Quality Assessment based on Attribute Series of Software Evolution

Jacek Ratzinger
Vienna University of Technology
Institute of Information Systems
A-1040 Vienna, Austria
ratzinger@infosys.tuwien.ac.at

Harald Gall, Martin Pinzger
University of Zurich
Department of Informatics
CH-8050 Zurich, Switzerland
{gall, pinzger}@ifi.unizh.ch

Abstract

Defect density and defect prediction are essential for efficient resource allocation in software evolution. In an empirical study we applied data mining techniques for value series based on evolution attributes such as number of authors, commit messages, lines of code, bug fix count, etc. Daily data points of these evolution attributes were captured over a period of two months to predict the defects in the subsequent two months in a project. For that, we developed models utilizing genetic programming and linear regression to accurately predict software defects. In our study, we investigated the data of three independent projects, two open source and one commercial software system. The results show that by utilizing series of these attributes we obtain models with high correlation coefficients (between 0.716 and 0.946). Further, we argue that prediction models based on series of a single variable are sometimes superior to the model including all attributes: in contrast to other studies that resulted in size or complexity measures as predictors, we have identified the number of authors and the number of commit messages to versioning systems as excellent predictors of defect densities.

1 Introduction

How does the course of software development over time influence defect densities? To address this question we focus on series classification techniques, where we generate value series from software evolution and take it as input for quality assessment.

Defect prediction models of previous studies often relied on metrics that represent the state of the software system at a specific moment in time (e.g. [7, 3, 17, 11]). Such metrics describe, for example, the sum of changes implemented on a certain part of the system or are other types of measures such as size and complexity metrics (e.g. [2]).

In this paper we show that change over time is an important aspect in software prediction models. Our previous study already incorporated time-related data into classification [20], where we measured values such as the *average number of days between changes* and the *peak month*, in which most changes took place within the learning period. As these time-related features have been very important for good prediction models, we go a step further and explicitly focus on series of metric values.

Sequential patterns are important in many domains, because they can be exploited to improve the prediction accuracy of classifiers. A sequence $x = \langle x_1, x_2, x_3, \ldots, x_n \rangle$ of change events during software development contains the information on the course of development additionally to the pure attributes of the sum of all change events describing the state at the final point in time. As one of the first studies we analyze value series of evolution data to create defect prediction models.

Our evolution attributes of source files refer to measures obtained from versioning and bug reporting data such as the number of bug fixes or the number of authors working on a particular file. Evolution attributes are measured daily over a period of two months to predict the number of defects in source files in the subsequent two months. The data of three independent software projects in our field study allows us to build prediction models with high accuracy using series of evolution attributes.

Our study can be compared to other prediction approaches since we included the same data of a previous study [20] taken from a commercial system into our analysis. Additionally, we broadened our evaluation by incorporating data from open source projects and created prediction models on the evolution data of these different software projects.

This paper is organized as follows. We present a description of our knowledge discovery process in Section 2. Section 3 lays the foundation through the preparation of evolution data. Section 4 explains how we set up evolution series, which are used as input to the series mining of Section 5.

Our results are reported in Section 6. In Section 7 we discuss the state the state of the art and Section 8 finishes with conclusions.

2 Knowledge Discovery Process

Several consecutive steps are executed in our knowledge discovery process to obtain prediction models based on value series. The basic process is as follows:

First, the data collection steps extract evolution data from two sources: versioning systems such as CVS and issue tracking systems such as Jira. Data items taken from different systems have to be assembled into a joined data model to establish an evolution database. Additionally, a relationship is established between the data items from a single data source (e.g. the transactions of the versioning system are reconstructed to group items into sets of co-changed elements).

Next, the evolution database is used to compute change attributes such as the *number of lines added for bug fixes*, the *number of co-changed files*, or the *number of modifications without a commit message*. These are the characteristics of our data items that are used to create value series for defect prediction. Fenton and Neil pointed out that a sound prediction model has to incorporate different types of attributes [4]. Accordingly, we analyze several types of attributes, where a value series is created for each attribute type. Additionally, series containing attributes of all categories represent the changes over time for a single instance (i.e. a file).

In the next step we take the value series of evolution attributes as the basis for our defect prediction models. To be able to apply classification algorithms to the value series we extract features describing the relevant characteristics of the value series. In data mining the input attributes used by the algorithms are called *features*. An example of such a feature is the *maximum number of files changed together regarding a particular file*. The feature extraction is done automatically with the help of genetic programming, in which several operations are applied on the data points in the value series. The genetic algorithm searches the feature space guided by a fitness function (i.e. the correlation coefficient of our defect prediction models). The best features discovered through genetic programming are the input of the regression algorithms to create the prediction model. The platform for our series mining activities is the YALE machine learning environment [14]. It allows the design of operator chains for a large number of learning problems and includes many data mining algorithms such as support vector machines, decision tree learners, bayesian learners, etc.

Finally, we describe the results of a field study, in which we applied the prediction models to several projects taken from three different domains to evaluate the accuracy of the prediction. The following sections describe each step in detail and present our results.

3 Preparing Evolution Data

There are many different systems that all record different aspects of the development and evolution of a software system. Project managers have to be able to observe the status of individual tasks as well as the progress of the entire project. Developers need information about what is requested from them and need storage systems for their results. Thus, different aspects are covered by different systems, which we have to integrate for our analysis.

3.1 Data Extraction

As data sources for our defect prediction we utilize versioning and issue tracking systems. Currently our approach supports the versioning system CVS and the issue tracking systems Jira and Bugzilla. CVS keeps track of all changes in source files. For each file we retrieve these change logs, parse, and store the extracted information into the evolution database [20]. Issue or bug report data is obtained from Jira or Bugzilla. These systems track bug and feature requests from users and customers. We process each request and add the information to the database.

3.2 Data Processing

Software components are related with each other through shared data or method calls, or inheritance relations. During software development a relationship is also established when developers work on several classes or modules to accomplish a certain task. Co-change coupling during the evolution of a software system provides valuable information in the context of maintenance [5]. We obtain couplings from the versioning systems by reconstructing transactions when files are submitted together to CVS. Thus, *transactions T_n are defined as a set of files, which were checked-in into a versioning system by a single author with an equal commit message*. To capture the entire transaction, possibly lasting several minutes we use a dynamic time window approach. Every file submission outside of a previous transaction defines the start of a new transaction lasting initially for 60 seconds. When another file submission is discovered within the time frame of this transaction then the file is added and the transaction time is expanded to last until 60 seconds after the last file submission time.

Co-change coupling is established based on common transactions of files. *Two entities (e.g. files) are coupled, if a modification of the implementation affected both entities.* The intensity of coupling between two entities a, b can be determined by counting all transactions where a and b are members of the same transaction, i.e., $C = \{\langle a, b\rangle | a, b \in T_n\}$ is the set of change couplings and $|C|$ is the intensity of coupling [19].

3.3 Combining Data Sources

To count defect densities of files we attach defect information stored in the issue tracking system with the file information from the versioning system. For this step we inspect the commit messages associated with revisions of source files for references to issues, which is accomplished with regular expressions. When a matching issue is found, a link between the issue and the corresponding CVS log entry is stored only if the creation date of this issue is before the submission of the file to CVS.

4 Generating Evolution Series

In this paper the focus is on the lifetime of source files during software evolution. For this we measure a set of evolution attributes for each source file over time and compose multiple value series describing the data points of the attributes as a sequence of measures. In our field study we use two months of development time to predict the defects of the following two months (see Section 6.2). The first two months comprise 61 days. On all days of this series period we measure the attributes for each file. For example the number of lines added within one day is summarized for the data points of this attribute in the value series. As a result many values in the series are zero, as in a development project not all source files are modified on each day. The number of defects is predicted for the entire period of the following two months for each source file. Thus, the instances for the prediction models are files. In the following we describe the different evolution attributes and the generation of series in detail.

A definition of generalized series is used for value series: In a series each element x_i is composed of two components. The first is the index describing a position on a straight line (e.g. time). The second is a vector of values. In our case we use two types of vectors. One is a reduced case where only one attribute represents the vector. In the second case the dimension of the vector is given by the number of evolution attributes [14].

4.1 Evolution Attributes

Using the information stored in the evolution database we compute a number of attributes that quantify the software evolution in a source file. According to previous studies, which showed that relative data outperforms absolute values in defect prediction [16], we use the following evolution attributes as foundation for our value series of relative measures. All measures are collected within a time frame of two months, where the data points are accumulated.

- *Lines Added* this measure represent the sum of lines added. This measure is one of the indicators of size, where the developer probably adds functionality through new source lines.

- *Lines Deleted* describes the number of lines removed from a file. When a certain line is changed the versioning system counts one line added and one line removed. The number of deleted lines is additionally an indicator for a "clean up" mentality to keep only the used code.

- *Number Changes* is the number of modifications implemented within a single day on a given file. This is a general activity indicator.

- *Number Authors* is the number of authors working on a single file. When several authors work on the same day on one file, we expect interferences between the changes.

- *Author Switches* describes the number of times the work of a file is handed over from one author to another. When, for example, two authors work in the sequence author1, author2, author2, author1 we denote two author switches. When the work of several authors is strongly interwoven, we expect the strongest impact on defects.

- *Commit Messages* indicates the number of different commit message from developers on changes. We see the commit message as an indicator for the discipline of developers, as developers sometimes tend to reuse the message of the last commit instead of describing the actual work.

- *With No Message* describes the number of changes without a commit message. This provides insight into the discipline of the developers. This could be an indicator that the developer is in a hurry.

- *Number Bugfixes* is the number of issues that caused changes in the file. A file with many defects in the past is expected to have defects in the future [7].

- *Bugfix Lines Added* is the counterpart to the number of lines added, but this time the number of lines is only taken into account if the change is a bug fix according to the information from the issue tracking system.

- *Bugfix Lines Deleted* measures the number of lines deleted from a file only for bug fixes.

- *Couplings* is the strength of co-change coupling of a file with other files. It counts how many times a change was done with other files. Coupling has been an indicator for architectural weeknesses [19].
- *CoChanged Files* – in contrast to the *Couplings* – describes the number of files that were changed together with the file of interest. For several modifications each co-changed files is counted only once. We expect the more files are changed together, the higher is the complexity and the more difficult it is to keep the consistency.
- *CoChanged New Files* is the number of files that were created together with a change to the investigated file. When new files are introduced into a system, it is an indicator for growth and new functionality.
- *Transaction Lines Added* is the number of lines added in all files that have couplings with the file of interest. This measure the entire work of a commit of files that are related to the file of interest.
- *Transaction Lines Deleted* is the number of lines deleted in all files with common transactions on changes.
- *Transaction Bugfix Lines Added* measures the number of lines added for all files during a change event that treats a bug.
- *Transaction Bugfix Lines Added* describes the number of lines deleted for all files during bug fixes together with the file of interest.

4.2 Value Series

The absolute values of the evolution attributes, which are described in the previous section, are used to construct the final value series containing relative measures ordered by time. For each day the relative attribute value is computed and added to the value series. For example, we use the number of authors relative to the number of changes on each day in our series period. The sequence 1/1, 0, 2/3, 1/1 would result for four days when one change is committed on the first day, no change happens on the second day, two developers implemented a total of three changes on the third day, and one change is committed on the fourth day.

The following list of relative measures is used to create value series per file for each day. For each relative feature a division of relative values from the previous section is computed.

- *LinesAdd*: Lines of code added within a day / Total lines of code until this day.
- *LinesDel*: Lines of code deleted within a day / Total lines of code until this day.
- *ChangeCount*: Number of changes within a day / Total number of changes in the history of the file until this day.
- *Authors*: Number of authors within a day / Number of changes within this day
- *AuthorSwitches*: Number of switches of the author / Number of authors
- *CommitMessages*: Number of different commit messages / Number of changes
- *WithNoMessage*: Number of changes without commit message / Number of commit messages
- *BugfixCount*: Number of bug fixes / Number of changes
- *BugfixLinesAdd*: Lines added for bug fixes / Number of lines added (any type)
- *BugfixLinesDel*: Lines deleted for bug fixes / Number of lines deleted (any type)
- *CoChangeCount*: Number of couplings / Number of changes
- *CoChangedFiles*: Number of co-changed files / Number of changes
- *CoChangedNewFiles*: Number of newly introduced files that are co-changed / Number of co-changed files
- *TLinesAdd*: Number of lines added in all co-changed files / Number of couplings
- *TLinesDel*: Number of lines deleted in all co-changed files / Number of couplings
- *TBugfixLinesAdd*: Number of lines added in all files for bug fixes / Number of lines added
- *TBugfixLinesDel*: Number of lines deleted in all files for bug fixes / Number of lines deleted

5 Predicting Defects based on Evolution Series

Given the value series of relative evolution attributes as described in the previous section, the aim of our approach is to derive models for predicting the number of defects in source files. For the model generation we use ”classical” data mining algorithms such as linear regression. These algorithms are not able to handle value series in the explicit representation, but can operate on sets of attributes instead of sets of series of values.

We generate a new representation of our series information that is suitable for linear regression. This task is called feature extraction, where each series is described by a set of relevant characteristics that make different evolution series distinguishable. In a similar manner we could describe a value series containing positions of the sun on earth with the following features: one cycle lasts for 24 hours, the maximum is reached at noon, sunrise and sunset are related with the degree of latitude on earth, ...

The feature extraction itself is decomposed into a set of basic operators. For example functions returning the minimum, average, or maximum of the values in a series are basic operators. Other basic operators return an index such as the location of a peak value within a given series. Such basic operators are assembled into an operator tree describing the extraction steps of the final features. However, the manual selection of an optimal set of operators is a tedious task. Therefore, machine learning is used to select appropriate operator tree, where the selection is done with the help of genetic algorithms.

Thus we have to carry out two learning tasks for our defect prediction.

1. Learning a set of operator trees for the feature extraction utilizing genetic programming. The resulting features describe relevant characteristics of evolution series for data mining algorithms such as linear regression.

2. Learning a model for defect prediction from the extracted features.

5.1 Extracting Features from Series

In the process of feature extraction a set of basic operators is organized into a tree, where each operator uses the output of the predecessor. The output of the operators at the leaves produce the features of the series. We distinguish two types of basic operators: Transformations and functions:

Transformations convert a series into another series. Different types of transformations are available for our defect prediction approach such as filters (e.g. smoothing), frequency transformations (e.g. Fourier transformation), generalized windowing, etc. Windowing operators apply a given function on a range of values within the series and additionally slide the window over the series. Others are branches that pass on the interim results to two successor sub-trees.

Functions generate single values based on the entire value series and are always the last step of the feature extraction process (i.e. the leaf nodes of the operator tree). Examples of functions are statistics such as average, variance, standard deviation. These functions may be applied on the values themselves or on the indexes of the values, where for example the index of a peak value could be extracted. For an extensive list of transformations and functions see [14].

5.1.1 Genetic Programming

The (locally) optimal feature extraction approach (i.e. operator tree of transformations and functions) is elicited with genetic programming utilized on the operator trees.

Mutations randomly insert a new operator, delete an operator, replace an operator, or change the parameters of an operator.

Crossover switches a sub-tree from one feature description tree by a sub-tree from another tree. According to the standard process of genetic programming the instances with the highest fitness are selected for the next generation.

Selection is done based on a tournament between all members of a generation in the genetic algorithm.

Fitness of the operator trees for the tournament selection is assessed based on the defect prediction capability of the resulting features. Our fitness function is the regression algorithm itself that is used for the generation of the prediction model. Thus, for each operator tree a regression function is generated based on a training set of a random sample containing 50 evolution series instances and the accuracy of the prediction of defects is used as the fitness value. As a result, the operator trees generating features that predict the defects best are selected for the next generation.

Initiation of the first generation in the genetic algorithm is based on 50 operator trees, where the operators are randomly selected from the pool of available transformations and functions.

We limited the maximal number of generations by 8. Further, the following parameters are defined for our approach: probability of adding a new operator = 0.4, probability of adding a branching operator to create new sub-trees = 0.05, probability of changing an operator = 0.4, probability of removing an operator = 0.2, probability of performing a crossover = 0.5, probability of changing a parameter = 0.1.

5.2 Applying Data-Mining Algorithms to Series Features

The best features selected by the genetic programming algorithm are used for the creation of the prediction of defects. The data mining algorithm for our prediction is linear regression, as our outcome as well as our features from value series are numeric. This is a staple method in statistics where the predicted value is represented by a linear combination of the input attributes (i.e. features) with weights $w_0, w_1, w_2, \ldots, w_n$ and attributes $a_1, a_2, \ldots, a_n$:

$$x = w_0 + w_1 a_1 + w_2 a_2 + \ldots + w_n a_n$$

The weights are derived from the training data set minimizing the sum of squares of the distance between the predicted value x and the actual one y. The distance is summarized for all instances (k) of the training data set:

$$\sum_k (y - \sum_n w_i a_i)^2$$

The numeric prediction algorithms are used twice in our process. First it is used for the evaluation of fitness in genetic programming, where prediction models are build on a small random sample of evolution series and the correlation coefficient is utilized to select the best features. Finally, we apply the prediction algorithms on the extracted features taking all instances of the training set (i.e. all evolution series) into account to create the final prediction model.

6 Evaluation

We evaluated the approach of defect prediction based on series mining with the help of a field study, where we selected different real world projects and analyzed the predictability of defects in the near future.

6.1 Field Study

In our field study we analyzed two open source projects (ArgoUML and the Spring framework) and a commercial software system, which we selected to get comparability with the results of previous studies ([20, 21]). The commercial software system is from the health care domain, written in Java and contains more than 8.600 classes with 735.000 lines of code. ArgoUML and the Spring framework are large well-known open source projects both developed in Java and consist of about 5.000 and 10.000 classes, respectively. In Java classes are almost equivalent to files, thus we use files as basic instances in our analysis.

6.2 Evaluation Setup

To estimate the accuracy of our defect prediction approach we use the same time periods for all projects, regardless in which development state the project is. In a previous study we have shown that defects that occur within a short time before a release can be better predicted than defects after a release [20]. In our current research activity we have two periods:

- *Series Period:* November-December 2005. In this period we take evolution attributes from the versioning system and construct value series to represent the flow of the development over time. Each series of the attributes from Section 4.2 has a length of 61 days given the two months of the series period. This information is used in our series mining to predict the defects of a source file in the next period.
- *Target Period:* January-February 2006. With our prediction models based on series mining we try to predict the number of all defects in the target period, where the defects are counted based on the information from the issue tracking system and are mapped to files as described in Section 3.3.

These two periods are also used in our previous study [20] and thus enable us to compare the results of these two approaches.

6.3 Measuring Prediction Performance

For the assessment of our prediction models we use the following metrics:

- *Correlation Coefficient* (Cor. C.) ranges from -1 to 1 and measures the statistical correlation between the predicted values and the actual ones in the test set. A value of 0 indicates no correlation, whereas 1 describes a perfect correlation. Negative correlation indicates inverse correlation, but should not occur for prediction models. The correlation coefficient is computed with the following formula, where p are the predicted values and a are the actual ones:

$$\frac{\sum_i (p_i - \overline{p})(a_i - \overline{a})/n - 1}{\sqrt{(\sum_i (p_i - \overline{p})^2/n - 1) * (\sum_i (a_i - \overline{a})^2/n - 1)}}$$

  where
  $\overline{p} = 1/n \sum_i p_i$ and $\overline{a} = 1/n \sum_i a_i$
  The correlation coefficient is our primary performance indicator.

- *Mean Absolute Error* (Abs. Error) is the average of the magnitude of individual absolute errors. This assessment metrics does not have a fixed range like the correlation coefficient, but is geared to the values to be predicted. As a result, the closer the mean absolute error is to 0 the better. A value of 1 denotes that on average the predicted value differs from the actual number of defects by 1 (e.g. 3 instead of 4). The mean absolute error is computed with the following formula:

$$\frac{|p_1 - a_1| + \ldots + |p_n - a_n|}{n}$$

- *Mean Squared Error* (Sqr. Error) is the average of the squared magnitude of individual errors and it tends to exaggerate the effect of outliers – instances with larger prediction error – more than mean absolute error. The range of the mean squared error is geared to the ranges of predicted values, similar to the mean absolute error. But this time the error metrics is squared, which overemphasize predictions that are far away of the actual number of defects. The quality of the prediction model is good, when the mean squared error is close to the mean absolute error. The formula for mean squared error is:

$$\frac{(p_1 - a_1)^2 + \ldots + (p_n - a_n)^2}{n}$$

As validation method we use 10-fold cross validation to estimate the performance of our prediction models. In this method the set of source files is randomly split into 10 disjoint sets of equal size. The validation is executed 10 times, where the linear regression is trained on 9 of 10 folds and the remaining one is used to calculate the error rates and the correlation coefficient. After the 10 turns the final performance estimates are generated through averaging the results of the 10 turns.

The validation used two times: First it is used for the assessment of the fitness of the features during genetic programming and finally it is used for the assessment of the prediction models resulting from linear regression with the best features (see Section 5).

6.4 Results

In the following we describe the field study with the selected software projects and discuss performance measures of our prediction models. Furthermore, we investigate the significance of evolution attributes.

6.4.1 How well can we predict the number of defects in source files with series mining?

To answer this question we take the entire evolution series containing values of all attributes such as *LinesAdd* or *Authors* (see Section 4.2). Table 1 describes the performance measures of our defect prediction models. The first remarkable number is the very high correlation coefficient of the commercial system from the healthcare domain. A correlation coefficient of 1 would indicate perfect correlation of the prediction with the actual value, where the received 0.946 indicates that very strong prediction models can be built based on evolution series. The other two projects reach a correlation coefficient greater than 0.7, which is still good.

According to the first performance indicator also the mean absolute error of all projects is low. The absolute error has to be measured in relation with the predicted quantities. In our case we predict the number of defects that lie in the range of 0 up to 7. As a result, the measured mean absolute errors of 0.208 to 0.306 are low. The commercial project has a higher absolute error than the two open source

	Cor.C.	*Abs.Error*	*Sqr.Error*
Commercial system	0.946	0.306	0.508
Spring framework	0.716	0.229	0.770
ArgoUML	0.730	0.208	0.624

Table 1. Defect prediction with series including all evolution attributes

Number of defects per file	Comm. System	Spring	Argo UML
1	46	80	47
2	11	15	9
3	5	3	2
4	7	2	0
5	2	0	0
6	1	0	0

Table 2. Defect distribution

projects because it has more files with multiple defects (e.g. 5 or 6 defects), which can be seen in Table 2.

The good prediction measures are supported by the mean squared error, which emphasizes outliers more than the mean absolute error. The squared error is lowest for the commercial project with a value of 0.508. This corresponds with the high correlation coefficient and indicates that the prediction is very accurate. However, also the squared errors of Spring with 0.770 and of ArgoUML with 0.624 are low. Thus, we conclude:

Accurate prediction models can be developed based on series mining of evolution data.

6.4.2 Which attributes are most significant for defect prediction?

In the previous section we presented prediction models based on series mining with a very high correlation coefficient and good error measures. These models are created from an evolution series containing all attributes described in Section 4. We are interested to find out which attributes are most significant to create accurate prediction models. For this we create prediction models on value series for each single evolution attribute. Table 3 presents the correlation coefficients of all generated models, as this performance indicator represents the relationship between the predicted values and the actual ones.

All three projects of the field study exhibit high values for the correlation coefficient on the series containing the number of authors. In the commercial system as well as in ArgoUML this single series is even the one with the highest correlation coefficient. For the Spring framework it is only

	Comm.	Spring	Argo
	Cor.C.	*Cor.C.*	*Cor.C.*
LinesAdd	0.616	0.195	0.161
LinesDel	0.305	0.111	0.234
ChangeCount	0.517	0.653	0.268
Authors	0.946	0.628	0.760
AuthorSwitches	0.622	0.210	0.357
CommitMsgs	0.943	0.480	0.459
WithNoMsg	0.273	0.008	-0.054
BugfixCount	0.455	0.290	0.253
BugfixLinesAdd	0.437	0.294	0.295
BugfixLinesDel	0.736	0.319	0.244
CoChangeCount	0.548	0.336	0.388
CoChangedFiles	0.481	0.240	0.409
CoChangedNew	0.426	0.171	0.233
TLinesAdd	0.598	0.622	0.442
TLinesDel	0.586	0.579	0.225
TBugfixLinesAdd	0.482	0.318	0.362
TBugfixLinesDel	0.460	0.319	0.296
series of all attributes	0.946	0.716	0.730

Table 3. Correlation coefficients of series with a single attribute and the summarizing series including all attributes

exceeded by the series with *ChangeCounts*, which describes the number of changes per day in relation to total number of changes for this particular file. In the two other projects the *ChangeCount* is ranked only in the middle-field.

Authors seems to provide good input to series mining, which contrasts the results of Graves et al. [7]. In our knowledge discovery process we use value series for defect prediction. Therefore, we measure how many authors have implemented modifications to a given file and set this measure in relation to the number of modifications implemented by these authors. We use relative measures, which have shown to be better predictors than absolute measures [16]. Moreover, we observe the alteration of the number of authors implementing modifications over time, which can provide more accurate data to the prediction models than metrics focusing on a fixed point in time.

Another interesting sub-series is the one containing the number of commit messages in relation to the number of changes. This *CommitMsgs* series has even the second highest correlation coefficient in the commercial project and ArgoUML. In the Spring framework it is on position five with a correlation coefficient of 0.48.

It is quite surprising that the highest performance measures are not reached by size or complexity metrics, but by process and workflow related aspects such as *Authors* and *CommitMsgs*. However, on the third position for ArgoUML and Spring appears the series of *TLinesAdd* (see Table 3). This attribute incorporates the number of lines changed within a commit transaction counting added lines of all files that are involved in the transaction. This series reflects an aspect of the interdependency in object oriented software systems, as we take changes to other (related) files within a transaction into account. Contrary, the pure size measure of added lines of a particular file is represented by *LinesAdd*. Although this sub-series plays a remarkable role for the commercial system, it has a very low correlation coefficient in the open source projects. For the sub-series we conclude:

> Projects have different rankings of sub-series, where common aspects can be identified, such as the number of authors or commit messages.

6.5 Limitations of the Study

Our models are based on evolution data taken from versioning systems and the number of defects is established with data from the issue tracking system. The matching between these two systems is based on heuristics as described in Section 3.3. Although, such an approach is frequently used in research (e.g. [17, 18, 7, 20]) we cannot assure that we have identified all bugs as we certainly miss the ones that were not reported to the issue tracking system.

In general our mining approach is strongly dependent on the quality of our data for the field study. Validity of our findings is related with the data of the versioning and issue tracking system. Versioning systems register single events such as commits of developers, where the data depends on the work habits of the developers. However, in our previous work we showed that an averaging effect supports statistical analysis in general [19]. Additionally, the data about work habits of people is by its own interesting information that we use for our quality prediction, where we can show that our prediction models rely heavily on such features (e.g., number of commit messages).

The data points of our value series are computed as sums of each day. As a result, if a developer works through the night and commits some modifications before midnight and the remaining parts of modifications after midnight, we count the work on two days. Although this influences our value series, such information could still be valuable for defect prediction, because the working over night might have consequences on the level of concentration and the resulting software quality.

We have selected different projects for our field study: commercial vs. open source; different domains such as health care, UML and application server. However, we cannot claim that these projects are representative for all different types of software projects. As a result the application of our approach to other software systems has to be re-evaluated on a per project basis.

7 Related Work

The focusing on software evolution as a key aspect in software development provided interesting results in previous research activities. Zimmermann et al. developed ROSE, a mining tool that suggests necessary changes to other files when a developer starts working on a certain file or group of files [22]. We already used historical data to identify hot spots within software architecture, which should be subject to re-engineering activities [5]. Both research activities rely on the idea of co-change coupling, where common changes to files are analyzed. Based on evolution data Mockus and Votta accomplished an in-depth analysis for the reasons of software changes [15]. Software evolution analysis is a very computational intensive task, where Bevan et al. have implemented a system called Kenyon for the efficient fact extraction from data sources such as software repositories and bug tracking systems and storage of the evolution information for further analysis [1].

Historical data is necessary to assess quality prediction models as it can be used to count defects in software systems. Graves et al. studied the aging of software and which factors lead to faults in future. They created a defect prediction model incorporating the sum of contributions from all changes to a given module, where large, recent changes had the highest impact [7]. Another study was done by Ostrand et al. [18] in which several aspects of the history of software systems were utilized to build a negative binomial regression model for defect prediction. This study focused on long time periods and investigated 14 releases of a software system.

Other research activities focused on conventional object-oriented metrics such as the ones of Chidamber and Kemerer [2]. Khoshgoftaar et al. use software metrics as input to classification trees to predict fault-prone modules. With the help of statistical tests subsets of modules were detected with uncertain classifications allowing enhancement strategies to resolve uncertainties [9]. A recent approach for defect prediction based on software metrics was described by Nagappan et al. [17]. They discovered in a study of five Microsoft systems that failure-prone software entities are statistically correlated with code complexity measures. However, they could not identify a single set of complexity metrics suitable for prediction in all five projects.

Menzies et al. argue that the research on defect prediction should focus on the methods instead on the search for an optimal subset of the available data. With the help of static code metrics they could identify only one out of six methods (Naive Bayes with log-filtered values), which had a median performance that was both large and positive [13]. Kim et al. does not focus on metrics but tries to identify entities that are in the locality of other bugs (or bug fixes). They exploit temporal and spatial locality and keep the information in a bug cache [10]. In our current research we focus on value series of evolution attributes.

The time series classification problem can be defined as follows: Given a universe of objects. Each object is described by a certain number of temporal attributes and classified into one particular class. The goal is to find a function f(o) which is as close as possible to the true classification c(o) [6]. Kadous solves the problem of multivariate time series classification with the help of parameterized event primitives (PEPs). The extracted events are clustered to create prototypical events. They are used as the basis for creating more accurate and comprehensible classifiers than hidden Markov Models or recurrent neuronal networks [8]. Manganaris developed a system for supervised classification of univariate signals using piecewise polynomial modeling combined with a scale-space analysis technique (i.e. a technique that allows the system to cope with the problem that patterns occur at different temporal scales) [12].

8 Conclusions

We presented a new approach to software defect prediction based on value series of evolution attributes: We conducted one of the first studies utilizing value series for defect prediction in software engineering. In this approach an entire series of measurements is used to predict a single label (the number of defects in a file containing object-oriented entities). For the evaluation of our approach we conducted a field study of three different software projects. Each of them has its independent timeline regarding the evolution phases and release cycles. We use a fixed date for the data extraction from these projects, which results in a randomized selection within the timeline of each individual project.

Our evolution measurements were obtained from software repositories such as the concurrent versioning system (CVS) where single information items such as the number of authors were gathered into value series. Our results showed that evolution series are excellent predictors of defect densities. We describe an analysis focusing on sub-series, where the prediction models based on series of a single variable are sometimes even superior to the overall model. An interesting proponent of this category is the number of authors, where good models can be created on (up to a correlation coefficient of 0.946). Other aspects of software evolution, which are often used in software prediction, are less important (e.g. lines added).

Future work will concentrate on the input data we use for series mining. In another study we already presented manifold features for software evolution [20]. We want to enrich our series mining approach to be able to analyze software projects in more detail and to get a better understanding of the forces that influence software quality. To accomplish this goal we also look for improvements of series mining and the understandability of the resulting prediction models. For example classification and regression trees provide the benefit that they provide a clear picture of the prediction model and the relationships of the used features. Kadous [8] presents interesting ideas in that direction.

9 Acknowledgments

Thanks to Patrick Knab and Emanuel Giger for their valuable feedback on earlier versions of this paper.

This work was partially supported by the Swiss National Science Foundation as part of the project "COSE - Controlling Software Evolution," and the Hasler Foundation as part of the project "EvoSpaces - Multi-dimensional Navigation Spaces for Software Evolution".

References

[1] J. Bevan, E. J. W. Jr., S. Kim, and M. W. Godfrey. Facilitating software evolution research with kenyon. In *Proceedings of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 177–186, Lisbon, Portugal, September 2005.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[3] G. Denaro and M. Pezzè. An empirical evaluation of fault-proneness models. In *Proceedings of the International Conference on Software Engineering*, pages 241–251. ACM Press, May 2002.

[4] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, September 1999.

[5] H. Gall, M. Jazayeri, and J. Ratzinger (former Krajewski). CVS release history data for detecting logical couplings. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 13–23, Lisbon, Portugal, September 2003. IEEE Computer Society Press.

[6] P. Geurts. Pattern extraction for time series classification. In *Proceedings of the European Conference on Principles of Data Mining and Knowledge Discovery*, pages 115–127, 2001.

[7] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[8] M. W. Kadous. Learning comprehensible descriptions of multivariate time series. In *Proceedings of the International Conference on Machine Learning*, pages 454–463, San Francisco, USA, June 1999.

[9] T. M. Khoshgoftaar, X. Yuan, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Uncertain classification of fault-prone software modules. *Empirical Software Engineering*, 7(4):297–318, December 2002.

[10] S. Kim, T. Zimmermann, J. E. James Whitehead, and A. Zeller. Predicting faults from cached history. In *Proceedings of the International Conference on Software Engineering*, pages 20–26, Minneapolis, USA, May 2007.

[11] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 119–125, Shanghai, China, May 2006. ACM Press.

[12] S. Manganaris. *Supervised Classification with Temporal Data*. PhD thesis, Computer Science Department, School of Engineering, Vanderbilt University, December 1997.

[13] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[14] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. YALE: Rapid prototyping for complex data mining tasks. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pages 935–940, Philadelphia, USA, 2006.

[15] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*, pages 120–130. IEEE Computer Society, 2000.

[16] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the International Conference on Software Engineering*, pages 284–292, St. Louis, MO, USA, May 2005. ACM Press.

[17] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the International Conference on Software Engineering*, pages 452–461, Shanghai, China, May 2006. ACM Press.

[18] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *Proceedings on the International Symposium on Software Testing and Analysis*, pages 86–96, Boston, Massachusetts, USA, July 2004.

[19] J. Ratzinger, M. Fischer, and H. Gall. Evolens: Lens-view visualizations of evolution data. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 103–112, Lisbon, Portugal, September 2005.

[20] J. Ratzinger, M. Pinzger, and H. Gall. EQ-Mine: Predicting short-term defects for software evolution. In *Proceedings of the Fundamental Approaches to Software Engineering at the European Joint Conferences on Theory And Practice of Software*, pages 12–26, Braga, Portugal, March 2007.

[21] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining software evolution to predict refactoring. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, page to appear, Madrid, Spain, September 2007.

[22] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the International Conference on Software Engineering*, volume 00, pages 563–572, Edinburgh, Scotland, UK, May 2004.