FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Master's in Computer Science	Spring semester, 2016 Open
Author: Emil Haaland	…… (signature author)
Faculty supervisor: Reggie Davidrajuh, UIS External supervisor: Derek Göbel, Avito LOOPS	
Title of Master's Thesis: Data Quality Indication for User Awareness and Automated Decision Making	
ECTS: 30	
Subject headings: Data Quality Indicators Machine Learning Support Vector Machines	Pages: 50 + attachments/other: code as a zipped file Stavanger, 15 June 2016

Data Quality Indication for User Awareness and Automated Decision Making

Emil Haaland

Faculty of Science and Technology
Department of Electrical Enginerring and Computer Science
University of Stavanger

June 2016

Abstract

Enterprise Resource Planning(ERP) systems help organizations with administrating and planning various business related tasks and give insight with Key Performance Indicators. These mechanisms are highly dependent on being able to interpret the data in order to make the right decisions. This thesis defines a set of Data Quality Indicators(DQI) to calculate and visualize the quality of a large variety of spreadsheet data, used in ERP systems. The DQIs are used to complement a Machine Learning Classifier for automatic quality decision making. With a Support Vector Machine(SVM) approach, the system is able to correctly classify some spreadsheets. But data noise and some quality indicators not directly indicating real quality issues made it difficult for the SVM to clearly distinguish good spreadsheets from bad.

Acknowledgments

I would like to thank my external supervisor Derek Göbel who gave invaluable support, insight, and guidance throughout my work on this thesis.

I would like to thank my faculty supervisor Reggie Davidrajuh for help and guidance with the thesis report.

I would also like to thank Paolo Predonzani for giving feedback and helpful suggestion in regards to Machine Learning.

Contents

Chapter 1

Introduction

A building can only be as sturdy as what the building materials allow. In the same way, the output of a computer algorithm can only be as precise as the input data provided. Every day, operations are set in motion to solve tasks and issues to make life easier for both large organizations and everyday people. Some systems provide users with search results based on their interests, others might manage warehouses or keep track of every car owner in the region. The applications are endless, but one thing they all have in common, is data handling. The decisions a system makes is based on the data input it has to work with, and if the data input is faulty, then the system is prone to be faulty. Even the smallest of errors can have a butterfly effect, capable of messing up complex enterprise systems. Such incomplete or erroneous data are examples of what we call bad quality data. In the business world, many protocols and algorithms work with a large amount of data on a daily basis. The result of algorithms and formulas, whether it's for decision making, statistical analysis or general storage, is heavily based on the data provided. In a perfect world, all data is correct, consistent, well formatted and clean. However, this is never the case with real world data. Data can be provided in large variety. Many systems handle data received from several sources, in many different formats, merged and stored with often little supervision. Such data is exposed to many types of errors. Sensor based or Computer generated data can render faulty data records from overlooked or unspecified situations. Humans are also prone to make errors, such as miss spelling names, miss placing records, overlooking faults or just entering wrong data. All these type of faults can have a huge impact on the operations of an organization. Whether it's for decision making, scheduling, customer statistics or system reliability, poor quality data will affect efficiency, progress and ultimately, revenue.

To handle the problem of bad data quality, we can implement data analysis and procedures to assess the quality and combat errors having an effect on the system. Such measures and algorithms expressing data quality are called Data Quality Indicators(DQI). With Data Quality Indicators, data can be monitored, analysed and cleaned on a regular basis. Though this is not always prioritized. The issue of bad data quality has been a known subject for many years, but it can often be overlooked or not get as much attention as it probably should. There are several reasons for this. For many business executives and project leaders, the

issue and impact of bad data quality is simply unknown and overlooked. Another reason is, on a limited budget, resources are more likely to be used on developing and expanding the business product rather than spending time implementing monitoring procedures and fault checks on data. Monitoring and cleansing data can also be expensive, it is not uncommon to hire third party experts or have dedicated analysers to monitor and clean data. This process could be both very time consuming with arbitrary large datasets, but also produce system down time as altering database records may require halting the system.

In Enterprise Resource Planning systems (ERP), spreadsheets can often be provided as complimentary data. These spreadsheets are usually not meant for computers, but for people to process. And what visual aids helps readability for people, are usually not very helpful for a computer. Spreadsheets can vary in all shapes and sizes, an contain all sorts of information about equipment, location, resources, licenses, etc. To cover all aspects of large projects and schedules. For the purpose of this thesis we will be working with spreadsheets provided to such an ERP system. These spreadsheets can often be unstructured, noisy or lack useful content. Such spreadsheets can be very difficult and complex for a computer to make sense of, and in many cases not useful for the system at all. I will in this thesis explore different aspects of data quality and define and implement a set of Data Quality Indicators to give insight on spreadsheet quality issues. The indicators are to give a quality score and visualised in an innovative way for system users. I will also investigate the use of Machine Learning to automatic classify data quality. The DQI scores will be used as data features to the classifier algorithm to ultimately determine if a spreadsheet is of good enough quality to keep, or if it should be denied.

Chapter 2

Challenges and Previous Work

This chapter describes the model for this project takes place. It describes the challenges involved with the task and what work has previously been done on the subject.

2.1 Problem Definition

The problem with measuring quality in undefined data structures, is the lack of data semantics. We know that data is provided in tabular form, and that each column contains data of a specific definition. To the human eye, this definition is usually easy to recognize. By recognizing the content of the column as for example city names, we can quickly read through the list and mark the entries that aren't city names. City names is a quite simple example, sometimes columns contain data that most people are unfamiliar with. In these situations, we humans are equipped with a strong sense of pattern recognition, that makes the most clueless of us capable of spotting anomalies in most lists of data. These are traits humans possess, but computers lack. To a computer the columns are seen as lists of strings, with no definition other than that. A column of peoples first names would be completely indistinguishable to a list of city names, as a computer has no idea to what compositions of letters constitutes as one or the other.
Because of the lack of data semantics, we need to be able to classify data quality in a general manner. The Data Quality Indicators must be able to figure out what values constitutes as wrong values, without knowing what the data describes. They must be able to handle data of any type, whether its numbers, dates, names or comments, and indicate the quality of the data as close to what human analysts would do. This is important as the goal of an automated quality classifier is to be proficient enough to replace human supervision
A functioning system like this could potentially be highly attractive as it would provide completely unbiased quality assessment independent of data semantics, based purely on intelligent measurements and hard data calculations.

2.2 Previous work

There's much literature to be found on the topic of Data Quality Assessment as the topic has been around for many decades. Much literature revolves around the subject of introducing the business world into the world of Data Quality, and how organizations can benefit from implementing Data Quality Assessment practices. Reasons for this being of such high focus could be for one, just creating awareness around Data Quality. As quality issues is often something people have to deal with on a daily basis, but most are unaware an entire subject on solving those types of issues exists. The other reason is that Data Quality Assessment is prominently the subject of expert analysts, and most would want to avoid the expenses that follows hiring third party analysts to assess data quality. Books and papers introducing data quality assessment, often introduces different concepts of quality measures and examples of how to find erroneous data. The methods presented often require tools to analyze and extract the data to be examined, and most importantly require business rules and constraints to be applied to the data by analysts. These business rules and constraints can be provided by the data specifications, documentation, the organization or the analyst himself. As mentioned in previous sections, the ability to apply business rules is based on information derived by expert analysis, a luxury we won't have in an autonomous system. What we want is an unsupervised algorithm that can measure quality and determine business rules on its own based on statistics, metadata and previous learning. Some papers and technologies have been introduced on this subject. The paper *Unsupervised assessment of microarray data quality using a Gaussian mixture model* [5] presents a method of assessing data quality of DNA Microarrays by calculating a set of quality indicators using statistical measures on microarray metadata along with a Naive Bayes classifier to assess data quality. Using Machine Learning for automated quality assessment has also been been discussed in *Automated data quality assessment in the intelligent archive* [6]. This paper explores various Machine Learning techniques and describe how they can be applied for the use of anomaly detection. They discuss both supervised and unsupervised algorithms, which bases the classification of errors on previous experiences or clustering of data.
Many tools for data visualisation and assisting with error detection exists like CluePoints[10], Talend[12], Pentaho[11] or Profiler[8]. These tools help with visualizing data as plots or charts, to detect anomalies and potential errors. Some tools assist with automatic detection, like Profiler [8], which uses outlier detection techniques through statistical and Data Mining methods to suggest potential errors.

Chapter 3

Background

This chapter introduces the different concepts and workings of Data Quality Indicators and Machine Learning Classification.

3.1 Data Quality

Data is in a sense another term for information, gathering data is gathering information. When relying decisions and findings on gathered information, you would want the information to be as correct and reliable as possible. Data Quality(DQ) defines to what degree we can trust the data to be correct and reliable. It's our level of confidence in that the data meets the requirements of its intended purpose. Data quality is a broad term in general, and will always be subjective to the information it provides and its usage. For a hospital it's important that patient data is timely and precise, so that patients can get the treatment they need, when they need. A post office needs consistent and valid data to be able to distribute letters and packages to their correct destinations. Though good quality on all levels are never a bad thing, what features weighs heavier than others changes from purpose to purpose. Bad DQ is data with high error rate. A data error is an instance of wrong data, inconsistent data or the lack of data where it's expected. In a set of weather data, a wrong value could occur from a sensor giving wrong output, or a person writing down the wrong value. Inconsistent data could be data that is derived from the same event, but gives wildly different values. Or data received from several sources but all of different formats. The subject of data quality helps us understand how reliable our data is.

3.2 Data Quality Assessment

Data Quality Assessment(DQA)[1] is the process of evaluating the quality of data through analysis and measures. DQA determines the overall quality of a data set. DQA can assess quality issues found in both the dataset structure or value by value. The DQA process contains mostly of assigning rules and constraints to sets of data, and detecting data records violating these rules. A DQA rule could for example be only allowing values within a certain range, no empty values, percentages has to add up to 100, names must start with capital letters, etc. All

these rules and constraints finds errors, which again counts up to a total sum of errors. These errors is then used to give an overall assessment of the quality of the data set.

3.2.1 Data Profiling

To be able to apply business rules to assess data quality, we have to have the information we need to apply the correct assessment rules. Gathering this type of statistics and information about data is called Data Profiling[2]. Data Profiling is analysing data, deriving metadata and figuring out semantics of the data. To profile data, we use statistical methods and counts to give a general informal overview of the data at hand. From the profiling, we gather metadata like data types, ranges, dependencies, patterns, counts, etc. This metadata indicates special traits in the datasets which we ultimately can use to build business rules upon. For example in a column of car registration plates, we would derive that a large majority of the values follows a specific pattern of number and letters, and that the majority has the same number of letters and characters. From this information we can say that all plates that does not follow this pattern of letters and numbers are in fact erroneous.

3.2.2 Quality Dimensions

It is common to separate different quality measures into a set of specified quality definitions. These definitions are called Quality Dimensions. Quality Dimensions describes a specific domain on which a set of measures operates on. The reason for using quality dimensions is so that we can group different measures together and put a description label on them. When the measures gives a quality score, we can use these labels to distinguish what area of quality was measured and what errors found tells us about the data. Ultimately the quality measures has to mean something for the reader of a quality report, and the dimensions balances on the line of giving an intuitive output, but also specific enough to provide the information needed.
There are many varieties of dimensions, some more common than others [3]. Though Quality Dimensions being a common term in data quality, there's no universally agreed upon standard for quality dimensions, and they are often subject for overlapping or changing definitions. It's therefor important for a DQ systems to have well documented dimensions to eliminate confusion. As each dimension describes a set of measures, its also important that the dimensions are isolated from each other and don't overlap, so we don't have two or more dimensions measuring the same errors. An example of this could be one dimension measuring amount of empty values, and another dimension measuring invalid values. If the last dimension counts empty values as invalid, we would get double the count of empty values.

3.3 Machine Learning

3.3.1 Classification

Classification is the process of assigning an observation to a category. The ability to identify what surrounds us is an essential part of our everyday life. We take in incredible amounts of sensory input, hearing and vision, and use these inputs to categorize every day occurrences like sounds an objects. We can distinguish between a house and a car, as we know a car has wheels, is made of metal and is mobile, where a house on the other hand is stationary, larger than a car and usually made of wood. These characteristics are called features, and it is by these features we are able to categorize objects based on previous experiences. In computer science, classification is often a subject of Machine Learning. Machine Learning is the subject of algorithms that can learn from data, and are able to make decisions and predictions based on that data. In many ways Machine Learning algorithms simulates how our brain works. By taking in large amounts of observations, the algorithm learns how to distinguish between different observations, and then classify new observations based on previously acquired knowledge.
Machine Learning presents many ways of classification, but they all share the same basic principle: To classify an observation you have to be able to pinpoint the features that distinguish observations of one category to observations of other categories. By defining each category by a set of feature specifications, we can look at the features of an observation, and assign that observation to the category that has the most similar features. Though easy to describe, the classification problem's complexity increases as the number of features increase. Real life data is never perfect and anomalies are expected, there will also be many instances where the same set of features describe two different categories. It is for this reason the science of Machine Learning has been of an increasing interest these last decades. With Machine Learning, scientists and engineers are able to analyze relations and make predictions in large amount of data.

Chapter 4

Method and Design

To be able to classify the quality of a spreadsheet, we need a robust set of quality measures. The measures must be able to indicate quality on a large variety of spreadsheets and represent the quality of the data as accurately as possible. The algorithm must also be effective enough, so that it's worth running large amount of data through it without it being a bottleneck in the overall system. We will in this chapter describe the methods used, the process of how the algorithm works, and explain the design choices made in measuring quality and classifying the sheets.

4.1 Data Quality Indicators

The first step of the algorithm is to give a set of quality scores to the spreadsheets which we can later use as features in a classifier. This set of quality scores are given by what we call Data Quality Indicators(DQI). Data Quality Indicators are the term we use for measuring and exposing data quality. The output of the DQIs can be any form of indication of the data quality, but is often given by a score. We say indicators in plural because we want to have a set of scores instead of just one overall quality score. Though one overall score might be a good exposure of the general quality, we want a more specific indication, where we can tell what type of quality is lacking in the document. These types of data qualities are what we call Quality Dimensions, described in 3.2.2. Each dimension contains a set of measures, and with these measures we can calculate a set of quality scores.

4.1.1 Chosen Dimensions

In this system, we are working on spreadsheets represented as comma separated files (CSVs). A CSV file contains nothing more than the values found in the spreadsheet and how the values are structured in a tabular form. This means that we have no metadata of the spreadsheet like font styling, charts, colors or other details about the overall look of the spreadsheet. The CSV tells us nothing about the values, each value is separated by a comma, where a comma is a separator of values between two columns. The computer initially reads the values as a list of strings, row by row. It's these list's of strings we are measuring quality upon,

we therefor have to choose measures that are able to find quality issues where not details or semantics about the data is present. With this in mind, I chose to measure the quality on five dimensions, specified in table 4.1. A system usually has a defined structure to which it will try to read data from. If the data doesn't follow this structure, most systems are unable to derive anything useful. It is with this in mind the five dimensions were chosen.

Table 4.1: Chosen DQI dimensions

Completeness	Is the dataset complete? Or does it contain many records of NULL values?
Validity	Is the data from the records logical? Does the data fall within logical boundaries; A person being negative years old.
Consistency	Is the value format consistent?
Structural coherence	Are the data structured in a good manner.
Integrity	Row Integrity: A row starts with a primary key. Column Integrity. Referential Integrity.

The dimensions are separated into two groups, general and special. In one group the dimensions measures quality on a completely general basis, where all the values are analysed for their string composition and not the value they actual represent. In the other group the dimensions tries to measure quality based on what the values represents.

General measures

Completeness

The completeness dimension measures how complete the dataset is. How complete a dataset is, is determined by the amount of missing values. A spreadsheet with perfect completeness is a spreadsheet where all the cells within the table contains a value. An empty cell is one form of missing value, but not the only one. Many data structures has rules that prohibits leaving a cell empty, to deal with this a substitute/default value is often inserted instead; like "Null", "Missing" or just a single special character like "-" or "#". Many cells are also generated by formulas and functions, and when these automated methods encounter issues, they often leave a cell with an error value. All these forms of missing values are counted and weighed against the amount of meaningful values. The end result of the dimension is the percentage of missing values in the document.

Row Integrity

Row Integrity is a measure based on the reason of thought that a dataset should contain a unique identifier for each data record. This means that for a spreadsheet to get a perfect Row Integrity score, the spreadsheet must contain a column of unique values, so that each row in the table is given a unique identifier. This is done by finding the column in the table that is closest to contain a unique value for each row in the table. The end result is the percentage of unique values that this column is able to provide to the total amount of rows.

Structural coherence

The structural coherence dimension measures the overall structure of the spreadsheet and how the values are arranged. Spreadsheets comes in all shapes and sizes, though inherently meant for data to be organized in tables, this isn't always the case. A spreadsheet can contain values scattered in all directions, contain multiple tables or sometimes almost no values at all or be completely empty. For a system to read and make sense of unstructured data can be extremely demanding and if not, almost impossible. What we want is well organized and structured data. Perfect spreadsheet structure is data arranged in a single table in a perfect rectangle. This dimension measures how well a spreadsheet suits the ideal model of containing only one table of data with no rows or columns fluctuating in length from the others. It also considers the amount of values found in the spreadsheet. A spreadsheet with just a few cells filled are usually of no real use.

General Consistency

The consistency dimension is divided into two separate measures, general and special. The general consistency measure sees only the values for their string composition, and how consistent the strings are to the rest of the strings in the dataset. This dimension measures column by column and compares the composition of each string to the general string composition of the rest of the strings in the column. You can call this the internal consistency of a column. The dimension collects metadata statistics on the strings in the column and count values that deviate from the general metadata found in the column as a whole. Four sets of statistics are collected, shown in table 4.2
For each statistic, an outlier detection method is applied to find deviations/outliers. The methods used, are explained in section 5.2.1. The outliers are then counted, and an overall consistency score is given to the column from the amount of outlier strings.

Table 4.2: String consistency measures

Statistic	Description	Outlier method
Word count	Count number of words in each string and compare to rest of column.	Tukey's range
Value Length	Get the length/character count of each string.	Tukey's range
Character types	Divide character into three types: "Letters", "Numbers" and "Special characters". And count how many instances of each type strings contain.	Tukey's range
Character frequencies	Find which characters are used and their frequency for each string, and compare if they are similar to the most common characters used in the column as a whole.	Cosine distance

Special measures

The dimensions that measures on special cases of quality are Validity and the Special Consistency. These two dimensions has special cases of measures dependant on the value type. The value types are:

- Text/Strings
- Person Names
- Numbers
- Dates
- Codes
- Acronyms
- Prices

These column types are chosen as they are common to the type of spreadsheets used in ERP systems in the oil and gas industry. How the algorithm predicts the value type is further described in 4.1.2.

Validity

Validity is a special case only dimension because it is a measure on finding invalid values based on value context. Invalid values are values that are not correct, they are not within the logical boundaries of the column. For example in a column of person names, a number would not be a valid value. Nor would a negative value be valid in a column of people's age. The reason we are able to apply constraints like this to a set of values are because we have knowledge about what values are logical in that specific context. If the only context we have is that the values

are a list of strings, then all values are valid. Therefor we have to figure out the column type to get the context we need to apply constraints.

Table 4.3: Validity constraints

Value type	**Constraints**
Text/Strings	All pass
Names	Title case, number of words, Only letters (+ a special character), Names only.
Numbers	Positive/Negative values, Numbers only.
Dates	Dates only
Codes	Codes only
Acronyms	All caps, Acronyms only.
Prices	Extreme values, Positive/Negative, Prices only.

Table 4.3 lists which constraints are applied to which value types. More specifics on how the constraints are measured is explained in later section 5.2.6. The invalid values are then counted and used give a score to the dimension.

Special Consistency

The special consistency dimension in the same way as validity, measures consistency based on value context. Though instead of applying predefined constraints, it uses statistical methods to detect outliers in the same way as for general consistency.

Table 4.4: Special consistency constraints

Value type	**Measures**
Text/Strings	All pass
Names	All pass
Numbers	Value range
Dates	Date pattern
Codes	Code pattern
Acronyms	All Pass
Prices	Value Range

4.1.2 Predicting value types

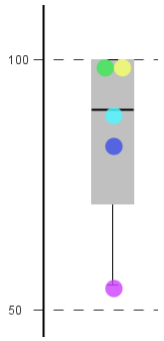
For the dimensions to be able to apply measures on special value type cases, we need to be able to predict the value type and provide it to the dimensions. This is done by collecting metadata on the values of a column and use these metadata values to place the values to a specific data type. After each value in the column is assigned a value type, the majority vote will decide the value type of the column.

The control statements are arranged in a tree based structure. The feature splits directing the value type decision is based on the typical features associated with those value types. Like codes being in one word or names being title cased.

4.1.3 DQI representation

When all the dimension have completed their measures, we collect the measuring results and calculate a set of indicator scores. The scores are generated to give the spreadsheet an overall quality score for each dimension. These scores are used for the automated decision making, but are also meant to provide the user with information about the state of the spreadsheet quality. The best way to present information in a quick and useful manner is to visualise the quality. A goal of this thesis was to develop a new way to display a data quality indicator chart to be used as a visual aid to the user. We found out that the best way to display the data quality would be a chart that could show the overall quality score for each spreadsheet tested, and at the same time show how the internal dimension scores ranged within each sheet. The end result became a customized box plot of the spreadsheet scores, with the score of each dimension placed on the box to show how the dimensions dictate the position and length of the box. The internal dimension scores are shown as a colored dot, with the dot representing the dimension.

Figure 4.1: Example of a box in the customized box plot



In figure 4.1 we see an example of how a box ends up looking. The colored dots green(Row Integrity) and yellow(Structural Coherence) give a perfect score of 100. Teal(validity) and blue(consistency) a little lower, and purple(completeness) lengthening the box with the lowest score. A long box indicates a wide spread in dimension scores, while a small box indicates shows that the dimensions score similar results. A perfect score would be a small box lying close to the top score line. An example of the box plot in its entirety can be seen in section 6.1.

4.1.4 Outlier Detection methods

The two main methods for detecting outliers used by the measures are the Tukeys Method and the similarity measure Cosine Similarity.

- Tukey methods relies around detecting outliers that deviates from the main portion of values by using quartiles and the Inter Quartile Range(IQR), this makes the method applicable to most value sets as the method is not dependent on the value distribution. The method works by calculating $IQR = Q3 - Q1$, where *Q3* and *Q1* are the third and first quartiles. a range is then calculated to range from $Q1 - 1.5 \cdot IQR$ to $Q3 + 1.5 \cdot IQR$. All values found outside this range are determined outliers.

- Cosine Similarity is a way to measure the angle between two vectors. The result is given as the cosine of the angle. Cosine of 0° is 1.0 and less for all other angles. The cosine similarity between two vectors *V1* and *V2* is calculated as:

$$Similarity = \frac{V1 \cdot V2}{||V1|| \cdot ||V2||}$$

4.2 Classification

The last step of the algorithm is the automatic decision making. We run the DQI scores through data a classification model that will decide whether a spreadsheet is of good enough quality to be used, or not.

4.2.1 Support Vector Machine

We chose the Support Vector Machine (SVM) method [4] to be used for the automated decision making. Support Vector Machines are Machine Learning methods for supervised learning that can be used for regression and classification. SVM suits our purpose well as it's efficient, versatile and excellent as a binary classifier. With SVM the DQI scores are placed as points in a hyper plane, where each point is a vector of the dimension scores. Based on these points, SVM draws a hyper plane that separates the points of one class from the points of the other class and maximizes the margin between the two classes. The beauty with SVM however, is that in cases where it is not possible to linearly separate the data points, we can transform the data into a higher dimensional space where it is possible to separate the data. This is done using kernel functions. And this is where the versatility of SVMs comes from, as there exist several kernel functions that suits different data purposes [7]. The most common kernels are the *Linear* kernel, and the non-linear *Polynomial*, *Radial basis function (RBF)* and *Sigmoid* kernels. Each kernel separates the data in their own way, and how well they perform is highly dependent on the data we use. With Support Vector Machines and the right kernel choices, we should be able to train a classifier to make automated decisions on spreadsheet quality.

Chapter 5

Implementation

In this chapter we will take a closer look at the implementation of the different modules in the system. All the parts of the algorithm is implemented in Java. The algorithm mainly consist of four modules: The data preparation and statistics gathering, Quality Dimension Measures, indicator scoring, and lastly the SVM classifier. Installation and how to run the code is described in Appendix A. Running and setting up tests is described in 6

5.1 Data preparation and analysis

The first step of the algorithm is the preparation step. In this step data is loaded from the files and parsed into sets of columns. We then gather statistics and metadata on those columns of data to be used later by other modules.

5.1.1 Parsing Data

The code for parsing data from source to algorithm can be found in the package *csv* in *CSV.java*. The source data we are working with is given as spreadsheets converted to .csv format. To parse these csv files into Java, I use the open source library OpenCSV[13]. OpenCSV reads files row by row, as *Cp1252* encoded text. where the rows are stored in String Arrays with each element being a value of a column. This is a easy and well structured way of parsing the tables, but for my purpose it would be more efficient to store the data column by column instead of row by row. This is because the measures mainly operates on a column by column basis, and we would eliminate a lot of operations by changing the data storage in this manner. This is completed by iterating over the String Array rows and appending the values over to a set of ArrayLists.
When converting Excel spreadsheets to .csv format, there's often an issue where a large amount of empty rows and columns are included in *CSV* file. These extra rows and columns occur because formulas in excel may extend further than the actual content. These types of cells are often called ghost cells. We deal with these ghost cells by iterating over the columns and rows and pinpointing the last cell containing a value. All the cells that follows are then determined as ghost cells and removed from the data structures.

We then have a set of data presented in columns, which we can start measuring quality upon.

5.1.2 Gathering metadata

The dimensions will one after the other run measures on the columns. In an effort to keep the algorithm as efficient as possible, I try to limit the number of times the dimensions have to iterate over the values of a column.
Instead I try to collect as much metadata from the columns as possible before the dimensions start measuring, so that the measures can use this metadata instead of having to collect it themselves. The code for the collection of metadata and statistics is found in the *analyze* package, in *ColumnData.java* and *ValueData.java*. The *ColumnData* represents the metadata for the column as a whole, while *ValueData* collects data about single values.
We start out by providing *ColumnData* with a list of string values. Each string in the list is passed to a *ValueData* object which contains methods for gathering string value metadata. The string metadata is then stored in arrays in *ColumnData* which are used by the measures later. *ColumnData* also makes a value map of all the strings in the column, which can be used for checking the value variety in the column.
The metadata we collect from the string values are: The number of words in the string, character types it contains, character frequencies, casing of the words, and lastly a check if the string follows a certain pattern.

Listing 5.1: findCasing() method

```
1   private String findCasing(String value){
2       if(value.equals(value.toUpperCase())){
3           return "Upper";
4       }
5       String[] split = value.split("\\s+");
6       for(int j = 0; j < split.length; j++){
7           if(!Character.isUpperCase(split[j].charAt(0))){
8               break;
9           }
10          if(!split[j].substring(1).equals((split[j].substring(1).
                toLowerCase()))){
11              break;
12          }
13          if(j == split.length-1){
14              return "Title";
15          }
16      }
17
18      return "Unknown casing";
19  }
```

Most of the methods are simple counting measures, except for *findCasing()* and *findPattern()*, shown in listings 5.1 and 5.2, which tries to categories the pattern and casing type of the values. These categories are used later when trying to detect the data type.

Table 5.1: Patterns

Pattern name	**Regex/Symbol**
pricePatterns	".+\d+[,]\d+$" ".+[,]+\d+[.]\d+$" "\d d+[,]\d+$"
currencyCharacters	"kr", "$", "£", "€", "NOK", "EUR"
numberPatterns	"\d+[.]\d+$" "\d+"
datePatterns	"^d{1,2}[\/.]\d{1,2}[\/.]\d{2,4}.+"

Table 5.1 show the different regular expression representations of patterns tested upon.

Listing 5.2: findPattern() method

```
1   private String findPattern(String value){
2       for(String pattern: pricePatterns){
3           if(value.matches(pattern)){
4               return "Prices";
5           }
6       }
7       for(String currency: currencyCharacters){
8           if(value.contains(currency)){
9               return "Prices";
10          }
11      }
12      for(String pattern: datePatterns){
13          if(value.matches(pattern)){
14              return "Dates";
15          }
16      }
17      for(String pattern: numberPatterns){
18          if(value.matches(pattern)){
19              return "Numbers";
20          }
21      }
22
23      return "No Pattern";
24  }
```

Table 5.1 shows which patterns and symbols are looked for when trying to categorize the pattern. It uses regular expression to see if the patterns of the string value matches one of the patterns in the list, or if the value contain one of the symbols. If able to match a pattern, then it's very likely that the data type is also set to that data type later on.

5.1.3 Data type prediction

Predicting data type is done with a set of control statements laid in a decision tree/forest structure. The code can be found in the method *DataTypeDetection* in the *analyze* package. The method in this class tries to decide which type of data a column contains, using the metadata from *ColumnData*. *ColumnData* has a method called *getColumnStatistics*, which provides the mean metadata values of a column. It returns the average values of the numerical data, and a category for casing, character types and pattern if a category in one of these has over 50% presence in the column.

Listing 5.3: decideType() method tree one

```
1        if(charType.equals("Numbers+Special") || charType.equals("All"
             )){
2            if(pattern.equals("Prices")){
3                return PRICES;
4            }else if(pattern.equals("Dates")){
5                return DATES;
6            }else if(pattern.equals("Numbers")){
7                return NUMBERS;
8            }
9        }
```

Listing 5.4: decideType() method tree two

```
1        if(nrOfWords > 5){
2            return TEXT;
3        }
4        else if(nrOfWords > 1.5){
5            if(casing.equals("Title")){
6                return NAMES;
7            }else{
8                return TEXT;
9            }
10       }
11       else{
12           if(charType.equals("Numbers")){
13               return NUMBERS;
14           }else if(charType.equals("Letters")){
15               if(casing.equals("Upper")){
16                   return ACRONYMS;
17               }else{
18                   return TEXT;
19               }
20           }else if(charType.equals("Numbers+Special")){
21               return PRICES;
22           }else{
23               if(pattern.equals("Dates")){
24                   return DATES;
25               }else{
26                   return CODES;
27               }
28           }
29       }
```

Listings 5.3 and 5.4 shows the two trees that decides a data type. Tree One runs first, if no values are decided, then tree Two runs. The control statements are based upon common traits of the types they decide. Though simple, it provides sufficient results for the dimensions that need a data type to measure on.

5.2 Quality Metrics

The main part of the algorithm is the quality measurements. The measuring forgoes in five different classes, found in the *metric* package. One file for each dimension. Structural Coherence and Row Integrity measures on the spreadsheet as a whole, while consistency and validity measures and scores column by column.

5.2.1 Utility and Measuring methods

- Tukeys Range
  The *metrics* package contains a class *Utils* which has two methods *findOutliers()* and *formatToNumbers()* used in different measures. *findOutliers()* takes a list of numbers and calculates a range with Tukeys method, described in 4.1.4. All values outside this range is determined outliers. The method then returns a list of the positions of the found outliers.

- Number formatter
  Also in *Utils* is the method *formatToNumbers()*. This method takes a list of string and converts the strings into doubles. If the method is not able to convert a string, a null value is put in the list instead. The method is primarily of use when encountering numbers that uses both commas and periods in the same string.

- Pattern Recogition
  Special Consistency uses a pattern recognition algorithm, developed by Morten Wærsland [9], to derive regular expression patterns on the values in a column. These patterns are then used to count how many values are using the same pattern. The pattern recognition methods are found in the *pattern.jar* library file.

5.2.2 Structural Coherence

Structural Coherence, found in class *Structure.java* is the first dimension measured. The class is given a set of columns as a nested list of strings in a list of columns. The method then apply a set of measures to this data. The end result is an overall score on the data's structure, this score is measured in two different ways. The first, measure the amount of content in the spreadsheet. If the spreadsheet contains under ten values, then the measure returns a percentage of how many values the spreadsheet contains, up to ten. If a spreadsheet had five values, then the measure would give it a 50 out of a top score of 100, based on the amount of content.
The next measure in this class is a measure on how much the column heights

vary. The optimum spreadsheet would have a perfectly rectangle shaped table, this measure gives us an indication of how closely this aspiration is met. It starts of by finding the length of each column. The length is determined by how far down actual values reach, not counting in trailing empty cells. Then we apply Tukey's method of finding outliers to these lengths, and count those that deviate to much from the average length. The score is then given by the squared percentage of columns that are inside the length range.
When the measures are completed, the class returns the lowest of the measured scores as the overall Structural Coherence score for that spreadsheet.

5.2.3 Completeness

Completeness, found in *Completeness.java* is the first of the column based dimension scores, described in 4.1.1. The measure iterates over the values of the column, removes white space and sets the values to lower case. By doing this we can easily check if a value is empty or match one of the null values described in 4.1.1. The completeness score is given as the percentage of complete values in the column. The algorithm then puts all the complete values in a new list of strings, which will be used by the dimensions so they won't consider empty or null values in their measures.

5.2.4 General Consistency

General Consistency, found in *GeneralConsistency.java*, contains a set of methods for measuring string consistency. The method *scoreMetric()* is given a *ColumnData* object, which contains the column metadata needed for the dimension's measures. The method measures consistency in four different ways, as described in 4.1.1, and returns a score of the columns internal string consistency based on these measures. The measures finds string anomalies/outliers in the column and the dimension score is given as the percentage of non-outlier strings.

- Word Count
  The word count outliers of the column is found by firstly calling *getWordCounts()* from *ColumnData*. *getWordCounts()* returns a list of all the strings' word counts. From this list we find the strings that have a significantly different word count compared to the other strings in the column, and count them as outliers. We use the Tukey method to find outliers by calling *findOutliers()*. The method returns a list of column positions that contain word count outliers.

- String length
  The string length outliers are found in exactly the same way as for the word counts. The lengths are received from *getTotalLengths()* and given to *findOutliers* which returns a list of outlier positions.

- Character Types
  Character type outliers are string values that have a different amount of letters, numbers and special characters, compared to the other values in the column. The outliers are found by making two vectors representing the number of different character types, and comparing the similarity of the two. One of the vectors contains the average number of letters, numbers and special characters in the column. This vector is then used to compare against individual value's vectors. The vectors are compared with *Cosine Similarity*. If a value has under 0.8 similarity with the average values, the value is determined an outlier. The code for finding character type outliers is found in *findCharacterOutliers()*, snippet of the code shown in Listing 5.5.

Listing 5.5: findCharacterTypeOutliers() snippet

```
1       ArrayList<Integer> outliers = new ArrayList<Integer>();
2       for(int i = 0; i < values.length; i++){
3           Vector<Double> vec = new Vector<Double>();
4           vec.add(values[i][0]);
5           vec.add(values[i][1]);
6           vec.add(values[i][2]);
7
8           if(cosineSimilarity(avgVector, vec) <= 0.8){
9               outliers.add(i);
10          }
11      }
12
13      return outliers;
14  }
```

- Character Frequency
  The last string measure of General Consistency is finding the character frequency outliers. The code for this is found in the method *findCharacterFrequencyOutliers()*. The method takes a list of hash maps from *ColumnData's getCharacterCounts()*, the maps contain frequencies of each character found in a string value. The method starts of by combining all the character frequency maps into one large map containing all the column's character frequencies. We want the characters of the map to be sorted from highest frequency to lowest. We do this by making a character array listing the characters in order. With a map of all the character frequencies and their frequency order, we can make two vectors of the columns character frequencies and the individual values' frequencies. The first vector is made by adding all the frequencies ordered from highest to lowest. The second vector, which is the frequency vector of a value, is made in the same way. But has a '0' frequency added for the characters it doesn't contain. Now we have two vectors we can compare using *Cosine similarity*. As an example, the procedure of two values in a column would go as shown in algorithm 1. Values with cosine similarity less then 0.4 is determined outliers.

Algorithm 1 Character frequency example

$Column\ values:$
$Value\ 1 \leftarrow$ "$Ola\ Nordmann$"
$Value\ 2 \leftarrow$ "$Uncle\ Sam$"

$Character\ frequency\ maps:$
$Map\ 1 \leftarrow \{N:3, A:2, O:2, D:1, L:1, M:1, R:1\}$
$Map\ 2 \leftarrow \{A:1, C:1, E:1, L:1, M:1, N:1, S:1, U:1\}$
$Maps\ Combined \leftarrow \{A:3, N:3, L:2, M:2 O:2, C:1 D:1, E:1, R:1, S:1, U:1\}$

$Frequency\ vectors:$
$Vector\ Combined \leftarrow \{3,3,2,2,2,1,1,1,1,1,1\}$
$Vector\ 1 \leftarrow \{2,3,1,1,2,0,1,0,1,0,0\}$
$Vector\ 2 \leftarrow \{1,1,1,1,0,1,0,1,0,1,1\}$

$Comparison:$
$Cosine\ Similarity(Maps\ Combined,\ Vector\ 1) = 0,91$
$Cosine\ Similarity(Maps\ Combined,\ Vector\ 2) = 0,82$

When all the outliers are detected, we are left with a long list of column positions where outliers are. Some outlier are counted twice, we therefore remove all the duplicate outliers and count the ones that are left. The General consistency score is returned as the percentage of the column's non outlier values.

5.2.5 Special Consistency

Special Consistency, described in 4.1.1, measures consistency on values of a specific type. The code is found in the class *SpecialConsistency.java*. The class takes three parameters; A *ColumnData* object, a number representing the column type and a list of the column's values. To get the Special Consistency score of a column, we call *scoreMetric()*, which runs the correct scoring method, based on the column type. Not all column types has special consistency measures, as shown in Table 4.4. If a column has a type without a special consistency measure, then a perfect score would be returned and not be considered by the end results as described in later sections. The column types with special consistency measures are *Numbers*, *Dates*, *Codes*, *Acronyms* and *Prices*, each type's measure is found in separate methods:

- *scoreNumberConsistency()*
  For numbers we have two consistency measures, a measure on the number format and a measure on the value range.
  The value range of the numbers is a measure for detecting outlier values, numbers that are far off from the rest of the numbers. But before we can start measuring on the actual values of the numbers, we have to convert the numbers from strings into a data type Java can understand. Numbers that are written without dots or commas can be directly converted from

string to *double*, but often numbers in spreadsheets vary in how they are formatted. The number 1000 can be written as "1000", "1000.0", "1000,0" or "1.000,0". In these cases we cannot just convert the strings into numbers with Java's built in methods. I therefor made a simple converter method; *formatToNumbers()*, described in section 5.2.1, that converts a variety of number formats into formats Java can understand. When the strings are converted into numbers, we use Tukey's Range to find outliers. The score is then given as the percentage of non-outlier values.
The other number consistency measure is a measure on the consistency of the number formatting used. Using the pattern recognition algorithm described in section 5.2.1. The algorithm returns the number of values in the column that followed the same number format. The score is then given by the percentage of values in the column that all use the most common number format.
After these scores are measured, the method returns a map of the score name and the score. Which score is chosen for the indicators later on is explained in section 5.3.

- *scoreDateConsistency()*
  Date consistency is measured in the same way as the number formatting measure. Dates in the same way as numbers can be written in several different ways. By using the pattern recognition algorithm from 5.2.1 we get a number of how many dates follow the same date format. The score is then returned as the percentage of dates in the column that follows the same format.

- *scoreCodeConsistency()*
  Code consistency is also measured in the same way as date consistency, as we also want codes to follow the same consistent pattern throughout the column. The measuring and scoring is done in the same way as for dates.

- *scoreAcronymConsistency()*
  For Acronyms we measure the consistency by how many of the values are all upper case. By getting the word casing from *ColumnData* we count how many values are written as upper case. The score is then given by the percentage of upper case values in the column.

- *scorePriceConsistency()*
  Lastly the price column type consistency is measured in the same way as numbers. We find and measure the amount of value outliers and measure the price format consistency. The scoring is then returned in the same way as for numbers.

5.2.6 Validity

The validity measure, described in section 4.1.1, is found in the file *Validity.java*. Validity operates in the same way as Special Consistency in that it has a method for each data type that measures quality and gives a score. Different from Special

Consistency however, is that it counts values that does not conform to a set of constraints, and ends up with one score, the percentage of values that are within the constraints. This is more like how general consistency scores a column. Special Consistency's methods however, are not able to return the index of outlier values, and we can therefor not combine the different measures into one score. As shown in 4.3, not all data types have validity constraints, and some have the same constraints. The ones that have the same constraints are measured in *scoreDefaultValidity()*.

- *scoreDefaultValidity()*
  Most of the validity constraints are based on the same principles as the *Data Type Detection* from the analysis step. All the validity measures for the different data types share one constraint, that a value that is not classified as the same data type as the column's data type, is determined invalid. Acronyms, Codes and Dates have no other constraints than this, and are all measured in *scoreDefaultValidity()*. This method calls a method called *findWrongValueTypes()*, this method is called by all the data type measures in validity. The method uses *ColumnData* for each value in the column and runs it through the data type detector. If the detector returns another data type than the column's data type, then the index of this value is set as a wrong value. When the method has been through all the values, it returns a list of all the wrong-value indexes. The validity score for this data type is then given by the percentage of correct values in the column.

- *scoreNameValidity()*
  The "name" data type is the data type with the most constraints. The constraints check for correct word casing, the number of words, which character types are used, and lastly the data type of the values.
  The casing constraint is based on that names should be title cased. The measure uses the list of word casings from *ColumnData*, and lists all the column indexes that contains a value with the wrong casing.
  The number of words constraints gets a list of the word count for each value from *ColumnData*. It then lists all the values that has a word count lower than two or higher than five.
  For the character type constraints, we check that the names only consists of letters, as no names should contain numbers. We also allow one special character as some lists of names divide first and last names with a special character. We get the character types from *ColumnData* and lists the indexes of values that are not within the constraints.
  Lastly we check for wrong value types with *findWrongValueTypes()*, which returns a list of indexes. From all the values that are now determined as invalid, we can give a score given by the percentage of the column consisting of valid values.

- *scorePriceValidity()*
  Price column validity is measured on three constraints, data type, extreme values and a positive/negative value check. The data type constraint is the same as described for the previous measures. The extreme value

contraint however, checks for prices that exceed certain extreme values. The constraint checks if a price is less than -1.000.000.000 or more than 1.000.000.000, if it is, then it's marked invalid. To check the value of a price, we convert the price string into a number using *Utlis.formatToNumbers()*. The positive/negative value check is found in method *posNegCheck()*. The method checks if 90% of the column consists of either positive numbers or negative numbers. If it it does, the method determines the numbers in the 10% or less, as invalid. The method then returns a list of indexes of the invalid values. All the invalid values are then combined and a score is given by the percentage of valid price values.

- *scoreNumberValidity()*
  The last of the data types in the validity measure is the number type. The numbers are scored in exactly the same way as the prices, except for the extreme value constraint. Numbers are only score on the data type and positive/Negative constraints.

5.2.7 Row Integrity

The Row Integrity as described in section 4.1.1, scores a spreadsheets on it's ability to provide unique identifiers for each row in the spreadsheet. The code for this measure is found in *RowIntegrity.java*. The measure iterates over and measures the amount of unique values in each column in the spreadsheet. The uniqueness score of a column is measured in *Uniqueness.java* found in the same package as the other measures. Other than the uniqueness scores, it also finds the columns with the most values. The length of that column is then used together with the uniqueness score to find the column that is closest to cover the longest column with unique values. A snippet of how the method finds the best column is shown in Listing 5.6. The Row Integrity score is then returned as the chosen column's percentage of unique values to the length of the longest column.

Listing 5.6: RowIntegrity snippet

```
1       for(int i = 0; i < columns.size(); i++){
2           Column column = columns.get(i);
3           double uni = column.getUniquenessScore();
4
5           //Converts the uniqueness score to score uniqueness
                compared to the longest column in the spreadsheet,
                instead of
6           // the length of the column itself.
7           double convertedScore = (uni*column.getColStat().getRows())
                /maxRowCount;
8           if(convertedScore > maxUniScore){
9               maxUniScore = convertedScore;
10              primaryKeyColumn = i;
11          }
12      }
```

5.3 Indicator scores

When all the dimension measures are completed, we are left with several column scores and two spreadsheet scores. These scores needs to be collected and merged into a set of DQI scores, with one score for each dimension. The code for this is found in the file *Spreadsheet.java* in the *dqi* package. The Spreadsheet object contains the basic spreadsheet information needed in later stages along with the spreadsheet's DQI scores. The class is given the spreadsheet file name, the two spreadsheet scores from Structural Coherence and Row Integrity and a list of *Column* objects, which contains the column scores given by the consistency measures and validity. The *Column* object is found in *Column.java* in the same package. The DQI scores are calculated by leaving the spreadsheet scores Row Integrity and Structural Coherence as is, while the final score for the column measures are given by the average column score for each dimension. As consistency gives a special and general score, the lowest of the two are chosen for each column. The scores are initially between 0 and 1, but we multiply them by 100 so that they range between 0 and 100 instead. We then end up with five dimension scores for the spreadsheet; Completeness, Validity, Consistency, Structural Coherence and Row Integrity.

5.3.1 DQI Visualization

As the rest of the system is implemented in Java, I ended up implementing the chart in Java also. There exist several libraries for graphical charts and statistics in Java, but as I needed to change and override code to make the chart look as I wanted, I figured it would be easier to implement the chart from scratch. The code for the box chart is found in the *visualization* package in the three files *ChartFrame.java*, *BasicChart.java* and *BoxChart.java*. *ChartFrame.java* initializes the chart graphics using the graphical user interface package *Java.Swing*. It initializes the *BoxChart* code which extends *BasicChart*. *BasicChart* draws the basics of a chart; the axis, spreadsheet names and color descriptions for the dimensions. *BoxChart* draws the content of the chart, boxes, whiskers and the dimension dots. The chart content is drawn with the Java library *awt*. The the length and position of the boxes is decided by the five dimension scores. From these five scores we calculate the first and third quartile which is then the bottom and top of a box. The length of the box's whiskers is set by the 1 and 99 percentile values, which is the same as the lowest and the highest scores as we only use five values.
The box chart can be seen in chapter 6, Testing, Analysis and Results.

5.4 SVM Classifier

The classification part of the algorithm is found in the *classification* package. The code is divided into five files. *SVMTestSuite* is the main file that runs all the classification parts. It loads the dataset, sets up parameters, trains and tests the SVM with cross validation. The testing in *SVMTestSuite* is described in more

detail in section 6.2.2. The labeled dataset used is structured to comply with the data parser code. The file can be generated from the scores stored to file from the Data Quality Indicators, and combining these scores with the quality labels running *CombineLablesAndScores.java*. The classification model is trained and built in *SVMTrainer.java* using the Support Vector Machine Library LibSVM, [15]. LibSVM provides a set of tools and abstractions making it easy to test different SVM kernels and parameters without having to struggle setting it up with dependencies like many other Java SVM libraries. in *SVMTrainer* we set the kernel and parameter specifications and train the SVM model which is returned to the test suite. The test suite then provides *SVMPredictor* the trained model and a test set, which is used to predict classes to the test data. The classes is then compared with the true classes, and scores are then generated based on the results. The classifier model can be serialized, if needed to be stored to file. This would be the next step after a sufficient model is found. As you would only want to train the model once, and use it to classify thereafter.

Chapter 6

Testing, Analysis and Results

6.1 Data Quality Indicators

Figure 6.1 shows an example of how the Data Quality Indicator box plot looks. This is the result of running the DQI algorithm on four small example spreadsheets, *Missing*, *Perfect*, *Terrible* and *Various*. The example spreadsheets are listed in Appendix B. Table 6.1 shows the dimension scores for the same spreadsheets.

Figure 6.1: DQI Boxplot example

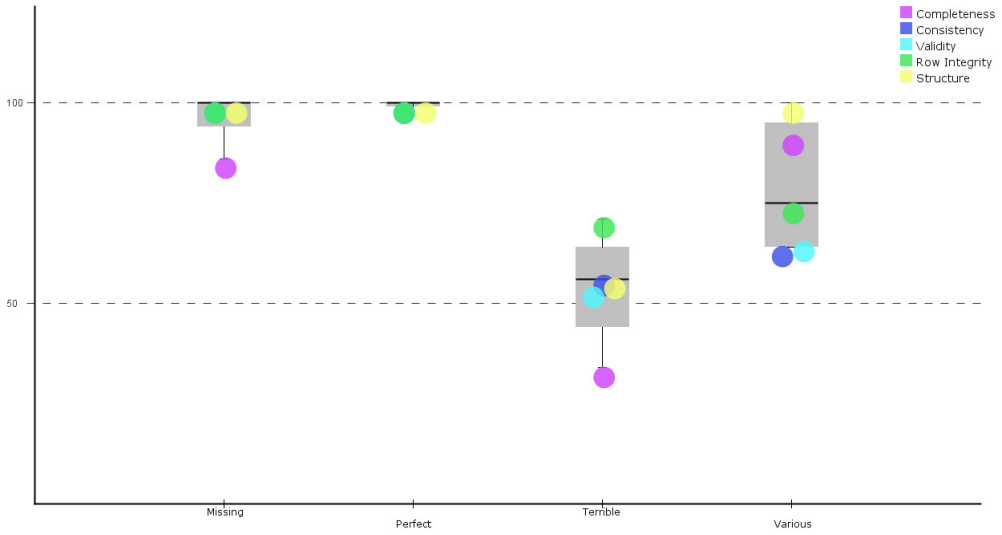


Table 6.1: Score data from DQI run in figure 6.1

Spreadsheet	Row Int.	Structure	Completeness	Consistency	Validity
Missing	100	100	86	100	100
Perfect	100	100	100	100	100
Terrible	71	56	34	57	54
Various	75	100	91	64	65

6.2 Classification

6.2.1 Data process and setup

We start of with a large number of spreadsheets, and a file containing those files pre-classified quality labels. The first step is to give quality scores to all the spreadsheets, we do this by running *QualityIndicator.java*, which runs all the spreadsheets through the quality measures and writes the DQI scores to file. The spreadsheet scores are stored in a text file one spreadsheet per row. Each row starts with the spreadsheet name followed by the spreadsheet's scores. To build and test the classifier, we need the spreadsheet scores together with the pre-classified labels. We map the labels to the scores by running *CombineLabelsAndScores.java*. This class combines the pre-classified labels and the scores, by taking the spreadsheet name and label from one file and mapping the label to a score set with the same spreadsheet name. It's important that no two spreadsheets share the same name, so that the labels are placed with the correct scores. The combined result is written to a new file, where each row consists of a label followed by DQI scores.

6.2.2 Running the tests

With the labels and scores combined, we now have a dataset which we can build and test classifiers on. To run a test, you run *SVMTestSuite.java*, loads the data, trains a model, tests its performance and prints the results. The classifiers are tested with 4-fold cross validation. 4-fold cross validation divides the data set into four sections, one section is chosen to be test set, and the three others are chosen as training set. The training set is used to train/build a classifier model and the test set is used to validate the performance of the model. The cross validation rotates the test and training sets until all the sections have been used as test set. When all the sections have been tested upon, the classifier's score is given by the average score from the cross validation rotations. Cross Validation helps us reduce the risk of picking a test set that could give a biased score, and wrongfully represent the classifiers performance.
When testing we want to test the performance of different SVM kernels. And each kernel has a set of parameters that have to be specified explicitly. The parameters have an effect on how the model performs and will vary from dataset to dataset.

SVMTestSuite tests the classifiers with several different parameters by running cross validation for each parameter combination, and ultimately choosing the parameter combination with the best performance.
The performance is measured on four scores:

$$Accuracy = \frac{TruePositive + TrueNegative}{TruePositive + TrueNegative + FalsePositive + FalseNegative}$$

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

$$F1 - score = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

True Positive = correctly classified "good" quality.
True Negative = correctly classified "bad" quality.

6.2.3 Data

We use 443 real world spreadsheets gathered from several different sources within the Oil & Gas industry. The spreadsheets were made for many different purposes, have a large variety in content, and they represent exactly which type of spreadsheets the the system is meant for. The spreadsheet qualities were labeled by students and employees at *Avito LOOPS*. From classifying the 443 spreadsheets, we have 355 sheets labeled as usable (good quality) this is roughly 80% of the spreadsheets. The rest were labeled not usable. The test results in this chapter are given by training and testing the classifier on these spreadsheets and labels.

6.2.4 Classification results

Table 6.2 shows the scores from a "dumb" classifier, which classifies all the spreadsheets as usable. These results will be used as baseline scores. As you can see the scores are not particularly low, as such a high percentage of the labels are in the "good quality" class.

Table 6.2: Results simple classifier

Classifier	**Accuracy**	**Precision**	**Recall**	**F-Score**
Baseline	0.800	0.800	1.000	0.887

Table 6.3, shows the best kernel results I got from testing the four kernels RBF, Linear, Polynomial and Sigmoid. The results are based on tests finding the parameters that provided the best F-Score for each kernel. They all show a little bit of an improvement on the baseline. RBF and Polynomial provides the best F-Score with 0.895, but as the RBF kernel builds significantly faster than the polynomial kernel. RBF is chosen as the best suited one.

Table 6.3: Results SVM kernels

Kernel	**Accuracy**	**Precision**	**Recall**	**F-Score**
RBF	0.823	0.838	0.963	0.895
Linear	0.795	0.811	0.883	0.883
Polynomial	0.823	0.840	0.960	0.895
Sigmoid	0.802	0.804	0.998	0.888

In table 6.4 we see the variation of scores between sections. The results in this table are given with the RBF kernel with the same parameters used in RBF in table 6.3.

Table 6.4: RBF section varitaton

Section	**Accuracy**	**Precision**	**Recall**	**F-Score**
Section 1	0.818	0.814	0.988	0.892
Section 2	0.909	0.933	0.970	0.951
Section 3	0.827	0.812	1.000	0.896
Section 4	0.764	0.786	0.953	0.862
Average	0.823	0.838	0.963	0.895

6.2.5 Feature selection

With the results from classifying on all the dimensions at the same time. It would be interesting to know what effect the individual dimensions have on the results. We analyse this with selecting different feature combinations (dimension scores to classify on) and see how the results are effected. All the results below were produced with cross validation and finding the optimal parameter combination for each test case. Table 6.5 show the results when testing the RBF classifier with only one dimension feature. Interestingly, all dimensions give the same score as the baseline classifier, except Completeness, which provides a better score than when we classify on all dimensions.

Table 6.5: RBF with a single feature

Dimension	**F-Score**
Row Integrity	0.887
Structural Coherence	0.887
Completeness	0.902
Consistency	0.887
Validity	0.887

Tables 6.6 and 6.7 shows different feature combinations together with completeness. Other than Structural Coherence and Validity, the other dimensions actually have a negative effect on the result. While Validity together with Completeness produced the best results. No other combination of dimensions were able to produce a better score than this.

Table 6.6: RBF with two features

Dimension	**F-Score**
Completeness & Row Integrity	0.878
Completeness & Structural Coherence	0.902
Completeness & Consistency	0.894
Completeness & Validity	0.903

Table 6.7: RBF with three features

Dimension	**F-Score**
Completeness & Structual Coherence & Validity	0.903

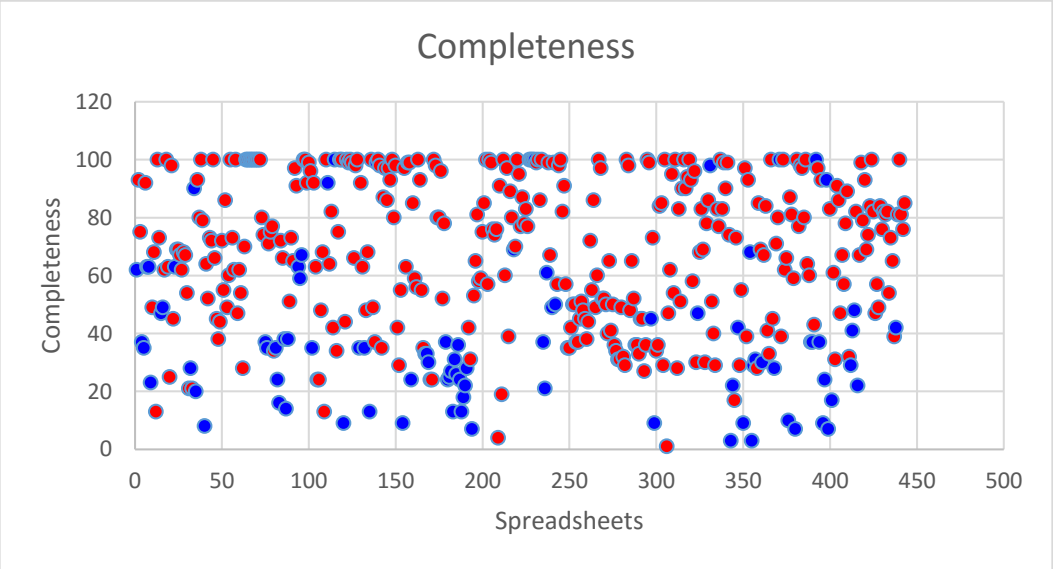
6.2.6 Dimension score and label comparison

Figure 6.2 shows how the data quality dimension scores are distributed and how they line up with the spreadsheet labels. The charts are lined up in two dimensional scatter plots, the red dots are spreadsheets labeled "usable" and the blue dots are the ones labeled "not usable". From the scatter plots we get an indication of how a support vector machine would potentially place a divider to separate the two classes. But as seen from the plots, there's a lack of distinction between class labels and the dimension scores, with the only exception being completeness.

Figure 6.2: Dimension scatterplots

In figure 6.3 we see how the completeness score compares to the labels, where we can see a clearer parting between the two classes than the the other dimension scores.

Figure 6.3: Completeness scatterplot



Chapter 7

Conclusion and Further Work

7.1 Conclusion

This thesis presents the design and implementation of a set of Data Quality Indicators to be used with a Support Vector Machine classifier, to decide whether a spreadsheet's quality is sufficient enough to be used in an ERP system. The Data Quality Indicators are to be used for visualising data quality as well as provide data for the Support Vector Machine. The Data Quality Indicators measure quality on a set of dimensions which represent different aspects of data quality. The dimensions should be able to cover the main quality issues that are prominent in spreadsheet data, to inform the user on where quality can be improved. The DQI dimension scores are evaluated on how well they are able to inform users on quality issues, and how they provide sufficient data for the automatic quality classification.
Section 6.1 shows how the Quality Indicator scores are visualised to the user. The box plot makes it easy for the user to analyze the quality of the spreadsheets in an efficient manner, and compare how the quality ranges and differs between each spreadsheet. From the information given by this chart, the user should be able to conclude why some aspects of the ERP system may not perform as expected, and suggest what parts of the spreadsheets could be improved.
In section 6.2.4 we find the Support Vector Machine results for automated decision making. The results show how the different SVM algorithms perform using the scores provided by Data Quality Indicators and a set of expert labeled spreadsheets. The results show limited improvement on the baseline accuracy. The RBF kernel showed the most improvement with an accuracy improvement by a little over 2%. But this also result in a reduced Recall score, which means some usable data would be discarded along with the unusable. This can always be expected as real life data will always present anomalies, but we have to consider if the improvement is significant enough to justify discarding good spreadsheets.
There's a number of possibilities for the classifier not being able to predict more spreadsheet qualities. For one, the spreadsheet labels were manually classified by several people. This could produce varying and subjective labels, which would ultimately create noise and confusion when training the classifier. Table 6.4 could be an indication of this to be true, as there's a big variation in scores between sec-

tions. Another possibility is that the quality indicator scores may not represent the quality of the spreadsheets clear enough, or that they are not significant when deciding the usability of a spreadsheet. The results show that the most representative quality indicator is the completeness indicator, this is probably because completeness is the most recognizable factor from a human perspective when classifying a spreadsheet. I would expect the classifier to perform better with a more thoroughly evaluated and consistent labeled dataset. The results show that there's a large potential of improvement on the quality measures, and that it would be reasonable to test other Machine Learning classifiers to potentially eliminate certain suspicions for bad performance.

7.2 Further Work

This section presents suggestions on what optimizations could be implemented in the Data Quality Indicators in further work to improve results.

7.2.1 Pattern dictionaries

A lot of the measures are based around having some sort of understanding of what the data is about, to apply measures. With a dictionary of number, price and date patterns, we would be able to better distinguish and recognize data types. With this we could apply the correct measures and design a more robust method for converting string into comparable dates, numbers and prices.

7.2.2 Table detection

Maybe the biggest factor of spreadsheet readability is the amount of tables it contains. Most spreadsheet readers assume the spreadsheet to be read as one big table, but this results in issues when a spreadsheet contain multiple tables. This structure based spreadsheet quality would be of valuable use and improve the overall Data Quality indicator performance.

7.2.3 Extending Validity and Consistency constraints

With testing and analysis of real world data examples we could investigate in which statistics are found in different data types, and use this to apply a more complete and accurate list of data type constraints.

Appendices

Appendix A

Program

A.1 Installation Guide

The system for this thesis is implemented in Java 8.To run it you would need to apply the java source file to a project and include the correct libraries to the build path. The code is dependent on four different libraries: OpenCSV [13], Apache Commons-Math [14], libSVM [15] for the classifier, and lastly *pattern.jar* [9], which should be provided with the code, as it is not currently available to download.

A.2 User Manual

The main method for running the DQI is found in *QualityIndicator.java*. To make it work you would have to alter the variables in the main method to point to a folder with *.csv* files and a file path to write scores to. To store store output in file, set *storeOutputToFile = true*. To show box chart set *showBoxChart = true* The classification part is found in the *classification* package. To run tests on the classifier, specify which paramaters you want to test in the *SVMTestSuite.java* class and run it. You would need a file with correctly structured labeles and features for the file to run.

A.3 File overview

Table A.1: Package and files overview

Package	Files
analyze	ColumnData.java DataTypeDetection.java ValueData.java
classification	CombineLabelsAndScores.java LoadDataset.java SVMPredictor.java SVMTestSuite.java SVMTrainer.java
csv	CSV.java
dqi	Column.java QualityIndicator.java Spreadsheet.java
metrics	Completeness.java GeneralConsistency.java RowIntegrity.java SpecialConsistency.java Structure.java Uniqueness.java Utils.java Validity.java
output	OutputScore.java
visualization	BasicChart.java BoxChart.java ChartFrame.java

Appendix B

Spreadsheet examples

Figure B.1: Spreadsheet: missing

Numbers	Names	Codes	Prices
1	Kjell Petterson	#MISSING	10$
2	Arne Lindmo	#MISSING	11$
3	Svein Harbo	#MISSING	16$
4	-	#MISSING	10$
5		#MISSING	
6		#MISSING	
7		#MISSING	
8	Arne Lindmo	#MISSING	11$
9	Svein Harbo	#MISSING	16$
10	Heisman Haa	#MISSING	10$
11	Alfred Siggurd	#MISSING	11$
12	Ting Skrottnes	#MISSING	16$
	Kjell Petterson	#MISSING	10$
	Arne Lindmo	#MISSING	11$
	Svein Harbo	#MISSING	16$
16	Heisman Haa	#MISSING	10$
17	Alfred Siggurd	#MISSING	11$

Figure B.2: Spreadsheet: perfect

Numbers	Names	Codes	Prices
1	Kjell Petterson	AS-1245	10$
2	Arne Lindmo	AS-1246	11$
3	Svein Harbo	AS-1247	16$
4	Heisman Haa	AS-1248	10$
5	Alfred Siggurd	AS-1249	11$
6	Ting Skrottnes	AS-1250	16$
7	Kjell Petterson	AS-1251	10$
8	Arne Lindmo	AS-1252	11$
9	Svein Harbo	AS-1253	16$
10	Heisman Haa	AS-1254	10$
11	Alfred Siggurd	AS-1255	11$
12	Ting Skrottnes	AS-1256	16$
13	Kjell Petterson	AS-1257	10$
14	Arne Lindmo	AS-1258	11$
15	Svein Harbo	AS-1259	16$
16	Heisman Haa	AS-1260	10$
17	Alfred Siggurd	AS-1261	11$

Figure B.3: Spreadsheet: terrible

Numbers	Names	Codes	Prices				
#ERROR	Kjell Petterson	AS-1245	10$	Extra	Extra	Extra	Extra
#ERROR	null	AS-1246	11$				
#ERROR	null	#VERDI!	16$	Extra	Extra	Extra	Extra
#ERROR	null	#VERDI!	null				
#ERROR	null	AS-1249	11$	Extra	Extra	Extra	Extra
#ERROR	null	AS-1250	16$				
#ERROR	null	AS-1251	10$	Extra	Extra	Extra	Extra
#ERROR	null	AS-1252	11$				
#ERROR	Ola Nordmann						
#ERROR	Ola Nordmann			Numbers	Names	Codes	Prices
#ERROR	Ola Nordmann			1	null	AB_B_2	
#ERROR	12	Ola Nordmann		2	null	AB_B_3	16$
#ERROR	13	Ola Nordmann		3	null	AB_B_4	10$
#ERROR	14	Ola Nordmann		4	null	AB_B_5	11$
#ERROR	15	Ola Nordmann		5	null	AB_B_6	16$
#ERROR	16	Ola Nordmann		6	null	AB_B_7	10$
#ERROR	17	Ola Nordmann		7	null	AB_B_8	11$
#ERROR	18	Ola Nordmann		8		AB_B_9	
		Ola Nordmann					

Figure B.4: Caption: various

Numbers	Names	Codes	Prices
1	Kjell Petterson	AS-1245	10$
2	Arne Lindmo	AS-1246	11$
3	Svein Harbo	#VERDI!	16$
4	null	#VERDI!	null
5		AS-1249	
6		AS-1250	16$
7	Kjell Petterson	AS-1251	10$
8	Arne Lindmo	AS-1252	11$

Numbers	Names	Codes	Prices
10	Heisman Haa	13-1AAB	10$
10	Alfred	13-2AAB	
10			
10	Kjell Petterson	13-4AAB	10$
10	Arne	13-5AAB	11$
10	Svein Harbo	13-6AAB	16$

Numbers	Numbers	Numbers	Numbers
10	13	14	0
10	13	14	0
10	13	1	0
10	13	14	0

Bibliography

[1] Arkady Maydanchik. *Data Quality Assessment.*

[2] Felix Naumann *Data Profiling Revisited* Qatar Computing Research Institute (QCRI), Doha, Qatar

[3] DAMA UK Working Group. *Defining Data Quality Dimensions: The six primary dimensions for data quality assessment* October 2013

[4] Asa Ben-Hur, Jason Weston *A Users Guide to Support Vector Machines* 2010

[5] Brian E Howard, Beate Sick and Steffen Heber *Unsupervised assessment of microarray data quality using a Gaussian mixture model* `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2717951/`

[6] David Isaac and Christopher Lynnes. *Automated data quality assessment in the intelligent archive* `http://disc.sci.gsfc.nasa.gov/intelligent_archive/AutoQualityAssessment.pdf`

[7] R. Sangeetha, B. Kalpana *Performance Evaluation of Kernels in Multiclass Support Vector Machines* International Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, Volume-1, Issue-5, November 2011

[8] Sean Kandel, Ravi Parikh, Andreas Paepcke, Joseph M. Hellerstein, Jeffrey Heer. *Profiler: Integrated Statistical Analysis andVisualization for Data Quality Assessment* Stanford University, University of California and Berkeley `http://vis.stanford.edu/files/2012-Profiler-AVI.pdf`

[9] Morten Wærsland
Text pattern discovery and extraction
University Of Stavanger, June 15. 2016

[10] Clue Points.
`http://cluepoints.com/risk-based-monitoring/`

[11] Roland Bouman, Jos van Dongen *Pentaho Solutions: Business Intelligence and Data Warehousing with Pentaho and MySQL* Wiley Publishing 2009 `http://www.pentaho.com/`

[12] Bahaaldine Azarmi *Talend for Big Data* Packt Publishing 2014 `https://www.talend.com/products/data-preparation`

[13] OpenCSV V3.7 http://opencsv.sourceforge.net/

[14] Commons Math3 V3.6.1
http://commons.apache.org/proper/commons-math/

[15] Chang, Chih-Chung and Lin, Chih-Jen
LIBSVM: A library for support vector machines
ACM Transactions on Intelligent Systems and Technology, volume 2, issue 3, 2011 http://www.csie.ntu.edu.tw/~cjlin/libsvm