DESIGN REPORT

MAKING AUTO COMPLETE AND DID-YOU-MEAN FUNCTIONALITY FOR KRAMP

By:

Tim Blok – s1467727

Frans van Dijk – s0200204

Yannick Mijsters – s1499459

Ramon Onis – s1461176

Tim Sonderen – s1465252

Faculty of Electrical Engineering, Mathematics and Computer Science

TI Design Project

Supervised by:

dr. Mannes Poel

UNIVERSITY OF TWENTE.

APRIL 13, 2016

Contents

1 Introduction **4**
1.1 Requirements . . . 4
1.1.1 User requirements . . . 4
1.1.2 System requirements . . . 5
1.2 Project phases . . . 6

2 Research into auto complete **8**
2.1 Defining search queries and search terms . . . 8
2.2 General idea . . . 8
2.3 Data . . . 8
2.4 Algorithm options . . . 9
2.4.1 Trie . . . 9
2.4.2 Ternary search tree . . . 10
2.4.3 Prefix tree . . . 10
2.5 Conclusion . . . 11

3 Research into did-you-mean **12**
3.1 General idea . . . 12
3.2 Definitions . . . 12
3.3 Algorithm options . . . 12
3.3.1 Associative array . . . 13
3.3.2 BK-tree . . . 13
3.3.3 Levenshtein automata . . . 14
3.3.4 Using string length to partition the dataset . . . 14
3.3.5 Winnow based spell checker algorithm . . . 15
3.3.6 Other possible trees . . . 15
3.4 Relevancy options . . . 16
3.4.1 Levenshtein distance . . . 16
3.4.2 Likelihood of search strings . . . 16
3.4.3 Search results per search string . . . 16
3.5 Data . . . 16
3.6 Conclusion . . . 16
3.6.1 Data structure . . . 16
3.6.2 Relevancy options . . . 17

4 Design auto complete **18**
4.1 Algorithms . . . 18
4.2 Class diagram . . . 19

5 Design did-you-mean 20
5.1 Introduction . . . 20
5.2 BK-tree . . . 20
5.3 Levenshtein automata . . . 20
5.3.1 Levenshtein automata using NFA to DFA conversion . . . 21
5.3.2 Improvements on Levenshtein automata . . . 21
5.4 Class diagram did-you-mean . . . 22
5.4.1 BK-trees . . . 22
5.4.2 Levenshtein automata . . . 22

6 Design choices 23
6.1 Designing different implementations of the same functionality . . . 23
6.2 No user-specific search . . . 23
6.3 Fill auto complete with did-you-mean results . . . 23

7 Implementation 25
7.1 Auto complete . . . 25
7.2 Did-you-mean . . . 25
7.2.1 BK-tree . . . 25
7.2.2 Levenshtein automata . . . 25

8 Testing 27
8.1 Test plan . . . 27
8.1.1 Introduction . . . 27
8.1.2 Implementation tests . . . 27
8.1.3 Measuring improvement . . . 27
8.1.4 User experience testing . . . 28
8.2 Evaluation . . . 28
8.2.1 Results of implementation tests . . . 28
8.2.2 Acceptance test . . . 28
8.2.3 User experience testing . . . 28

9 Performance 30
9.1 Auto complete . . . 30
9.2 Did-you-mean . . . 30
9.2.1 BK-tree . . . 30
9.2.2 Levenshtein automata . . . 30
9.2.3 BK-tree versus Levenshtein automata . . . 31

10 Integration 32
10.1 The API . . . 32
10.1.1 Database controller . . . 32
10.1.2 The auto complete . . . 32
10.1.3 The did-you-mean . . . 32
10.1.4 The controller . . . 33
10.2 General use . . . 33
10.3 Use per category and/or per user . . . 33
10.4 Possible improvements/additions . . . 34

11 Discussion & Conclusion **35**
11.1 Discussion . . . 35
11.2 Conclusion . . . 35
11.2.1 Must . . . 35
11.2.2 Should . . . 36
11.2.3 Could . . . 36
11.2.4 Won't . . . 36

Bibliography **37**

A Example prefix tree **38**

B Example BK-tree **39**

C Example Levenshtein NFA **40**

D Algorithms **41**
D.1 Trie . . . 41
D.2 Auto complete . . . 42
D.3 Did-you-mean . . . 44
D.3.1 BK-tree . . . 44
D.3.2 Levenshtein automata . . . 46

E Class diagrams **49**

F Global overview **53**

G Test plan **54**

H GUI **57**

I Boxplots of did-you-mean query times **59**

J Meetings with Kramp **61**
J.1 Meeting 0 . . . 61
J.2 Meeting 1 (Skype) . . . 61
J.3 Meeting 2 (Skype) . . . 62
J.4 Meeting 3 (Skype) . . . 62
J.5 Meeting 4 (Skype) . . . 63
J.6 Meeting 5 (Skype) . . . 63
J.7 Meeting 6 (Skype) . . . 63
J.8 Meeting 7 (Skype) . . . 64
J.9 Meeting 8 (Skype) . . . 64

K Sprint reports **65**
K.1 Sprint 1 . . . 65
K.2 Sprint 2 . . . 65
K.3 Sprint 3 . . . 65
K.4 Sprint 4 . . . 66
K.5 Sprint 5 . . . 66
K.6 Sprint 6 . . . 66

Chapter 1

Introduction

Kramp is a company based in Varsseveld that stores and sells items for agricultural business and garages. This can range from tractors to boots and work outfits. On their part Kramp promises that for most parts, if an item gets ordered, the item will be delivered early in the morning the day afterwards. This generally means that they will deliver it before the businesses opens the next day. Most of the orders that these businesses make are done on the web shop of Kramp.

This web shop, as most web shops, has a search functionality. To help out the user with searching for items, Kramp has an auto complete and did-you-mean functionality. The auto complete functionality is the menu that appears under the search field with suggestions for the current search string that you are typing. The did-you-mean functionality is the function that shows suggestions of a better search string after a search is executed and possibly did not yield desired results. These suggestions generally start with 'Did you mean ...?'.

Kramp currently has both these functionalities running on their web shop. However, they are not completely satisfied with the current system, since it does not deal with product numbers and typing mistakes very well.

This report shows the research of a group of students (the writers of this report) of the University of Twente to find a better implementation of both these functionalities for the web shop of Kramp.

1.1 Requirements

Firstly, a list of user requirements are made. These are gathered from meetings with and emails from Kramp. From this, a list of system requirements is made and categorized in different priorities.

1.1.1 User requirements

1. Implement a did-you-mean functionality
2. Implement an auto complete functionality
3. A manual should be made on how to integrate the functionality into Kramp's system
4. Pre-processing is preferred over calculations at runtime
5. The user should not have to wait long before getting a suggestion

6. The functionality should not be built directly into the currently existing Kramp system
7. Test measurable differences between old and new system
8. Kramp's software should be able to communicate with the system
9. The did-you-mean should only return one suggestion
10. The auto complete should be able to give back multiple suggestions
11. Give an elaborate report about the system's performance
12. Kramp should be able to use javadoc to understand specific code segments
13. A GUI should be made to show the proof of concept

1.1.2 System requirements

The system requirements are prioritized according to the MoSCoW-method: Must, Should, Could and Won't. Every table corresponds to one of the priorities. Furthermore, they are documented using the SMART-principle, however, without the time component, since there is no real planning yet and saying everything should be done before the end of the project is redundant and irrelevant.

Must

These are the requirements that must be done before the project can be called done.

1. The system must be able to use auto complete to give suggestions for a search string
2. The system must be able to use did-you-mean to give a suggestion after a search string has been used
3. The system must contain a simple GUI for the proof of concept, which showcases both functionalities

The first three requirements are the core of the project, these are specifically what Kramp gave as core requirements and therefore are essential to be completed. The GUI is a more technical must. There must be a GUI for the proof of concept (PoC), because otherwise no clear example for the auto complete functionality can be shown, since it should change for every letter that you type which is not possible in a TUI.

Should

These are the requirements that should be done at the end of the project, they are preferred, but not crucial for the project.

1. The system should learn every time a user enters a search string
2. The system should learn which search strings are successful by looking at previous searches
3. The system should correct typing mistakes in both the did-you-mean and auto complete

4. The system should link similar search strings and only suggest search strings that give the best results

The first and third requirement are smaller parts of the did-you-mean or auto complete functionality that are important for an optimal implementation of these functionalities and for the best user experience when using these functionalities. The second and fourth requirements are about making the most of the data that could be gathered for an optimal user experience. All these are parts of possible implementation and improvements on the most basic way to implement these functionalities. Therefore, they are not a must, but definitely a should for a result that will make the user happy.

Could

These are requirements that could be implemented, they would be nice features for a final product, however, far from essential to finishing the project.

1. The system could support fully personalized search functionality
2. The system could link (mistyped) searches with previously bought items in a personalized way
3. The system could keep track of single search words instead of entire search strings

All these requirements are ways to improve a basic implementation of the did-you-mean and auto complete functionality. They are far from essential for the project. However, they would make for an improved user experience and more precise results. Therefore, they would definitely be positive additions to the project.

Won't

These are the requirements that will not be implemented. These are important to show the bounds of the project and to show Kramp what they can not expect from this project.

1. The system won't be fully implemented system in Kramp's webshop
2. The system won't improve product information with product data
3. The system won't improve the entire search algorithm

1.2 Project phases

The project will go through multiple phases. The first phase, in which goals and requirements are defined, is covered in the project proposal, which is a separate document on its own. For this reason, it will not be covered again in this design report, except for the requirements and meetings, which are listed below.

The second phase will consist of research into the different functionalities. In this phase, different algorithms will be researched and looked at which might be the best solution. A conclusion will be made from all the research, and a specific approach will be chosen.

The third phase will be the designing of the chosen approach in the previous phase. This will include pseudo-code and explanations of the algorithms deemed best, and a diagram of the expected class structure.

The fourth phase will be the implementation of the design. This part will not be documented in this report aside from design choices. Except for from programming and documenting algorithms, unit tests will also be written to make sure that the code does not contain any errors.

In the final phase the final documentation and explanation of the code will be done. This includes a manual on how to integrate the code with another code base, and how to extend the code if desired.

Chapter 2

Research into auto complete

Research is conducted to find different ways to design the auto complete. Different options are listed and advantages and disadvantages are derived. Additionally, information is gathered about the data that is used and how it is processed and inserted in the algorithm. This will help to make an educated decision about which algorithm to use.

2.1 Defining search queries and search terms

To clarify the difference between input from a user and already collected data, two different terms will be used.

A 'search query' is a full search term (can be multiple words) which has been used by a user and has been captured in a database. A search query is therefore a piece of data which is always available.

A 'search string' is a full search term (can be multiple words) which has been input by a user at the time of searching. This is only momentarily available. A search string can be stored, however, making it a search query.

2.2 General idea

The auto complete functionality completes the current search string to a likely search query. For example, when a customer types 'batte' it is very likely that this will result in the word 'battery'. The auto complete will, therefore, suggest this word together with other related search queries that are possible completions of 'batte'. The auto complete function will suggest a complete search query from the point where only one letter was typed. Data structures will be used that are filled with search queries that were used in the past, by users, to search for products. The weight of search queries will be the deciding factor in the choice if a search query is suggested or not. The weight is a number indicating the predicted usefulness of a search query.

2.3 Data

The data that was received from Kramp contains a list of search queries. For each search query it is listed how many times it was searched for, the number of page views per search, the percent-

age of search exits, the percentage of search refinements, the time that is spent on that page after a search and the average search depth. The most important data for this project is the search query, the number of times it was searched and the percentage of search refinements. From the percentage of search refinements a percentage of successful searches can be derived after which a weight can be calculated for every search query. These weights with their corresponding search query will be used in the chosen algorithm so that only search queries that are generally successful are suggested. How the weights will be calculated is to be determined by testing different formulas for calculating the weights.

2.4 Algorithm options

From the provided data suggestions must be given as quickly as possible. How fast suggestions can be given depends on the algorithm used which in turn closely relates to the data structure into which the data is put. During the research period several data structures were found. All of them are trees, as this facilitates an efficient way to search for a string. The disadvantage of a tree is that it needs to be initialized, which is time consuming. However, this only needs to be done once. Furthermore, after the data is stored in a tree in memory, the searches can be performed much faster than a linear search ($O(\log n)$ instead of $O(n)$, where n is the search word length). The amount of memory that is used and the amount of processing power it saves depends on what kind of tree is used. In this section multiple options will be discussed.

2.4.1 Trie

The National Institute of Standards and Technology defines a trie as "A tree for storing strings, in which there is one node for every common prefix. These strings are stored in extra leaf nodes."[1] In other words, a trie is a tree where the path to every node is a prefix where every node in the path adds one letter so that at every leaf there is a complete word. The value of a node is the representation of the likelihood that that node is the desired end node. This value will only be assigned to the node if the word at that point exists in the dictionary.

To enhance the trie for auto complete specifically, a number of words that are likely to be the desired word of a certain node can be added to the trie. This way the algorithm will use more memory, but the execution time of a search will be faster.

The trie will be built when the algorithm is initialized and then stored in memory. This way the trie can be used very quickly for searches, but it is relatively slow to update it. Because of this, it is required to regularly re-initialize the algorithm to keep the auto complete up-to-date.

Advantages

- Good search time complexity compared to other options

Disadvantages

- Needs initializing which is slow
- Bad space complexity

2.4.2 Ternary search tree

A ternary search tree (TST)[3] a type of trie. The root of a TST is a letter, instead of an empty string as in a trie. Instead of creating a child node for every letter that could be next as is done in a trie, a TST has a maximum of three child nodes. The left child node contains a letter which has a lower value than its parent, the right child node a letter with a higher value. The child node in the middle is the next letter, which can come after its parent.

When a search for a specific string is conducted in a TST, first is checked if the value of the current node is the equal to the first letter of the string that is searched. If this is true, the search continues to the next node and the first letter of the string is deleted. This continues until either the string ends or a leaf is reached.

If the first letter of the string is not equal to the value of the current node, the search is continued in either the left or the right child node, depending on whether the value of the first letter of the string is higher or lower than the value of the current node.

Advantages

- Decent space complexity
- Uses less memory than a normal trie

Disadvantages

- Needs initializing which is slow
- little worse time complexity than a normal trie

2.4.3 Prefix tree

In a trie a single node could contain multiple however to reduce the time needed for building and updating the tree the tree can also be build with only a single character per node. This also has some adventages when querying the tree. This type of trie implementation will be called prefix tree in this report. Furthermore to reduce memory costs the whole words are not stored in the leaf since the entire word is already stored on the path to the leaf, the leaf only contains the weight or score of the word. And for faster querying the top k the highest score of the children of a node is also stored in the node. In Figure A.1 an example of a prefix tree is depicted.

By implementing the prefix tree this way, a given prefix (the letters that are already typed by the user) corresponds to one node in the tree. By recursively descending the tree, choosing each child node which corresponds to a letter with the highest weight, the keyword with the highest weight, in a dictionary of n keywords, can be found in a time proportional to $O(\log n)$. Getting the best k keywords by weight, which is the problem to be solved, by a given prefix can be done in $O(k \log n)$. Furthermore, the data structure that is needed requires space proportional to $O(n)$[2].

Advantages

- Able to provide the best k results
- Good time complexity

Disadvantages

- Needs initializing which is slow
- Bad space complexity

2.5 Conclusion

The normal trie and the TST are roughly the same complexity, except for the fact that the trie uses more memory to search faster while the TST saves memory and searches a little slower because of that. The prefix tree is a lot like a normal trie, but it stores a little extra data to be faster and to be able to give the best k keywords. This requires a little extra memory. Because Kramp preferred pre-processing, the best choice is the prefix tree. The prefix tree has the best time complexity together with the normal trie and is perfect for this application, because more than one suggestion is required. The fact that it uses more memory does not weigh up against the advantages.

It was expected that the data that was provided by Kramp would limit the options that were listed, because there was a chance the right data was not available. Luckily this was not the case.

Chapter 3

Research into did-you-mean

3.1 General idea

A did-you-mean function is meant to help the user after they have already searched for something. After pressing enter or clicking the search button there may be very little to no results for the search string, because it could be misspelled for example. A did-you-mean functionality then gives one or more suggestions to help the user improve their search string. A good example is Google's did-you-mean: they use the did-you-mean to help the user after their search, by suggesting corrections for possible spelling mistakes.

This differs from the auto complete functionality. First off, the auto complete suggests search strings before the user actually presses the search button. Additionally, it gives suggestions that will help improve the users search speed by suggesting popular search strings while the did-you-mean helps the user improve his search by correcting spelling mistakes and the sort.

The algorithm should look for the word that the user most likely meant after searching for something with no results. There are several options to determine what would classify as most correct search query, which will be chosen later.

3.2 Definitions

To help with the understanding of certain data structures, two definitions will be defined here.

The Levenshtein distance is the minimum amount of deletions, insertions and substitutions to change one word to another [9]. This is also known as the edit distance. For example, the Levenshtein distance between 'ofxd' and 'food' is 3: an 'f' is inserted at the start, the second 'f' is substituted with an 'o' and the 'x' is deleted.

An n-gram is an n-character long slice of a longer string w [10]. For example, the 3-grams of the word 'text' are _TE, TEX, EXT, XT_ and T__. The underscores (_) represent blanks.

3.3 Algorithm options

As with the auto complete the main difference between did-you-mean options is the data structure used by the algorithm. To create a good did-you-mean functionality it is important that a lot of data is stored, containing a list of existing words or often used search strings. Of course

storing all this information costs a lot of space and searching through this data will take increasingly more time as the amount of data grows. Making sure that the data is stored as efficiently as possible separates the good algorithms from the bad ones. In this first section we will go through some of these algorithms that were considered for this project.

3.3.1 Associative array

This is one of the most straightforward ways of making a spell corrector. The search string is compared to a list of known words and the Levenshtein distances between the string and the words in the list are calculated. For words with an equal Levenshtein distance, probability is used to determine which word is suggested (or suggested first).

An associative array (also known as a map), is a list of *(key, value)* pairs. This way search queries can be mapped to their 1-grams, 2-grams and so forth. The search string's 1-grams will be calculated and searched for in the map. If this returns nothing useful, try again with the search string's 2-grams, and so forth, until something valid returns.

This algorithm is also the basis for a lot of other algorithms that will be discussed next. The concept of checking the distance between the search string and a collection of words and using probability to determine the best is used by a lot of algorithms. However, the way these collections of words are stored, created and searched differs much.

Advantages

- Easy to implement

Disadvantages

- Takes a lot of space
- May take a long to find something relevant

3.3.2 BK-tree

A BK-tree[5] a tree created from the current collection of search strings creating a kind of dictionary. This is done in the following way. Choose a random word as root of this tree. Then for each other word in the dictionary look at the Levenshtein distance of the word to the root and go over the edge that has the same distance, where each edge corresponds to the Levenshtein distance between the connected nodes. Recurse over all the child nodes, until there is no edge for the calculated distance. Finally that will be the place of this word.

When querying the tree with a certain error range n. Calculate the Levenshtein distance d between the current root and the search string, if it is smaller than or equal to n add this term to your output list and continue querying going through the child notes on the edges that are in the range of d - n to d + n.

An example of this can be found in Figure B.1.

Advantages

- Small storage space
- Can be queried efficiently

Disadvantages

- Pre calculation takes long
- Loses a lot of efficiency as the error range grows

3.3.3 Levenshtein automata

This has another way of storing the search dictionary. The dictionary can be stored as a trie or as a deterministic finite automaton (DFA), but it is queried as if it were a DFA either way.[4]

The first step is to convert the search string into a nondeterministic finite automaton (NFA). This NFA then should be converted to a DFA. There are a few different algorithms that can accomplish this, but they will not be discussed here.

A dictionary stored as a trie can be seen as a special case of a DFA. Querying this can be done by intersecting the dictionary with the DFA created from the search string. This means that both DFA's are followed only traversing edges that both DFA's have in common. Every time both automata are in a final state, the word in the dictionary is put into the result set.

A second option is storing the dictionary as a list and then, after converting the search string to a DFA, the Levenshtein distance calculation can be done very efficiently by traversing the DFA for each word in the list.

Advantages

- Can be queried efficiently (more so than BK-trees)
- Loses less efficiency as dictionary grows and error range grows

Disadvantages

- Hard to implement
- Lots of preprocessing
- Going from NFA to DFA can be very inefficient

3.3.4 Using string length to partition the dataset

This is a fairly straightforward way to significantly improve the search speed. For this algorithm the idea is that a list of search queries is kept as a dictionary. However, instead of going through the entire list looking for words similar to the search string, only the words in the range of the edit distance around the length of the search string are queried. This already highly reduces the number of words to check.

Furthermore, since many words in any language are fairly similar, not every combination of letters have to be checked. However, trees such as the BK-tree are less efficient, since they will query all these combination. Therefore, if the dataset would be partitioned, it means that algorithms can get an answer quicker than if the full dataset would be used.

Advantages

- Fairly efficient query especially when lists are shorter
- Very little preprocessing

Disadvantages

- Loses a lot of efficiency as the dictionary grows
- Takes a lot of space

3.3.5 Winnow based spell checker algorithm

Another popular spelling correction algorithm is the Winnow algorithm[7], which is used by Microsoft Office 2007 for example. This algorithm is based solely on machine learning.

Instead of using a fixed dictionary, statistics are used to decide which is the most probable word. This algorithm can even be trained to perform contextual spell checking. When someone writes ‘to big’ this spell checker could correct this to ‘too big’. However, this is not very useful for this project.

Advantages

- Will become increasingly more efficient over time
- Very little preprocessing

Disadvantages

- Inefficient querying
- Bad space complexity
- Has unnecessary functionalities

3.3.6 Other possible trees

Alongside the previously discussed BK-trees, there are a few other trees that make for a good data structure for the did-you-mean functionality.

The vantage point tree (VPT)[8] is a tree that just like the BK-tree chooses a random node as a root. However, instead of creating edges for every Levenshtein distance, every node only has two nodes separating the words in two equal parts. The 50% closest to the node go to the first edge and the other 50% goes over the second node. This will create a binary tree. Querying this tree will consist of checking the current node with the search string and going down the corresponding edge, the first edge, when it’s closer than 50% and the second edge when it’s not. Every node found within the error range will be added to the output set. This tree could also be implemented as a tertiary tree having three edges for each node instead of two.

The bi-sector tree picks two words for each node. The tree is again separated in two child segments. The first child node will consist of all words closer to the first word of the parent node and the second subtree will have the words closer to the second word. Querying a node will mean going down the subtree of the word closer to the search string. However, if both words are in the error range of the search string, both subtrees are queried.

3.4 Relevancy options

The second aspect of the did-you-mean is giving a score to a word, which represents the likelihood a user meant to search for that word. There are multiple factors that can influence this score, for example the number of misspelled letters, but also the popularity of a certain word amongst users.

The main scoring function will consist of weighing different factors. Some possible factors are listed below.

3.4.1 Levenshtein distance

The Levenshtein distance is the amount of changes it takes to get from one word to another. This will most likely be the main factor in deciding the result. After all, one spelling mistake is a lot more likely to happen than four spelling mistakes.

3.4.2 Likelihood of search strings

After computing a set of results that the user could possibly mean, the likelihood of all the results should be computed. In other words, searches that have been used will get a higher score than searches that have barely been searched for. This factor likelihood will also change throughout the years. For example, a word like 'Whatsapp' would not have been given as a suggestion 10 years ago, but will now, at least by all major search engines, be given as a suggestion when someone types 'Watsapp'.

3.4.3 Search results per search string

The last relevancy option is search results per search string. Strings with more results would get a slightly higher score than strings with a lower amount of results. However, amount of search results does not necessarily correlate to a higher probability of a user wanting to search for that term.

3.5 Data

For both the auto complete and did-you-mean data is needed to fill the data structures. The auto complete should give suggestions for search strings. These suggestions should be based on previously used search queries that resulted in clicks. On the other hand the did-you-mean function should give alternative search strings for instance when the used search string was misspelt, therefore data from the product database should be used.

3.6 Conclusion

3.6.1 Data structure

The main decision in data structure will be choosing between BK-trees and Levenshtein automata. These two offer the most efficient way of computing relevant search queries at runtime.

The most important aspect is the speed of obtaining correct relevant search queries. This is to ensure that the user gets the best experience possible.

Both data structures share the same downsides, namely the time it takes to first build up the structure (preprocessing time). Yet BK-trees and Levenshtein automata will still be the best choice, because of the reasons mentioned above.

To make the final choice, both structures will have to be implemented and tested, as it is not certain which one will behave the best. This will be done after the end of the designing phase. Different values will be looked at to determine the best choice for the did-you-mean function. These values will be:

- Speed of querying
- Time to create initial structure
- Amount of space needed to hold structure

After looking at results, a final decision will be made as to which data structure will be used for the did-you-mean function.

3.6.2 Relevancy options

Combining all aforementioned options will give a useful score to search strings similar to the entered search string. The term with the highest score will be presented to the user.

The most important aspect will be the Levenshtein distance. This is the most objective and non-statistical way to calculate words that are closest related to a searched term.

After this, the likelihood of a search will impact the score of a search query. However, this will weigh less than the Levenshtein distance. This is because this should largely be used to differentiate between similar search strings that have the same Levenshtein distance to the original search string.

Lastly, it has been decided that the option of number of search results affecting the score will not be realized. This decision was made because of the reason stated in the discussion of the relevancy options, namely that the amount of search results does not necessarily relate to a better search query.

Chapter 4

Design auto complete

4.1 Algorithms

For the auto complete implementations, several algorithms were designed to get suggestions within acceptable time-constraints (within milliseconds).

algorithm 1 *addOrIncrementWord* describes how the tree will be build. As this algorithm traverses down the tree for every single query, this algorithm executes with a time complexity linear in $|D|$, where D is the list of queries to be included in the tree. Note that this algorithm can also be used to increment (as its title implies) the weight of a leaf. This means that it's possible to update the tree in real-time

algorithm 2 *setWeight* sets the weight of a query and, if necessary, increases the weight of its parent.

algorithm 3 *maxNode* recursively determines the string corresponding to the maximal weighed keyword, using a given set of nodes to search in. The algorithm searches for the node with the maximum weight in the given sets and repeats the same process using its children as nodes to search in. This process goes on until a leaf has been reached after which the corresponding string value will be returned. When assuming the number the children each node has during the recursion steps stays constant, the algorithm will execute with a time complexity linear in the size of the resulted word. In reality, the number of children tend to decrease when traversing the tree, which means that the time complexity could be lower than mentioned before.

algorithm 4 *excludeKeyword* returns a list of sub trees of the given node that do not contain an already found suggestion. From this list the next suggestion can be retrieved.

algorithm 5 *getTopKeywords* is the most important algorithm. This is called to return a list of suggestions given a prefix that has already been typed by the user. It maintains a list of nodes available to search in (initially the children of the root element). After a keyword with the biggest weight has been found using algorithm 3, algorithm 4 is used to filter out that keyword. The algorithm terminates when the desired amount of keywords has been found, or when no more leafs could be found in the tree with the given prefix.

4.2 Class diagram

To develop the auto complete a class diagram was designed. This class diagram can be found in Figure E.1 and is the class diagram that will be implemented in order to develop the auto complete. The layout of the classes as seen in the class diagram will be discussed in this section. In general a model-view-controller pattern is used.

First of all, a simple GUI is needed as a main entry point. This is because the auto complete has to be updated with every letter that is typed. This is not doable in a TUI or by giving the program an argument when it is started. This GUI class will be the class that contains the 'main' function so this is where the program is started. The GUI class also contains the auto- complete class. Of course, the GUI class also contains the necessary functions to show and use a GUI.

The auto completer class is the main class for the auto complete. This is the class where other classes can ask for completion suggestions. This class asks for data from the database control and commands the tree control to make, update or find possible completions in its tree.

For the database control an interface is designed made so that later on it will be no problem if a different kind of database will be used. The function for calculating the weight of each search query is already in the interface class. Besides that, the only thing the database control has to do is to return a list of search queries with an appropriate weight.

Also connected to the auto completer class is the tree control. The tree control contains the actual tree in which the data is stored. This class contains most of the algorithms: It makes, edits and searches in the tree.

Finally there is the tree. How the tree works is explained in section subsection 2.4.3.

There will be a few more classes, mostly utility classes, which will be used to help the classes depicted in the class diagram. These classes are not relevant for the bigger picture for which the class diagram is made, so they are left out.

Chapter 5

Design did-you-mean

5.1 Introduction

For the did-you-mean functionality two different algorithms will be implemented. The research concluded that both the use of BK-trees and the use of Levenshtein automata are efficient implementations of this algorithm. There is no clear winner when looking for the most efficient implementation. Therefore, both will be implemented and tested on the dataset to find the best algorithm for this did-you-mean functionality.

5.2 BK-tree

First, a tree with all search queries needs to be constructed from the dataset. This is done by algorithm 7. For this tree, a list of nodes that are within Levenshtein distance of a given word can be collected with algorithm 6. Lastly, a score is given to all nodes found by the algorithm. This is done via the formula

$$Score = \frac{weight}{LD^{ldWeight}}$$

where $weight$ is the same as described in section 2.3, LD is the Levenshtein distance between the search string and a node, and $ldWeight$ is how much the LD should influence the final score. The last variable is 6 by default.

Then the string with the highest score is returned.

5.3 Levenshtein automata

The research phase of the did-you-mean also resulted in the implementation of the did-you-mean functionality using Levenshtein automata. Further research concluded that there are multiple improvements possible on the concept explained there. Furthermore, there are two different implementations of this concept possible. The first implementation is more understandable, it is easier to explain and makes the concept very clear. The second implementation, however, manages to increase the efficiency and lower the calculation time per word by performing a series of pre-calculations.

First the concept will be explained using the first implementation and then the improvements of the second implementation will be explained.

5.3.1 Levenshtein automata using NFA to DFA conversion

The core concept of using Levenshtein automata for a did-you-mean functionality is to traverse two automata in parallel, which results in finding a correction without having to calculate the Levenshtein distance of every word in the dictionary.

The first automaton is actually a prefix tree, which is the same structure as used in the auto complete implementation. Note that this automaton is deterministic and finite, thus it qualifies as a DFA.

The second automaton is the Levenshtein automata. The most obvious form of such automaton is to make a grid with $n + 1$ rows and $w + 1$ columns, where n is the maximal allowed Levenshtein distance and w the length of the word (see Figure C.1). For every primitive single-character edit (deletion, insertion and substitution), a transition is added to every node. Note that this automaton is finite but non-deterministic and thus qualifies as an NFA. When a user searches for a word, this word is constructed into a NFA with a certain error range using algorithm 8. This error range is the maximum allowed Levenshtein distance between the search string and the words in the dictionary. The accepting language of this NFA, therefore, are all words inside the error range of the search string.

However, this NFA cannot yet be used to traverse both automata in parallel efficiently. As an NFA is non-deterministic, multiple transitions are possible when applying a single character to a state. Therefore, the NFA is translated to a DFA.

An NFA can be turned into an DFA using power set construction. However, as the states in the resulting DFA correspond to a set of states in the original NFA, the resulting DFA may have up to 2^n states, where n is the amount of states in the NFA. For example: the NFA for a word of length 9 and maximal distance 3 has 40 states which may result in a DFA with up to 2^{40} states.

Now there are two DFAs, one containing the dictionary as a tree and a Levenshtein automaton of a word and a particular error range. These DFAs can be intersected, by following the transitions that both Automata have in common. When both DFAs are in an accepting state, a word is found in the dictionary that is inside the range of the search string. After finding a possible word, or reaching a dead end, the algorithm backtracks to the past few states to find a possible other transition that leads to a valid word. This is also described in algorithm 9.

Finally, this will result in a list of words that are all inside the Levenshtein distance of the search string. A score mechanism using both the Levenshtein distance and the popularity of the search queries is used to sort this list on relevancy and the top two or three words are returned as a did-you-mean for the user.

5.3.2 Improvements on Levenshtein automata

As explained above, the conversion from an NFA to a DFA can be extremely time and space consuming. However, there exists a method to directly construct a Levenshtein DFA in linear time in $|W|$ for a word W[2]. This method reduces the amount of states by leaving out the states that are redundant, leaving only a fraction of the states possible. Furthermore, states that are identical in a way that they can overlap each other by shifting their corresponding NFA-states left or right can be reduced to a single parametric description. This reduces the amount of states to be considered to a parametric list of states. The amount of these 'parametric states' is independent of the word length. However, this amount grows quickly if the maximal Levenshtein distance is increased. In addition to the description of possible states, the transition function has to be described in the form of transition tables.

After the (relatively time-consuming) pre-calculations have been done, the Levenshtein automata for a word W can be constructed in linear time in $|W|$. However, this approach can be

improved further by skipping the actual construction of the complete DFA and instead simulating the Levenshtein DFA by determining next states on demand. This is possible as the pre-calculated transition tables yield the parametric descriptions of the transition function of the DFA (which determines it's structure).

5.4 Class diagram did-you-mean

Firstly, a few classes will be used that are shared with auto complete. These are the classes that communicate with the database (classes that implement IDBControl), and the GUI that controls everything.

The DidYouMean class is the main class for the did-you-mean part of this project. This class interacts with the GUI, database control and the general classes processing search data.

The class diagrams can be found in Figure E.3 and Figure E.4. The former is the initial class diagram that was designed, the latter is an updated version that corresponds to changes made during the implementation phase.

5.4.1 BK-trees

The BK-tree will only use two classes, namely a Node class and a BK-tree class. The BK-tree class is responsible for keeping track of the data structure and calculating the actual did-you-mean suggestion.

The BK-tree will use its own Node class to not interfere with any specifications set by the auto complete class.

5.4.2 Levenshtein automata

The two main classes of this part are LevenshteinAutomata and LevenshteinAutomataFactory. The LevenshteinAutomata is the overlapping static class, which is mainly used to intersect the DFA from the database's search queries and the DFA from a search that just has been made.

The LevenshteinAutomataFactory contains a lot of the algorithms explained in section 5.3. All the necessary utility classes are nested in this class: initial state, accepting states and transition table. This class also has the list of all the parametric states, criteria when these states are accepting, and finally the transition table.

Chapter 6

Design choices

6.1 Designing different implementations of the same functionality

It was decided that for both functionalities two different algorithms would be implemented. This eventually didn't happen, because the Auto complete research showed that there was only one real possibility: a tree. There were several kinds of trees that could be used so the most efficient one was chosen.

The idea behind this design choice was to be able to make a well substantiated and unbiased decision. Also in case that one of the chosen algorithms would not result in a working implementation, the project could continue with the other one.

Not all design choices will be discussed here as some of them are discussed in other chapters.

6.2 No user-specific search

A user-specific completion functionality was not implemented. Kramp did not have user specific information stored for their customers, therefore this could not be implemented yet. However, Kramp could still use the design for user-specific search by using the existing algorithms with user specific datasets. This is further explained in section 10.4.

6.3 Fill auto complete with did-you-mean results

A decision had to be made for when the auto complete would not result in the a full list of suggestions. There were several possibilities. First off, The results that were found could be shown, which could be no results at all. The second option was appending the auto complete suggestions with the results of the did-you-mean algorithm applied on the search string. The third and final option was to use the auto complete to get a list of completions, and if this list is too short, to use the did-you-mean to get one suggestion on which the auto complete is used on again.

We tested all three options. The first one was of course the easiest, but the conclusion was that some additional results would improve the system, thus something extra was preferred. Then the second result was implemented, this gave very some undesired results, for example

after typing the search string ‘addu’, which most likely means ‘accu’, a suggestion was made for led and for radio.

The third option, applying the auto complete on the search string and then applying the auto complete again, gave the best results. Multiple search strings were tested, and good results came out. Therefore, this became the actual implementation.

Chapter 7

Implementation

This chapter will discuss the final implementation of the did-you-mean and auto complete functions. This will mainly contain differences (if any) from the initial design, and general information about how the design was used.

7.1 Auto complete

The auto complete has been implemented fully as described in chapter 4. There were a few adjustments in terms of class management, this can be seen in Figure E.2. The TreeController was deemed useless, and therefore removed. All algorithms were moved to a specific class Algorithm. Lastly, the trie has been constructed in such a way that the correctness of the trie is guaranteed when the publicly exposed methods are used to mutate or query the trie.

7.2 Did-you-mean

7.2.1 BK-tree

The BK-tree is implemented as described in chapter 5 without making any changes. Because of this, the tree part in the initial diagram (Figure E.3) and the finalized diagram (Figure E.4) are equal, aside from some obsolete methods that were deleted in the BKTree class. The main algorithms, aside from *addChild*, are put in the BKTree class. Everywhere where arguments may be invalid, exceptions are thrown if these arguments are invalid. This is to guarantee that the behaviour of the tree stays as expected.

7.2.2 Levenshtein automata

The Levenshtein automata is, unlike BK-tree, vastly different from the initial class diagram. This is because after making the class diagram, a more efficient algorithm was found. This is not documented in the research part of did-you-mean, but is documented in section 5.3. This is because the algorithm was found during the implementation phase, a few weeks after the researching phase. The two methods look similar, which is why the method which was later found was thought of as the same. After further inspection during the implementation phase, however,

it was found that the algorithm that is currently implemented is more efficient in both space and time, while being a lot harder to implement. This new algorithm was explained in section 5.3.

Chapter 8

Testing

8.1 Test plan

8.1.1 Introduction

The tests performed on the project can be separated into three different types. Of course first off, the implementation itself has to be tested for correct functionality. It is important to know if the written code is working as expected and to check if rare unexpected cases could harm the product. The second test is measuring improvements. This is about if the client is satisfied with our product. Kramp said they cared most about measurable facts, when it comes to proving improvement. This should be measured in standards that are often used for comparing search algorithms. The third type of testing will be user experience testing. This will be explained in more detail below.

8.1.2 Implementation tests

First the implementation itself will be tested. This means testing if the code is work as expected and testing and if there are no unexpected exceptions or problems when it comes to unexpected cases. For example what would happen when a huge string is entered as search string. These tests will be implemented as unit tests using JUnit. Important aspects here are coverage of the code and the handling of unlikely cases with possible malicious intent. A detailed implementation test plan with the functionalities that will be tested and the expected results of these tests can be found in appendix G.1.

8.1.3 Measuring improvement

The second test is based on the project being compared to Kramp's old search functionalities. This test is about if both the functionalities indeed increase the search efficiency compared to the current situation. This test is about measuring the results of the project. This could be done using the mean reciprocal rank, which is a measuring unit for the correctness of responses to a query. For the auto complete this means looking at the i-th result that corresponds to the search string the user had in mind, which will increase when more characters are typed. For did-you-mean it is also calculated using the i-th suggestion of the did-you-mean that turned out to be correct.

8.1.4 User experience testing

Finally some user experience tests may be performed. Customers of Kramp may be asked to test the prototype. The customer will be asked to search some of the products they would usually also search for in the Kramp webshop. Then improvement will be measured using an interview or a questionnaire to measure their satisfaction with the new functionalities and a measurement of the time that different searches take using the old and new functionalities.

8.2 Evaluation

8.2.1 Results of implementation tests

For the implementation, unit tests were used to test the most amount of code possible. While testing, it was apparent that the code was not fully robust and had some bugs in it. For example, the Levenshtein distance calculator would, if the first word was smaller than the second, return a negative distance. This and other bugs have all been fixed. All the unit test used can be seen again in the test plan in appendix G.1. The final row of the table is the name of the class were the dicussed test is stored. This was added after implementation of the tests. Finally, after fixing the bugs that were mentioned before, all the test mentioned in the testplan were passed and the actual result was the same as the expected result mentioned. While running all the unit tests, the line coverage of the code is about 80%. This was deemed sufficient after looking at what was covered and what was not. The most important classes and parts of the code were namely covered for a hundred percent. The API and the trees are, for example, both hundred percent.

8.2.2 Acceptance test

The “measuring improvement” and “user experience testing” have been merged into one overlapping test, namely the acceptance test. This is the test which only succeeds if Kramp thinks the current implementation is acceptable. From talking with Kramp, it was apparent that there was no possibility to get a pure number as discussed above in “measuring improvement”. This is because there is not enough data to check results with.

However, Kramp internally has knowledge about what current problems are and what they would like to see fixed. A demo was made and sent to Kramp for review (see appendix H). This demo is a GUI with a search bar, and under that the list of auto complete suggestions. After hitting the search button or pressing enter, the did-you-mean result for the search string is displayed. Kramp was able to use this demo to judge whether it is acceptable or not. This last part is the user experience testing part.

Two weeks after the demo was sent, Kramp concluded that the implementation is acceptable. It was clear that it is mostly dependant on the data you put in; bad/little data results in bad auto complete and did-you-mean results, while good and high amounts of data results in good results.

8.2.3 User experience testing

User experience testing was done with the help of Kramp. They gave information about what feedback they got from their customers. However, eventually no real user tests were done. The information mentioned before was used as user experiences and this wat taken into account

in the design of our project. The main user complaints were that product numbers were not processed properly and that the auto complete functionality was fairly slow.

Chapter 9

Performance

When using the API it is important to know the time and space complexity of its sections. With knowledge about the number of API instances that will be active concurrently, one can make an estimation of how much resources are needed to host a server that uses it. Therefore a the complexity of the most important methods was calculated and a series of tests were conducted to determine the time and space needed in practice. These tests are conducted on a pc with an i5-4460 @ 3.20GHz CPU using a dataset with a size of 20,000.

9.1 Auto complete

The initialization of the auto complete has a time complexity of $O(nk)$ where n is the size of the dataset and k the average length of the word. In the test this resulted in a time of 20ms.

Getting suggestions from the auto complete has a time complexity $O(k(\log n - p))$ where, again, n is the size of the dataset and k is the average length of a word and p is the length of the given prefix. In the test this resulted in a time of 3ms.

The auto complete currently uses around 21MB when it is active using the given dataset.

9.2 Did-you-mean

9.2.1 BK-tree

Initializing the BK-tree for the did-you-mean has a time complexity of $O(k^2 n \log n)$ where n is the size of the dataset. In the tests this resulted in a time of 731ms.

The complexity of getting a suggestion from the BK-tree consists of two parts; First a list of possible suggestions is created, then the best one is selected. The complexity for creating the list is $O(nk^2)$ where n is the size of the data and k the average word length. The complexity for selecting the best suggestion from that list is $O(l \log l)$ where l is the size of the list.

The did-you-mean using BK-tree uses around 7 MB when it is active using the given dataset.

9.2.2 Levenshtein automata

The did-you-mean using a Levenshtein automata is a bit different, it initializes and uses a Levenshtein automata factory, which only has to be made once and can then be used by different

instances, even when they use different dataset, as long as the same maximum Levenshtein distance is used. This means that when it is used for more instance, the impact will become smaller. In the tests, this factory needed roughly 515ms to initialize and then took up 201MB of space using the given dataset. This was done with a maximum Levenshtein distance of 3.

Both initialization without factory as well as the search have the same complexity as the auto complete, because it uses the same tree. This also means it uses the same amount of space.

9.2.3 BK-tree versus Levenshtein automata

To give some leverage in a decision between using the BK-tree or using the Levenshtein automata, another performance test was done to get a better insight into the performance of both. For this test a query set of 16399 queries was used. This query set was generated by randomly deleting one or two letters from every word in the original dataset. Lastly, some queries were filtered out because they did not fit in the desired format anymore.

For each query in this set both the BK-tree and the Levenshtein automata method are timed. This resulted in the box-plot as seen in Figure I.1. The average time of the BK-tree is 3.35 ms, the Levenshtein automaton has an average of 4.50 ms. In the box plot mentioned before, it is visible that the BK-tree is very consistent and fast almost every time. The Levenshtein automaton, however, is much more spread out. The maxima of the BK-tree and the Levenshtein automaton are 18ms and 41ms respectively. While one is half of the other, both are still negligible time-wise.

While the BK-tree seems to be faster and more consistent, the Levenshtein automata has some considerable advantages space wise, especially when a large number of instances of the API are used. The BK-tree uses 7 MB for each instance while the Levenshtein automata can use the same tree as the auto completer which makes it effectively 0 MB in addition to the 201 MB for the Levenshtein automata factory of which one will suffice for all the API instances. This makes that, with the current dataset, when 29 or more instances of the API are active, the Levenshtein automata is more efficient. This is specific for this dataset, because the size of the Levenshtein automata factory is only dependent of the Levenshtein distance and not of the size of the dataset. When a larger dataset is used, less instances of the API are needed for the Levenshtein automata to be more efficient and vice versa.

Chapter 10

Integration

10.1 The API

The API consists of three main packages: a database controller (DBC), the auto- complete (AC) and the did-you-mean (DYM). At the top of these three packages there is a controller that contains all relevant methods of the API so that there is only one object needed to use the complete API. This is visible in Figure F.1 It is also possible to create the other classes and use them separately.

10.1.1 Database controller

The DBC provides data to the AC and the DYM. As many different ways of storing data exist, the API contains a DBC interface. The interface specifies a single method that returns data, all search queries with their score, and it has already defined a static method which calculates the weight for a keyword.

For basic usage one implementation of this interface is included in the API: CSV control. This implementation processes CSV files and returns the data from these files when requested.

10.1.2 The auto complete

The AC is used for suggesting completions for a given prefix. When created it requests data from the DBC. This data will be stored in a trie. This trie can be reset and also more data can be added. Furthermore, there is a method to request completions. The AC uses another class which contains all the algorithms as described in chapter 4.

10.1.3 The did-you-mean

The DYM has two methods of finding a suggestion. BK-tree is the default method, but a Levenshtein automata can also be used. The DYM also requests data from the DBC when it is created to build its BK-tree and the trie used for the Levenshtein automata. When the Levenshtein automaton is used, a DFA is emulated as well when a search is done.

10.1.4 The controller

This is the main class of the API. It contains the DBC, AC and DYM instances as well as most methods that can be used from outside the API. Furthermore, it contains an extra method, *getAdvancedTopN* that uses both AC and DYM to give a list of completions. This methods first uses the AC to get a list of completions and when this list is too short the DYM is used to get one suggestion for which also possible completions are retrieved using the AC. The *getAdvancedTopN* method should be used to get completions most of the time. In Figure H.3 you can see the *getAdvancedTopN* method in action.

This controller is made so that only one object is needed to access the whole API.

10.2 General use

First of all, one would want to create an implementation of the DBC interface. What this implementation will look like depends on what type of database is used. At first, one would just put the data of the last x months in the database. This data should contain at least the search queries and enough data to get an appropriate weight for each search query.

The API works with pre-build trees, the trees are directly constructed from the database when a Controller instance is created. However data can also be added to the trees using several API methods. Additionally, the data can be updated in the database and then one can let the API rebuild its trees. This will consume more time though.

While testing and implementing the API, data of three months was used. This data consisted of 20,000 search queries with their appropriate weights. Using 20,000 search queries satisfactory results were achieved. Moreover, while testing, all data from those three months was used, but it could be good to filter out the search queries that have a weight under a certain level. If there is a minimum weight limit that a search query has to have in order to get suggested, it will give no suggestions rather than the lower rated ones. This will help make sure that no undesirable suggestions will be given and improves the overall performance since the trees stay smaller.

10.3 Use per category and/or per user

If one wishes to use the API for category or user specific suggestions, one merely has to provide different data. For suggestions based purely on the users' history, the API only needs the search history of that user. However, it is probably better to also partially base the suggestions on the complete history. To do this a good ratio is needed between the weight of the complete history and the user history. The same goes for category specific suggestions.

Needless to say, not every user or category will result in the same data for the API. Therefore, one will need several instances of the API; One general instance, for users that are not logged on, or users that don't have a history yet. Besides that, an instance is needed for every user in order to give the right suggestions.

Our advice is to have the one instance of the API with the complete history and besides that have an instance for every user that is using the website. This can be done by making a new instance of the API once a user logs in and by disposing it when the user logs out. Category and/or user specific history should be collected in order to do this.

10.4 Possible improvements/additions

Although the implementation of AC and DYM was designed to be space- and time-efficient, several improvements and additions could be made to increase the performance of the implementations and the usefulness of the suggestions.

First of all, some small improvements should be made to make the whole API space- and time-efficient. Below is a list of relatively simple improvements that were discovered too far into the project to be actually implemented:

- AC and the Levenshtein implementation of DYM rely on the same tree structure. However, the API currently initializes two separate instances of this tree. To save initialization time and (most importantly) memory, the two instances should be replaced with a single instance that both AC and DYM can use.
- AC and the Levenshtein implementation of DYM have different implementations to do the same thing, namely to get the k highest ranked keyword in the tree. AC uses an implementation that consists of multiple methods while the Levenshtein implementation combines the functionality of these methods in a single method. The proposed improvement is to replace the methods in the AC class Algorithm with the method used in LevenshteinAutomata, as it keeps track of the the nodes to search in a smarter way.
- When currently initializing a DidYouMean object, both the Levenshtein and BK-tree implementations are initialized for testing purposes. When using the API in production, only the implementation that is actually used should be initialized, which should save memory and initialization time.

As a first addition, the implementations should also take the personal search history into account to make the suggestions personal. This would greatly improve the quality of the suggestions as different customers may be interested in different items and their search history is an indication for their interests.

In order to achieve personal suggestions, additional instances of the controller must be created: one for each active user. Instead of only getting suggestions from a single global controller, a personal controller could be used in addition to the global controller. By giving the personal results a specific weight over the non-personal results, the personal search history can influence the suggestions. As the search history of a single user will probably be very small compared to the global history, a new personal controller can be initiated once a user logs into the web-shop.

However, the current controller class is not optimized for being initialized multiple times. For example, an instance of a LevenshteinAutomataFactory is identical every time, given that the maximal Levenshtein distance remains equal. Therefore, this class only has to be initialized once (like a singleton class) after which all controllers can use the same instance. A better solution might be to let a single instance of a controller be able to contain multiple trees: a global tree and one for every active customer.

To give more advanced suggestions that bring a user from 'accu' to 'battery', more specific data should be stored. If a user unsuccessfully searches for 'accu' and right after that searches successfully for 'battery', that should be saved. When enough of this data is gathered, the did-you-mean function can be expanded to suggest these kind of changes, using a mapping from wrong and unhelpful search queries to their better and (mostly) synonymous alternatives.

Lastly, the API could take into account that different permutations of a search string consisting of multiple words yielding the same results in the search engine. Therefore the strings 'blue paint' and 'paint blue' could be merged into one element.

Chapter 11

Discussion & Conclusion

11.1 Discussion

There are some things concerning whole project process that could have been done better.

- Although a Trello scrumboard has been made in the beginning of the project, it has not been used throughout the different design phases. Instead of Trello, simple To-Do lists were used to divide tasks. In the main phases of the project (research, design, implementation), no task management tool was used. In the future, a tool dedicated to tracking scrum progress would be useful, as it makes it clear how fast a scrum group is progressing.
- During the implementation of did-you-mean, it was discovered that implementing Levenshtein automata had to be done very differently than was thought during the research and design phase. Therefore, the final implementation of Levenshtein automata does not correspond to the initial research and design. To improve this, the research of the Levenshtein automata should have been conducted more thorough. Additionally, make sure that seemingly similar algorithms are actually similar (as was assumed in the research phase).

11.2 Conclusion

11.2.1 Must

- ✓ The system must be able to use auto complete to give suggestions for a search string
- ✓ The system must be able to use did-you-mean to give a suggestion after a search string has been used
- ✓ The system must contain a simple GUI to show the proof of concept

All Musts have been finished, as they were required for Kramp to find the project acceptable. All information about design, implementation and possible integration can be found in the design report.

11.2.2 Should

These are the requirements that should be done at the end of the project, they are preferred, but not crucial for the project.

- ∼ The system should learn every time a user enters a search string
- ✓ The system should learn which search strings are successful by looking at previous searches
- ✓ The system should correct typing mistakes in both the did-you-mean and auto complete
- ✗ The system should link similar search strings and only suggest search strings that give the best results

Machine learning has not been directly implemented in the implementation of both algorithms. However the full intended use of both algorithms does fall into the category of machine learning. It is possible to use the method *learn* to update auto complete and did-you-mean instances without completely recreating them. The fourth requirement was not implemented, because Kramp did not have the data required for this.

11.2.3 Could

- ∼ The system could support fully personalized search functionality
- ✗ The system could link (mistyped) searches with previously bought items in a personalized way
- ✗ The system could keep track of single search words instead of entire search strings

Fully personalized search functionality is supported, but not implemented. More about this can be read in section 10.3.

Linking searches was not implemented, as there was not enough data. Only Google Analytics data was available, which does not specifically show which link is clicked after a search.

Keeping track of single words was deliberately not implemented, as it was a design choice. This was done because of the data provided had information about full search queries. This information was used to give a score to a specific search query. Breaking up this query will render this score useless, or at least less correct.

11.2.4 Won't

- ✗ The system won't fully implemented system in Kramp's webshop
- ✗ The system won't improve product information with product data
- ✗ The system won't improve entire search algorithm

These requirements were explicitly noted as 'will not do'. Therefore, these requirements have not been done. Fully implementing in the Kramp Webshop would be too complex. The last two requirements were simply not part of what the client wanted to see implemented.

Bibliography

[1] Black, Paul E. (2009-11-16). 'trie'. Dictionary of Algorithms and Data Structures. National Institute of Standards and Technology. Archived from the original on 2010-05-19. http://www.webcitation.org/5pqUULy24.

[2] Matani, Dhruv. (2011-09-02). An $O(k \log n)$ algorithm for prefix based ranked auto complete. http://www.dhruvbird.com/autocomplete.pdf (alternative source http://web.archive.org/web/20160316141448/http://dhruvbird.com/autocomplete.pdf).

[3] Efficient auto complete with a ternary search tree. (n.d.). Retrieved February 18, 2016, from http://igoro.com/archive/efficient-auto-complete-with-a-ternary-search-tree/ http://igoro.com/archive/efficient-auto-complete-with-a-ternary-search-tree/

[4] Johnson, Nick. Damn Cool Algorithms: Levenshtein Automata. Nicks Blog. 2016-03-16. URL:http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata. Accessed: 2016-03-16. (Archived by WebCite® at http://www.webcitation.org/6g38r0HFC)

[5] Johnson, Nick. BK-trees. http://blog.notdot.net/. 2016-04-06. URL: http://blog.notdot.net/2007/4/Damn-Cool-Algorithms-Part-1-BK-trees. Accessed: 2016-04-06. (Archived by WebCite® at http://www.webcitation.org/6gZ6LoWHK)

[6] Schulz, Klaus U & Mihov, Stoyan. Fast string correction with Levenshtein automata. URL:http://link.springer.com/article/10.1007%2Fs10032-002-0082-8

[7] Nick Littlestone (1989). "Mistake bounds and logarithmic linear-threshold learning algorithms". Technical report UCSC-CRL-89-11, University of California, Santa Cruz.

[8] Ada Wai-chee Fu. Dynamic VP-Tree Indexing for N-Nearest Neighbor Search Given Pair-Wise Distances http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.7401&rep=rep1&type=pdf)

[9] V.Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals",1966 SOL Phys Dokl.

[10] Cavnar, William B. and Trenkle, John M., "N-Gram-Based Text Categorization", P.O. Box 134001 Ann Arbor MI 48113-4001.

Appendix A

Example prefix tree

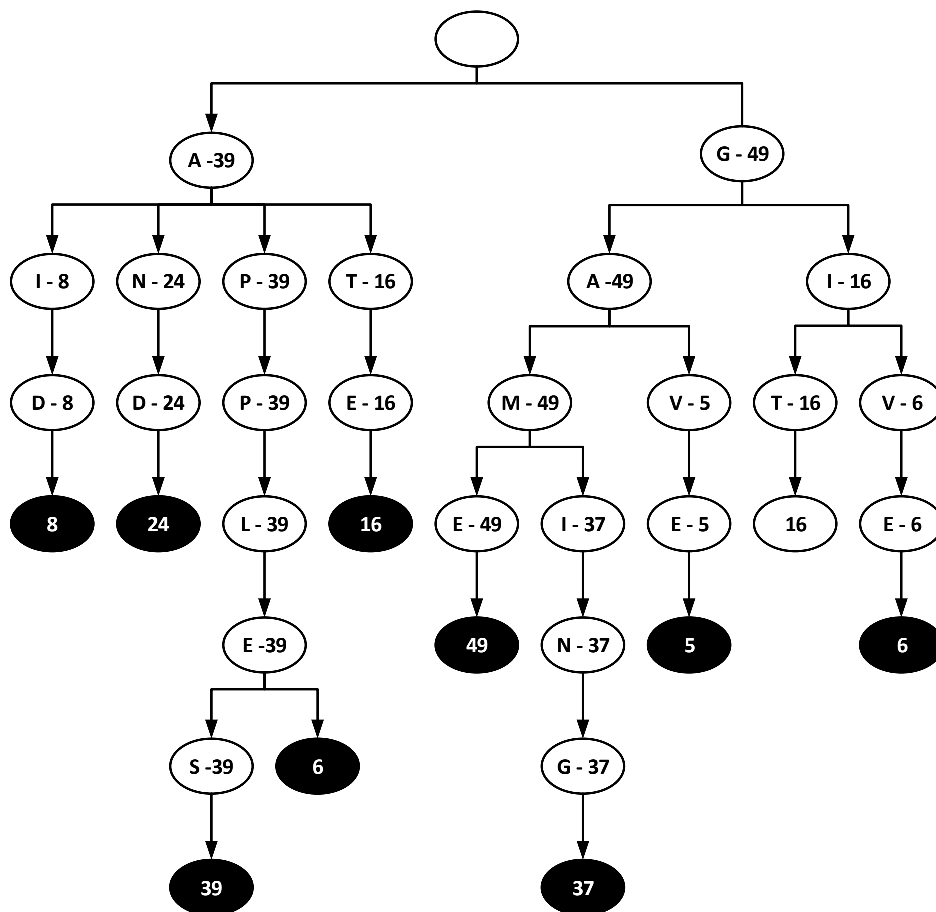

Figure A.1: This is a prefix tree in which to the following words and their scores are stored: [aid: 8, and: 24, apples: 39, apple: 6, ate: 15, game: 49, gaming: 37, gave: 5, git: 16, give: 5].

Appendix B

Example BK-tree

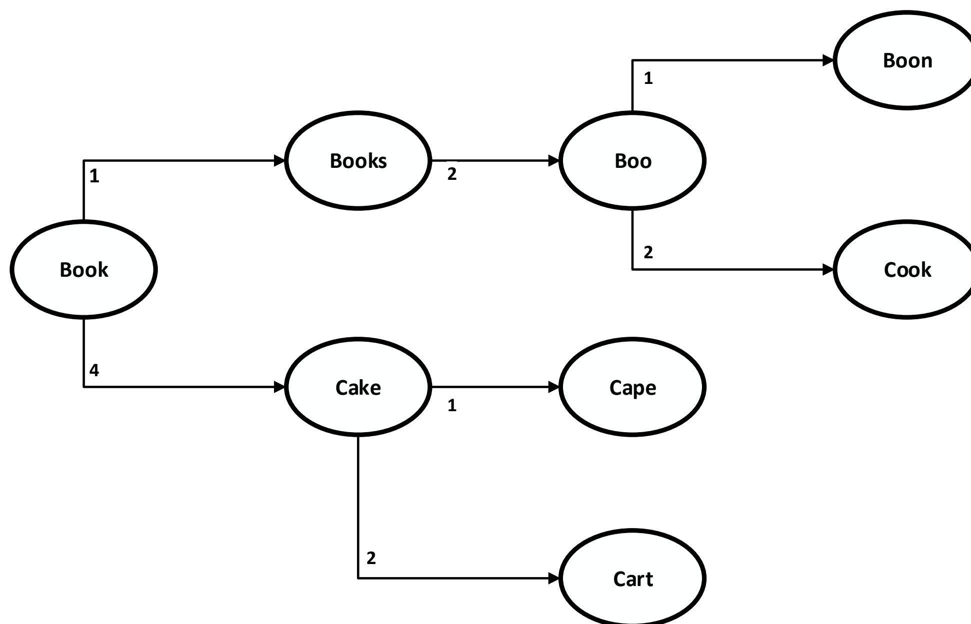

Figure B.1: An example BK-tree without weights.

Appendix C

Example Levenshtein NFA

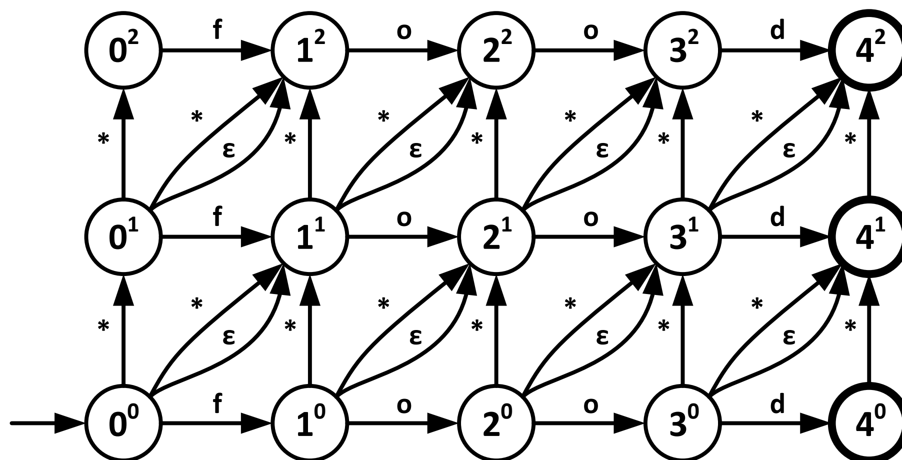

Figure C.1: An example Levenshtein NFA for the word ‘food’ and with a maximum Levenshtein distance of two.

Appendix D

Algorithms

D.1 Trie

```
1 Function addOrIncrementWord (k, d)
      Data: k: null-terminated keyword to search for, d: the weight to add
2     child = this.addOrGetChild(k[0]); if child.isLeaf() then
3         child.setWeight(child.weight + d);
4     else
5         child.addOrIncrement(k.tail(), w);
6     end
```

Algorithm 1: Add keyword or increment the weight

```
1 Function setWeight (w)
      Data: w: the new weight for this element
2     this.weight = w; if this.parent != null && this.parent.weight < w then
3         this.parent.setWeight(w);
4     end
```

Algorithm 2: Function to update the weight of the element

D.2 Auto complete

```
1  Function maxNode (ns)
       Data: ns: a list of nodes in which to look for the max-weighed keyword
       Result: A string corresponding to the max weighed keyword in the given nodes
2      maxWeight = -1;
3      maxN = null;
4      foreach n in ns do
5          if n.weight > maxWeight then
6              maxWeight = n.weight;
7              maxN = n;
8          end
9      end
10     if maxN.isLeaf() then
11         return maxN.word;
12     else
13         return maxNode(maxN.subNodes);
14     end
```

Algorithm 3: Get max-weighed keyword

```
1  Function excludeKeyword (n, k)
       Data: n: the node to exclude the keyword from, k: the keyword to exclude, p: the prefix
             up until node n
       Result: A list of tuples with nodes containing all keywords in n excluding k and the
               corresponding prefixes
2      result = [];
3      foreach child in n.subNodes do
4          if child.letter == k[0] then
5              result.extend(excludeKeyword(child, k.tail());
6          else
7              result.append(child);
8          end
9      end
10     return result;
```

Algorithm 4: Exclude a keyword out of a node

```
1  Function getTopKeywords (n, k, p)
     Data: n: the root node, k: the amount of keywords to return, p: the given prefix
     Result: A list of the top-k keywords, if any exist
2    result = [];
3    n = searchNode(n, p);
4    if n == null then
5        return result;
6    end
7    searchNodes = [n.subNodes];
8    i = 0;
9    while i++ < k && !isEmpty(searchNodes) do
10       keyword = maxNode(searchNodes);
11       result.append(keyword);
12       foreach n in searchNodes do
13           word = n.word;
14           if keyword.startsWith(word) then
15               searchNodes.remove(n);
16               searchNodes.extend(excludeKeyword(n, keyword.removeFirst(word)));
17               break;
18           end
19       end
20   end
21   return result;
```

Algorithm 5: Exclude a keyword out of a node

D.3 Did-you-mean

D.3.1 BK-tree

```
1  Function searchTree(root,string, range)
       Data: root: root node of the search tree, string: the searchstring of the user, range: the
             maximum allowed Levenshtein distance
       Result: List of all nodes in the tree with a word inside the Levenshtein distance range
               of the search string
       /* result is a list in which the resulting words will be saved.       */
2      result = [];
       /* This method calculates the Levenshtein distance between the word of
          the current node and the search string.                            */
3      n = calculateLevenshteinDistance(root.word, string);
4      if n < range then
           /* If this is inside the error range then this word should be in the
              result list.                                                   */
5          result.add(root);
6      end
       /* Check all the outgoing edges of the current node for a child inside
          the error range of the word                                        */
7      foreach child in root.children do
8          if abs(root.child.distanceToParent-n) < range then
               /* Recurse this function on all the child nodes inside that range.
                  */
9              SearchTree(root.child,string,range);
10         end
11     end
       /* If there are no child nodes left return the result list.           */
12     return result;
```

Algorithm 6: Search for all search queries within a certain error range

```
1  Function buildTree(list)
       Data: list: list of all words that will be in the tree, node: root of the resulting tree,
             current: current element in the list
       Result: Root of the tree as object
       /* Assign the first word of the list to the root node                   */
2      root.word = list[0];
       /* Then for each word in the list add the word to the tree (seperate
          function)                                                              */
3      foreach word in list do
4          addWord(root, word);
5      end
       /* Return the root node                                                   */
6      return root;
7  Function addWord(root,word)
       Data: root: Currently inspected note in the tree, word: inputword to be added in the tree
       /* Calculate the distance between the word, that is currently being
          processed and the root node.                                          */
8      n = calculateLevenshsteinDistance(root.word, word);
       // Check if there is already a child with that distance, if not add this
          word as child node.
9      if root.getChildWithDisance(n) == null then
10         root.addChild(word,n);
11     end
       /* If there is one, recurse this function on the child node.             */
12     else
13         addWord(root.getChildWithDistance(n),word);
14     end
```

Algorithm 7: Creating a tree from a list of words

D.3.2 Levenshtein automata

```
1  Function makeNFAFromWord(string, errorRange)
       Data: string: search string to be converted, errorRange, the allowed error range of this
             string
       Result: the nfa of the search string using with allowing token words inside the error
               range
       /* New NFA with start state 0,0                                                   */
2      nfa = new NFA(0,0);
3      int i = 0;
       /* For each letter in the word there are 3 transitions to be added.  One
          to the left for when the letter of the input word corresponds to the
          word of the NFA one straight up, for when there is a letter in front
          of that word (insertion) and one diagonal for a missing letter
          (deletion)                                                                     */
4      foreach letter in string do
5          for int e, e < errorRange, e++ do
6              //correct letter
                nfa.addTransition((i,e), letter, (i+1, e));
7              //insertion
                nfa.addTransition((i,e), any,(i, e+1));
8              //deletion
                nfa.addTransition((i,e), epsilon, (i+1, e+1));
9              //substitution
                nfa.addTransition((i,e), any, (i+1, e+1));
10             i++;
11         end
12     end
       /* Add final states and transition for when a letter is exchanged for a
          random letter in the input word (substitution).                                */
13     foreach j in range errorRange do
14         if j < errorRange + 1 then
15             nfa.addTransition((len(string),e), any, (len(string),e+1));
16         end
17         nfa.addFinal_State((len(string),e));
18     end
19     return nfa;
```

Algorithm 8: Creating an NFA from a word

```
1  Function findNextValidString(DFA, input)
       Data: DFA: dfa to be inspected, input: input word used as token word for the DFA
       Result: if the word is in the dictionary: the input itself otherwise a valid word
2      state = DFA.startState;
3      stack = [ ];
4      for i, x in enumerate(input) do
           /* Save current state in stack                                          */
5          stack.append((input[0..i], state, x));
6          state = DFA.nextState(state, x);
7          if not state then
8              break;
9          end
10         else
11             stack.append((input[0..i+1],state,None))
12         end
13     end
14     if DFA.isFinal(state) then
           /* Word is in the dictionary, no did-you-mean necessary                 */
15         return input;
16     end
17     while stack.isNotEmpty() do
18         path, state, x = stack.pop();
19         x = DFA.findNextEdge(state,x).word;
20         if x != null then
21             path += x;
22             state = DFA.nextState(state, x);
23             if DFA.isFinal(state) then
24                 return path;
25             end
26             stack.append((path, state, none));
27         end
28     end
29     return none;
```

Algorithm 9: Retrieving the word most likely meant by the user

```
1  Function findNextEdge(DFA,s,x)
       Data: DFA: The dfa that is currently being traversed, s: the current state of the DFA, x
             the current path to that state
       Result: The word with tokens to the lexicographically closest outgoing edge
2      if x is None then
3          x = 0;
4      end
5      else
6          x = unichr(ord(x)+1);
7      end
8      stateTransitions=DFA.Transition.get(s, ,);
9      if x in stateTransitions or s in DFA.defaults then
10         return x;
11     end
12     labels = sorted(stateTransitionKeys());
13     pos = bisect.bisectLeft(labels, x);
14     if pos < len(labels) then
15         return labels[pos];
16     end
17     return None;
```

Algorithm 10: Retrieving the word closest to a given state

Appendix E

Class diagrams

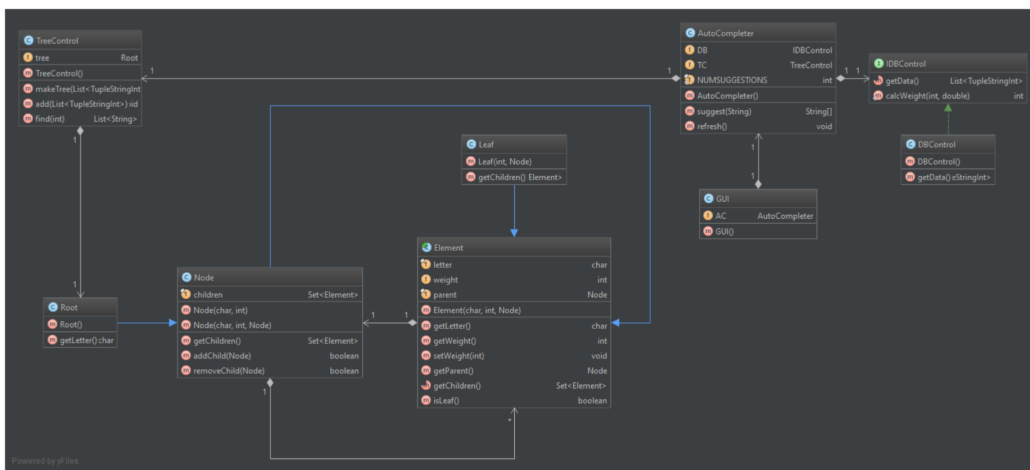

Figure E.1: This is the class diagram as designed during the design phase for the auto complete.

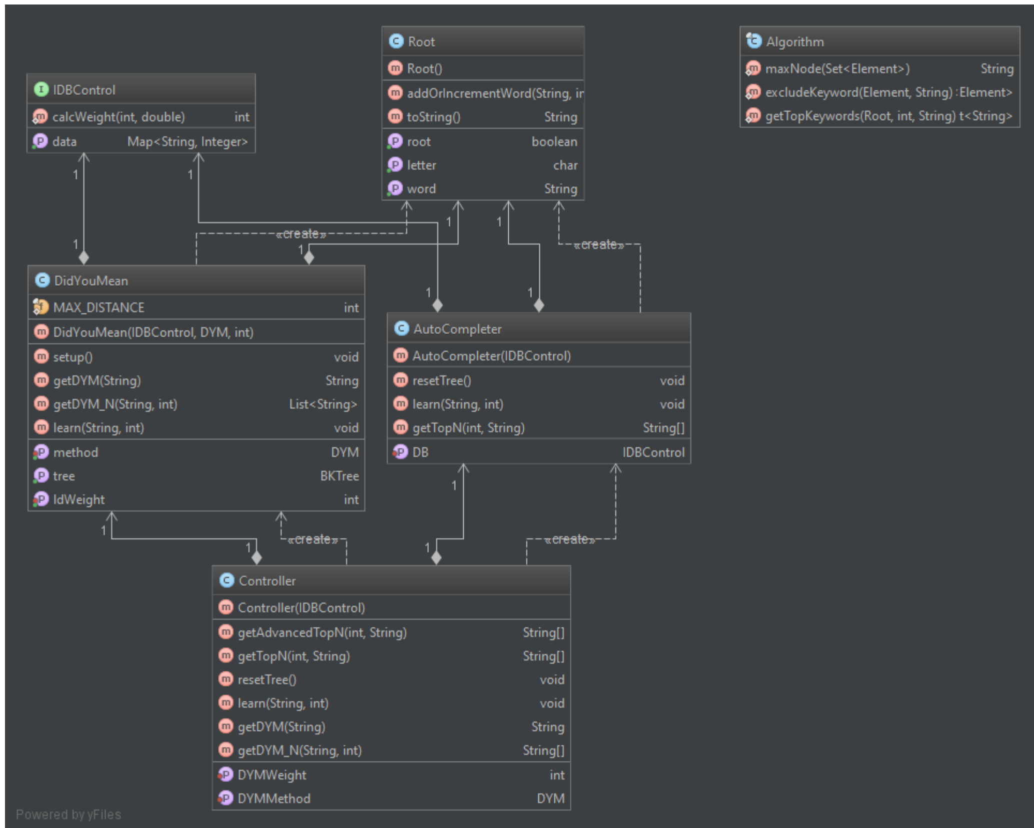

Figure E.2: This is the class diagram of the auto complete after implementation.

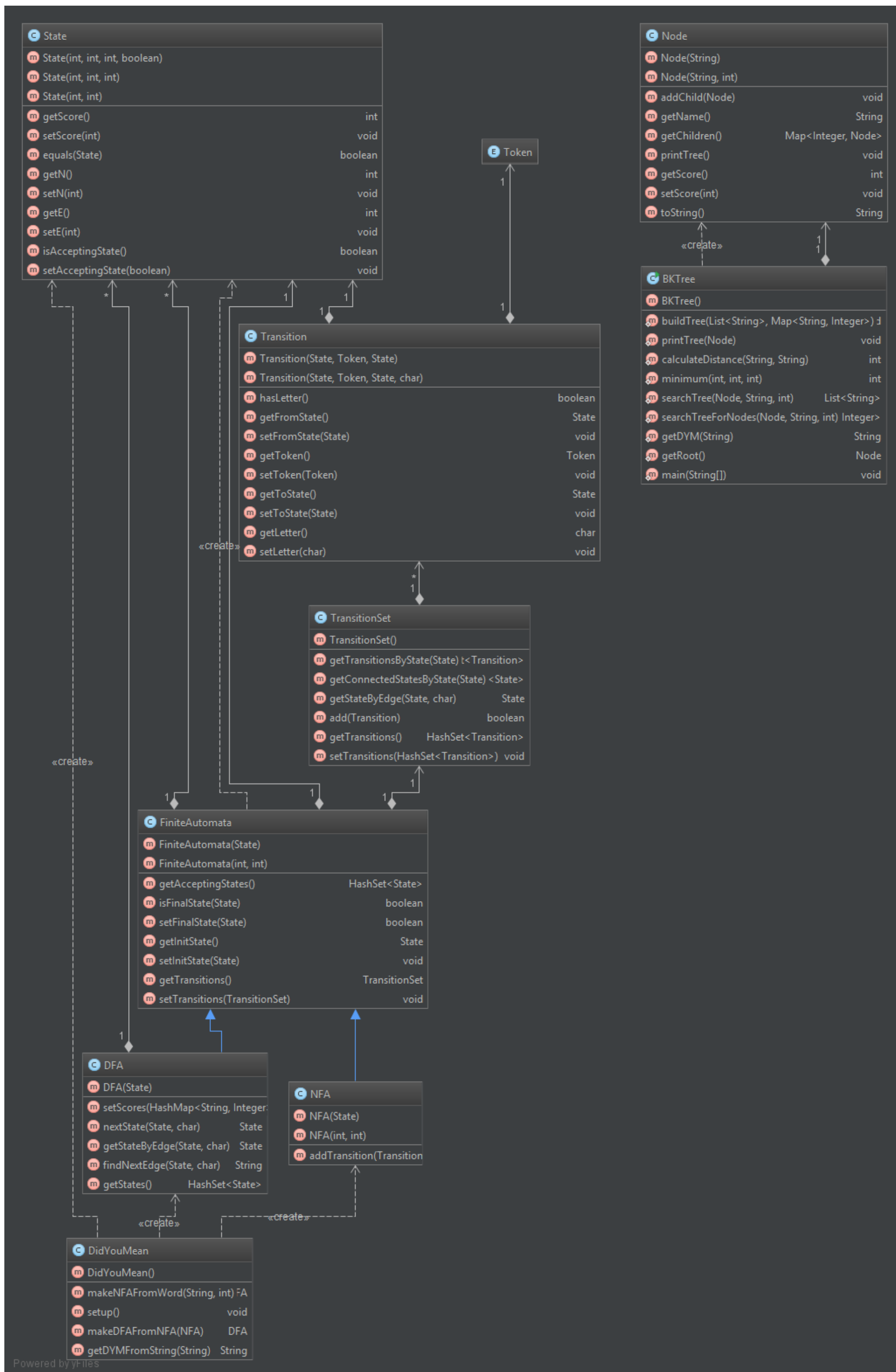

Figure E.3: This is the class diagram as designed during the design phase for the did-you-mean functionality.

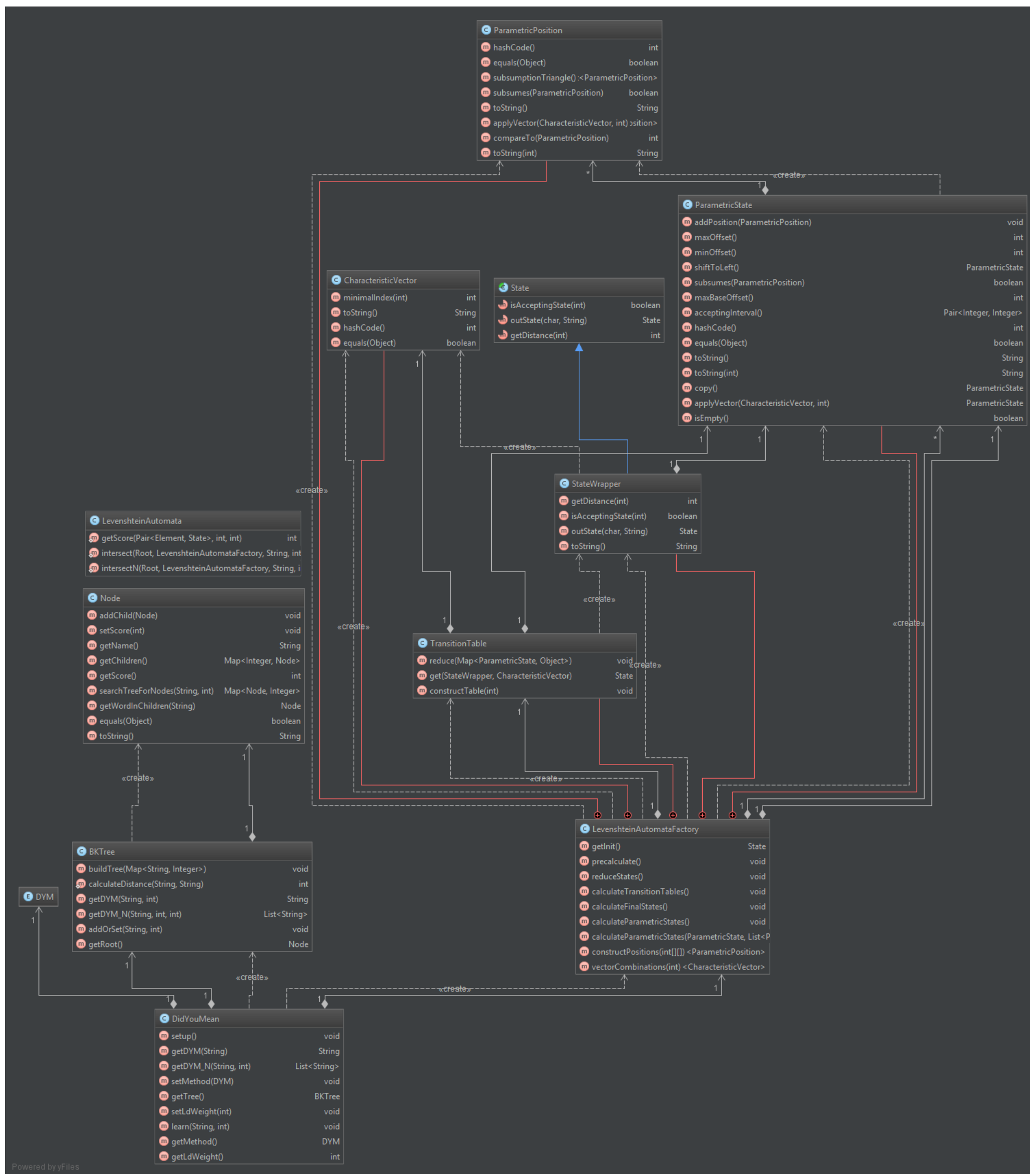

Figure E.4: This is the final version of the class diagram corresponding to the implemented did-you-mean functionality.

Appendix F

Global overview

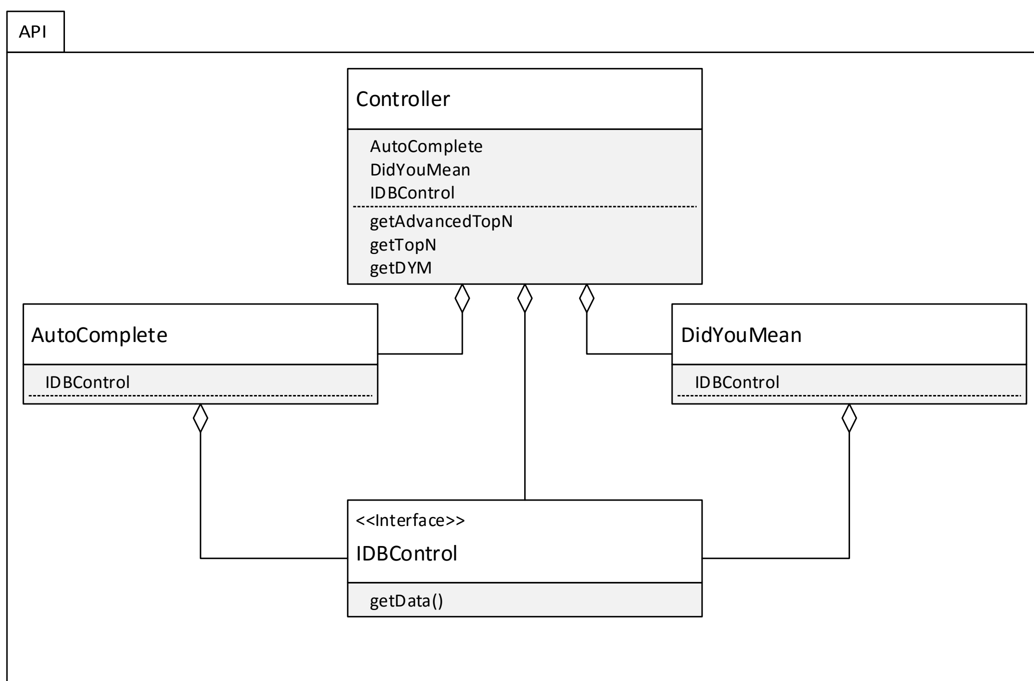

Figure F.1: A global overview of the system.

Appendix G

Test plan

Table G.1: Test plan

#	Test name	Test description	Expected result	Test Class
1	Test max node	Find the node with the highest score in a certain tree. Should be possible on a tree of the entire dataset as well as a subtree.	The node with the highest score in the tested tree.	autocomplete. AlgorithmTest
2	Test exclude keyword	Make sure that when exclude keyword is used, that the keyword is actually not in the list, however, also that no unnecessary words are include in the list.	A set of elements that only contains the right words for this keyword.	autocomplete. AlgorithmTest
3	Test get top keywords	Assert that the function that selects the top n search results from a certain node does not return too many words and does not leave out words that actually should be in the list.	A list of strings of the correct length containing correct strings.	autocomplete. AlgorithmTest
4	Test auto complete	Test the entire auto-complete functionality. Make sure that given a keyword the right words are suggested as results. Also make sure that the right number of words is suggested, if possible.	Given a dataset and a keyword, the list with the right words that the auto-complete returns.	autocomplete. AlgorithmTest

#	Test name	Test description	Expected result	Test Class
5	Test controller get advanced top n	Test if the advanced top n returns the right results, This will test both the auto complete and the did-you-mean. A keyword will be given and the auto complete class will be used to find suggestions, when no five suggestions are found did-you-mean will be used to find typing errors, the new keyword will be used to find improved auto complete suggestions	Given a dataset and keyword, the right list of words that the auto-complete returned, possibly complemented by a did-you-mean correction.	controller. ControllerTest
6	Test CSV control	Test if a CSV file is imported correctly. Test if the words have the right scores attached to them. Test if the right data is actually obtained. Test if an import of a wrong file actually fails.	A score for the query if the import was supposed to be successful and an error for the failing import.	database. IDBControlTest
7	Test IDBControl	Test the actual data gathering class in the API. Make sure that the method used there to import data also retrieves the right files with the right words and correct scores connected to them	Data consisting of words which have a correct weight.	database. IDBControlTest
8	Test BK-Tree nodes	Test that a word of a node cannot be null. Test that a parent of a node cannot be null, Test that the getters of a node is working correctly. Test if adding a child to a node works correctly.	An illegalArgumentException for creating a null node and for creating a node with a null parent. Testing the getters should result in the right values and the add child function should create a new node that is stored in the correct place in the tree: On an edge with the right distance and with the right parent.	didyoumean. bktree. BKTreeTest
9	Test BK-Tree search	Test search strings to see if they are corrected to the right known search queries. This means the search query with the highest weight, when combining the distance and the score of the search query.	The node with the highest score for this search string.	didyoumean. bktree. BKTreeTest
10	Test BK-Tree build and increment	Test that a tree is correctly build from a dataset that has words combined with scores. Furthermore, after a tree is created test that this tree can be incremented with newer data. Where new words can be added and the score of existing words is incremented.	The tree is build correctly with the right nodes on edges with their corresponding distance to the parent. On future updates scores of words are incremented with the correct amount and new words are added on the right place.	didyoumean. bktree. BKTreeTest

#	Test name	Test description	Expected result	Test Class
11	Test BK-Tree Calculate levenshtein distance	Test the method that calculates the levensteihn distance by checking the distance between two words and make sure that deletions, insertions and substitutions all count as one.	The respective Levenshtein distance when the words are different, zero when the words are equal.	didyoumean. bktree. BKTreeTest
12	Test BK-Tree	Test the entire BK-Tree from the did-you-mean class in the API. Check the top results for wrongly typed and correct search strings. Make sure that they return the most popular search query with the smallest distance.	The word with the highest weight. There are no results that have a Levenshtein distance outside the error range of the did-you-mean.	didyoumean. bktree. BKTreeTest
13	Test prefix tree for illegal arguments	Test that no null words or letter can be added to nodes, except for the root. Test that parents of a node, except for the root, cannot be null.	IllegalArgumentException for all these tests.	tree.TreeTest
14	Test prefix tree get children and has children	Test that when getting the children of a node that all of the children and the correct children are returned in the list. Also make sure that this is not possible for a leaf node. For has child make sure that only true is returned for nodes that have children and not for any other nodes.	For the get children test the result of a node is the right set of children that has all the children of that node and no other. For the has child test true is returned for nodes with children and for no other.	tree.TreeTest
15	Test other getters and checkers	Test that the getters for a letter for a node or leave are working correctly. Test that the is root and is leaf methods also work correctly and only return true if they are a root or respectively a leaf.	For getters, their respective values so either the name or the letter. For the checkers true is only expected for roots or leaves.	tree.TreeTest
16	Test prefix tree search	Test different search strings to see what words or search queries they return. Also test that nothing is returned when no correct search queries can be found in the data.	For different search strings the result is the correct suggestion, the one with the highest weight in the used dataset.	tree.TreeTest
17	Test prefix tree add or increment functionality	Test that the tree can be updated correctly with new data. The data should be stored in the correct place in the tree and should also have the correct parent attached to itself. The scores of existing words should also be incremented correctly.	After adding new words to the tree, the search is able to find the word, the new word has a parent and the given score. For an increment the incremented word has the correct new score: the old one plus the added score.	tree.TreeTest

Appendix H

GUI

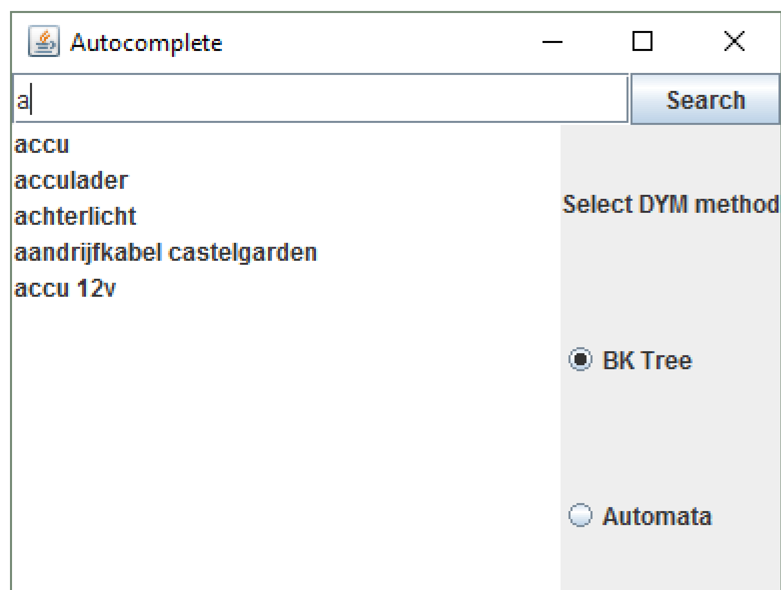

Figure H.1: GUI with auto complete active giving suggestions to complete ‘a’.

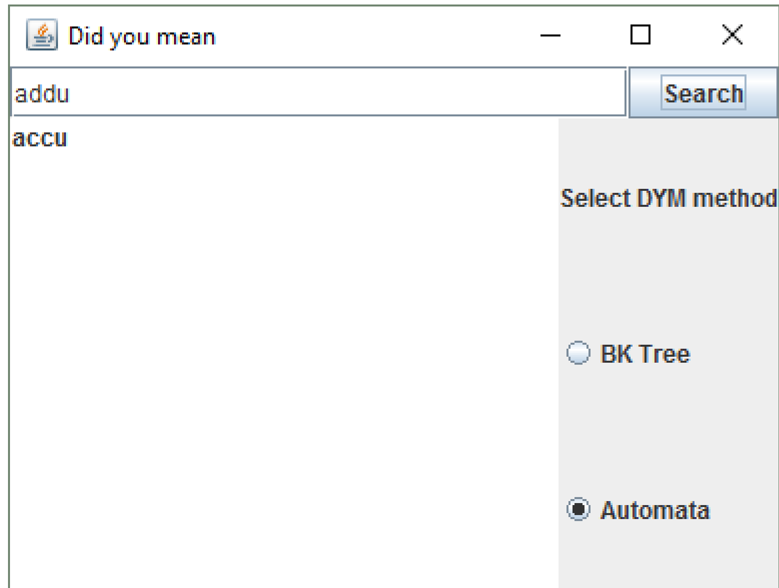

Figure H.2: GUI with did-you-mean active giving an alternative for ‘addu’.

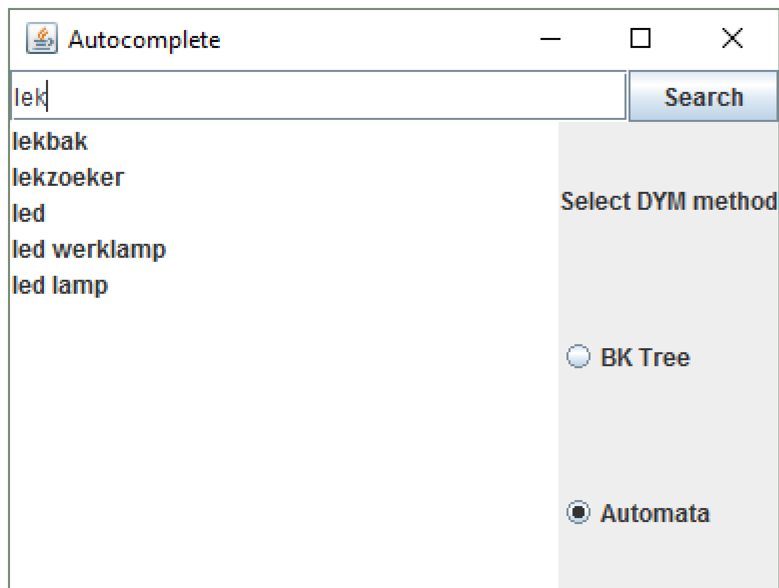

Figure H.3: GUI which uses the *getAdvancedTopN* method to give suggestions to complete and correct ‘lek’.

Appendix I

Boxplots of did-you-mean query times

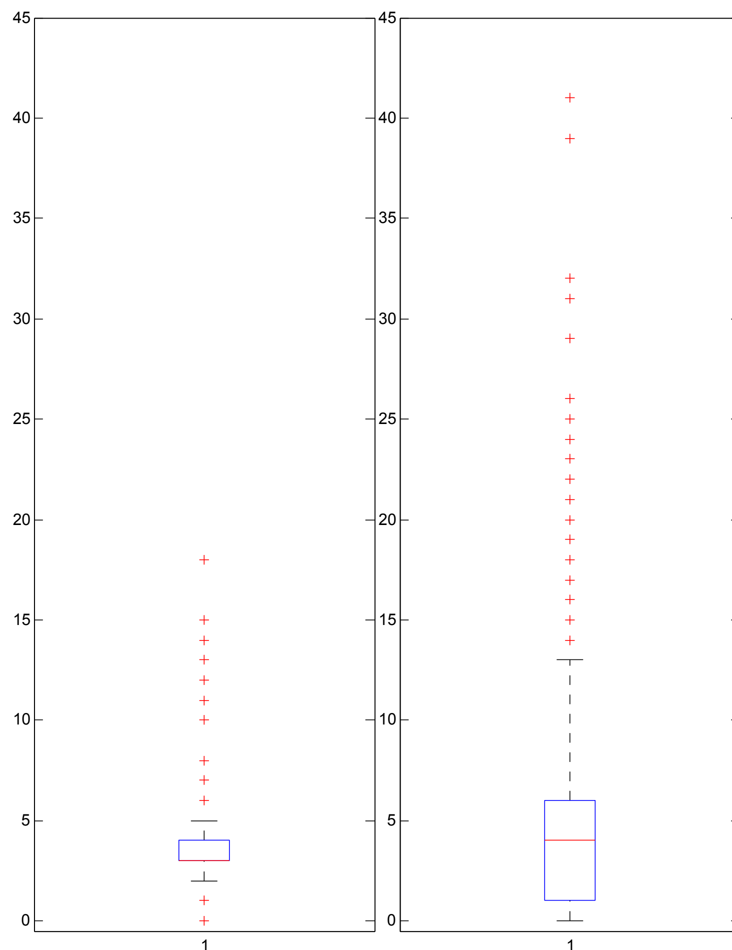

Figure I.1: Boxplots of the query speed of the two did-you-mean implementations, BK-tree on the left and Levenshtein automata on the right.

Appendix J

Meetings with Kramp

J.1 Meeting 0

For the first meeting (before the start of the module), we went to Varsseveld to visit the office of Kramp. Here we met Ron Huiskamp, he is a product owner. We talked about what we have to do for the design project and what our and their interests are. It is very important that we enjoy doing the project, but also that in the end it is useful for Kramp. The reasoning behind this was that it's no fun to do a project for 10 weeks and then nothing is done with it ever again. Also, the project had to be independent of the systems they already have. For example, a part of the website was nearly impossible, because then we had to acquire a basic understanding of the website, which is huge, and this would take too much time. Ron also gave us a tour trough the office, which mainly existed of a huge warehouse. Ron would meet with other project owners to decide what would be a good project for us.

Later that week Ron sent us the project description and brought us in contact with Maxim Klimenko, another product owner and our client for the module.

J.2 Meeting 1 (Skype)

This was our first meeting with Maxim. He tells us Kramp wants to become a modern e-business, that is, do everything trough their website. This meeting we mainly talked about the project description that was sent to us, which was somewhat vague.

Do we just make a modular application? What is the in- and output? To integrate something in the internal system is too complex. Just the functionality of auto complete and did-you-mean is not good enough. 1 million items have to be managed, they all have to be managed in different languages. Kramp is a B2B company, so professionals are using the search function. For example, someone said battery, but meant accu for tractor. So there are 2 parts of the project: the auto complete and the did-you-mean. Auto complete will solve 50-60% of the problems. These 2 things are tied together, however. If possible use personalized data too.

Is there an auto complete already? or is this part of the assignment? There is one already, it is based on live index control, it just suggests the first options (alphabetically) that are possible. Based on 3-5 fields of the store. Categories customers can search in are product numbers, product name and category.

How to interact with the DB's? Kramp has a normal DB with complex models. There is a relation between item and product. It is possible to export data so that we can work indepen-

dently of Kramp's environment. We can get access to Google Analytics. We have to sign a Non-Disclosure-Agreement.

Is there a dev-environment? Question became obsolete, we develop independent of Kramp.

What do we have Access to? Actual data, test data, etc. To get access, we have to mail Maxim to tell him exactly what data we want.

How often will we schedule (scrum) meetings? How long are the sprints? At Kramp they work with sprints of 2 weeks, Maxim thinks 2 weeks is also fine for us. Start of with 1 week sprints, then after we start implementing, make it bi-weekly. Rough planning: one part how to implement (conceptual, what data we need, how to analyse it) Make a plan, what is in the scope, what can we do. Conceptual presentation, saying how we will handle everything. 1 part actually implementing it.

We are also thinking about the ethical aspects. It is okay. The goal is to make the customer happy, so customer first. We don't want to damage them.

The project will take 9 weeks, including reports, priority is university. This is also fine.

Other things We will get an account for the web shop. Next meeting we will have a Trello board. We will think about what is doable.

J.3 Meeting 2 (Skype)

This meeting was mainly about what data we want from Maxim. After doing some research, the data we think we will be needing is a history of search strings (possibly personalized) and whether this was search string had success or not.

Can we get a history of search strings and which of those were successful? There is no telling exactly which search strings were successful, but they do have Google Analytics, which has some statistics. This data also contains enough information to calculate an appropriate weight for each search string. Maxim would send us the data after the meeting.

Can we use (a part of) the current search algorithm as a base for our project? It is not necessary to use a search algorithm, we just need to give suggestions for auto-completion and did-you-mean.

Other things It is very important to quantify the result. They want measurable results, like for example more sales. We will have to make it clear what we want to achieve with the improvements.

J.4 Meeting 3 (Skype)

What is cheaper? More processing power or more memory? -> Is pre-calculation preferred or not? Pre-calculation is preferred, because this will be faster at the time of search. They will soon get new machines with a lot of processing power.

You told us to improve these functionalities, but what is improved? Should we test the new version with clients? We can use mean reciprocal rank (MRR) to test if the algorithms work. Testing with customers is too gut-feeling based. Keeping track of how many clicks it takes for a customer to reach a specific product is also a possibility.

Is there a way to interview or survey clients so we can find out what the current problems are with the auto complete/did-you-mean? This is not preferred, Kramp has customer support that knows about the problems customers have. If we have a questionnaire, Maxim can distribute it.

Is it visible in the data whether an item was clicked after a search or not? Although indirect, it appears to be so.

Other things Maxim showed us trough Google Analytics. They had roughly 20,000 different search queries in the last 5 months. They are missing product data for now. If one needs a very specific product, they will search for the manufacturer number, not its name. For this reason, a lot of searches are simply numbers. The auto complete only works after logging in and after 3 letters. This could possibly be removed. An example of how it works: 'bat' gets nothing until you fully type 'battery'. One customer may expect battery, another something else. Technical terms such as 'battery 85mA' are not supported well.

J.5 Meeting 4 (Skype)

This meeting we did not have a lot of questions, it was mainly to give an update to Maxim. We told Maxim what we had done so far and we will send a weekly update on our design report (if it is useful at least).

Other things If the data has to be cleaned we can contact Maxim. It was not necessary so far though. Also, if we need more personalized data we can contact Maxim. In the end it will have to work on a server, we should take note of that. When we are ready to show something, Maxim will invite some other people to the conversation so they can also check it out.

J.6 Meeting 5 (Skype)

This meeting was a badly scheduled, because effectively there was only 1 workday between the last meeting and this one. Also, today Maxim was joined by Eric.

What do you think of our design report so far? Shall we send a new version every Thursday afternoon? Maxim did not have time to read it completely. He said the design and the approach are good, we should continue like this. Also, Maxim found it very technical.

Can we test with users? Kramp has previous search queries, so they know what users are looking for. Google Analytics is all we have, but at Kramp they know about queries that result X, but should result in Y. We will have to think about how to do this. If there is not enough information we still have to use actual testing.

Eric Eric is responsible for changing Kramp into a full web shop. He has some feedback on what the functionalities should be doing. Kramp has an API for search efficiency. We should think about how we can test this, and how they can access it as well. Be sure to supply Kramp with a manual.

Other things At the moment we have enough data to implement the algorithms. In the future we may need more data to supplement the personalization of searches.

J.7 Meeting 6 (Skype)

This week we became a bit unsure if the did-you-mean did enough, so that mainly was the subject of this meeting. Besides that it was a short meeting. Also, Maxim was sick, so we had a meeting with Eric.

What is ignored while currently searching? (quotation marks, dots, etc.) Eric doesn't know, we will hear that later from Maxim.

What do they expect from the did-you-mean function? Should it work on product names or search queries? It should work on search queries. Product data can contain 'unhandy' terms which can be very unnecessary.

Can we possibly get indexed product data for the did-you-mean? Question obsolete because of the last question.

We want to test whether what we made is good enough, how do we go about this? User tests, MRR tests, etc.? As said a few times before, no user tests. It will be hard to make good structured tests. The MRR is probably still the best way to go.

J.8 Meeting 7 (Skype)

We discussed the prototype we send to Kramp last week and reacted on their feedback.

Auto complete feedback

1. Auto complete only works from the start of the searchterm In case of a searchterm that consists of multiple words, the auto complete only works for suggestions coming from the first word. This will be fixed once someone actually searches with these words in the different sequence, because then that term will also be in the list of terms and therefore will be returned.

2. Integration of did-you-mean in the auto complete When a spelling mistake is made while typing a search string then the did-you-mean should be applied on the searchterm and the auto complete should be tried again on the improved search string. This was immediately implemented.

did-you-mean feedback The main feedback on the DYM functionality was if the it is possible to result multiple options for the DYM, this was immediately implemented and is now possible, however, the relevance of this functionality is still up for discussion.

Other things The acceptance testing is discussed, but no further decision were made on this front. Furthermore, the date for the final page was discussed that will be done in week 10 of the project.

J.9 Meeting 8 (Skype)

We discussed with Maxim and Erik how we would finish up the project.

1. When will we do a presentation at Kramp? We decided to do the presentation on the 12th of April at 9:00.

2. How will we do an acceptence test for this project? The people at Kramp did not find this necessary, they were very satisfied with the result and they did not know a good acceptence test.

3. We send you the design report, what were your thoughts about them? They did not read it entirely (which is not surprising since it is quite long)

Appendix K

Sprint reports

K.1 Sprint 1

The project proposal was made, along with the major part of the ethics report.
It was decided that the project would be split in auto complete and did-you-mean. Frans van Dijk, Tim Sonderen and Ramon Onis would work on auto complete. Tim Blok and Yannick Mijsters would work on did-you-mean.

Auto complete Research was completed on possible auto complete algorithms. It was decided that prefix trees would be used in auto complete.

K.2 Sprint 2

Auto complete A pseudocode algorithm and class-diagram was made for the auto complete functionality.

Did-you-mean Research was done and completed on different did-you-mean algorithms. It was decided that both BK-trees and Levenshtein-automata would be implemented to compare effectiveness and speed.

K.3 Sprint 3

Auto complete The algorithm and tree structure was implemented, along with a basic GUI to demonstrate its functionality.

Did-you-mean Pseudo code algorithms were designed for BK-tree and Levenshtein automata algorithms.

K.4 Sprint 4

Auto complete Unit tests were made for the auto complete implementation and, as a result, several bugs were fixed in the implementation.

Did-you-mean The BK-tree was implemented and integrated into the demo-GUI to be demonstrated in a client-meeting.
Due to illness, Yannick was unable to work on the project for a major part of the sprint. Therefore, Ramon was to implement the Levenshtein-automata algorithm.

K.5 Sprint 5

All the code was cleaned up, and everything was checked for correctness one last time. Besides that all the javadoc was checked, corrected an compiled.

K.6 Sprint 6

The last sections of the report were added and finished, then the whole report was checked collectively so every last error was out. This took longer than expected, but it was done in time.