unique, meaning that any variable can only exist once in one of the two forms (negated or not negated).

2. Alternating a leaf must result in a value different from the original. The new value can be the negation of the original value.

3. Alternating an operator must result in an operator different from the original.

\(^7\)The negation node is not seen as a node by itself but seen as part of its child node instead.
3.4 Mutation Model

The first invariant eliminates the possibility of constructing strange expressions like $X_1 \land \neg X_1$, which would always be false. The last two invariants ensures progress by preventing that the original tree is returned to the Sampler. A perfect copy would, as mentioned earlier, be discarded when inserting it into the forest, so it would be a waste of time to allow it. It can of course not ensure that the \texttt{lrModel} always returns a tree that will be accepted, since two almost identical trees could be mutated into the same tree in the process.

I use the last template parameter \texttt{select\_policy}, when having to select a new variable for either a new leaf or a leaf alternation. The policy is deciding how a value is selected, and I have been working with two different types of policies. The first one is called \texttt{Shutgun\_policy} and selects a variable from the set of possible values with equal likelihood. The second is called \texttt{Sniper\_policy} and selects a value based on a normal distribution around a mean value. This means that values close to the mean value have a higher probability of being selected than values far away. In the experiments, I have only used the \texttt{Shutgun\_policy} since preliminary experiments showed that it performed significantly better than the \texttt{Sniper\_policy}.
4 Experimental Results and Discussion

Now that the algorithm has been presented and the models have been described, it is time to find out how well the algorithm performs on data. A range of experiments have been suggested, setup, and analyzed. In the section for each specific experiment, I start out by describing what is investigated followed by what the expected result would be. Subsequently, a description of what the experiments showed and a discussion of it follows.

All data used in the experiments I have performed has been simulated. The reason for this is that I wanted data containing information on the trait marker and the data-specific variables. The information I needed was: trait marker position; trait marker genotype information; ratio between wildtype risk, $w$, and mutation risk, $m$; and recombination rate. The data used has been simulated with the program Coasim [Mailund et al., 2005], which generated three files containing information on haplotype, positions, and trait marker position. More details on how data has been simulated is given in Section 4.1.

The algorithm was setup to calculate the resulting forest for each data set. Subsequently, the resulting forests were analyzed by a secondary program which made it possible to analyze the results with different models. The analysis is used to find a candidate SNP for being the trait marker and to determine its distance to the correct trait marker. There has been generated a large number of data sets for each configuration of variables used in the data simulation. The results from the data sets are plotted as a curve where the x-axis is the distance between the candidate SNP and the trait marker and the y-axis is the accumulated number of data-sets. See Section 4.2 for more details on how the analysis are performed.

The experiments have been divided into two groups. The first group of experiments are performed on data that only contains information on the position of the trait marker. These experiments are used to determine how to setup the algorithm to obtain the best results and to see how the data parameters influence the results. The second group of experiments are performed to investigate the potential of the algorithm. To be able to perform these experiment, I assume knowledge of the genotypes for the trait marker when performing the analyzing of the results. The experiments assuming that the position of the trait marker is known is found in Section 4.3 and the experiments investigates the potential can be found in Section 4.4.

After having presented the results for the various experiments, I will give a recapitulation and a discussion of the results.
4.1 Simulation of Data

To simulate data with help from CoaSim, I have implemented a small script in Python which run the simulation with various settings. For the simulation I need to set five variables, where three of these are constants and the last two can be varied to generate a large range of different data sets.

The three constants are the number of individuals, the number of SNPs, and the wildtype risk, $w$. The number of individuals is set to 10,000 such that there is generated a large number of both cases and controls. I demand from both case and controls that they amount to at least 4% of the individuals, otherwise the simulation is discarded and rerun. This is to ensure a sufficient statistical confidence in the data. The number of SNPs is set to 250 which was chosen to give enough SNPs to work with, but still not too many to slow down the program too much. The last constant is the wildtype risk, which represents the probability that a given individual with the wildtype genotype $a^8$ is affected by the given disease. This value is set to 0.04 which is close to the frequency of several real diseases.

The two variables that are used to set up a simulation are the mutation risk, $m$, and the recombination rate, $\rho$. There is also a third variable, but it is an index variable used to distinguish different data sets that otherwise have been simulated with the same configuration. The mutation risk is the risk for a individual with the genotype $A$ to be affected with a given disease. This is, together with the wildtype risk, used to classify the individuals in the simulation based on the trait marker. The mutation risk $m$ is varied over the interval $[0.4, 0.9]^9$ with a step of 0.1. The higher the recombination rate, the weaker LD becomes. The variable $\rho$ is varied over the interval $[50, 200]$ in steps of 50.

4.2 Analysis of Results

The algorithm is used to generate a forest which is containing those trees with the highest scoring seen. This forest has to be analyzed such that a candidate SNP can be selected. To achieve this, the algorithm stores the resulting forest in a file which is used as input in a second program that only has to analyze the forest. The analysis program calculates a score for each SNP and then the SNP with the highest score is selected as the candidate

---

8 The genotype $a$ is the wildtype, while $A$ is mutant type.

9 A larger range has been used in preliminary experiments, but these showed that the effect of a larger $m$ was very small.
trait marker. The reason for using a second program to analyze the forests instead of using the main program is that I want to be able to try different scoring models on the same forests.

The score, $S$, for each SNP is calculated by Equation (4.1) and is an accumulation of weighted correlations between the trees in the forest and the SNP. The correlation between the SNP and a tree is calculated by the Equation (4.2).

$$S_i = \sum_{n=0}^{N-1} W_{n,i}(r_{n,i}^2(tree_n, SNP_i))$$ \hspace{1cm} (4.1)

$$r_{X,Y}^2 = \frac{E((X - \mu_X)(Y - \mu_Y))^2}{(\mu_X - \mu_X^2) \cdot (\mu_Y - \mu_Y^2)}$$ \hspace{1cm} (4.2)

The weight function $W$ has been varied in an attempt to obtain different results. This function can be a simple selector function that only returns the score of a specific tree or a more complex function that varies the weight of each tree based on various states.

Examples of functions I have tried are a simple selector function that only uses the calculation of the first tree, a function weighed by the position of the tree in the forest, and a function that is weighed by the unmodified score of the tree. The simple version was discarded as being to unreliable since the result varied too much. The last two functions performed almost equally well, so I could use any of them.

### 4.3 Experiments

The experiments in this section are run to evaluate the influence of four different variables on the performance of the algorithm. Two of these are set while simulating the data and the last two are set in the algorithm. The two data variables are the mutation risk $m \in [0.04, 0.09]$ and the recombination rate $\rho \in [50, 200]$. The variables for the algorithm are the number of iterations and the variable $\beta \in [0.2, 1.0]$ from Equation (2.2) which is used to punish the score based on the size of the tree.

First I will describe the experiments that have been performed to investigate some of the parameters that can be varied while running the algorithm. The two parameters that have been investigated are $\beta$ (see Section 4.3.1), a parameter used when evaluating the trees, and the number of iterations (see Section 4.3.2), that the algorithm should use to get the final results.
Secondly I will describe the experiments that are performed to see how the data influenced the performance of the algorithm. This is achieved by varying the two data parameters recombination rate \( \rho \) (see Section 4.3.3), and the mutation risk \( m \) (see Section 4.3.4), while generating the data.

Lastly, to see how well the algorithm performs compared to other known methods, I have performed an experiment that compares my implemented algorithm with two other programs. The first program \textit{chisquared}\footnote{This program is implemented by using the implemented data handler which has a method calculating the \( \chi^2 \) value for each marker.} used a very simple method based on a \( \chi^2 \) calculation, while the second program, \textit{blossac} [Mailund et al., 2006] used a more advanced method. In the simple case, the program calculated the \( \chi^2 \) for each SNP, and the SNP with the highest score was selected as the candidate trait marker. \textit{Blossac} is an association mapping tool that uses linkage disequilibrium and genealogies to find candidates for containing disease affecting variation. Here again, I use the results from the program to find the highest scoring marker and calculate the distance to the real marker. In the case of a tie between two candidate markers, the one that is closest to the real marker is chosen.

Examples of other algorithm parameters that could have been investigated are the forest size, the different probabilities for the possible mutations, inclusion of extra operators in the trees, and extensions to the mutation model.

Examples of other data experiments that could have been performed is a variation in the number of SNP, individuals and a variation in the ratio between case and controls.

4.3.1 Experiment: Score Function Modification (\( \beta \))

This experiment is designed to show how the variable \( \beta \) in Equation (2.2) influenced the algorithms performance. The variable is used to determine to how large degree the size of the trees are influencing the score when evaluating the forest. A low value means that the size does not influence the score very much, which again means that the trees are allowed to grow larger than with a high value. It is expected that large trees have a higher risk of overfitting the data and thereby are less likely to find a good candidate SNP. Avoiding large trees also prevents unnecessary time consumption since most of the time is spent on copying the tree in the mutation phase.
The experiment is set up to run on the same data sets with different values of \( \beta \). Preliminary experiments had shown that with \( \beta = 0.5 \) the algorithm was sufficiently fast and gave forests that have an average tree size of six nodes which is considered as a wellformed tree. To see if other values would have been more appropriate, an interval was selected such that \( \beta \in [0.2, 1.0] \) with steps of 0.2. Selected results of these runs can be seen in Figure 4.1 and Figure 4.2.

The analysis of the results showed that the best results for \( \beta \) depended on the recombination rate \( \rho \) and to a smaller extend by the mutation risk \( m \).

It is noticed that the results get better when \( \beta \) increases, when looking at the result in general. Especially, when the recombination rate increases, the results get better with a high \( \beta \). An example of this can be seen in Figure 4.1 and Figure 4.2. The two plots show the result for two experiments that are run with different values of the recombination rate. It can be seen that the results for the runs with a \( \beta \) lower than 0.5 is in, both cases, below the rest, but in the plot with the higher recombination rate the difference is much more clear. This is a general tendency observed in most experiments.

For very low values of \( \rho \) it has been observed that values of \( \beta \) lower than 0.5 perform equally well compared to values of \( \beta \) higher then 0.5. This indi-

Figure 4.1: Results of the experiments that have been run to see the influence of \( \beta \) on data simulated with a high recombination rate, \( \rho \). There is a clear difference between the results with a \( \beta \) lower than 0.5 and those with a higher \( \beta \).
Figure 4.2: Results of the experiments evaluating the influence of $\beta$ on data simulated with a low recombination rate, $\rho$. Compared to Figure 4.1 the difference between the results is not as clear but there still can be seen a difference between the results with $\beta$ lower and higher than 0.5.

cates that the value of $\beta$ can be set to any value around 0.5. It has also been noticed that when the value of $\beta$ is set too high the results have a tendency to vary more often from run to run. This behavior starts for $\beta$ larger than 0.6 for low values of $\rho$ and for $\beta$ larger than 0.8 for high values of $\rho$.

The mutation risk, $m$, also seemed to have a small influence on the results. The influence of $m$ was very small and was often not detectable. When $m$ increased, the results with a higher $\beta$ seemed to give a better result than when $\beta$ was low.

4.3.2 Experiment: Iterations

This experiment is performed to determine how many iterations is needed to obtain the best results. Further, it is used to see if the results would start to degenerate when the algorithm is allowed to continue for a long time. The experiment are run as normal, but instead of only printing the resulting forest, the algorithm prints the current forest after a set number of iterations before continuing. In this way, it is possible to follow the development of the results.

The experiment is run in two steps, one with a low iteration step to
4.3 Experiments

determine the number of iterations required and another experiment with larger iteration steps to investigate possible degeneration. The experiment with small iteration steps performed 200 iterations and stored the forest after every 10 iterations. The other experiment performed 1,500 iterations and stored the forest after every 100 iterations. The two experiments were set up with $\beta = 0.5$ and were running on data simulated with $\rho = 100$ and $m = 0.08$. The resulting forests were analyzed and plotted (see Figure 4.3 and 4.4).

The expectation was that the result would get better and better until a certain point where the result would stay fixed. Further, it was expected that the algorithm would stay fixed or only vary a little when the algorithm was allowed to continue for a long time.

The two plots in Figure 4.3 show the results of the experiment with analysis for every 10 iterations. The plots show, as expected, that the number of iterations has a impact on the results. They also show that the number of iterations needed for the results to converge is around 50-60. After this, the results still change, but not significantly.

The two plots in Figure 4.4 show the result of the experiment investigating for degeneration. The plots show that the algorithm do not degenerate, though it can be seen that the algorithm still makes small changes in the forest.

The results are, naturally, dependent on how the algorithm is setup, especially the size of the forest has a great impact on how fast the results converge. In the general case, the initial forest size is set to 50 and expanded to double size and subsequently reduced to the initially size. This setup ensured that the results converged around 50 iterations or after 2,500 trees\textsuperscript{11} have been generated, which is $10 \cdot \#SNPs$ in the data. The association between number of SNPs and the required number of iterations has not been investigated further, but an association between them is expected.

\textsuperscript{11}Not necessarily unique trees.
Figure 4.3: (a) The plot show the results of the analysis the algorithm after every 10th iteration. (b) Details of (a).
Figure 4.4: (a) The plot show the results of the analysis the algorithm after every 100th iteration. (b) Details of (a).
4.3.3 Experiment: Recombination Rate ($\rho$)

The influence of the recombination rate on the algorithm is investigated by simulating data with values of $\rho$ varying in the interval [50, 200].

In the case where the trait marker is included in the data, preliminary experiments showed that the algorithm worked very well in all cases where mutation risk was higher than wildtype risk, even though it performed slightly better on data simulated with a high recombination rate. The algorithm found the correct marker in up to 90% of the data sets.

Data that have the trait marker excluded is expected to give the best results when the LD is high. It is therefore expected that when there is a high recombination rate the algorithm would performed worse than when the recombination rate was low.

Figure 4.5 shows the results of four different data sets that have been generated with varying values of $\rho$. The trend is a reduction in result quality when $\rho$ is growing. The data set with $\rho = 50$ does not follow this trend in the plot and this behavior have also been seen for other values of $\rho$ in other setups. These abnormalities can be caused by normal fluctuations in experiments and data, and as can also be seen from Figure 4.5 that the variation among the curves is not very large. The conclusion is that $\rho$ is affecting the results by giving slightly better result when data is simulated with a low recombination rate.

![Experiments with varying Rho: Beta = 0.5](image)

Figure 4.5: The plot shows the results of the experiment with varying values of the recombination rate. The tendency is that the results gets better with a low recombination rate.
4.3.4 Experiment: Mutation Risk Variable ($m$)

The mutation risk $m$ is the last variable that has been investigated. It was used when simulating the data and is representing the risk that an individual with the mutated allele in the trait marker is affected by the corresponding disease.

It was expected that this parameter had great impact on the results, because a higher mutation risk would make it easier to find a good candidate SNP. The reason for this is that when the mutation risk increases the number of affected individuals that have the mutation allele also increases. At the same time the number of affected individuals with the the wildtype allele remains constant and therefore the mutation allele gets easier to locate statistical. An example of a single run with $\rho = 100$, $\beta = 0.5$ and wildtype risk $w = 0.04$ can be seen in Figure 4.6.

It shows that the algorithm, as expected, gives better result when the mutation risk increased. Preliminary experiments run to investigate the influence of $m$ showed that the improvement of the results would not become much better than the results shown in Figure 4.6. This effect can already be seen in the plot by comparing the two last curves that are the results for the runs with $m = 0.08$ and $m = 0.09$.

![Experiments with varying Mutation Risk: Beta = 0.5, Rho = 100](image)

Figure 4.6: The plot shows the results of the experiment with various values of the mutation risk. As can be seen the results get better when the mutation risk increases.
4.3.5 Experiment: Comparison to Other Methods.

The last experiment performed is an experiment performed to see how well the implemented algorithm performed compared to other known methods. The two methods that have been selected is Blossoc [Mailund et al., 2006] and a simple single marker score test where the $\chi^2$ value is calculated for each marker and the marker with the highest score is selected as the candidate.

The data used for both programs was the exact same data used for my implemented algorithm, such that the results from the three programs is directly comparable, and these are shown in Figure 4.7.

The plots show that Blossoc is the program that that is best at finding good candidate SNPs, while chisquared the worst. Plots showing the results for mutation risk equal to 0.04, 0.05, 0.07, and 0.09 can be found in Appendix A in Figure A.1 to Figure A.4.

4.4 Investigation of Potential

In order to explore the potential of the implemented algorithm I allow the implemented analyzer to select the trait marker as the candidate SNP. This shows how correlated the forest is to the trait marker. The forest are generated on data that did not have the information on the trait marker included, but are expected to contain trees that is correlated to the trait marker. The result of this can be seen in Figure 4.8 and shows that only in a few cases the trait marker is selected.

The results shown in Figure 4.8 did not give great confidence in the algorithm since the trait marker was only rarely selected. Therefore I ran a few experiments that is supposed to show how strongly the forest is correlated to the trait marker.

The experiment in Section 4.4.1 is run to investigate the dependency between mutation risk and the correlation between forest and trait marker. Thereafter, in Section 4.4.2, an experiment is described that is run to investigate the dependency between $\beta$ and the correlation between trait marker and forest.

In the following experiments I will use a new terminology called Ranking. This is a value assigned to the trait marker and is simply a value based on the number of SNPs that would be selected before the trait marker. The value is found by calculating the score, as mentioned in Section 4.2, for each
4.4 Investigation of Potential

Figure 4.7: The plots shows the result of the experiment that compares the three different methods for scoring the SNPs for different mutation risk values. The figures show the same result, namely that Blossoc is performing better than Logic Regression that in turn performs better than ChiSquared.
<table>
<thead>
<tr>
<th>Mutation Risk</th>
<th>Trait Marker Selected %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.04</td>
<td>0</td>
</tr>
<tr>
<td>0.05</td>
<td>0</td>
</tr>
<tr>
<td>0.06</td>
<td>10</td>
</tr>
<tr>
<td>0.07</td>
<td>9</td>
</tr>
<tr>
<td>0.08</td>
<td>18</td>
</tr>
<tr>
<td>0.09</td>
<td>17</td>
</tr>
</tbody>
</table>

Figure 4.8: The table shows the percentage of the data sets where the analysis picked the trait marker as the candidate SNP when the trait marker was included in the analysis but excluded in the generation of the forest.

SNP including the trait marker and sorting the SNPs based on this score. The rank of the trait marker is then its index in this sorted list.

### 4.4.1 Experiment: Mutation Risk and Ranking

This experiment is setup to calculate the ranking for the trait marker and then plot the accumulated number of data sets that have an equal or lower ranking. This has been done for six different data sets with varying mutation risk $m$. The result of this is shown in Figure 4.9:

The plot in Figure 4.9 shows that the forest seems to be very well correlated with the trait marker even though that the table in Figure 4.8 showed differently. Especially for the data with a mutation rate above 0.07 it can be seen that if the algorithm was allowed to select a candidate SNP between the first 10% of the best scoring SNPs the trait marker could be selected in at least 99% of the cases. If only allowing the algorithm to select among the 2% best (five SNPs) scoring SNPs the trait marker could be selected in at least 85% of the data sets.

### 4.4.2 Experiment: $\beta$ and Ranking

This experiment is run to see how $\beta$ is influencing the ranking result. This is done by running the algorithm on the same data, but with various values of $\beta$. Thereafter the first ten rankings are used for plotting. I run this experiment for two different data sets with different values of the recombination rate. The mutation rate $m$ is in both cases set to 0.08 and the algorithm was performing 100 iterations.
4.4 Investigation of Potential

The result for the data set with $\rho = 100$ is shown in Figure 4.10 and for $\rho = 200$ in Figure 4.11. In the plots, the average tree size is also shown. In Figure 4.10 the average tree size for $\beta = 0.0$ is 24.66, and in Figure 4.11 it is 25.49.

The plot in Figure 4.10 shows that the algorithm has the largest potential when $\beta$ is close to 0.0. When only allowed to use the best scoring SNP, the plot shows that the best results are achieved with $\beta = 0.0$ while this value should be higher when allowed to use more SNP to choose from. The value of $\beta$ should be chosen in the interval $[0.0, 0.6]$ dependent on the ranking used. When looking at the last few curves with rank eight and nine this interval seems to include also 0.0.

The plot in Figure 4.11 shows almost the same as Figure 4.10, except that $\beta$ should be chosen in the interval $[0.2, 0.8]$, and that in the case with rank 0 the value of $\beta$ should be closer to 0.2 instead of 0.0.

Both plots shows is that when setting the $\beta$ low the size of the tree is rising drastically as can be seen by the average tree size curve.

![Figure 4.9](image_url)

Figure 4.9: The plots show how the accumulated number of data sets depends on the rank of the trait marker. The plot only shows the first 10% of the possible ranking values.
Figure 4.10: The plot shows how the ranking and average tree size is influenced by $\beta$. The data used was simulated with a recombination rate $\rho = 100$, and mutation risk $m = 0.08$.

Figure 4.11: The plot shows how the ranking and the average tree size is influenced by $\beta$. The data used was simulated with a recombination rate $\rho = 200$, and mutation risk $m = 0.08$. 
4.5 Discussion

Comparing the results for $\beta$ in Section 4.3.1 and Section 4.4.2 shows that they do not agree completely on the interval $\beta$ should be selected from. When looking at the results from the experiments performed with recombination rate $\rho = 100$, the interval in the first case was found to be $[0.5, 0.8]$ while in the latter case it was found to be $[0.0, 0.6]$.

First of all, the results cannot be directly compared since the later experiment does not tell anything about which SNP would have been chosen instead of the trait marker. That experiment can only be used to conclude that the forest generated is in fact highly correlated to the trait marker. At the same time it can be used to see that the correlation is highest when $\beta$ is close to 0.0. The result in Section 4.3.1 hints that the algorithm does find a forest that is very highly correlated with the trait marker when using a very low $\beta$. But the earlier result hints that overfitting is occurring, since the forest is not necessarily highly correlated to the SNPs close to the trait marker. This fits very well with hypothesis that the tree size, which can be seen as the complexity of the model, has to be kept low to avoid this overfitting.

In the result from the experiment exploring the effect of the recombination rate, $\rho$, it can be seen that the result for $\rho = 50$ does not follow the trend of the other results. This behavior has also been observed in other experiments performed to investigate $\rho$. The reason for this is that with a small recombination rate it is hard to distinguish between the SNPs close to and those further away from the trait marker. Another reason can be variations in the results.
5 Conclusion and Future Work

In this section I will recapitulate the results and give an evaluation of the designed model described and analyzed in the previous sections. Subsequently, I will try to give an idea catalog of possible extensions to the described model and ideas to potential improvements inside the framework of logic regression.

5.1 Conclusion

In this thesis I have described a model using logic regression and an evolutionary algorithm that can be used to find disease trait markers. Furthermore, I have implemented the algorithm and the model and performed a number of experiments.

The experiments showed that the best setup for the algorithm was to set \( \beta \) to a value in the interval \( [0.3, 0.8] \) dependent on the recombination rate. Since the recombination rate is not ensured to be known, \( \beta = 0.5 \) is a good choice for the values of \( \rho \) and \( m \) used in the experiments. A better way to set \( \beta \) would be to estimate the recombination rate in the data based on the linkage disequilibrium.

The experiments also showed that the number of iterations needed is fairly low, but further experiments are needed to see if there is an association between the number of iterations and the number of SNPs. For the data used, containing 250 SNP, the experiments showed that the number of needed iterations was 50-60.

The experiments also showed that the mutation risk had the expected effect on the performance of the algorithm namely that when the ratio of the risk, \( R = \frac{\text{mutation risk}}{\text{wildtype risk}} \), became larger, the algorithm produced better results.

The experiments also showed a performance decrease for the algorithm, when the recombination was large. This was expected since the recombination affects the correlation between the SNPs in the regions of the data where recombination happens.

Finally, the experiments show that the implemented algorithm seems to has potential to perform better. As the model is at the moment the forest is too well correlated to the data that have been used to generate it. It had been expected that the forests would be stronger correlated to the trait marker since the trees were supposed to be selecting the SNPs that were correlated to the trait marker by the LD effect. The results from Section 4.4 show that the forest is actually highly correlated to the trait marker, but, as mentioned, is more strongly correlated to some of the SNPs.
5.2 Future Work

The first extension to the presented work could be to try and look for associations between multiple trait marker. The idea would be to find association between not just one gene and a disease but two or more genes that have an effect on the risk of developing a disease. The implementation described in Section 3 already supports this feature in almost all parts. The only part that really needs a reimplementation is the analysis tool, since to analyze for multiple SNP requires a complete new strategy compared to the one used.

A second extension that the designed and implemented framework supports is an extension that works on Panel Data. The idea is to work on data that is split into two parts namely a panel part and a case/control part. The case/control is a normal data set like those used previously, but it is only containing a subset of the SNPs that the panel data contain information on. The panel data is only containing a few individuals without a known phenotype. The idea is then to generate a forest on the case/control data and use it to explore the panel data.
References


A Supplementary Experimental Figures

Figure A.1:

Figure A.2:
Figure A.3:

Figure A.4: