Tool for Journalists to Edit the Text Generation Logic of an Automated Journalist

Master of Science Thesis
University of Turku
Department of Future Technologies
Software Engineering
2019
Kyösti Puro

UNIVERSITY OF TURKU
Department of Future Technologies

KYÖSTI PURO: Tool for Journalists to Edit the Text Generation Logic of an Automated Journalist

Master of Science Thesis, 79 p., 10 app. p.
Software Engineering
May 2019

Automated journalism means writing fact-based articles based on structured data using algorithms or software. The advantages of automated journalism are scalability, speed and lower costs. The limitations of it are fluency, quality of writing and limited perception.
In this thesis, the different implementation methods of automated journalism were compared. These implementation methods were templates, decision trees, fact ranking method and different machine learning solutions. It was found out that no implementation method was strictly better than others but all had distinct advantages and disadvantages. When selecting an implementation method these factors should be taken into account and weighed.
Finnish national broadcasting company Yle's automated journalist Voitto-robot was discussed. Voitto's implementation is based on templates and decision trees.
While Voitto's text generation is easily modifiable and transparent due to its implementation method, this was only available to programmers. The decision trees were implemented directly in the code which made them hard to understand and the template files were too complex to be easily edited. In this thesis, a proof-of-concept web application was made to allow journalists and other content creators the possibility to edit the templates and decision trees of Voitto independently.
The created software was analysed and it was found that it helped journalists understand the text generation and modify it as they wanted. Even in its proof-of-concept state, it was good enough to be used to automate election reporting for the Finnish parliamentary election of 2019.

Keywords: automated journalism, software engineering

Contents

1 Introduction

Automated journalism (or algorithmic journalism or robot journalism) means writing coherent and fluent articles based on data using software or algorithms [1]. Automated journalists read structured data, analyse it to find newsworthy facts, and then write an article based on the facts. They write the entire articles autonomously from data gathering to publishing [1]. Automated journalism can be used to write fact-based articles about subjects that can be meaningfully represented with data and good quality data exists [1].

Automated journalism can help with the growing demand for content and reduced resources of newsrooms [2]. The advantages of automated journalism are that it is scalable, fast and cheap and can have fewer errors than articles written by human journalists [1]. In some studies, readers have even deemed articles written by automated journalists as more credible than human-written articles. [1][3]. However, it is not likely that automated journalism will ever replace human journalists, as the automated journalists cannot innovate, they can only write about specific subjects and they always have a limited perception on the subjects [1]. It is more likely human and automated journalists will work together in the future [1][4].

While automated journalism is not yet in widespread use, [5] some of the largest newsrooms in the world such as Associated Press already use automated journalism [1]. There are still issues with automated journalism systems. The readability of automatically created articles is worse than human written equivalents [1]. Articles written automatically are generally less liked and are of worse quality than human written equivalents [3].

In the second chapter, we will go through the basics of automated journalism. This will include the advantages and limitations and different use cases of automated journalism. We will also discuss the requirements that are put on automated journalists for them to be held to the same standards as human journalists.

There are multiple different ways to implement automated journalism. These range from manually writing rules and text templates to machine learning solutions. In the third chapter of this thesis, we will go through the different implementation methods and compare them. The objective is to find the advantages and disadvantages of each method and to find out if different solutions work better for different use cases.

The first research question this thesis will try to answer is: What different ways are there to implement automated journalism?

In the fourth chapter, we will showcase Finland's national public broadcasting company Yle's automated journalist known as "Voitto". Voitto writes articles about sports matches and elections and creates graphics to be used by human journalists. Voitto is an in-house solution that does not use any existing automated journalism software. In this thesis, Voitto's text generation method is presented and analysed.

In the fifth chapter, we will try to create a tool that makes it possible for journalists to edit the text generation logic of Voitto. Currently, the texts are designed by journalists and then implemented by programmers. Journalists cannot independently add texts and they have a hard time understanding the text selection logic. The journalists are also unable to edit the rule-based logic of Voitto. These problems will be solved by creating a tool that can be used to edit the texts and the logic. There already are some existing pieces of software that try to help implement automated journalism. These preexisting tools are also explored to find if any of them solve the problem at hand.

In the sixth chapter, the software is analysed to see how usable it is and to see whether it solves the problem. The analysis is done from both a user experience perspective and by conducting interviews with journalists that used the tool.

The second research question that this thesis will try to answer is: Is it possible to create a tool for journalists to edit an automated journalist's text generation logic?

1.1 Terminology

Decision tree is a special case of a tree data structure. In a decision tree, every **branch node** (a node that has a child or children, aka internal node, parent node) contains a value, that can be used to select one of its branches. For example, in a binary decision tree, every branching node has two branches. Each branching node contains a boolean value (true or false / yes or no). If the branching node's value is true, it selects one branch, otherwise, it selects the other. Binary decision trees can be walked from the root node by checking the values of the branching nodes and selecting the corresponding branches, eventually reaching a leaf node. This is very similar to the use of a flowchart.

A simple if-else if-else structure, where all outcomes return a value, is a decision tree. The branching nodes are the conditions of "if" and "else if" statements. A simple decision tree, displayed as a graph, can be seen in Appendix A.1. A corresponding if-else if-else code fragment written in the Scala programming language (or the Java programming language, implementations are identical in this case) can be seen in Appendix A.2.

Machine learning means computer systems that can learn to perform a task without being given direct instructions. Instead, they are given teaching data which contains input data and corresponding desired outcomes. The system tries to infer how to do things based on the data. Different machine learning systems have different algorithms used for learning, making them more suitable for specific tasks.

Template is used in this thesis to refer only to text templates. Simply put, text templates are text strings that have gaps where values are later placed. For example, a text template might have a gap where a name, time or place is put according to data. These gaps are referred to as **attributes**. A custom "mustache" template language will be used in this thesis' example templates i.e. attributes in templates are surrounded with double brackets. Take the following template as an example:

`{{firstGoalPlayer}} scored the first goal`

In this template `{{firstGoalPlayer}}` is an attribute which will be replaced with the actual name of the player that scored the first goal.

Content creator (or **content producer**) is used as a term for anyone who designs and/or implements the text generation logic or templates of an automated journalist. This person might be a programmer, designer or a journalist or anyone who designs the texts.

Deployment in this thesis will refer to moving the application to a remote environment where it will run. We will also refer to two different environments, where the software can be deployed: test and production. These are generally on remote servers or in cloud services. Users will only ever see the results created by the production environment. The test environment is used to test the software before it is deployed to the production environment.

Personification means creating content from the same topic differently according to the reader's preferences. In the context of automated journalism, this can mean, for example, translating the article to a different language or changing the tone of the article, depending on the reader.

Version control means managing changes done to files, especially the source code of a piece of software. It stores the history of changes done and allows examining the history and returning back to old versions.

2 Automated Journalism

Automated journalism means using an algorithm or software to generate articles based on structured data [2][1]. Automated journalists analyse the structured data to find newsworthy facts from it and then write articles based on the facts. Automated journalism does not require human interaction after the automated journalist has been implemented [1]. The algorithm or software goes through the entire article writing process from data gathering and analysing to writing and publishing the article [1].

For example, an automated journalist might write reports about football matches. The data this kind of automated journalist gathers might include info on how the game progressed, who scored goals and when, and other statistics such as the number of offsides, corners and fouls. The automated journalist will then analyse this data and try to find the newsworthy facts of it. Newsworthy facts might be how the game progressed and highlighting important players, based on the player stats in the data. This analysis answers the question “What is said” in the article [6].

The previously selected newsworthy facts are then turned into fluent text and the article is formed. This part of the process answers the question “How are they said” in the article [6]. The automated journalist will finally publish the finished article.

The articles written by an automated journalist do not need to be used as-is, they can also be used as a base where journalists add more information [1]. Automated journalism will most likely not replace all journalists, but perhaps allow them to focus on in-depth analysis and other work that is hard or nearly impossible for automated journalists [1]. In future, automated and regular journalists will likely work together [1][4]. For example, the Los Angeles Times' Quakebot only writes drafts that are then checked and published by humans [1]. This saves time while being less error-prone, which is important due to the delicate subject matter (reporting about earthquakes).

Automated journalism has already seen a fair share of use [1][2]. Some of the largest news agencies use automated journalism. For example, Associated Press (AP) uses it for financial reports and sports reporting in smaller leagues [1]. Other major users of automated journalism include Forbes, New Yorker and Los Angeles Times [2]. These newsrooms have stated that their reason for using automated journalism is the reduced resources and growing demand [2].

However, it cannot be said that automated journalism is widely adopted [5]. Companies that provide automated journalism solutions have reported that they have few customers [1]. Automated journalism is considered to still be "experimental" or in an "early-market expansion phase" [5]. According to Andreas Graefe, this might quickly change [1]. This is due to there being more and more high-quality data available and newsrooms being pressured by consumers into creating more content cheaply.

2.1 Limitations

The major limiting factor for what automated journalism can be used for is the necessity of data [1]. As the whole purpose of automated journalism is to write articles based on data, there are many requirements on the data. The data has to accurately describe the topic and it has to be of good quality [1], otherwise, the news will not be accurate. The data should also be timely, or the news will be late and be less newsworthy. The data also needs to be machine-readable [1].

What the data contains depends on the use case. Generally speaking, it should contain everything required to make newsworthy articles about the subject [2]. The data should support writing interesting articles based solely on it. The data available puts boundaries on the possibilities available to the automated journalist: it is not possible to write about things not in the data. Therefore the design of the automated journalist needs to start with going through what data is available and what is not [2].

Another limitation is that reports written by an automated journalist have to be repetitive i.e. multiple articles written from the same subject [1]. There is no advantage to an automated journalist if it does not write multiple articles. Otherwise, it is most likely faster and cheaper to write the articles by hand. This means that automated journalism has to be about a subject that requires continuous reporting.

The repetition can also come in the form of personification [2]. Personification requires writing an article multiple times for different readers, where automated journalism can help. Personification in journalism can mean that the article is, for example, written in multiple languages or more focused e.g. an election report for each municipality instead of writing a single report about the entire election.

Unlike human journalists, automated journalists have to predict what to write about. They cannot react to it [7]. This is due to the fact that the automated journalist has to be implemented beforehand. Therefore the subject matter has to be somewhat predictable.

2.2 Use Cases

There are many different use cases for automated journalism. Most common automated journalism use cases are weather, financial and sports news [2]. In financial and weather news the entire news can be derived from the data. In sports, it is relatively easy to collect data from the match and turn the data into articles, even though there are aspects of sports that cannot be represented with data (we will return to this later on in this thesis). Another common use case is election reporting, as elections results can be easily represented with data. All of these examples are also repetitive: Weather reports are written continuously, financial reports are often yearly, half-yearly or quarterly, there are many sports matches and elections happen every few years. Because sports and elections are common use cases, they will be used as examples throughout this thesis.

There are also some less common use cases. For example, the Los Angeles Times uses automated journalism for murder and earthquake reporting [1]. In both of these use cases, the limitations of automated journalism are not a problem: there are (unfortunately) multiple homicides and earthquakes in Los Angeles so the reports are repeated. Also, there was data available from L.A. County coroner's office and U.S. Geological Survey's Earthquake Notification Service respectively which made the automated reports possible [1].

2.3 Advantages

There are many advantages of automated journalism. Most of these advantages are similar to other automation tasks: scalability, costs reduction, speed and precision.

Scalability is an obvious advantage of automated journalism. If there is an automated journalist that can write an article about a subject, it can be used for writing other articles about the same subject, if they use the same data. For example, creating an automated journalist that handles a single sports match or a single election can handle other similar matches or elections as well. Automation allows writing articles that were previously impossible to cover due to limited resources [1]. Also, it is much easier to manage if multiple articles must be made at the same time because efforts don't have to be coordinated. Automated journalists can also handle larger sets of data than what is possible for human journalists [2].

Automated journalism is generally faster as the automated journalist does not have to type or stop to think. Articles written by an automated journalist can be published almost as soon as data is available [1]. Reporting the news quickly is important as it ensures that the article maintains newsworthiness. An example where speed is of the essence is the previously mentioned Los Angeles Times Quakebot which reports about earthquakes in the Los Angeles area [1].

Automated journalists can also be more precise. Automated journalists do not make simple mistakes or neglect important facts unless they are implemented incorrectly [1]. They do not misspell things and they do not tire [1]. However, there still are mistakes made by automated journalists. They can often happen due to unforeseen consequences. For example, stock being split was reported as the stock losing value [1]. When designing the text generation of an automated journalist, predicting possible anomalies should be done thoroughly, even though it might be difficult [7].

Automated journalism can also help with the personification of news [2]. The scalability of automated journalism also lends itself to personification: making articles according to certain needs or interests requires writing a lot more articles. An example of such a use case is “Vaalibotti Valtteri” by Leppänen et al. [2]. Vaalibotti Valtteri wrote election news about Finnish municipal elections of 2017. The articles could be modified to be about a single party in a single municipality. [2]. If this would be done with human journalists it would require writing thousands of articles which is not feasible. The articles can also be written in multiple languages [1][2]. Vaalibotti was able to write the same article in Finnish, Swedish and English. In the extreme case, personification can mean that articles are written for one person [1]. Automated journalists could also focus on different aspects of the article or change the tone according to the reader’s preferences [1].

In addition to personifying articles to certain needs, automated journalism can also provide “news on demand” [1][2] meaning that news is created automatically according to user input. Vaalibotti Valtteri is an example of this also: the reader can select a party and a municipality and the article written will focus on the selected municipality and party [2].

According to Andreas Graefe, while some advocates promote automated journalists as being more objective, this might not be the case [1]. Automated journalists always use the same algorithms so they should not have a bias, but the people implementing them might have a bias, which could have transferred. The data also has to be unbiased for the article to be unbiased. However, data suggests that people perceive automated journalists as being more credible [1][3]. Also if articles are personified according to the reader's preferences, then they cannot also be unbiased [1]. An automated journalist can be unbiased, but it is not always the case and cannot be taken for granted [1].

Automated journalism could allow human journalists to write more interesting articles, as simple and repetitive articles can be automated. This allows human journalists to focus on articles the automated journalists cannot write, such as analytical pieces. [1]

2.4 Six Requirements

Leppänen et al. listed 6 requirements that automated journalists should adhere to [2]. Automated journalists need to follow the same standards as human journalists. Human journalists are expected to write truthfully and accurately, their text must be fluent and their subject newsworthy. Any automated journalist that fulfils these six requirements should uphold the same standards. The requirements are presented in the following section.

While these requirements might not be comprehensive, they provide a good baseline for comparisons. Therefore they will be used in this thesis to compare different implementation methods of automated journalism and to plan and evaluate the practical part of this thesis.

2.4.1 Transparency

Transparency means that the process of creating a text from the data is clear and open. For example, it should be clear why the automated journalist decided to mention specific aspects of the data. This is comparable to human journalists being open about the way an article was written, e.g. mentioning the use of reliable sources.

This requirement can sometimes be achieved by having the source code of the automated journalist be open-source. However, that can be uncommon due to the fact that commercial solutions may lose their competitive advantage if they reveal their source code. Some times being open-source does not help due to the implementation method being a "black box" i.e. the inner workings of it cannot be directly observed.

2.4.2 Accuracy

Accuracy means that the text should be truthful to the data it was made from. This is not in any way different from human journalists.

While not specifically mentioned in the report by Leppänen et al. we would also add that not saying something important is also indicative of poor accuracy [2]. This is similar to human journalists omitting important info, even if it is more likely not purposeful when it comes to automated journalism. Omitting facts can happen accidentally more often as the automated journalism system has to predict what to write about as was discussed earlier.

2.4.3 Modifiability and Transferability

Automated journalists should be easily transferable from one subject to another, just like human journalists can write about many subjects. This makes them much more generally usable. If the automated journalist is easily modifiable it will also be more easily transferable. Designing the system to be modular generally helps adhere to this requirement as it allows modifying only some modules when switching to a new context.

2.4.4 Fluency of Output

Automated journalists are expected to write fluent text just like human journalists. Fluency means that texts written by a computer should not read like they were written by a computer, but instead, they should be as easy to read as human-written articles.

2.4.5 Data Availability

The workings of automated journalists rely completely on data. Therefore the data used by the automated journalist must be newsworthy and of good quality. Otherwise, the automated journalist is unable to write accurate and interesting enough articles. This is comparable to a journalist having to have enough info on a subject before writing an article.

2.4.6 Topicality of News

Just like regular journalists', the automated journalists' texts should be topical. As an automated journalist is not as suitable for long-form articles or complex analysis, topicality is often achieved via personification, localization or just timeliness. This means that the automated journalist either writes articles meant specifically for certain readers or they write about events that just happened.

2.4.7 Discussions

We think that splitting modifiability and transferability into two different requirements should be considered. A system can be easily modifiable without being easily transferable and vice versa.

For example, a text generation system based on a decision tree can be modified by adding new branches or editing existing ones. However, changing from one subject to another will probably require the design and implementation of a completely new decision tree. The modular design will help as you only have to change the module that converts data into values used in the tree and the tree itself, but it might still be plenty of work.

The difference between modifiability and transferability can also be perceived the other way around: An easily transferable system might not always be easily modifiable. As an example: A perfect machine learning system might be able to turn any data into a coherent article with minimal work, but it might not be at all modifiable as machine learning is often a "black box" meaning that its inner workings cannot be inspected or modified directly.

Because of these reasons modifiability and transferability will often be discussed separately in this thesis. Although as stated previously modifiability often helps with transferability so there are also reasons why these requirements are grouped together.

While it was not stated in by Leppänen et al. we would add that fluency also requires for different articles by the same automated journalist to be sufficiently different from each other [2]. For human journalists, this is pretty much given, even though articles from the same subject will have similarities. However, for automated journalists, different articles being samey is much more likely because all the articles are created with exactly the same process. While the text itself might be fluent when reading a single article, when reading multiple articles it might become too obvious that the text was written by an automated journalist.

This issue came up in studies conducted by Melin et al. where humans read articles that were based on the same data written by humans and automated journalists [3]. At first, readers had a hard time determining which articles were written by humans and which were written by a robot. After reading a few articles by both sources, readers got better at recognizing the writer. This would hint that automatically written articles bear more similarities with each other than human-written ones. This type of fluency may not be required in cases where automated journalism is used for one-time article personification (e.g. different election result articles for each municipality) as people will probably not read multiple articles.

While in the paper these requirements were introduced as equals, we think that some are more important than others. We think accuracy is the most important of all of the requirements as the point of articles is to tell what happened. An inaccurate automated journalist can be very problematic. We think transferability is very case dependent. If the goal is to use an automated journalist for a single use case, this requirement is not as important as others. The importance of different requirements depends on the use case. This will be discussed more when different implementation methods are analysed and compared.

3 Implementing Automated Journalism

In this chapter, we will introduce and compare the different ways to implement automated journalism. Automated article generation is divided into two separate problems: “What is said?” (content selection) and “How is it said?” (surface realisation) [6]. There are also solutions where a single end-to-end process handles both tasks i.e. they handle the text generation as a single process from start to finish.

Due to the fact that most automated journalists deployed in practice are implemented by private companies using closed-source products, there is little information on the implementation methods used [2]. Therefore there might be other implementation methods not described here. However, most implementations seem to be template and rule-based [2].

3.1 Content Selection

Content selection means trying to find the most important facts to mention in the article. Basically, it means finding the most newsworthy aspects of the data.

3.1.1 Decision Tree

The decision tree data structure introduced in Section 1.1 can be used to select the texts the automated journalist outputs. The data used by the automated journalist is refined into boolean values. These boolean values are placed into the branch nodes of a decision tree and texts are placed into the leaf nodes. When an article is created, the tree's value is determined by calculating the boolean values from the data and then determining the leaf node that should be selected. The creation of the tree is most often done manually [8].

A decision tree is a rule-based system as the branches are rules which determine which template to select. There are most likely other hard-coded, rule-based systems for text selection which are closed-source, as most solutions appear to be rule-based [2]. Unfortunately, there is little info on them, but if the rules are made manually they most likely have the same advantages and disadvantages as the decision tree model.

The decision tree is often designed and implemented by programmers, journalists or even data analysts working together [1]. The design of the decision tree decides the newsworthiness of different aspects of the data [8]. The nodes at the top of the tree determine what is the most important thing to take away from the data. The decision tree can also be extracted automatically with machine learning [9].

3.1.2 Fact Ranking Method

The fact ranking method means finding significant, comparable (numerical) facts from the data and comparing them to find outliers (e.g. facts that are furthest from average). These facts are then selected and an article is written based on them. [2]

This type of system was used in the Vaalibotti Valtteri automated journalism project. Possible facts were, for example, the number of votes a party or a member received, the change of votes compared to last election etc. A newsworthy fact could be getting the most votes, losing the most votes compared to previous election etc. The facts can also be used to create other facts. For example, in elections, the number of votes can be used to rank the parties to create a new fact about rankings e.g. party ranked first or second. [2]

3.2 Surface realisation

Surface realisation means changing the data into readable text. After selecting what needs to be said, the selected facts have to be changed into fluent text. Aside from end-to-end solutions, templates are the only way used to turn data into text that we are aware of.

3.2.1 Templates

The text templates introduced in Section 1.1 are often used to represent data as text [2]. Templates can range from simple to complex [2]. Simple templates can form just a fragment of a sentence while complex templates can be multiple sentences long. Simple templates are generally more universal and portable, but they might be less vibrant than templates that only fit specific cases [2]. For example, simple templates were used in Vaalibotti Valtteri. Having more general templates also means that fewer templates are needed, which makes translations easier. The entire article can be a single template [8]. In fact, most newsrooms use such templates, which are hard to transfer and require rewriting everything when changing domain or language [2]. All in all using templates can be a bit of a balancing act between portability and fluency [2].

Even seemingly minor things can make templates less reusable even in a relatively similar context. For example, using the term “score a goal” in a team ball sport (such as football) means that the same template cannot be used in basketball.

Templates can also be extracted automatically using machine learning [9]. Machine learning systems can read articles and then extract sentences and figure out the attributes in said sentences, creating new templates.

Improving Templates

Templates don’t have to be used as-is; they can be processed before they are added to the article. These processes can make the template-based text more natural and fluent. [2]

If the templates are simple, one-sentence templates, they can be combined automatically with conjunctions [2]. This makes the text more natural as it is not just a list of simple one-clause sentences. In Vaalibotti this type of system was used. However, it did create some strange and even factually inaccurate sentences. An example by Leppanen et. al.: “Aggregating ’X got most votes’, and ’X got 5 seats in the previous elections’ into ’X got most votes and 5 seats in the previous elections’ changes the meaning of the first text snippet” Nevertheless, this kind of system can be used to combine sentences, as the problems found in Vaalibotti are just bugs that need to be ironed out [2].

In the Vaalibotti project, a piece of software called a “Recurring expression generator” was used with templates to make the text more natural. The software would recognize that a certain attribute value (such as a person’s name) was mentioned multiple times and change the following mentions accordingly. For example, if a specific election candidate was mentioned many times, the first time would use their full name, but the second time it would refer to them by last name alone or say “he”/”she” in place of saying their full name multiple times. [2]

Templates can have synonyms written into the template [9]. This means that the template can have multiple options in place of a word. The selection of specific synonym can be random, or it can take the data into account. For example in the template

```
{{winningTeam}} won {{losingTeam}}
```

the word "won" can be replaced according to data, e.g. if the winning team won with many goals, it can say

```
{{winningTeam}} crushed {{losingTeam}}
```

Exact synonyms can also be selected randomly to improve the fluency by adding variation to the articles.

3.3 End-to-End Solutions

End-to-end solution means that a single software takes the data and outputs natural language, without there being a way to divide the process into two parts [6]. The most prominent example of these are different machine learning systems.

3.3.1 Machine Learning

Machine learning systems can be used to implement an automated journalist. These types of systems are fed a huge amount of teaching data. The teaching data includes the structured data, from which to write the articles, and also articles that are written based on the same data [6].

The requirement to have enough teaching material is an issue in the machine learning systems [6]. There needs to be a plethora of articles and data related to those articles. For example, Wiseman et al. introduced a set of over 4000 basketball articles with scores and other match data [6]. The data contained player stats such as points scored, rebounds and assists, and team stats such as points scored and previous games won and lost.

Wiseman et al. also used the data to compare many different machine learning implementations. The different machine learning systems had different features that would improve the outcome by changing the learning process. The machine learning implementations all had features where they would copy words from the data directly for improved accuracy. These words would generally be names of players and teams, and scores. There were two different copying mechanisms: Joint Copying and Conditional Copying. These differed in the way the machine learning determined whether or not to copy a word or to generate a new word. [6]

A reconstruction based technique was also used to improve the results. This means that during training the machine learning system reconstructs the database from the output text. The reconstructed database is compared to the input data and scored. If the reconstructed database is similar to the input database, the score is better. Total variation distance (TVD) penalization can be added to the reconstruction loss. This means that there is a penalization if different reconstructed databases are too different from each other. [6]

Some of these techniques were combined and the tested combinations were:

- Joint Copy
- Joint Copy + Reconstruction
- Joint Copy + Reconstruction + TVD
- Conditional Copy

Of these models, the Conditional Copy model and Joint Copy with Reconstruction and TVD performed best. However, even the best models had noticeable issues with accuracy with only approximately 71% of records being correct. [6]

Nie et al. were able to improve upon these with a machine learning system called Operation guided Attention-based sequence-to-sequence network (OpAtt). They used the same teaching data as Wiseman et al. so the results are comparable. OpAtt adds additional processing to the input data before text generation. It also uses copying models to achieve better results. OpAtt was able to improve both the correctness and the fluency of the output. It had approximately 81% of records correct on the same dataset, which is an improvement but still quite far from solving the accuracy issues of machine learning. [10]

3.4 Comparisons

In this chapter, we will use the previously introduced seven requirements of automated journalists to compare the different implementation methods. Modifiability is separated from transferability for this comparison for reasons discussed in Section 2.4.7. While data availability and topicality of news are not really related to the implementation method used [2], the other five requirements can be used to compare different implementation methods.

Unfortunately, not much research is done comparing different implementation methods. While the subject of different implementation methods has been researched there is little research comparing them. An automatic system has been implemented for comparing automated journalists which will be discussed later. Though, it has not been used to do a wide analysis yet.

For a lot of these different implementations, the accuracy and fluency depend on the humans that create the actual content. For the sake of argument, we will compare the best possible result, but also take the human element into account. We will leave the consideration on whether the systems are open-source or not out of this comparison and only compare the transparency of the actual systems themselves.

3.4.1 Decision Tree

Decision trees have poor transferability, as they are based on a manually created set of rules which determine what is newsworthy and what is not. While some parts can be reused if the change of topic is to another similar topic (e.g. from one sport to another, from one election to another), if the domain is changed altogether, the decision tree has to be created from the beginning [2]. Creating the new decision tree does not only require redesigning the tree but also implementing all the boolean variables used in it. Therefore it is expensive to transfer from one subject to another using decision trees.

As the decision tree needs to be manually implemented, they have perfect modifiability. The accuracy of decision trees depends on the implementation. If not reporting something newsworthy is considered poor accuracy, decision trees often have at least some problems with accuracy, as it is often expensive to create rules for everything. Also, there are probably inaccuracies due to the fact that the tree was manually implemented, so there are likely to be some mistakes. Fluency depends on the templates used with the decision tree. Decision trees have good transparency as it is easy to see why a specific template was selected.

3.4.2 Fact Ranking

Fact ranking has better transferability then decision trees as there are generally fewer facts to be ranked than decision tree decisions. However, some work is still required to move from one subject to another, as the different facts being read from the data need to be implemented. The modifiability of the fact ranking method is slightly worse than the modifiability of decision trees, as one cannot directly state that one fact is more newsworthy than another [2]. It is possible to edit the importance of different facts, just not to such a fine-grained level.

The accuracy should be good as it is easier to have all the facts in the article and it is less likely that there are bugs as not all decision need to be programmed manually. The transparency is quite good, but it is not as good as a decision tree's [2] as decision trees are easy to read. On the other hand, if there are multiple automated journalists, this might flip the scales, as the human journalists do not have to consider different decision trees for different journalists [2]. Fluency depends greatly on the templates used with the fact ranking method.

Unfortunately, fact ranking method does not work in all use cases [2]. It does not work for storylike articles where one thing leads to another, such as a sports match. It is more suited to articles where it is more important to point out the outliers. Election news is a good use case as the story often is who got the most votes, who gained and who lost the most. Not being able to be used in all articles is the biggest detriment of this method. Nevertheless, of the common automated journalism use cases, fact ranking method fits weather, election and financial news.

3.4.3 Templates

The transferability of templates depends on the templates. Simple templates that just state a single fact are more transferable while specific templates are not as transferable. This due to the fact that fewer simple templates are needed to write an article and therefore fewer templates need to be rewritten when changing domain or language. The fluency of templates also depends on how complex they are. More complex templates can have more flavour, at the cost of transferability [2][8].

Templates have excellent modifiability, as they are manually written and can be edited. They also have good transparency as they clearly state what is said. The accuracy of templates depends on the part that implements "what is said". As templates are implemented manually they can have accuracy issues due to human errors.

The texts generated by Vaalibotti Valtteri were analysed with humans readers. The words "Repetition" and "Listing-like" were respectively the second and the fifth most common negative words used. The implementation based on repeating simple templates may have caused these issues. It would appear that the problems of template-based solutions only crop up when reading multiple articles. The readers were better at figuring out which texts were written by humans and which were written by an algorithm, after reading multiple articles. [3]

3.4.4 Machine Learning

Machine learning should have good transferability as the same machine learning system can be used in different use cases, it just has to be taught with different data. Modifiability is quite poor as it is not possible to directly edit the system. It can only be changed by editing the parameters of the teaching process. The fluency of machine learning systems is good, as they learn from articles written by human journalists. The accuracy of machine learning systems is relatively bad. The transparency of machine learning systems is minimal, as they are a "black box" [2]. It is difficult or even impossible to say why a machine learning system decided to use a certain phrase or mention a specific fact from the data.

It is difficult to find enough teaching data for machine learning systems [6]. In a perfect scenario, the articles in the teaching data would have no info that is not in the data, otherwise, there are facts in the article that are not supported by the data [6]. This might affect the quality of the machine learning system. However, it is difficult to find a large number of articles that are written based solely on structured data with no additional info.

3.5 Automated Comparisons

Wiseman et al. created an automated system for comparing different text generation methods and especially comparing different ways machine learning can be used to implement automated journalism [6]. The test used two data sets: one with data from over 4 000 basketball matches and statistic heavy articles written from the same matches and another with over 10 000 basketball matches and matching articles that are less formal [6].

For figuring out how well the different systems performed, an automatic system was created. Human evaluation was argued against, as it was stated by Wiseman et al. that “we believe that current text generations are sufficiently bad in sufficiently obvious ways that automatic metrics can still be of use in evaluation”[6]. Instead, they used an automatic system which used metrics known as Bilingual Evaluation Understudy (BLEU), Content Selection (CS), Relation Generation (RG) and Content Ordering (CO). The systems extract facts from both a generated article and from an article written by a human journalist about the same match. [6]

BLEU is a metric that is designed for testing machine translations [11]. It compares human written translations to machine translations and the closer they are the better the BLEU score is. While BLEU is designed for comparing human and machine translations, it can also be used to compare human written and machine written articles [6].

BLEU mostly checks how fluent and adequate the translation is [11]. However, Wiseman et al. noted that BLEU does not really take into account "What is said" i.e. it does not check that the report mentions the most important aspects of the data [6]. Content Selection amends this as it checks what facts are mentioned in the generated article and compares those to the facts mentioned in the corresponding article written by a human journalist [6]. Relation Generation checks that the facts are correct by comparing the facts in the generated article to the data the article was based on [6]. Content Ordering checks that the order in which the facts were mentioned in the generated article is similar to the order of facts in the human-written equivalent [6].

The test was conducted on multiple different machine learning models and a template-based solution. The outcome of the experiment was that while the template-based solution had a lower BLEU score (less fluent) and was slightly worse at selecting the correct facts, it performed better on all other metrics. [6]

However, the template-based reference implementation is extremely basic: it only states the results and then lists the six biggest scoring players one by one with exactly the same wording and finally a sentence about both team's next match [6]. Not surprisingly, the text is not very natural and scores low points in fluency. It should be tested with a more complex template-based system with natural language templates. The template-based system did not appear to have any kind of template selection system, but always used a single template, which might reduce its Content Selection score.

The automated comparison of article automation methods could also be used to test a decision tree-based system vs. a fact ranking system. For example, a decision tree system and fact ranking based system could both be used in an election data case and compared against each other. The work needed and the costs to get each of them working could also be compared. It must be noted that other article generation methods might not be sufficiently bad enough for the comparison methods used by Wiseman et al. to work.

These tests are outside of the scope of this thesis as they would require implementing multiple automated journalists using different methods.

3.6 Discussions

None of the implementations of automated journalism compared is strictly better than others. Machine learning is very fluent, but has issues in accuracy and requires teaching data. Text selection with a fact ranking method is quite fluent and transferable but doesn't fit a story-like article structure where events happen in order. In Table 3.1 there is a summary of different implementation methods. While it is very difficult to state objectively how good or bad different aspects of the systems are, the table works as a guideline for comparing the systems to each other.

Some criticise machine learning systems as being inferior and not viable compared to template-based systems due to their "black-boxness" and poorer accuracy [8]. However there is constant research into better machine learning systems [6][10] that improve upon the accuracy, so this might change eventually.

It is difficult to state how fluent and accurate decision trees and templates are. It is possible to edit and polish template files and decision trees to perfection. In theory, it could be possible to replicate a machine learning-based text generator with decision trees and templates, although this would be extremely expensive. Improving decision trees and templates also has diminishing returns: Each new branch in the decision tree has less value as the addition is relatively smaller. Each completely new segment added to an article also has less value as it is a smaller relative increase to the entire text of the article.

Adding new branches to the decision tree makes the logic more and more complex, which makes it harder to grasp and less transparent. Adding templates generally requires a lot of manual work. The goal of this thesis' practical work is to make editing templates and decision trees much easier, to reduce these problems.

Table 3.1: Comparison of different implementation methods

	Transf.	Modif.	Accur.	Transp.	Fluency
What is said?					
Decision tree	Bad	Good	Good	Good	Medium - Good
Fact ranking	Medium	Medium	Good	Medium - Good	Medium
How things are said?					
Complex templates	Bad	Good	Good	Good	Medium - Good
Simple templates	Medium	Good	Good	Good	Medium
End-to-end solutions					
Machine Learning	Good	Bad	Medium	Bad	Good

It is harder to translate machine learning-based automated journalists than template-based solutions as they would require data from sports matches and articles from the same matches in all languages for which translations are desired. For template-based solutions, it is always possible to add new languages by just translating the templates, although that does require manual work.

While strides are being made to create better and better machine learning systems[6][10], it is difficult to say if machine learning systems can replace all other implementation methods. As was stated previously, the biggest hurdle of the machine learning solutions is the teaching data. There seems to be only a handful of data sets used in research (a total of six that we know of [6][10]) in English, which is one of the most spoken and written languages. For smaller languages, it might not be feasible to find enough articles and matching data to create viable machine learning solutions based on them.

3.7 Selecting an Implementation Method

If there is plenty of teaching material available and being in complete control of the automated journalist is not vital, machine learning is probably the way to go. Especially if there is a requirement for different unrelated use cases as the transferability is better.

If the automated article is more about finding the outliers in the data, the fact ranking method with templates is probably the way to go. These types of use cases could be election or financial news. This approach gives more control as it is clear how things are said, but deciding what is said is perhaps harder to grasp and balance. It might be more difficult to give an explicit ranking on how newsworthy some aspects are as the fact ranking method just find outliers.

If the article has a story-like approach, or there is an explicit ranking of newsworthiness (i.e. some aspects are always more important to mention than others) a decision tree-based approach is most likely the best implementation method. These types of use cases are, for example, sports matches. This approach gives the tightest control of the outcome as it has clear rules on what is said and how is it said.

Also as stated previously, decision trees and complex templates are not transferable, if they are fluent and have good accuracy (accurate as in everything newsworthy is written) [2]. Therefore they might not be the best solution if the goal is to create multiple different automated journalists that write about different subjects.

When selecting a suitable implementation method for a use case one should consider all the requirements for an automated journalist and weight them according to the use case. For example, if only one automated journalist is desired, transferability is not too important. After the requirements are weighed it should be easier to see what implementation fits the use case best.

3.8 The Common Challenges

There are many ways to implement automated journalism and all of them have their own challenges. However, some challenges persist over different implementation methods and use cases.

It is difficult to create implementations that are truly portable [2]. While some parts of text generation are reusable, none of the different implementation methods really supports changing the subject matter without requiring plenty of work. While simple templates and a fact ranking text selection can be quite portable, it doesn't really support all kinds of article types and still requires some work when changing domain. Machine learning systems can be portable, but they require an abundance of data and matching articles to be able to learn how to write articles based on the data.

While automated journalists can look for correlation in data and find interesting relationships, they cannot provide any real analysis. They are unable to ascertain why these relationships exist [1]. In other words, the automated journalists can tell what happened, but they have a hard time telling why it happened [1].

Earlier in this thesis, we asserted that not saying something important is also indicative of poor accuracy. It would seem that most of the previously explored implementations will miss very rarely occurring events. Human journalists can react to unexpected events while automated journalists have to predict them as the automation has to be implemented beforehand [7].

For example, if multiple hat tricks were scored in a football match, most implementations would miss it. A machine learning system probably would not have seen enough matches with multiple hat tricks to be able to generate text about it. A decision tree-based system would not necessarily have a rule for it, as it happens extremely rarely. A fact ranking system might pick it up, but then it would require one ranked fact to be the number of hat tricks and it may not know to underline how important this is. As stated by Andreas Graefe "– contribution is limited to providing answers to prewritten questions by analyzing given data." i.e. algorithms lack ingenuity and cannot innovate [1]. Therefore algorithms have limited perception and cannot come up with anything new, unlike human journalists [1].

The problem of not writing about rarely happening, yet newsworthy occurrences is a problem that could probably be easily fixed with human and automated journalists working together. While an automated journalist might miss the rare occurrence it should be obvious to the human reporter and they could update the automatically generated article with the additional info. This again speaks for the advantage of having both human and automated journalists working hand in hand. The automated journalist can write about the scores and all other data-based text and the human journalist can add everything important outside of that.

When compared to the way humans write articles, it is difficult to write as lively articles from data alone [7]. For example, if a human journalist watches a football match, they might see an incredible goal and highlight that in their article. It is quite difficult to accurately portray this information with just data as these are not hard, numerical facts but are up to debate. Simply put it is hard to put emotion into data. While this is not a problem in all use cases, it is definitely an issue in sports journalism, which is one of the most common automated journalism use cases. The data also lacks a human element which interests readers [7]. This again is a problem that has been solved with a human journalist adding a "human touch" to the automated articles [1].

In addition, the amount of data that can be used is usually limited while human journalists might have all kinds of important info that can be used in the article [7]. For example, human journalists might know something about players' history of injuries or personal life. While these facts are not vital to writing the article, they do give it some spice. It would be impossible to give all the data to the automated journalist, even though it would help with writing more human-like content. In Thurman et al. a journalist noted that in sports the events that happen at the pitch might be less interesting than other events surrounding the game that are not in the data [7]. For example, in a football match between West Ham United and Manchester United in 2016, there was a riot before the match which delayed it, and after 112 years it was the last time West Ham played at Upton Park Stadium [7]. This info would most likely be missed by all automated journalists.

4 Case Voitto-Robot

Voitto is a virtual robot that is created by the Finnish national broadcasting company Yle. It is a sort of hypernym for multiple different projects. All of these projects have to do with automation, personification or assisting journalists or users. Automated journalism is one of the first things Voitto started with. Since then the Voitto-name has been used in automatic goal clip generation from football and using machine learning to recommend articles to readers. In this thesis, we will only focus on the automated journalism aspect of Voitto as the other aspects are unrelated to this thesis.

4.1 Acknowledgements

We have worked in the Voitto robot project for over a year. Most of this chapter is based on personal experience and discussions with other members of the project team. Most implementation details can be verified from the documentation and the source code of Voitto available in GitHub[1]. Some of the features presented in this chapter are in newer versions of Voitto and therefore cannot be found in the open-source repository.

4.2 Articles

Voitto has mostly focused on sports articles to this point. Only non-sport articles Voitto has written were about elections in Finland. In these elections, Voitto wrote about the

[1]`https://github.com/Yleisradio/avoin-voitto`

results of all municipalities in Finland. The sports Voitto writes articles about are ice hockey, football and floorball.

In addition to writing articles about single matches, Voitto also writes automatically updating sports guides that contain the latest results, upcoming matches, league table and top player's stats. These guides also have links to the individual articles written by Voitto. Also during the FIFA World Cup of 2018 Voitto published preview articles 30 minutes before matches started. These preview articles contained a short intro about which players to follow and who was predicted to win. More importantly, they had embedded live streams of the match. On days with multiple matches, Voitto also writes aggregates that go through all matches played on that day.

The goal of Voitto is to write articles that would be too expensive or time-consuming to write by regular journalists. For example, writing election articles for all of the approximately 300 municipalities of Finland is not feasible. Also writing articles from all the major European football leagues is quite a lot of work: A single season for all series' combined sums up to over a thousand articles which is not quite feasible for human writers, but is manageable for an automated journalist.

Voitto also generates images from the data in most articles. For example, Voitto creates bar charts from the match's stats and a visual list of events that happened in the match. These images are used in Voitto's articles but human journalists are also encouraged to add these graphics to their own articles. This thesis will only focus on the text generation side of Voitto's articles. Appendix B.1 is an article written by Voitto, which also contains the images.

The articles written by Voitto are also somewhat targeted to gain better topicality. Because Voitto writes many articles their visibility has been limited on the main Yle sports news site. This means that they don't show up on the front page or the list of the latest news, otherwise the articles would flood the lists. However, the articles can be read from Yle's mobile application "Uutisvahti". Uutisvahti allows the user to subscribe to index terms and the app will send out a notification when an article is published with that index term. Therefore users can affect which articles by Voitto they will read. Index terms can be team and player names, and the user will get a notification when that team or player has played a match.

Voitto has also written articles in Swedish, with the name Victoria.

4.3 Implementation

The implementation of texts in Voitto uses the previously featured decision trees and templates but with some twists to accommodate Finnish language and to make the texts more lively. Voitto is implemented in the Scala programming language but its resource files use the Extensible Data Notation (EDN) data format.

The articles written by Voitto are divided into multiple smaller sections based on context. This division varies by article subject. In ice hockey reports, the texts are divided per period. The football reports are based on different highlights of the match. For example, a player scoring multiple goals or the match having many yellow and red cards. Each of these sections has its own decision tree and templates. Not all of these texts are required to select a template if no template in the decision tree fits. For example, if no player scored multiple goals, a decision tree that selects a template about star players will not select a template. However, some texts, such as the title, are always required.

Before text selection, the data received by Voitto is derived into easier to use values. For example, the number of goals scored by both teams (e.g. 2 and 5) is derived into goal difference (e.g. 3) and the outcome of the match (e.g. “2-5”). For text generation purposes these values can be divided into two categories: boolean values (true/false) and attributes. Boolean values are used in decision tree decisions and attributes are used to fill templates.

4.4 Decision Trees

The decision trees of Voitto is relatively basic. They are a collection of simple if-else if-else structures written in the Scala programming language. The values in the nodes of the decision trees are the aforementioned boolean values derived from the data. These structures are used to select the right template. A simplified example of a decision tree can be seen in Appendix A.2. The `loadTemplate` function loads the selected template from a resource file.

4.5 Attributes

The attributes in Voitto's templates use a similar custom Mustache template language that has been used in examples in this thesis, i.e. attributes are surrounded with double brackets.

There are many different kinds of attributes in Voitto's templates. There are plain text strings, series names, numbers, "helpers", player and team names and lists of names. "Helpers" are specific words that can be used to spice up the text according to data. For example, instead of repeating team names, a helper can be used to say "the home team" or "the away team" instead, depending on the data. Take the following template for example:

```
{{{homeOrAwayTeam}} scored the first goal.}
```

The attribute `{{homeOrAwayTeam}}` will either have the value "Home team" or "Away team", depending on which team actually scored the first goal. While these helpers could be replaced with having different templates for either case, it would require having many very similar templates and more complex text selection logic.

The template attributes of Voitto have info on which inflection to use. Adding inflections to attributes is absolutely necessary for Finnish, otherwise, the text is syntactically broken or it will sound very robotic. All the available inflections for names and helpers are listed in separate resource files. The resource files are long lists of player and team names that have a specific set of available inflections. The resource files are used because no reliable way was found to do inflections on the fly. First 10 numbers also have written inflections but for larger numbers, the numbers are used as-is with inflections in the end, e.g. "20:ssä ottelussa" ("in 20 matches"). All attributes except plain texts have inflections. Attributes are marked in template files with a colon after attribute name e.g. `{{winner:genitive}}` would be the name of the winner in genitive form.

Attributes can also have other options. For example, player name attributes can have options to only use the last name without the first name. These options are mostly shortcuts that reduce the amount of code that needs to be written. For example, the programmer does not have to create a separate attribute for each player's full name and last name. The options can also be, for example, player or team stats or the name of the opposing team. Attributes with options can also have inflections. Options are noted with a dot. For example `{{player.totalGoals}}` would be a player's total number of goals and `{{player.totalGoals:genitive}}` would be the same but in genitive form.

Attributes also can be forced to be capitalized so that they can be used as the first word in a sentence. Capitalization is also noted with a colon, but it is at the end of the attribute string. It can also be used with options and inflections e.g.
`{{player.totalGoals:genitive:capitalize}}`
would be the number of goals scored by a player, capitalized and in genitive form.

4.6 Variations

Each of Voitto's templates contains a list of variations that say the same thing but using different words and turns of phrases. Each language has its own list of variations. The point of variations is to make the articles not sound samey even if the same templates are selected. The variations might use different attributes to make them less similar. If there were no variations, reading multiple articles from the same subject would make the articles sound more robotic and predictable.

Variations are selected with weighted randomness. This means that all variations have attached weights that can be changed to determine how likely it is for that specific variation to be selected. This is used to lower the chance of a certain text from appearing if it starts to seem repetitive. For example, if a certain turn of phrase is used in multiple templates and starts to become grating, the weights can be lowered on all variations that use this phrase.

Variations try to achieve the same goal as the previously discussed synonyms in templates. As the variations are always picked at random, they cannot have different tones, as they would not be supported by data. Instead, different tones are achieved by having different templates. For example, a crushing win will use a different template from a close win.

4.7 Template Files

While some automated journalists have quite universal/simple templates [2], the templates of Voitto are quite specific/complex. While this hurts the reusability of the templates, it does make the text more lively as it opens up the possibility of using exact terms and turns of phrase. A simplified example of a template file used by Voitto can be seen in Appendix C.1.

While Voitto's templates are quite basic there are some special features. Voitto's template files have a loose hierarchy. There are some common template files that fit many sports. These are at least somewhat independent from the sport and the decision tree that uses them. An example of a common template is the "crushing-win" titles template; the winner won with a sizeable margin. The same text is applied to floorball, ice hockey and football titles. However, the decision tree logic is different for all these sports. While in football 3 goal difference is enough to select this template, in ice hockey 4 goals are needed and in floorball, a more sophisticated calculation is used. There are also quite a few templates that are used in both ice hockey and floorball as the sports are quite similar.

Common templates have some advantages which are similar to not having copy-paste code. If there are mistakes or typos in the text, the fix doesn't need to be implemented into multiple places. It also provides transferability to similar domains, as some texts don't need to be written again. Adding variations to these common templates helps all sports that use these templates. All in all, common templates give some of the advantages of having more general templates without the disadvantages, such as reduced fluency.

Decision trees can use templates from multiple template files. For example, football titles use a template file for common titles and another file for football-specific titles. There is no checking that the different template files don't have same names for templates and there are no overriding rules. This is an issue that we will return to in the practical part of this thesis.

4.8 Text Rendering

After the template is selected with a decision tree, all of its variations are rendered. This means that the attributes that the template contains are fetched from the values, inflected and then placed in the templates.

Sometimes there are variations that cannot be rendered. For example, there is a player name that does not have inflections in the resource files or an attribute is unavailable (most often caused by a bug). If a variation cannot be rendered then that variation is not used and another variation is selected. If no variation fits then that text will be left out. If the text is mandatory, such as the title of the article, then the article is not generated. This mostly happens when there are bugs or oversights in the text generation code. This can also happen if there is a typo in attribute or inflection name, however, those are usually caught by automatic tests.

As it is possible that there are player and team names without inflections, most templates have at least one variation were attributes don't have inflections. As these variations tend to sound quite unnatural and inarticulate, they often have minimal weights so that they are only selected when all other variations fail. In Appendix C.1 there is an example of a text template file used by Voitto. The final variation of the `:team-won` template:

```
["{{loser}} kaatui, kun {{winner}} otti voiton lukemin
 ↪ {{goalsPerPeriod}}." {:weight 0.1}]
```

is the variation without inflections and with minimal weight.

After all of the variations are rendered, a random variation is picked from the variations that rendered successfully. After each text's variation is picked, the texts are concatenated to form the article. The finished article is pushed into a publishing pipeline.

4.9 The Six Requirements

In this chapter, we will compare Voitto against the six requirements of automated journalists. Due to the fact that these requirements are qualitative, this analysis will not be exact.

Data availability has been very uneven as Voitto uses multiple data providers. What data is available completely depends on sport and series and some articles are left out altogether due to the poor quality of the data.

Topicality has been pretty good. The articles arrive at the latest about 5 minutes after the sports match is over. Also, the previously mentioned filtering of articles via Uutisvahti improves the topicality via personification.

The accuracy of articles is pretty good, there are not many bugs that would cause the text to not reflect the actual match data. As we previously stated, we would also include missing information as a part of accuracy. Voitto has some issues with this. This is partially due to the poor quality of some of the data providers but also due to the implementation of texts. As Voitto uses templates and decision trees, all texts need to be manually added. Because of this, and limited resources, there are limits on what Voitto has been taught to mention. Therefore some important facts will not be mentioned if there has not been time or foresight to add a fitting decision tree branch and template.

Voitto's fluency is mediocre. Sometimes bugs cause issues with fluency but there are other issues with fluency as well. Voitto does not have the "Recurring expression generator" used in Vaalibotti Valtteri which was discussed in Chapter 3.2.1. Therefore if a name is mentioned multiple times, the full name is used every time. This makes the text a bit clunky and unnatural. As the templates are independent, the same things are sometimes accidentally said in multiple seemingly unconnected templates. It is difficult for text designers to spot these problems as the texts are separate and it is hard to see the big picture.

Transferability of Voitto is quite bad. The templates of Voitto being very specific makes the transferability worse. Even changes to similar use cases, such as from one team sport to another, can take a lot of work. Voitto's "common templates" functionality diminishes this problem slightly and some parts of the code refining data to easy to use values can be and are reused.

Changing to completely different domains requires a lot of work. For example, in the case of election news, only some parts of Voitto could be reused. The decision trees, templates and the code refining values from data needed a complete rework. What can be reused is the base architecture (API), the template rendering code, the template file reading code etc.

Modifiability is relative good for Voitto. The template files and decision tree are easy to edit for programmers. For non-programmers, the syntax of the template files can be a bit too complex. The decision trees will be even harder to edit due to the fact that they are code written in the Scala programming language.

The transparency of Voitto is quite good. Its source code is available as open-source.

Though similarly to modifiability, the open-source code only really helps people who understand Scala. Non-programmers might only understand the documentation that does not really go into too much detail. Also, the decision trees are just embedded in the middle of the codebase which makes it harder to find them.

While modifiability and transparency are quite good due to Voitto's implementation method of templates and decision trees, they cannot be fully leveraged as they are only available for programmers. This acts as a bottleneck for improved accuracy via more decisions in the decision tree and fluency via more variation.

In the following chapter, we will try to tackle the problems with modifiability and transparency by creating a tool for non-programmers to understand and edit the text generation logic of Voitto. Even though the biggest problem with Voitto is transferability it also generally quite difficult to solve[2] and outside of the scope of this thesis. As stated earlier, the increased modifiability might help with the poor transferability.

5 Software to Edit the Texts of Voitto

5.1 Introduction

The objective of this practical work is to create a software/tool that helps human journalists edit the texts of robot journalist Voitto. This includes adding new text templates, changing existing ones and editing the text selection logic (the decision trees). The goal is to implement the tool as a web application that is easy to use even with minimal technical knowledge.

The tool will change the process of adding new texts monumentally and removes programmers as a middle man in the text adding process. It should provide content creators autonomy while limiting their options as little as possible. The tool should be safe to use so that the content creators should not be able to break the text generation system.

In this chapter, we will first introduce the problem and set the objectives for the practical work. Then we will go through the solution's creation process and present the outcome. In the following chapter, we will analyse the solution, how it answers the problem, what is not working and how it could be improved.

5.2 The Problem

We did an interview with a journalist and a designer who had designed nearly all the texts of Voitto. The interview was conducted as a group discussion with some premade questions to get the conversation started. The questions were mostly about the process of adding new texts. The goal was to find out what exactly is the problem with the current process and what the optimal process would be like.

Currently, the process of adding texts begins with a content creator coming up with an idea for a text that could be added to Voitto's repertoire. The content creator then writes a template for that text for all languages. Next, they inform the programmers that this text should be added and gives rules when the text is used (the decisions to be made in the decision tree). The programmer then adds the text to the right template resource file and implements the necessary variables needed by the decision tree and the attributes. Then they add a new decision/branch to the decision tree, which has the new text as the final leaf node.

The interview revealed that content creators are completely unable to edit Voitto's text selection logic. It is impossible to add even simple rules to the logic if you don't understand the programming language used. Journalists and designers are completely reliant on programmers when it comes to altering the decision trees of Voitto.

Editing the text templates also has its own problems. While it is at least possible to edit the texts as they are in somewhat human-readable resource files, it is difficult to understand the syntax and editing it properly requires an editor with syntax highlighting. Appendix shows C.1 a shortened example of a template file. The syntax is quite complex as it is a map data structure (template names as keys and templates as values), which contains a map (language codes as keys and translations as values), which contains a list (different variations) which contains vectors (for a variation and it's settings) and those vectors contain a string (the text itself) and sometimes a map (a map of settings, most often the weight). Additionally, the variations' texts have attributes, which makes the syntax even more difficult to follow. As the file is written in EDN, lists, vectors and maps use different brackets, which makes it difficult to remember which one is used where. Also remembering to close all the brackets had caused issues for content creators, if they were not using an editor that will notify about syntax errors.

Adding inflections to Finnish language texts was found problematic as it is hard to remember all the different forms and their exact names. For example "accusative" and "genitive" forms are very similar or exactly the same on many occasions, except with team names that are plural, such as “Kärpät”. Also "genitive" was often mistyped as "genetive". Another issue content creators faced was that not all attribute types (e.g. players, teams, numbers) had all possible inflections. Therefore the content creators had to remember which inflections were available and which were not for each different attribute type. This would cause unnecessary back-and-forth in text design, as the content creators would design a template, then the programmers would say that an inflection is not available, forcing the content creators to redesign the texts.

Another unrelated problem that came up, is the difficulty of coming up with different ways to say the same things in variations. Content creators often found themselves just changing some words to synonyms or changing the word order (this is very much viable in the Finnish language without the text sounding strange). Though, the content creators admitted that this problem is mostly due to limited imagination and not necessarily a technical issue.

Journalist/content producers also had issues with understanding the big picture in text generation. Unlike the programmers, the content creators had no way to see the actual decision trees of Voitto. They had a hard time remembering what text had and had not been implemented. It was hard to realise what an actual complete article would look like, as the content creators had to just design texts without seeing the result until the entire text generation was finished. It was hard to pinpoint whether same or similar texts were used in many parts of the full article.

Understanding what data is available for each sport also had been problematic. The data was fetched from different HTTP APIs and the journalists did not know how to read the data, so they had no way of knowing what is and what is not possible with the available data.

5.3 Hopes

When asked what the optimal process for adding texts would be like, the answer was that the programmers would create a base for the domain in question. The base would have some basic attributes and boolean values to be used in the template attributes and decision tree rules. Then, using a tool, the content producers could easily create the text generation logic and templates. The tool should be graphical and have a drag-and-drop UI for picking boolean values that would form the decision tree. Texts would then be added to that decision tree with the same tool.

Additional features of this tool would be automatically picking the correct inflection by analysing the text. The tool would also proofread the texts in the templates. A WYSIWYG (what you see is what you get) editor for image generation was also mentioned as a possible feature of this ideal tool. The tool could use some automatic natural language generation system to create new variations for templates, based on existing ones.

The content producers had ideas for some other helpers that could be integrated into the tool. Automatic translation for new templates would help a lot if there are many languages and templates to translate. Also related was the ability to easily add new languages to existing templates. Another idea was having a reminder to add a variation without inflections. As was previously discussed in this thesis, this is necessary because there is no guarantee that all team and player names have inflections.

5.4 Other Implementation Methods

Most of these problems are due to the implementation method of Voitto. The previously discussed different implementation methods could perhaps be used to solve some of these problems. For example, machine learning implementations would no longer require manually written rules or templates. Likewise, the fact-ranking based solution would not require writing rules.

Unfortunately, the machine learning solutions cannot be used as there is not enough teaching data available for all the languages and subjects that Voitto covers. While the fact ranking method could be used for some aspects of Voitto, it cannot be used to create a story-like structure that is wanted in some use cases [2].

Also, there is a desire for the journalists to be in complete control of the texts. This is only really possible when using templates and decision trees as they are the only ones with perfect modifiability. Therefore the existing implementation method has to be kept.

5.5 Existing Solutions

There are some existing tools that try to make the adoption of automated journalism easier. In this section we will discuss the tools of which we are aware of: Wordsmith by Automated Insights, Quill by Narrative Science, Yseop Compose by Yseop, Arria Studio by Arria and Data2Text Studio.

5.5.1 Wordsmith and Arria Studio

Wordsmith by Automated Insights [1] and Arria Studio by Arria [2] seem like they are closest to helping with the problem. Both tout complete control of article writing and ease of use for non-technical people. However the information available on them is limited. There is no info on language support which would lead to believe that it is limited. They are most likely closed-source and commercial, which is a limitation that will be discussed later on.

[1] https://automatedinsights.com/wordsmith/

[2] https://www.arria.com/studio/studio-overview/

5.5.2 Yseop Compose and Quill

Yseop Compose by Yseop[3] and Quill by Narrative Science[4] both appear to be designed with business intelligence and finances in mind and they both seem to be complete solutions that do not support adding custom rules or templates. They appear to be designed to ease the processes of internal reporting instead of writing articles for consumers. Therefore they are too focused to solve the problem at hand.

5.5.3 Data2Text Studio

Data2Text Studio by Duo et. al. [9] is a complete solution for automated journalism that creates templates and rules via machine learning. The models created with machine learning can then be edited. Unfortunately, there is not enough teaching data to cover all the aspects of Voitto and therefore this solution does not help that much. While the templates can be edited manually the greatest strength of the platform is removed so there is little reason to use the platform.

5.5.4 Conclusions

While there are tools that make automated journalism easier to implement for human journalists, they do not solve the problem at hand. Yseop and Quill are too focused, and cannot be used in the domains were Voitto works in. Wordsmith and Arria Studio are the closest to fixing the problem, but they would require a complete redesign of Voitto, as they most likely cannot be directly integrated into the existing system.

All of the systems focus on English, which makes the tools unviable in the case of Voitto. As was discussed before, inflections make Finnish a harder language to automate, as the tool has to have the ability to inflect the words. Therefore no tool truly answers the problems at hand.

[3]`https://compose.yseop.com/`

[4]`https://narrativescience.com/products/quill/`

In addition, all the tools appear to be closed-source, which means that the project would depend on the tool and its creator, which is also undesirable. This also means that the tools are harder to customize and extend. None of the tools helps with the problem enough that this trade-off should be made. Therefore a new tool must be created to answer the problem. Aside from Data2Text studio which has no info, all the tools appear to be pay-per-use. In the long run, this will probably cost more than creating a custom tool as most of the text generation infrastructure already exists in Voitto. Only resource file editing is needed.

As no tool fits the requirements, a custom tool has to be made. In the following sections, we will set the exact objectives and scope for this new tool and go through its creation process.

5.6 Objectives

The objective of this work is to create a software that makes it possible for journalists to edit the decision tree of Voitto and makes it easier to edit the templates. The software must be simple enough that the content creators only need to understand the basic concepts of text selection with decision trees and text generation with templates and attributes. The tool should require minimal knowledge of automated journalism and no knowledge of programming. The tool should be intuitive and easy enough to learn without much help or training.

After the tool is implemented the process of adding a text would be as follows: A content creator comes up with an idea for a text that should be added to Voitto. They write the needed template with the tool. The software will immediately show them that they have selected correct inflections and they have no way of making syntax errors. After the text is done, they add a new branch to the decision tree which uses the newly created template. Then they deploy the text to a test environment and they can see how the text looks like in practice. Later on, the text is deployed into the production environment. There is no need for a software developer if no new variables or attributes are needed.

The journalist should be able to work autonomously throughout most of the content creation process. The only work that is left for programmers is deriving attributes and values from the data and handling the placement of texts within the article (i.e. which text is the title and which is the lead etc.). We will return to these shortcomings when discussing problems with the solution.

The tool should never be a hindrance to text generation, even if the person writing the texts is very experienced in editing the resource files and the decision code manually. The tool should be more efficient in text generation as the writer does not have to think about the syntax at all and they don't have to double-check the inflections. The decision tree editing should also be about as easy as adding new if-clauses into any part of the decision tree.

The software should not be able to break the entire text generation. It should not be possible to add attributes or boolean variables that do not exist in the code yet. There should be enough safeguards that the texts generated with the tool will always output something.

Of the problems listed in the previous chapter, the tool should fix all issues with the complex syntax when writing templates. The tool should help with understanding the big picture. Selecting the right inflection should be much easier and mistyping inflection names should be impossible.

The tool should take some hints from good programming practices as the editing of text generation logic is quite similar to programming. In programming, code reviews are often used before new code is added to the main codebase. It means that another programmer checks the code before it is added to the main branch. A similar system could be used in the tool: another content creator should check the changes made to the text generation logic before they are deployed.

It should be possible to document the text generation files in order to inform other users of the purpose of a text. The tool should have some kind of version control system so that if the logic is faulty, the previous, working version can be brought back. It should also have automatic checks for problems with the logic or templates. In the context of the tool, it means that before a new text is deployed, some basic checks are done. If errors are found, the new text cannot be added. All of these features will not be implemented in the limited scope of this thesis, but they should be kept in mind when thinking about further improvements.

5.7 Limiting the Scope

Of the features requested by the content creators, a few were left out to have a manageable scope. However, all features required to make the content creators be able to work autonomously are kept. We will return to some of these features later when discussing possible improvements that could be done to the tool.

We decided to leave out automatic spell checking and automatic intelligent suggestion of variations. Automatic spell-checking would probably not be too difficult using some spell checker library. It was decided to be left out because it is not really necessary at this stage. Automatic suggestion of variations would most likely be extremely difficult to implement and would probably require a complex system and machine learning to get something useful.

The problem of not knowing what data is available is quite difficult to solve and not really a part of the problem of editing templates and text selection logic. The data source of Voitto is actually a sort of aggregate of multiple sources. The HTTP API that provides data to Voitto fetches its data from multiple other sport data providers and reforms it so that all sports have similar looking data in the same format. Because of the fact that the data is provided by multiple services makes it hard to know exactly what data is available.

For example, Finnish ice hockey leagues and NHL have very different data even though they are the same sport because they are provided by different sources.

There are even differences within the same provider: smaller series' often have much fewer data. It is very difficult to know what data is available for each series without checking the data itself, which requires reading somewhat complex and large JSON files. It would be possible to create a tool that checks all keys and values of the JSON objects and then tells what is and what is not available, but that is well outside the scope of this project, even though it will help the journalist create articles.

We decided to leave the possibility to create custom attributes and boolean values outside of the scope. We were unable to come up with a way for non-programmers to be able to derive less abstract values from the data. While it would probably be possible to create such a tool it would require a lot of work. For now, the programmers' job is to create the values and attributes, and content creators can use these to create the actual content. A further improvement for this tool could be a way to derive multiple attributes from a single piece of data. For example, a goal scored in the data could be automatically derived to the following values: goal scoring team and player, time of the goal, the standing after the goal etc. This would give journalists much more to work with, with little effort from the programmers.

We left out the ability to actually deploy texts to Voitto from the tool. It is not really a part of the problem this thesis is trying to solve. However, we did design it thoroughly and did proof-of-concept work to make sure that the files generated by the tool are usable as-is by Voitto. Therefore adding the deploy process would be possible, it would just require quite a bit of infrastructure work, which is outside the scope.

Similarly, we decided to leave out sending boolean variables and attributes from Voitto to the tool. Again we did proof-of-concept work to make sure this is possible to implement. Both of these features should be implemented before the tool is completely usable in practice.

5.8 The Six Requirements

The six requirements of robot journalism will be used as a reference on how much the tool actually helps, in addition to some interviews. Of the previously introduced 6 requirements of automated journalism, this tool will help mostly with transparency and modifiability. It will also indirectly help with fluency and possibly accuracy.

Before this tool, programming knowledge was required to fully understand the logic of how templates are picked. This meant that for a large number of people the text selection logic was not as transparent as possible. With the tool's visual decision tree, it is much easier to understand the selection process. The transparency is improved for the template files as well because the tool makes it possible to read them without having to understand the complex syntax.

The tool should make it easier to edit the templates and text generation logic which would help most with modifiability but also with fluency and accuracy. As the tool will take away the requirement of having to understand the template file syntax it should speed up the template-modifying process and make it more accessible to less tech-savvy individuals. The fluency and accuracy of Voitto's texts should increase with having more templates and more variations in the templates. Therefore the easier addition of texts should also improve fluency and accuracy indirectly. As the tool will detect issues and warn the content creators of them, it should also further improve the accuracy and reduce error rate (i.e. reduce bugs that will cause the article generation to fail).

5.9 The Solution

The solution we created is a web application that has a separate decision tree editor and template editor i.e. one view for editing what is said and another for editing how things are said. These editors are used to create, read, update and delete the resource files that Voitto uses. In addition to these two editors, there is a settings page for adding more languages to the project. All changes to resource files are automatically saved to the backend.

The text generation logic of Voitto is represented in the tool as having a list of "automated texts". Each "automated text" represents a single decision tree in Voitto's text generation. For example, the title of an article would be its own text. Each text has a decision tree file and a list of template files amongst other settings.

Each text has a setting that determines if this text is allowed to be empty. For example, each article has to have a title, so the title text cannot be empty, but a text about a key player can be empty, as not every match has a key player. The "allow empty" setting is used in error checking, which will be discussed later on.

Each article type is defined as a single project. Each project has its own texts, templates, decision trees and settings (e.g. languages used in this project). For example, football articles and ice hockey articles are their own projects. Currently, the tool only handles one project but support for many different projects should be added later on.

5.9.1 The Front Page

The front page shows a list of all resource files in the project and links to them. It also has a list of errors and warnings. The situations when errors are warnings are displayed is discussed later on in this chapter. Each warning and error has a link to the template or decision tree that caused the warning. The front page can be seen in Figure D.1.

5.9.2 Template Editor

The template editor has a list of all available templates which can be edited by the content creator. It is also possible to add new templates. Each template has a list of all languages the project uses and each language has a list of variations created for that language. A picture of the template editor can be seen in Figure D.2.

Each variation has its own component where its text and attributes can be edited. Editing attributes is much easier with the tool, as the tool only shows the available inflections, so the content creator cannot select an inflection that does not exist and they cannot be mistyped.

The tool also shows an example text for each variation which also makes selecting inflections easier as the content creator does not have to remember what each inflection means. There will be more information about example texts later on. New variations can be added easily and existing ones can be deleted. The weights of variations can also be edited.

When adding new attributes, the tool will suggest existing attributes that are implemented in Voitto's code. However, the content creator can also add their own attributes in order to create texts before a programmer implements the attributes needed by the text. The goal of this is that the content creators should not have to worry about what attributes are implemented when creating the texts. Otherwise, they would need to first design the texts with some other tool, ask the developers to implement the attributes, and then use the tool to add the texts with the newly implemented attributes. If the texts have attributes that are not yet implemented, the resource files cannot be deployed to test or production environment as they would probably break the text generation.

5.9.3 Decision Tree Editor

The decision tree editor shows a graphic of the decision tree that has all the branching nodes with boolean variable names and leaves with template names. It is very similar to a flowchart, where all the branching points are boolean variables. If the variables names are easily understandable, it should be possible to read it as easily as a flowchart. The decision tree editor can be seen in Figure D.3.

The content creator can change these boolean variables and template names. Currently, they are changed by first selecting the node by clicking on it, and then selecting what content it should have from a list of variables/templates. Adding drag and drop functionality should be considered when thinking about further improvements to the tool.

When selecting a branching point, the tool shows a list of all available boolean variables. When selecting a leaf node, the tool shows a list of available templates, according to what templates are available for this text. Adding new templates with the template editor adds the templates to the list. All leaf nodes have links to the templates so that the content creator can quickly see what the templates contain.

The content creator can also add new branching nodes. Between every two nodes, there is a button with a plus sign. Clicking the plus sign adds a new branching node there. New branches can also be added to the top of the three. By default, the branching node's boolean variable is marked as being empty. If there are empty branching nodes, the resource files cannot be deployed to test or production environments. Deleting branching points is also possible and the tool automatically reconnects the separated parts.

5.9.4 Automated Texts Page

The automated texts page has info on what automated texts (the building blocks that have their own decision trees; e.g. title, lead, etc.) this project contains. Texts can be edited, added and deleted. Each text has a list of template files used by that text, and the decision tree the text uses. New template files can be added in the automated texts page. The page can be seen in Figure D.4 The automated texts page also has download links for all resource files and the possibility to upload existing template files.

5.9.5 Settings Page

The settings page only has the option to add languages. It used to have all the features of the automated texts page, but they were moved as they deserve their own page due to them being a key feature. Currently, the software does not support having multiple projects. After this feature is implemented, project-related settings, such as the article type of the project could be added to this page. The settings page in its current form can be seen in Figure D.6.

5.9.6 Example Text Page

The example text page was quickly put together and of these pages, it is probably furthest from complete. The page can be seen in Figure D.5. On the left, the page shows a list of all automated texts in the project. These texts can be selected and deselected one by one. When the user clicks "Create example text" the software picks one template randomly from each of the texts and creates an example text from that template. The example texts are concatenated and shown in the text area on the right side of the page. A text area was used in order to allow the reader to manually add paragraph changes so that they can read the example text better.

The order of the texts can be changed by de- and re-selecting texts in the correct order from the texts list. This is quite bad UX design and should be replaced with a drag-and-drop interface.

The order of selected texts is stored locally so different users can each have their own example text settings stored. This would help if different users want to test different paragraphs on a longer article. The settings storing was implemented using JavaScript programming language's `localStorage`-functionality.

5.10 Additional Features

The tool has some features that are not directly related to editing the text generation logic. It would be possible to edit the templates and logic even without these features, but they make it more user-friendly and less error-prone.

When editing the templates the writer is constantly shown an example text i.e. what the template might look like after rendering. This can be seen in the template editor, Figure D.2. The example text is created by having a single hardcoded example text for each combination of the attribute type, option and inflection. The example text makes it much easier to spot possible spelling errors, wrong inflections and wrong types of attributes.

While adding attributes the tool will also show what the attribute might will look like when rendered. This fixes the issue of having to remember the names of declensions: The user can test them and select the one that fits the sentence.

The software contains an error checking feature, that detects issues in the template files and decision trees. There are two levels of issues: errors and warnings.

Errors are displayed in the following scenarios:

- A text is not allowed to be empty, but the decision tree has leaves without text
- A template has player and team names, but there is no variation without inflections in a text that is not allowed to be empty
- A single text has same template names in different template files (i.e. two template files have clashing template names)
- A decision tree has a branching node that does not have a boolean value.
- The templates have attributes that are not implemented in the code
- A decision tree has branching nodes that have unimplemented values

Warnings are displayed when:

- There are missing translations
- A template has player and team names, but there is no variation without inflections, but the text is allowed to be empty
- There are variations that have no text in them, even though other variations with text exist

If there are errors in the templates they cannot be deployed to either environment. This is somewhat similar to having automated tests in software development: if the tests fail, the new version cannot be deployed as it is broken. Warnings are just to notify of possible issues; they are things that cannot break the entire text generation, they just might make the text lower quality.

These lists of warnings and errors detected by the tool could be expanded if needed. After the tool has had more use and more feedback is gathered, some stumbling blocks for content creators will probably be found and more checks for errors and warnings should be added.

The text editor allows writing a short explanation/documentation for every automated text. This should help in the same way as documenting code. The goal of this helper text is to explain to other content creators what the purpose of each piece of text is. For example, in the lead text of Voitto's football articles, the first sentence always tells the outcome of the match. To avoid repetition, the following sentences should never mention the result again. The explanation text/documentation of the first sentence of the lead text could inform the user that they should include the result in all templates and variations.

The documentation could also help content creators if they return to edit an article after a break. For example, it could help when revisiting old election reports when the next election is coming up or returning to sports reports after the off-season.

5.11 The Creation Process

We started the creation of the tool by having the first interview where we asked about the problems with the current situation and the desired optimal text generation process. From the information gathered in the first interview, we designed and implemented the first version of the tool. This version had all the features that are needed to edit the texts and logic but editing them would be very arduous. For example, there was no way to save the changes made which means that the software was reset every time the web application was reloaded. The only way to use it was downloading all files after editing them and then reuploading them the next time when using the tool.

However, it was enough to gather early feedback for the UI and UX (user interface and user experience). This was the only real goal of getting early feedback. Without the backend, it would be impossible to get feedback on the actual usefulness of the tool. The feedback we got was mostly positive and more related to minor details than the big picture.

The most important feedback we got of the UI was that the front page was aversive because it had too much information. The front page had to be made more simple and welcoming so that the user is not immediately scared away as that page was more complex than the actual text editor. We decided to remove the left-hand column of the front page and move the info from it to the "automated texts" and "settings" pages which will be discussed later on. The errors and warnings column was kept on the front page, as it works as a TODO-list for users and it is important to see possible problems quickly. The old front page can be seen in Appendix D.7.

After this iteration, we started working on creating the backend and adding automatic data saving when users make changes. After these steps, the tool should be much more usable and would have actual content that can be modified. The only things missing will be Voitto fetching the templates directly from the tool, deploying the resource files to test and production environments, and sending variables to the tool. Only the deploy processes will be missing, which was deemed outside of the scope for this thesis. However, a small proof-of-concept was done on all of these features to make sure that these are not technical hurdles that make the tool unusable. When the backend was done it was already possible to analyse the usefulness of the tool, without having wasted time to work on the complex deploy processes if the tool is found useless.

First, a UX test session was done, which will be discussed in the following analysis chapter. After the session, many usability and clarity improvements were made. The biggest changes included moving automated texts -section from settings to their own page as settings did not seem like the right place for them. Another major improvement was the possibility to create boolean values in the decision tree editor in addition to using existing ones. After these changes, the tool was ready for final analysis.

After the UX test was done, it was agreed upon, that the tool was usable enough to be used for designing the texts for Voitto for automated reports of the Finnish parliamentary election of 2019. At this point, we had to change the tool's attribute types to fit elections. Instead of players and teams, the tool would now have municipalities, candidates and parties. During the text creation process, it was discovered that an example article view is needed to see how well the different templates fit together. Therefore we implemented such a feature.

After the election articles were implemented it was time for final analysis which was conducted with the journalist who created the texts using the tool. The analysis will be discussed in the following chapter.

5.12 Changes to Voitto

Some changes must be done to Voitto's code to make it possible to use the tool for text addition and editing. We tried to minimize the number of changes required, but some things are impossible or at least very impractical to do without changing the code of Voitto. Implementing these features was left outside of the scope of this thesis, however, they had to be designed as the implementation of the tool depended on them.

5.12.1 Moving Logic to Resource Files

Text selection logic must be removed from the code and moved to resource files as it is very difficult, if not impossible to reliably edit the code itself via the tool. This change required nearly a complete rewrite of the text selection logic. Previously the decision trees were Scala code with simple if – else if – else logic. Extensible Data Notation (EDN, .edn file format) was selected for the new resource file format as that was already used in templates, so we would not need multiple file format parsers in both Voitto and the resource file editing tool. An example of the new file format can be seen in Appendix A.3 and can be compared to the original Scala code in Appendix A.2.

We decided to go with a very simple data structure for the logic: the logic is formed of nested vectors that all contain three values. The first value is always an EDN keyword that is the name of a boolean variable in Scala. The second and third value is either a nested decision tree (i.e. another three-item vector with a similar structure), the name of a template, or the keyword "None". This type of structure represents a binary decision tree, where each first item in a vector represents a boolean variable and all template names and "None"s are leaves. "None" is used to signify that no text should be used.

Using the resource file to pick templates uses a recursive algorithm to reach a leaf node. Template selection begins with the outermost vector and checks the boolean value that corresponds to the keyword in the first place of the vector. If the boolean value is "true" the logic selects the second item in the vector. If it is "false", the third item is selected. If the selected item is a nested decision tree (i.e. another branching node, not a leaf), its value is checked in exactly the same way. This recursion ends with either a template name or "None". While the format is simple and easy to parse, it is unfortunately not very human-readable, even with proper formatting.

5.12.2 Resource File Deployment

It should be possible to change the texts and decision tree in test and production environments without needing to re-deploy the Voitto software. Currently, new texts and logic are deployed with the code. This would mean that even if the tool is used, it would be the programmers' job to actually make Voitto use the new resource files in the production environment. This also ties in with the previous change: Even if it would be possible to edit the Scala code directly with the tool, it would be even more difficult to send the new source code file to a deployed application and edit the code on the fly. So deployment would still need to be done by a programmer, which would be against the stated objective of the content creator being able to work autonomously.

It would be ideal to add a new file storage service such as Amazon Web Services S3 for Voitto's templates. The tool would, when asked to, push its template and logic files to the file storage, and Voitto would fetch files from the same storage. Alternatively, files could be sent directly to Voitto. The plus side of using a storage service for the files is that if Voitto is restarted, or redeployed, the files can be fetched instead of having to push them again from the tool.

While the tool should have the ability to deploy resource files to Voitto, it raises the question of what kind of process the deployment should have. It is possible to make mistakes in the logic and template files that are not immediately obvious and cannot be automatically tested. For example, using wrong but existing attributes. We decided that it should be possible to deploy files to the test environment and see what kind of texts the test version produces with the new resource files. As was previously stated, deployment should not be possible, if the tool detects errors in the texts. Deployment to production environment should not be possible to do without someone checking the templates first. This way the changes are double-checked and therefore less likely to have issues. However, these processes are outside the scope of this thesis.

5.12.3 Combining the Attribute Lists

Before the tool, all of Voitto's texts had their own list of attributes that each text used. After the tool is implemented, all texts need to be able to use all attributes as the template editor does not limit the available attributes per text. Therefore all attributes must be made available everywhere, and instead of using different lists for different texts, a bigger, common list must be used.

5.12.4 Moving Values to the Tool

The tool needs to know some information about Voitto's code, such as all the available boolean variables that can be used in the decision tree and all the available attributes and their types. This is necessary as it allows the tool to check that the texts don't contain non-existing attributes. Therefore a method needs to be added to export lists of boolean values and attributes from Voitto's code to the tool. While this feature was not yet implemented, we did a proof-of-concept where the backend of the tool stored lists of all attributes and boolean values. When new attributes and values are implemented Voitto should automatically update these lists.

5.13 Technical Aspects

The tool is a somewhat complex single-page web application with a separate frontend and backend. The frontend is used to edit the templates and the decision trees while the backend handles saving the changes to a database.

The tool was implemented as a single page web application. There is a lower threshold for someone to test using the tool if it is a web application as the users don't have to download anything. Updating the application is easier for users as they don't have to do anything to update which is vital when you want the users to test small changes. Having automatic updating or a way to notify users of available updates would require lots of unnecessary work. We decided to make it a single page app mostly because we found it easier and we had experience in these types of applications.

Having a downloadable offline software also would have had its own advantages. It is much easier to handle files in a local application and we think that the application would have been easier to implement. However, we felt like the easier testing and updating of the web application made it a clear choice for this practical work.

The tool is implemented in Scala.js programming language (Scala converted to JavaScript programming language to be used in a browser) and uses the React frontend framework. React is a framework that makes it easier to create complex web applications. We selected Scala.js as the programming language mostly because Voitto is written in Scala and we felt like the frontend would become complex enough that a strongly typed language will help in the long run. Also, we were familiar with Scala so jumping to Scala.js was quite simple.

We decided to use the React framework for the frontend as there was a library for using React with Scala.js that allowed writing React code with static typing. We were somewhat familiar with React which made this a simple choice. React makes it much easier to divide the UI into manageable components and it makes the data flow much easier to comprehend. We also used Bootstrap CSS templates to get a slightly nicer look with minimal effort.

The backend is a simple HTTP API that serves the existing resource files and settings from a database. It is implemented in Clojure programming language and uses a PostgreSQL database. This language and database were picked as we were familiar with them, and Clojure is almost a de facto standard at Yle in API creation and PostgreSQL is well supported in Clojure. The API has endpoints for fetching project settings and all the resource files edited with software. It also has endpoints for saving changes to the files and for receiving available attributes and values. The backend also serves the static HTML, CSS and JavaScript files that make the web application. The backend has HTTP basic access authentication for authorization. Heroku platform-as-a-service was used to run the backend server in the cloud as it was free and easy enough to use.

Information of all available attributes and boolean values needed to be imported from Voitto to the tool so that the tool can display lists of what attributes and values are available. Therefore there needed to be a way to list all attributes and values in Voitto's code. We came up with three different ways to implement this feature: Creating a map data structure that contains all the different values, that is manually updated by programmers;

using reflection to find all the values by checking the values' types, or using custom annotations to mark the attributes and values.

Unlike the other options, the map allows more customisation and does not require any kind of reflection. However, it would require manually updating the map, which could become arduous and programmers might forget to do it. Finding all attributes with reflection requires no additional programming when creating attributes or boolean values, but it permits from creating values that are not visible to the tool (e.g. creating intermediate values for code clarity that are not usable as-is, sort of private values in Voitto's code) It might also make the code unclear as the values being added automatically is in no way visible in the files where the values are.

We did proof-of-concept work on the detection of attributes and booleans with annotations. We chose annotations to minimize the amount of coding needed to turn a regular Scala value into a template attribute or decision tree value. It requires minimal programming/remembering to add a new value while also permitting the use of values that are not visible to the tool. While it is easy to use, it requires using reflection, which might be slightly slower than other implementations and reflection might reduce code clarity. The annotated attributes replace the aforementioned lists of attributes that each of Voitto's texts had before.

6 Analysis and Results

6.1 Simple UX Analysis

The tool was analysed first from a user experience (UX) perspective. User experience refers to how it feels to use the software and its perceived usability among other factors. The UX test was conducted with a journalist that was already familiar with Voitto but had no prior programming experience. We decided to conduct a UX test first to make sure the tool was as usable as possible, before testing if it solved the stated problem and achieved the set goals. This way the tool's possibly poor UX would not affect the perceived usefulness of the tool itself.

The UX test was done with a simple task list that used all the major features of the tool. A simplified version of the tasks is as follows:

- Add a new template with attributes that have inflections.
- Add the previously generated template to the decision tree at the right position.
- Find the documentation of a specific variable.
- Add a new automated text
- Fix an error that is on the front page
- Add a new language

The study showed the user experience to be mostly fluent, but it also revealed many small problems and a few major ones. While we had realised that it should be possible to create attributes that are not yet implemented in Voitto's code, we had not realised that it should be possible to create new decision tree variables as well. Therefore it was not possible to design new decision trees with the tool if all the variables were not implemented. This greatly reduced the usefulness of the tool in designing texts beforehand. Designing the text before implementing the variables was vital for the process to work as it would be very difficult to know what variables are really needed before the decision tree is designed.

We implemented the possibility to create new decision tree variables so that in the final analysis this would not be a hindrance. Unimplemented variables would show up as errors so the decision tree cannot be deployed if variables used by it are not implemented in Voitto's code.

Another issue discovered in the UX test was the poor naming in the navigation. Editing of automated texts was under the "Settings" page, which made them difficult to find. Decision trees were under "Logic" page, which was also unclear. We created a new page specifically for automatic texts and renamed "Logic" to "Decision trees".

When fixing an error or warning it was impossible to see the error message while solving the problem. We fixed this issue by adding a bar with the error message when clicking a link in the error message on the front page.

It was found difficult to follow the decision tree logic. We had to interrupt and say that branching right in the decision tree meant that the variable was true and when branching downwards, it was false. This made it immediately more clear. We decided to try to make this more obvious by adding arrows pointing the direction of true and false. Figuring out whether or not it is feasible for content creators to learn the decision tree logic using the tool is one of the major objectives of future analysis.

Other issues that came up in the UX testing were mostly things that are not immediately apparent, but easy to learn. For example, adding attributes in the middle of a template text required typing a curly bracket. It might be that the tool is not simple enough that it would be immediately useful without any training. This is not a major issue, as the tool will most likely not have so many users that it would be a major burden to teach them all how to use the tool.

Aside from poor naming, the navigation was easy to follow as all the different pages where on the top navigation bar. The template editor was also found easy to use and the attribute editor especially seemed to make attribute creation much easier.

6.2 Finnish Parliamentary Elections of 2019

The final analysis was done before the Finnish parliamentary election of 2019. The tool was used to create the texts and decision trees for Voitto's articles about the parliamentary election. As this was a complete transfer of Voitto to another subject from sports, this will also help analyse whether the increased modifiability actually helps with transferability.

After the Finnish parliamentary election of 2019, Voitto wrote an article about the election for each municipality in Finland. These articles were about who received the most votes in the municipality, how many new MPs are from the municipality, how the turnout changed compared to the previous election etc. All of the templates and decision trees for these reports were created with the tool by the same journalist with whom the UX test was conducted. The programmers did help a bit by warning about possible pitfalls. For example, that there is not always a person with the most votes, there can be a tie at the top.

This change of domain also proved how the tool could be modified for different use cases. The attribute types had to be changed for this use case from player and team names to municipality, party and candidate names. Changing them was simple enough, but there was no way to have different projects use different attribute types. This is a feature that should be implemented before the tool is completely ready. Perhaps each project should have an "article type" setting (e.g. sports, election, weather, etc.), which would affect the available attribute types.

During the creation of the texts, it was found vital to have a way to see what the texts would look like when combined. As Voitto does not have the “Referring expression generator” discussed in Chapter 3.2.1, there was repetition in the texts. For example, when the most popular candidate was mentioned multiple times in different templates, their full name was always used instead of using only using their last name on subsequent mentions. While it seemed fine when reading the texts separately, it was repetitive when reading them together. We implemented the previously discussed example texts page to help with these issues.

During these tests the tool did not have all the changes done to Voitto yet: the decision tree resource files and the resource file deployment process were not yet implemented. Therefore the programmers still had to implement the designed texts manually. However, according to a quick interview that was done with the other developer working on the project, the tool made it easier to implement the texts. The programmers did not have to think about the decision tree priority and the tool provided the texts easily in the correct format. All that was left for the programmers was creating a simple if-else if-else decision tree and implementing all attributes and variables. Hence the tool was found useful by the programmers even in its unfinished state.

The programmers and the content creators did not have to manually edit the template files, as they could be downloaded directly from the tool. Therefore issues with the complex syntax were removed by using the software.

6.3 Final Analysis

The journalist who used the tool to create the templates and decision trees of the parliamentary election of 2019 was interviewed in person to find out how well the tool achieved its goals. The questions that were asked were as follows:

- Using the tool, were you able to create the decision trees that you wanted?
- Could the decision tree editing process have been easier?
- Did the tool make it easier to create new texts?
- Were you able to create the texts autonomously?
- Did the tool make it easier to understand the text selection logic of Voitto?
- Did the tool help you understand what templates will be selected and why?
- Did the tool help you understand what the finished article will look like?
- Was it easier to edit existing templates with the tool?
- What possible shortcomings did the tool have?
- Were there any features missing?
- Were there any positives that were not yet mentioned?

The interviewee stated that they were able to create the decision trees they wanted, though it did take a while to understand how the decision trees worked. Especially the fact that different texts have their own trees, took a while to figure out. In the final interview, the decision tree editor was praised for tying it all together and making the entire text selection process much clearer.

One of the biggest hurdles was understanding and remembering the terminology, especially the terms “decision tree”, “template” and “automated text” got mixed up. As they were used as the names of different pages it was hard to remember which page did what. The hierarchy of the data was also tough to remember, i.e. automated texts have a single decision tree and the decision trees use templates.

The naming of variables, attributes and templates was also a source of confusion. In the resource files, variables and attributes use "camelCase" (words are strung together and each word, except the first one, starts with an uppercase letter) and template names use "kebab-case" (words are connected with hyphens). This made them harder to read and understand. This could be fixed by having the tool automatically convert the different cases to plain text.

The interviewee also suggested that there could be example texts in the decision tree to help remember what the different templates contained. Perhaps the previously discussed change of using normal language in template names would help, as the names should be descriptive enough so that example texts are not needed. Nevertheless, the example text can be added in the future if need be.

According to the interviewee, they were able to create the texts they wanted autonomously. Therefore it can be said that the tool did achieve its major objective.

The software made the template creation process much more efficient. The way the texts were designed previously was much more disorderly, now all required templates can be seen in the decision trees. Seeing empty decision tree leaves immediately told the journalist that more templates are needed. The decisions that had to be made to get to the empty node told the designer what the new template should state.

The example texts made it much easier to figure out which inflection to use. The attribute editor was found easy to use and immediately showed what different inflections and options were available. Therefore, editing existing texts was slightly easier than without the tool. The tool also made it more efficient and "smoother" to add translations and variations.

The tool did help with the transparency: The interviewee was able to better understand how the text selection logic using decision trees works even though they were already quite familiar with the inner workings of Voitto. They had a better understanding that the article was formed from independent text fragments instead of being a single large decision tree. They also realised that the decision tree logic starts with more newsworthy aspects instead of "bigger" decisions. For example, they expected the match being a tie or a win to be the first decision in the tree of football article headlines. Instead, the first decision was whether there is a hat-trick scoring player, as that is generally more newsworthy.

Unlike in sports articles, many more attribute options were used in the municipality election articles, which worked well with the tool. For example, each candidate attribute had options for total votes and percentage of votes in the municipality and nationwide, their party and their home municipality. Having various options for every attribute gave the content creators more attributes to use while requiring little effort from the programmers.

All in all the tool was able to fill its set goals but there is still a lot of room for improvement. The tool was able to improve the modifiability and it did also help change to a new domain so the transferability was enhanced as well. The transparency also improved as the visual decision tree and example texts made it much more clear what the article will look like in the end. In the following chapter, the remaining problems with the tool will be discussed.

6.4 Problems

Multiple problems rose up during the final testing done to the tool. While the tool achieved its goals as it is, these problems greatly limit its usefulness and at worst can lead to issues with accuracy.

The content editors don't have absolute control over the entire article. They cannot add completely new texts or change the order of texts in the article independently. This still requires some work from the programmers. We decided to leave this feature out as it is not vital and it would require a lot of work. A new editor would have to be made, which would list all the automated texts and allowed placing them within an article (e.g. title, lead, body text) and selecting their order and even adding paragraph changes to the correct positions. It would require even more work in Voitto's end. Voitto would need to receive this information and then build the article with it.

The user experience probably is not as good as it should be. It should be redone or at least overhauled with the help of a UX design professional. The tool overall is not too pleasant to look at. The template editor page has maybe a bit too many components and looks cluttered. In the front page, it is not immediately obvious what the user should click in order to edit the decision trees or the templates. There is little support for undoing changes in both the template editor and the decision tree editor. The terminology is difficult to understand. Translating the terms could also help understand them. The poor UX probably makes it significantly harder to start using the tool, especially if the domain is unfamiliar.

The final interview revealed that there is sometimes a need to copy and paste templates and attributes because some templates can be quite similar. This requires plenty of manual work, especially if there are many attributes, as they have to be created by hand.

The missing copy and paste feature also almost caused issues with accuracy during the creation of the parliamentary election articles. The content editors duplicated templates by copy-pasting the example text so that only attributes needed to be added. However, if they forgot to change a word to an attribute it would cause the attribute's value to be hardcoded to whatever was in the example text. For example, if the example text was `John Example got the most votes in Turku` the content creators forgot to change "Turku" back into an attribute so all texts using that template would always say "Turku" instead of the actual municipality. This again speaks for the advantage of having someone else go through the changes before they are deployed.

The error checking of the tool is done in the frontend. This is mostly due to the fact that it was easier to implement and good enough for this proof-of-concept work. Error checking should be moved to backend before deployment is implemented. Otherwise, the backend cannot check if there are errors prior to deployment and malicious users can deploy changes that will break the text generation logic.

Concurrent editing is not really tested and probably has its own issues. The server never sends the changes made by other users currently editing the resource files. Therefore, concurrent editing will save over changes made by other users. Fixing this would require big modifications to the communication between backend and frontend. If this is not fixed, users will have to communicate with each other every time they want to use the tool, so that they don't accidentally write over each other's changes. At worst, overwritten changes can cause issues with accuracy as already implemented texts are removed. The tool should also lock templates and decision trees if someone else is editing them at the same time to avoid issues caused by concurrent editing.

The tech decision of using the Scala.js programming language with the React frontend framework was probably the wrong one. Using React with the JavaScript programming language would have much more support and is more familiar to other developers. The advantage of using Scala for Voitto and for the tool is diminished due to the fact that the backend anyways uses the Clojure programming language, and the domains of Voitto and the tool are completely different. Also, we were not familiar enough with the framework when starting the project so parts of the code are of poorer quality and probably difficult to maintain.

There is no support for having multiple projects in the web application of the tool. Each project would be a different article, e.g. election articles, football articles etc. However, the backend server does support storing multiple projects. Multiple project support in the frontend should include a view for selecting which project to edit and it should support having template files that are available for multiple projects. It should also support having different attribute types, attributes and variables to be only available to specific projects.

The example text creation algorithm is too simple. Each attribute type only has one example text. For example, every municipality attribute is “Turku”. This shortcoming was especially problematic when trying to see if there was too much repetition in a text where the top three parties of a municipality were listed. Normally, they would always be three different parties but because all of the example attributes had the same party, it listed the same party thrice. This made it much more difficult to see if there was repetition, as the example text created repetition that would not be there in an actual text. The attribute example text should randomly pick from a list of attribute examples and make sure that different attributes in the same template would use different examples.

6.5 Further Improvements

There are many improvements that could be done to the tool outside of fixing the issues raised in the previous chapter. Some of these ideas should be implemented if this tool is taken to regular use, as they further increase the usability of the tool. Others are much more optional and “nice to have” features. The features left out of the scope in Chapter 5.7 should be added. For example, automatic spell checking and version control would make the tool much safer to use.

The decision tree editor should have support for “and” and “or” boolean functions so that the content creators could have multiple conditions in one branching node. It is possible to implement the same logic, it would just require creating multiple branches that are the same.

An error or a warning fixing mode could also be added. In this mode, the tool would show all of the problems, one after the other. The user would first fix the issue and then click to move to the next issue. Currently, the user has to first fix an issue then go back to the front page to see what the next problem is. An error fixing mode would make fixing all errors require less back and forth.

As the software uses the free version of the Heroku platform-as-a-service, the start-up time of the page is very high. A comprehensive rework of the software's infrastructure should be done to improve the start-up time and make the deployment of the tool easier for developers. At the time of writing, the deployment process requires the developers to manually build JavaScript files from the Scala.js files and copy them to the server's resource files. This should be automated.

Automated tests should be added for both the frontend and the backend code. These were mostly neglected due to the proof-of-concept nature of the work. Before the tool is taken into production, a comprehensive automated test suite should be programmed to reduce bugs.

6.6 Other Possible Use Cases

As the tool outputs generic EDN resource files, the tool could possibly be used in other cases as well. Of course, other use cases will require some modifications to the tool, e.g. changing the available attribute types. As the tool's template and decision tree editing components are mostly separated, they can be used separately. EDN is relatively widely supported so there are existing parsers that will help with reuse in other use cases. Though custom template attribute parsing must be written.

The template editor can be used for anything that uses templates and especially if the templates have to be translated. Therefore it could be used to create translations for software. For example, game translations could be a good use case as they might use all the features of the template structure: translations for different languages, attributes (e.g. player character name) and multiple variations (helps if the same text is repeated multiple times).

The decision tree editor is just a generic binary decision tree editor so it can be used anywhere where it makes sense to have decision trees in resource files. The tool does require the values in the leaves to be known, so that they can be displayed as a list, which limits reuse as-is.

There are limited use cases where both the template editor and the decision tree can be used. The only one we could think of is a dialogue or decision-based game. The player selects what they want to say or do, and the game selects templates from the decision tree based on the choices of the player.

The tools usability as-is is limited by the limitations of Voitto: only some inflections are available for certain attribute types and there is a limited set of different attribute types. However these limitations are separated from other code of the tool, so editing them should not be too difficult a task.

7 Conclusions

In this thesis, we explored the topic of automated journalism. Automated journalism means writing articles automatically based on structured data. It can be used on subjects were repetitive reporting is required and there exists plenty of good quality data. Such topics are most commonly sports, finance, weather and election reports. After implementation, automated journalists are independent and do not need human input.

The advantages and disadvantages of automated journalists were also discussed. The advantages of automated journalism include speed, scalability and lower costs. The articles written automatically can also be personified for the reader. The disadvantages are that automated journalists have limited perception of subjects and the articles are generally of worse quality, e.g. they are less fluent.

Six requirements put on automated journalists were also discussed. The requirements are transparency, modifiability and transferability, data availability, topicality, fluency and accuracy. If an automated journalist fulfils these requirements it can be considered to hold the same standards as a human reporter.

We also went through different implementation methods of automated journalists and compared them against the aforementioned requirements. These implementation methods were manually written rules, data analysis based fact ranking and machine learning solutions. None of the implementation methods was found strictly better than the others, but all had their distinct advantages, requirements and problems.

Selecting the correct implementation method for a use case is context-dependent. The requirements should be weighed and a suitable implementation method could be picked according to them. For example, if only one automated journalist is required, transferability can be ignored.

More research should be done to find out more about the differences between implementation methods. Further studies should be done to find out if there are other requirements that should affect the implementation method selecting, outside of the six requirements presented in this thesis.

Finnish national broadcasting company Yle's automated journalist Voitto was also explored in this thesis. Voitto is an in-house solution which is based on templates and decision trees. Voitto was also measured up against the requirements put on automated journalists. While Voitto's structure provided it with good modifiability and transparency, these were only available to programmers who understood the Scala programming language. For others, the decision trees were impossible to understand and the templates were difficult to read.

A piece of software was created to tackle the fact that the good modifiability and transparency were only available to programmers. The software was implemented as a single-page web application with Scala.js programming language and the React framework. The application was a proof-of-concept that had all the template and decision tree editing tools while still requiring programmers to implement the actual texts.

Pre-existing tools were also explored to find if any of them fit the use case. Unfortunately, none of them truly fixed the issues. Some tools were too focused while others did not support the Finnish language. Also, the tools were closed-source which was considered a minus as the project would be dependent on them.

The created piece of software was analysed multiple times. First, it was analysed from a user experience perspective so that the user experience could be improved before the final analysis. This was done to prevent bad UX from skewing the results of the actual usability of the tool.

After the UX analysis and improvements, the tool was used to create templates and decision trees for the election reports written by Voitto about the parliamentary election of Finland in 2019. After the content was created, an interview was conducted with the content creator who used the tool and with the software developer who implemented the texts. The tool was found useful by both and it achieved its goal of improving the transparency, modifiability and transferability of Voitto. However, it was found that the tool might take a while to learn.

The tool would still require much work until it is completely usable. For example, the UX should be improved to fix the flaws that came up in during the final testing. The features that were left out for the scope of this thesis should be implemented. These include automatic deployment of the resource files to Voitto, version control for the texts and support for multiple projects. These features would make it possible for journalists to edit the texts with minimal support from programmers.

Further analysis should be done to see how well the software works when fully implemented. It should also be tested with journalists who have never before worked with an automated journalist. It could also be tested for other use cases, as the tool creates resource files that can be used by other software as well.

References

[1] A. Graefe, “Guide to automated journalism”, 2016.

[2] L. Leppänen, M. Munezero, M. Granroth-Wilding, and H. Toivonen, “Data-driven news generation for automated journalism”, in *Proceedings of the 10th International Conference on Natural Language Generation*, 2017, pp. 188–197.

[3] M. Melin, A. Bäck, C. Södergård, M. D. Munezero, L. J. Leppänen, H. Toivonen, *et al.*, “No landslide for the human journalist”, *IEEE Access*, 2018.

[4] T. C.-G. Linden *et al.*, “Algorithms for journalism”, *The journal of media innovations*, 2017.

[5] K. N. Dörr, “Mapping the field of algorithmic journalism”, *Digital Journalism*, vol. 4, no. 6, pp. 700–722, Aug. 2016.

[6] S. Wiseman, S. Shieber, and A. Rush, “Challenges in data-to-document generation”, in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 2253–2263. DOI: `10.18653/v1/D17-1239`. [Online]. Available: `https://www.aclweb.org/anthology/D17-1239`.

[7] N. Thurman, K. Dörr, and J. Kunert, *When reporters get hands-on with Robo-Writing*, 2017.

[8] L. Leppänen, M. Munezero, S. Sirén-Heikel, M. Granroth-Wilding, and H. Toivonen, “Finding and expressing news from structured data”, in *Proceedings of the*

21st International Academic Mindtrek Conference, ser. AcademicMindtrek '17, Tampere, Finland: ACM, 2017, pp. 174–183.

[9] L. Dou, G. Qin, J. Wang, J.-G. Yao, and C.-Y. Lin, "Data2Text studio: Automated text generation from structured data", in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2018, pp. 13–18.

[10] F. Nie, J. Wang, J.-G. Yao, R. Pan, and C.-Y. Lin, "Operation-guided neural networks for high fidelity data-to-text generation", in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 3879–3889. [Online]. Available: `https://www.aclweb.org/anthology/D18-1422`.

[11] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation", in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02, Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318.

Appendix A Decision tree

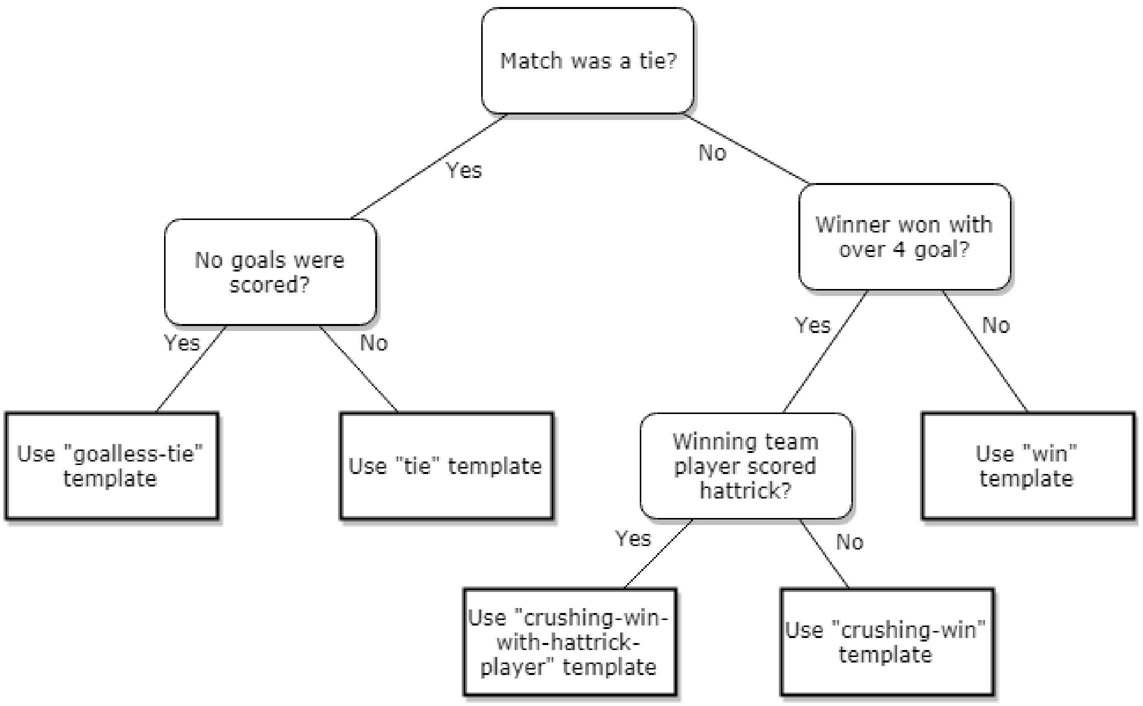

Match was a tie?
Yes
No
No goals were scored?
Winner won with over 4 goal?
Yes
No
Yes
No
Use "goalless-tie" template
Use "tie" template
Winning team player scored hattrick?
Use "win" template
Yes
No
Use "crushing-win-with-hattrick-player" template
Use "crushing-win" template

Figure A.1: Decision tree as a graph

A.1 Graph

A.2 Scala Code

```
if (tie)
  if (noGoals) loadTemplate("goalless-tie")
  else loadTemplate("tie")
else if (fourGoalDifference)
  if (winnerHasHattrickPlayer)
    ↪  loadTemplate("crushing-win-with-hattrick-player")
  else loadTemplate("crushing-win")
else loadTemplate("win")
```

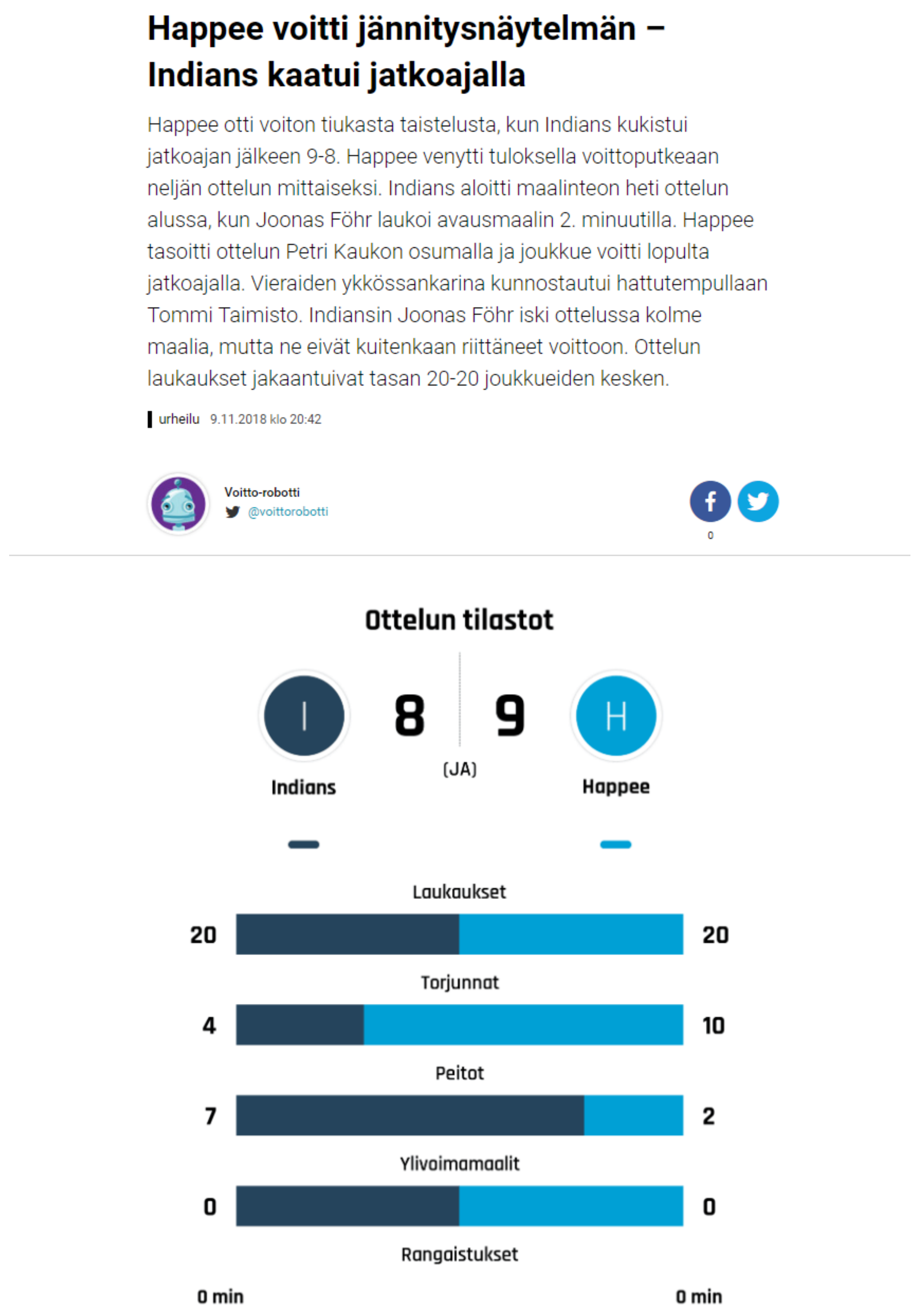
A.3 EDN Resource File

```
[:tie
 [:noGoals
  "goalless-tie"
  "tie"]
 [:fourGoalDifference
  [:winnerHasHattrickPlayer
   "crushing-win-with-hattrick-player"
   "crushing-win"]
  "win"]]
```

Appendix B Article Written by Voitto

Source: Yleisradio Oy, `https://yle.fi/urheilu/3-10501743`

B.1 Randomly Selected Article Written by Voitto



Happee voitti jännitysnäytelmän – Indians kaatui jatkoajalla

Happee otti voiton tiukasta taistelusta, kun Indians kukistui jatkoajan jälkeen 9-8. Happee venytti tuloksella voittoputkeaan neljän ottelun mittaiseksi. Indians aloitti maalinteon heti ottelun alussa, kun Joonas Föhr laukoi avausmaalin 2. minuutilla. Happee tasoitti ottelun Petri Kaukon osumalla ja joukkue voitti lopulta jatkoajalla. Vieraiden ykkössankarina kunnostautui hattutempullaan Tommi Taimisto. Indiansin Joonas Föhr iski ottelussa kolme maalia, mutta ne eivät kuitenkaan riittäneet voittoon. Ottelun laukaukset jakaantuivat tasan 20-20 joukkueiden kesken.

urheilu 9.11.2018 klo 20:42

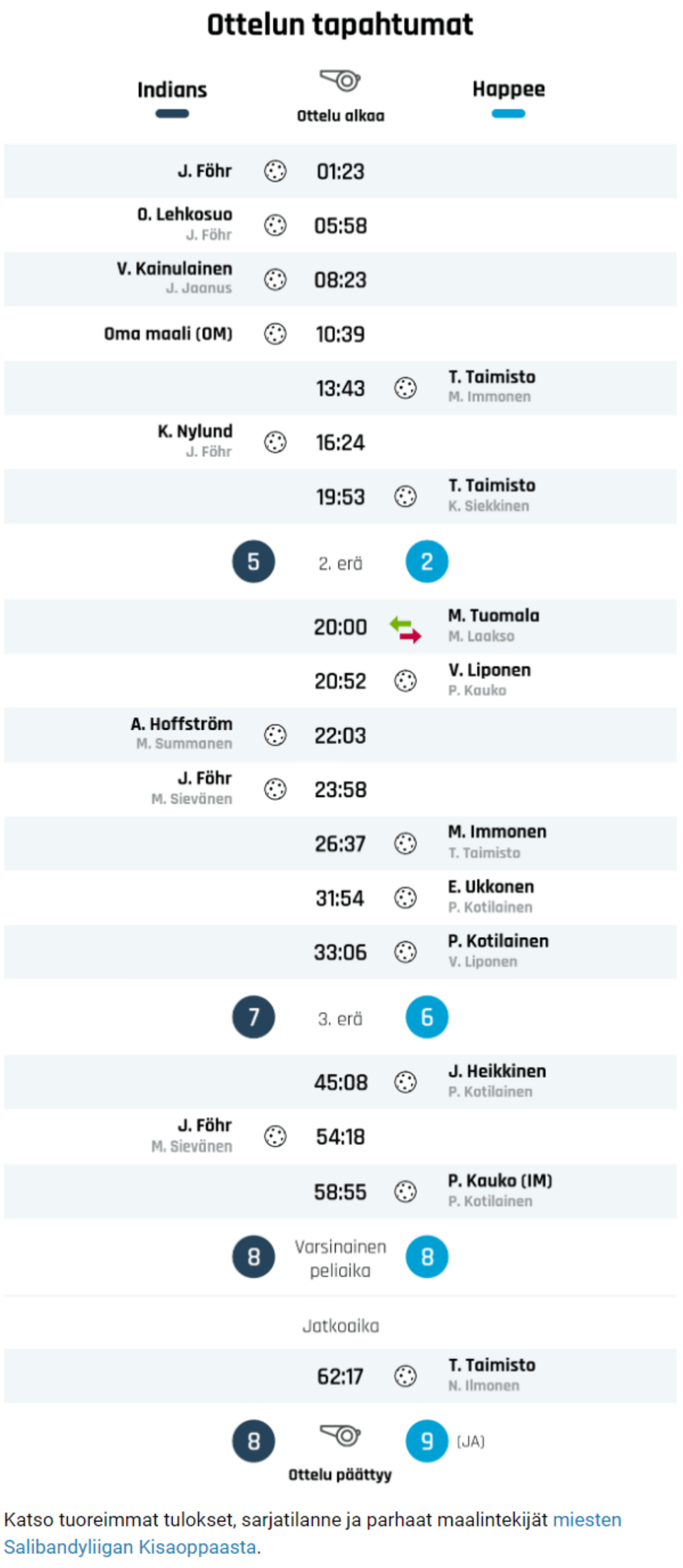
Voitto-robotti
@voittorobotti

0

Ottelun tilastot

I 8 | 9 H
(JA)
Indians Happee

Indians		Happee
20	Laukaukset	20
4	Torjunnat	10
7	Peitot	2
0	Ylivoimamaalit	0
0 min	Rangaistukset	0 min



Ottelun tapahtumat

Indians	Ottelu alkaa	Happee
J. Föhr ⚽	01:23	
O. Lehkosuo ⚽ J. Föhr	05:58	
V. Kainulainen ⚽ J. Jaanus	08:23	
Oma maali (OM) ⚽	10:39	
	13:43	⚽ T. Taimisto M. Immonen
K. Nylund ⚽ J. Föhr	16:24	
	19:53	⚽ T. Taimisto K. Siekkinen
5	2. erä	2
	20:00	⇄ M. Tuomala M. Laakso
	20:52	⚽ V. Liponen P. Kauko
A. Hoffström ⚽ M. Summanen	22:03	
J. Föhr ⚽ M. Sievänen	23:58	
	26:37	⚽ M. Immonen T. Taimisto
	31:54	⚽ E. Ukkonen P. Kotilainen
	33:06	⚽ P. Kotilainen V. Liponen
7	3. erä	6
	45:08	⚽ J. Heikkinen P. Kotilainen
J. Föhr ⚽ M. Sievänen	54:18	
	58:55	⚽ P. Kauko (IM) P. Kotilainen
8	Varsinainen peliaika	8
	Jatkoaika	
	62:17	⚽ T. Taimisto N. Ilmonen
8	Ottelu päättyy	9 (JA)

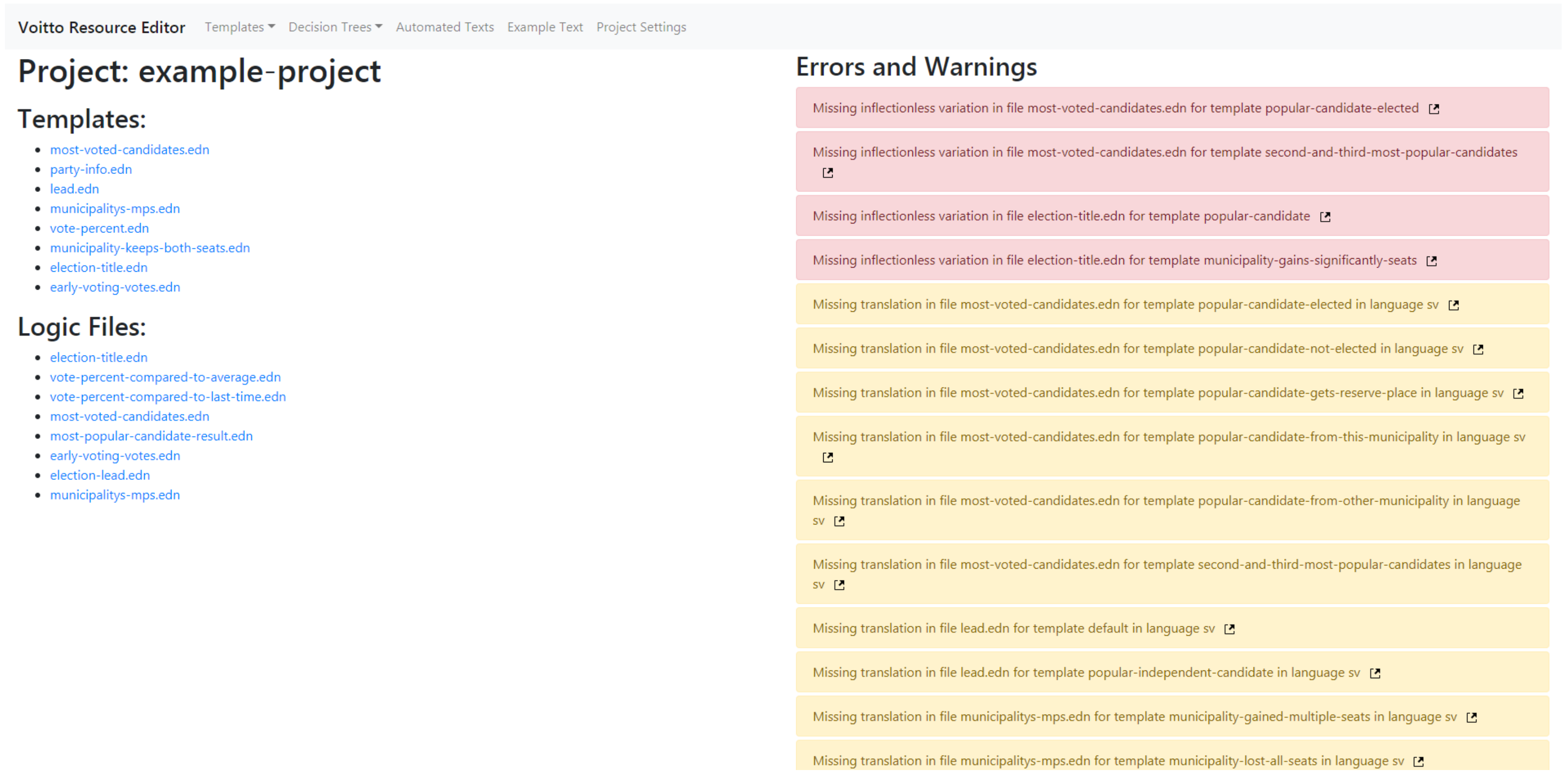
Katso tuoreimmat tulokset, sarjatilanne ja parhaat maalintekijät miesten Salibandyliigan Kisaoppaasta.

Appendix C Text Template Used by Voitto

C.1 Template File Used by Voitto

```
{
 :team-won
 {:fi
  (["{{winner}} vei voiton {{loser:elative}} tuloksella
   ↪ {{goalsPerPeriod}}."]
   ["{{winner}} otti voiton {{loser:elative}} lukemin
    ↪ {{goalsPerPeriod}}."]
   ["{{winner}} kaatoi {{loser:accusative}}
    ↪ {{series:inessive}} luvuin {{goalsPerPeriod}}."]
   ["{{winner}} voitti {{loser:accusative}}
    ↪ {{series:inessive}} lukemin {{goalsPerPeriod}}."]
   ["{{loser}} kaatui, kun {{winner}} otti voiton lukemin
    ↪ {{goalsPerPeriod}}." {:weight 0.1}])
  :sv
  (["Matchen mellan {{homeTeam:nominative}} och
   ↪ {{awayTeam:nominative}} avgjordes med siffrorna
   ↪ {{goalsPerPeriod}}."])}

 :crushing-win
 {:fi
  (["{{winner}} vei murskavoiton {{loser:elative}} lukemin
   ↪ {{goalsPerPeriod}}."]
   ["{{winner}} otti murskavoiton {{loser:elative}} luvuin
    ↪  {{goalsPerPeriod}}."]
   ["{{winner}} tyrmäsi {{loser:accusative}}
    ↪  {{series:inessive}} lukemin {{goalsPerPeriod}}."]
   ["{{winner}} jyräsi selvään {{result}} -voittoon
    ↪  {{loser:elative}}."]
   ["{{winner}} vei murskavoiton {{loser:elative}}, kun
    ↪  joukkueiden kohtaaminen päättyi luvuin
    ↪  {{goalsPerPeriod}}."]
   ["{{loser}} kaatui murskalukemin, kun {{winner}} otti
    ↪  {{result}} -voiton." {:weight 0.1}])
  :sv
  (["{{homeTeam:nominative}} krossade
   ↪  {{awayTeam:nominative}} med siffrorna
   ↪  {{goalsPerPeriod}}."])}

 :tie
 {:fi
  (["{{homeTeam}} ja {{awayTeam}} eivät saaneet selvitettyä
   ↪  paremmuuttaan, vaan pisteet jaettiin numeroin
   ↪  {{goalsPerPeriod}}. {{lastTyingGoalTeam.opponent}}
   ↪  menetti johtoasemansa {{lastTyingGoalMinutes}}.
   ↪  minuutilla, kun {{lastTyingGoalPlayer}} sinetöi
   ↪  loppulukemat."]
   ["{{homeTeam}} ja {{awayTeam}} päätyivät tasapeliin
    ↪  {{goalsPerPeriod}}. {{lastTyingGoalTeam:genitive}}
    ↪  {{lastTyingGoalPlayer}} tasoitti ottelun
    ↪  {{lastTyingGoalMinutes}}. minuutilla."]
   ["{{homeTeam}} ja {{awayTeam}} pelasivat {{result}}
    ↪  -tasapelin. {{lastTyingGoalTeam:genitive}}
    ↪  {{lastTyingGoalPlayer}} sinetöi loppulukemat
    ↪  tasoitusmaalillaan {{lastTyingGoalMinutes}}.
    ↪  minuutilla."]
   ["{{homeTeam}} ja {{awayTeam}} tahkosivat tasapelin
    ↪ maalein {{goalsPerPeriod}}. Viimeisen tasoittavan
    ↪ maalin iski {{lastTyingGoalTeam:genitive}}
    ↪ {{lastTyingGoalPlayer}} {{lastTyingGoalMinutes}}.
    ↪ minuutilla."])
  :sv
  (["Matchen mellan {{homeTeam:nominative}} och
   ↪ {{awayTeam:nominative}} slutade oavgjort
   ↪ {{goalsPerPeriod}}.
   ↪ {{lastTyingGoalTeam.opponent:nominative}} tappade
   ↪ ledningen i den {{lastTyingGoalMinutes}} minuten."])}

 :goalless-tie
 {:fi
  (["Maalivahdit nousivat sankareiksi 0-0 -lukemiin
   ↪ päättyneessä tasapelissä. {{homeTeam}} laukoi
   ↪ {{homeShotsOnTargetCount}} kertaa maalia kohti ja
   ↪ {{awayTeam}} {{awayShotsOnTargetCount}}."]
   ["Hyvin pelanneet maalivahdit varmistivat 0-0
    ↪ -tasapelin. {{homeTeam}} onnistui laukomaan
    ↪ {{homeShotsOnTargetCount}} ja {{awayTeam}}
    ↪ {{awayShotsOnTargetCount}} kertaa maalia kohti."])
  :sv
  (["Matchen mellan {{homeTeam:nominative}} och
   ↪ {{awayTeam:nominative}} slutade oavgjort 0-0. Bägge
   ↪ lagens målvakterr gjorde sitt arbete väl -
   ↪ {{homeTeam:nominative}} sköt
   ↪ {{homeShotsOnTargetCount}} mot mål, medan
   ↪ {{awayTeam:nominative}} avfyrade
   ↪ {{awayShotsOnTarget}} skott mot mål utan
   ↪ resultat."])}

 :won-in-overtime
 {:fi
  (["{{winner}} otti voiton tiukasta taistelusta, kun
   ↪ {{loser}} kukistui jatkoajan jälkeen
   ↪ {{winnerResult}}."])
  :sv
  (["{{winner:nominative}} slog {{loser:nominative}} efter
   ↪  förlängning med {{winnerResult}}."])}

 :won-in-penalty-shootout
 {:fi
  (["{{winner}} sai taisteltua itselleen voiton, kun
   ↪  {{loser}} taipui rangaistuslaukauksilla."])
  :sv
  (["{{loser:nominative}} förlorade mot
   ↪  {{winner:nominative}} efter straffar."])}}
```

Appendix D Images of the Software



Project: example-project

Templates:

- most-voted-candidates.edn
- party-info.edn
- lead.edn
- municipalitys-mps.edn
- vote-percent.edn
- municipality-keeps-both-seats.edn
- election-title.edn
- early-voting-votes.edn

Logic Files:

- election-title.edn
- vote-percent-compared-to-average.edn
- vote-percent-compared-to-last-time.edn
- most-voted-candidates.edn
- most-popular-candidate-result.edn
- early-voting-votes.edn
- election-lead.edn
- municipalitys-mps.edn

Errors and Warnings

Missing inflectionless variation in file most-voted-candidates.edn for template popular-candidate-elected

Missing inflectionless variation in file most-voted-candidates.edn for template second-and-third-most-popular-candidates

Missing inflectionless variation in file election-title.edn for template popular-candidate

Missing inflectionless variation in file election-title.edn for template municipality-gains-significantly-seats

Missing translation in file most-voted-candidates.edn for template popular-candidate-elected in language sv

Missing translation in file most-voted-candidates.edn for template popular-candidate-not-elected in language sv

Missing translation in file most-voted-candidates.edn for template popular-candidate-gets-reserve-place in language sv

Missing translation in file most-voted-candidates.edn for template popular-candidate-from-this-municipality in language sv

Missing translation in file most-voted-candidates.edn for template popular-candidate-from-other-municipality in language sv

Missing translation in file most-voted-candidates.edn for template second-and-third-most-popular-candidates in language sv

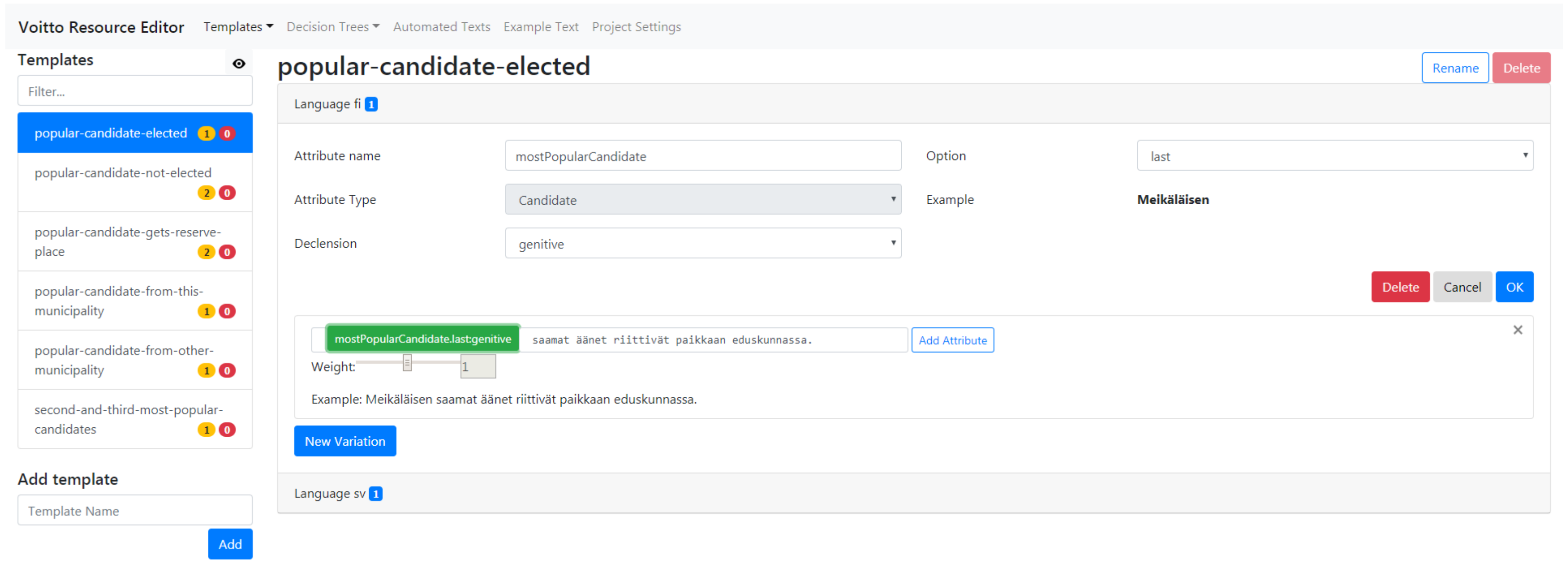
Missing translation in file lead.edn for template default in language sv

Missing translation in file lead.edn for template popular-independent-candidate in language sv

Missing translation in file municipalitys-mps.edn for template municipality-gained-multiple-seats in language sv

Missing translation in file municipalitys-mps.edn for template municipality-lost-all-seats in language sv

Figure D.1: The Front page



Templates

Filter...

popular-candidate-elected 1 0

popular-candidate-not-elected 2 0

popular-candidate-gets-reserve-place 2 0

popular-candidate-from-this-municipality 1 0

popular-candidate-from-other-municipality 1 0

second-and-third-most-popular-candidates 1 0

Add template

Template Name

popular-candidate-elected

Language fi 1

Attribute name: mostPopularCandidate

Option: last

Attribute Type: Candidate

Example: **Meikäläisen**

Declension: genitive

mostPopularCandidate.last:genitive `saamat äänet riittivät paikkaan eduskunnassa.`

Weight: 1

Example: Meikäläisen saamat äänet riittivät paikkaan eduskunnassa.

Language sv 1

Figure D.2: Template editor with the attribute editor open

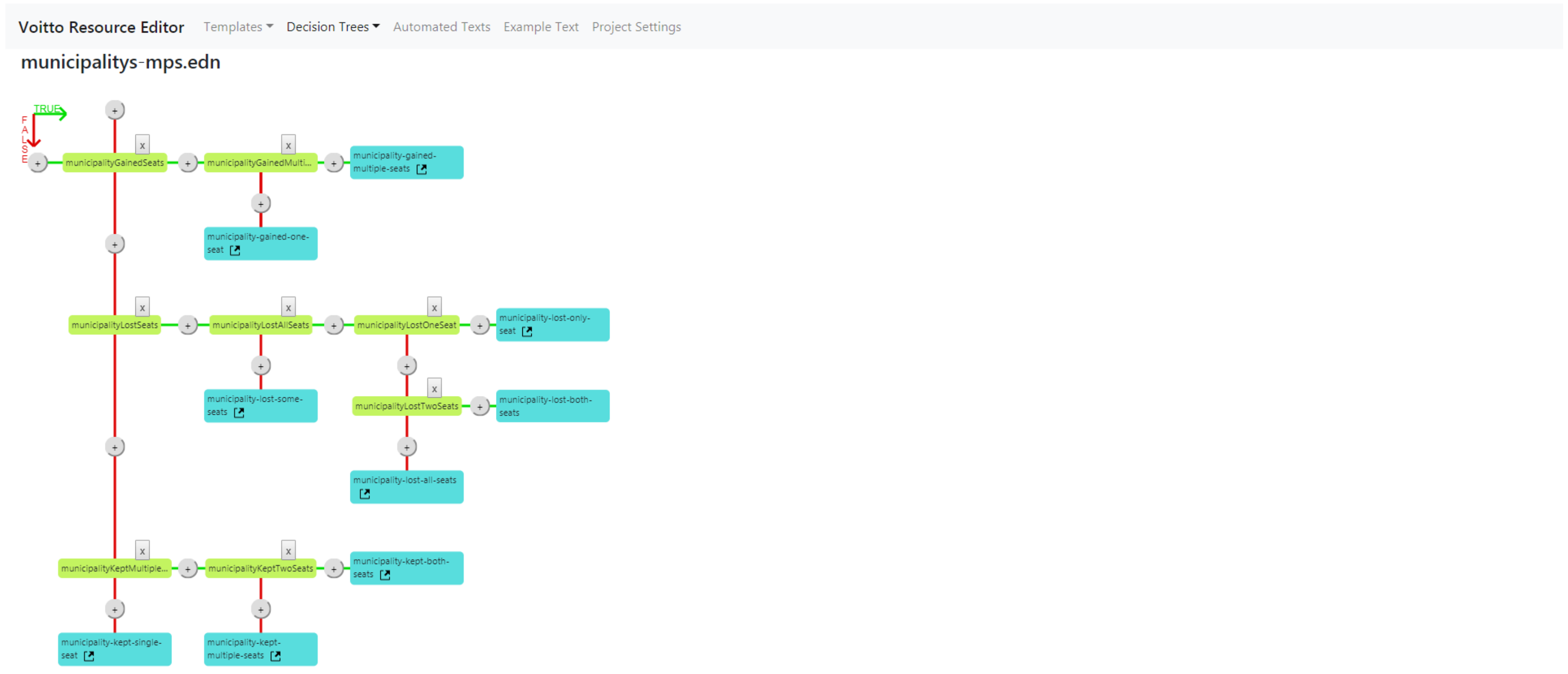
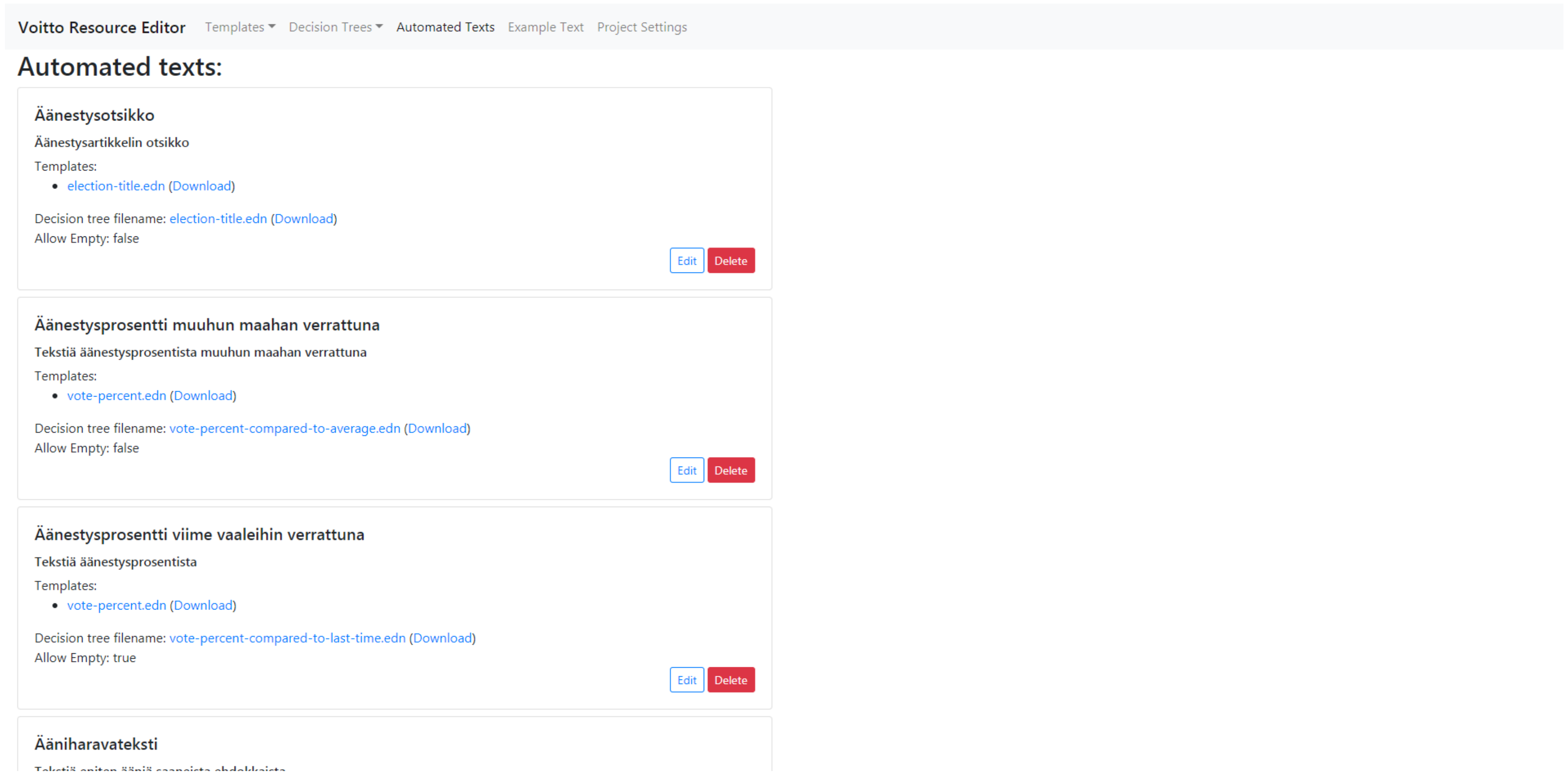

Voitto Resource Editor
Templates
Decision Trees
Automated Texts
Example Text
Project Settings
municipalitys-mps.edn
TRUE
FALSE
municipalityGainedSeats
municipalityGainedMulti...
municipality-gained-multiple-seats
municipality-gained-one-seat
municipalityLostSeats
municipalityLostAllSeats
municipalityLostOneSeat
municipality-lost-only-seat
municipality-lost-some-seats
municipalityLostTwoSeats
municipality-lost-both-seats
municipality-lost-all-seats
municipalityKeptMultiple...
municipalityKeptTwoSeats
municipality-kept-both-seats
municipality-kept-single-seat
municipality-kept-multiple-seats

Figure D.3: Decision tree editor



Automated texts:

Äänestysotsikko

Äänestysartikkelin otsikko

Templates:

- election-title.edn (Download)

Decision tree filename: election-title.edn (Download)
Allow Empty: false

Äänestysprosentti muuhun maahan verrattuna

Tekstiä äänestysprosentista muuhun maahan verrattuna

Templates:

- vote-percent.edn (Download)

Decision tree filename: vote-percent-compared-to-average.edn (Download)
Allow Empty: false

Äänestysprosentti viime vaaleihin verrattuna

Tekstiä äänestysprosentista

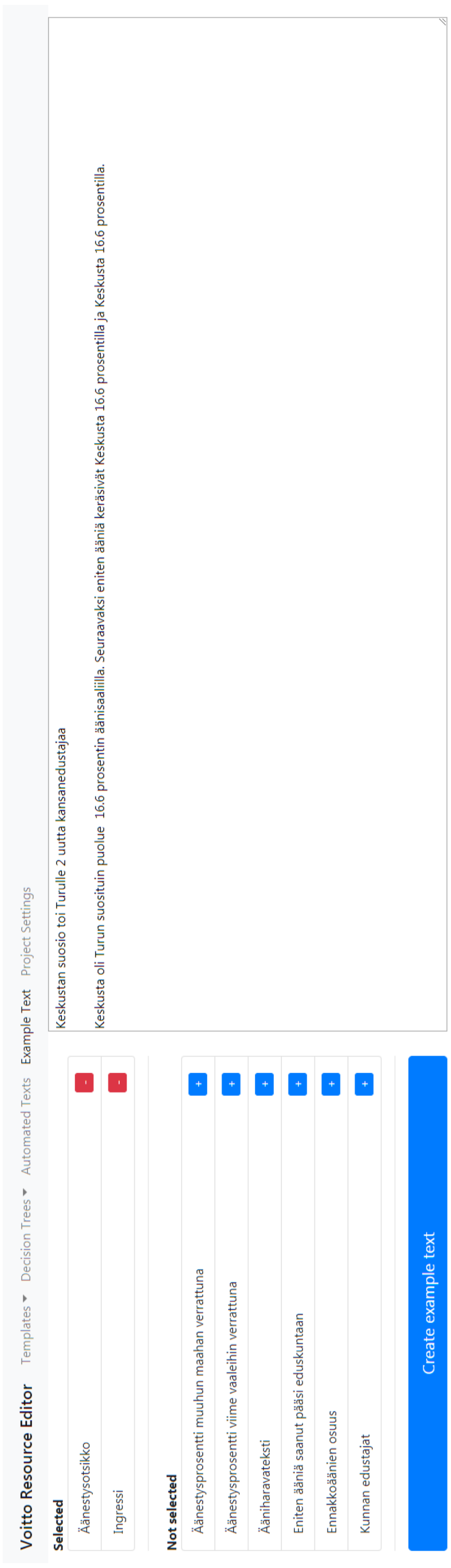
Templates:

- vote-percent.edn (Download)

Decision tree filename: vote-percent-compared-to-last-time.edn (Download)
Allow Empty: true

Ääniharavateksti

Figure D.4: Automated texts editor



Selected

- Äänestysotsikko
- Ingressi

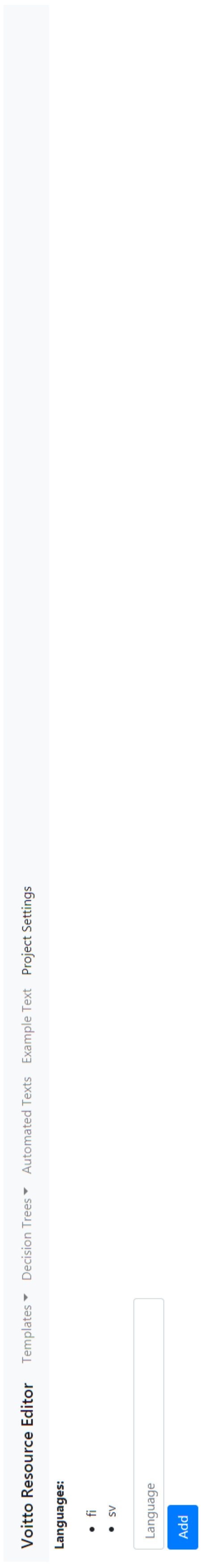
Not selected

- Äänestysprosentti muuhun maahan verrattuna
- Äänestysprosentti viime vaaleihin verrattuna
- Ääniharavateksti
- Eniten ääniä saanut pääsi eduskuntaan
- Ennakkoäänien osuus
- Kunnan edustajat

Keskustan suosio toi Turulle 2 uutta kansanedustajaa

Keskusta oli Turun suosituin puolue 16.6 prosentin äänisaalilla. Seuraavaksi eniten ääniä keräsivät Keskusta 16.6 prosentilla ja Keskusta 16.6 prosentilla.

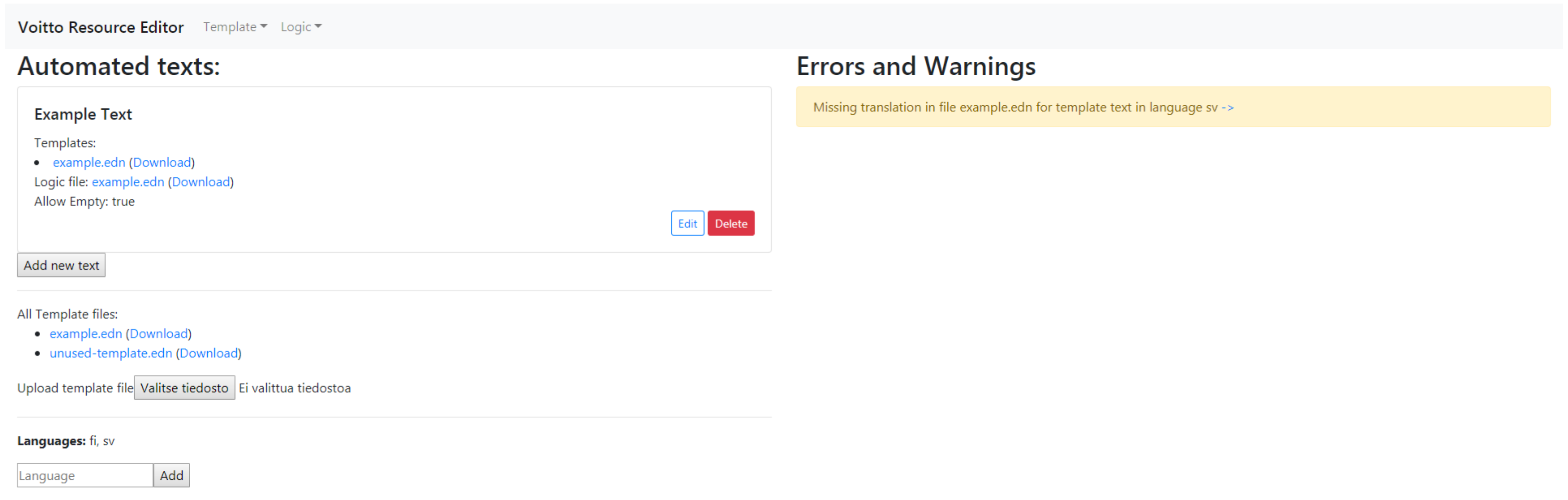
Figure D.5: Example text page



Languages:

- fi
- sv

Figure D.6: Settings



Automated texts:

Example Text

Templates:

- example.edn (Download)

Logic file: example.edn (Download)

Allow Empty: true

All Template files:

- example.edn (Download)
- unused-template.edn (Download)

Upload template file Ei valittua tiedostoa

Languages: fi, sv

Language

Errors and Warnings

Missing translation in file example.edn for template text in language sv ->

Figure D.7: Old front page