Predicting the Maintainability of XSL Transformations

Siim Karus∗,a, Marlon Dumas∗

∗University of Tartu, Estonia

Abstract

XSLT is a popular language for implementing both presentation templates in Web applications as well as document and message converters in enterprise applications. The widespread adoption and popularity of XSLT raises the challenge of efficiently managing the evolution of significant amounts of XSLT code. This challenge calls for guidelines and tool support for developing maintainable XSLT code. In this setting, this paper studies two hypotheses: (i) that the maintainability of XSL transformations, measured in terms of code churn, can be predicted using a combination of simple metrics; and (ii) that the purpose of an XSL transformation (presentation template or document transformation) has only a minor effect on code churn and one single general model can be used to predict code churn for both types of transformations.

These hypotheses are tested on several open-source software project repositories. An outcome of this empirical study is a statistical model for predicting the maintainability of XSL transformations with relatively high accuracy.

Key words: Software Maintenance, XML, XSLT, Software Metrics

1. Introduction

Despite considerable efforts put into document schema standardization, especially in the context of XML, modern information systems predominantly use their own internal representations in order to achieve higher functionality or to optimize document processing. As a result, documents moving across different domains (e.g. from a healthcare information system to an accounting system) often need to be transformed in order to cope with schema mismatches. XSLT (Extensible Stylesheet Language Transformations [1]) is the dominant transformation language in contemporary document mediation and enterprise application integration products.

In multi-tiered Web information systems, XSLT is also widely used for transforming between internal (XML) formats used at the data and business logic layer, and formats used at the presentation layer (e.g. HTML, RSS, XSL-FO,
SVG and others). In addition, due to its portability, XML has found its way into desktop applications as well.

Oftentimes, XSLT is used in a supportive role as a language for accomplishing specific tasks and amounts to a rather small percentage of the total code base of a project. However, this is not always the case. For example, according to ohloh.net\(^1\), a few popular open-source products such as Smarty\(^2\) and DocBook\(^3\) are written mostly in XSLT. Specifically, more than 70\% of the code base of these projects is made of XSLT. Documentation projects\(^4\) like “Linux From Scratch”\(^5\) have even higher percentage of XSLT code – over 90\%, when excluding XML data files. More business like projects like Dragon SOA Governance\(^6\) and TYPO3 Phenix\(^7\) have more than 60\% of program code in XSLT. This suggests that the bulk of the maintenance efforts in these projects goes into keeping XSLT code up to date. The maintainability of XSLT code is thus key to the sustainability of these and similar projects. But while there are many metrics for evaluating the maintainability of software units coded in mainstream general-purpose languages\(^8\), there are no available techniques for evaluating the maintainability of XSLT code.

Maintainability refers to the level of effort required to evolve a software unit over time in order to correct defects and to cope with new requirements or changes in its environment. A common approach to measure maintainability a posteriori is in terms of code churn: the number of lines of code added, deleted or modified over a number of versions of a software unit [2]. Empirical studies have shown that code churn is a good indicator of the software development effort [3]. Naturally, this does not mean that code churn should be equated to maintenance effort. In some cases modifications may be the result of standard refactoring operations (e.g. method renaming) that do not necessarily require a lot of effort from the developer, while in other cases small code changes may be preceded by an in-depth analysis of a part of a program.

Code churn measures have shown to be useful, not only as indicators of development effort, but also as indicators or predictors of defects. Studies have shown that there is a high correlation between churn measures and the number of faults found during testing [4] – which entail additional corrective maintenance – and that relative code churn measures are good predictors of system defect density [5].

\(^1\)http://www.ohloh.net/
\(^2\)http://www smarty.net/
\(^3\)http://www.docbook.org/
\(^4\)Documentation projects are projects where instructions or manuals are made. For example, “Linux From Scratch” project builds documentation on how to build and configure a Linux system from sources as opposed to using distribution packages.
\(^5\)http://www.linuxfromscratch.org/
\(^6\)http://dragon.ow2.org/
\(^7\)http://typo3.org/
\(^8\)We can cite for example Visual Studio’s Maintainability Index and the Software Engineering Institute (SEI) Maintainability Index, which are based on linear combinations of complexity metrics such as McCabe and Halstead.
In this setting, this paper is concerned with predicting (ex ante) the code churn of an XSL transformation. The hypotheses studied are:

1. the future code churn of a given XSL transformation can be predicted using a combination of simple count metrics;
2. the purpose of an XSL transformation (presentation template or document transformation) has only a minor effect on code churn and one single general model can be used to predict code churn for both types of transformations.

Specifically, the paper studies the prediction of the level of LOC churn in the next revision of a given XSL transformation. Rather than predicting the specific values of LOC churn, we divide the possible values of code churn into ranges (low, medium and high) by examining the distribution of LOC churn across all projects considered, as detailed later in the paper. The question is then how to predict the level of code churn of an XSL transformation in the next revision of the transformation. To this end we systematically identify a set of plausible XSL code metrics and we investigate the applicability of machine learning techniques to answer the above question.

The contribution of the paper is an application of machine learning techniques to build models for predicting the level of code churn of XSLT code with relatively high degree of accuracy. This model has been embodied in an online tool that calculates a score indicating the maintainability of an XSL Transformation given a set of basic count metrics. The tool is available at: http://sandstorm.cs.ut.ee/VCSAnalysisServices/.

The rest of the paper is structured as follows. Section 1 introduces and justifies the candidate metrics considered in this research. The evaluation method and data sets used are described in section 3. The results of the study are presented and discussed in section 4, followed by related work in section 5.

2. Background

2.1. XSLT Overview

An XSLT template takes as input an XML document and produces another document (generally an HTML or XML document, but not necessarily). A typical XSLT stylesheet is composed of templates that match elements in the input XML document. The matching of templates with XML elements is based on a “match” expression associated with the template. This expression is written in XPath, generally using path expressions. A “match” expression can refer to the location of elements by using concrete element names (e.g. /Book/Title) or it may use function calls and wildcards (e.g. “*” which matches any sequence of characters). In the first case we talk about “simple expressions”, while in the latter case we talk about “complex expressions”. For example, the following snippet specifies a template that matches all elements that end with an ‘s’:

```
<xsl:template
  match="*[substring(name(.), string-length(name(.))) = 's']">
  ...
</xsl:template>
```

3
When a template matches an XML document, the body of the template is applied. Typically this results in some output being produced.

Templates may have parameters attached to them. A parameter holds data that is extracted from the matched element (by means of an XPath expression), but it can also hold data external to the matched element since XPath allows navigation to the parent and sibling elements of a given element.

The `<apply-elements>` element inside a template is used to indicate that the XSLT processor should apply all templates on the element being processed and its child elements. One can restrict the set of child elements that should be matched by adding a “select” statement to the `<apply-elements>` element. In this case, only those templates that match the “select” XPath expression will be considered – other elements will not be processed at this stage even if there is a template that matches them. The XPath expression in the select may be simple (only concrete element names) or complex (uses functions and wildcards).

For further details about XSLT, the reader is referred to [6]

2.2. Choice of Metrics

An essential pre-requisite to train a machine learning model in order to predict a given phenomenon, is to identify a set of features that may serve as indicators of the phenomenon in question. In the context of mainstream programming languages (procedural and functional) many complexity measures have been studied, ranging from Lines of Code (LOC) counts, to logical complexity measures such as McCabe’s and Halstead’s ones. We could have simply adapted some of these metrics to fit the characteristics of XSLT and tested their predictive power. However, we adopted a more rigorous and exhaustive approach by inspecting the structure of XSLT and identifying 84 different metrics to describe XSL transformations. We then built and evaluated models to assess the impact of these metrics on code churn.

The focus was on metrics that are suitable for real-time analysis of XSL transformations and thus can be calculated in linear time. This allows us to use the models trained on the projects on current content being written by the developer. That data can be used at coding time to aid developer with suggestions and warnings when the solution design takes turns into high maintainability (and is as such likely to make a mistake). This is also the reason why no history information is used for training the models. The metrics are listed in table 1.

The metrics in the first column of this table are count metrics, that is, metrics calculated by counting the lines or elements (of different types) in an XSL transformation. The first five metrics in this column are generic (e.g. LOC, comments, counts of XML elements/attributes). The metrics number six and onwards correspond to counts of different types of XSLT elements. We considered all possible XSLT elements except for `<output>`, `<transform>`, `<decimal-format>`, `<preserve-space>`, `<strip-space>` and `<stylesheet>` elements. We did not count these elements as they are present at most once per stylesheet (or effectively combined into one). In addition, `<transform>` and `<stylesheet>`
elements can only be the root elements and thus cannot be used to discriminate between two stylesheets. Also, we did not consider elements that are specific to XSLT 2.0, because we found that these features are still not widely used in practice and we could not find enough data to evaluate them. Importantly, despite the fact that we focus on XSLT 1.0, our results are still largely applicable in the context of XSLT 2.0 due to backward compatibility between XSLT 2.0 and 1.0.

Additionally, we differentiated between XPath expressions that contain brackets or wildcards versus those that do not. Expressions with wildcards and/or brackets (called complex expressions) match a wide set of source structures, as opposed to expressions without these constructs, which only match very specific source structures. Expressions with wildcards and/or brackets are more generic and this genericity may have an effect on maintainability. The metrics defined

<table>
<thead>
<tr>
<th>Basic count metrics</th>
<th>Semantic metrics</th>
<th>Derived metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Lines of code (LOC)</td>
<td>47. # of test expressions containing wildcards</td>
<td>56. Total # of attributes and elements</td>
</tr>
<tr>
<td>2. # of XML nodes</td>
<td>48. # of select expressions containing wildcards</td>
<td>57. # of output elements and attributes (elements and attributes in target schema + &quot;element&quot; and &quot;attribute&quot; elements)</td>
</tr>
<tr>
<td>3. # of XML elements</td>
<td>49. # of match expressions containing wildcards</td>
<td>58. # of complex test expressions (total of complex test expressions)</td>
</tr>
<tr>
<td>4. # of XML attributes</td>
<td>50. # of test expressions containing brackets</td>
<td>59. # of complex select expressions (total of simple test expressions)</td>
</tr>
<tr>
<td>5. # of XML comments</td>
<td>51. # of select expressions containing brackets</td>
<td>60. # of complex match expressions (total of complex match expressions)</td>
</tr>
<tr>
<td>6. # of XML processing instructions</td>
<td>52. # of match expressions containing brackets</td>
<td>61. # of complex expressions (total of simple test, select and match expressions)</td>
</tr>
<tr>
<td>7. # of first level elements (direct children of root element)</td>
<td>53. # of test expressions without brackets or wildcards (simple test expressions)</td>
<td>62. # of simple expressions (total of simple test, select and match expressions)</td>
</tr>
<tr>
<td>8. # of output literals (elements and attributes in target schema)</td>
<td>54. # of select expressions without brackets or wildcards (simple select expressions)</td>
<td>63. # of globals (total of global parameters and variables)</td>
</tr>
<tr>
<td>9. # of &quot;template&quot; elements</td>
<td>55. # of match expressions without brackets or wildcards (simple match expressions)</td>
<td>64. # of externals (total of &quot;include&quot; and &quot;import&quot; elements)</td>
</tr>
<tr>
<td>10. # of elements and attributes inside templates</td>
<td>11. # of &quot;include&quot; elements</td>
<td>65. Average # of elements and attributes inside templates</td>
</tr>
<tr>
<td>11. # of &quot;include&quot; elements</td>
<td>12. # of &quot;import&quot; elements</td>
<td>66. Average # of elements and attributes inside parameters</td>
</tr>
<tr>
<td>12. # of &quot;import&quot; elements</td>
<td>13. # of &quot;namespace-alias&quot; elements</td>
<td>67. Average # of elements and attributes inside variables</td>
</tr>
<tr>
<td>13. # of &quot;namespace-alias&quot; elements</td>
<td>14. # of &quot;key&quot; elements</td>
<td>68. Average # of &quot;when&quot; elements inside &quot;choose&quot; element</td>
</tr>
<tr>
<td>14. # of &quot;key&quot; elements</td>
<td>15. # of &quot;fallback&quot; elements</td>
<td>69. Ratio of complex expressions to all expressions</td>
</tr>
<tr>
<td>15. # of &quot;fallback&quot; elements</td>
<td>16. # of &quot;message&quot; elements</td>
<td>70. Ratio of globals to all elements</td>
</tr>
<tr>
<td>16. # of &quot;message&quot; elements</td>
<td>17. # of elements and attributes inside messages</td>
<td>71. Ratio of first level elements to all elements</td>
</tr>
<tr>
<td>17. # of elements and attributes inside messages</td>
<td>18. # of &quot;attribute-set&quot; elements</td>
<td>72. Ratio of templates to all elements</td>
</tr>
<tr>
<td>18. # of &quot;attribute-set&quot; elements</td>
<td>19. # of &quot;element&quot; elements</td>
<td>73. Ratio of first level elements to all elements</td>
</tr>
<tr>
<td>19. # of &quot;element&quot; elements</td>
<td>20. # of &quot;attribute&quot; elements</td>
<td>74. Ratio of output literals to all output literals</td>
</tr>
<tr>
<td>20. # of &quot;attribute&quot; elements</td>
<td>21. # of &quot;comment&quot; elements</td>
<td>75. Ratio of inline expressions to inline expressions</td>
</tr>
<tr>
<td>21. # of &quot;comment&quot; elements</td>
<td>22. # of &quot;processing-instruction&quot; elements</td>
<td>76. # of operands</td>
</tr>
<tr>
<td>22. # of &quot;processing-instruction&quot; elements</td>
<td>23. # of &quot;text&quot; elements</td>
<td>77. # of unique operands</td>
</tr>
<tr>
<td>23. # of &quot;text&quot; elements</td>
<td>24. # of &quot;number&quot; elements</td>
<td>78. # of operators</td>
</tr>
<tr>
<td>24. # of &quot;number&quot; elements</td>
<td>25. # of &quot;copy&quot; elements</td>
<td>79. # of unique operators</td>
</tr>
<tr>
<td>25. # of &quot;copy&quot; elements</td>
<td>26. # of &quot;copy-of&quot; elements</td>
<td>80. Halstead vocabulary size</td>
</tr>
<tr>
<td>26. # of &quot;copy-of&quot; elements</td>
<td>27. # of &quot;value-of&quot; elements</td>
<td>81. Halstead program length</td>
</tr>
<tr>
<td>27. # of &quot;value-of&quot; elements</td>
<td>28. # of &quot;call-template&quot; elements</td>
<td>82. Halstead difficulty level</td>
</tr>
<tr>
<td>28. # of &quot;call-template&quot; elements</td>
<td>29. # of &quot;apply-templates&quot; elements</td>
<td>83. Halstead program volume</td>
</tr>
<tr>
<td>29. # of &quot;apply-templates&quot; elements</td>
<td>30. # of &quot;with-param&quot; elements</td>
<td>84. Halstead effort to implement</td>
</tr>
<tr>
<td>30. # of &quot;with-param&quot; elements</td>
<td>31. # of inline expressions</td>
<td>85. Halstead program complexity</td>
</tr>
<tr>
<td>31. # of inline expressions</td>
<td>32. # of &quot;param&quot; elements</td>
<td>86. Halstead code volume</td>
</tr>
<tr>
<td>32. # of &quot;param&quot; elements</td>
<td>33. # of elements and attributes inside parameters</td>
<td>87. Halstead cyclomatic complexity</td>
</tr>
<tr>
<td>33. # of elements and attributes inside parameters</td>
<td>34. # of &quot;variable&quot; elements</td>
<td>88. Halstead cohesion of methods</td>
</tr>
<tr>
<td>34. # of &quot;variable&quot; elements</td>
<td>35. # of elements and attributes inside variables</td>
<td>89. Halstead coupling of classes</td>
</tr>
<tr>
<td>35. # of elements and attributes inside variables</td>
<td>36. # of global parameters (i.e. defined outside any template)</td>
<td>90. Halstead complexity of classes</td>
</tr>
<tr>
<td>36. # of global parameters (i.e. defined outside any template)</td>
<td>37. # of global variables (i.e. defined outside any template)</td>
<td>91. Halstead cohesion of attributes</td>
</tr>
<tr>
<td>37. # of global variables (i.e. defined outside any template)</td>
<td>38. # of &quot;for-each&quot; elements</td>
<td>92. Halstead coupling of attributes</td>
</tr>
<tr>
<td>38. # of &quot;for-each&quot; elements</td>
<td>39. # of &quot;sort&quot; elements</td>
<td>93. Halstead complexity of attributes</td>
</tr>
<tr>
<td>39. # of &quot;sort&quot; elements</td>
<td>40. # of &quot;choose&quot; elements</td>
<td>94. Halstead cohesion of methods (i.e. defined outside any template)</td>
</tr>
</tbody>
</table>
by counting expressions with wildcards and brackets are listed under column semantic metrics since they play with the semantic of expressions.

Table 2: XSLT structure constructs and their corresponding object-oriented constructs.

<table>
<thead>
<tr>
<th>XSLT structure</th>
<th>Object-oriented structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;include&gt;</code> and <code>&lt;import&gt;</code></td>
<td>Import and include directives</td>
</tr>
<tr>
<td><code>&lt;template&gt;</code></td>
<td>Method or function</td>
</tr>
<tr>
<td><code>&lt;variable&gt;</code> in template</td>
<td>Local variable</td>
</tr>
<tr>
<td><code>&lt;variable&gt;</code> outside templates</td>
<td>Global, static or class member</td>
</tr>
<tr>
<td>Direct child element <code>&lt;param&gt;</code> of <code>&lt;template&gt;</code></td>
<td>Method or function parameter</td>
</tr>
<tr>
<td><code>&lt;param&gt;</code> element outside templates</td>
<td>Class constructor parameter or application input argument</td>
</tr>
<tr>
<td>Other <code>&lt;param&gt;</code> element</td>
<td>Local variable</td>
</tr>
<tr>
<td><code>&lt;call-template&gt;</code></td>
<td>Method or function call</td>
</tr>
<tr>
<td><code>&lt;with-param&gt;</code></td>
<td>Method or function argument</td>
</tr>
<tr>
<td><code>&lt;for-each&gt;</code></td>
<td>Loop</td>
</tr>
<tr>
<td><code>&lt;choose&gt;</code></td>
<td>Switch clause</td>
</tr>
<tr>
<td><code>&lt;if&gt;</code></td>
<td>If clause</td>
</tr>
<tr>
<td><code>&lt;when&gt;</code></td>
<td>Switch case clause</td>
</tr>
<tr>
<td><code>&lt;otherwise&gt;</code></td>
<td>Switch default clause</td>
</tr>
<tr>
<td>Attribute “select”</td>
<td>Assignment</td>
</tr>
<tr>
<td><code>&lt;message&gt;</code></td>
<td>Writing to log</td>
</tr>
</tbody>
</table>

The metrics shown in the third column of table 1 are derived metrics, meaning that they are defined in terms of the metrics in the first column. Here, we considered sums of counts of elements or attributes with similar purposes, and ratios of certain common types of elements relative to the total number of elements in the transformation. These metrics were mostly inspired from similar metrics defined for object-oriented programming languages [7]. Even though XSLT structures cannot be directly mapped to object-oriented structures, we identified some parallels between these structures (see table 2) that we exploited in order to adapt OO metrics to XSLT.

The last few metrics in table 1 are inspired by Halstead’s code metrics [8]. In order to make these metrics applicable to XSLT, variables, parameters and constants were considered to be operands, and all XSLT elements other than generators of constant values, were considered to be operators.

We also considered the possibility of adapting metrics based on control-flow graphs (e.g. cyclomatic complexity, knots, template fan-out and fan-in). However, it turns out that these metrics cannot be effectively applied to XSLT. The key obstacle is the `<apply-templates>` element in XSLT, which tells that a template must be matched and called for all source document nodes that match the XPath expression specified in the “select” attribute. This construct makes the use of control-flow metrics impractical, because sophisticated program analysis techniques would be needed to determine how many templates are called when processing an `<apply-templates>` element. In fact, this problem is likely to be intractable, since it has been proved that the satisfiability of XPath expressions is intractable or even undecidable depending on the assumptions made [9].
3. Empirical evaluation

Supervised learning techniques were used to build models to predict code churn. The training and evaluation of these techniques was done by following a five-step knowledge discovery in databases (KDD) process: 1) data understanding; 2) data pre-processing; 3) application of data mining algorithms; 4) post-processing; 5) analysis of results.

3.1. Understanding the data

In order to evaluate our hypotheses, we built a set of data collection and analysis tools. These tools imported files and their change history from 15 open-source software project repositories (accessed using SVN, CVS and Git) chosen to cover two large classes of usage scenarios of XSLT. The chosen projects and their characteristics are the following.

1. Business applications containing document transformations. This category of applications was represented by four WSO2 projects (http://ws2.org/), namely commons, esb, wsf and wsas, totaling 374 XSLT files with 59939 lines of code. Two years of revision info from WSO2 projects was collected.

2. Web and desktop applications containing presentation-layer transformations. This category was by 10 projects listed in table 3. These projects totaled 1860 XSLT files with 681476 lines of code.

<table>
<thead>
<tr>
<th>Project</th>
<th>URL</th>
<th>Years</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocBook</td>
<td><a href="http://docbook.sourceforge.net/">http://docbook.sourceforge.net/</a></td>
<td>9</td>
</tr>
<tr>
<td>DocBook2X</td>
<td><a href="http://docbook2x.sourceforge.net/">http://docbook2x.sourceforge.net/</a></td>
<td>8</td>
</tr>
<tr>
<td>DIA</td>
<td><a href="http://www.gnome.org/projects/dia/">http://www.gnome.org/projects/dia/</a></td>
<td>12</td>
</tr>
<tr>
<td>FeedParser</td>
<td><a href="http://www.feedparser.org/">http://www.feedparser.org/</a></td>
<td>5</td>
</tr>
<tr>
<td>eXist</td>
<td><a href="http://exist.sourceforge.net/">http://exist.sourceforge.net/</a></td>
<td>7</td>
</tr>
<tr>
<td>GnuCash</td>
<td><a href="http://www.gnucash.org/">http://www.gnucash.org/</a></td>
<td>12</td>
</tr>
<tr>
<td>Groovy</td>
<td><a href="http://groovy.codehaus.org/">http://groovy.codehaus.org/</a></td>
<td>6</td>
</tr>
<tr>
<td>TEI</td>
<td><a href="http://tei.sourceforge.net/">http://tei.sourceforge.net/</a></td>
<td>5</td>
</tr>
<tr>
<td>Valgrind</td>
<td><a href="http://valgrind.org/">http://valgrind.org/</a></td>
<td>7</td>
</tr>
</tbody>
</table>

Even though the number of XSLT files in web and desktop applications was six times higher than the number of XSLT files in business applications, business applications had more revisions per file. The total number of revisions was almost equal between business and web applications.

The mean size of XSLT file revision was 262 LOC. The smallest file had 3 LOC, the largest 7391 LOC, the mode was 13 LOC and the median 116 LOC. The histogram of distribution of file sizes is given in figure 1.

In addition to calculating metrics for XSLT files, for each file revision, general information about the revision was gathered. The information gathered includes the committer’s name and commit date. Number of LOC added, number of LOC removed and number of LOC modified were calculated using the GNU diff command.
utility\textsuperscript{9}. Importantly, all these values were calculated using the same technique. These values were used to calculate code churn for each revision. Mean churn per revision is 17 LOC, minimum 2 LOC, maximum 1122 LOC, mode 2 LOC and median 6 LOC.

Before building code churn prediction models based on combinations of measures, we first checked if individual metrics could serve already as accurate predictors of churn. For this purpose, we extracted data for XSLT file revisions, and for each revision, we calculated each of the metrics as well as the code churn. We then calculated the correlation between each metric and the observed code churn. It was found that none of the individual XSLT metrics had a strong correlation with code churn, thus justifying the construction of models based on combinations of variables. Low correlation of single features was expected as it is less likely for a single feature to correlate significantly than for a combination of features to correlate significantly with code churn. In fact, the more different features influence code churn, the lower would the correlation between a single feature and code churn be. This is in line with findings from other studies like [10].

Among the results of correlation analysis, however, some metrics did stand out from the others. The metrics with highest correlations are shown in table 4; other metrics had a correlation of less than 0.07. Most importantly, the number of comments in XSLT had a remarkably higher positive correlation than any other metric.

The correlations between features themselves was not studied as it would not have added to the research. The correlations between features have no effect on the training algorithms as they do their own feature selection and can

\textsuperscript{9}http://www.gnu.org/software/diffutils/
benefit from the differences between highly correlated features (e.g. algorithms tended to make use of both the number of nodes and LOC, which have Pearson correlation higher than 80). That is, our primary goal was not to obtain explainable models, which benefit from elimination of dependent features, but rather to obtain models with the best prediction accuracy instead.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of comments</td>
<td>0.2868</td>
</tr>
<tr>
<td>Ratio of comments to all nodes</td>
<td>0.0885</td>
</tr>
<tr>
<td>Ratio of first level elements to all elements</td>
<td>-0.0853</td>
</tr>
</tbody>
</table>

3.2. Data Pre-processing

In order to identify causes of high code churn, code churn values were discretized into three classes: “small churn” (1–4 LOC), “medium churn” (5–16 LOC) and “large churn” (over 16 LOC). The most important ranges are “small churn”, which characterizes smaller bug fixes, and “large churn”, which means either addition of a new large template or a major rewrite of the transformation. The distribution of LOC churn is shown in figure 2. Interestingly, code churn was more than 40 LOC in only about 10% of cases and code churn up to 10 LOC was present in 65% of the cases.

The choice of LOC churn ranges was based on an analysis of the distribution of LOC churn across all revisions of all systems. The chosen ranges were:

- High churn (> 16 LOC): This corresponds to the top-quartile of LOC churn size per revision. It also corresponds to changes greater than 5.5% of the average size of a file across the entire dataset.
- Low churn (1–4 LOC): This corresponds to the bottom 40% of the distribution of LOC churn size. It also corresponds to changes less than 1.5% of the average file size. Moreover, the range 1–4 LOC corresponds to simple changes like localized renaming, addition or removal of an element, attribute or any other node.
- Medium churn: 5–16 LOC.

Microsoft SQL Server Analysis Services\(^{10}\) was used to train and test different models on the dataset. The source dataset was split randomly into training set (70% of dataset) and testing data (30% of dataset). The training and testing set were the same for all the models. Additionally, models were tested with data from [11]. Splits 50:50 and 90:10 were also tried. Split 50:50 and split

\(^{10}\)Microsoft SQL Server Analysis Services is Microsoft’s offering for business analysis bundled with Microsoft SQL Server. It offers data mining, multi dimensional and key performance indicators functionality that can be used with various database and reporting engines. [http://www.microsoft.com/sqlserver/2008/en/us/analysis-services.aspx](http://www.microsoft.com/sqlserver/2008/en/us/analysis-services.aspx)
70:30 resulted in similar models, however split 90:10 resulted in strongly over-fit models. Due to these characteristics, 10-fold cross-validation would not be suitable (maximum of three folds is viable). In the process of trying different splits, three different 70:30 splits were made and we found that the models obtained from these three splits had similar performance (precision and recall stayed within ±0.03 error range).

3.3. Application of algorithms

Models were trained using the following algorithms, as implemented in SQL Server Analysis Server:

1. Decision Trees – A hybrid algorithm, which uses regression, classification and association methods for creating a decision tree.
2. Neural Networks – A back-propagated delta rule network with three layers.
3. Logistic Regression – A variation of the Neural Networks algorithm without the hidden layer.
4. Clustering – using the k-means method. We also considered clustering using expectation maximization, but k-means gave better results.

Additionally, Linear Regression with second order interactions ($^2$) and Linear Regression with stepwise randomised fitting by AIC and Kendall correlation (gen) were used as implemented in R Statistics Suite\(^\text{11}\).

The models’ parameters were adjusted experimentally to increase the fit of the models to the testing data. The aim of adjusting fitness was to increase models ability to accurately predict churn classified as “high churn”.

\(^{11}\text{http://www.r-project.org/}\)
3.4. Performance measures

To evaluate the performance of the models we use the notions of precision, recall and lift. To understand these notions, we first observe that each model under consideration is a function that takes as input a revision of an XSLT stylesheet, and predicts the category of this stylesheet (low churn, medium churn or high churn) in the next revision of the stylesheet. The precision of a model for a given category \( C \) (e.g. \( C = \) low churn) is the number of stylesheet revisions correctly predicted by the model as being in \( C \), divided by the total number of stylesheet revisions that the model predicted as being in \( C \) (the denominator includes incorrect predictions).

Meanwhile, the recall of a model for a category \( C \) is the percentage of stylesheet revisions belonging to \( C \) that the model correctly predicted as being in \( C \). In other words, the recall for category \( C \) is the number of stylesheet revisions that the model predicted as being in \( C \), divided by all stylesheet revisions that indeed belong to \( C \).

The lift allows us to compare the performance of a model with respect to the performance of a random model (i.e. a model based on random guess). To calculate the lift for a given category, the set of stylesheet revisions (the population) is sorted from those that the model predicted as being “most likely” to be in this category, to those that the model predicted as “less likely” to be in this category. Stylesheet revisions can be sorted in this way because every model returns a number indicating the likelihood that a stylesheet revision falls under a given category. The top-\( X\% \) of the population is defined as the \( X\% \) of stylesheets that the model predicted as having the highest likelihood of falling under the category in question. The lift at \( X\% \) is the ratio of the precision of the model for the top-\( X\% \) of the population, divided by the precision of the random model for the top-\( X\% \) of the population. For example, a lift of 1.5 at 10\% for category “low churn” means that the model performs 50\% better than random when we consider the 10\% of stylesheets that the model classified as being in category “low churn”. Lift at 10\% population (how many times more accurate is the model compared to random guess at choosing 10\% of the best hits) for each model is shown in the tables. A lift chart is a plot of the lift values for different percentages of the population. For convenience, a lift chart displays the performance of an ideal model, that is, a model that always classifies all stylesheets correctly.

As a global measure of the quality of a model, we use the notion of overall precision (Correct\%): The number of correct predictions made by the model (regardless of the category), divided by the total number of predictions (i.e. the total number of stylesheet revisions).

4. Results

This section presents the results of the experimental evaluation from three perspectives: (i) the performance of the models using the performance measures defined above; (ii) the influencers of code churn uncovered by the models; and
(iii) the effect of the type of project on the performance of the models and the influencers they uncover.

4.1. Accuracy of predictions

In order to verify whether the features are sufficient for building useful models, we built six models using different algorithms and calculated the values of their performance metrics. Each model was designed to classify a file revision into one of three LOC churn ranges described before.

4.1.1. Classification

We start by studying the models' ability to correctly classify transformations into the three classes described earlier. The models' ability to classify transformations is useful when trying to evaluate a transformation against code churn in the next version.

The performance measures for each model are shown in Table 5. The table gives the precision, recall and lift (at 10% population) for each LOC churn range. Additionally, the table displays the precision and recall of each model when classifying transformations in the category \(>4\) LOC churn, that is, the category that combines the medium and the high churn categories. Finally, overall precision of each model (i.e., percentage of correct predictions in the three churn ranges) is also shown in the table. The best values attained by each model are highlighted in bold.

The best overall precision (46.95%) was achieved using the Clustering algorithm. Clustering also has the highest precision for classifying transformations in the category \(>4\) LOC churn and “high churn”. Its high recall for “low churn” makes it a useful model for identifying transformations with little churn. But despite this positive characteristic, closer inspection shows that the Clustering algorithm is not the preferable model overall.
In contrast, Linear Regression models had the lowest overall precision, but they scored highest at identifying the middle churn range (5 – 16 LOC) files. They also offer the best recall values for ranges other than the “low churn” range and the best lift for the “high churn” range. The stepwise fitted model (gen) does, however, highlight the stronger points of Linear Regression models, having about 10% higher recall values for churn ranges 5 – 16 and > 4.

This suggests that a combination of Clustering, Neural Networks (or Logistic Regression) and Linear Regression models could be used to classify transformations with higher accuracy than any of the models alone. For example, Linear Regression (gen) model can be used to verify classification 0 – 4 made by Clustering model (there is less than 7% probability for Linear Regression (gen) model to misclassify a transformation in > 4 range).

4.1.2. Identification

In addition to giving a classification, the models also give their confidence in the classification. This allows the models to be used on a set of transformations to identify those most likely to belong to certain churn range. This can be used to select or exclude transformations from review before release to reduce possible problems during maintenance phase. In this section, the models ability to identify the top likely transformations for each class, is studied.

As seen in the lift chart in figure 3, Decision Trees are better at identifying the top 10% transformations with low churn (i.e. those transformations that are most likely to fall in the “low churn” category). But beyond this point, Decision Trees fall behind Neural Networks. In other words, Decision Trees would be best, if given a repository of transformations, we wanted to select up to 10% of transformations with the least code churn in the next revision. This can be

Figure 3: Lift of models at predicting not “low churn”.

As seen in the lift chart in figure 3, Decision Trees are better at identifying the top 10% transformations with low churn (i.e. those transformations that are most likely to fall in the “low churn” category). But beyond this point, Decision Trees fall behind Neural Networks. In other words, Decision Trees would be best, if given a repository of transformations, we wanted to select up to 10% of transformations with the least code churn in the next revision. This can be
useful when choosing model transformations that new developers can learn from or the style of which older developers can follow in new transformations.

In terms of lift, Linear Regression models generally fall behind other models. However, for identifying the top revisions with “high churn”, they are the best. All models give relatively good results at higher population ranges. Differences are notable only in the first 30%, where Neural Networks and Decision Trees are far superior to Logistic Regression and Clustering models. The lift chart also shows that the top confidence level of the Clustering model gives poor results when identifying churn higher than 4 LOC (top 5% lift is less than 1, making it worse than random guess), however, it greatly improves in the larger ranges. In other words, the Clustering model is not suitable for identifying the very best or the very worst transformations.

In summary, Decision Trees or Neural Networks (if more than 10% is targeted) should be preferred for identifying the top best and Linear Regression models should be preferred for identifying the top worst transformations. Clustering model can help most at identifying transformations most likely to belong to “medium churn” range.

4.1.3. Conclusion

Predictions with precision higher than 46% (and over 65% precision for higher LOC churn ranges) confirm the hypothesis that simple metrics of XSL transformations can be used to predict high LOC churn in the next revision of a transformation. The models generally also have good ability to identify the top transformations in different churn classes.

4.2. Influencers of churn

To get better insight into what influences code churn, we looked for features with the most influence to the predictions. The main influencers can be used alone to make predictions on LOC churn with reduced accuracy.

The Decision Trees model identified six major influencers:

1. Ratio of first level elements
2. Average template size
3. Ratio of comments
4. Ratio of inline expressions
5. Number of call-templates
6. Number of simple expressions

Figure 4 shows the strength of these influences as the width of the connections between attributes. As can be seen from figure 5, the first decision point is number of simple expressions. It can be seen that higher numbers of simple expressions encourages higher code churn (the percentage of “low churn” transformations is highest in classes with low number (<22) of simple expressions and the percentage of “high churn” transformations is highest in class with high number (≥176) of simple expressions). It is interesting to note that the complementing feature “the number of complex expressions” was not identified as an influencer.
High number of call-templates has an interesting effect of encouraging “medium churn” in the next revision of transformation. As “medium churn” is normally caused by creating new templates, high number of call-templates can be a sign of well designed XSL files. Decision trees also identified that smaller ratio of comments per nodes results in higher future churn, implying that comments in XSLT play important role for maintainers. The model suggests that files having at least 9% of XSLT nodes as comment nodes have reduced the probability of “large churn” to less than 5%.

The Clustering algorithm determined that the following attributes can be used to identify “high churn”:

1. Long choose constructs (average number of ‘when’ elements inside ‘choose’ between 13.5 and 42.0)
2. Large parameters (average parameter size between 5.7 and 18.5)
3. High number of text nodes ($\geq 48$)
4. High number of simple tests ($\geq 46$)
5. High number of ‘when’ elements ($\geq 60$)
6. Lack of inline expressions
7. Lack of parameters

These discriminators were also common for Neural Networks and Logistic Regression models. Additionally, High number of first level elements was identified as indicator for high churn rates. No model identified any Halstead metrics as predictor of future code churn. “Number of Fallbacks” was also considered of no or low influence by all of the models. All other features had high influence for some models or moderate influence on many models.

4.3. Impact of Project Type

To test, whether the relations between the features and code churn are similar in different types of project (i.e. business or desktop), we created models
Figure 5: First three layers of Decision Tree model.

for different project types separately. The performance metrics of these models were then compared to the corresponding general models.

4.3.1. Impact on Churn

The study of the histograms of different types of projects (figure 6) shows that business projects dataset seems to have less churn than web and desktop projects. The average churn for business type projects was 16 LOC, for presentation type projects 19 LOC. The difference was mostly in the first two churn ranges (4% in 0-4 LOC and 3% in 5–16 LOC), distribution in “high churn” range differed less than a percent. Thus, the type of the transformation has only a minor effect on code churn.

4.3.2. Impact on Classification

In order to determine, if the general model can be used for both scenarios, the performance of general models was calculated for both business-oriented and presentation-oriented projects separately. Evaluating the models separately for business and presentation projects showed that the general models are better (ratio of correct predictions differed 3-5%) at making predictions on business type projects, which can be expected from the more skewed distribution of business data sets. The comparison of models can be seen in table 6. Improvements compared to the general model are shown in green, declines in red and differences of at least 10% are in bold.
Figure 6: Histogram of code churn in different types of projects.

Table 6: Gain or loss of performance for specific models compared to general models.

<table>
<thead>
<tr>
<th></th>
<th>Decision Trees - business</th>
<th>Decision Trees - presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>Prec</td>
<td>Rec</td>
</tr>
<tr>
<td>0 - 4</td>
<td>0.0159</td>
<td>0.0216</td>
</tr>
<tr>
<td>5 - 16</td>
<td>0.0455</td>
<td>0.0354</td>
</tr>
<tr>
<td>&gt;16</td>
<td>0.1858</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Prec</td>
<td>Rec</td>
</tr>
<tr>
<td>0 - 4</td>
<td>0.042</td>
<td>-0.2888</td>
</tr>
<tr>
<td>5 - 16</td>
<td>0.0386</td>
<td>0.3366</td>
</tr>
<tr>
<td>&gt;16</td>
<td>-0.066</td>
<td>0.0892</td>
</tr>
<tr>
<td></td>
<td>Prec</td>
<td>Rec</td>
</tr>
<tr>
<td>0 - 4</td>
<td>-0.0095</td>
<td>0.0591</td>
</tr>
<tr>
<td>5 - 16</td>
<td>0.0355</td>
<td>-0.0651</td>
</tr>
<tr>
<td>&gt;16</td>
<td>-0.0108</td>
<td>0.0332</td>
</tr>
<tr>
<td></td>
<td>Prec</td>
<td>Rec</td>
</tr>
<tr>
<td>0 - 4</td>
<td>0.0273</td>
<td>-0.0767</td>
</tr>
<tr>
<td>Correct%</td>
<td>2.01%</td>
<td>0.70%</td>
</tr>
</tbody>
</table>

The comparison shows that predictions of models trained specifically on and for business-oriented projects were correct more often than the predictions made by the general model on business-oriented projects. This might be due to the fact that revisions in business-oriented projects more frequently fall under the same churn range (0-4 LOC) than in web/desktop projects. It is also possible that transformations in business-oriented projects are more homogeneous due to possible use of mapping tools, making typical cases stronger – to the benefit of specialized prediction models. Still, the differences in prediction accuracy between the business-oriented models and the desktop-oriented models are modest (less than 1% of relative improvement).

Decision Trees and Clustering models for presentation-oriented projects were more conservative in ranges other than “low churn”. Models trained on web/desktop
projects only, were slightly worse than the general ones more often than better
than the general ones. However, these differences are not sufficient to make any
definite conclusions. The only model to constantly get worse in the specialized
case was the Neural Networks model. This might be due to the relatively high
volume of training data required by Neural Networks in order to make reliable
predictions. The performance from the Clustering model also displayed signs of
decrease, however, the lift for identification of “low churn” for business-oriented
projects versus “medium/high churn” improved greatly (as did the lift for iden-
tifying “medium/high churn” for presentation-oriented projects).

The decreases in recall and lift in specialised models gives a good indicator
that general models can cope better with larger variety. In any case, the lack
of consistent and significant benefit of training separate models, makes using
general models the preferred choice for most scenarios. Thus we conclude that a
general model, independent of the type of system, can be used to make predictions
of XSLT code churn with an accuracy comparable to that of models built for
specific types of systems.

4.3.3. Impact on Identification

Study of lift showed that lift charts for different project types differ greatly.
However, it does not necessarily mean that project type influences code churn.
For that purpose, a more detailed look into lift was taken.

![Lift Chart](image)

Figure 7: Lift for identifying churn higher than 4 LOC in business type projects.

The lift chart for identifying churn higher than 4 LOC shows that specialised
versions of Decision Trees and Neural Networks models offer significant improve-
ment in smaller population ranges (shown on figure 7, suffix “b” means special-
ized model in business projects). The general Decision Trees model is, in fact,
the worst at all ranges of population. However, Clustering model did not offer
almost any visible improvement at all, being on par with specialised versions of Neural Networks and Decision Trees models. The lift chart for identifying churn higher than 4 LOC

![Lift Chart](image)

Figure 8: Lift at identifying “low churn” in presentation type projects.

The lift of identifying “low churn” in presentation type projects reaches almost 2 at 10% population and is equal to ideal model on 5% population (meaning it has 100% precision at identifying 5% of the “low churn” transformations). This shows that

For presentation projects, specialised Clustering Model achieved the best lift at identifying top transformations with churn higher than 4 LOC. The Decision Trees models (both specific and general) did, however achieve the same or even higher lift in population ranges of 15% or higher. The rest of the models did not reach such high lift. The lift for identifying “low churn” was very high (over 1.7 at 10%, where maximum possible lift is 2.3) at small population ranges for all models (except for presentation projects specific Logistic Regression model). The steep decline and high lift can be seen in figure 8.

It is important to note that while lift improved for some models in some population ranges, it also decreased for others. Neither did any model improve reliably for all churn classes. Thus we conclude that a general model, independent of project type can be used to identify top transformations belonging to specified code churn classes.

### 4.4. Limitations of the study

A limitation common to using data mining is the choice of features, which might not include all possible influencers of the predicted measure. Our systematic selection of metrics ensures that we have covered at least the space of count metrics, but other more complex metrics might turn out to be better
predictors of code churn. Another important limitation of the study relates to the training and evaluation data sets, which might not accurately represent all types of projects. Especially, commercial projects can have substantial differences in their code base due to different development and quality management practices. Thus, the models trained with this study might not give the same results in those settings.

Another limitation of the study is that it is restricted to metrics that can be computed directly from the contents of an XSLT file. This choice was made deliberately in order to give a focus to the study. Still, during the course of the study we experimented with one organizational metric, namely the next committer’s average LOC churn. A committer’s average LOC churn is the average LOC churn per file revision that can be attributed to this committer. For a given file revision, the next committer’s average LOC churn is the average LOC churn of the committer of the next revision of the file in question. We gave this parameter as an optional (nullable) input to each model alongside the parameters listed in Table 1. However, it turned out that this parameter had a low significance. We also considered including the next committer’s maximum and minimum LOC churn among the inputs. But we found that the maximum LOC churn was affected by outliers – for example some developers occasionally import files from other repositories or projects, while the minimum LOC churn per developer is usually close to 1 – thus, it has a low variance. In any case, there is room for extending the study in order to incorporate further organizational metrics.

4.5. Churn prediction service

The models obtained from this study have been made available through a public web service. The web service allows users to upload their XSLT file for evaluation. The 86 metrics considered in this study are extracted from the XSLT file and these values are fed into the general models. The service provides as output the stylesheet’s metrics and the predicted churn range.

In addition to predicting the future churn for a given XSLT file, the service allows users to play out different scenarios for improving the stylesheet. For example, the service allows the user to see how changes in the characteristics of the stylesheet are likely to influence its maintainability. To this end, the user is given an opportunity to tweak XSLT characteristics after the stylesheet has been processed (as shown in figure 9).

Figure 9: Screenshot showing scenario testing from a service for predicting code churn.

There are at least two use cases for the churn prediction service. First, a developer can use it to choose between multiple implementations of a stylesheet. For example, if the developer is considering two approaches to implement a
feature, one using `<template>` and `<call-template>` versus one using `<for-each>`, he can load the original file into the prediction service and alter the values of the relevant features (e.g. “# of template”, “# of call-template”, “# of attributes”) in order to examine the impact of the alternative design choices on future churn. Second, project managers can use the churn prediction service in order to identify stylesheets that are highly prone to future churn.

5. Related Work

Our work falls under the umbrella of a body of research aiming at mining software repositories in order to derive predictive models of software properties (e.g. predicting and locating defects, predicting changes, etc.). For example, it has been shown that high relative code churn tends to be a good predictor of system defect density [5]. Despite the rich body of work in this field [12], we are not aware of any technique that deals specifically with XSLT. In fact, almost none of the techniques developed in this field deal with XML and related languages.

The techniques we use are reminiscent of those used by Ratzinger, J. et Al. [13] to mine software repositories in order to identify future refactoring of Java code. They predict the number of future refactorings of files in java projects in short time-frames. However, they do not try to identify maintenance effort of a single refactoring, which is the case in this paper.

There is a large body of research related to mapping-driven transformations, meaning transformations derived from a mapping between the elements in the source and the target schemas [14]. These mappings can be derived using automatic schema matching techniques [15] and may be visualized and edited by developers using graphical schema mapping tools. Graphical schema mapping tools are incorporated in enterprise application development platforms such as Microsoft BizTalk [16]. While these approaches enhance developer productivity, they are not designed to achieve change-resilience of the resulting XSL transformations. In fact, the idea of these tools is that the XSL transformations are re-generated whenever a change is made to the mapping. This approach is not suitable for all applications as evidenced by the considerable number of manually developed XSL transformations found in commercial and open-source projects. In this respect, the guidelines studied in this paper are complementary to this body of work.

6. Conclusion and Future Work

6.1. Main Findings

The study reported in this paper helps to better understand the factors affecting the maintainability of XML transformations. The study confirmed that the maintenance effort measured by code churn required for updating the service can be predicted from simple characteristics of XSLT. Moreover, the
identification of transformations that are likely to undergo high churn is very accurate with low margin of error.

The study showed that even though the nature of the transformation seems to affect the maintainability effort, there is no need for separate models to be trained for predicting churn ranges for different types of transformations. In fact, models trained for both cases, performed better or similar to the ones trained for a specific type of transformation.

The churn prediction service built as part of the study can help developers to identify problematic XSL transformations that may warrant refactoring in order to reduce maintenance efforts. In future, we intend to extend this service in order to improve the diagnostics it provides and to improve its usability by integrating it into mainstream development environments.

6.2. Guidelines

Based on the study, following guidelines can be used to reduce code churn for the next revision of a XSLT file:

1. Complex expressions (which were found not to influence code churn) should be used instead of simple expressions (which were associated with high churn).
2. Templates with “match” expressions (and if needed, “mode” attribute) should be preferred over <choose> and <when> constructs as high number of <when> elements and long <choose> constructs are associated with high churn.
3. Reoccurring constants should be stored as parameters or variables and used through these if possible. The study showed that high number of text nodes characterise high churn while use of inline expressions and parameters encourages lower churn.
4. Parameter definitions should be kept short (less than 6 LOC).
5. XSLT code should be commented. At least 9% of nodes being comments was identified a good ratio as it reduced high churn probability to mere 5%.

6.3. Future Work

This study can be extended by considering more complex metrics. In mainstream programming languages, metrics based on control-flow graphs, such as cyclomatic complexity, have shown to be highly correlated with maintainability [7]. However, as discussed in Section 2, it is not straightforward to adapt metrics based on control-flow graphs to XSLT. Other metrics worth considering are organizational and project metrics (e.g. size and experience of the development team, age of the project), which could turn out to be complementary to the code metrics studied in this paper.

Another avenue for future work is to design automated refactoring techniques in order to improve the maintainability of XSL transformations. To this end, we plan to identify common types of changes in XSL transformations and develop techniques to determine which change types can be applied to a given transformation in order to improve its maintainability.
7. Acknowledgements

This research was partly conducted during a visit of the first author to the Software Engineering Group at University of Zurich, funded by the ESF DoRa 6 Program. We thank Harald Gall and the members of his group for their valuable advice.

References

[1] Xsl transformations (xslt).
   URL http://www.w3.org/TR/xslt


