Preventing overfitting in Mixed Integer Optimization based classification tree construction

Maaike Elgersma
(4585011)

A thesis presented for obtaining the degree of
Bachelor of Science in Applied Mathematics

Optimization
Delft Institute of Applied Mathematics
Delft University of Technology
The Netherlands
3 July 2019

Supervisor:
Dr. ir. L.J.J. van Iersel

Assessment Committee:
Dr. ir. L.J.J. van Iersel
Dr. B. van den Dries
Dr.ir. W.T. van Horssen

Abstract

The global use of azole antifungals as treatment against infections caused by *Candida* has led to an increase in azole resistance. The primal goal of this BSc thesis is to improve existing Mixed Integer Optimization models to classify azole resistance of *C. glabrata* more accurately by preventing overfitting. Moreover, these classification methods can generally also be used for the classification of any kind of numerical of categorical data. The classification method that we used is based on an MIO formulation that was first introduced by Bertsimas and Dunn, and later adapted by Van Dijk. We first made the output of the model much easier to interpret, both from a mathematical and biological point of view. We also applied feature sampling to reduce the run time of the program, making it possible to create deeper trees, and to prevent overfitting. To further prevent overfitting in these deeper trees, we added the option of forcing at least a certain number of training data points in the leaves to the MIO formulation. We verified our MIO model on a data set constructed by combining two data sets from the Westerdijk Fungal Biodiversity institute and a data set from the Center for Disease Control and Prevention Atlanta, all containing the FKS1 and FKS2 gene sequences from *C. glabrata*. We automated the preprocessing steps and merging process of these data sets with a Python program, and wrote a user manual on how to use this program. By processing a bigger data set we were able to classify more data correctly than Van Dijk, and we outperformed the CART algorithm. Similar accuracy results were obtained when applying feature sampling as when not, and the run time was drastically reduced. Deeper trees did not change out-of-sample accuracy much, though this may be because our data sets did not require deeper trees. When also forcing at least a certain number of training data points in each leaf of these deeper trees, we were able to slightly increase the out-of-sample accuracy, which means overfitting was indeed prevented slightly. Lastly we interpreted the results in biological context, and found some resistance-related mutations that were already identified previously in other research, as well as some additional ones for which the biological relevance is yet unknown.

Acknowledgements

I would like to thank the following people:

- Leo van Iersel, my supervisor at TU Delft,
- Mick van Dijk, from TU Delft,
- Mark Jones, from TU Delft,
- Teun Boekhout, from Westerdijk Fungal Biodiversity Institute and UvA,
- Leen Stougie, from CWI and VU
- Amir Arastehfar, from Westerdijk Fungal Biodiversity Institute,
- Jorge Navarro, from Westerdijk Fungal Biodiversity Institute,
- Aimi Stavrou, from Westerdijk Fungal Biodiversity Institute,
- Shawn Lockhart, from Center for Disease Control and Prevention Atlanta.

Contents

1 Introduction

Since 1940 there has been an enormous increase in the occurrence of fungal infections by different *Candida* species. Today, invasive candidiasis is the third-to-fourth most frequent infection received in hospitals worldwide. One of the reasons for this increase is the inclination of Candida species to infect medical devices [4]. An increased use of immunosuppressive therapy as well as broadspectrum antibiotic and antifungal therapies have therefore resulted in an increasing number of cases of *C. glabrata* infections [8]. Another reason is the increase in the number of individuals sensitive to invasive fungal infections in recent years [9]. Compared with all other *Candida* species, *C. glabrata* infections are the most threatening since the mortality rate associated with this species is the highest [8].

Global use of azole antifungals such as *fluconazole* as treatment against infections caused by *Candida* has lead to an increase in azole resistance. Several mechanisms have been reported as potential reasons for azole resistance in Candida species [4]. We elaborate more on this topic in Section 2.1.

Currently, when a patient is diagnosed with a *Candida* infection, a sample of that fungus is taken. The fungus is grown in a lab, and then it is determined whether the isolate is resistant to certain antifungals or not. However, this takes about a week due to the incubation period. The goal of our research is therefore to determine whether a *C. glabrata* isolate is resistant or not based on a section of its DNA by using classification trees. It is much quicker to sequence part of the DNA and put it into an algorithm, than to determine whether it is resistant or not via the conventional way. In this research we focused on *C. glabrata*, a fungus which is commonly associated with infections. However, the classification methods applied on this *Candida* species can generally also be used for the classification of any kind of numerical or categorical data.

1.1 Related work and our contribution

Bertsimas and Dunn have already introduced a basis Mixed Integer Optimization formulation for constructing optimal classification trees [3]. Van Dijk used and adapted this formulation in his MSc thesis [2].

The goal of our research was to tackle certain problems that they ran into. One of those problems was overfitting of the trees on the training data. To tackle this problem we extended the MIO formulation introduced by

Bertsimas, Dunn and Van Dijk, which we did in Sections 5.2 and 5.3. These authors had issues regarding the run time of the program, which made it difficult to produce trees of a depth higher than two in a reasonable time. We tackled this problem with a method called 'feature sampling', which is also used in [5] (called "threshold sampling" there) for a different MIO formulation with column generation. To prevent overfitting we added a constraint to the MIO formulation that forces at least a certain number of training data points in each leaf.

Furthermore, the Westerdijk Fungal Biodiversity Institute had two data sets on *C. glabrata* that they wanted to be analyzed. We used their data sets to validate our model, along with a data set from the CDC in Atlanta, and put our results in a biological context that would be useful to them. Van Dijk already processed one of these data sets, but not the other two. Therefore we improved his results by processing a larger data set. To perform this efficiently we wrote a Python program to preprocess the data automatically, which is a further improvement to the work described by Van Dijk.

We then verified our model on these data sets in Python, using Gurobi as a solver. In most cases we outperformed the CART algorithm. Features sampling allowed to creater deeper trees, which, in combination with forcing at least a certain number of training data points in each leaf, indeed resulted a slight reduction of overfitting. Using protein sequences instead of DNA sequences gave even higher accuracies, especially when merging the data sets of both genes into one data set. We were also able to find some results that could be interesting for further biological research.

2 *Candida glabrata*

In this chapter we will first explain something about the drug resistance of *Candida glabrata*. Furthermore, we will describe and explain the *C. glabrata* data sets that we received from the Westerdijk Fungal Biodiversity Institute and the Center for Disease Control and Prevention Atlanta.

2.1 Drug resistance of *Candida glabrata*

We will start by explaining the basis of drug resistance in *Candida glabrata*. In our research we focused on genetic alteration related resistance, rather than physiological (see [2] for more biological details).

The use of *fluconazole* as primary antifungal therapy against *Candida* infections worldwide, has led to an increase in azole resistance. Therefore, *Echinocandins*, a class of antifungal drugs such as *caspofungin*, *micafungin* and *anidulafungin*, are now used as a primary therapy for the treatment of invasive candidiasis. Recent studies, however, show that the wide use of Echinocandins affects the susceptibility of *Candida* species, especially *C. glabrata*. Between 2001 and 2010, resistance to Echinocandins increased from 4.9% to 12.3% according to a study on data from patients with *C. glabrata* bloodstream infections [8]. Recent studies show that mutations in two hot-spot regions of the FKS1 and FKS2 genes might be responsible for reduced susceptibility of *C. glabrata* to Echinocandins [7].

2.2 Our *Candida glabrata* data set

The Westerdijk Fungal Biodiversity Institute has provided us with several data sets with the DNA sequences of the FKS1 and FKS2 genes of *C. glabrata* isolates. We received several '`.fasta`' files containing either the DNA sequence of the entire FKS1 and FKS2 genes or merely the hotspots, of both susceptible and resistant isolates. Along with this fasta files we also received Excel files containing certain Minimium Inhibitory Concentration values. Such MIC value is the lowest concentration of an antifungal drug which prevents visible growth of the fungus. We received the MIC values of three Echinocandins antifungal drugs, being *caspofungin*, *micafungin* and *anidulafungin*. We also received excel files from Shawn Lockhart at the Center for Disease Control and Prevention Atlanta, containing the DNA sequences of the FKS1 and FKS2 genes of *C. glabrata*. In Section 3.1 we will go into more detail of what the data set looks like.

3 In depth explanation of our data and the preprocessing steps

There are several preprocessing steps that we applied on the three different data sets that we worked with. In this chapter we will explain in more detail what the data sets that we worked with looked like, and we will describe the preprocessing steps we took before being able to input the data into our model. In Subsection 3.2 a an explanation on how to automatically process data sets using our Python files is given, as well as some details on the way it works.

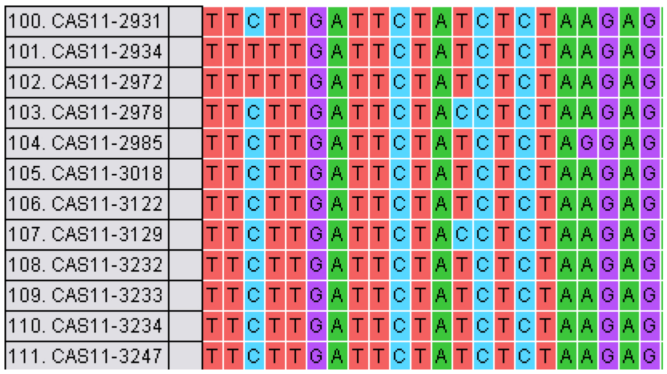
3.1 A general explanation of the data sets and the preprocessing steps

All data sets contained numerical data. Meaning they consisted of nucleotides with the letters A, T, C and G. However, when analyzing such data, binary data is required, so we converted all data sets to binary arrays. We wanted to do this in such a way that a zero indicates that a nucleotide is not mutated, and a one indicates that it is mutated. Each row of the array then represents the DNA sequence of an isolate, and each column of the array then represents a nucleotide position in the DNA sequence.

3.1.1 Our fasta data sets and the preprocessing steps

The first data set that we processed were '`.fasta`' files containing around twenty DNA sequences of the hotspots in FKS1 and FKS2. However, the sequences did not all have the same length. Therefore, we first aligned these sequences in MEGA, Molecular Evolutionary Genetics Analysis, a "sophisticated and user-friendly software suite for analyzing DNA and protein sequence data from species and populations" [1]. Then we isolated the hotspots, which consist of around 30 nucleotides. We subsequently changed this numerical data to binary data. Therefore we first wrote a program in Python to find the most common nucleotide at each position. This could be considered as the reference sequence, meaning a sequence of DNA with every nucleotide being the most common nucleotide of all isolates on that position. After taking these steps we were able to convert our numerical data to binary data, as described before. Lastly we wanted to remove redundant columns where none of the isolates showed a mutation, because positions where each isolate contains the same nucleotide are irrelevant for the model, and they increase the running time of the program considerably. The data set of the first hotspot of the FKS1 gene only consisted of 2 columns after processing it. The data set of the first hotspot of the FKS2 gene contained no mutations and was therefore empty and not useful for further research.

We also had another much larger data set of two fasta files, both containing the full-length DNA sequences of the FKS1 gene and the FKS2 gene, of length 5592 and 5694 respectively. These data sets came from 240 isolates, so there were 240 rows, of which 220 were drug susceptible and 20 were drug resistant. In figure 1 a snapshot of this data set is shown. The first nucleotide position in the data set is also the first nucleotide position in the FKS1 or FKS2 gene. Hence these sequences were already aligned. However, in one sequence of the FKS2 gene a deletion of three nucleotides had taken place, meaning three nucleotides were 'cut out' of the DNA sequence. Therefore, the part of the sequence after the deletion was shifted, so we first had to enter three empty nucleotides in the sequence. Then the only thing left to do was to convert it to binary data, and remove the redundant columns where no isolate had a mutation. This resulted in only 138 and 187 columns for the FKS1 and FKS2 gene respectively, a significant downsizing of the data set.



```
100. CAS11-2931   TTCTTGATTCTATCTCTAAGAG
101. CAS11-2934   TTTTTGATTCTATCTCTAAGAG
102. CAS11-2972   TTTTTGATTCTATCTCTAAGAG
103. CAS11-2978   TTCTTGATTCTACCTCTAAGAG
104. CAS11-2985   TTCTTGATTCTATCTCTAGGAG
105. CAS11-3018   TTCTTGATTCTATCTCTAAGAG
106. CAS11-3122   TTCTTGATTCTATCTCTAAGAG
107. CAS11-3129   TTCTTGATTCTACCTCTAAGAG
108. CAS11-3232   TTCTTGATTCTATCTCTAAGAG
109. CAS11-3233   TTCTTGATTCTATCTCTAAGAG
110. CAS11-3234   TTCTTGATTCTATCTCTAAGAG
111. CAS11-3247   TTCTTGATTCTATCTCTAAGAG
```

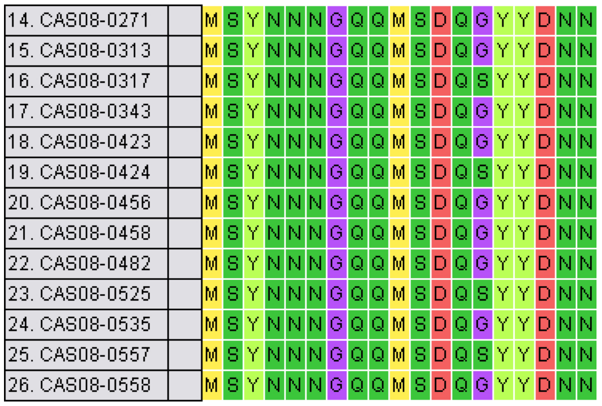
Figure 1: A snapshot of a small part of the fasta data with the DNA sequences.

For all fasta files we also had Excel documents containing several MIC values. Therefore we also wrote a program to determine from these MIC values of each organism whether it was resistant or not.

3.1.2 Exporting fasta files as protein sequences

Rather than saving the data as DNA sequences in a '`.fasta`' file, one could also decide to save them as protein sequences. The further processing of the data with the Python program remains exactly the same. Saving them as protein sequences means that every triplet of DNA nucleotides is translated to an amino acid, indicated in MEGA by 20 different letters. There are several advantages to this. First of all, the data set is hugely reduced in size, for there are now three times fewer columns, since every three nucleotides are translated to a single amino acid. Also, the number of columns is reduced even further, since silent mutations are filtered out when removing redundant columns. Silent mutations are mutations of a nucleotide that do not result into a different amino acid. Normally these mutations would lead to the columns being present in the remaining data set after removing redundant columns. However, if the mutation is silent, this does not lead to a different amino acid, and hence this change is irrelevant. Therefore, the column of that amino acid is removed when removing redundant columns (provided there are no mutations in the other two nucleotides of that same triplet).

Since our data sets that we mentioned before consisted of fasta files, we were also able to translate the DNA sequences into amino acids. By doing so we threrefore reduced the amount of columns even further, to only one third of the original data set. In figure 2 a snapshot of this data set of amino acids is shown.

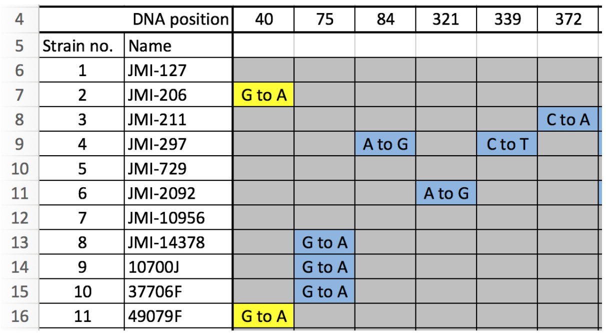


14. CAS08-0271		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	G	Y	Y	D	N	N
15. CAS08-0313		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	G	Y	Y	D	N	N
16. CAS08-0317		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	S	Y	Y	D	N	N
17. CAS08-0343		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	G	Y	Y	D	N	N
18. CAS08-0423		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	G	Y	Y	D	N	N
19. CAS08-0424		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	S	Y	Y	D	N	N
20. CAS08-0456		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	G	Y	Y	D	N	N
21. CAS08-0458		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	G	Y	Y	D	N	N
22. CAS08-0482		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	G	Y	Y	D	N	N
23. CAS08-0525		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	S	Y	Y	D	N	N
24. CAS08-0535		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	G	Y	Y	D	N	N
25. CAS08-0557		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	S	Y	Y	D	N	N
26. CAS08-0558		M	S	Y	N	N	N	G	Q	Q	M	S	D	Q	G	Y	Y	D	N	N

Figure 2: A snapshot of a small part of the fasta data with translated protein sequences.

3.1.3 Our Excel data sets and the preprocessing steps

Lastly we had a data set in Excel, containing the DNA sequences of the FKS1 and FKS2 gene of 55 drug susceptible organisms, and 28 resistant organisms. The FKS1 gene and FKS2 gene data sets consisted of 76 and 121 columns respectively. In figure 3 a snapshot of this data set is shown. These data sets only contained the nucleotide positions where at least one of the sequences was mutated. Therefore there was no need to remove redundant columns. If there was no mutation, the cell was empty, and if there was a mutation it was indicated which mutation had taken place. For example: ‘A to C’ meant that the reference sequences had an ‘A’ at that nucleotide position, but for this sequence the nucleotide was mutated to a ‘C’. This made it rather easy to convert this data to binary data. If a cell was empty, it became a 0, and if it was not empty it contained a mutation, and therefore it became a 1.



4		DNA position	40	75	84	321	339	372
5	Strain no.	Name						
6	1	JMI-127						
7	2	JMI-206	G to A					
8	3	JMI-211						C to A
9	4	JMI-297			A to G		C to T	
10	5	JMI-729						
11	6	JMI-2092				A to G		
12	7	JMI-10956						
13	8	JMI-14378		G to A				
14	9	10700J		G to A				
15	10	37706F		G to A				
16	11	49079F	G to A					

Figure 3: A snapshot of a small part of the fasta data with DNA sequences.

3.1.4 Combining the large fasta and Excel data sets

It was mentioned before in this section that the positions of the nucleotides in both the large fasta data sets and the Excel data sets in the DNA of the FKS1 and FKS2 gene were indicated. Therefore, after importing the sequences in Python, we were able to align both data sets and create two large data sets, one for each gene. We decided to use this data set to validate our model in Section 7. The total data set then consisted of the DNA sequences of 323 isolates, with 160 and 230 columns for the FKS1 and FKS2 gene respectively. We mainly used these combined data sets for validating our model in Chapter 7.

We were also able to combine the data sets of both genes into one large data set. Each row then consisted of the FKS1 gene first, and the FKS2 gene after it, so the number of rows remained the same as those of the separate data sets. We mainly used this data set for assessing the results in a biological context.

3.2 Explanation of the preprocessing steps in the Python program

In this section we have provided a user manual to show how the preprocessing steps explained in Section 3.1 can be done automatically with the help of the Python files `final_OptimalClassificationTrees.py` and `EditData.py`, from which the code can be found in the Appendices A.1 and A.2 respectively. Occasionally a small explanation on the way certain functions work is also given.

3.2.1 DNA data in fasta files

Before using the python programs to process the data, make sure that the sequences are aligned in MEGA. We have done this with the alignment tool 'ClustalW'. Then export the alignment as a fasta file and save it in the same folder as where your two python files are located. The files can be exported as either 'DNA sequences', consisting of nucleotides, or as 'translated protein sequences', as explained before in 3.1.2. This makes no difference for the further use of the Python program to process the data. At the top of the `final_OptimalClassificationTrees.py` program, in between the import statements and the first function `Create_train_and_test` enter the file name:

`filename =` *filename without* '`.fasta`'.

Then add the amount of sequences in the data:

`total_data =` *number of sequences.*

Then add the statement:

```
X_unordered, columns_withmutations =
EditData.createbinarydata(filename_data).
```

This refers to the python file `EditData.py`. This is the file where all data processing functions are stored. The function `createbinarydata` creates a binary array of the data in `filename.fasta` file. The function first refers to the function `readdata_fasta`. This function imports the data from the fasta file and makes a list (called '`X`') of the data from that file.

We will now explain what the data structure of the output of the function `readdata_fasta` is. As mentioned before, we wanted the data set to be converted to a binary array of the data, where the rows are the DNA sequences of the isolates, and the columns are the nucleotide positions of the sequences. However, we decided to work with nested lists rather than arrays. So the list `X` is a list consisting of nested list. Every sequence in the fasta file is stored as a nested list in `X`. Therefore the number of nested lists of X is equal to the number of sequences, and each nested list has the same length as the sequences. So for example `X[0][3]` is the fourth nucleotide letter in the first sequence of the data.

Then, to convert this numerical data to binary data, we need a 'reference sequence' that we can compare all our sequences to. This is done with the function `createwildtype`, where with 'wild type' we actually mean the reference sequence. The function works in the following way. It creates a single sequence, being the reference sequence, by referring to the function `findwildtype` for each nucleotide position. This function finds the most common nucleotide (the letter A, C, T or G) in all sequences at that nucleotide position. Once this reference sequence is determined, all sequences from `filename.fasta` can be converted to binary sequences.

This is done as explained before, namely if a letter at a certain position is equal to the nucleotide at that position in the reference sequence, it becomes a 0. However, if it is a different nucleotide, meaning it has a mutation at that position, it becomes a 1. We have then obtained a very large binary data set. Lastly we want to reduce the number of columns by removing redundant columns. These are the positions where there is no mutation at that position in any of the isolates. This is also done in the function `createbinarydata` by referring to the function `removeredundantcolumns`. This function also returns the position numbers of the columns that were not removed from the data, so these are the DNA positions where a mutation identified.

We have now created a binary data set, with only the relevant columns left. We then still need the data to be ordered in the following way: all susceptible isolates first, then all resistant isolates. This is necessary for the function that creates the Optimal Classification Tree. This is done by the `createordereddata` function, which can be called adding the statement:

```
X, Y, data_S, data_R =
EditData.createordereddata(X_unordered,resistant_sequences,
total_data)
```
Here `resistant_sequences` is a list containing the numbers of the isolates in X_unordered that are resistant. For the '`.fasta`' files that we worked with, the MIC values that indicated whether the isolates were resistant or not were in an excel file. How to process these files is explained in the next section.

3.2.2 MIC values in an excel document

We received excel files containing the MIC values of all isolates to three antifungal drugs, *caspofungin*, *micafungin* and *anidulafungin.* For anidulafungin for example a value lower or equal to 0.12 mg/L meant that the isolate was susceptible, and a value higher or equal to 0.5 mg/L indicated that the isolate was resistant. We therefore first converted all these values to a binary value: 0 if the value was smaller or equal to 0.12, 1 if the value was larger or equal to 0.5, and 0.5 if it was between 0.12 and 0.5. We did this for all values of all sequences and all three medicines in excel. For *anidulafungin* the formula would look like this:

`=IF(`*value cellposition* `<= 0,12; "0";`
`IF(`*value cellposition* `>= 0,5;"1";"0.5")),`

where '*value cellposition*' was 'C10' for example.

After these calculations in excel, the data can be read and processed automatically in Python again. Add the statements:

`filename_MICvalues =` *file name of the MIC values without '.excel'*
`sheetname =` *sheet name of the MIC_values*
`filedirectory =` *directory of your file with the MIC values*

Then we can use the function `determineclass` to find the resistant isolates in the data. To call the function add:
```
Y_unordered, resistant_sequences =
EditData.determineclass(filename_MICvalues, filedirectory,
sheetname, total_data)
```
This function reads in the file. It then calculates the sum of the binary MIC values of each sequence. We decided that an isolate was susceptible if the sum of the three binary MIC values was 1 or lower, and it was resistant if the sum of the three binary MIC values was 1.5 or higher.

3.2.3 DNA data in Excel files

We received two Excel files containing DNA data. One with the information of susceptible sequences of the FKS1 gene and the FKS2 gene, and one with the information of resistant sequences of both genes. The information of the different genes were on different sheets. These files where both in the format as explained before in Section 3.2.3: they only contained the nucleotide positions where at least one of the sequences was mutated. Therefore there was no need to remove redundant columns. The only thing we had to do is convert them to binary data, which we did in a seperate sheet, for example 'FKS1', with the following formula:

`=IF('FKS1 mutations per strain'!` *nucleotide cellposition* `="";0;1)`, where 'FKS1 mutations per strain' is the sheet containing the DNA data, and 'nucleotide cellposition' is 'C10' for example. The rest of the processing can be done using the Python files again. We eventually wanted to merge the data of the susceptible and resistant sequences of the FKS1 gene, and do the same for the FKS2 gene. We first need to import the data, which can be executed by the following lines:

`filename =` *file name without '.xlsx'*

`filedirectory =` *the directory of the file*

`total_data =` *number of sequences of the data*

`gene =` *either 'FKS1' or 'FKS2'*

```
X_unordered = EditData.readdata_excel(filename,filedirectory,
total_data,gene)
```

Now we want to merge both files. This can be done with the function `mergefiles`. To execute this, you first have to mention what the resistant sequences are:

`resistant_sequences =` *list of the numbers of the sequences that are resistant*

Everything that is mentioned above should be done for both documents. Then the files can be merged by the following statement:

```
X_unordered, columns_withmutations, resistant_sequences, total_data
= EditData.mergefiles(X_1[1:],X_1[0],resistant_sequences1,total_data1,
X_2[1:],X_2[0],resistant_sequences2,total_data2)
```

Here 1 and 2 indicate the properties of both data sets that you want to merge. The function `mergefiles` merges both of them, taking the positions of the nucleotides in the DNA into consideration. Therefore this function not only returns the `X_unordered` but also a list of the DNA locations of the columns of the data.

The data is actually already sorted now, since the function `mergefiles_excel` adds the sequences from the first data set, being the susceptible data set, first, and then sequences from the second data set, being the resistant data set. For unicity and convenience however we keep the statement that we also used for '`.fasta`' files:

```
X, Y, data_S, data_R =
EditData.createordereddata(X_unordered,resistant_sequences,
total_data)
```

3.2.4 Merging fasta and Excel data sets

The function `mergefiles` that we mentioned in Section 3.2.3 can actually also be used to combine fasta data sets with data sets in Excel, which is what we also decided to do, to have one large data set for both the FKS1 and the FKS2 gene. This can be done in a very similar way as previously described for combining data sets in Excel, after having defined the `X_unordered`, `columns_withmutations`, `resistant_sequences` and `total_data` for both data sets, with the following statement:

```
X_unordered_combined, columns_withmutations,
resistant_sequences,total_data = EditData.mergefiles(
 X_unordered1,columns_withmutations1[:-1],resistant_sequences1,
 total_data1,X_unordered2,columns_withmutations2[:-1],
 resistant_sequences2,total_data2)
```

4 Classification trees

In this chapter we will explain the method that is used to classify data, being classification trees. We will start by introducing the general concept of classification trees, and then how to construct them using the CART algorithm, which is a popular heuristic for constructing such trees.

Classification trees are a form of decision trees, which are supervised machine learning methods, meaning they work with labeled data rather than unlabeled data. They are widely used in statistics and data mining as a method to classify numerical and categorical data, or they are used as a tool to extract information from data. The main reason for their popularity for such purposes is their capability to handle high dimensional data where, but with only a limited number of samples. This situation is very common in both bio-informatics and statistical genetics, and it is also the case for our data. Another reason for their popularity is that classification trees are, in contrast to other classification methods such as neural networks, easy to interpret as they closely resemble human reasoning [2]. It was for these two advantages of classification that we decided to use this classification method in our research.

We used classification trees for both purposes mentioned before: we wanted to use the labeled data that we had to construct classification trees and extract information from those trees, and we wanted to be able to classify new unlabeled data in the future. The data that we received was categorical data, which is different from numerical data. In numerical data each data point has a numerical value that can be binary, integer or real. In categorical data however, the data does not have such a numerical value. In our case for example, the data is DNA sequences of organisms that are either susceptible or resistant, which is a class rather than a numerical value. These classes can of course be converted to numerical data, which is usually done in such cases, without loss of generality.

4.1 The general concept

The objective when creating a classification tree is to construct a tree model based on numerical or categorical training data that can then be used to classify new data [2].

An example of such a tree is shown in figure 4. This classification tree classifies humans as male or female, based on their height and body weight.

By assessing two characteristics of a human being, whether he/she is taller than 180cm or not and whether he/she is heavier than 80kg or not, a persons sex is predicted.

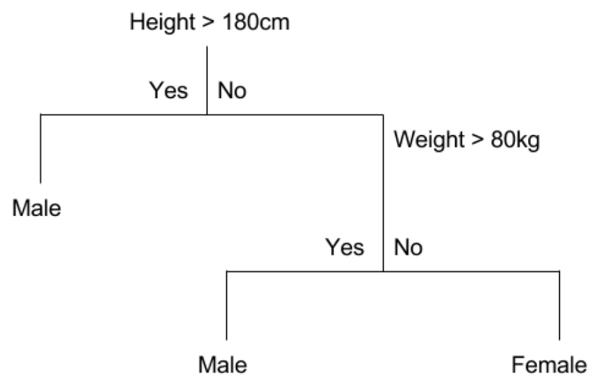


Figure 4: Simple classification tree of depth two. Adapted from Classification And Regression Trees for Machine Learning by Browniee, J. (2016).

Algorithms to create such trees partition the training data recursively at each step, by choosing a feature, or a combination of multiple features, to split on. This process is repeated until it has constructed a model that classifies the data perfectly, or when a specific maximum tree depth is reached. The final tree will consist of a root node and a number of branches that represent the order of splits applied for the classification. When there is new data available, the algorithm can classify this by following the correct path through the obtained classification tree [2].

The first step in the construction of a classification tree is to split the data set into training data and test data. It is imported to realize that all this data has to be labeled. Training data is used by the algorithm to construct the tree. The in-sample misclassification of a tree is determined by assessing the labels of the training data. For the evaluation of the performance, labeled test data is used with which the out-of-sample misclassification is calculated. The quality of a model is determined by its ability to fit the training data and the test data, and the complexity of the trees that it constructs, of which the out-of-sample misclassification is most important [2].

The misclassification of a tree is calculated by comparing the label of each data point predicted by the tree, compared to the actual label of the data point. These predicted labels are determined in the following way. The predicted label of a leaf of a tree is the most common label of training data points in that leaf. Every training data point and each test data point is classified into a leaf by the tree. Therefore the predicted label of a data point, being the label of the leaf that it is classified into, can be compared to the actual label of the data point. The total misclassification is therefore defined as the ratio between the number of wrongly labelled data points and the total number of data points[2].

4.2 Constructing Classification trees

The goal when constructing classification trees, is finding the tree with the lowest in-sample misclassification error, i.e. the tree that fits the training data the best. There are several methods to construct such classification trees. A common method is tree constructing algorithms, for example the CART algorithm. The method we used was Mixed Integer Optimization, which we compared to the CART algorithm. The way the CART algorithm works is explained is Section 4.3. We will now discuss several advantages and disadvantages of both methods.

To minimize the total misclassification error, most tree constructing algorithms recursively solve local optimization problems, to determine the best feature to select on at each split. This top down approach is greedy in nature, which can affect the global optimality of the solution [2]. However, their time complexity is very low, so a massive advantage is that they can easily process large data sets.

To tackle the goal of achieving global optimality of a tree, Mixed Integer Optimization can be used, for integer programming problems find optimal solutions. Therefore classification trees were formulated as an MIO problem by D. Bertsimas and J. Dunn in [3]. A problem that they and M. van Dijk in [2] ran into however, was overfitting. For classification trees overfitting means that a tree is too fitted on the training data that was used to construct it. Consequently, mutations found in the training data may be split on in the tree, when in reality these mutations do not affect susceptibility or resistance. Therefore the training data may be classified extremely well, whereas the test data will be classified incorrectly, and the tree may still not perform as well as desired. Part of the cause of overfitting is a data set that is too small, i.e. when the number of samples is small.

We will explain this concept of overfitting some more using the example tree shown in figure 4. For example, in the Netherlands the general population is relatively tall, so if you used a data set of many people from the Netherlands to create such a classification tree for the world population, based on the characteristics height and weight, it might look different. The first split on the tree will probably be based on a higher limit, for example 'Height $>$ 185cm'. For people from the Netherlands, this first question is then appropriate, but this is probably not the case for rest of the world population. Therefore, more males would be classified wrong than when using a data set of people from many different countries to create the tree. In conclusion, if your data set is small, it is less likely to be representative of the general population of the data that you are researching, and will lead to the problem of overfitting.

For the biological sequence data that we worked with, the problem of overfitting is caused by the fact that there are lots of random mutations that are not related to resistance. If you have enough columns, it will be possible to perfectly classify your data using these random mutations. However, such trees will perform very poorly on test data. To tackle the problem of overfitting we added some more variables and constraints in Section 5.2 to the basis MIO formulation of Section 5.1, and we used a larger data set to construct the trees.

4.3 CART algorithm

As mentioned before we compared our MIO-based approach to the CART algorithm. In this section we will explained the way this decision tree constructing algorithm works.

The CART algorithm is a greedy algorithm, meaning it solves local optimization problems at each stage. In the context of classification trees this means that the algorithm starts at the root node of a tree with a predefined depth. To decide which feature would be best to split on, it calculates the Gini impurity of the child nodes:

$$G = \sum_{k \in C} p_k(1 - p_k),$$

where C is the set of classes, and p_k is the fraction of samples in that node that belong to class k. This impurity indicates the purity of leaf. A node is 'pure' if all of the data points in that node belong to the same class, otherwise the node is 'impure'. If a node is pure, then the Gini impurity is equal to zero. The CART algorithm then makes its decision by minimizing the sum of Gini impurity measures over all child nodes of that node. After doing so for the root node, it then executes the same process for each child node of the root node, and so on.

For an example on the way the CART algorithm works, we recommend the MSc thesis by Van Dijk, [2] page 15.

5 Univariate optimal classification trees

In this chapter we will introduce the Mixed Integer Optimization formulation of the classification trees.

5.1 Basis mixed integer optimization formulation

In this first section we will start by introducing the basis Mixed Integer Optimization formulation that we took from the article of Bertsimas & Dunn on optimal classification trees [3], which Van Dijk also used an adapted in his MSc thesis [2].

5.1.1 The data and tree structure

We will start by explaining the notation of the data from the *Candida glabrata* data sets that we used. As explained before in Section 3.1, the data that we work with is binary. It is in the general form (x_i, y_i) for $i = 1, \dots, n$. Here $x_i \in \{0,1\}^p$ are the sequences of DNA of length p, meaning they can be seen as vectors where the indices $j = 1, \dots, p$ are the features. These features are 0 if there is no mutation on that position in the DNA, and 1 if there is. The y_i are the binary values that indicate whether the organisms that the DNA is from were resistant or susceptible. These are therefore the two classes k that the data points can belong to.
We will now look into the structure and formulation of the classification trees that we will be creating. A tree of depth D contains $T = 2^{D+1} - 1$ nodes, of which $t \in \mathcal{T}_B = \{1, \dots, \lfloor T/2 \rfloor\}$ are the branch nodes, and $t \in \mathcal{T}_L = \{\lceil T/2 \rceil, \dots, T\}$ are the leaf nodes. The branch nodes can apply splits, which we denote by $a_t^T x_i < 1$. This is the split applied by node t. It is the inner product of the split vector a_t and the data point x_i. The split vector $a_t \in \{0,1\}^p$ contains at most one entry that is 1, the others are all 0, since the classification tree that we are creating is univariate. The index of the entry that is 1, indicates therefore what feature is being split on. If the inner product $a_t^T x_i$ is equal to 0, the data point will descend down the left branch, and if it is equal to 1, it will descend down the right branch. A data point always ends up in a leaf in our MIO formulation. We then denote all the ancestors of the leaf t where such a data point took the left branch by $A_L(t)$ and all the ancestors where it took the right branch by $A_R(t)$. Hence, $A_L(t) \cup A_R(t)$ denotes all the ancestors of leaf t. Furthermore, the parent

node of a node t is denoted by $p(t)$.

5.1.2 The variables and parameters

Now that we have all the basic notation we can start introducing variables used in the MIO formulation from [2].
The split vector $a_t,\ t \in \mathcal{T}_B$ that we talked about before is the main decision variable in our MIO problem. The variable decides which feature every branch node splits on. We start off by requiring each branch node to apply a split on exactly one feature, by adding the following constraint to our problem:

$$\sum_{j=1}^{p} a_{jt} = 1, \quad \forall t \in \mathcal{T}_B \tag{1}$$

There are several indicator variables that we will introduce for an easier formulation of all the other constraints. Eventually we want to label every leaf with the class of the most common class of training data points in that leaf, and minimize the amount of training data points that are in a leaf of a different class. Or, in other words, we want to minimize the total misclassification error.

Therefore we will first introduce the indicator variable $z_{it} = \mathbb{1}\{x_i \text{ is in node } t \in \mathcal{T}_L\}$. To make sure this variable receives the correct values, we set the following constraints:

$$a_m^T x_i - (1 - z_{it}) \leq 0, \quad \forall t \in \mathcal{T}_L, \quad \forall m \in A_L(t) \tag{2}$$

$$a_m^T x_i + (1 - z_{it}) \geq 1, \quad \forall t \in \mathcal{T}_L, \quad \forall m \in A_R(t) \tag{3}$$

Now, if a data point i is in a leaf t, meaning $z_{it} = 1$, the it should suffice the splits $a_m^T \leq 0$ for all it's left ancestor branch nodes, and $a_m^T \leq 0$ for all it's right ancestor branch nodes.

Then, to enforce that each data point is assigned to exactly one leaf, we add the constraint:

$$\sum_{t \in \mathcal{T}_L} z_{it} = 1, \quad i = 1, \ldots, n \tag{4}$$

Now we can determine the total number of points in leaf node t:

$$N_t = \sum_{i=1}^{n} z_{it}, \quad \forall t \in \mathcal{T}_L \tag{5}$$

We then want to determine the predicted class k of leaf t, which is the most common class of the data points in that leaf. We therefore introduce the parameter matrix $Y_{ik} = \mathbb{1}\{y_i = k\}$, indicating the class of each data point. With this parameter we can then define the number N_{kt} of data points of class k in leaf t with the following constraint:

$$N_{kt} = \sum_{i=1}^{n} Y_{ik} z_{it}, \quad k = 0, 1, \quad \forall t \in \mathcal{T}_L \tag{6}$$

Now the predicted class k of leaf t becomes $c_{kt} = \mathbb{1}\{c_t = k\}$, where $c_t = \text{argmax}_{k=0,1}\{N_{kt}\}$. We want to ensure that each leaf gets assigned exactly one class. Therefore we add the constraint to limit c_{kt}:

$$\sum_{k \in \{0,1\}} c_{kt} = 1, \quad \forall t \in \mathcal{T}_L \tag{7}$$

The objective is to minimize the misclassification error, so:

$$\min \sum_{t \in \mathcal{T}_L} L_t \tag{8}$$

where

$$L_t = N_t - \max_{k=0,1}\{N_{kt}\} = \min_{k=0,1}\{N_t - N_{kt}\}$$

In [2] and [3] L_t was linearized by the following constraints:

$$L_t \geq N_t - N_{kt} - n(1 - c_{kt}), \quad k = 0, 1, \quad \forall t \in \mathcal{T}_L \tag{9}$$
$$L_t \leq N_t - N_{kt} + n c_{kt}, \quad k = 0, 1, \quad \forall t \in \mathcal{T}_L \tag{10}$$
$$L_t \geq 0, \quad \forall t \in \mathcal{T}_L \tag{11}$$

The constraint (9) is indeed necessary, because if the predicted class of a leaf t is indeed k, so $c_{kt} = 1$, then L_t must be equal to the total number of data points in leaf t, N_t, minus the number of data in leaf t belonging to class k, since we defined this to be the in-sample misclassification. However, if the predicted class of a leaf t is not equal to k, so $c_{kt} = 0$, then this should not define the lower bound of L_t. We have found, however, that the constraint (10) was not necessary, since the objective is to minimize L_t. Therefore, we do not need an upper bound for L_t.

Combining all of this together yields the following basic MIO formulation for constructing optimal classification trees:

$$
\begin{aligned}
\min \quad & \sum_{t \in \mathcal{T}_L} L_t \\
\text{s.t.} \quad & L_t \geq N_t - N_{kt} - n(1 - c_{kt}), \quad k = 0, 1, \quad \forall t \in \mathcal{T}_L \\
& L_t \geq 0, \quad \forall t \in \mathcal{T}_L \\
& N_{kt} = \sum_{i=1}^{n} Y_{ik} z_{it}, \quad \forall k = 0, 1, \quad \forall t \in \mathcal{T}_L \\
& N_t = \sum_{i=1}^{n} z_{it}, \quad \forall t \in \mathcal{T}_L \\
& \sum_{k \in \{0,1\}} c_{kt} = 1, \quad \forall t \in \mathcal{T}_L \\
& a_m^T x_i - (1 - z_{it}) \leq 0, \quad \forall t \in \mathcal{T}_L, \quad \forall m \in A_L(t) \\
& a_m^T x_i + (1 - z_{it}) \geq 1, \quad \forall t \in \mathcal{T}_L, \quad \forall m \in A_R(t) \\
& \sum_{t \in \mathcal{T}_L} z_{it} = 1, \quad i = 1, \dots, n \\
& \sum_{j=1}^{p} a_{jt} = 1, \quad \forall t \in \mathcal{T}_B \\
& a_{jt} \in \{0, 1\}, \quad i = 1, \dots, n, \quad \forall t \in \mathcal{T}_B \\
& z_{it} \in \{0, 1\}, \quad i = 1, \dots, n, \quad \forall t \in \mathcal{T}_L
\end{aligned}
$$

5.2 Forcing at least a certain number of training data points in each leaf

The main issue in other papers on this subject when creating deeper optimal trees, seemed to be the run time of the program and overfitting. To tackle the problem of overfitting, we now want to enforce the tree to have a minimal number of training data points in each leaf. It might then be better to not apply a split at all at a certain branch node, and therefore have some empty leaves, to prohibit overfitting. The adaptations that this extension on the basic MIO formulation requires, were already described in [3], but they were not used by M. van Dijk in [2]. The adaptations are that we will add a parameter $N_{\min}$, the required minimal number of training data points in a leaf, and an indicator variable $d_t = \mathbb{1}\{\text{node } t \text{ applies a split}\}$, and indicator variable $l_t = \mathbb{1}\{\text{leaf } t \text{ contains any points}\}$.

We start by introducing l_t, defined by the constraint:

$$z_{it} \leq l_t, \quad i = 1, \ldots, n \quad \forall t \in \mathcal{T}_L \tag{12}$$

Note that we can have $l_t = 1$ even if the leaf t contains no points. However, this is not a problem for our further formulation so we do not add more constraints to define l_t. Introducing l_t means we also need to adjust constraint (7), since if a leaf t is empty, it should not be labelled with a class. The constraint to limit c_{kt} then becomes:

$$\sum_{k \in \{0,1\}} c_{kt} = l_t, \quad \forall t \in \mathcal{T}_L \tag{13}$$

Now we can enforce the $N_{\min}$ in the tree, by simply adding the constraint:

$$\sum_{i=1}^{n} z_{it} \geq l_t N_{\min}, \quad \forall t \in \mathcal{T}_L \tag{14}$$

Lastly we have to add the option for a branch node to not apply a split. We start by simply requiring all entries of a_t to be 0 when a branch node t does not apply a split, by changing constraint (1) to:

$$\sum_{j=1}^{p} a_{jt} = d_t, \quad \forall t \in \mathcal{T}_B \tag{15}$$

Furthermore, a branch node may not apply a split if the parent does not apply a split. This is to preserve the structure of the tree. We can enforce this by adding the constraint:

$$d_t \leq d_{p(t)}, \quad \forall t \in \mathcal{T}_B \setminus \{\text{root}\} \tag{16}$$

This concludes the extension to the basis MIO formulation of forcing at least a certain number of training data points in each leaf. However, when running the program, we found that there were still some issues when interpreting the results in a biological context. In Section 5.3 we explain the further changes we made to the MIO formulation to improve this.

5.3 Avoiding 'meaningless' splits

When running this program, we ran into a problem that troubled the biological interpretation of the tree. Since the tree was forced to apply a split at each branch node, this sometimes meant that when forcing an $N_{\min}$, the tree would apply a 'meaningless split' at a branch node t, rather than setting d_t to zero. With a 'meaningless split' we mean a split that still sends all training data points down the same branch, because it splits on a feature that had already been split at at an ancestor node, or it splits on a feature that is the same for all training data points in that branch node. This made the biological interpretation the tree troublesome, because if a tree splits on such a feature it does not mean that that feature has a mutation that is relevant for resistance of an organism or not. We therefore altered the MIO-formulation, so that in such cases the d_t would be set to zero, rather than applying a 'meaningless split'.

Therefore we changed z_{it} to be for all nodes $t \in \mathcal{T}$, so we could add the constraint:

$$\sum_{i=1}^{n} z_{it} \geq d_{p(t)}, \quad \forall t \in \mathcal{T} \setminus \{\text{root}\} \tag{17}$$

This constraint ensures that if a node contains no training data points, the d_t of the parent node is set to zero. We then required some more constraints for the values of z_{it}, $t \in \mathcal{T}_B$. Therefore we changed constraints (2) and (3) to:

$$a_{p(t)}^T x_i - (1 - z_{it}) \leq 0, \quad \forall t \in \mathcal{T} \setminus \{\text{root}\}, \text{ if } t \text{ is the left child of } p(t) \quad (18)$$

$$a_{p(t)}^T x_i + (1 - z_{it}) \geq 1, \quad \forall t \in \mathcal{T} \setminus \{\text{root}\}, \text{ if } t \text{ is the right child of } p(t) \quad (19)$$

These constraints imply that if the parent $p(t)$ of node t does not apply a split, so $a_{p(t)}^T$ is a null vector, then constraint (18) is always satisfied, and constraint (19) is never satisfied. This means that if a branch node does not apply a split, all data points will go to the left child of the node.

Now that we changed these constraints to only be for the parent, rather than all ancestors, we must add a constraint to preserve the structure of the tree. We must require z_{it} to be zero if it is zero in its parent node, which is executed by the following constraint:

$$z_{it} \leq z_{ip(t)}, \quad i = 1, \ldots, n \quad \forall t \in \mathcal{T} \setminus \{\text{root}\} \quad (20)$$

This constraint also ensures, together with constraint (4) that $z_{it} = 1$ for exactly one node at each depth of the tree, for all $i = 1, \ldots, n$. Therefore, the value of z_{it} is automatically set to one at the root node for $i = 1, \ldots, n$. So now z_{it} is correctly defined for all nodes $t \in \mathcal{T} \setminus \{\text{root}\}$, and not just the leaf nodes.

Combining all of this together yields the following extended MIO formulation for constructing optimal classification trees with at least N_{min} training data points each leaf and the option for a branch node not to apply a split, where the constraints that we added to the basis MIO formulation are highlighted. Note however that we, from now on, include the added constraints regarding d_t when referring to the 'basis' MIO formulation, since these do not affect the misclassification. They only help to interpret the results in a biological context. The 'extended' MIO formulation, for classification trees that have at least a certain number of training data points in each leaf, thus only contains constraint (14) as an extra constraint compared to the 'basis' MIO formulation, which is highlighted in red.

$$
\begin{aligned}
\min \quad & \sum_{t \in \mathcal{T}_L} L_t \\
\text{s.t.} \quad & L_t \geq N_t - N_{kt} - n(1 - c_{kt}), \quad k = 0,1, \quad \forall t \in \mathcal{T}_L \\
& L_t \geq 0, \quad \forall t \in \mathcal{T}_L \\
& N_{kt} = \sum_{i=1}^{n} Y_{ik} z_{it}, \quad \forall k = 0,1, \quad \forall t \in \mathcal{T}_L \\
& N_t = \sum_{i=1}^{n} z_{it}, \quad \forall t \in \mathcal{T}_L \\
& \sum_{k \in \{0,1\}} c_{kt} = l_t, \quad \forall t \in \mathcal{T}_L \\
& a_{p(t)}^T x_i - (1 - z_{it}) \leq 0, \quad \forall t \in \mathcal{T} \setminus \{\text{root}\}, \ \text{if } t \text{ is the left child of } p(t) \\
& a_{p(t)}^T x_i + (1 - z_{it}) \geq 1, \quad \forall t \in \mathcal{T} \setminus \{\text{root}\}, \ \text{if } t \text{ is the left child of } p(t) \\
& \sum_{t \in \mathcal{T}_L} z_{it} = 1, \quad i = 1, \dots, n \\
& z_{it} \leq l_t, \quad i = 1, \dots, n \quad \forall t \in \mathcal{T}_L \\
& \sum_{i=1}^{n} z_{it} \geq l_t N_{\min}, \quad \forall t \in \mathcal{T}_L \\
& \sum_{j=1}^{p} a_{jt} = d_t, \quad \forall t \in \mathcal{T}_B \\
& d_t \leq d_{p(t)}, \quad \forall t \in \mathcal{T}_B \setminus \{\text{root}\} \\
& \sum_{i=1}^{n} z_{it} \geq d_{p(t)}, \quad \forall t \in \mathcal{T} \setminus \{\text{root}\} \\
& z_{it} \leq z_{ip(t)}, \quad i = 1, \dots, n \quad \forall t \in \mathcal{T} \setminus \{\text{root}\} \\
& a_{jt} \in \{0,1\}, \quad i = 1, \dots, n, \quad \forall t \in \mathcal{T}_B \\
& d_t \in \{0,1\}, \quad \forall t \in \mathcal{T}_B \\
& z_{it} \in \{0,1\}, \quad i = 1, \dots, n, \quad \forall \mathcal{T} \\
& l_t \in \{0,1\}, \quad \forall t \in \mathcal{T}_L
\end{aligned}
$$

6 Feature sampling

In [5] a method to reduce the number of options of features to split on for each branch node is used, for a different MIO formulation with column generation. This could reduce the running time of the Optimal classification tree program, which may help to be able to construct deeper trees, which may help to further reduce overfitting.

6.1 Feature sampling procedure

The feature sampling procedure can be described by Algorithm 1. The input of the algorithm are the parameters τ, α and q_t, and it initializes $C_{\text{set}}(t) = \emptyset,\ t \in \mathcal{T}_B$ and $r = 0$. The way that the algorithm works is it selects $\alpha\%$ of the input data, and uses the CART algorithm to construct a tree CARTtemp. The features, in our case the nucleotide positions, that are being split on are returned. The selected feature of each branch node t, indicated by $C^{\text{CART}_{\text{temp}}}(t)$ is then added to the $C_{\text{set}}(t)$. $C_{\text{set}}(t)$ is a list of the features that was split on by the CART algorithm at branch node t. Therefore, the union operator here means adding the feature that is split on at branch node t, which is part of the output of the CART algorithm, to the list C_{set}. Then, if the feature that is chosen to split on at the root node has already been selected to split on at the root node a previous run of the CART algorithm, the r is increased with 1. If the feature that is chosen to split on at the root node has never been selected to split on at the root node at a previous run of the CART algorithm before, the r is again set to 0. After adding the features to the C_{set}'s and adjusting the value of r, the CART algorithm is run again. The algorithm keeps running the CART algorithm until the feature that is chosen to split on at the root node has already been selected to spit on at the root node τ times before, then the algorithm stops. The algorithm returns the q_t most frequent features j in $C_{\text{set}}(t)$ for all $t \in \mathcal{T}_B$.

The values that we used for the parameters are also taken from [5], being: $\alpha = 90\%, \tau = 300, q_{\text{root}} = \lfloor \frac{150}{\mathcal{T}_B} \rfloor, q_t = \lfloor \frac{100}{\mathcal{T}_B} \rfloor\ t \in \mathcal{T}_B \setminus \{\text{root}\}$

Algorithm 1 Feature sampling procedure

```
1: **INPUT:** Parameters $\tau, \alpha, q_t \in \mathbb{Z}_+$
2: Initialize: $C_{\text{set}}(t) = \emptyset, \forall t \in \mathcal{T}_B$ and $r = 0$
3: while $r < \tau$ do
4:    Randomly select $\alpha\%$ of data, and use CART to construct a tree $\text{CART}_{\text{temp}}$
5:    $C_{\text{set}}(t) \leftarrow C_{\text{set}}(t) \cup \{C^{\text{CART}_{\text{temp}}}(t) = j \in \{1, \dots, p\}\}, \forall t \in \mathcal{T}_B$
6:    if $C^{\text{CART}_{\text{temp}}}(\text{root}) \in C_{\text{set}}(\text{root})$ then
7:        $r \leftarrow r + 1$
8:    else
9:        $r \leftarrow 0$
10:   end if
11: end while
12: OUTPUT: The $q_t$ most frequent features $j$ in $C_{\text{set}}(t), \forall t \in \mathcal{T}_B$.
```

The difficulty in implementing the algorithm is mainly in line 5. Reason being that the feature output from the CART algorithm is not a simple list with all features listed in order. This is since the CART algorithm allows branch nodes not to apply a split. We also implemented that, however our tree then still remained fully balanced, with all data points ending in a leaf node. However, if the CART algorithm decides not to apply a split at a branch node, the branch node is turned into a leaf node and the rest of the tree is pruned, making it unbalanced. Therefore the structure of the tree has to be represented in the feature output. We will explain the structure of the feature output of the CART algorithm with the following example. Say we have a depth 3 tree, where branchnodes are numbered as shown in figure 5, and it has the feature output of the CART algorithm is the following:

$$[f_0, f_1, f_3, -2, -2, -2, f_2, -2, -2]$$

The first element of the feature output indicates the feature that is split on at the root node. It then descents left down the tree, and the next element indicates the feature at node 1, and the next at node 3. However, if it reaches a branch node that does not apply a split, or a leaf node, the element in the list is -2. It then goes back up the tree and descents the first possible new branch down the right. Then it continues the same. The tree in our example therefore actually look like the tree shown in figure 6.

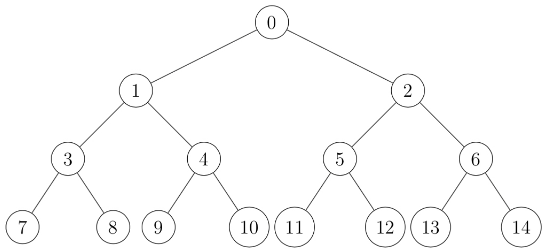


Figure 5: An example of a classification tree of depth 3.

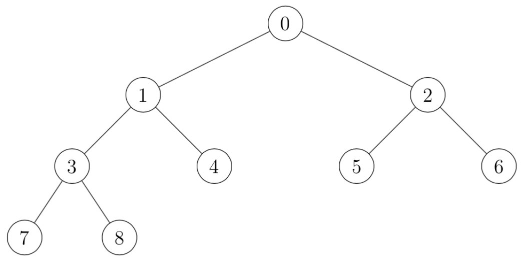


Figure 6: The classification tree of CART algorithm example.

The difficult part is keeping track of the node number that the list is at. To read the feature output from the CART algorithm we executed the exact procedure mentioned earlier in Algorithm 2. Here $C^{\text{CART}_{\text{temp}}}$ is the list that is the feature output from the CART algorithm. Then it iterates over that list. If the item in the list $C^{\text{CART}_{\text{temp}}}$ is a feature, it is added to the appropriate branch node, starting at the root node, $t = 0$. Again, the union operator here means adding an element to a list. Then the next node is the left child node, thus t is set to $2t + 1$. If the item in the list $C^{\text{CART}_{\text{temp}}}$ is not a feature, so it is -2, then we want to go back up the tree, and down the first not yet explored branch to the left. Therefore t goes back to its ancestors, until it reaches a node that is a left child. Then t is set to be the right child of the parent of that left child. This process is continued until we have iterated over all elements in the $C^{\text{CART}_{\text{temp}}}$ list. Automatically, t never becomes higher than the number of nodes minus one, where the number of nodes is $2^{D+1} - 1$. Then the algorithm is finished, and the output is the list of features that is split on by the CART algorithm, in the order of the node numbers, from the root node to the bottom right leaf.

Algorithm 2 Reading the features output from the CART algorithm

```
1: INPUT: $C^{\text{CART}_{\text{temp}}}$
2: Initialize: $t = 0$
3: for element $f$ in $C^{\text{CART}_{\text{temp}}}$ do
4:    if not $f$ equal to $-2$ then
5:       $C_{\text{set}}(t) \leftarrow C_{\text{set}}(t) \cup f,\ f \in \{1, \dots, p\}$
6:       $t \leftarrow 2t + 1$
7:    else if $k$ is not the last element in $C^{\text{CART}_{\text{temp}}}$ then
8:       Initialize: ancestor = $t$
9:       while ancestor is a right child do
10:         ancestor ← parent of ancestor
11:      end while
12:      t ← ancestor + 1
13:   end if
14: end for
15: OUTPUT: $C_{\text{set}}(t),\ t \in \mathcal{T}_B$
```

6.2 Feature sampling in MIO formulation

Now that we have the algorithm to execute feature sampling, we are yet to implement it into our MIO formulation. Luckily, this is rather simple. The only extra constraint needed is to set all entries in $a_t,\ t \in \mathcal{T}_B$ of the features that are not in $C_{\text{set}}(t)$ to zero, which can be exercised by the following constraint:

$$a_{jt} = 0, \quad \forall j \in \{1, \dots, p\} \setminus C_{\text{set}}(t), \quad \forall t \in \mathcal{T}_B \tag{21}$$

7 Results

We have now created an extended version of the MIO model that [2] and [3] constructed. First we made the output of the model much easier to interpret, in both a mathematical and a biological context. We did this by adding many `print` statements, and by circumventing forced splits, which we did by adding more constraints for d_t. Furthermore, we extended the MIO formulation by adding the option to force at least a certain number of training data points in the leaves. And lastly we added the option to apply feature sampling before running the MIO model. In this chapter we will mainly be validating our different extensions of the MIO model by running the different versions on the data sets that we received from the Westerdijk Fungal Biodiversity Institute and the CDC Atlanta combined for many different parameters, and by comparing it to the CART algorithm, in Python. We executed these validations to check whether the extensions to the program have made the program more accurate, have reduced the problem of overfitting, and have reduced the running time of the program, in which case we could create deeper trees.

7.1 Information on the Python program

We ran our MIO model on the data by implementing it in Python. The solver we used to find the optimal solution for the MIO formulation is Gurobi. The results in this chapter were determined by running the program 100 times, and calculating the accuracies. The in-sample/out-of-sample accuracy is defined as:

$$\text{accuracy} = \frac{\text{mean number of correctly classified training/test data points}}{\text{total number of training/test data points per run}} \cdot 100\%$$

.

The basis shell of the program simply looked like this for running the program once:

```
from OptimalClassificationTree import *
print('Data is imported!')
test = percentage of data used as test data
depth = depth of the tree
```

```
feature_sampling = enter 'y' or 'n'
Nmin_and_dt = enter 'n' or a list of N_min values
OCT(text,depth,feature_sampling,Nmin_and_dt)
```

And when running the program multiple times to calculate the accuracy the last line of the basis shell was changed to:

```
nr_of_runs = number of times you want to run the program
OCT_Error(text,depth,feature_sampling,Nmin_and_dt,nr_of_runs)
```

For determining the accuracy of the CART algorithm on our data this last line was changed to this:

```
CART_Error(test,depth,nr_of_runs)
```

In the validations in this chapter, the `nr_of_runs` was always set to 100.

7.2 The hotspot data set

We first ran our MIO model on the data set from the Westerdijk Fungal Biodiversity Institute that only consisted of the known first hotspot of the FKS1 gene, containing three mutations. There was only one resistant isolate in the data set, which was the only one to contain a mutation in the hotspot on amino acid position 1895: GAT $\longrightarrow$ GGT, meaning amino acid mutation D $\longrightarrow$ G on position 632. We therefore conclude from this that this is a relevant mutation for causing resistance, though the data set is too small to be certain.

7.3 Comparing different portions of test data

From the MSc thesis by Van Dijk [2] it was clear that if you take a larger portion of training data from the data set, and therefore a smaller portion of test data, the in-sample accuracy would decrease, and the out-of-sample accuracy would increase. In this section we will show the results of when we tested whether this was also the case for our two large data sets of the genes FKS1 and FKS2.

	MIO		CART	
Portion test data	in-sample accuracy	out-of-sample accuracy	in-sample accuracy	out-of-sample accuracy
12.5%	88.02%	86.83%	87.09%	85.6%
25%	88.17%	86.62%	87.15%	85.59%
37.5%	88.18%	86.48%	87.3%	85.78%
50%	88.47%	86.16%	87.53%	85.46%

Table 1: Accuracy results for depth two trees, created with the basis MIO formulation and CART, for different portions of test data from the data of FKS1.

	MIO		CART	
Portion test data	in-sample accuracy	out-of-sample accuracy	in-sample accuracy	out-of-sample accuracy
12.5%	89.51%	88.80%	89.31%	88.6%
25%	89.59%	88.20%	89.1%	88.69%
37.5%	89.87%	87.17%	89.2%	88.31%
50%	90.01%	86.94%	89.38%	87.89%

Table 2: Accuracy results for depth two trees, created with the basis MIO formulation and CART, for different portions of test data from the data of FKS2.

As shown in table 1 and 2, the in-sample accuracy did not change much when adjusting the portion of the data used as test data. Hence, we have decided that from now on we will only test an extension of the program by taking either 12.5% 25% of the data as test data, instead of testing it on multiple different portions. We chose these percentages, since if the number of training data points is lower, the run time is shorter, but it is always better to use sufficient training data to get accurate results, therefore we decided on these distribution. We generally chose 25%, for run time reasons, when we wanted to run the program for many different values of parameters, and 12.5% if we wanted to place results in a biological context.

7.4 Comparing basis MIO model with and without feature sampling

In this section we will compare the accuracy of the basis MIO formulation to the accuracy of the MIO model with feature sampling applied. We will also talk about the run time improvement by using feature sampling, which was another aim of our research.

Gene	feature sampling applied	in-sample accuracy	out-of-sample accuracy
FKS1	no	88.17%	86.62%
	yes	87.24%	86.30%
FKS2	no	89.59%	88.20%
	yes	89.49%	88.85%

Table 3: Accuracy results for depth two trees, created with the basis MIO formulation, with and without feature sampling, with 25% used as test data.

When running the MIO model for depth 2 trees 100 times without feature sampling, with 12.5% of the data used as test data and N_{min} set to zero, the run time of the program for both the FKS1 and FKS2 gene was around half an hour. When using feature sampling this was reduced to only one third of the time.

7.5 Comparing different depths

In this section we will compare the accuracy of the basis MIO model for trees of different depths. We will also be comparing these trees of different depths to trees of such depths created by the CART algorithm. For run time reasons we were obliged to apply feature sampling. The results are shown in Tables 4 and 5. It is also interesting to note that the run time hardly increased when increasing the depth of the trees, a fantastic result of feature sampling.

	MIO		CART	
Depth	in-sample accuracy	out-of-sample accuracy	in-sample accuracy	out-of-sample accuracy
2	87.23%	86.22%	87.21%	85.53%
3	88.28%	86.23%	87.87%	86.15%
4	88.85%	86.41%	88.4%	86.22%
5	88.84%	86.10%	88.73%	86.16%

Table 4: Accuracy results for trees of different depths, created with the basis MIO model and feature sampling, and the CART algorithm, from the data of the FKS1 gene, with 25% used as test data.

	MIO		CART	
Depth	in-sample accuracy	out-of-sample accuracy	in-sample accuracy	out-of-sample accuracy
2	89.49%	88.85%	89.31%	88.56%
3	90.26%	88.44%	90.07%	87.70%
4	91.24%	87.83%	91.06%	87.98%
5	91.82%	87.68%	91.53%	87.79%

Table 5: Accuracy results for trees of different depths, created with the basis MIO model and feature sampling, and the CART algorithm, from the data of the FKS2 gene, with 25% used as test data.

7.6 Comparing different depths in combination with different N_{min}

In this section we will compare the accuracy of the basis MIO model to the accuracy of the MIO model, when extended with the option to force at least a number of N_{min} training data points in each leaf, for different depths. We first determined, using the basis MIO model, the number of training data points in the leaf containing the least training data points for trees of depth 2, and then took the mean of that number of 100 trees. For FKS1 this mean was equal to 3.37 training data points, and for FKS2 this mean was equal to 2.44 training data points. Based on these numbers we decided which different values for N_{min} we were going to look at. The results are shown in Tables 6 and 7 for the FKS2 and FKS1 gene respectively.

depth	N_{min}	in-sample accuracy	out-of-sample accuracy
2	0	89.55%	88.47%
	1	89.63%	88.26%
	2	89.54%	88.41%
	3	89.2%	88.22%
	4	88.8%	87.67%
3	0	90.31%	87.99%
	1	90.21%	88.14%
	2	90.05%	87.64%
	3	89.58%	88.01%
	4	88.99%	87.80%
	5	88.56%	87.05%
4	0	91.22%	87.64%
	1	91.31%	87.54%
	2	90.14%	88.07%
	3	89.72%	88.28%
	4	89.02%	87.32%

Table 6: Accuracy results for different depth trees, created with the extended MIOM and threshold sampling, for different values for N_{min}, from the data of the FKS2 gene, with 25% used as test data.

depth	$N_{\min}$	in-sample accuracy	out-of-sample accuracy
2	0	87.33%	85.98%
	1	87.19%	86.52%
	2	87.08%	86.22%
	3	86.99%	86.36%
	4	86.79%	86.69%
	5	86.74%	86.6%
	6	86.58%	86.33%
3	0	86.99%	86.36%
	1	88.24%	86.02%
	2	87.83%	86.25%
	3	87.39%	86.30%
	4	87.44%	86.40%
	5	87.28%	86.30%
	6	86.97%	86.16%
4	0	88.92%	85.88%
	1	88.84%	85.94%
	2	87.81%	86.1%
	3	87.62%	86.22%
	4	87.27%	86.48%
	5	87.07%	86.68%
	6	86.87%	86.15%

Table 7: Accuracy results for different depth trees, created with the extended MIOM and threshold sampling, for different values for $N_{\min}$, from the data of the FKS1 gene, with 25% used as test data.

7.7 An example of a classification tree created by the MIO model

In Figure 7 an example of a classification tree produced by our MIO model is shown.

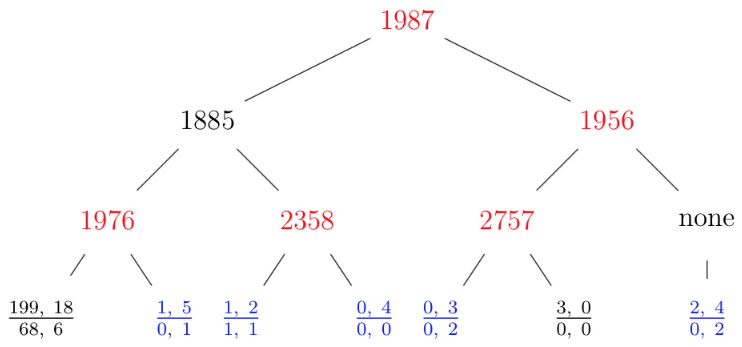


Figure 7: An example of a classification tree of depth 3 created by the extended MIO model, with feature sampling, $N_{\text{min}} = 3$, 25% as test data, where the black feature in a branch node is a nucleotide position in the FKS1 gene, and the red features in the branch nodes are nucleotide positions in the FKS2 gene. The leaves are black if their label is susceptible, and blue if it is resistant. In the leaves the distribution is shown, where the top numbers represent the total number of 'susceptible, resistant' training data points in that leaf, and the bottom numbers represent those of the test data.

7.8 Using protein sequences instead of DNA sequences

In MEGA we are able to translate the DNA sequences into protein sequences, as explained in Section 3.1.2. In this section we will discuss the accuracy of the MIO model when using these protein sequence data sets. In Section 7.9 we will elaborate on the biological context of these results.

gene	sequence type	in-sample accuracy	out-of-sample accuracy
FKS1	DNA	87.24%	86.30%
	protein	92.23%	90.85%
FKS2	DNA	89.49%	88.85%
	protein	92.73%	91.10%
both combined	protein	92.77%	91.23%

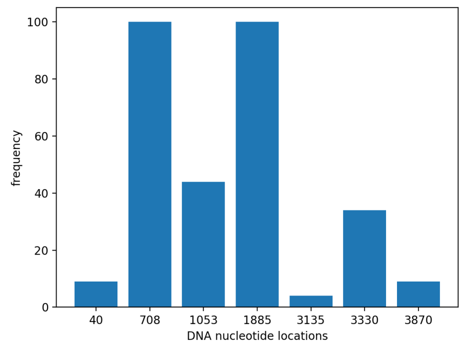
Table 8: Accuracy results for trees of depth 2, created with the basis MIO model and feature sampling, from the data of the FKS1 and FKS2 gene, with 25% used as test data.

7.9 Interpreting the results in biological context

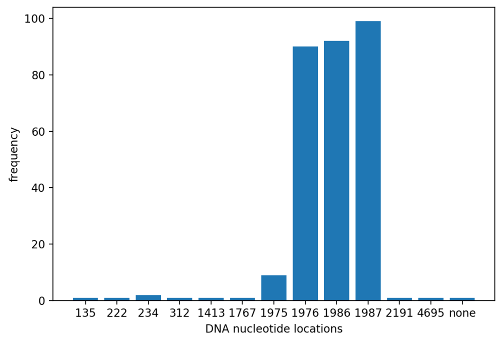
After running the basis MIO model 100 times on the DNA sequences data in Section 7.3, we also let the program generate a dictionary containing all the locations in the DNA of the FKS1 and FKS2 gene that are chosen to split on in those 100 trees, and the number of times that they are chosen. With this information we were able to make the bar plots 8a and 8b, from which we can identify the nucleotide positions of mutations for which there is most evidence of being related to resistance.

We did the same for the protein sequence data sets, which gave the bar plots as shown in Figure 9a and 9b for the FKS1 and FKS2 gene respectively.

We then repeated this process one more time, for the DNA sequence data sets of the FKS1 and FKS2 gene combined, which is shown in Figure 10a, and for the protein sequence data sets of both genes combined, which is shown in Figure 10b

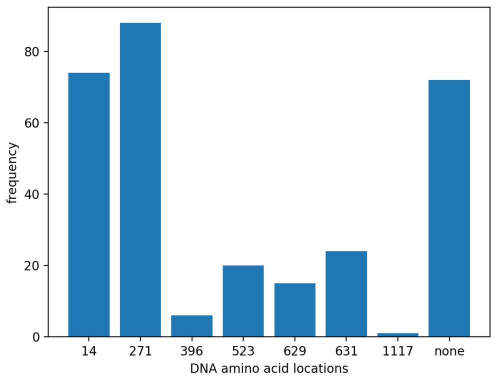


(a)

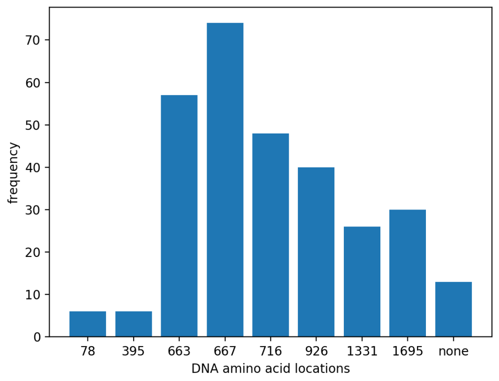


(b)

Figure 8: Two bar plots of the number of times that nucleotide locations in the DNA sequences of the FKS1 (a) and FKS2 (b) gene are chosen to split on, in depth 2 trees without feature sampling and $N_{\text{min}} = 0$, with 12.5% of the data used as test data.

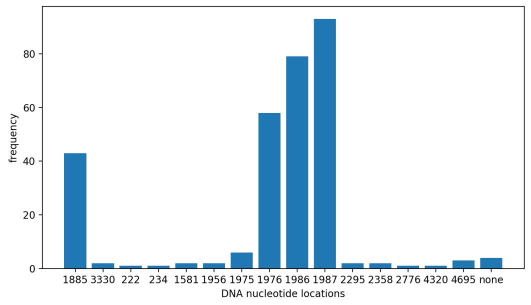


(a)

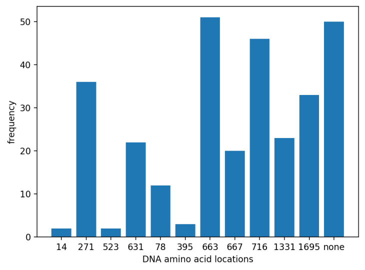


(b)

Figure 9: Two bar plots of the number of times that amino acid locations in the protein sequences of the FKS1 (a) and FKS2 (b) gene are chosen to split on, in depth 2 trees without feature sampling and $N_{\min} = 0$, with 12.5% of the data used as test data.



(a)



(b)

Figure 10: Two bar plots of the number of times that nucleotide (a) and amino acid (b) locations in FKS1 and FKS2 combined are chosen to split on, in depth 2 trees using feature sampling, with $N_{\min} = 0$, and with 25% of the data used as test data. The two left nucleotide locations and the four left amino acid locations refer to FKS1, and the other locations refer to FKS2.

8 Conclusion

We wrote a Python program to preprocess and merge all data sets from the Westerdijk Fungal Biodiversity Institute and CDC Atlanta automatically, and wrote a user manual on how to use it in Section 3.2. We then succeeded in making the output of the python program of the model by Van Dijk much easier to interpret, in both a mathematical and a biological context, by returning more information as output. We then extended the basis MIO formulation by circumventing meaningless splits, and forcing at least a certain number of training data points in each leaf, as explained in Chapters 5.2 and 5.3. Lastly we succeeded in applying feature sampling, which allows us to create deeper trees without run time issues. We will now discuss the results that we found in Chapter 7 when validating our model.

Generally we have found that the percentages did not fluctuate much in terms of out-of-sample and in-sample accuracy for different generated training and test sets. Van Dijk found that the quality of his model was highly dependent on the distribution of the test and training data. We resolved this problem by using a much larger data set to perform our analysis on and using 100 repetitions when validating our method, while Van Dijk used only 20.

In Table 1 we can see that if the portion of the data on the FKS1 gene that is used as test data is increased, the in-sample accuracy also increases, but the out-of-sample accuracy decreases. The same is true for the data on the FKS2 gene, as shown in Table 2. This is also what Van Dijk found in his MSc thesis [2].

Furthermore, our MIO model has both a higher in-sample accuracy as well as a higher out-of-sample accuracy than the CART algorithm for the data on the FKS1 gene. For the FKS2 gene, the in-sample accuracy of our MIO model is higher than that of the CART algorithm, though this is not the case for the out-of-sample accuracy, which is roughly equal for the two algorithms.

Lastly, the in-sample accuracy of our MIO model is 2.5% to 5% higher (in absolute percentages) than what Van Dijk found, and the out-of-sample accuracy is around 15% higher, which we think is mostly due to the larger data set.

In Table 3 the accuracy of the basis MIO model with and without using feature sampling is shown for both genes, with 25% of the data used as test data. From these results we can conclude the following. For FKS1 the in-sample accuracy is around 0.9% lower when using feature sampling, and the out-of-sample accuracy only around 0.3% lower. For FKS2 the in-sample accuracy is around 0.1% lower, though the out-of-sample accuracy is 0.65% higher when using features sampling than without. These are nice results, especially considering that the run time is around 3 times as short for depth 2 trees, and that the run time does not increase for deeper trees.

In Tables 4 and 5 one can see that for both genes the in-sample accuracy increased when the depth of the trees was increased. The out-of-sample accuracy did not change significantly for trees of different depths for FKS1, and it decreased slightly for FKS2, but that could be because it is not necessary to make deep trees for this data set. For both genes and for all depths, our MIO model was more accurate or approximately equally as accurate as the CART algorithm.

In Tables 6 and 7 forcing at least a number of N_{min} training data points in each leaf definitely had a positive impact on the out-of-sample accuracy. While the in-sample accuracy decreased when increasing the N_{min} for different depths, the out-of-sample accuracy was slightly, up to 0.5%, higher for certain values of N_{min} than when $N_{\text{min}} = 0$. Therefore we can conclude that forcing an N_{min} can reduce overfitting, especially when creating deeper trees.

In Table 8 we found that both the in-sample accuracy as well as the out-of-sample accuracy is much higher when using the protein sequences of the FKS1 and FKS2 gene, than when using the DNA sequences, namely around 5% for FKS1 and around 3% for FKS2. When combining the data sets of both genes into one, both the in-sample accuracy and the out-of-sample accuracy were the highest that we found over all, namely 92.77% and 91.23% respectively.

In Sections 7.2 and 7.9 we have found several nucleotide and amino acid positions that seem important in the mechanisms that cause resistance in *C. glabrata*.

In Section 7.2 it is explained that from the small hotspot data set we found that the mutation in the hotspot on amino acid position 1895: GAT $\longrightarrow$ GGT, meaning amino acid mutation D $\longrightarrow$ G on position 632, might be a relevant mutation for causing resistance, thoug the data set is too small to be certain.

In Figure 8a we can see that the nucleotide positions 708 and 1885 of FKS1 are selected most often by the MIO model. Nucleotide position 1885 translates to amino acid position 629, which is indeed located in the first hotspot of the FKS1 gene [6], and therefore an accurate result. However, we also found that nucleotide position 708 appears often in trees constructed by the MIO model, but when translating it to a protein we realized that this is a silent mutation. Hence, it is unlikely to be indeed related to resistance. Such errors can be avoided by using amino acid sequences as input.

In Figure 9a we can see that the most often occuring amino acid positions for FKS1 are 14 and 271. However, a mutation in position 271 is only found in one isolate, so there is not enough information to say that this mutation is relevant for causing resistance. The biological consequence of this mutation could be researched further to check whether this is relevant or not. Increasing N_{min} could help to avoid finding mutations that appear only in a limited number of sequences. Amino acid position 14 translates to the mutation in nucleotide postion 14, which is also found relevant in Figure 8a, though it is not in a known hotspot. Investigating the biological consequence of this mutation might be interesting for further research. The only amino acid positions that are found that are located in a known hotspot are those on position 629 and 631, although they are not very frequent in our bar plot.

In Figure 8b we can see that there are several nucleotide positions for FKS2 that seem important in causing resistance, namely 1975, 1976, 1986 and 1987, which are all located in a known hotspot, and non of these are silent mutations. However, only position 1987 out of these four positions also shows up in Figure 9b, translated to amino acid location 663. The other amino acid locations that have a high frequency in Figure 9b are not located in a known hotspot, so they may also be interesting for further biological research.

Lastly in Figure 10 we can see the important nucleotide and amino acid positions that we found when running the program on the combined data set of both genes. The most frequent nucleotide positions are 1885 of FKS1, and 1976, 1986 and 1987 of FKS2, which are all located in known hotspots and are not silent mutations. Therefore these mutations could indeed be related to resistance. The most frequent amino acid positions are 271 and 631 of FKS1, and 663, 716 and 1695 of FKS2. Again, a mutation in position 271 of

FKS1 is only found in one isolate, so we do not think that we have enough evidence to conclude that this mutation is related to causing resistance. The same is true for position 716 and 1695 of FKS2. However, position 631 of FKS1 and 663 of FKS2 are located in known hotspots, so these results are promising.

Our overall conclusion of the results found in Chapter 7 is that a bigger data set indeed led to both a higher in-sample and out-of-sample accuracy. We outperformed the CART algorithm. Feature sampling indeed led to a much shorter run time, while still being nearly as accurate as when the program was run without using it. Moreover, feature sampling made it possible to create deeper trees, without problems with running time or overfitting. For deeper trees, the out-of-sample accuracy remained roughly the same. By forcing at least N_{min} training data points in each leaf, we were able to increase the out-of-sample accuracy slightly, for all depths, with certain values of N_{min} depending on the depth, thus we were able to reduce the effect of overfitting slightly. Running the program on the protein sequences instead of the DNA sequences gave unexpectedly high in-sample and out-of-sample accuracies for both genes, especially for the data set of both genes combined. When placing our result in biological context, we validated some of our results with the already available knowledge on hotspots in the FKS1 and FKS2 gene, and found that some seemingly insignificant mutations were still being picked up by our MIO model. This can be avoided by forcing at least N_{min} training data points in each leaf, and by using protein sequences as input, rather than DNA sequences. Moreover, we have also found some mutations that may be interesting for further research.

9 Discussion

9.1 Interpreting the validations of our model

When executing a comparative analysis on the results found in Chapter 7, it is important to keep in mind that there were still some fluctuations of around 0.2% in the results.

Furthermore, we used the data set that Van Dijk used from the CDC Atlanta to validate our MIO model in Chapter 7, along with a different data set from the Westerdijk Fungal Biodiversity Institute. This not only meant that we had a larger data set than Van Dijk had in his MSc thesis, but also that the ratio of susceptible and resistant isolates was different. Only 15% of our data set was resistant, which was 48 isolates, whereas 35% of Van Dijks data was resistant. Furthermore, he distributed the data into training and test data differently than we did. Therefore, if the tree did not apply any splits, then all of the data points would end up in the bottom left leaf, and because there were more susceptible data points, the label of the leaf would be susceptible, and all resistant data points would be misclassified. Because of the way that we distributed our data into training and test data, if we used 25% of our data as test data, which was 323 isolates in total, the in-sample accuracy was $\frac{323 \cdot 0.75 - 48 \cdot 0.75}{323 \cdot 0.75} = 85.12\%$ if the tree did not apply any splits, and the out-of-sample accuracy was $\frac{323 \cdot 0.25 - 48 \cdot 0.25}{323 \cdot 0.25} = 85.19\%$. Therefore one should also keep in mind, when executing a comparative analysis on the results found in Chapter 7, that an in-sample accuracy of 85.12% and an out-of-sample accuracy of 85.19% were the trivial lower bound on the optimum, using our MIO model and distribution of the data. That is also the reason that the results in Table 7 do not get worse after $N_{\text{min}} = 10$, and after $N_{\text{min}} = 20$ in Table 6, after increasing this value further, for they have reached the lower bound on the optimum.

9.2 Using protein sequences instead of DNA sequences

As we discussed previously, there are certain advantages and disadvantages to use protein sequences instead of DNA sequences.

A major advantage is the three-fold downsizing of data set, since three nucleotide columns are replaced by a single amino acid column. This results in a shorter run time. Furthermore, if a nucleotide mutation is silent (meaning it does not change the encoded amino acid), it will be removed from the protein input sequence, which further downsizes the data set. Moreover, removal of silent mutations makes the data set biologically more relevant, since silent mutations can not confer resistance. This is illustrated in our data set: when we ran the program using DNA sequences we identified nucleotide 708 as a hotspot 8a. However, this is not a known hotspot, and it turns out that mutation is silent (GCC $\longrightarrow$ GCT, which are both translated into Alanine). This error was avoided when using protein sequences as input 9a.

There is also a major disadvantage of using protein sequences. We converted the protein sequences to binary data, meaning changes were indicated by a one for any given column, and no changes by a zero. However, there are 20 amino acids, and it could be that at a given position, 19 amino acids are drug sensitive and only 1 amino acid change confers drug resistance (this appears indeed to be the case at some positions: see reference [6]). In such a scenario, our model may fail to detect the importance of this change, since the mutation is indicated equally as any other non-relevant mutation.

This is the case for the nucleotide position 1976 in the FKS2 gene for example. In the Excel data file it is shown that three different types of mutations have taken place on this position in different isolates: a T $\longrightarrow$ C mutation in two resistant isolates, a T $\longrightarrow$ A mutation in one resistant isolate, which both are not silent mutations, and a deletion in three isolates. On nucleotide position 1975 there is also a deletion in three isolates, and a T $\longrightarrow$ G mutation in one, which is not silent. All these different mutations in these two adjacent nucleotide positions translate to different amino acids on amino acid location 659, but since the data is binary our model will fail to differentiate between these different mutations in the amino acid locations.

This is also an issue when using DNA sequences, but here the detection problem is much smaller (there are only 4 nucleotides, and usually only two per column). This difference in detection sensitivity may explain why the hotspots that were identified in the DNA sequences (fig 8) do not exactly align with the hotspots in the protein sequences (fig 9).

However, when we compare DNA and protein hotspot sequences and look for overlap, we may be able to find the most important hotspots. Indeed, hotspot amino acid 629 of FKS1 corresponds to hotspot nucleotide 1885 and hotspot amino acid 663 of FKS2 corresponds to the hotspot nucleotides 1986 and 1987. Both amino acids were indeed shown to be true hotspots when using drug sensitivity tests [6].

Note that these problems would be resolved when using one hot encoding when converting the data sets to binary data, or by using ternary trees instead of binary trees, as Van Dijk did [2].

9.3 Further research

As mentioned before, this BSc thesis was based on improving the results found by the MSc thesis by Van Dijk. However, there are of course still more things that could be researched within the topic of Mixed Integer Optimization to create optimal classification trees to classify *Candida* isolates. Considering the success regarding overfitting of applying feature sampling combined with forcing at least a certain number of training data points in a leaf, it might be interesting to apply this on data sets where deeper trees are needed, for this did not seem to be the case with our data sets.

We also discovered the limitations of converting our data sets to binary data when placing the results in a biological context. A solution would be to combine this research with that of Van Dijk, by not converting the data to binary data, and creating ternary trees as done by Van Dijk, in combination with our extensions of feature sampling and forcing at least a certain number of training data points in each leaf.

Other suggestions for further research are to not just look at the frequency of chosen locations, but also take into consideration which locations were chosen to split at at the root node, for example, since this is probably the most important split.

We also found very high in-sample and out-of-sample accuracies when running the program on the protein sequences, rather than the DNA sequences, especially when combining the data sets of the FKS1 and FKS2 gene together. So further research on running different versions of the MIO on these data sets might give more interesting results, not just regarding the mathematical point of view, but also the biological. We decided not to do all our validations in Chapter 7 on the protein sequences, because we could not translate the Excel data into protein sequences, and we wanted to do our validations on as large a data set as possible.

Lastly, as mentioned in the conclusion, some locations in the FKS1 and FKS2 gene that were marked as significant for causing resistance by our model are not in known hotspots. Further research on the biological influence of these mutations would give more clarity on whether these mutations are relevant or not.

A Appendix

A.1 OptimalClassifiationTree.py

```
from gurobipy import *
import numpy
import Bio
import sklearn.datasets
from Bio.Seq import MutableSeq
from Bio import SeqIO
from Bio.Seq import Seq
from numpy import zeros
from numpy import ones
from numpy import vstack
from numpy import hstack
from sklearn import tree
from sklearn.preprocessing import normalize
from numpy import linalg
from sklearn import datasets
import random
import EditData

################################################################################
# Data import

gene = 'FKS1'


## Data from CLC excel files

filename_data5 = 'CLC_Data_Analysis_Susceptible_strains_finished'
total_data5 = 55

filename_data6 = 'CLC_Data_Analysis_Resistant_strains_finished'
total_data6 = 28

filedirectory = r'/Users/maaike/Dropbox/Documenten/TU/TW_bachelor_jaar_3/'
filedirectory += 'BEP/programma/data/'
```

```
X_5 = EditData.readdata_excel(filename_data5,filedirectory,total_data5,gene)
X_6 = EditData.readdata_excel(filename_data6,filedirectory,total_data6,gene)
resistant_sequences5 = []
resistant_sequences6 = list(range(total_data6))
print('Number of susceptible sequences from excel: ',len(X_5))
print('Number of susceptible columns from excel: ',len(X_5[0]))
print('Number of resistant sequences from excel: ',len(X_6))
print('Number of resistant columns from excel: ',len(X_6[0]))

X_unordered1, columns_withmutations1, resistant_sequences1, total_data1 = \
    EditData.mergefiles(
    X_5[1:],X_5[0],resistant_sequences5,total_data5,
    X_6[1:],X_6[0],resistant_sequences6,total_data6)
print('Number of sequences from excel: ',len(X_unordered1))
print('Number of columns from excel: ',len(X_unordered1[0]))



## Data from new fasta files

total_data2 = 240

if gene == 'FKS1':
    filename_data2 = 'PRJNA524686_FKS1_edited'
    ##filename_data2 = 'PRJNA524686_FKS1_proteins'
else:
    filename_data2 = 'PRJNA524686_FKS2_edited'
    ##filename_data2 = 'PRJNA524686_FKS2_proteins'

X_unordered2, columns_withmutations2 = EditData.createbinarydata(
    filename_data2)
print('Number of sequences from fasta: ',len(X_unordered2))
print('Number of columns from fasta: ',len(X_unordered2[0]))

filename_MICvalues = 'MIC_values_edited'
sheetname = 'All isolates'
filedirectory = r'/Users/maaike/Dropbox/Documenten/TU/TW_bachelor_jaar_3/'
filedirectory += 'BEP/programma/data/'
```

```
Y_unordered, resistant_sequences2 = EditData.determineclass(
    filename_MICvalues,filedirectory,sheetname,total_data2)


## combining both data sets

X_unordered_combined, columns_withmutations, resistant_sequences, total_data =\
    EditData.mergefiles(
    X_unordered1,columns_withmutations1[:-1],resistant_sequences1,total_data1,
    X_unordered2,columns_withmutations2[:-1],resistant_sequences2,total_data2)


X, Y, data_S, data_R = EditData.createordereddata(
    X_unordered_combined,resistant_sequences,total_data)
print('Number of sequences combined: ',len(X))
print('Number of columns combined: ',len(X[0]))


################################################################################

def Create_train_and_test(test,data_feature_sampling):

    test = test/100
    train = 1 - test

    #Create indices train and test
    if data_feature_sampling == 'n':
        n_train = round(total_data*train)
        train_index_resistant = sorted(random.sample(
            range(data_S,total_data),math.ceil(data_R*train)))
        train_index_susceptible = sorted(random.sample(
            range(data_S),n_train-len(train_index_resistant)))

        test_index_susceptible = sorted(list(set(
            range(data_S))- set(train_index_susceptible)))
        test_index_resistant = sorted(list(set(
            range(data_S,total_data)) - set(train_index_resistant)))
    else:
        total_data_feature_sampling = len(data_feature_sampling[0])+len(
            data_feature_sampling[1])
        n_train_feature_sampling = round(total_data_feature_sampling*train)
        train_index_resistant_full = data_feature_sampling[0]
        train_index_susceptible_full = data_feature_sampling[1]

        train_index_resistant = sorted(random.sample(set(
            train_index_resistant_full),math.ceil(
                len(train_index_resistant_full)*train)))
        train_index_susceptible = sorted(random.sample(set(
            train_index_susceptible_full),n_train_feature_sampling-len(
                train_index_resistant)))

    #Create train and test sets
    X_Train = []
    for i in train_index_susceptible:
        X_Train.append(X[i])
    for i in train_index_resistant:
        X_Train.append(X[i])
    X_Test = []
    if data_feature_sampling == 'n':
        for i in range(len(test_index_susceptible)):
            X_Test.append(X[test_index_susceptible[i]])
        for i in range(len(test_index_resistant)):
            X_Test.append(X[test_index_resistant[i]])

    Y_Train = [0]*len(train_index_susceptible) + [1]*len(train_index_resistant)
    if data_feature_sampling == 'n':
        Y_Test = [0]*len(test_index_susceptible) + [1]*len(test_index_resistant)
    else:
        Y_Test = []

    return X_Train,X_Test,Y_Train,Y_Test,train_index_resistant,\
           train_index_susceptible

################################################################################

def CART_features(alpha,depth,X_Train,Y_Train):
    # Tree CART
    D = depth
    clf=tree.DecisionTreeClassifier(max_depth=D)
    clf=clf.fit(X_Train,Y_Train)
    features_CART = clf.tree_.feature

    return features_CART

################################################################################

def CART_algorithm(test,depth):

    X_Train,X_Test,Y_Train,Y_Test,t_i_r,t_i_s = Create_train_and_test(test,'n')

    # Tree CART
    D = depth
    clf=tree.DecisionTreeClassifier(max_depth=D)
    clf=clf.fit(X_Train,Y_Train)

    #calculate misclassification
    out_of_sample_misclassification = 0
    in_sample_misclassification = 0

    for i in range(len(Y_Test)):
        if clf.predict([X_Test[i]])!=Y_Test[i]:
            out_of_sample_misclassification += 1
    for i in range(len(Y_Train)):
        if clf.predict([X_Train[i]])!=Y_Train[i]:
            in_sample_misclassification += 1

    #print(clf.tree_.feature)
    return in_sample_misclassification, out_of_sample_misclassification

################################################################################

def Feature_Sampling(depth,data_feature_sampling):
```

```
tau = 300
alpha = 90
D = depth
Tb = int((2**(D+1)-1)/2)
q_root = int(150/Tb)
q_j = int(100/Tb)

C_set = [[] for i in range(Tb)]
r = 0

while r < tau:
    test = 100 - alpha
    X_Train,X_Test,Y_Train,Y_Test,t_i_r,t_i_s = Create_train_and_test(
        test,data_feature_sampling)
    C_CART_temp = CART_features(alpha,D,X_Train,Y_Train)

    if C_CART_temp[0] in C_set[0]:
        r += 1
    else:
        r = 0


    node_index = 0

    for k in range(len(C_CART_temp)):

    #this code reads the features output given by the CART algorithm
    #which walks through a tree via the left
    #and indicates a non-split by -2

        #if a node applies a split, add it to C_set
        if C_CART_temp[k] != -2:
            C_set[node_index].append(C_CART_temp[k])
            node_index = node_index*2 + 1

        #if node does not apply a split, and it is not the last node,
        #go back up the tree to a new branchnode
        elif k != len(C_CART_temp)-1:
            ancestor = node_index
            while ancestor % 2 == 0: #while parent right child
                ancestor = int((ancestor-1)/2)
            node_index = ancestor + 1 #go back up

features_notused = []
features_restricted = []

for t in range(Tb):

    #make a dictionary of the frequency of the features
    max_features = {}
    for feature in C_set[t]:
        if feature in max_features.keys():
            max_features[feature] += 1
        else:
            max_features[feature] = 1

    #make a list of the features sorted by frequency
    sorted_max_features = sorted(max_features, key=max_features.get, \
                                 reverse=True)

    #determine select q_j most frequent features
    if t == 0:
        features_restricted.append(sorted_max_features[0:q_root])
    else:
        features_restricted.append(sorted_max_features[0:q_j])

    #determine features that are not in those most frequent features
    features_notused_t = []
    for feature in range(len(X[0])):
        if not (feature in features_restricted[t]):
            features_notused_t.append(feature)
    features_notused.append(features_notused_t)

print('Set of restricted features: ',features_restricted)
locations_restricted = [[] for i in range(len(features_restricted))]
for node_nr in range(len(features_restricted)):
        for elt_feature in features_restricted[node_nr]:
            locations_restricted[node_nr].append(columns_withmutations[
                elt_feature])
    print('Which are the locations: ',locations_restricted)
    return features_notused, features_restricted

##############################################################################

def OCT(test,depth,feature_sampling,Nmin_and_dt):

    X_Train,X_Test,Y_Train,Y_Test,train_index_resistant,\
    train_index_susceptible = Create_train_and_test(test,'n')

    if feature_sampling == 'y':
        data_feature_sampling = [train_index_resistant]+[
            train_index_susceptible]
        features_notused, features_restricted = Feature_Sampling(
            depth,data_feature_sampling)
        print('Feature sampling done!\n')


    if Nmin_and_dt == 'n':
        Nmin_lst = [0]
    else:
        Nmin_lst  = Nmin_and_dt


    ##############################################################################
    #loop to run program several times with other N_min values

    in_sample_misclassification_lst = []
    out_of_sample_misclassification_lst = []
    text = ''

    for run_time in Nmin_lst:

        #Create a new model called Optimal Classification Tree (OCT)
        m = Model("OCT")
```

```
#Number of classes
K = 2

#Set tree depth D
D = depth

#Number of features to select on
p = len(X_Train[0])

#Minimum number of elements in leaves
Nmin = run_time

#Number of sequences
num_of_seq = len(X_Train)

#Leaf nodes and branch nodes
T = 2**(D+1)-1
Tb = range(0,int(T/2))
Tl = range(int(T/2),T)

#Left- and right ancestors
Ar=[] #list of right ancestors per leaf
Al=[] #list of left ancestors per leaf
for i in Tl:
    left_ancestors=[]
    right_ancestors=[]
    node=i
    while node>0:
        parent=int((node-1)/2)
        if node%2 == 1: #if i odd
            left_ancestors.append(parent)
        else: # if i even
            right_ancestors.append(parent)
        node = parent
    Ar.append(right_ancestors)
    Al.append(left_ancestors)
```

```
#Create variables
a = m.addVars(p,len(Tb), vtype=GRB.BINARY, name="a")

z = m.addVars(num_of_seq,T, vtype=GRB.BINARY, name="z")

l = m.addVars(len(Tl), vtype = GRB.BINARY, name="l")

N = m.addVars(K, len(Tl), vtype=GRB.CONTINUOUS, name="N")

N2 = m.addVars(len(Tl), vtype=GRB.CONTINUOUS, name="N2")

c2 = m.addVars(K,len(Tl), vtype=GRB.BINARY, name="c2")

L = m.addVars(len(Tl), vtype = GRB.CONTINUOUS, name="L")

d = m.addVars(len(Tb), vtype=GRB.BINARY, name="d")

m.update()


#Objective
sum_list=[]
for i in range(len(Tl)):
    sum_list.append(L[i])
m.setObjective(sum(sum_list), GRB.MINIMIZE)


# (15)
for t in Tb:
    sum_list = []
    for j in range(p):
        sum_list.append(a[j,t])
    m.addConstr(sum(sum_list)==d[t])

# (16)
for t in Tb:
    if t != 0:
        pt = int((t-1)/2) #parent
        m.addConstr(d[t]<=d[pt])

# (12)
for t in range(len(Tl)):
    for j in range(num_of_seq):
        m.addConstr(z[j,t+len(Tb)]<=l[t])

# (4)
for j in range(num_of_seq):
    sum_list = []
    for t in range(len(Tl)):
        sum_list.append(z[j,t+len(Tb)])
    m.addConstr(sum(sum_list)==1)

# (20)
for j in range(num_of_seq):
    for t in range(1,T):
        pt = int((t-1)/2)
        m.addConstr(z[j,t]<=z[j,pt])

# (18)
for t in range(1,T):
    if t%2 == 1:
        q = int((t-1)/2)
        for j in range(num_of_seq):
            sum_list=[]
            for k in range(p):
                sum_list.append(X_Train[j][k]*a[k,q])
            m.addConstr(sum(sum_list)-(1-z[j,t])<=0)

# (19)
for t in range(1,T):
    if t%2 == 0:
        q = int((t-1)/2)
        for j in range(num_of_seq):
            sum_list=[]
            for k in range(p):
                    sum_list.append(X_Train[j][k]*a[k,q])
                m.addConstr(sum(sum_list)+(1-z[j,t])>=1)

# Create Y adjacency matrix
Ymatrix = numpy.zeros((num_of_seq,K))
for i in range(num_of_seq):
    for k in range(K):
        if Y_Train[i]==k:
            Ymatrix[i,k]=1
        else:
            Ymatrix[i,k]=-1

# (6)
for k in range(K):
    for t in range(len(Tl)):
        sum_list=[]
        for j in range(num_of_seq):
            sum_list.append((1+Ymatrix[j][k])*z[j,t+len(Tb)])
        m.addConstr(N[k,t]==0.5*sum(sum_list))

# (5)
for t in range(len(Tl)):
    sum_list=[]
    for j in range(num_of_seq):
        sum_list.append(z[j,t+len(Tb)])
    m.addConstr(N2[t]==sum(sum_list))

# (13)
for t in range(len(Tl)):
    sum_list=[]
    for k in range(K):
        sum_list.append(c2[k,t])
    m.addConstr(sum(sum_list)==l[t])

# (9)
for t in range(len(Tl)):
    for k in range(K):
        m.addConstr(L[t]>=(N2[t]-N[k,t]-num_of_seq*(1-c2[k,t])))
    m.addConstr(L[t]>=0)

# (14)
if Nmin_and_dt != 'n':
    for t in range(len(Tl)):
        sum_list=[]
        for j in range(num_of_seq):
            sum_list.append(z[j,t+len(Tb)])
        m.addConstr(sum(sum_list)>=l[t]*Nmin)

# (17)
for t in range(1,T):
    sum_list=[]
    for j in range(num_of_seq):
        sum_list.append(z[j,t])
    pt = int((t-1)/2) #parent
    m.addConstr(sum(sum_list)>=d[pt])

# ()
if feature_sampling == 'y':
    for t in Tb:
        if features_restricted[t]:
            for j in range(p):
                if j in features_notused[t]:
                    m.addConstr(a[j,t]==0)
        else:
            m.addConstr(d[t]==0)


#Create optimal classification tree
m.optimize()

########################################################################

##Determine structure of OCT en classify test data

#determines the splitting features (columns)
split_features=[]
```

```
split_features_print=[]
no_split_applied = len(X[0])-1
for t in Tb:
    sum_indices_a_t = []
    for i in range(p):
        sum_indices_a_t.append(round(a[i,t].X))
        if round(a[i,t].X)==1:
            split_features.append(i)
            split_features_print.append(i)
    if sum(sum_indices_a_t) == 0:
        split_features.append(no_split_applied)
        split_features_print.append('no split')

#classifies test data
leaf_test=[]
leaf_test_susceptible = [0]*len(Tl)
leaf_test_resistant = [0]*len(Tl)
for t in range(len(Tl)):
    data_points_per_leaf = []
    for i in range(len(X_Test)):
        #determines if datapoint i is in leaf t
        #p is not a feature but an ancestor node
        if all([X_Test[i][split_features[pt]]== 0 for pt in Al[t]]) \
           and all([X_Test[i][split_features[qt]]==1 for qt in Ar[t]]):
            data_points_per_leaf.append(i)
            if Y_Test[i]==0:
                leaf_test_susceptible[t] += 1
            else:
                leaf_test_resistant[t] += 1
    leaf_test.append(data_points_per_leaf)

#determines out of sample misclassification
out_of_sample_misclassification = 0
leaf_classes = []
for t in range(len(Tl)):
    leaf_class = numpy.argmax([c2[0,t].X,c2[1,t].X])
    leaf_classes.append(leaf_class)
    if leaf_test[t]: #if leaf is non-empty with test data
```

```
        for i in leaf_test[t]:
            if Y_Test[i]!= leaf_class:
                out_of_sample_misclassification += 1

#determines in sample misclassification
in_sample_misclassification = int(m.objVal)
in_sample_misclassification_lst.append(in_sample_misclassification)
out_of_sample_misclassification_lst.append(
    out_of_sample_misclassification)

#determines locations in DNA that was split on
split_locations = []
for split_feature in split_features:
    split_locations.append(columns_withmutations[split_feature])

#########################################################################

#prints useful information about OCT
text += '\nN_min = ' + str(Nmin)
if Nmin_and_dt != 'n':
    text += '\ndt = '
    for t in Tb:
        text += str(int(d[t].X))+" "
text += '\nThe features that are split on: ' + str(
    split_features_print)
text += '\nWhich are locations: ' + str(split_locations)
text += '\nAmount of susceptible traindatapoints in leaves: '
for t in range(len(Tl)):
    text += str(int(N[0,t].X))+" "
text += '\nAmount of resistant traindatapoints in leaves: '
for t in range(len(Tl)):
    text += str(int(N[1,t].X))+" "
text += '\nClasses of the leaves: ' + str(leaf_classes)
text += '\nThe in sample misclassification is ' + str(
    in_sample_misclassification)
text += '\nAmount of susceptible testdatapoints in leaves: ' + str(
    leaf_test_susceptible)
text += '\nAmount of resistant testdatapoints in leaves: ' + str(
            leaf_test_resistant)
        text += '\nThe out of sample misclassification is ' + str(
            out_of_sample_misclassification)
        text += '\n'+'#'*50 + '\n'

    print(text)

    return in_sample_misclassification_lst, \
           out_of_sample_misclassification_lst, split_locations

################################################################################

def OCT_Error(test,depth,feature_sampling,Nmin_and_dt,n_runtimes):

    if Nmin_and_dt != 'n':
        Nmin_and_dt = [Nmin_and_dt]

    #run OCT n_runtimes times and calculate total misclassification errors
    sum_lst_in_sample_misclass = []
    sum_lst_out_sample_misclass = []
    split_locations_lst = []
    for run_nr in range(n_runtimes):
        print('This is the ', run_nr+1, 'th run')
        temp_in_sample_misclass_lst, temp_out_sample_misclass_lst, \
            split_locations = OCT(test,depth,feature_sampling,Nmin_and_dt)
        sum_lst_in_sample_misclass.append(temp_in_sample_misclass_lst[0])
        sum_lst_out_sample_misclass.append(temp_out_sample_misclass_lst[0])
        for split_location in split_locations:
            split_locations_lst.append(split_location)

    #print('sum lsts ',sum_lst_in_sample_misclass,sum_lst_out_sample_misclass)

    #calculate mean misclassification errors
    mean_in_sample_misclass = sum(sum_lst_in_sample_misclass)/n_runtimes
    mean_out_sample_misclass = sum(sum_lst_out_sample_misclass)/n_runtimes

    #calculate the misclassification error percentages
    n_train_data = round(total_data*(100-test)/100)
```

```
n_test_data = total_data - n_train_data
perc_in_sample_misclass = (mean_in_sample_misclass/n_train_data)*100
perc_out_sample_misclass = (mean_out_sample_misclass/n_test_data)*100
accuracy_in_sample = 100 - perc_in_sample_misclass
accuracy_out_sample = 100 - perc_out_sample_misclass

#create dictionary of the frequency of split locations
split_locations_dict = {}
for split_location in split_locations_lst:
    if split_location in split_locations_dict.keys():
        split_locations_dict[split_location] += 1
    else:
        split_locations_dict[split_location] = 1

text = ''
text += 'The program has been run ' + str(n_runtimes) + ' times \n'
text += 'Data from gene '+gene+' was used'
text += ' with test = ' + str(test) + '% \n'
if feature_sampling == 'n':
    text += '  Feature sampling was not used \n'
else:
    text += '  Feature sampling was used \n'
if Nmin_and_dt == 'n':
    text += '  The Nmin was 0 \n'
else:
    text += '  The Nmin was ' + str(Nmin_and_dt[0]) + '\n'
text += '  The depth of the trees was ' + str(depth) + '\n'
text += 'The mean in sample misclassification error is: ' + str(
    round(mean_in_sample_misclass,2)) + '\n'
text += 'and the mean out of sample misclassification error is: ' + str(
    round(mean_out_sample_misclass,2)) + '\n'
text += 'So the in sample accuracy is: ' + str(round(accuracy_in_sample,2)) +
text += 'and the out sample accuracy is: ' + str(round(
    accuracy_out_sample,2))
text += '\nThe frequency of locations in the DNA being chosen to split on: \n'
text += str(split_locations_dict)
text += '\nWhich in order of most frequent to least are: \n'
text += str(sorted(split_locations_dict, key=split_locations_dict.get, \
                        reverse=True))

    print(text)
    print(str(round(accuracy_in_sample,2))+'\% & '+str(round(
        accuracy_out_sample,2))+'\%')

    return accuracy_in_sample, accuracy_out_sample

################################################################################

def CART_Error(test,depth,n_runtimes):

    sum_lst_in_sample_misclass = []
    sum_lst_out_sample_misclass = []
    for run_nr in range(n_runtimes):
        print('This is the ', run_nr+1, 'th run')
        temp_in_sample_misclass, temp_out_sample_misclass = CART_algorithm(
            test,depth)
        sum_lst_in_sample_misclass.append(temp_in_sample_misclass)
        sum_lst_out_sample_misclass.append(temp_out_sample_misclass)

    #calculate mean misclassification errors
    mean_in_sample_misclass = sum(sum_lst_in_sample_misclass)/n_runtimes
    mean_out_sample_misclass = sum(sum_lst_out_sample_misclass)/n_runtimes

    #calculate the misclassification error percentages
    n_train_data = round(total_data*(100-test)/100)
    n_test_data = total_data - n_train_data
    perc_in_sample_misclass = (mean_in_sample_misclass/n_train_data)*100
    perc_out_sample_misclass = (mean_out_sample_misclass/n_test_data)*100
    accuracy_in_sample = 100 - perc_in_sample_misclass
    accuracy_out_sample = 100 - perc_out_sample_misclass

    text = ''
    text += 'The program has been run ' + str(n_runtimes) + ' times \n'
    text += 'Data from gene '+gene+' was used'
    text += ' with test = ' + str(test) + '% \n'
    text += 'The depth of the trees was ' + str(depth) + '\n'
    text += 'The mean in sample misclassification error is: ' + str(round(
        mean_in_sample_misclass,2)) + '\n'
    text += 'and the mean out of sample misclassification error is: ' + str(
        round(mean_out_sample_misclass,2)) + '\n'
    text += 'So the in sample accuracy is: ' + str(round(accuracy_in_sample,2)) +
    text += 'and the out sample accuracy is: ' + str(round(
        accuracy_out_sample,2))

    print(text)
    print('& '+str(round(accuracy_in_sample,2))+'\% & '+str(round(
        accuracy_out_sample,2))+'\%')

    return accuracy_in_sample, accuracy_out_sample
```

A.2 EditData.py

```
import Bio
from Bio.Seq import MutableSeq
from Bio import SeqIO
from Bio.Seq import Seq
import pandas as pd



def readdata_excel(filename,filedirectory,total_data,gene):
    #imports DNA data from excel documents

    #import data
    file_directory = filedirectory+filename+'.xlsx'
    data = pd.read_excel(file_directory, sheet_name = gene)

    #create X
    X = []
    for row_nr in range(total_data+1):
        row = data.iloc[row_nr,:]
        X.append(list(row))
    return X

def mergefiles(X_unordered1,position_nrs1,resistant_sequences1,total_data1,
               X_unordered2,position_nrs2,resistant_sequences2,total_data2):
    #merges two DNA data sets, both X and Y

    #combine data
    columnindex_1 = 0
    columnindex_2 = 0
    locations = []
    X_combined = [[] for i in range(len(X_unordered1)+len(X_unordered2))]
    while columnindex_1 < len(position_nrs1) and columnindex_2 < len(
        position_nrs2):

        #if both column indices are at the same DNA position:
        if position_nrs1[columnindex_1] == position_nrs2[columnindex_2]:
```

```
        #add elements
        for rowindex_1 in range(len(X_unordered1)):
            X_combined[rowindex_1].append(X_unordered1[rowindex_1][
                columnindex_1])
        for rowindex_2 in range(len(X_unordered2)):
            X_combined[rowindex_2+len(X_unordered1)].append(
                X_unordered2[rowindex_2][columnindex_2])
        locations.append(position_nrs1[columnindex_1])
        columnindex_1 += 1
        columnindex_2 += 1
    #if columnindex_1 is at an earlier DNA position:
    elif position_nrs1[columnindex_1] < position_nrs2[columnindex_2]:
        for rowindex_1 in range(len(X_unordered1)):
            X_combined[rowindex_1].append(X_unordered1[rowindex_1][
                columnindex_1])
        for rowindex_2 in range(len(X_unordered2)):
            X_combined[rowindex_2+len(X_unordered1)].append(0)
        locations.append(position_nrs1[columnindex_1])
        columnindex_1 += 1
    #if columnindex_2 is at an earlier DNA position:
    else:
        for rowindex_1 in range(len(X_unordered1)):
            X_combined[rowindex_1].append(0)
        for rowindex_2 in range(len(X_unordered2)):
            X_combined[rowindex_2+len(X_unordered1)].append(
                X_unordered2[rowindex_2][columnindex_2])
        locations.append(position_nrs2[columnindex_2])
        columnindex_2 += 1
locations.append('none')

#calculate total_data
total_data = total_data1 + total_data2

#determine resistant_sequences
for item_nr in range(len(resistant_sequences2)):
    resistant_sequences2[item_nr] = resistant_sequences2[item_nr] +\
                                    total_data1
resistant_sequences = resistant_sequences1 + resistant_sequences2
```

```
    return X_combined, locations, resistant_sequences, total_data


def determineclass(filename,filedirectory,sheetname,total_data):
    #determines whether sequences are resistant or not by MIC value

    file_directory = filedirectory+filename+'.xlsx'
    data = pd.read_excel(file_directory, sheet_name = sheetname)

    #read values and calculate their sums
    Y_unordered = []
    for row_nr in range(0,total_data):
        #print(data.iloc[:,13])
        row = []
        MIC1 = data.iloc[row_nr, 11]
        MIC2 = data.iloc[row_nr, 12]
        MIC3 = data.iloc[row_nr, 13]
        row = sum([float(MIC1),float(MIC2),float(MIC3)])
        Y_unordered.append(row)
    #print(Y_unordered)

    #determine their class by their sum
    resistant_sequences = []
    for elt_nr in range(len(Y_unordered)):
        if Y_unordered[elt_nr] < 1.5:
            Y_unordered[elt_nr] = 0
        else:
            Y_unordered[elt_nr] = 1
            resistant_sequences.append(elt_nr)

    return Y_unordered, resistant_sequences


def readdata_fasta(filename):
    #makes list of data from document

    X=[]
    for seq_record in SeqIO.parse(filename+".fasta", "fasta"):
        seq = seq_record.seq
        mutable_seq = seq.tomutable()
        sequence=[]
        for j in range(0,len(mutable_seq)):
            sequence.append(mutable_seq[j])
        X.append(sequence)
    return X

def findwildtype(X,i):
    #finds wildtype letter of column i

    #create dictionary of column i of the data
    wildtype_dict = {}
    for elt_index in range(len(X)):
        elt_str = X[elt_index][i]
        if elt_str in wildtype_dict.keys():
            wildtype_dict[elt_str] += 1
        else:
            wildtype_dict[elt_str] = 1
    #print(wildtype_dict)

    #find most common element in column
    wildtype_val = 0
    wildtype_elt = 0
    for key in wildtype_dict.keys():
        if wildtype_dict[key] > wildtype_val:
            wildtype_val = wildtype_dict[key]
            wildtype_elt = key
    return wildtype_elt



def createwildtype(filename, X):
    #creates wildtype list

    wildtype = []
    for i in range(0,len(X[0])):
        wildtype_elt = findwildtype(X,i)
        wildtype.append(wildtype_elt)

    return wildtype


def createbinarydata(filename):
    #creates binary array of data

    X = readdata_fasta(filename)
    wildtype = createwildtype(filename, X)

    #binarydata = np.zeros((len(X),len(X[0])))
    binarydata = []
    for i in range(len(X)):
        binaryseq = []
        for j in range(len(X[0])):
            if X[i][j] != wildtype[j]:
                binaryseq.append(1)
            else:
                binaryseq.append(0)
        binarydata.append(binaryseq)
    X_shorter, columns_withmutations = removeredundantcolumns(binarydata)

    return X_shorter, columns_withmutations

def removeredundantcolumns(X):
    #removes columns where all sequences have the same value

    X_shorter = [[] for i in range(len(X))]
    columns_withmutations = []
    for column_nr in range(len(X[0])):
        column_sum = 0
        for row_nr in range(len(X)):
            column_sum += X[row_nr][column_nr]
        if column_sum != 0:
            columns_withmutations.append(column_nr+1)
            for row_nr in range(len(X)):
                X_shorter[row_nr].append(X[row_nr][column_nr])
    columns_withmutations.append('none')

    return X_shorter, columns_withmutations


def createordereddata(X_unordered,resistant_sequences,total_data):
    #creates an ordered X with susceptible first, then resistant
    #and creates a Y
    #from a resistant indices list

    X_susceptible = []
    X_resistant = []
    for sequence_nr in range(total_data):
        if sequence_nr in resistant_sequences:
            X_resistant.append(X_unordered[sequence_nr])
        else:
            X_susceptible.append(X_unordered[sequence_nr])
    X_ordered = X_susceptible + X_resistant
    data_S = len(X_susceptible)
    data_R = len(X_resistant)

    #add a zeros column, used when introducing d_t
    for row in range(len(X_ordered)):
        X_ordered[row].append(0)

    Y = [0]*data_S + [1]*data_R
    print('number of susceptible isolates: ',data_S)
    print('number of resistant isolates: ',data_R)

    return X_ordered, Y, data_S, data_R
```

References

[1] MEGA. https://www.megasoftware.net. Accessed: 28-06-2019.

[2] M. Dijk, van. Classifying candida species using mixed integer optimization based optimal classification trees. Master's thesis, Delft University of Technology, 2019. `http://resolver.tudelft.nl/uuid:068ff836-099b-4abe-8d9e-cf96706169df`.

[3] Bertsimas D. Dunn, J. Optimal classification trees. *Machine Learning*, 106(7):1039–1082, 2017.

[4] Rajendra Prasad (ed.). *Candida albicans: Cellular and Molecular Biology*. Springer, 2017.

[5] Crognier G.-Gabor A. F. Zhang Y. Hurkens C.A.J. Firat, M. Constructing classification trees using column generation. *arXiv:1810.06684 [cs.LG]*, 2018.

[6] Alastruey-Izquierdo A. Healey K. R. Johnson M. E. Perlin D. S. Edlind T. D. Katiyar, S. K. FKS1 and FKS2 are functionally redundant but differentially regulated in candida glabrata: Implications for echinocandin resistance. *Antimicrobial Agents and Chemotherapy*, 56(12):6304–6309, 2012.

[7] Bolden-C. B. Kuykendall R. J. Lockhart S. R. Pham, C. D. Development of a luminex-based multiplex assay for detection of mutations conferring resistance to echinocandins in candida glabrata. *Journal of clinical microbiology*, 52(3):790–795, 2014.

[8] Negri M.-Henriques M. Oliveira R. Williams D. W. Azeredo J. Silva, S. Candida glabrata, candida parapsilosis and candida tropicalis: biology, epidemiology, pathogenicity and antifungal resistance. *Journal of fungi*, 36(2):288–305, 2017.

[9] N. Yapar. Epidemiology and risk factors for invasive candidiasis. *Therapeutics and clinical risk management*, (10):95–105, 2014.