Article Duplicates

Brian Lynnerup Pedersen

Kongens Lyngby 2014

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
DK-2800 Kgs. Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The goal of this thesis is create is to document my work with implementing a working prototype of using algorithms to find article duplicates in a large corpus of articles. I will describe how the article inflow is currently working and how Infomedia plans on implementing my work in this inflow.

I will look at various algorithms for text comparison, and look into the possibility of creating my own or tweak an existing algorithm to better suit the needs for this thesis.

Finally I will create a summary of the work done, problems I have come upon and what the future of the project will be.

Summary (Danish)

Målet for denne afhandling er at dokumentere mit arbejde med at finde artikel duplikater, i et større artikel corpus, ved brug af algoritmer. Jeg vil beskrive hvordan Infomedias artikel inflow virker nu, og hvordan Infomedia planlægger at bruge mit arbejde i fremtiden.

Jeg vil undersøge forskellige tekstsammenlignings algoritmer, og undersøge muligheden for at lave min egen algoritme, eller modificere en eksisterende algoritme til at bedre udføre det arbejde jeg laver i denne opgave.

Til sidst vil jeg gennemgå det arbejde jeg har lavet, hvilke problemer jeg løb ind i og hvad fremtiden for projektet vil være.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an B.Eng. in Informatics.
The project is equal to 20 ECTS points.

This thesis is protected by confidentiality. No information from this thesis can be handed off to any party without signed permission.

All excerpts from articles are owned by the respective authors / papers.

The thesis deals with the issue of finding articles that are duplicates in a large corpus of articles. This is done using various algorithms and is implemented in C#.

The thesis consists of ...

Lyngby, 09-June-2014

Not Real

Brian Lynnerup Pedersen

Acknowledgements

I would like to thank my supervisors from DTU, Inge Li Gørtz and Philip Bille, for the help they provided to my project. Also I would like to thank my company Infomedia, for letting me do my project with them, in particular my project leader Klaus Wenzel Jørgensen and Rene Madsen.

Contents

Chapter 1

Introduction

As mentioned above, I have done this thesis for the company I work for, Infomedia[1]. I have been working for them since my fourth semester at DTU (march 2012), while studying as an IT engineer (B.Eng.). Infomedia is in short a company that deals with news monitoring.

Infomedia is the result of a fusion between Berlingske Avisdata and Polinfo in 2002, which means that Infomedia is partly owned by JP/Politikens Hus[2] and Berlingske Media[3]. It is a company with around 150 employees, of which 107 is contract employees and the rest is student aides in the various departments. Infomedia has various departments, which includes an economy, sales, analysis and an IT department amongst others.
I am employed in the IT department (*PIT - Product Innovation and Technology)* as a student programmer (student aide).

Infomedia deals with news monitoring, which means that we have an inflow of articles[4] from various newspapers, news sites, television and radio media, which we then monitor for content that is of interest to our clients. This can be a client that wishes to know when their firm is mentioned in the press or

[1] `www.infomedia.dk`
[2] `www.jppol.dk`
[3] `www.berlingskemedia.dk`
[4] Articles are sent to Infomedia daily, this can be more than 40,000 articles per day.

a product they are using, if that is being mentioned. Currently (while writing this thesis) the upcoming EP[5] election is going on, and politicians are using the monitoring service from Infomedia to track how they are doing in the media Infomedia[Alb14]. Infomedia had a free for all news monitoring of the EP election, meaning that all interested could sign up for a getting a daily (weekdays) news monitoring mail from Infomedia, containing the top stories about the EP election. Amongst other things a candidate visibility monitoring was created[Inf14]. Infomedia have also begun monitoring social media. Infomedia sells various solutions to clients, so they can get the kind of media monitoring they want.

One of the things that Infomedia tries to do, is that we want to present our clients with a fast overview of the articles in which terms[6], that trigger our news monitoring, appear. Many local newspapers are today owned by bigger media houses (like the owners of Infomedia) and as such, they will feature a lot of the articles that have also been printed in the "mother paper". This will make the same (or roughly the same[7]) article appear many times in news monitoring. In an effort to make the list of articles presented to the clients, easy to look at, and preventing a client having to read the "same" article many times, Infomedia has a wish to cluster article duplicates. Infomedia can then present the client with a list of articles and in that list have further sub lists that contains duplicates of the original aritcle[8]. This also have an economic factor as clients are charged per article read.

Another issue, is the issue of copyrights and when the same article will appear in different media, but without content given from the author of that article. An example that is often happening is that news telegrams from Reuters[9] or Ritzau[10] is published in a newspaper, but without the source indication. All news media are of course interested in knowing when their material is being published in competing media. This how ever can be tricky business, as official rules on the matter is incredible fuzzy.

[5]EP - European Parliament.

[6]A term is, in short, a word or a combination of words. For the rest of my thesis a term will however only be a single word.

[7]Articles can be slightly edited in order to make them fit into the layout of the various papers.

[8]Or the longest article rather, as this will tend to contain the most information.

[9]`www.reuters.com`

[10]`www.ritzau.dk`

1.1 Thesis Statement

I will in this thesis try and look into various ways of identifying article duplicates (or articles that have a lot of text in common) within a test corpus[11] of articles, by using algorithms. The long term goal for Infomedia is having this being implemented in the inflow of articles, and having a look back functionality so that we can group duplicates not just for one day, but for a longer period of time.

1.2 Limitation

Due to the time constraints, this will be done as prototype. I will show through testing how the algorithms works, and analyse their results. These results will then be evaluated and I will comment on these findings.

[11] A days worth of articles from 10/31/2013 - totalling 22.787 articles.

Chapter 2

General - Terms and Rules

I will in this chapter cover the essentials of the expressions and terms used throughout this thesis.

2.1 Terms

As there is a lot of terms used in this thesis, a short introduction to the most used are in order.

- **Article:** For this thesis, a digital document containing the contents of a piece of news. Could originate from papers, magazines, TV or other forms of media. For this thesis a document corresponds to an article. Articles are in their electronic form stored at Infomedia as XML files, I will throughout this thesis only deal with the part of the XML files that contains data of value to me in this assignment. This being *Tags* (see below), *Headline*, *Sub headline* and *Article Text*.

- **Corpus:** From Latin meaning *body*. In this thesis that describes the test set of articles being used a test set throughout my thesis.

- **Monitoring:** In relation to the news monitoring (news surveillance) that Infomedia does, is the act of collecting news that holds information of value to our customers.
- **Tag:** Used in Ontology[1] to create words that describes the contents of an article.
- **Term:** Basically a word. A term will be something that can be searched for.

2.2 Matching

I will in this thesis talk about false and true positives and negatives. A match will mean that two articles to some extend have the same content.

- **False Positive:** When an algorithm wrongfully identifies two articles as a match.
- **False Negative:** When an algorithm wrongfully identifies two articles as not being a match, when in fact they are a match.
- **True Positive:** When an algorithm correctly identifies two articles as a match.
- **True Negative:** When an algorithm correctly identifies two articles as not being a match.

2.3 Duplicates

In this thesis I will often use the term 'duplicates' or 'match' about article comparisons. A duplication (match) can be an article that has been taken directly from a news feed and posted in a newspaper. Many local newspapers is owned by larger newspapers, and they will often receive articles from their owning paper. They will then print this in their own paper. Sometimes they will only use parts of the article and this will also be considered a duplicate for this thesis. As such duplication in this thesis is a way of describing how similar two articles are, rather than saying different papers are doing conscious fraud. That is a matter for another thesis.

[1] http://en.wikipedia.org/wiki/Ontology_(information_science)

2.3.1 Topic Matching

When looking at article matching, there is also the possibility of having articles score pretty well by the algorithms because they are dealing with the same topic. There are cases where the article have been heavily modified, and then there would be no basis to talk about duplication, then one could talk about topic matching. The article no longer contains the same phrases, but deals with the same topic. Of course two articles could describe the same topic, but never have been related to begin with. I will not try and dissect whether this is the case, only try and indicate when I find two articles that are dealing with the same topic, and mark them as such.

Hvornår er en artikel et duplikat (korte artikler (breaking news), hvornår er to artikler "tilstrækkeligt" forskellige?) blah about copyright rules in Denmark...

CHAPTER 3

Analysis

This chapter describes the considerations taken in picking an algorithm, as well as what is already implemented at Infomedia.

3.1 Algorithms in General

Before any sort of work can be done, one must consider various algorithm to work with. There are several text matching algorithms available for free on the Internet, and if one has the money for it, there are companies that can develop a specialized algorithm for you. As I do not have a lot of money (and paying for someone else, to do an algorithm for me, kind of defeats the purpose of this whole thesis) I have gone with the first option and found a free basic algorithm on the Internet, as well as contemplated to create my own algorithm from scratch.

3.1.1 Requirements Analysis

The algorithms should be able to live up to the following demands:

- The algorithm should be able to work with text.
- Performance is of secondary importance, but should have good accuracy (performance $<$ accuracy).
- The algorithm shall be able to return a score based on how identical two articles are.
- The algorithm should be focus on doing one thing only (not try to do several forms of text comparisons).
- The algorithm shall be available in an open source, or free to use, licence.

3.2 Initial Considerations

Infomedia already have an algorithm implemented in the inflow to make a rough comparison of the articles coming in. How ever, the thought is that a combination of several algorithms would provide a better and more granular view of the articles as they are being compared. A new algorithm should be one that was specialized in text matching. It should also be an algorithm that would work in different manner than what Infomedia already have implemented[1], as having two algorithms that work in more or less the same fashion would not produce results of much interest.

As the current implementation is rather fast, it could prove useful to have the algorithm that is already implemented, do the initial rough split of matches and no matches, and then have a slower (but more thorough) algorithm look at the *interesting* article comparisons. Initially I have looked at two algorithms to fill this need, *Longest Common Substring* and *Semaphore Tag Matching* - an algorithm I would make from scratch.

3.2.1 System Architecture

As mentioned the various algorithms should work in different ways, meaning they should have various ways. This is to ensure a balanced image of how much an article comparison is actually a match. The thought is, that instead of having a few algorithm having to have many focus areas. It is better to have many that only focus on one thing, then combine their scores into a broad representation of how similar two articles is.

[1] More on the algorithm already implemented in the next section.

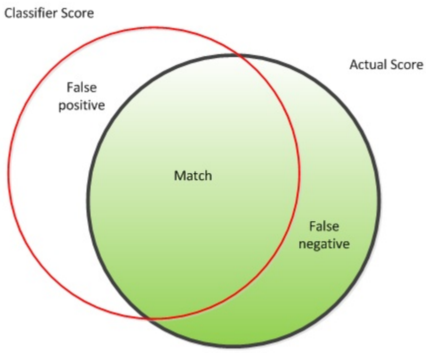

Figure 3.1: Variation between how a single algorithm might score an article comparison, and how the actual score should be[ML14].

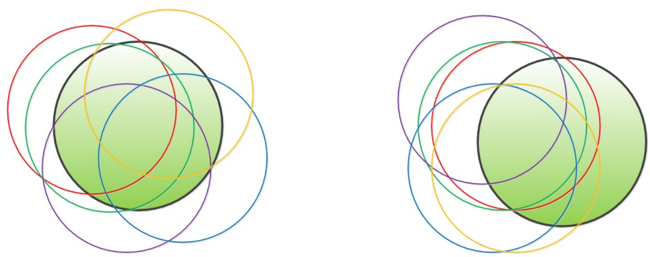

Figure 3.2: **Left:** A balanced score, obtained by using several algorithm with different focus areas. **Right:** An unbalanced score, obtained by using several algorithms with the same focus area[ML14].

The essential thing in this, is to ensure that the various algorithms works in different ways. If one were to use many algorithms that all focused on the same way of doing text comparisons, the results would not provide that broad image of scores that is wanted for a more accurate score representation.

The idea is that the algorithms should do 99.9% of the work in identifying article duplicates, and then have the last 0.1% be verified by humans. This is already how things are working at Infomedia, but only with one algorithm at the moment. My work will the be the first algorithm to do a check of the work done by the already implemented algorithm. This will hopefully narrow the field of possible article duplicates that has to be verified by human eyes.

The system would then ideally work as follows.

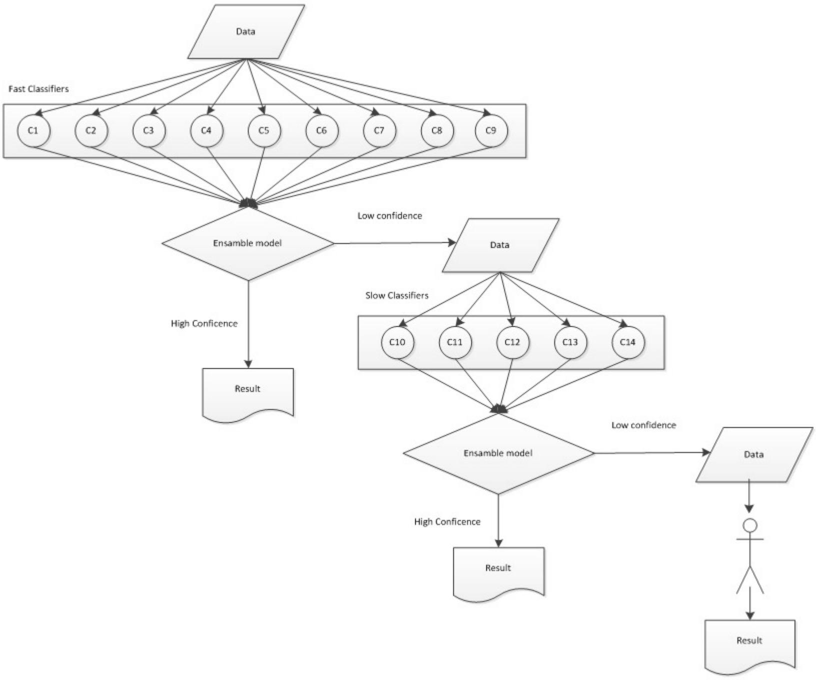

Figure 3.3: System Architecture, with many algorithms doing text analysis[ML14].

When data (articles) arrives in the inflow, a number of fast algorithms would then do the rough split of the comparisons. All of these comparisons would then receive scores, that would then be evaluated through the ensemble model. If the scores leaves no doubt (is over a given threshold) that the article comparison is a match, the comparison would stored as a match. If the scores of the comparisons leaves some degree of doubt as to whether a comparison is a match or not (the scores being between a set of thresholds), the comparisons are then passed on to the slower algorithms, that then in turn would evaluate the comparisons. If these algorithms find that the comparison is a match, it would be stored as such. If there is still some doubt to whether it is a match or not, the articles would then finally be sent for human evaluation. The task of my thesis is to create a secondary algorithm (one of the slower ones) to evaluate the results given from the faster (and already implemented) algorithm.

3.3 Algorithms Used

Term Frequency - Inverse Document Frequency[Wik14b] (Cosine), generates a vector from each document. **I will not cover this algorithm in detail, as I have not done any work on it, only used some of the methods in it for my thesis.** This is the algorithm already implemented in the inflow today (a modified version of it, that is based on the Vector Space Model[2]). Each word in every article is added to a *word map* which contains all the words of all articles in the corpus being checked (which also is used to create the document vector). The word map is used to generate a weight of each word (a word occurring in many articles will have less weight than a word that is only present in a few articles). Each word generates a bit of the articles total vector, a word that occurs in all documents will have a very short vector, a more rare word will have a longer (and therefore weigh heavier) in the article vector.

Once the word map is created, the articles are then scored. The way that this is done, is that the algorithm compares two article's vectors with each other and then returns a score based on the cosine angle between the two article vectors. This is done one article comparison at a time (although done with parallel coding to speed up the process). As the word map is generated each time the algorithm is run, the word map can (and probably will) differ from each run (if the corpus of articles are being changed). Infomedia is therefore talking about implementing this bit differently, and building a constant word map, that only gets updated with each run, not overwritten. This will ensure that common words will always have a short vector.

The algorithm then returns a list of article comparisons (based on a threshold set by the user), with article ID[3] and scores. So each comparison has the ID of *"article 1"* and *"article 2"* and their score (the angle between the two vectors, in a multidimensional universe). The closer the score is to the value 1.0 the more similar are two articles. A score of 0.0 indicates that two articles has nothing in common (according to this algorithm, I will discuss this point in the next paragraph), whereas a score of 1.0 indicates two perfectly identical articles (according to this algorithm). This algorithm and it has a good O-notation (O(log N)).

Problem: This string comparison is very sensitive to articles containing many of the same words, for instance "A man walks his dog in the park" and "A dog walks his man in the park" would result in returning a cosine value of 1.0, due to the way the algorithm works. The two strings are clearly different, but the

[2] `http://en.wikipedia.org/wiki/Vector_space_model`

[3] Each article has their own unique ID.

words in each string is the same, therefore the Cosine algorithm will find them to be identical. This problem is very unlikely to yield false positives.

Longest Common Substring (LCS), compares documents in pairs. A general implementation would be to have a list of documents and then compare a document to every other document in the list. The algorithm will then return the length of the longest common substring. By default the LCS algorithm[Wik14a] checks the contents of a string character by character against another string. When initialized, the algorithm creates an double array and each time a match is found (when two identical characters are found ('a' and 'a' for instance)) the algorithm marks that in the array by adding a number. It then checks if this substring is longer than what has previous been found, if so, it discards the old substring and keeps the newly found.

	s	k	ø	n	t	r	u	m
s	1	0	0	0	0	0	0	0
k	0	2	0	0	0	0	0	0
ø	0	0	3	0	0	0	0	0
n	0	0	0	4	0	0	0	0
n	0	0	0	1	0	0	0	0
e	0	0	0	0	0	0	0	0
r	0	0	0	0	0	1	0	0
u	0	0	0	0	0	0	2	0
m	0	0	0	0	0	0	0	3

Figure 3.4: An example of how two words are compared in LCS. The fields in yellow are the two words broken into characters ('Skøntrum' and 'Skønnerum'). The fields in blue indicates when LCS finds a match, the number indicates the length of the substring. In this case, LCS finds three sub strings:'skøn', 'n' and 'rum', the longest of the three is the first, and this will be the result that LCS returns to the user.

I will use the basic implementation of this algorithm in my thesis, I will then try and modify it to work better in context with finding article duplicates, instead of just the longest common substring.

Problem: This algorithm is very prone to fail in cases where there have been made alterations to the article in question. A word change in the middle of one of two otherwise identical articles will result in a 50 % match. If the article in question have been obfuscated with many changed words, the LCS will be extremely short. This is a high risk problem, as article duplication will often involve changing words. This algorithm is also substantially slower than the Cosine algorithm, having an O-notation of O(n*m). Ideally this algorithm is therefore best used on a selection of articles, rather than on the entire corpus.

3.4 Optimizing Performance

3.4.1 Stop Words

Stop word[4] removal would improve running time (performance) of both algorithms, and would pose little threat of causing either algorithm to fail (finding false positives). The exception to this could be very short articles (like breaking news articles[5]), that only contains common words, like "Man walks away". Depending on the stop word list, this article could end up being *null*[6]. We can safely (with respects to the previously addressed problem) remove stop words, as they don't provide any *semantic*[7] value to the text.

3.4.1.1 Cosine

For the cosine algorithm removal of stop words would improve performance, by reducing the size of the *Magnitude Vector*.

3.4.1.2 LCS

In regards to the LCS algorithm, the performance would also be improved, as the algorithm wouldn't need to traverse as many characters. This would reduce the length of the longest common substring, but it would be unlikely that it would affect the outcome of the algorithm, as stop words are rarely changed when duplicating articles.

3.4.2 Stemming

Stemming[8] Stemming is the act of conjugating a word to the base form (Danish: *'Grundform'*). It is done in order to reduce the amount of noise in a text. Words will then be conjugated and instead of having the same word represented

[4] `http://en.wikipedia.org/wiki/Stop_words`

[5] Breaking News articles is a thing of the 2000s. With the spread of media onto the Internet, an article is no longer a static printed piece of news in a paper. News can be published instantly on the web, and then updated as information are received. How ever, Breaking News articles can still be printed in the paper article with only little text attached to it.

[6] An article with no text data.

[7] `http://en.wikipedia.org/wiki/Semantics`

[8] `http://en.wikipedia.org/wiki/Stemming`

several time in different conjugations they will all be noted as the same word. Stemming does not ruin the semantic content of the text.

3.4.2.1 Cosine

Stemming improves the performance of the cosine algorithm as this will reduce the number of words in the *vector space*. As words would be reduced to their base form (Danish:*Grundform*), words used several times, but with different endings would be counted as the same word. When using weighed evaluation this would actually improve performance to some extend. It can have a slight impact of the Cosine algorithm. This is because that without stemming the same word can occur several times in a text in different conjugations, and thus each conjugation would have a larger vector than when the conjugations are all counted as the same word. This impact will be minimal because the rare words are still occurring less often than normal words (stop words).

3.4.2.2 LCS

Stemming would not improve the performance of the LCS algorithm by much. As this algorithm matches characters one by one, it would make little difference if the words are stemmed or not. As the 'Grundform' is the shortest form of a word in Danish, it will make a slight difference, but this is hardly worth noting.

3.5 Semaphore Tagging

Another way of finding duplicates is by looking at each articles semaphore tags. When Infomedia receives articles in the inflow, these articles are enriched with *Semaphore Tags*[9]. A tag also has relations to other tags. A politician would be tagged with politics, political party, other people in politics that are in some way associated with that person. The tag is also enriched with a score, that is based on the relevance for that tag in that given article. So if a politician is the main focus of an article, the name of that politician is high than if the article deals with something that the politician only have a remote connection to. Each

[9] Semaphore tags are tags that describes the contents of the article, an article about financial fraud would contain the tag *Economic Crime*. These tags are created by the Infomedia Ontology (`http://en.wikipedia.org/wiki/Ontology_(information_science)`) team. They are creating rules for when a certain word (or words) appear in certain context, then an article will be tagged with a certain semaphore tag.

article then gets a number of tags based on what terms are found in the article. A way of finding article duplicates could be through creating an algorithm that would check a pair of articles with their respective semaphore tags.

3.5.1 Cons to Semaphore

As these tags are more of a general indicator to an articles' contents than an actual text matching algorithm, it will provide very little value as a stand alone implementation. Each article will only contain tags for the terms, for which the ontology team has created semantic rules for. If 100 articles all contained various doings of a minister (picking up the children, going to meetings, being involved in a crisis) many of the same tags would be present in the 100 articles, this could be the ministers name, political party and other general tags that are linked to this minister. It would therefore be hard to decipher much information that could truly and uniquely link two articles together just by doing this. However this could be used to enhance another algorithm (for instance one of the two mentioned above). If the articles compared contains the same tags, they are likely to have some sort of relevance to each other, and if the tags also have matching scores, that would strengthen the possibility of a match. I will not look any further into implementing this algorithm in this thesis, as this approach is difficult in providing a definite result.

3.6 Text Preparation

To reduce the chance of the algorithms failing in detecting duplicates, the text should be 'normalized'. As there are quite a few pitfalls in text analysis, one should try and take as many precautions as possible. A common source of error is common spelling errors, I will not check my article for spelling errors. These can have a rather big impact on the LCS algorithm. However as all text editors today have spell checking, this will be a tiny error source.

As the focus of this thesis have been on proving the thesis statement, there has been done no text preparation other than what is already implemented.

Another problem with text analysis is localized spelling. An example of this can be the Swedish town of Malmö. The issue here being the 'ö' letter which is in the Swedish alphabet. This town's name can be spelled in a few different ways.

- Malmö (Swedish spelling)

- Malmø (Danish and Norwegian spelling)
- Malmo (English spelling)
- Malmoe (Phonetic spelling of the 'ö' character)

The same article could be in different newspapers, but with different spelling of the word. All words, or rather words containing special characters (non English characters) should therefore be normalized. In this case, the character 'ö' could be normalized into the letter 'o'. The phonetic spelling of the 'ö' character is highly unlikely to occur, as the normalized way of spelling 'Malmö' in languages without the 'ö' vowel, would be the third option. There is a lot of issues in this regard. This could be words or characters in a non Latin alphabet (Cyrillic, Arabic, Greek letters used in SI references or others). For this thesis, there is few non Danish texts or words in the test corpus, so it will have little influence. I will in this thesis not normalize text, but accept that minor deviance in scores can occur due to this.

Another good choice in text preparations is to lower case and remove all non alpha numerical characters. This can be advantageous for instance when looking at the name of the current Danish prime minister, Helle Thorning-Schmidt. As the hyphen can often be forgotten the last name can easily be misspelled. Removing the hyphen and replacing it with a blank space character would improve the accuracy of the algorithms.

Stemming is another good way of normalizing text, I have already covered this topic in a previous section, and will not cover this more in detail here.

Stop words will reduce the number of words in an article, and with less words, there are less error margin in terms of spelling errors, also less words in the article text means less words that has to be analysed. As stop words have little meaning when trying to figure out if two articles match, it is a good idea to remove these in order to improve performance.

Unfortunately due to the time issue, I will not be able to look into any of text the preparations mentioned, but wanted to describe what should be kept in mind, when implementing this project into Infomedia's inflow.

Finally it is important that all files are stored with the same encoding[10], as different encoding could cause havoc in the systems ability to read the text in the files. This is already done in the system, so I will not worry about this factor in my thesis.

[10] http://en.wikipedia.org/wiki/Character_encoding

3.7 Technology

As the inflow system at Infomedia is created in C$^\sharp$, it is the easy (and logical) choice to create my work in C$^\sharp$ as well. This would help integrating my work into the existing systems without too much trouble. I am using Visual Studio 2013 and .NET version 4.5 for my code development, which is provided to me by Infomedia. I am using Infomedias Team Foundation Server (TFS) for version control.

Chapter 4

Implementation

This chapter deals with the description of implementing and modifying the LCS.

For this thesis there was created a test project in the Infomedia TFS where I could do my work. This was so that I would not mess up Infomedia's inflow while trying to make the algorithms work correctly.

4.1 Basic Implementation

First off the Cosine algorithm was implemented in said test project, and then the basic implementation of LCS was implemented.

Then the basic LCS algorithm was implemented in the project, to be used for further development.

```
public int LongestCommonSubstring(string str1,
          string str2)
{
    if (String.IsNullOrEmpty(str1)
         || String.IsNullOrEmpty(str2))
```

```
    return 0;

        int[,] num = new int[str1.Length, str2.Length];
        int maxlen = 0;

    for (int i = 0; i < str1.Length; i++)
        {
        for (int j = 0; j < str2.Length; j++)
        {
                if (str1[i] != str2[j])
               num[i, j] = 0;
            else
            {
                if ((i == 0) || (j == 0))
                       num[i, j] = 1;
                else
                    num[i, j] = 1 + num[i - 1, j - 1];

                    if (num[i, j] > maxlen)
                    {
                        maxlen = num[i, j];
                        }
                }
        }
        }
    return maxlen;
}
```

Listing 4.1: Basic LCS implementation in C$^\sharp$

The LCS works by taking in two strings as arguments and then comparing them character by character (see figure 3.4. This works really well for finding substrings that can have been obfuscated in long lines of text. How ever in this thesis, the main objective is to find substrings of plain words rather than finding bits and pieces.

When comparing two articles with the LCS, the algorithm finds a lot of substrings, but only keeps the longest by default

The green rows seen in Figure 4.1 is an article (one article along the topmost edge of the y-axis and one along leftmost edge of the x-axis) split into words (in the basic implementation the articles would be split in characters as seen in figure 3.4 3.4, this figure is from the updated version of LCS). All the purple boxes indicates where a match has been found, a diagonal line indicates a

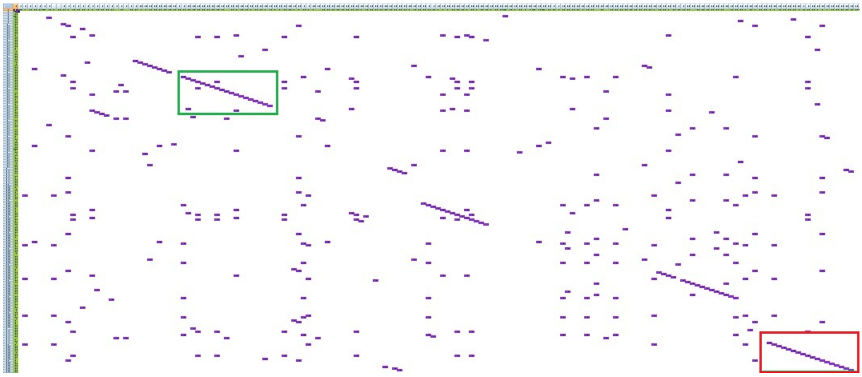

Figure 4.1: Diagram showing the result of two article being compared by using LCS, and plotted in Excels. See Appendix A.1 for bigger image. The green and red boxes indicates longest common substrings found.

row (substring) of matches. The longest one would be the *'Longest Common Substring'* and the length of that would be returned by the algorithm. In the example from Figure 4.1 there are two substrings of equal length (each having a length of 19 words (in the basic LCS the length would the number of characters, including white spaces)), how ever as LCS only returns the length of a single longest common substring (the longest found) and the second one (marked in the red box) is same length, LCS returns the length of the first (marked in the green box) substring. Again, as this example is made out of words rather than characters, the result could vary in case of counting actual characters, but for demonstration purposes the figure explains the idea.

In the case of a perfect match (require both articles to be of the exact same length). A line along the diagonal will be drawn.

In Figure 4.2 there is two article that are almost 100% identical. One of the articles [Win13a] is slightly shorter than the other article [Win13b]. The on line content of the second article is protected by a payment wall, the article content can be found in Appendix B B.1.

The next step was to modify the LCS algorithm to make suit the needs of this thesis. When looking at article duplicates, it makes more sense to look for entire words rather than single characters. This is because when an article is duplicated, any obfuscation or alteration would be done by cutting out sections of the article or moving sections around or adding new sections.

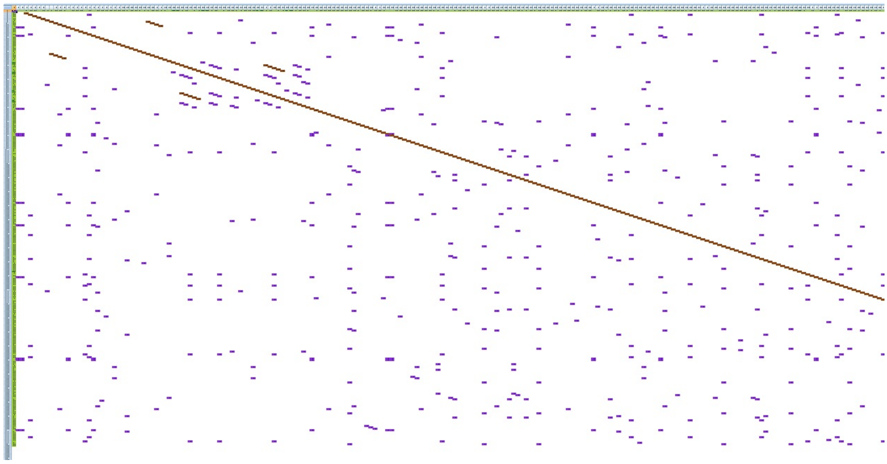

Figure 4.2: Diagram showing a part of an almost perfect match (the article along the y-axis is slightly longer than the article along the x-axis). The long brown diagonal line indicates the longest common substring found.

4.2 Modification of LCS - String Comparison

As hinted both in the text of the last section and also in figure 4.1, the next step was to modify LCS to compare whole words (string) instead of single letters (char). There are of course pros and cons to this approach. One of the pros would be that we could improve the performance of the algorithm. If we assume that the average word length is roughly five characters long[1], that means that comparing articles on a character by character basis increases the number of comparisons by a factor of 25 as compared to doing it word by word. This will therefore mean we can get a rather big performance boost, and when talking about an inflow that just for my test corpus contains 22800 articles, but can be as many as 40000 articles daily, this factor will make quite the impact. Another pro is that we are looking for word duplication, not for sentences obfuscated within other sentences, therefore we do not really need to look at single characters.

On the cons side is the fact that if we are looking for words rather than single characters the algorithm becomes more prone to spelling errors. If the word *'doomsday device'* was included in an article and in an article that is a duplicate of that article was a spelling error *'doomday device'*, this would cause the

[1] `http://www.wolframalpha.com/input/?i=average+english+word+length` - although it is for English words, it is most likely not that different from Danish.

character based substring to be slightly longer than than the words based substring. The character based substring would return *'sday device'* whereas the word based substring would only contain *'device'*. This can be an issue within texts that have many spelling error, but that case is highly unlikely to appear in articles this thesis is dealing with, as one would assume that journalists are pretty okay with correct spelling.

So even though modifying LCS to compare words with words rather than characters, can return incorrect results, the likely hood of this being a substantial error source is negligible.

4.2.1 Issue With Word (String) Comparisons

The issue with comparing words rather than single characters in data science, is that the way comparison is done with data types, which is not as easy as when a human would do text comparison. For a human, reading and comparing two list of single letters, would be substantially slower than comparing two lists of words. This is based on how we recognize both single letters and words. A computer does things in a completely different way.
The basic implementation of LCS compares chars with chars, a char being a basic data type in most modern objective oriented programming languages (Java, C♯, Objective C). Comparing primitives is a simple operation, just checking their values. The modification being implemented would compares Strings with Strings, a String being a data type *'object'* in object oriented languages. A String is internally a list of chars, that makes up the whole String, so comparing a String object with another String object is basically doing a char comparison (with some minor differences, such as String comparison also checks the length of the Strings).

However, in a future implementation of the modified version of LCS, one could transform the Strings into an int (one could use the word map generated by the Cosine algorithm for this), so that each word gets it's own int value, comparing the words as int values would be much faster than both the String and char comparison. This is because an int is a primitive, like the char, and instead of having to check a lot of chars for each "word", there would only be a need to check a single int value for each word. This would improve the overall performance of LCS dramatically (approximately by a factor 25, by applying the same theory as above, each word on average being five characters long) as the list of words (int) would be substantially shorter than the lists of chars.

Due to the fact that I did not contemplate this until late in the work process, there have been no time to implement a String to int conversion in the LCS modification.

For testing purposes it is nicer and easier to read text split up into words rather than text split in chars, so that is a bonus of the string comparison modification of LCS.

So even though it adds no immediate performance boost, this is both a step on the way and helping hand in terms of evaluating the results from LCS.

4.3 Collection of Substrings

Another modification to the LCS that would help finding article duplicates would be to make LCS create a collection of substrings. The benefits of this would to some extend help eliminate the error prone ways of LCS. If looking at two articles that would be considered a perfect match (same length, same article text), except for the word in middle of the text in of the articles. If this have been changed, misspelled or forgotten, this would cause the basic implementation of LCS to return that the length of the longest common substring would be 50% of the article's length. How ever, if we create a collection of substring we can in part work around this issue. Then we would find two substrings, one before the middle word (that is missing or in other way not present in the same form as in the other article) and one after the middle word. Our combination of substrings would then return a length that is close to 100% of the article length.

When doing this modification one should consider using a threshold that indicates the minimum length a substring should have in order to be included. As seen in both figure 4.1 and 4.2 there is a lot of single word matches found when LCS is traversing the article texts. So an effective threshold would be one that filters away the noise, but it not so high that important (in terms of finding duplicates) sentences are filtered out. For this thesis the threshold have been set to four, meaning that all substrings consisting of four or more words are being stored as a result.

When creating the collection of substrings their lengths will be added and compared to the total length, thus creating a hopefully more correct image of whether two articles match or not.

```
public Dictionary<String, String>
 LongestCommonSubstring(List<String> str1,
  List<String> str2, int threshold)
    {
```

```
    if (str1.Count == 0 || str2.Count == 0)
        throw new Exception
        ("One or both documents was empty.");

int[,] num = new int[str1.Count, str2.Count];

var combined = new Dictionary<string, string>();

for (int i = 0; i < str1.Count; i++)
{
    for (int j = 0; j < str2.Count; j++)
    {
        if (str1[i] != str2[j])
            num[i, j] = 0;
        else
        {
            if ((i == 0) || (j == 0))
                num[i, j] = 1;
            else
                num[i, j] = 1 + num[i - 1, j - 1];

            if (num[i, j] >= threshold)
            {
                // Find the start index
                // of the current substring
                String index =
                ((i - num[i, j]) +
                ", " + (j - num[i, j]));

                String insert = "";
                // Do we already have this key stored?
                if (!combined.ContainsKey(index))
                {
                    var words = new List<String>();
                    for (int x = 3; x > 0; x--)
                    {
                        words.Add(str1[i - x]);
                    }
                    foreach (String word in words)
                        insert += word + " ";
                    insert += str1[i] + " ";
                    combined.Add(index, insert);
                }
                else
```

```
                {
                    String valueStore = combined[index];
                    valueStore += str1[i] + " ";
                    combined[index] = valueStore;
                }
            }
        }
    }
  }
  return combined;
}
```

Listing 4.2: Modified version of LCS

As seen in figure 4.3 the addition of substring collection significantly alters the result from what we would have seen, had we only been using the basic LCS implementation. Judging from this diagram the two articles obviously have a lot in common. To tell if the article really have something in common it is needed to take a look at their semantic content. This can be done by using semantic algorithms[2], which is something that could be a part of a future implementation of this project, but it is not implemented in this thesis. Another way of doing this is by reading it the old fashion way, with your own eyes. All of the verification done in a prototype is expected to be done this way, once the users are satisfied that no or only very few errors would bypass the algorithms, then the verification would be left to the algorithms (as per figure 3.3).

[2] http://en.wikipedia.org/wiki/Semantic_matching

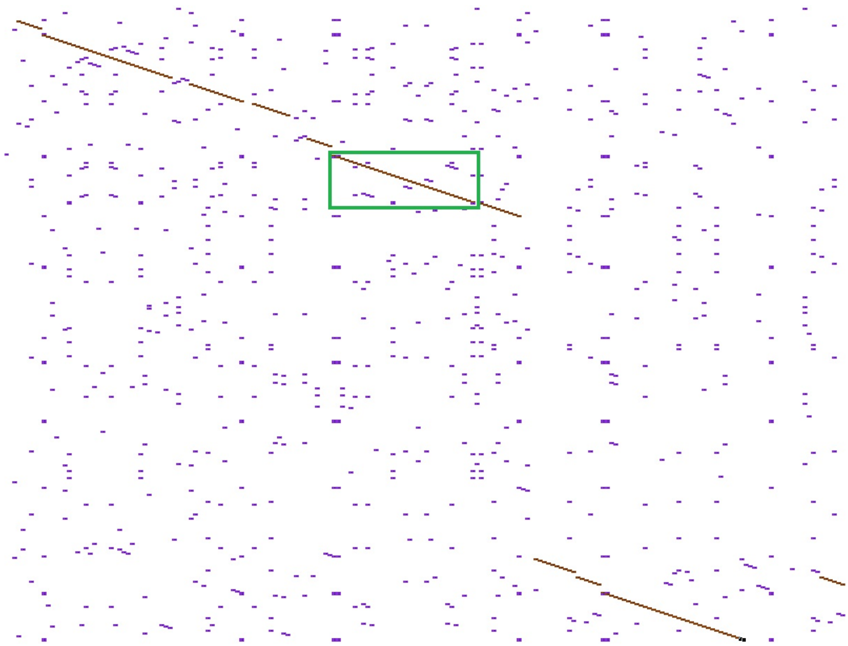

Figure 4.3: The result of having a collection of substrings. All the brown lines indicates where a substring with the minimum length of four have been found. If only the longest substring had been returned as the result, only the substring (marked by the brown line) in the green box would have been returned.

Appendix A

Appendix

This appendix is full of stuff ... add more

Document from Infomedia about the general Vector Spaced Search.
Thesis document from Inge
Links to Wikipedia
The algorithm course book..
Example XML document (article) - to show what the XML documents look like.

An example of comparing two articles with LCS.

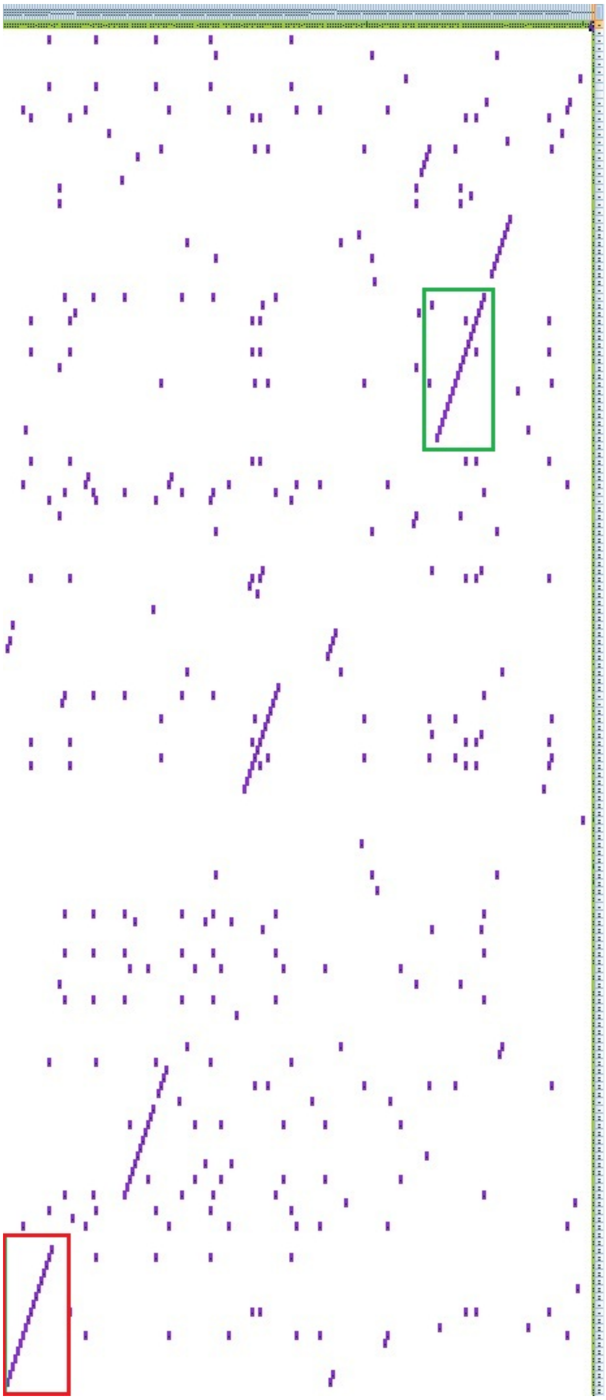

Figure A.1: Diagram showing the result of two article being compared by using LCS.

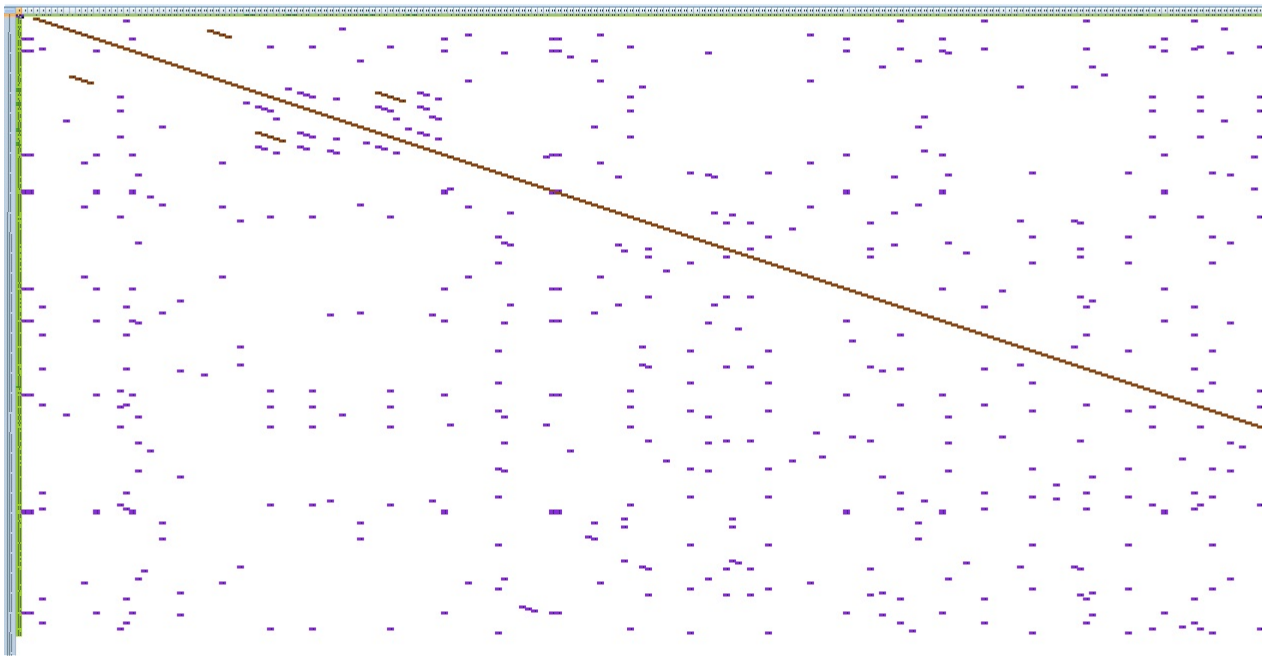

Figure A.2: Diagram showing part of the result from running LCS on two articles that are almost a perfect match.

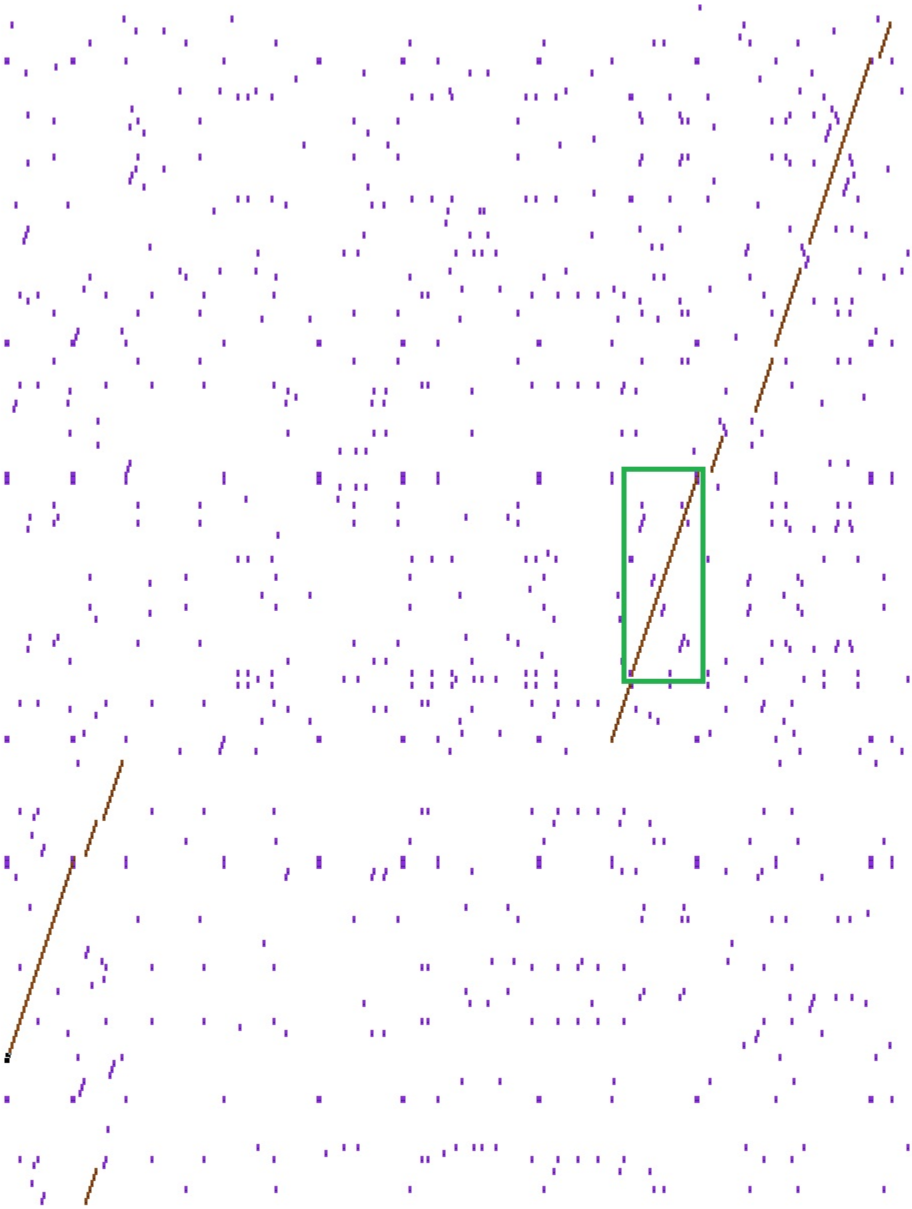

Figure A.3: Diagram showing part of the result of running LCS on two articles and marking up all substrings with a length larger than three words.

Appendix B

Article Content

B.1 Danfoss fastholder stabil forretning

Termostatkæmpen Danfoss holder sit indtjeningsniveau, mens omsætningen falder en smule. Ingen store dramaer i Danfoss. Det kunne være overskriften på selskabets regnskab for årets første ni måneder. Omsætningen falder en smule fra til 25.528 mio. kroner i år mod 25.985 mio. kroner i samme periode sidste år, mens indtjeningen lander på 2.938 mio. kroner i år mod 2.952 mio. kroner sidste år. Med andre ord en stabil forretning uden overraskelser, og det er koncernchef Niels B. Christiansen tilfreds med. »Takket være vores strategiske fokus på en stærk kerneforretning er vi i stand til at opveje både et kollapset europæisk solcellemarked og faldende valutakurser. Det er tilfredsstillende, at vi har formået at tilpasse os og levere så stærkt et resultat trods en generelt lav markedsvækst,« siger han. Selskabet anfører samtidig, at korrigeret for valutaeffekt er Danfoss' samlede omsætning på sidste års niveau. Det europæiske solcellekollaps har ellers ramt Danfoss hårdt, da selskabet producerer invertere til solcelleanlæg. Og solcellemarkedet var blandt de direkte årsager til, at Danfoss for tre måneder siden skar ned og fyrede 69 medarbejdere i Danmark. Den største vækst oplever Danfoss i Rusland og Brasilien, mens det kinesiske marked følger med i lavere tempo. Generelt forventer selskabet dog, at det globale marked vil være præget af lav vækst »et stykke tid endnu«. Og det åbner op for flere opkøb udover de seneste køb af Danfoss Turbocor og de sidste aktier i Sauer-Danfoss. »Derfor kigger vi meget på, om vi kan styrke forretningen gennem fokus på nye markeder og opkøb - såvel af nye teknologier som virksomheder. Vi har styrken til at kunne finansiere sådanne opkøb. Det giver en stor handlefrihed og mulighed for at forbedre Danfoss' position yderligere,« forklarer Niels B. Christiansen. For hele året venter Danfoss »beskeden vækst« i omsætning og indtjening. Danfoss' koncernchef Niels B. Christiansen venter "beskeden vækst" i år".

B.2 Danfoss fastholder stabil forretning - w/o Stop Words

Termostatkæmpen Danfoss holder indtjeningsniveau, omsætningen falder smule. Ingen dramaer Danfoss. Det overskriften selskabets regnskab årets måneder. Omsætningen falder smule 25.528 mio. kroner 25.985 mio. kroner år, indtjeningen lander 2.938 mio. kroner 2.952 mio. kroner år. Med stabil forretning overraskelser, koncernchef Niels B. Christiansen tilfreds med. Takket strategiske fokus kerneforretning opveje kollapset solcellemarked faldende valutakurser. Det tilfredsstillende, formået tilpasse levere stærkt trods generelt markedsvækst, han. Selskabet anfører samtidig, korrigeret valutaeffekt Danfoss' samlede omsætning års niveau. Det europæiske solcellekollaps ellers ramt Danfoss hårdt, selskabet producerer invertere solcelleanlæg. Og solcellemarkedet blandt årsager til, Danfoss måneder skar fyrede 69 medarbejdere Danmark. Den største vækst oplever Danfoss Rusland Brasilien, kinesiske følger lavere tempo. Generelt forventer selskabet dog, globale præget vækst endnu. Og åbner opkøb udover seneste køb Danfoss Turbocor aktier Sauer-Danfoss. Derfor kigger på, styrke forretningen gennem fokus markeder opkøb - såvel teknologier virksomheder. Vi styrken finansiere sådanne opkøb. Det giver handlefrihed forbedre Danfoss' position yderligere, forklarer Niels B. Christiansen. For året venter Danfoss »beskeden vækst« omsætning indtjening. Danfoss' koncernchef Niels B. Christiansen venter beskeden vækst år.

Bibliography

[Alb14] Rikke Albrechtsen. Messerschmidt: Ja-partierne taler vælgerne efter munden. http://www.altinget.dk/eu/artikel/messerschmidt-ja-partierne-taler-vaelgerne-efter-munden, 2014. Accessed: 2014-05-20.

[Inf14] Infomedia. Top 10: Kandidaternes synlighed i medierne. http://www.infomedia.dk/viden-indsigt/europaparlamentsvalg-og-patentdomstolsafstemning-kandidaternes-synlighed-i-medierne, 2014. Accessed: 2014-05-20.

[ML14] René Madsen and Niels Buus Lassen. Ensemble classifiers. 2014.

[Wik14a] Wikipedia. Algorithm implementation/strings/longest common substring. http://en.wikibooks.org/w/index.php?title=Algorithm_Implementation/Strings/Longest_common_substring&action=history, 2014. Accessed: 2014-05-20.

[Wik14b] Wikipedia. tf-idf. http://en.wikipedia.org/wiki/Tf-idf, 2014. Accessed: 2014-05-20.

[Win13a] Johan Winther. Danfoss fastholder stabil forretning. http://www.jv.dk/artikel/1633979:Business--Danfoss-fastholder-stabil-forretning, note = Accessed: 2014-05-20, 2013.

[Win13b] Johan Winther. Danfoss fastholder stabil forretning. *folkebladetlemvig.dk*, 2013. Online content protected by payment wall, article text included in Appendix B.