Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Computer Science	Spring semester, 2016 Open / ~~Confidential~~
Author: Morten Wærsland	[illegible] …………………………………………… (signature author)
Supervisor: Reggie Davidrajuh External Supervisors: Derek Göbel & Paolo Pedronzani	
Title of Master's Thesis: Text Pattern Discovery and Extraction. Norwegian title: Tekstmønster Oppdagelse og Utvinning.	
ECTS: 30	
Subject headings: Trie Structure, Regular Expression, Machine Learning, String Matching Algorithm, Neural Network, Data Mining, Java.	Pages: 81 + attachments: 1 Stavanger, 15/06/2016

Text Pattern Discovery & Extraction

by
Morten Wærsland

June 2016

Abstract

This thesis presents a technique for discovering and extracting unknown patterns for structured data. There is no need for pre-knowledge to be able to discover patterns. But by applying pre-knowledge these patterns can be classified. When merging information from structured data, it is important that correct information is merged together. To achieved this multiple techniques are needed to analyse the information. This thesis provides a technique that can increase the accuracy. By collecting unique values using a trie structure, unknown pattern is discovered and extracted. These patterns are represented by using regular expressions and classified by using a decision tree. The technique presented provides regular expressions that are efficient and accurate. Along with the decision tree that classifies correct with a score greater than 80%. This technique can be used to improve the accuracy when merging structured data, increases the knowledge about a file, detect ID values, calculate other measurement including the consistency of a file, and if there are typographical errors.

Blank page.

Acknowledgements

First of all I would like to express my gratitude to Derek Göbel, Paolo Predonzani and Esther Postuma from Avito Loops for their vulnerable time, and for every inspiring meeting.

I would also like to thank my supervisor Reggie Davidrajuh for valuable comments and guidance.

Last but not least, I would also like to thank my family, fellow students, and all my friends.

Contents

List of Figures

Blank page.

List of Tables

Blank page.

Listings

Abbreviations

GUI	graphical user interface
ID	Unique Identifier

1. Introduction

1.1 About This Thesis

Pattern matching is a common technique for extracting information from textual documents. The inputs of pattern matching are a pattern and a text document. Its output is the response of whether the pattern matched the text and, possibly, the positions in the text where the pattern was matched. The problem that this thesis is trying to solve is different in that the patterns are not known beforehand and cannot be provided as input. Text documents can be provided as input and it is known that they contain combinations of words that have a degree of similarity and that could be collectively expressed in terms of a patterns, but the pattern itself is not know. Examples of such words are product codes, job titles, numbers formatted in particular ways, e.g. currencies. The purpose of the research is to find techniques and algorithms that are able to discover similarity and repetitions in the structured data and ultimately to extract their underlying patterns in a certain syntax. These techniques and algorithms can be used to increase the accuracy when merging information from structured data.

1.2 Thesis Outline

Chapter 2 - Background:
This chapter describes the background theory used as an inspiration and to solve the problems presented in this thesis.

Chapter 3 - Overall Design:
This chapter describes the overall design used to solve the problem presented in this thesis.

Chapter 4 - Design:
This chapter explain in detail why the different techniques are applied to solve the problem in this thesis.

Chapter 5 - Implementation:
This chapter explains how the techniques are implemented.

Chapter 6 - Testing, Result and Analysis:
This chapter describes how and why the different tests are performed.

Chapter 7 - Discussion:
This chapter discuss the techniques used to solve the problem presented in this thesis.

Chapter 8 - Conclusion:
This chapter presents a conclusion about the design presented in this thesis.

2. Background

This chapter presents algorithms and techniques that are used in the field of data mining, machine learning, string matching algorithms, pattern search and different data structures.

2.1 Copycat

The Copycat[21][22] algorithm compares different strings and find relation between the different text patterns. The idea of the Copycat can be used to solved other similar pattern matching problem. Copycat analyses when a string changes from one state to another state, and will try to append the same equivalent changes on a different string. What is interesting is how it detects the similarities. The core of this algorithm is to find the relationship between the letters and the textual structure. Then use that knowledge on the other string to figure out what the best change would be. To illustrate how Copycat works look at the example:

- **ABC = ABD**
- **IJK = ?**

The question is what should **IJK** be changed into? In this example there are some different ways to think. What you see is that the **C** is changed into **D**. And at a very basic level you can do the same to **IJK** and change the **K** with **D**. This is not wrong since you have done the same change as in the first string. This is something a computer could gives as an answer. The great property about the human brain is the capability to see the relations between complex patterns. Relations that are not obvious in the first place. This is something a computer struggles to do. The computational power provided by a computer combined with a algorithm that operates like human brain, is a powerful combination. Every human can see the relationship between **C** and **D**. **D** is the successor of **C**. Another important information is that the last character in the string that is changed. Since **C** changes with it is successor, a natural thought is to change **K** with it successor **L**. This is simple example but it highlights some interesting problems.

An Analogy of Copycat is ants trying to find food. Ant will go in different directions to try to find food. When someone finds food the path to that food source gets stronger, meaning more and more ant will follow that path. But there are always someone checking other places as well and if a new place is better, this will become stronger as well. This analogy matches Copycat quit well since this algorithm starts out with nothing and starts searching for patterns and similarities in all directions. It has a list of patterns it can find and every pattern in that list has different weights. Pattern can be group, different length information, actual letters and predecessor/successor. A pattern with higher weight will be harder to discard for the algorithm. As the search goes on, some pattern will be explore more deeply based on their wighting at the time. The unique aspect of this algorithm is that at any point it can find a new more interesting pattern. When this happens the algorithm stops the previous main search and focus on the new pattern. This new pattern can be totally different from the previous. When a strong solution is created and reached a desire threshold it stops. Sometimes it can change to a weaker combination of pattern, before its gets stronger.

2.2 Regular Expression

Regular expression is a way of describing a pattern of characters. A Regular expression consist of multiple regular expression structures. A regular expression structure will match a desire character type. The sequence of regular expression structures are important for the pattern, different sequence with the same structure types will represent different patterns. This pattern is used by computers to search among text to find a match and regarding the pattern search for the complexity can be high. Therefore if the regular expression is written in a inefficient and overcomplicated way the performance will be negatively effected. When creating regular expression it is possible to add a option called quantifiers after each regular expression structure. A quantifier tells how the behavior of the search is gonna be. The correct quantifier can improve performance. These quantifiers are called; *greedy*, *reluctant*, *possessive.* A greedy quantifier "+" will first try find the longest match for a given sub-pattern as possible. This means that for each regular expression structure it will try to max the number of characters that matches the structure the first time. This approach is used for each regular expression structure until the regular expression is a match or a non-match. A reluctant quantifier "+?" is the lacy type, where it tries to match as few character in the text as possible to fulfil the requirements. A possessive quantifier "++" will not give up any character, whereas other quantifier can if necessary backtrack and give up characters to match the rest of the pattern. To backtrack means even if 4 character matches a structure it can give away the 4th character to fulfill another regular expression structure requirement. Possessive quantifier match string as blocks that cannot be backtracked into [12].

2.3 Trie Structure

A trie structure is data structure for fast search and look-up. It can be used for storing keywords or as a dictionary. It is also called a prefix tree structure since it will only based on a prefix extract every word that starts with that prefix. This property gives it a good search performance, since every other word that does not have the same prefix will not be considered. It easy to see in the Figure 2.1, that words that share the same start, also called prefixes, has the same parent node. This property results in that the trie only contains unique values. A trie structure can be constructed with the property to collect more information, e.g. how often a node is activated. This information along with the unique values can easily be extracted from the trie. In short a trie structure removes any duplications from input text and can return a list consisting of unique words.

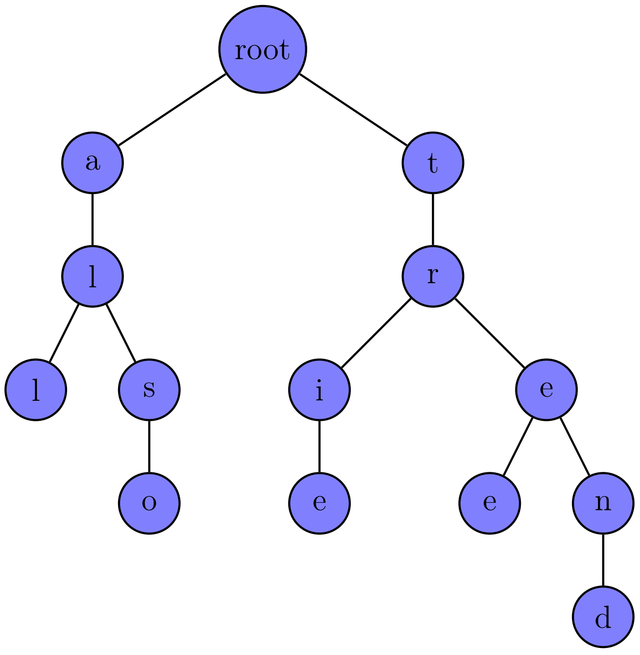


Figure 2.1: A trie containing the words "all", "also", "trie", "tree", "trend".

2.4 Interface Set<E>

Another way to store unique values is to use the *Set<E> Interface*. This structure is a *Collection* that can not contains duplicated values and it is not possible to expand this structure to store more information. The different *Set<E>* implementation are:

- *HashSet*
- *TreeSet*

- *LinkedHashSet*

Both *HashSet* and *TreeSet* can not guarantee the order of the element stored. *LinkedHashSet* is insertion order based, so with this you can stored the corresponding value in a array and use the index to keep track. The functionality of hash based data structure is that it creates a hash value for the given input value and uses this hash value to do insertion, deletion and search. *HashSet* offer constant performance on operations like insertion, deletion and search, assuming that the elements are spread properly among the buckets[13]. The big O notation for each of the implementations are: The average performance for *HashSet* is O(1) and in the worst case it is O(n). For *LinkedHashSet* it is O(1). And for *TreeSet* it is O(log n), where n is size of structure.

2.5 String Matching Algorithms

These algorithms are used to figure out if a small pattern appear in a larger text[3][4]. When comparing different string matching algorithms speeds and efficiency are important features.

2.5.1 Brute-Force Algorithm / Naive String Matcher

The straight forward method is to check each character in the string until a sub pattern matches. For each time a mismatch is detected the pattern slides one step to the right. This technique is inefficient because it ignores the information about where the mismatch happened. This is information that other techniques uses to improve the performance, see Figure2.2

```
1)ABCABD
  ABD

2)ABCABD
   ABD
     .
     .
4)ABCABD
     ABD
```

Figure 2.2: Example of Brute-force algorithm.

2.5.2 Knuth - Morris - Pratt Algorithm

This algorithm uses the information about where the mismatch occurred to calculate the next position to start the search from. The benefit from this is that the the algorithm become more efficient, see Figure 2.3.

```
1)ABCABD
  ABD

2)ABCABD
     ABD

3)ABCABD
        ABD
```

Figure 2.3: Example of Knuth-Morris-Pratt algorithm.

2.5.3 Finite Automata

Finite automaton describes a pattern that is used to search a string value with. Finite automaton is created by using other string-matching algorithms it must be constructed in a preprocessing step before it can be used. This pattern examine each text character only once and therefor taking constant time for each text character. The unfavorable part with Finite automaton is that the time it takes to build the automaton can be large, see Figure 2.3.

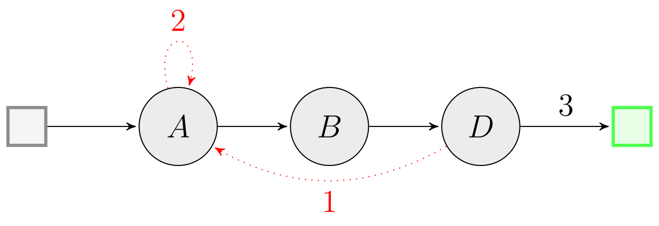


Figure 2.4: The solid line represents a match transition and the dotted line represents non-match transition. This finite automata represent the text pattern and case from Figure 2.3.

2.5.4 Boyer - Moore Algorithm

Like the other string matching techniques mention above Boyer - Moore Algorithm shift the pattern to the right after a mismatch is detected. How far to the right is decide by *bad character rule* and *good suffix rule*. *Bad character rule* checks to see if the mismatch character from the text exist in the pattern. If this character exist it skips the alignment until the mismatch character match the pattern. Otherwise it moves past the mismatch character position. *Good suffix rule* is similar to the other rule. Instead of using one character this rule states that when a mismatch occur the substring matched is used to skip the alignments until the pattern match this substring again. Otherwise move past the mismatch position and start a new search. The unique property is that the character comparison is done right to left. This algorithm is more complex and more efficient than the *Brute-force algorithm*. Boyer - Moore algorithm is a popular algorithm to use in string matching problems, see Figure 2.5.

```
1)ABCABD
  ABD

2)ABCABD
    ABD
     .
     .
4)ABCABD
     ABD
```

Figure 2.5: Example of Boyer – Moore Algorithm.

2.5.5 Rabin - Karp Algorithm

Rabin-Karp uses hash function to search for a pattern in a subtext, instead of character values. The hash value for the pattern and for a given sub-sequence is calculated. Each hash value for a sub-sequence are compared against the hash value for the pattern. This means for each iteration only one comparison is made. When a match is detected *Brute-force algorithm* is applied to make sure it is a match. Since *Brute-force algorithm* is used only once this speeds up the process. In the Figure 2.6a a small example of hash value comparison is displayed. The hash value represent the pattern used in the previous sections. Where *50* represents *BBB* and *47* represents *BBA*. When the hash values match each other a Brute-force comparison is used, see Figure 2.6b.

```
1)50
  47

2)47   3)BBBA
  47      BBA
 (a)      (b)
```

Figure 2.6: Small example of Rabin - Karp Algorithm.

2.6 Back Propagation

Neural network is related to artificially intelligence, machine learning, statistics and other fields. This is a technique used when the problem is hard to solve by using traditionally computational methods. It is design to resemble the human brain in a much simpler design. The design is based on the biological neurons. In neural network neurons is divided into layers. Each neuron in one layer is attach to every neuron in the next layer, see Figure 2.7. This means that the signals flow only in one direction across the neural network.

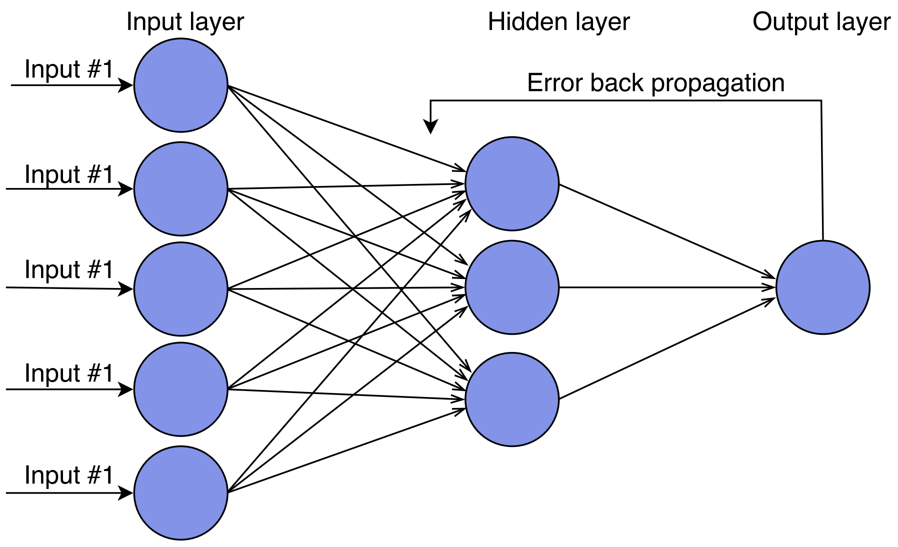


Figure 2.7: Example of a Neural network.

The neurons in the first layers represents the input values. Each time a signal is send to a neuron in another layer this signal is multiple with a weight value. To be able to get the desire result the weight value needs to be adjust. Back Propagation is a technique used to train the weight values in a neural network. By using a training set with training values and the desire output value, each weight values will be adjusted so that the desire result is achieved. Along with the output value there is another value called error value. The goal is to reduce the error value so it drops under some threshold values. Each time the error value is to high the old weight values is recalculated and adjusted. This is done by reversing through the network. This process is repeated until a satisfied error values is achieved[28], see Figure 2.7.

2.7 Data Mining

Data mining is a popular research field. Data mining discovers information and pattern evolving among large amount of text. This information can be used to predict the future trends, behavior, and cut cost for companies. These patterns is not about the structure of the words, but about a the combination and meaning of words. Another term that is also used is text mining. Text mining refers to analysing text that is unstructured. While data mining is used on structured data.

The power of finding good quality information for the user is enormous and in the industry a high priority. Take for example Google search engine where the user give some sort a pattern to be search for. This pattern is normally a word or a sentence. The result is documents that are ranged based on the relevance to the pattern searched for. For a company that has a web store and getting a high rank will have a huge impact on their income. Another example is to discover new patterns and information that the is unknown for the user. And also to increase already known information. This is called text mining and knowledge discovery. Top 10 most used algorithm for data are[30]:

- Naive Bayes
- Support Vector Machine (SVM)
- k-Means
- k Nearast Neighbor (kNN)
- PageRank
- AdaBoost
- C4.5
- CART
- EM
- Apriori

When looking into the algorithms used in these fields somethings that are common, is that most of them need some sort of input to produce an output. And some are also based on a training set and a test set. Training set is often large then the actual test set.The training set is labeled, this means that a golden answer is provided by an expert. These labels are used to to training and build a model. The test set is used to test the model.

One important sub field for data mining is pattern mining[23] Pattern mining uses data mining algorithm to extract information and useful pattern. Pattern mining can fins pattern like; associations, trends, periodic pattern, rules and sequential patterns and is often used in database system. One of the most popular is Apriori[24]. This one is used in transaction database to discover pattern in the transaction. This can be used in other areas as well, but Apriori needs an input by the user to set a threshold and a transaction database.

2.8 Machine Learning

Machine learning uses similar techniques as data mining, but the objective is different. The objective in machine learning is to use algorithm that learns from the data. It makes it possible for computers programs to change and grow as they analyses new data. It can be thought of as a type artificial intelligence. In machine learning there are two type of learning techniques, unsupervised learning and supervised learning. With unsupervised learning the training data does not have a class label. This technique is good when you want to cluster data, find similarities, or extract general rules. With supervised learning the training set has class labels that are used to train and correct a model. When this model achieves a desire threshold it is ready to be used to classify data.

2.8.1 k-Nearest Neighbor

k-Nearest Neighbor is a clustering algorithm that uses a training set to predict class type. The training instances consist of known values where the class value are known. This algorithm is a instance based classifier. This means that each unknown value is classified straight away. This classification is based on the k-nearest neighbors around the unknown value. If there are several different class type inside the k-nearest neighbor, the majority class used. What the value of k should be is a optimization question. If this is to small, it can be more sensitive against noise. If k value is to big, other classes can be to dominated, see Figuer 2.8.

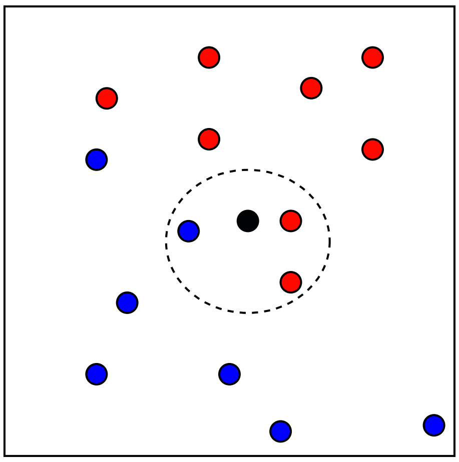


Figure 2.8: 3-Nearest Neighbors. The blue and red dots reprenset known class instances from the training set. The black dot represent a unknown value, where the dotted line is the neighborhood size.

2.8.2 K-means

Like the algorithm mention above K-means is also a clustering algorithm. K-means is one of the most common clustering technique to use. The algorithm needs to declare the number of clusters, called centroids. The first step is to generate the centroids. This can be done randomly or by user specified criteria. After this each value is inserted based on the closes centroid. When every value are placed in the closes cluster, then for each cluster a new centroid is calculation. The new position for the centroid is the minimum distance of each value to its closes centroid. This process is repeated until the new centroid values stays the same. Depending on what the initial centroid values are, different cluster can be made, see Figure 2.9.

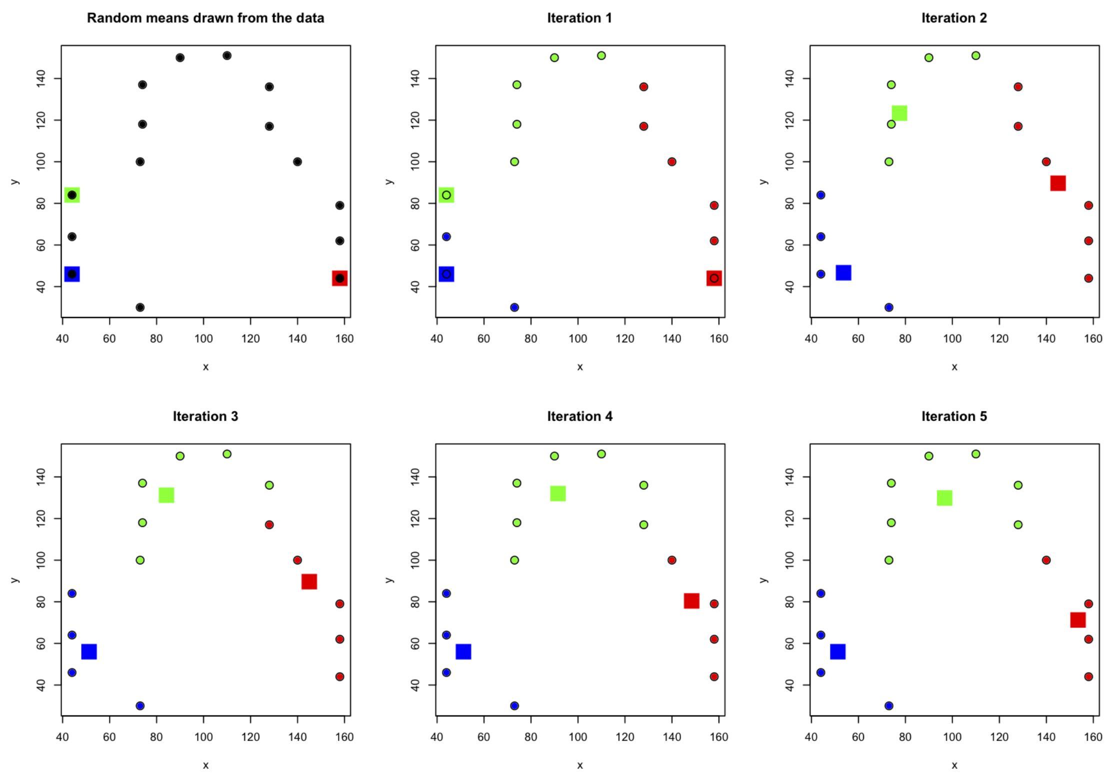


Figure 2.9: In the first window the black dots is random values. And the green, red and blue box represent a centroid. For each iteration the centroids positions changes and also the random values changes into the color of the closest centroid. This picture is taken from[31].

2.8.3 Decision Tree

In machine learning one of many techniques is called decision tree. Decision tree is a supervised learning algorithm. A decision tree can be thought of as a set of rules, and to be able to traverse down the tree each rule needs to be fulfilled. When a leaf node is reach the conclusion about what class the value can be made. The leaf nodes represents the class labels, see Figure 2.10.

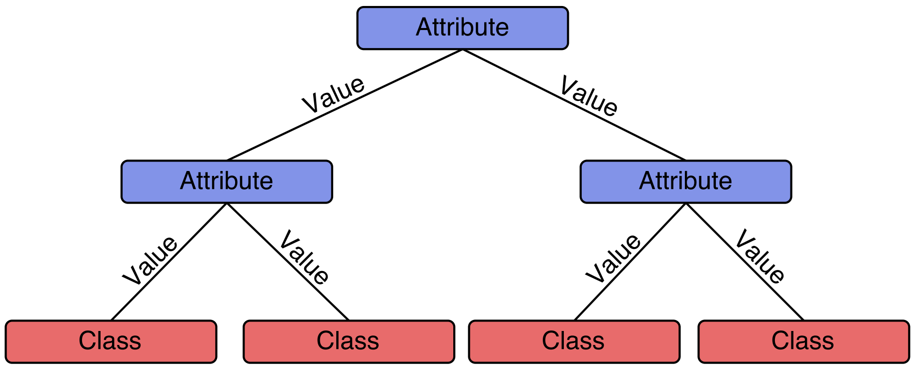


Figure 2.10: Example of a Decision tree.

Decision tree is used for classifying data by using a pre-build decision tree that is created on a large training set. The training set consist of data that are common for that scenario it is trained for. When training a decision tree the question is which attributes to split on first and second and so on. There are exponentially many different decision tree that can be created from a given set of attributes. And creating the optimal decision tree is computationally infeasible[32]. When the decision tree is build it is tested on a much smaller test set. The values in the test set are not the same as in the training set. The decision tree can not be completely trusted, since it can classify the input wrong occasionally.

2.8.4 Decision Tree Algorithms

The use of regression tree/decision tree has been used in several decades. The first regression tree algorithm called Automatic Interaction Detection (AID) is from 1963 *(Sonquist and Morgan)*. This was followed by new techniques, where one of them was called Chi-squre Automatic Interaction Detector (CHAID) *(Gordan V. Kass 1980)* which is a technique that discover the relationship between variables by creating a tree or a predictive medel[20]. Antoher technique is *Classification and Regression Trees* (CART) published in 1984. Both techniques splits the data into smaller segment using an recursively approach. CHAID compared to CART[25]:

- CHAID uses chi-square measure to decide what attribute to split on, whereas CART uses GINI rule. GINI is a measure of impurity and “quotechi-square is a statistical test used to compare expected data with what we collect”[27].
- CHAID support multiple splits, whereas CART support only binary splits.
- CART prunes the tree by testing it against a validation data set or by using cross validation. CHAID does not prune the tree after it is created.
- CHAID is most usefull in analysis, whereas CART is more suitable for prediction[26].

- CHAID has a limitation to categorical data, whereas CART work with either categorical or continous values.

In the same decade another algorithm was presented called ID3. ID3 is tree-learning algorithms and was design regarding machine learning and is also used in Natural Language Processing (NLP). ID3 is similar to CHAID and CART, but it uses entropy information gain measure to determine what attribute to split on. Entropy is like Gini a measure of impurity. ID3 is supervised learning. The next generation based on ID3 was called C4.5 (1993) both created by *John Ross Quinlan*.

C4.5 is popular to use by computer science[30]. The reason for this is the improvement made from ID3 to C4.5 is[19]:

- C4.5 accept continuous and discrete feature.
- C4.5 handles unknown/missing attribute values.
- C4.5 prunes the tree after creation (Solve over-fitting).
  - Pessimistic prediction error.
  - subtree raisting.
- C4.5 can add different weight to the attributes that comprise the training data.

There is also a decision tree algorithm called C5.0[19] that is for commercial use. The improvement from C4.5 to C5.0 is:

- Faster.
- Memory usage is more efficient
- Smaller decision tree for the same results as C4.5.
- Support boosting which improves the accuracy.
- Winnowing (attributes that may be unhelpful can be removed).
- Handle new data types like dates (not applicable values)[19]

3. Overall Design

This chapter explain the general design used to solve the problem presented in this thesis. The techniques used her are describe in detail in the next chapter.

The algorithm presented in this thesis is based on structured data from excel spreadsheets. Each column of interest are converted into text files. This is for the purpose of testing the technique presented in this thesis and not spending time on using "Apache POI"[11] to read and write Excel files. The desire text file is loaded into the program for analysis and the result is displayed in the GUI.

3.1 Architecture

The goal is to discover similarities and repetitions in textual documents. This is done by extracting the patterns. The first thing that needs to be done is to get rid of duplicated values. Since it is not necessary to keep duplicated values when finding pattern. It is only necessary to have one instance of each unique value. By using ordered tree data structure called trie, this is achieve. See section 2.3 for more information about trie. One of the benefit of a trie is that the repetition for each unique word is stored.

Now that we have just unique value it is time to analyse the data to summarize the data in patterns. Pattern can be found and display in different ways. From simple normalization of character like; L for letter, N for number, and S for special characters. To more complex and more powerful expression like regular expression, see section 2.2. Regular expression is used in this solution. What is nice about regular expression is that it can be read just as easily as the simple normalization representation if you know a little about regular expression.

From having data where it was hard to tell something about the patterns and the consistency. It is now quite clear what patterns there are and the consistency of the input. After the general patterns are discovered, the next step is use the information gained to detect typographical errors. Whenever a input value is corrected the corresponding regular expression is updated. If a general regular expression no longer represents a input value it is deleted. The next process is to make these regular expression more specific. Each structure is analysed, and the the minimum and maximum range is added to the end for each structure. These patterns is used to extract more information about the input values. Since the input is different codes and IDs. It is possible to to classify if a column consist of IDs by looking for incrementing digit values. An ID in this thesis is not just a unique value, but unique values where the number values are in a sequence to each other. If it is an ID column there should be a small number of pattern, in other word good consistency, preferably only one pattern. If this is true, the next step is find the majority pattern and analyse this one to see if it contains regular expression structure for numbers. If there is multiple regular expression structures representing numbers, it needs to evaluate which one that an be the ID field. This is done by finding every value that matches that regular expression and extract information for each regular expression structure representing numbers. The regular expression structure with the most unique value will be analyse further. Those number are analyse to see if there is a sequence between them. If the score is better than some threshold it can be declared as an ID.

There is a lot of information stored in the regular expression and by break it down into pieces it can be analyse using a decision tree. If you combine discovered pattern with some pre-knowledge it is possible to tell even more about what the column could be. The decision tree is trained on data that contains the different types of codes and IDs that is expected to be seen in the excel sheets. See Figure 3.1.

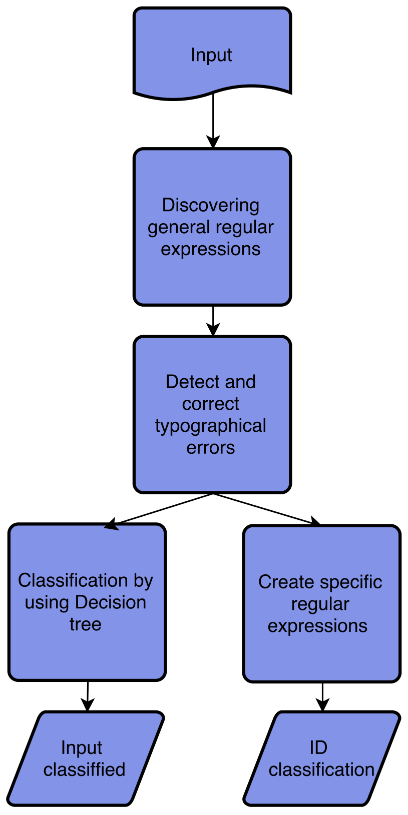
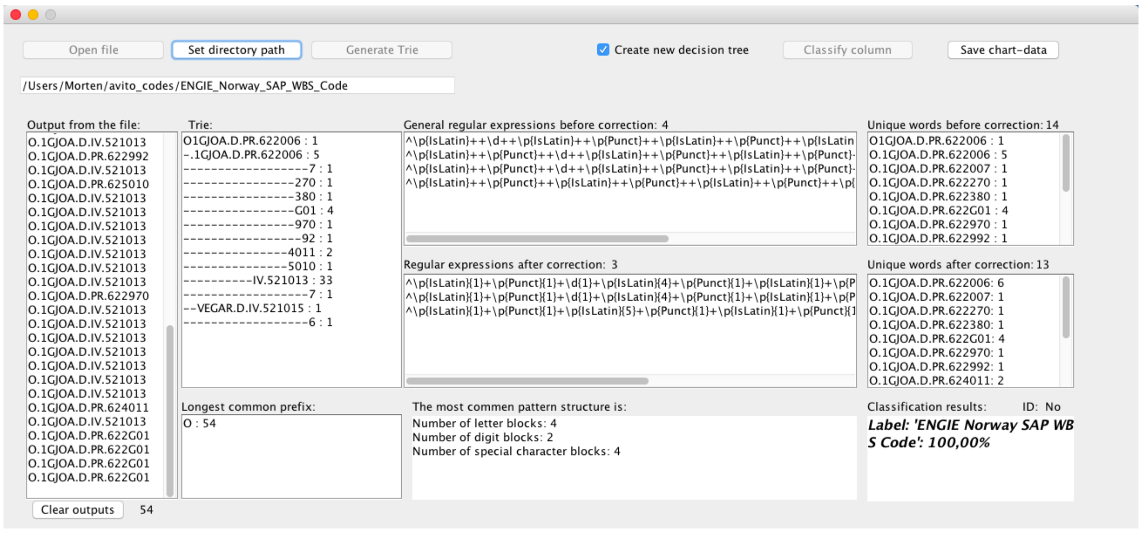

Input
Discovering general regular expressions
Detect and correct typographical errors
Classification by using Decision tree
Create specific regular expressions
Input classiffied
ID classification

Figure 3.1: Overview of the design.

3.2 Graphical User Interface

A informative GUI is implemented to display all the different information collected and found. The user interface consist of several buttons. One for choosing the file, another to find the patterns, a third one to classify the input. There is also a button to reset the program so that it is possible to analyse a new file. The first window from left displays the original input file without any correction or changes made. The second windows from left display the result from the trie. The first value displayed is from top to bottom in the trie. If some of the node has multiple children these are printed out by taking the node that got the lowest level first. To represent a common branch this "-" symbol is used. Since a text area component [15] prints digit and letter exactly the size of the characters, the print displaying the result from the trie can be a little bit off regarding the branch representation. The windows displaying the regular expressions represents the pattern discovered from the input file. The window at the top displays a more general regular expression with the distribution value at the end. The window below displays the regular expressions with the range for each regular expression structure. At the end of each regular expression the distribution value are displayed. Another difference is that the first set of regular expression is displayed before any analysis are done, to see if there is a special character missing or if there is to many many special character. If a correction is made on a value this means that a regular expression may not match any value longer and needs to me deleted. This result is displayed in the other set of regular expression with the range also added. The last windows to the right displays the list of unique word before and after correction and also the distribution. The window in the lower right corner displays the classification result with a score. The score is calculated based on how many vote the majority class achieved compared with the total number of votes. Right above this window a label displays if this input can be considered to be an ID type.



/Users/Morten/avito_codes/ENGIE_Norway_SAP_WBS_Code

Output from the file:

```
O.1GJOA.D.IV.521013
O.1GJOA.D.PR.622992
O.1GJOA.D.IV.521013
O.1GJOA.D.PR.625010
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.PR.622970
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.IV.521013
O.1GJOA.D.PR.624011
O.1GJOA.D.IV.521013
O.1GJOA.D.PR.622G01
O.1GJOA.D.PR.622G01
O.1GJOA.D.PR.622G01
O.1GJOA.D.PR.622G01
```

54

Trie:

```
O1GJOA.D.PR.622006 : 1
-.1GJOA.D.PR.622006 : 5
------------------7 : 1
----------------270 : 1
----------------380 : 1
----------------G01 : 4
----------------970 : 1
-----------------92 : 1
---------------4011 : 2
---------------5010 : 1
----------IV.521013 : 33
------------------7 : 1
--VEGAR.D.IV.521015 : 1
------------------6 : 1
```

Longest common prefix:

```
O : 54
```

General regular expressions before correction: 4

```
^\p{IsLatin}++\d++\p{IsLatin}++\p{Punct}++\p{IsLatin}++\p{Punct}++\p{IsLatin
^\p{IsLatin}++\p{Punct}++\d++\p{IsLatin}++\p{Punct}++\p{IsLatin}++\p{Punct}-
^\p{IsLatin}++\p{Punct}++\d++\p{IsLatin}++\p{Punct}++\p{IsLatin}++\p{Punct}-
^\p{IsLatin}++\p{Punct}++\p{IsLatin}++\p{Punct}++\p{IsLatin}++\p{Punct}++\p{
```

Regular expressions after correction: 3

```
^\p{IsLatin}{1}+\p{Punct}{1}+\d{1}+\p{IsLatin}{4}+\p{Punct}{1}+\p{IsLatin}{1}+\p{P
^\p{IsLatin}{1}+\p{Punct}{1}+\d{1}+\p{IsLatin}{4}+\p{Punct}{1}+\p{IsLatin}{1}+\p{P
^\p{IsLatin}{1}+\p{Punct}{1}+\p{IsLatin}{5}+\p{Punct}{1}+\p{IsLatin}{1}+\p{Punct}{1
```

The most commen pattern structure is:

```
Number of letter blocks: 4
Number of digit blocks: 2
Number of special character blocks: 4
```

Unique words before correction: 14

```
O1GJOA.D.PR.622006 : 1
O.1GJOA.D.PR.622006 : 5
O.1GJOA.D.PR.622007 : 1
O.1GJOA.D.PR.622270 : 1
O.1GJOA.D.PR.622380 : 1
O.1GJOA.D.PR.622G01 : 4
O.1GJOA.D.PR.622970 : 1
O.1GJOA.D.PR.622992 : 1
```

Unique words after correction: 13

```
O.1GJOA.D.PR.622006: 6
O.1GJOA.D.PR.622007: 1
O.1GJOA.D.PR.622270: 1
O.1GJOA.D.PR.622380: 1
O.1GJOA.D.PR.622G01: 4
O.1GJOA.D.PR.622970: 1
O.1GJOA.D.PR.622992: 1
O.1GJOA.D.PR.624011: 2
```

Classification results: ID: No

Label: 'ENGIE Norway SAP WB S Code': 100,00%

Figure 3.2: User interface

3.3 Modular Approach

This section explains how some of the different challenge is resolved.

3.3.1 Regular Expression Structures

The question is how to make regular expression that is suitable for this solution. One of the problem with having a computer making a regular expression is that there is many regular expression structures that have a lot in common. Here is some regular expression structure for letters:

- \p{Lower}, A lower-case alphabetic character: [a-z]
- \p{Upper}, An upper-case alphabetic character:[A-Z]
- \p{Alpha}, An alphabetic character
- \p{IsLatin}, A Latin script character (script)
- \p{InGreek}, A character in the Greek block (block)
- \p{Lu}, An uppercase letter (category)
- \p{IsAlphabetic}, An alphabetic character (binary property)
- \P{InGreek}, Any character except one in the Greek block (negation) letter (subtraction)

It is not possible for the computer to know what type of regular expression you want without an input[1]. To handle this problem an algorithm that creates regular expression structures from a toolbox implemented. This toolbox contains desired regular expression structure. The desire behavior was that the pattern should distinguish between basic letters, digit and special characters. The algorithm checks one input value character by character. Based on the result it will add a new regular expression structure to a new regular expression value. When it is finish building the regular expression it is then stored in a list. The first time the algorithm runs it will always create a regular expression based on the first input. If the next input is not a match, then a new regular expression is made.

When a list of regular expression is presented without any additionally information, it can not tell what the distribution of each regular expression are, see Table 3.1a. Therefor a counter is implemented to keep track of the distribution for each regular expression. This gives a much richer picture of the input data, used to detect typographical errors, and can be used in other analysis like consistency measures, see Table 3.1b.

General regular expression without distribution
^\pIsLatin++\d++\pIsLatin++\d++$
^\pIsLatin++\pPunct++\d++\pIsLatin++\d++$

(a)

General regular expression with distribution
^\pIsLatin++\d++\pIsLatin++\d++$: 94
^\pIsLatin++\pPunct++\d++\pIsLatin++\d++$: 1

(b)

Table 3.1: Example of regular expression with and without distribution

3.3.2 Alphabetic Issues

Norwegian alphabet is based on the latin, with *æ,ø,å* added to the alphabet[2]. This means that some of the classic regular expression structure does not match the letters in the Norwegian alphabet. The most understandable way to write a regular expression for letter is [a-zA-Z] and for digit is [0-9]. As you can see this does not work for the Norwegian alphabet. A cleaner way to write regular expression structure for digit is the "\d" and for characters \w. \w will detect letters a-z, but this will also accept digit 0-9. In this algorithm it is important that letters and digit are separated. Therefor the \w can not be used. To handle Norwegian letter the structure can be written as \pIsLatin. This will only detect letters, including the Norwegian letters. It is also important to create a regular expression structure that will detect special character in the text given as input. A regular expression that will detect exactly this is "\p{Punct}".

Blank page.

4. Design

This chapter explains in detail why the different techniques are used to solve the problem presented in this thesis.

4.1 Trie

The first technique that is applied is the trie structure. The benefits of using this technique are that it guarantees non-duplicated values. This property increase the performance of further analysis. The time it takes to run through the data can be reduced drastically since it will not analyse the same word twice. The disadvantage of removing duplicated values is that information about distribution will be gone. This can be fixed by adding a property that counts how many times a trie node is a leaf node in the algorithm. This results in a trie structure with unique words along with their corresponding representations.

Trie structure is useful if you have data where the search performance is an important feature, like fixed dictionary. The text search in a trie structure is independent of the size of the documents being searched, since the trie has already been build [9]. The technique presented in this paper is a twist of what it is normally used for. This trie is actual build on the document wanted to be analysed. This means that each unique word in that document are stored in the trie strucuture. Instead of using this trie to do search on other documents it return the values stored in the trie. These values is later for finding patterns and extract information for that document.

The limitations of the data structure mention in section 2.4 are the lack off possibilities, complexity and functionality. Whereas trie structure can return each value along with the corresponding distribution. It can also find the longest common prefix for the input. This is information that is not possible to extract from the Set<E> interface. The input values in this thesis are different codes and IDs. This means that the input is quite often duplicated values or has similar prefixes. Based on this the use of trie is more optimal. The performance for the trie is good, the big O for lookup O(1) time[10].

When dealing with big data set that contains a lot duplicated values the space needed to stored the results from the trie are the size of unique value instead of original size of the data. This means that the more duplications there are the greater the benefit of using trie is.

4.2 Regular Expression

One of the criteria was that the technique chosen would represent discovered patterns in a informative and efficient way. The simplest version mention in section 3.1 will give the user a good understand of what pattern there is. But it is not computational efficient to find them. Since it is necessary to iterate over each character for each word to extract each pattern. The benefits by using regular expression is that regular expression represent a pattern. And this pattern can be applied directly into a pattern search algorithm. This algorithm can match patterns with strings without doing any preprocessing. The techniques present in this paper have two set of regular expression levels. This first step is to have regular expression that is more general and the reason for this is that a pattern can be the same even if the length of some components are different. e.g. *123b2* has the same pattern (*digit, letter, digit*) as *12b3*.

When building these regular expressions a possessive quantifier is added after each regular expression structure. This is done to give the regular expression the behavior that each regular expression structure needs to match at least one or more times. Possessive quantifier will not allow backtrack into already match fragments. The whole text needs to match every component for a given regular expression and therefor looking at each match fragment as a block that can not be backtracked are a desire behavior. The performance will be good since when there is not a match it will be detected earlier than by using the other quantifiers.

The next step is to create a more specific regular expression where the minimum and maximum length are added at the end of each regular expression structure. If specific regular expression had been discovered instead of the general regular expression in step one. There would have been a lot more regular expression created. Since the property for a regular expression with the range added to it would increase the number of mismatches although the pattern is the same. e.g. *123b2* has the pattern (*digit3, letter1, digit*) and *12b3* has the pattern (*digit2, letter1, digit*). This will not be the result in this thesis, since it uses the previous created regular expressions through the second step and creates new regular expression with the range added to it. After each value are finished the range found is used to update the minimum or maximum range value to the new corresponding regular expression.

When analysing a pattern that got this range information it is easier to differentiate between two similar patterns. The range value will also tell how good the internal consistency for a pattern is regarding the input values. This is because when input values has the same length for each regular expression structure, the range value will consist of only one number. The internal consistency can be poor if the range is big.

Without specifying that the whole regular expression should match the input value it will not do that. By using "^" in front of the regular expression and "$" at the end it will force the whole text to match."^" symbol means from the beginning of the line and "$" symbol means at the end of the line.

4.3 Detection and Correction

The analysed files comes from different sources and therefore a typographical error can occur. Other errors can be that the same code type is written differently. Where some code values are written with special characters and some without. In this thesis only special character is corrected and not letter and digit. The reason for not fixing missing letters is because there is not always an easy answer to what letters to add or remove. A code type can have several instances where there is different letter on the same position, see Table 4.1. This type of fixes require a another intelligent algorithm.

Code instances
O.1GJOA.D.PR.624011
O.1GJOA.D.IV.521013

Table 4.1: Code extracted from this thesis

The list of regular expressions distribution is analysed. The threshold value is 10 percent of the total number of elements. Each regular expression that is below this threshold is flagged to potential represent values that has typographical errors. The are no rule or information stored to tell what a pattern should be or not. The regular expression that is above the threshold is the only guide for what a correct pattern looks like. These regular expression is considered to be correct. Before further analysis are done it is important to normalized the values before comparing the to each other. Since each combination of digit can be different for the same code. To handle this each sequence of number is normalized into a sequence of zeros. The letters will not be changed. e.g. *kn.123m01* and *kn444m01* will become *kn.000m00* and*kn000m00*. For the human eye it is easy to see after the normalization that the difference between them are a dot.

Levenshtein distance is calculated based on number of deletion, insertions and substitution between two input values. Therefore it is critical to normalize digit before to get a more correct distance.

Levenshtein distance between two string a and b is given by $lev_{a,b}\left(|a|,|b|\right)$ where:

$$lev_{a,b}\left(i,j\right) = \begin{cases} max(i,j) & if min(i,j) = 0, \\ min \begin{cases} lev_{a,b}\left(i-1,j\right)+1 \\ lev_{a,b}\left(i,j-1\right)+1 \\ lev_{a,b}\left(i-1,j-1\right)+1_{(a_i \neq b_j)} \end{cases} & otherwise. \end{cases} \tag{4.1}$$

For interested reader visit this article for more information[35][36]. Each value from a suspected regular expression are compared against each majority value from the correct regular expression. The word that are the closest in distance is used to compare the two values. The two words are compared character by character. Whenever there is a mismatch between the words and if it is a special character it gets fixed. And the counter value representing the number of correction done is incremented. Then it start from the beginning again. If the counter value and the distance value is the same it means that it has fixed all the element needed to be fixed. When this is the case the new value is stored. Also the word counter value and the distribution value for the wrong word are decrease or removed if there are no one left. If the counter value and the distance value are not the same it means that it is not just special character that is different, but also letters. This time the corrected value is not stored.

4.3.1 Unique Identifier

By doing this analysis it is possible to extract more information about what type of values the algorithm are facing. There are different types of ID properties. ID in this thesis represents unique code values where the digit values are in a sequence of each other, see Table 4.2a and 4.2b.

Code instances
O.1GJOA.D.PR.622006 : 5
O.1GJOA.D.PR.622270 : 1
O.1GJOA.D.PR.622380 : 1
O.1GJOA.D.PR.622992 : 1
O.1GJOA.D.IV.521017 : 1

(a) Not considered to be an ID value.

Code instances
P025 : 1
P026 : 1
P027 : 1
P028 : 1
P029 : 1

(b) Considered to be an ID value.

Table 4.2

The less pattern discovered the better probabilities are there for the input value to be ID. It is important that the value analysed has the same pattern. This is solved by taking the majority pattern found and finding every value matching that pattern. Even if there is only one pattern there is no guarantee that this is an ID type. The majority pattern is analysed to see if it contain regular expression structure for digit. It keeps track of how many digit structure it finds. This information is used to make sure that when extracting digit information later, the correct digit information is extracted. Since some digit structures has a range that crosses other digit structures ranges values. This means that some digit structures can match earlier than it is intent to match. The first problem is that quite often a code got multiple field of digit and the question is what part should is the ID field. Each regular expression strcuture representing a digit extract their values into a *HashSet<String>*. And the *HashSet<String>* with the most value will most likely be the correct field to analyse futher. Since the property of *HashSet<String>* is that it only stores unique values.

The next process is to analyse to see if there are any sequence between the number stored the larges *HashSet<String>*. The definition of an ID in this thesis is that the digits should be in a sequence regarding to each other. Digits that are spread to wide apart are not classified as an ID. The reason for this is that it is not possible to tell if these digits are random value or ID values.

4.3.2 Decision Tree

Decision tree is supervised learning algorithm that can deal with codes that has multiple patterns. Decision tree is made up by rules, where each rule tells something about the structure. Since each class label has the same probability, it is better to use rules to decide what the input value should be classified to be than a probability algorithm.

In this case there are good pre-knowledge about what code that is typically seen in a excel data sheet. And therefore it is possible to use supervised learning to extract even more information. This technique uses information about the patterns found from the regular expression and the length of the words to construct a decision tree. This information provide enough information to differentiate code type from each other most of the time. There is some limitations since different code types can have instances where they got the exact same pattern. This makes it impossible to differentiate between them. This could be handle by using the value for the characters and digits when building the decision tree. But since the focus in this thesis is about discovering pattern and extracting as much information as possible from the patterns, this is not implemented. Another argument is that if there is a new code it is no guarantee that it will be classified correct. But since each code has a set of patterns that are common, this new value will most likely have one of these pattern and it will be more likely classified correct. Based on these arguments the decisions is made to use decision tree to classify data.

Each information extracted from the regular expressions are used as an attribute in the attribute table used by the decision tree. The information extraced are:

- The length of the value.
- Number of regular expression.
- If it is only letters.
- If it is only digit.
- If it is letters and digit.
- Number of special character blocks.
- Number of letters.
- Number of leading letters.

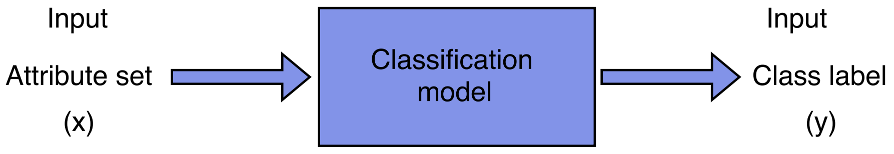


Figure 4.1: Basic classification model[32, p.146].

To be able to classify a column of text from a excel data sheet it needs to classify each row and then find the class label that is the majority. This majority class label is the answer to what type of code a column contains. Since it goes through each row it is important that the classification is as fast as possible. The benefit with a tree structure is that it has good performance for search. This property is important for the algorithm presented in this thesis.

In a perfect world the training set is balanced which makes it easier to build a good model. The data analysed in this thesis provides a training set that is imbalanced. This is since some code types have multiple different patterns, whereas other code types only have one pattern. With a imbalanced training set there are some problems. One problem is that the decision tree tends to become big and complex. To comprehends this it is possible to prune the decision tree. Based on different rules it works from the bottom-up and trim the nodes that can lead to over-fitting and bad results. A prune decision tree can classify differently than the original unpruned decision tree. Another problem with imbalanced training set in this thesis is that some code type had a 1:100 relationship with other code types. The result is that these code types can and will most likely be overlook. This is because when training the model and the data set become small it will not create a new branch. Instead it will give that leaf the value of the majority class label and that is not in favor of the code with a few representations. That being said, decision trees often perform well on imbalanced data sets.

Techniques to handle imbalanced training set are *sub-sampling*, *Panalized models*, *Naive Bayes*, *oversampling* and *Synthetic Minority Over-sampling Technique*

(SMOTE), *SMOTEBoost*, *cross-validation*. *Naive Bayes* can be used to collect samples from the training set. *Oversampling* can be done by create copies of the minority class and try to force the model to learn the desire class label. *SMOTE* creates synthetics samples instead of creating direct copies. This is done by selecting two similar instances and using different distance measure to calculate the new attribute value. The distance is in between the two samples [14]. *Cross-validation* is a technique that randomly portioning the data into n folders. Where n-1 folder is used for training and the last folder is used for testing. This is repeated n times and the average error is computed. Each instance is in the test folder only once. Based on the average score a model is build. The advantage of this technique is that it does not matter how the data is divided.

The techniques used to solve this problem was a combination of *over-sampling*, since the training set was quite small. And *sub-sampling*, since the it was very imbalanced. The number of duplications of each instance is based on a threshold. The goal was to increase the representation for the minority classes. When the original training set had been increased by duplication the next step was to use sub-sampling. The benefit with sub-sampling is that the class distribution spread was chosen to be 10. A 10:1 relationship is the biggest distribution spread allowed. This meant that the old class distribution of 1:100, had become at most 1:10. And since we had increase the number of instance for each class this relationship would be closer. A technique that tell something about how good the decision tree model is, is called cross-validation evaluation. This is used to figure out the accuracy of the generated model. It divides the training set into a multiple sub-training set. How many can be specified with a parameter. For each sub-training set it picks out a couple of value to be used as a test set values, and then test the model.

4.3.3 Decision Tree Algorithms

The WEKA Library

It is important that the algorithm support multiple splits, handle categorical data and has good prediction performance. Based on the property like; how powerful it is, good execution time and it is one of the most preferred method in machine learning. The algorithm used in this paper is C4.5.

There exist libraries that provides different type of machine learning algorithms. The reason for choosing a library to build the decision tree is that creating a decision tree algorithm that has good performance regarding execution time, accuracy and other technique useful to improve the algorithm takes time. The decision tree is created by using the WEKA library[16][17]. The WEKA library is created by *The University of Waikato*. WEKA library provides multiple machine learning algorithms for data mining task. This library is an open source software under the *GNU General Public License*[18]. WEKA can be used as an desktop application or used in your own java code. The documentation is really good. This makes it effortless to implement it into an existing code, and to tune it to get the desire behavior for the chosen machine learning algorithm.

Decision Tree Attributes

The attribute types that is used to classify the input text is:

- The length of the word.
- Number of regular expression structure.
- If it contains only letters (Boolean).
- If it contains only digit (Boolean).
- If it contains both letters and digit (Boolean).
- Number of special character.

The information possible to extract from the pattern are mostly continuous value and Boolean value. This is not a problem for C4.5 algorithm, but the problem by having continuous values is that a binary split is performed. The values get sorted and the gain is calculated. The best split point is used to split the attribute values on. Since there is many categories it is important that the attributes values get differentiated enough. A code that got five regular expression structures to represent its pattern. Does not have a more similar meaning to a code that got six regular expression structures compared to a code with two regular expression structures. This problem is fix by convert a continuous value into nominal values.

The the benefit to create this decision tree is that is only based on the structure, and not the combination of the words or different numeric values. This means that if a code type gets a new value with new letter combinations and numeric range, it will still be classified correct if the pattern is the same.

4.4 Design Alternatives

4.4.1 Simple String Pattern Matching

One approach would be to normalize the values and then just compare string [5] by string to check if it is the same or not. One way is to write every letter as L, number as N and special character as S. This result is fine if you stop after this step, but there are several problem with this approach. The first issue is the loss of information, since the original text has been rewritten regarding to the pattern found, the text has gone from specific information to general information. The second problem with this result is that it is all *String Objects*[5]. Even if the patter found is stored in a new list, it is not possible to extract more information by compare the original input text to see if it matches one of the pattern discovered, without doing a lot of preprocessing of the input text to match the pattern structure. *String Objects* do not know the different from a word or a pattern when this approach is used. Since it checks the actual character values against each other.

4.4.2 Hash Value Comparison

Another approach is to use hash value to compare the patterns against each other. This idea is based on the Rabin-Karp algorithm in subsection 2.5.5. The approach would be to calculated the hash value for each input value and compare it against each other. If you calculate the hash value to the actual word, it would result in that only the exact same word would be a match. To avoid this you can normalize each word and then compare. This will achieve that the actual pattern is compared and not the characters and meaning of the word. The advantage of the use of hash value is that the algorithm only need to check against a number once and not comparing every character. At first glance it looks very promising, but to get the value it needs to check every character to calculate the hash value. This need to be done for the every input value provided. This means a lot of preprocessing needs to be done to be able to the match hash values and this will give a bad performance. Although the algorithm that calculate the hash value is more optimized than the Java string comparing method used in the section mention above. One problem is that hash value is a one way structure. This means it hard to convert the hash value back to the original value. The use of hash value can be a problem if this algorithm was used on distributed systems. Since the hash value could differ from each computer regarding the same word. Therefor it can not be used. And there is no point in storing the hash values to use later, since it can differ from each time the application is run. Another problem is if there is many different pattern a hash collision[6] can occur. The probability for a hash collision to happen in this case is very little, since the value are so small. This means that different pattern have the same hash value. Once again this could ruin the hole trust of the pattern recognition. If there is a possibility that different pattern is considered to be the same pattern. The consequence is that another analysis algorithm that depends on the patterns found, will not work correctly and give wrong result.

4.4.3 Clustering Algorithms

A technique that could be used to classify the input value is K-nearest Neighbor. The training set would consist of desire code types classes. Both the code type instances and the input values needs to be converted into values that can differentiate each code type. After every input values are classified, the conclusion about what class they belongs to are made. The majority class is chosen.

Another clustering technique that could be used, is inspired by K-means algorithm. The properties wanted is partitional clusters that is well-separated. The centroids are initial generated to represent the different code types. Instead of calculating new centroid after each iteration, this technique stop after the first iteration. The reason for this is when new centroids are calculated, values can change to another cluster. The result can be that input values that are of different code types can be classified to be the same code type class.

Both these technique has challenges. One challenge is how to represent the different codes so that it is easy to differentiate between them. One problem is that a code can have multiple patterns. One solution is to use multiple clusters representing each pattern.

4.4.4 Neural Network

This technique is based on neural network, more precis back-propagation. The idea is to use the information about the patterns to create a neural network. This network is used to classify the input values. Each neuron that represents information about the pattern structure are connected to neuron representing the different code types. Neurons that contains characters, number of digit, or range can be applied to increase the performance. Each connection are trained to give the a result that is below a error threshold. This threshold is decided by the user. To build and train the neural network the data set is divided into two set. One set is used for training the network. This training set is labeled. After the neural network has reached a desire score it is evaluated by using the test set. In Figure 4.2 the weight parameters are represented by $w_{i,j}$, where i represents a input neuron and j represent hidden layer neuron. These weight needs to be adjusted to give the desire result. The result is calculated based on the neurons in the *Hidden layer*, that has the highest score.

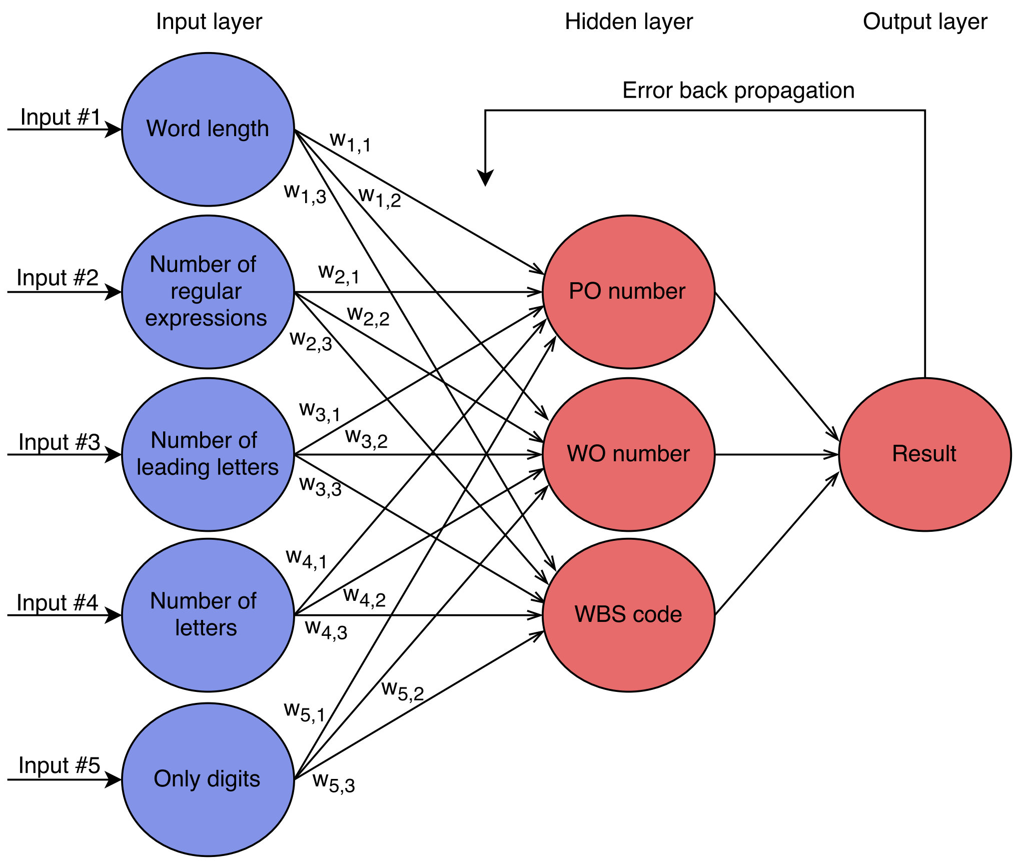


Figure 4.2: Example of a neural network design for this thesis. Where the blue neurons in *Input layer* denotes information about the patterns. And the red neurons in *Hidden layer* denotes information about code types. The *Result* neuron represents the result.

Blank page.

5. Implementation

This chapter explains how the the algorithm design to solve the problem presented in this thesis is implemented. The input is a file that contains hundreds of codes values. Every unique value is extracted by using a trie structure. Based on these values regular expressions is build. If some pattern has low representation these are analysed to see if they are wrong. After this more specific regular expressions are build and also a classification is performed. The code snippets are extracted from the source code. Some places the code snippets converted into pseudo code.

5.1 Trie

When a new column or new input text is presented it takes the first input value and analyse that value character by character. It creates a new *trienode object* if the parent node for the previous character does not contain a child node with that character values. *Trienode object* is implemented to have the property to count how many times it has been used and also counts how many times it is the last character, see Listing 5.2. This last counter is used to display the distribution of each word. The reason for it is necessary to have two different types of counters is that sometimes a small word is inserted, and the leaf node is incremented. The next value has the previous value as prefix meaning that the previous leaf node become a child node for this value.

The process of constructing the trie consist several steps. The first step of filling the trie structure is to take the input and analyse each row one value at the time. When a word is analyse to see if it is new word or not, we need to look at each character, see Listing 5.1.

Listing 5.1: Creating a trie extracted from Trie.java

```
public void add(String s){
  word = s;
  child = root.child;
  boolean leaf;
  for(int i = 0; i<s.length(); i++){
    if(i == s.length() -1){
      leaf = true;
    }else{
      leaf = false;
    }
    check(s.charAt(i), leaf, i);
  }
}

public void check(Character c, boolean last, int index){
  TrieNode trienode;
  if(child.containsKey(c)){
    trienode = child.get(c);
    child.get(c).prefix = word.substring(0, index);
    trienode.incrementNumberOfHits();

    if(trienode.isLeaf && last){
      trienode.wordCount++;
    }

  }else{
    trienode = new TrieNode(c);
    trienode.prefix = word.substring(0, index);
    child.put(c, trienode);
  }

  trienode.level = index;
  if(last){
    trienode.isLeaf = true;
  }
  child = trienode.child;
}
```

To begin with a root node is created with no value attached. This node is a *TrieNode* object. A *TrieNode* object has a HashMap that contains all of its children, called child. This property is used to check if a character is new or if it is already a child for that node. This HashMap is empty to begin with.

Listing 5.2: The Trienode Class from trieNode.java

```
public class TrieNode {
  public HashMap<Character, TrieNode>  child = new
     HashMap<>();
  public boolean isLeaf;
  public int numberOfHits = 1;
  public int wordCount = 1;
  public Character character;
  public int level = 0;
  public String prefix = "";

  public TrieNode(){}

  public TrieNode(char c){
    character = c;
  }

  public void incrementNumberOfHits(){
    numberOfHits ++;
  }

  public int getNumberOfHits(){
    return numberOfHits;
  }
}
```

After this root node is created the first word can be inserted into the trie. Since this is the first word, every node needs to create a new child node with the corresponding character. When the second word gets inserted into the trie the root node search through the HasMap to see if it already has a child with that character value. If a child represent that character value the root node will not create a new child node. But instead use this child node to check the next character value to see if this child node has children that contains this new character values. And this is the process as long as the word still has more character to be inserted. When the last character of a word is analysed a boolean value called *isLeaf* is used to mark that node as a leaf node. see line 57-59 in Listing 5.1 If the character was a new character, then the rest of the characters are new as well. This means that a new *TrieNode* object is created with one child for each character until the last character for the that word.

5.1.1 Extracting the Result from the Trie Structure

The number of children and the construction of the trie will be different for each time new file is analysed containing different words. The property of the trie algorithm design in this thesis is that the best way to display the trie structure is to use recursive method call. There is two ways of displaying the result, one is by displaying the trie structure with the branches and the other one is to display the unique word in a list. The benefit by displaying the trie structure is that for human it will increase the knowledge about what type of patterns there are and how similar the input value are to each other. This is much harder to see by just looking at the unique words. The list of unique words is used in further analysis. The technique for extracting each unique word from the trie is to start at the root node and for each child check to see if it is leaf node or not. A leaf node has the prefix stored so by adding it own character to the prefix the word is complete and this value is stored in a list, see Listing 5.3.

Listing 5.3: Extracting the result from the trie extracted from Trie.java

```java
public void startExtractinWords(){
  child = root.child;
  extractUniqueWord(child);
}


public void extractUniqueWord(HashMap<Character,
   TrieNode> child){
  for (Character c: child.keySet()){
    if(child.get(c).isLeaf && !(child.get(c).character
       == '-' && child.get(c).level == 0)){
      uniqueWordExtracted.add(child.get(c).prefix +
         child.get(c).character);
      uniqueWordCounter.add(child.get(c).wordCount);
    }
    extractUniqueWord(child.get(c).child);
  }
}
```

5.2 Regular Expression

5.2.1 Step One, General Regular Expressions

The first step is to analyse the result from the trie structure to extract every unique pattern in the unique words list. The first check that is done is to see if the list containing regular expressions is empty. If the list of regular expression is empty, it means that it is the first time the algorithm runs. If the list is not empty the word is matched against each pattern stored in the list to see if it match a previous discovered pattern. If the word do not match one of the patterns it needs to create a new regular expression based on that word, see the pseudo code and the real code in Listings 5.4.

Listing 5.4: Pseudo code for checking new input value against discovered patterns

```
check regular expression(row value , word counter)
  if expression list is empty then
      Create new regular expression and updates regular
      expression lists and arrays
    else then
      for each pattern do
          if row value matches one of the patterns then
              do not create new regular expression and
                 updates
```

```
                regular expression lists and arrays

    if row value did not match one of the patterns then
        Create new regular expression and updates regular
        expression lists and arrays
    else then
        update the regular expression counter for the
           regular
        expression that matched the row value
```

The *Character* class[29] provides the necessary method to check characters types.

- **isLetter()** returns true if the character match one of the following criteria:
  - Uppercase letter.
  - Lowercase letter.
  - Titlecase letter.
  - Modifier letter.
  - Other letter.
- **isDigit()** returns true if the character match the following criteria:
  - A character is a digit if its general category type, provided by *getType(codePoint)*, is DECIMAL_DIGIT_NUMBER.
- **isWhitespace()** returns true if the character match one of the following criteria:
  - Space separator.
  - Line separator.
  - Paragraph separator.
  - Horizontal tabulation.
  - Vertical tabulation.
  - Line feed.
  - Form feed.
  - Carriage return.
  - File-, group-, record-, unit-, separator.

The main focus is to check for letter, number and special characters. There is a regular expression structure for letter, digit and special character. In this algorithm everything that is not a letter, digit or white space is considered to be a special character, see Listing 5.5.

Listing 5.5: Regular expression structure extracted from Variables.java

```java
public String letter = "\\p{IsLatin}++";
public String digit = "\\d++";
public String special = "\\p{Punct}++";
public String whitespace = "\\s++";
```

When creating a new regular expression the input needed is the word itself. The first check is to see if it is not just one character. If the length is greater than one it analyse each character and add the corresponding regular expression structure to the new regular expression, see Listing 5.6. In addition to the regular expression structure for letters, digit, and special characters, the "^" symbol is added at the begging of the regular expression, and at the end of the regular expression "$" symbol is added. A line in this scenario refer to the input word.

Listing 5.6: Pseudo code for creating a new regular expression

```
Create regular expression(row value)
    new regular expression value
    if the number of character is > 1 then
        for characters except the last character do
            if letter and next is not then
                add a letter structure
            else if digit and next is not then
              add a digit structure
            else if white space and next is not then
              add a white space structure
            else if is neither letter,digit or white
               space and the next one is then
              add a special character structure

            if second to last character then
                if last is letter then
                      add a letter structure
                else if last is digit then
                      add a digit structure
                else if last is not white space then
                      add a special character structure
    else the number of character equals 1 then
        if letter then
              add a letter structure
        else if digit then
              add a digit structure
        else then
              add a special character structure

    return new regular expression value
```

Each new regular expression gets stored in the list, this means that the list is no longer empty. When the next input value is given it will be matched against the regular expressions stored in the list, see Listing 5.4. If there is a match it means that the input value is of the same pattern as one of the previous input values. The regular expressions in this step are very general so that the length of each input value can be different as long as the pattern itself are the same. The use of "++" after each expression/structure make this possible.

The resong for that every regular expressions created are stored as a string in this list, is that we use the string value to split on to create a new more specific regular expression in the second step of this algorithm. If the method that checks if a new input value match a already created regular expression uses this raw string to compare, the performance drops. Since the algorithm needs to compile this expression every time it checks. This is done for every regular expression stored in the list until it find a match or the list is empty. The performance can be optimized by compiling the pattern once and then stored it in a new list *ArrayList<Pattern>*. Then iterate through the this list and use the *Pattern Object*[7] representing the regular expression to match a input value. This way the performance increases drastically.

To give a better analysis of the discovered patterns, a list that contains a counter representing each pattern is implemented. This will give each pattern a weight that tell how the distribution is. Since the property of *interface List<E>* [8] is that it preserves the order of the elements, the same index for multiple list can refer to the same word. In this algorithm the index of the list containing the counters refers to the regular expression with the same index in the regular expression list. When the *for loop* runs through the list of regular expressions and finds a match, the index where the match happened is used to increment the counter value representing that pattern.

5.2.2 Step Two, Specific Regular Expression

It is here it makes sense that we stored the regular expression as *Pattern* object in the first step. Since we are using them once more in this step. This step is all about creating a more specific regular expression based on the input values used in the step one. Since the same input values is used, one of the regular expressions made previously will match the input. When a match is found the index and the value is given as a input to a method that will analysis this further. The reason for this method takes the index as a input, is that it uses this index to extract the correct regular expression from the list containing the string representation of the regular expressions. Based on this value the range for each regular expression structure is discovered and stored in a HashTable. Hashtable consist of key-value pair, where the key is the index and the value is the range. This way the correct range result is stored to the corresponding regular expressions in the list. Since a HashtTable by definition only can have one value per key, this is fix by saving the minimum and maximum range value for each regular expression structure as one value.

The design for this algorithm is that it checks character by character and count the length of how many character there is of the same type consecutively. The logic of this algorithm is based on the same logic used in step one, except that in this scenario the focus is on the range of each type. When there is a change in character type the algorithm add the value of the counter that counts the number of time that the character type was consecutively, combined with separator to a string value. The separator is "-". Then the counter is reset after each time it detects a character type change. This string value will store the range found for each structure in the regular expression. This means that for every value added to the string value will represent one of the regular expression structure.

Each time the analysis of the word is finish the minimum and maximum range value for that particularly regular expression needs to be updated, see Listing5.8. Since a HashTable will override the value when a new input value is given with the same key. The HashTable will only be as big as the List with regular expression are.

If HashTable's own get() method is called with a given key and there is no value stored it will return "null", see listing 5.7. This property is used to check if it is the first time this key is used and if that is the case the string is added as the value with this key. Otherwise it is necessary to check by using the technique in Listing5.8 to see if the new range should be used to update the minimum value or maximum value for that pattern.

Listing 5.7: The logic of updating the range extracted from FindPatternInformation.java

```
    if(regexpRange.get(index) == null){
      regexpRange.put(index, newRegexpRange);

    }else{
      String newValue = "";
      String[] tempRegexp =
         regexpRange.get(index).split(separator);
      String[] tempNewRegexpRange =
         newRegexpRange.split(separator);
      if(tempRegexp.length != tempNewRegexpRange.length){
        System.out.println("error");
      }

      for(int i = 0; i< tempNewRegexpRange.length; i++){
        String value = updateMinMaxValues.getNewValue(
           tempRegexp[i].split(","),
           tempNewRegexpRange[i]);
        newValue += value + separator;
      }
      regexpRange.put(index, newValue);
    }
```

To be able to update the stored value in the HashTable with the new value, it is necessary to split them. The split done with regards to the *separator* variable with the "-" symbol, see listing5.7. Both the new value and the HashTable value is created by using the same separator symbol "-". The split method returns a array that is used to figure out if the new value is a minimum or maximum, or just somewhere in between. The "," symbol is used to represent the range for each regular expression structure in the HashTable. The method in Listing 5.7 is used to achieve this. e.g. *2,4-5,6*. The first character type has the range from 2 to 4 and the other has a range from 5 to 6. The *newRegexpRange* variable contains range values for each regular expression structures that denotes to a new input value. The HashTable representing the range for a pattern has two value for each regular expression structures. This means that after we have called the split function on the value from the HashTable we need to split once more on the "," to be able to calculate if we need to update the old values stored in the HashTable. see Listing 5.8 and Listing5.7

Listing 5.8: Finding the minimum or maximum extracted from UpdateMinMaxValues.java

```java
public String getNewValue(String[] symbol, String
    spesific){
  long low;
  long high;
  long tempNew = Long.parseLong(spesific);
  long tempLow = -1;
  long tempHigh = -1;
  String res = "";
  String rangeInRegexp = ",";
  if(symbol.length == 1){
    low= Long.parseLong(symbol[0]);
    if(low < tempNew){
      tempHigh = tempNew;
      tempLow = low;
      res = tempLow + rangeInRegexp + tempHigh;
    }else if(low > tempNew){
      tempHigh = low;
      tempLow = tempNew;
      res = tempLow + rangeInRegexp + tempHigh;
    }else{
      res = Long.toString(low);
    }
  }else{
    low = Long.parseLong(symbol[0]);
    high = Long.parseLong(symbol[1]);
    if(low > tempNew){
      tempLow = tempNew;
    }else{
      tempLow = low;
    }

    if(high < tempNew){
      tempHigh = tempNew;
    }else{
      tempHigh = high;
    }
    res = tempLow + rangeInRegexp + tempHigh;
  }

  return res;
}
```

When every value in the data set are analysed it is time to create new regular expression based on the range found for each of the general pattern. The *createNewRegexp* method needs the list of the general regular expression as an input, see Listing 5.9. Since every structure in the regular expression is written with a "++" at the end, this value is used to split the general regular expression into the structures. For each structure the corresponding range is added to it with the "++" symbol at the end. Since syntax for range in regular expression in java is "," this was used when we created the range, see Listing 5.8. This means that we don not need to do extra work to get the correct syntax.

Listing 5.9: Updating the regular expression with the range added to it extracted from FindPatternInformation.java

```java
      for(int j = 0; j<tempRange.length; j++){
        regexp += tempGeneral[j];
        regexp += "{" + tempRange[j] + "}+" ;
      }
      regexp += tempGeneral[tempGeneral.length - 1];
      newRegexp.add(regexp);
      regexp = "";
    }
    return newRegexp;
  }
}
```

5.2.3 Extracting Text Information

This algorithm was design to extract information that could be used when classifying input values. In this thesis this algorithm is not used, but is implemented regarding further work. This method can be used after the general expressions are discovered. The input needed are a string value and index representing the pattern. See section 7.3 for more information about how this extracted information could be used.

The logic in this algorithm is based on the algorithm used in the previous steps. There have been some changes to be able to get the desire results. It is important that we do some more checking on the input value. Instead of just checking if the next character is of another type than the current, it also checks if the next one are of the same type as the current one. As long as the current one and the next are of the same type a counter (ref: valueStop) is incremented by one for each time this is true. When this check is false it means that the current one and the next one is of different types. Two variable called *valueStop* and *valueStart* are used to collect the substring of the input value. Every time a new character type is found the *valueStart* and *valueStop* gets updated with the a new start position. The value of the substring is added to the variable called *inputValue* with the same separator used in step two, "-". There will be same number of structure in the regular expression as there will be "-" in the string representing the values collected. The reason for this is that the check for if the next character is not of the same type as the current one, refers to the change in structure in the regular expression.

The same technique used in step two is used to extract digit information once more in this algorithm, see Listing 5.8. Except that there is a new method is called *checkTextValues()*, see Listing 5.10. The reason for this is that letter values are skipped, so that the same method can be used without any changes. Each letter values is compared against the previous discovred letter value. Only unique letter values is stored. Information about the special character is not stored, since this is not considered important in this case.

Listing 5.10: checkTextValues logic extracted from ExtractInformation.java

```
private String checkTextValues(String[] text, String
   value){
  String res = "";
  int count = 0;

  if(Character.isDigit(text[0].charAt(0))){
    res  = updateMinMaxValues.getNewValue(text, value);
  }else{
    for(int i = 0; i < text.length; i++){
      if(!text[i].equals(value)){
        count ++;
      }
      res += text[i] + ",";
    }
    if(count == text.length){
      res += value + ",";
    }
  }
  return res;
}
```

5.3 Detection and Correction

This algorithm starts by normalizing every unique value. This is done by iterate through the unique word list and append the normalize value to a new list. After the unique values are normalized there can be some duplicated values stored in the new list. A hashtable structure is used to update the distribution regarding the new normalized values. This hashtable is used when two values has the same levenshtein distance for a suspected value. The value with the highest distributions is used.

The processes of finding the closes match between a suspected values and a correct values consist of several steps. These steps are done for each suspected value. The first step is to start with the first value stored in the normalized values list. The levenshtein distance is computed between this value and the suspected value. Only if the distance is lower than the previous lowest distance and if it has a higher distribution seen so far, the index for this value is stored. To make sure that the closest value is not a value from another suspected regular expression. Every suspected regular expression is compared against the closest value.

After the correct value with the highest degree of similarity is found. The next step is to see if the distance is greater than the length of the suspected value divided by two. This is done to make sure that the relation between the distance and the length of the word are big enough. Based on the smallest length of the correct value and the suspected value it compares each character. If the mismatch is a special character this is corrected. Whenever a correction is done it start the loop over again. Each time it updates if possible the suspected value. For every correction that is done a variable called *numberOfCorrection* is incremented. If *numberOfCorrection* got the same value as the distance it means that the algorithm correct every missmatch and the new value is stored.

It is important that the *wordCounter* and *uniqueWords* get updated. The *wordCounter* contains the distribution for the original list of unique words. If the suspected regular expression only has one value and this one get corrected. This regular expression is deleted and the regular expression that matches the corrected value increments the distribution value. If the suspected regular expression has mulitple values where some get corrected it will only decrement the distribution value.

5.4 ID

The first step is to find the majority regular expression. This is done by looping over the list containing the regular expression counters. This list shows the distribution for each regular expression. When a new regular expression has higher counter values than the previous highest counter value seen, a variable called *majority* is updated with the index for this new regular expression. When the for loop is done the index for the regular expression with the highest representation is found.

Now the regular expression with the index value of *majority* is extracted from the list containing the regular expression with the range. This regular expression is strip down to only the regular expression structures. It is necessary to remove the leading (^) and trailing symbol ($). Since when a split is done regarding to this regular expression structure *"++"* the first structure and the last structure from the split will contained one of these symbols. Every structures still contains the range. The reason for the regular expressions containing the range are used, is that these contains the most updated information. The range is not needed and will conflict when trying to match with the regular expression structure symbol for a digit. Therefore another split is done to get the desire result, see Listing5.11

Listing 5.11: How to split regular expression logic extracted from SequentialIDRecognition.java

```
private void stripRegexpStructure(){
  regExpWithRange.replaceItem(
     regExpWithRange.getItem(majority).substring( 1,
     regExpWithRange.getItem(majority).length()-1),
     majority);
  regexpStructureWithRange =
     regExpWithRange.getItem(majority).split("\\++");
  regexpStructureWithoutRange = new
     String[regexpStructureWithRange.length];
  indexDigitRegexp = new ArrayList<>();

  for(int k = 0; k < regexpStructureWithRange.length;
     k++){
    String[] value =
       regexpStructureWithRange[k].split("[{\\d}]");
    regexpStructureWithoutRange[k] = value[0];
  }
}
```

The desire result is a array with only the general regular expression structures stored. This array is loop over and analysed to see if the digit structure is found. For each digit structure discovered the position are stored. The next step is to figure out which position is the most likely ID field. The list containing the position for each digit structure is iterated through. For each position the value are stored in a *HashSet*. This means that only unique values are stored. Each *HashSet* is stored in a array. This is makes it possible to add the correct value for the correct position. When every word that match the majority regular expression are analyses. The position that has the biggest *HashSet* is stored. All the unique values are stored in sorted *ArraList<Long>*. This is done so it is possible to tell if the values extracted are in a sequence with each other. To begin with the sequence distance is one. And for each time this is true, it increments by one. This is because the first value is used to compare with as long as the sequence is correct. When the sequence is of of by one, the next value where the break happened is used. The same process is performed once again with the new value. Whenever the sequence is correct a counter is incremented. When the iteration of the *ArrayList* is done, the counter is compared against the size of the *ArrayList*. This score is used to conclude if the input value can be an ID type or not. Other measurement that is taken into consideration are. The number of values that match the majority pattern, number of unique values regarding the pattern, number of sequence compared with the size of the *ArrayList*, and number of duplicated values regarding the pattern. If the last measurement is low it does not consider it. This is because if there are few duplicated value it is a good indication. Therefor by not using this low value the result is more correct. And when this measurement is high is does not consider the number of unique value compared regarding the pattern. This is since some of these measurement is contradictory with each other. The threshold for classifying a input value as an ID, needs to be tuned regarding the desire performance.

5.5 Decision Tree

5.5.1 Construction of Decision Tree

The Weka library uses entropy gain to calculate what attribute values to split on. The attribute with the lowest degree of impurity is chosen. To measure the impurity for a attribute information gain is computed:

$$\mathit{Gain} = \mathit{Entropy(p)} - \sum_{i=1}^{m} \frac{N(v_i)}{N} Entropy(v_i) \tag{5.1}$$

Where m is the number of attributes, N is the number of elements for the parent node. $N(v_i)$ is the number of element regarding the child node v_i. The entropy for two classes(C=0, C=1):

$$\mathit{Entropy(t)} = - P(C=0|t)\log_2 P(C=0|t) - P(C=1|t)\log_2 P(C=1|t) \tag{5.2}$$

To compute the information gain for each split. The entropy before and after a split is calculated. The process starts at the top of the tree and recursively works it way down to the leaf node. It can stop earlier depending on how the decision tree algorithm is designed. In each step the attribute with the highest information gain is chosen, e.g.

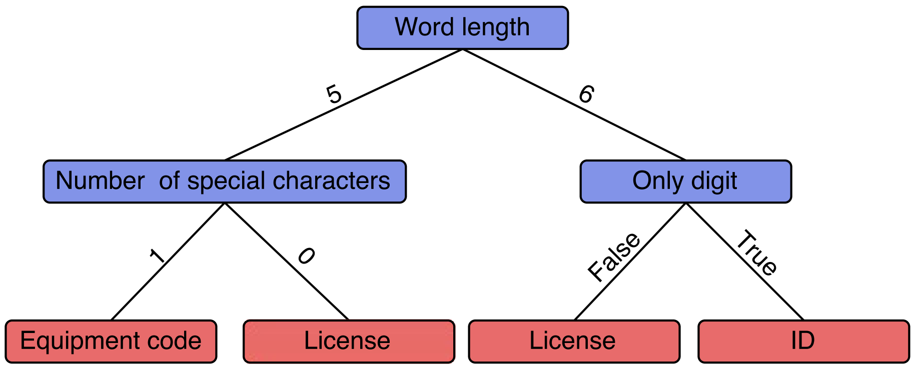


Figure 5.1: Decision tree structure relevant for this thesis. The leaf nodes represent different code types.

5.5.2 Attribute Relation File Format

The training set value used in the decision tree algorithm is create from a text file with all the different code types and values stored. In this file each code label begins with "=_=" followed by the name of the code, and each line below are a code value. This symbol is used to differentiate between a class label and a input value. The first time this code runs it will detect the "=_=" symbol and attach this label to each line of code until a new "=_=" is detected. This algorithm read line by line from file, see Listing 5.12.

Listing 5.12: Short example of the training set file used in this thesis.

```
=_=Cofely Fabricom Work Order Number
370003
370004

=_=ENGIE Norway SAP WBS Code
O.1GJOA.D.IV.521013
O.VEGAR.D.IV.521016

=_=ENGIE Norway COMOS PO Number
PO158760
PO161336
```

The WEKA library that provides the decision tree algorithm need to have the data in a specific format. This format is called *Attribute-Relation File Format* (ARFF). In this format the *Header* contains the attributes and their types. The data is placed after @data, each line below this represents a code value with its class label at the end of the line, see Listing 5.13.

A small program converts the training set file into ARFF file.The setup of the training set file is displayed in Listing 5.13. The pattern found for each code is found by using the same regular expression technique used mention earlier in this thesis. Since this file contains most likely duplicated values it need to keep track on the pattern and the associated word length found so that it will only add new pattern or a pattern that is the same with different word length.

Listing 5.13: Short ARFF example

```
@relation code

@attribute wordLength {zero,one,two,three,four,five,six}
@attribute regexpStruct {zero,one,two,three}
@attribute onlyLetters {TRUE,FALSE}
@attribute onlyDigit {TRUE,FALSE}
@attribute lettersAndDigit {TRUE,FALSE}
@attribute class {'Cofely Fabricom Work Order Number'}

@data
six,one,FALSE,TRUE,FALSE,'Cofely Fabricom Work Order Number'
```

6. Testing, Result and Analysis

In this chapter the performance of the trie structure, pattern discovery algorithm, and decision tree algorithm are analysed. The specification of the computer and software used are: MachBook Air, OS X El Capitan, 1,7 GHz Intel Core i5, 4 GB 1333 MHz DDR3 and Eclipse IDE Mars 4.5.1.

6.1 Trie

6.1.1 Testing

The interest for this test was to compare the trie structure with *LinkedHashSet*. This was because *LinkedHashSet* was an alternative technique for storing unique values. Since the memory of the computer, and the user can effect the test results. Each file was tested several times by using a loop. The average performance was calculated and is displayed in the Table 6.1.

6.1.2 Results

N	P	Trie structure	LinkedHashSet
2948	2912	13.326 ms	2.186 ms
118	17	1.095 ms	0.159 ms
12919	7920	31.122 ms	3.420 ms
198	118	1.078 ms	0.060 ms
3661	886	11.434 ms	0.609 ms

Table 6.1: Where N is number of elements and P is number of unique elements

6.1.3 Analysis

The performance of the Trie is displayed in Table 6.1. The performance was good. In theory the trie structure performs really good O(n), where n is the length of the word. The part that can slow it down is how the trie was implemented. In this thesis the trie was implemented by using hashMap. Where the key was the character value and the value was a *TrieNode*. This mean that for each character that was inserted each hashMap in the path needed to compare the hash values. Meanwhile with the use of *LinkedHastSet* only one check was needed for each word.

It is clear from the result that *LinkedHastSet* has the best performance. The relation between the trie structure and *LinkedHastSet* is divided by a divisor of 10.

6.2 Regular Expression

6.2.1 Testing

The analysis the performance for each regular expression quantifiers were done by using several different files. Each file contained values that had different types of patterns. Some files had only a few patterns, while other had many patterns. The time it took for each file to create all the regular expressions were tested several times. The testing was done by using a loop and storing the time it took to analyse the same list of unique words each time. This test was repeated 40 times. This was done to remove the possibility for that a human could effect the result. When the loop was done the average was calculated and the result is presented in Table 6.2

6.2.2 Result - Regular Expression Quantifiers

N	P	Greedy	Reluctant	Possessive
1010	50	47 ms	46 ms	30 ms
3661	26	15 ms	16 ms	10 ms
12919	62	74 ms	73 ms	59 ms

Table 6.2: Where N is number of lines and P is number of patterns

6.2.3 Analysis

The result displayed in Table 6.2 shows that possessive quantifier was the best quantifier to use in this thesis. The Greedy and the Reluctant quantifier had nearly the same performance. The reason for that possessive quantifier had the best performance is that it does not allow backtracking.

6.3 Decision Tree

6.3.1 Testing

Several of the decision tree algorithms mention in this thesis were tested. The test was done to see what technique that had the best performance and to figure what technique that was the best to use in this thesis. Even if a model had the best cross-validation evaluation, it did not mean that this model was the best performing model to use to classify the input values. Every model was test to see how it performed on some local test samples. There were 25 different code files used to test the model against.

6.3.2 Result

Measurement	*Score*
Correctly classified instances	81.717 %
Incorrectly classified instances	18.283%
Kappa statistic	0.804
Mean absolute error	0.014
Root mean squared error	0.084
Relative absolute error	17.369 %
Root relatice squared error	44.368 %

Table 6.3: Duplication, subsampling and cross-validation.

In Table 6.3 the technique used was to duplicate every instance in the training set. The second step was to use sub-sampling to minimize the relation between each class label type. The final step was to use cross-validation to build the decision tree model.

Measurement	*Score*
Correctly classified instances	89.232 %
Incorrectly classified instances	10.768%
Kappa statistic	0.884
Mean absolute error	0.0086
Root mean squared error	0.066
Relative absolute error	11.120 %
Root relatice squared error	33.427 %

Table 6.4: Duplication, subsampling, duplication and cross-validation.

In Table 6.4 the technique used was to duplicate every instance in the training set. The second step was to use sub-sampling to minimize the relation between each class label type. The third step was to increase the training set by duplicate the sub-sampling set. The final step was to use cross-validation to build the decision tree model.

Measurement	*Score*
Correctly classified instances	81.919 %
Incorrectly classified instances	18.080%
Kappa statistic	0.806
Mean absolute error	0.014
Root mean squared error	0.092
Relative absolute error	18.234 %
Root relatice squared error	46.714 %

Table 6.5: Duplication and subsampling.

In Table 6.5 the technique used was to duplicate the instances and then use sub-sampling. The sub-sample set was used to build the decision tree model.

Measurement	*Score*
Correctly classified instances	93.655 %
Incorrectly classified instances	6.345%
Kappa statistic	0.9103
Mean absolute error	0.005
Root mean squared error	0.051
Relative absolute error	8.245 %
Root relatice squared error	29.726 %

Table 6.6: Duplication and cross-validation.

In table 6.6 the technique used was to duplicate every instance to increase the training set. After this a cross-validation was performed.

6.3.3 Analysis

The main parameters that were used during the decision of what technique to use were *Correctly classified instances* and the score from a manual test. Other measurements included *kappa statistic* provided valuable information. The Kappa statistic compares the observed accuracy against expected accuracy. The measures for the predictions of the agreement is from 0-1. Where 1 represented perfect agreement. This statistic takes into account that sometimes observations can agree or disagree by chance[33][34]. Every technique provided good kappa statistic. The technique described in Table 6.6 had the best kappa statistic result.

The problem regarding the technique used in Table 6.3 was that the training set became to small. This meant that the model misclassified more than the other models build, when it was tested on the data manually. The idea by using the technique used in Table 6.4 was to increase the training set first. Then make the training set less imbalanced. The goal was to make a model that was more accurate and better to classify. Different number of duplications were used tested. Up to some point it increased the performance. The same duplication technique was used in Table 6.5. The duplication increased the number of instance both for the majority class instances and the minority class instances. This was not a problem since when the sub-sampling technique was used the relation between the majority and the minority were at maximum 10:1. The model build from this sub-sample had good performance. This model was tested manually and had the best classification performance with a success rate of 80%. The technique used in Table 6.6 builds the model that had the best cross-validation evaluation. This model had the second best classification performance when it was tested manually, with a success rate of 72%. The problem was that the model was trained on a training set that was imbalanced.

The decision tree was not pruned after it was created. Since a unpruned model had the best cross-validation evaluation.

The classification technique that gave the overall best performance was the technique used in Table 6.5. This technique provided a satisfying *Correctly classified instances* score, along with the best manual classification score. When a misclassification appeared, the correct class label was given as one of the other possible result. But since the representation was lower it was not used. By using the decision tree as a step to limit the number of possible class label, the process of classifying a file becomes easier. This is since the decision reduced the number of value drastically. The time it took to classify each word was good since the decision tree was shallow.

6.4 Describe the Data that you use for Testing

The data used to test the algorithm in this thesis was structured data belonging to real world companies. This was information that are confidential. The data used to build the decision tree was collect from these files. These files contained different information including names, deadlines, companies, job positions, codes, ID, charts and so on. The algorithm presented in this thesis was only concerned to code and ID column, therefore column that contained other information was not of interest. Each column of interest was extracted out and saved in a text file.

6.5 Pattern Discovery

When a unknown file was presented the information about, what type of code it was, the consistency, patterns and if it was an ID, is not known. Figure 6.1 shows files that went from a large number of code values to a small set of patterns. Each line in represent a file. The first iteration represent the number of input values. The second iteration represents the number of unique values extracted from the trie. The third iteration represent if some code value had been corrected. The fourth iteration represent the general regular expression discovered from the input file. The last iteration was the number of specific regular expression that were made. This was the final conclusion about what pattern that were discovered. The purpose of this chart was to show that each file used in this test can be describe with a low number of patterns. The maximum number of pattern display in the chart was 5. The pattern discovered described the patterns in a file in a compact and informative way.

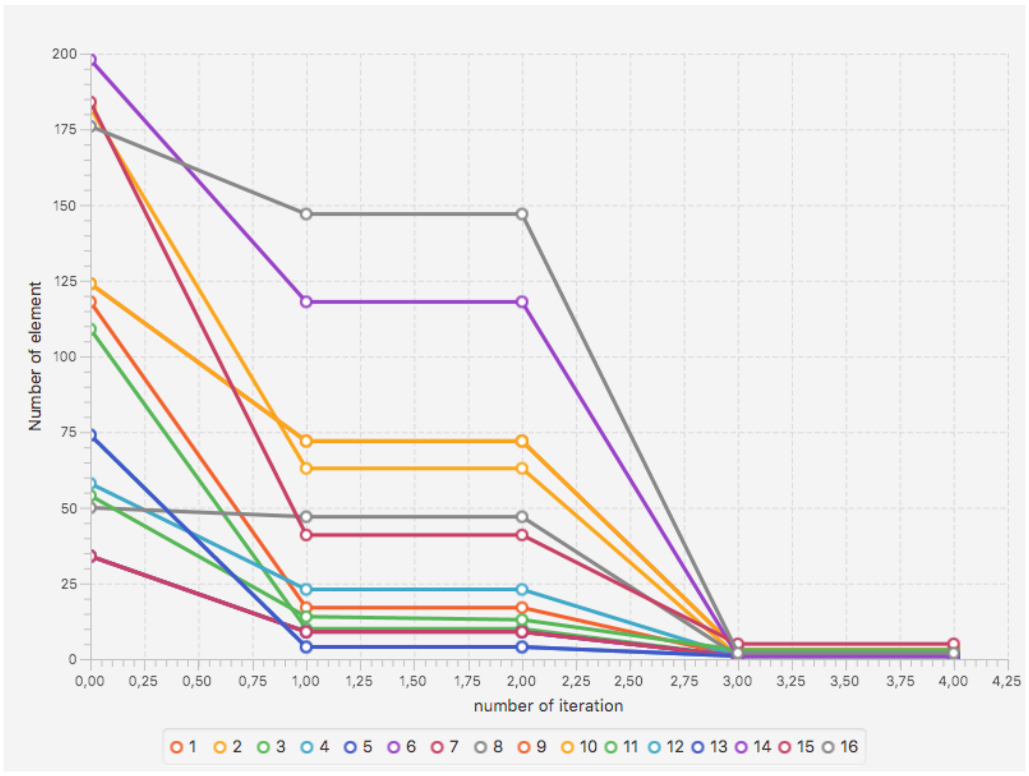


Figure 6.1: Chart displaying the result of the analysed files. Regarding iteration 0-2; Number of elements is number of rows in a structured data. Regarding iteration 3-4; Number of elements is number of patterns.

Blank page.

7. Discussion

The opportunity to discover pattern in textual document that are not known for the user, will increase the quality of information retrieved from the textual documents. In this scenario it can tell how consistent the content is in a structured text document and the distribution of patterns just to mention some benefits.

As the flowchart in section 3.1 indicates the decision tree is build based on the general regular expressions and not the specific regular expressions. There are several reason for this. The first reason is that the detection and correction is performed before the general regular expressions are used. This means that they are up to date. To build the decision tree there is no need for the range. If the specific regular expressions had been used, more preprocessing would be needed before the model could be build.

The result from the test done in subsection 6.1.1 is interesting. If the objective is extract just unique word *LinkedhashSet* is more suited for that. In this thesis the possibility to expand the information stored in trie is a important property. Even if this structure is slower than *LinkedhashSet* it is a more convenient data structure to use to solve the problem presented in this thesis.

7.1 Originality

Trie structure is a familiar technique applied when a pattern search is performed. The main difference is that these patterns a known. The technique proposed in this thesis creates these patterns continuously. It is common to use regular expression to represent a pattern. Since regular expression provides fast and powerful search performance. The algorithm that concludes if a value is an ID type is not based on a familiar technique but several analysis that look for a desire properties. The algorithm that detects typographical errors and corrects them is inspired by levenshtein distance. It uses the distribution of the regular expression and the result from the trie to do the analysis. The two algorithm is design to provided the desire behaviour.

When classifying input values it is common to use the character to build a decision tree model. In this thesis the decision tree model is build based on the structure of the patterns discovered. Instead of concluding with the result provided by the decision tree. Further analysis as mention in the section 7.3, should be done to increase the overall accuracy of merging structured data.

7.2 Limitations

The regular expression does not distinguish between lower case or upper case, since the requirement is to show the pattern. And by the definition made in this thesis upper-case and lower-case representation for the same letter is considered to be the same "pattern". If there would be a typographical error with lower-case letter when all the other instances is written with upper-case. This error means that a new regular expression would be created.

It can be considered a limitation that all the different special character in this thesis is displayed with the same general regular expression structure.

The decision tree performance good regarding the limitation it has. The limitations concerned is that it is based on the regular expression structure and pattern information and not word and range of the number. Multiple code got so similar pattern that they can not be distinguished between. This can be analyse further by using the information stored in the leaf node in the decision tree. The leaf node got information about the distribution for each class label. This means that the result from the decision tree can be used with analysis of the word and range of the number to find out what type of code it really is. Or if this technique is used together with the whole excel sheet it is possible to use the contextual information from the columns.

Sometimes a row value consist of the same code-pattern multiple times after each other. This can be hard to detect. But by using the the other regular expression found it could be possible to try get a match. The idea is that if there is a match somewhere in the subtext this can be a sign that there are multiple code value in one line. Even if there is a match this match can be wrong. Since if the pattern is quite general it will more easily be a match lines where it only is one code written. By using the length of the words associated with their pattern it can tell if a new row value could match a previous value. The technique would be to see if it is a multiplication of the length by a factor x. If this was detected correctly each duplicate code value should become a new row value. Looking at the whole excel sheet each column would then need to insert at the correct row index a new value based on the correction done for one column.

7.3 Further Work

The more analysis and correction of typographical error that is done, the less noise will be presented as a solution. Since if there missing a letter, this will most likely create not match the other patterns. Therefor by create a algorithm that could detect missing letter and then analyse to see what letter it most likely should be inserted. Another function that would be increase functionality of the algorithm is when there is a missing value in the input file. Instead of skipping that line, be able to fill that row with the most correct value possible. If it is an ID type the highest ID value found plus one is the new value inserted into the missing row. If it is not a ID the value most represented could be inserted, or based on the neighbors values a new value can be added into the missing position.

The result from the decision tree could be used to make a better classification statement. If the leaf node contains multiple possible class label it should be analysed further to see what value that is the most correct. This can be done by looking on the context of the structured data, or by looking at the range and characters. This is a more specific analyses that can use the algorithm presented in subsection 5.2.3.

Blank page.

8. Conclusion

The approach presented in this thesis discovers and extracts patterns without any pre-knowledge needed. By combining pre-knowledge more specific information can be extracted. This information is gained by using a decision tree algorithm. The use of trie structure to extract unique words provides the possibility to collect much more information that other data structure. The regular expression design discover patterns with high achievement. and the regular expression algorithm collects important information about the repetition for each pattern. This information increases the understanding about what the distribution is for each pattern. Typographical errors are investigated and correct if the criteria is met. Further analysis is performed to see if the input value can be an ID type. This analysis combined with the patterns and the decision tree increases the information retrieved from a file. Based on these techniques the pattern and information about what value the file contains can be made. The algorithm presented in this thesis discovers, corrects typographical errors, and extracts patterns in a effective and compact approach.

Blank page.

Bibliography

[1] *Automatically Generating Regular Expression*, http://www.regexmagic.com/autogenerate.html

[2] *Norwegian Alphabet*, http://www.learnnorwegiannaturally.com/norwegian-language/norwegian-alphabet/

[3] *String Matching Algorithms*, http://www.mif.vu.lt/cs2/courses/ds99fa6.pdf,

[4] Francisco Gómez Martín, *Exact String Pattern Recognition*, Escuela Universitaria de Informática, October 2009 - January 2010.

[5] *Class String*, https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

[6] *Hash Collision*, http://preshing.com/20110504/hash-collision-probabilities/

[7] *Class Pattern*, https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

[8] *Interface List<E>*, https://docs.oracle.com/javase/7/docs/api/java/util/List.html

[9] H. Shang, T.H. Merret, *Tries for Approximate String Matching*[p. 100-145], September 8, 1995.

[10] *Using Tries*, https://www.topcoder.com/community/data-science/data-science-tutorials/using-tries/

[11] *Apache POI*, https://poi.apache.org/

[12] *Mastering Quantifiers*, http://www.rexegg.com/regex-quantifiers.html

[13] *HashSet<E>*, https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html

[14] *8 Tactics to Combat Imbalanced Classes in Your Machine Learning Dataset*, http://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/

[15] *Text Area Swing Component*, https://docs.oracle.com/javase/7/docs/api/javax/swing/JTextArea.html

[16] *Use WEKA Library in your Java Code*, https://weka.wikispaces.com/Use+WEKA+in+your+Java+code

[17] *WEKA Library Homepage*, https://weka.wikispaces.com/

[18] *GNU General Public License*, http://www.gnu.org/licenses/gpl.html

[19] Badr HSSINA, A. Merbouha, H. Ezzikouri, M. Erritali, *A comparative study of decision tree ID3 and C4.5*, in *(IJACSA) International Journal of Advanced Computer Science and Applications, Special Issue on Advances in Vehicular Ad Hoc Networking and Applications*[p.13-19].

[20] *Chi-square Automatic Intereaction Detector (CHAID)*, http://www.statisticssolutions.com/non-parametric-analysis-chaid/

[21] Melanie Mitchell, *Analogy-Making as a Complex Adaptive System*, New York: Oxford University Press, 2001.

[22] Douglas Hofstadter, Melanie Mitchell, *The Copycat Project: A Model of Mental Fluidity and Analogy-making*, Chapter 5, [p.205-267].

[23] Philippe Fournier-Viger, *The Data Mining & Research Blog*, http://data-mining.philippe-fournier-viger.com/introduction-frequent-pattern-mining/

[24] *Apriori*, https://docs.oracle.com/cd/B28359_01/datamine.111/b28129/algo_apriori.htm

[25] Michele Chambers, Thomas W Dinsmore, *Advanced Analytic Methodologies: Driving Business Value with Analytics* Person FT Press, 22. September 2014

[26] Prof. Galit Shmuelo, *Calssification Trees: CART vs CHAID*,Friday April 20, 2007, *Accessed: 19. May 2016*, http://www.bzst.com/2006/10/classification-trees-cart-vs-chaid.html

[27] $WhatisaChi - SquareTest? - Definition\&Example$, http://study.com/academy/lesson/what-is-a-chi-square-test-definition-example.html

[28] Pete McCollum, *An Introduction to Back-Propagation Neural Netwroks Accessed: 1. June 2016*, http://www.seattlerobotics.org/encoder/nov98/neural.html

[29] *Character Class*, https://docs.oracle.com/javase/7/docs/api/java/lang/Character.html

[30] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, Dan Steinberg, *Top 10 algorithms in data mining*, *Springer-Verlag London Limited 2007*, 4.December 2007.

[31] Christopher Potts, *Analysis: Clustering Words by Tags in the SwDA*, LSA Linguistic Institute, 2011. http://compprag.christopherpotts.net/swda-clustering.html

[32] Tan Steinbach Kumar, *Introduction to Data Mining*, Pearson, 2014.

[33] Anthony J. Viera, Joanne M. Garrett, *Understanding Interobserver Agreement: The Kappa Statistic*, in 2005 *Family Medicine* [p 360]. May 2005.

[34] *Weka Data Analysis*, http://www.cs.usfca.edu/~pfrancislyon/courses/640fall2015/WekaDataAnalysis.pdf

[35] *Levenshtein Distance Formula*, https://en.wikipedia.org/wiki/Levenshtein_distance.

[36] *The Levenshtein-Algorithm*, http://www.levenshtein.net/

Blank page.

A. Source Code

The source code (src.7z) is embedded.