International Journal of Software Engineering and Knowledge Engineering

TEMPORAL SOFTWARE CHANGE PREDICTION USING NEURAL NETWORKS

MEHDI AMOUI
Department of Electrical and Computer Engineering
University of Waterloo, Ontario, Canada
mamouika@uwaterloo.ca

MAZEIAR SALEHIE
Department of Electrical and Computer Engineering
University of Waterloo, Ontario, Canada
msalehie@uwaterloo.ca

LADAN TAHVILDARI
Department of Electrical and Computer Engineering
University of Waterloo, Ontario, Canada
ltahvild@uwaterloo.ca

Predicting changes in software entities (e.g. source files) that are more likely to change can help in the efficient allocation of the project resources. A powerful change prediction tool can improve maintenance and evolution tasks in software projects in terms of cost and time factors. The vast majority of research works have focused on determining "where" the most change-prone entities are, and "how" the change will be propagated through a system. This article suggests that knowing "when" changes are likely to happen can also provide another consideration for managers and developers to plan their maintenance activities more efficiently. To address this issue, a Neural Network-based Temporal Change Prediction (NNTCP) framework is proposed. This novel framework indicates "where" the changes are likely to happen (i.e. hot spots), and then adds the time dimension to predict "when" it may occur. In proving this concept, the NNTCP framework is applied in two large-scale open source software projects, Mozilla and Eclipse. The results obtained indicate NNTCP can predict the occurrence of several future revisions with reasonable performance.

Keywords: Software change prediction; Temporal prediction; Neural network.

1. Introduction

Maintenance and evolution are inevitably costly and time-consuming activities for almost any software-intensive system. The changes made by these tasks are due to different reasons. They may be *corrective* for fixing bugs, *adaptive* for adapting the software to new environments, *perfective* for updating the software according to requirements changes, and finally *preventive* for making the software more maintainable [10]. Moreover, a key characteristic of the maintenance/evolution of

large software systems is that making changes becomes increasingly difficult and expensive over time [3]. These problems are well-known as software aging and code decaying, which have gained growing importance both in academia and industry.

Change prediction plays a key role in the effective planning for maintenance/evolution. It can be performed for different purposes like reverse engineering and cost estimation [4]. Change prediction is often applied to determine change-prone entities and change propagation patterns of a product [26]. The entities may include any set of software modules such as files, classes, packages, and so on. The entities can be selected from both source code and documentation artifacts. A good prediction model can be used for future resource allocation, cost estimations, and specifying *where* maintenance/evolution tasks are "better" to start. In large-scale systems, this issue is crucial to the early detection of potential changes that may occur in the future, and will consequently reduce the costs and risks of applying those changes.

The main motivation of this work is that knowing *where change-prone entities are*, may not be enough to attain prediction objectives. Knowing *when the changes will occur* can help managers and developers to plan better for later revisions of change-prone entities. Consequently, this can result in more effective change prioritization and efficient resource allocation. This paper proposes a framework to predict the future change date of constituent entities in a typical system. To achieve this goal, a novel framework, called Neural Network-based Temporal Change Prediction (NNTCP), is considered. This framework uses a history log of software changes from a software repository to measure applicable development metrics. The metrics are fed into an artificial neural network, which estimates the future change date of a given entity. To validate this framework, the concept is applied to two large-scale software systems: Mozilla and Eclipse.

The rest of this paper is organized as follows: Section 2 reviews related work in the software change prediction domain. Section 3 provides an overview of the proposed framework for software change prediction. Section 4 describes experiments conducted on two case studies and discusses the results. Finally, Section 5 presents the conclusions and outlines several ideas for future work.

2. Related Work

Generally speaking, research efforts on software change prediction can be analyzed in terms of *what* the researchers try to predict and *how* they perform the prediction. From the *what* perspective, most efforts focus on predicting: i) the number of changes, bugs or faults, and ii) the probability or rate of change (i.e. fault-proneness) of the software entities [6]. Some researchers like Girba *et al.* [4] add a time layer to structural information, which can facilitate combining historical data extracted from a change log with other sources of information. The *how* viewpoint classifies prediction models as either stateless or stateful. The former relies only on the current state of a system, while the latter is based on the historical data, regardless of

its current state.

Time-series prediction for software entities (such as software change/fault prediction) can be performed through two approaches: mathematical/statistical models, and AI/soft computing techniques. Several mathematical/statistical prediction models have been utilized for this purpose. For example, some research efforts are based on regression models [5, 2], while other efforts in this category predict probability of change using product metrics [19]. A notable drawback of these models is sensitivity to input data.

There are numerous efforts on software prediction using AI and soft computing methods. These efforts mostly address the prediction of quality indicators such as maintainability and reliability. Khoshgoftar *et al.* [14] use neural networks to predict the number of software development faults for Ada applications using software metrics as predictors. Huang *et al.* [9] proposed a novel neuro-fuzzy constructive cost model (COCOMO) for software cost estimation, and Ramanna [21] used neural networks to predict the number of required changes in a file or a module, in order to achieve quality assurance standards. In another research, Shepperd [22] compared four prediction techniques using simulation, namely, regression, rule induction, nearest neighbor, and neural nets.

Besides the prediction techniques mentioned earlier, predictors play an important role in the effectiveness of the whole process and the quality of the outcomes. Prior efforts have identified important predictors for software change prediction (e.g. Khoshgoftaar *et. al.* [12], and Mockus et. al. [18]). Predictors are classified as product metrics, development metrics, deployment and usage (DU) metrics, and software and hardware configuration (SH) metrics [17].

Most of the prediction methods investigated are based on the analysis of software history and its evolution. The information is normally captured from software repositories and/or documentation. As a software application becomes older, extraction, manipulation, and analysis of its history data become more time-consuming and complex. Therefore, a remarkable number of tools have been developed to assist evolutionary code extraction from software repositories. As an example, Zimmermann *et al.* present an extractor which maps changed lines to their containing entity (such as functions) in code [27, 24]. In addition, in a recent research effort, Hassan and Holt [7] developed an evolutionary code extractor tool called C-REX, to recover information from source control repositories in order to study source code evolution. C-REX tracks source code at the change list level and presents the evolutionary change data as an XML file.

The choice of prediction techniques and predictors is case dependant. However, any selected technique and predictor set is required to provide appropriate information with minimal error and the flexibility to refine the configurations. One missing dimension in the prediction models is temporal information. This dimension, particularly for the most change-prone entities, could give another useful viewpoint in planning maintenance tasks.

3. Proposed Framework

As mentioned before, this article deals with predicting "when" software entities will change. For addressing the problem, a change prediction framework based on a neural network is proposed. The framework has been initially introduced briefly in our previous work [1]. This framework, *Neural Network-based Temporal Change Prediction (NNTCP)*, can support change prediction in a flexible and easy-to-tune environment. The high level schema of NNTCP is illustrated in Figure 1. The framework is composed of the following high-level processes:

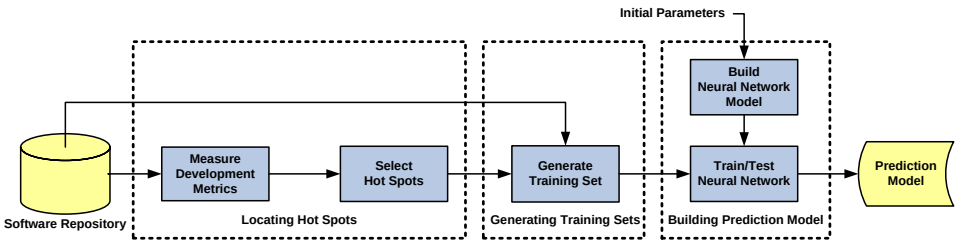

Fig. 1. Neural Network-based Temporal Change Prediction (NNTCP) Schema

- *Locating Hot Spots* - allows for finding change-prone software entities. This helps to focus on change-intensive and critical entities known as hot spots; other software entities are discarded.
- *Generating Training Sets* - processes the hotspots, change logs and prepares the appropriate training data sets.
- *Building Prediction Model* - this process is at the heart of the prediction framework and is responsible for building a neural network prediction model for each hot spot.

The input of the framework is a software repository, which usually contains source code history, change logs, and the release bundles of an application. This provides a source of information for locating hot spots and generating training data sets. The output of the framework is a set of prediction models that can predict the future revision dates of selected hot spots.

This temporal prediction is theoretically applicable for a longer time span in the future, but practically, the prediction error will incrementally increase with time. Therefore, as observed in the experiments, the prediction only provides a limited view of the future. One interesting question is how far the model's prediction can proceed in the time dimension to generate valid predictions. The rest of this section elaborates more on details of the three main constituent components of the proposed framework. Later on, configuring the framework and the quality of obtained results will be discussed in more detail.

3.1. *Locating Hot Spots*

Hot spots are software entities of interest that are used to predict future changes. These entities can be selected by an expert, and are based on various characteristics such as having been recently edited by a new developer, or communication with a new external interface. Thus, those entities that are proven to have a high probability of change are labeled as hot spots.

The probability of change can be measured using a wide range of techniques. These techniques mainly use two sources of information: *software product metrics* (like object-oriented design metrics) of the last software release, and software history data *(software development metrics)* derived from repository logs. Some techniques use only the former or the latter, whereas some approaches benefit from a combination of both.

Development metrics measure the attributes of a development process. In this article, these metrics are mainly used to locate hot spots. The chief reason is that emphasis is placed on the temporal information hidden in these metrics. The product metrics can mark the change-prone entities (hot spots) based on product characteristics such as coupling or complexity, but do not clearly reveal information about the time dimension. On the other hand, development metrics can investigate how the development team changes entities that are not necessarily based on product metrics in all cases. Several research studies have shown the suitability of development metrics as indicators of hot spots, including Khoshgoftar *et al.* [13], [11], [15], and Li *et al.* [17].

3.1.1. *Measuring Development Metrics*

In order to measure development metrics, a tool called *CVS2DB* has been developed. This tool takes a software history log as input, parses the data, performs data conversion, and finally stores it in a relational database. The user can easily calculate the development metrics using a rich set of provided SQL queries.

A set of development metrics introduced in [17], is used as our basic metrics set. However, composite or case-specific development metrics can also be developed using SQL queries and statistical data analysis in addition to the basic metrics. The basic metrics set includes:

- *TotalUpdate:* Total number of updates (revisions) during a development period.
- *BotHalfC:* The number of changes made by inactive developers in a development period. (A developer is considered inactive if she/he is in the lower 50% range of developers. The ranking is based on the number of changes made by each developer.)
- *Difference:* Lines added minus lines deleted from a file during a development period.

In this research, our metric set is narrowed to those which are measurable solely

by Code Versioning Systems (CVS) data. This is due to the fact that the basic elements needed to calculate some of the metrics proposed in [17] depend on human factors or request tracking systems (RTS). For the sake of the generality of the framework and its applicability to those projects which do not have request tracking data, only CVS data is used. It would appear that adding change requests and their assigned priorities would help to improve the quality of prediction models. This issue will be investigated in future research.

3.1.2. *Selecting Hot Spots*

In this framework, hot spots are selected based on their development metrics values. After computing the metrics for each entity, the most likely change-prone entities are selected based on Pareto law. This method was validated by Koru *et al.* for two large-scale case studies including Mozilla, which is also studied in this article [16]. 20% of entities which have 80% of changes are selected as hot spots, and are the focus of the prediction phase.

3.2. *Generating Training Sets*

As discussed before, many valuable information for software change prediction is derived from software change logs. There are some similarities with the weather forecasting process. Tomorrow's weather can be predicted using recent weather changes and the same day's weather in the past. With this kind of problem, a time series is predicted based on retrospective data.

Fortunately, in the case of software change prediction, modern source code versioning systems store the source code along with all corresponding change log histories. For example, CVS stores revisions of each file attached, along with their supporting information. This supporting information includes the author of each file revision, the date and time stamp of the revision, the number of lines added/subtracted from the previous revision, the revision state, and the text description (comment) provided by the author.

The entire retrospective change information of an entity can decrease prediction quality. This is due to the fact that the change rate of the software entities can vary during its lifecycle. There are usually several identical stages in a lifecycle with respect to change rate, and each stage has its own change pattern. In addition, the stages and their corresponding change patterns are unique for each entity.

Although one can think of a prediction model that considers all historical change patterns and predicts the ongoing changes, this model will be too complex and costly to build. Therefore, previous unrelated change patterns are ignored and historical data is narrowed to accommodate recently occurring change patterns and similar patterns that occurred in the past.

As an example, Figure 2 illustrates revision dates of the Hibernate 3 open source

project[a]. The plateau in the middle of the curve, between revisions 5000 and 8500, indicates a rapid change in a short time span in the system. This analysis can assist the user of NNTCP to select the range of revisions for the training set from a revision number greater or equal to 8500. The reason is that abrupt changes from 5000 to 8500 do not indicate the normal rate trend in the given data. Of course if there were several of these sharp changes, for example due to several releases, the prediction model would have learned those patterns. For the Hibernate example, in the case of having adequate revision numbers for the selected hot spot, the user can set the starting point of the prediction range at revision number 11500, in which the most recent stable change rate period begins.

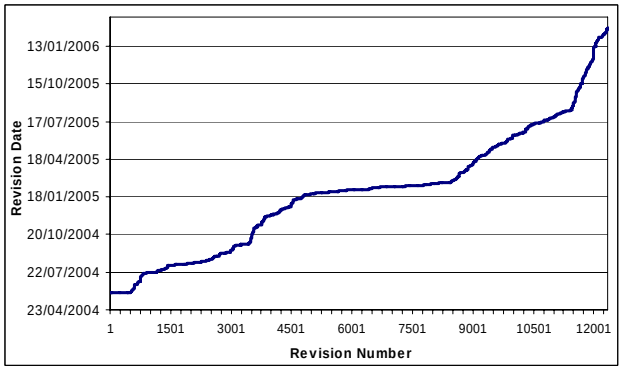

Fig. 2. Change History Trend of Hibernate 3

Now the question is how to set the starting point of the prediction range to cut down the historical data. The most simple and practical answer is to have this point as a parameter that is set by a domain expert. The second approach is to analyze the entire software change pattern anticipating that the change pattern corresponding to any hot spot follows that global pattern. This range is fixed for all hot spots. The age of the latest software release or the average age of all releases are appropriate nominees for the range of training data. The ultimate step for any selected data range is to test the prediction's performance with a set of testing data (cross validation).

3.3. *Building the Prediction Model*

The prediction model of NNTCP is based on artificial neural networks. Neural networks are proven to have acceptable performance as prediction models [8, 20], especially for predicting time-series facts. This make these networks a suitable candidate for our prediction model.

[a]http://www.hibernate.org/

3.3.1. *The Neural Network Model*

The neural network model in NNTCP is called Time-Lagged Feedforward Network (TLFN) [8]. It is a Multi-Layer Perceptron (MLP) with memory components to store past values of the data in the network (illustrated in Figure 3). The memory components allow the network to learn relationships over time. It is the most common temporal supervised neural network, which consists of multiple layers of neurons connected in a feedforward fashion. The number of memory layers, known as prediction length, indicates the number of past and current samples, in order to predict a future sample. The number of memory layers controls the involvement of past information by limiting the recalling time frame.

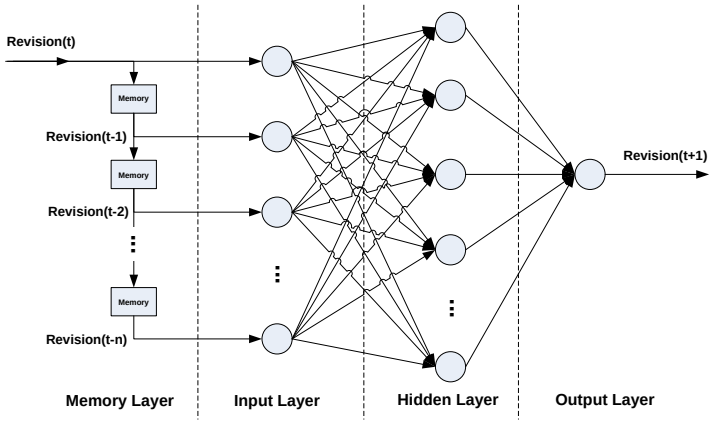

Fig. 3. The Model of Time-Lagged Feedforward Network (TLFN) for Temporal Change Prediction

The training algorithm used with TLFNs is Backpropagation Through Time (BPTT), which is more advanced than the standard Backpropagation [25]. In BPTT, the network runs forward in time until the end of the trajectory, and the activation of each neuron is stored locally in a memory structure for each time step. The output error is then computed, and the error is backpropagated across the network (as in static backpropagation) through time. This error is used to adjust the weights, such that the error decreases with each iteration, and the neural model gets closer and closer to producing the desired output. This process is known as *training*. The error between the network output and the desired output is computed and fed back into the neural network. The neural network uses this error to adjust its weights so that the error is decreased. This sequence of events is usually repeated until an acceptable error is reached, or until the network no longer appears to be learning.

3.3.2. *Initial Parameters*

Besides the network topology and its training algorithm, several other parameters are required to be set for the network. Some of these parameters and their corresponding values in NNTCP are as follows:

- **Activation Function:** The BPTT training algorithm requires differentiable continuous nonlinear activation functions. In this article, sigmoid function is used:

$$o = \sigma(s) = \frac{1}{1 + e^{-s}} \tag{1}$$

  where:

$$s = \sum_{i=0}^{d} w_i x_i \tag{2}$$

  for weights w_i and inputs x_i.
- **Hidden Layers:** The number of hidden layers to be used is not clearly known before modeling. It is suggested to start with a single hidden layer, and if the performance is not satisfactory, the number of hidden layers will be increased.
- **Prediction Length:** To use the current and past inputs to predict future desired outputs, the number of samples is set in order to predict the desired output. In NNTCP, the prediction performance is tested with the prediction length ranging from 2 to 10. In all tests, the best performance is achieved when the prediction length is set to 2, which is the minimum valid value.

3.3.3. *Training and Testing the Network*

There are two major practical aspects related to training the network. Firstly, how to select the training set and its size, and secondly, when to stop the training. Unfortunately, there are no "formulas" to set these parameters. Only some general rules apply and a different number of trials may be required depending on each case.

The size of a training set is of fundamental importance to the practical usefulness of the network. If training patterns do not convey all characteristics of a problem, a discovered mapping from the training session would apply only to the training set. Thus, the evaluated performance for the test set will be worse than the results from the training set. The only general guideline is to have a large enough set of data containing representative data from the whole set. This rule restricts the down-scalability of the approach, so that the network must be trained with an adequate set of training data in terms of the number of revisions.

A percentage of the training data is reserved for cross validation. Cross validation computes the error in a test set at the same time a network is being trained with a training set. Cross validation can be performed in several ways. NNTCP uses holdout validation, in which a fixed set of data is the test set. There is a trade

off between the size of the training set and that of the test set. Usually, 5-15% of the total data set is used for cross validation. However, to achieve the best performance, when there is a big discrepancy between performance in the training and test sets, deficient learning might occur. Note that one can always expect a drop in performance from a training set to a test set. This requires a large drop in performance of more than 10-15%. In cases like this, it is recommended to increase the training set size and/or produce a different mixture of training and test samples as in k-fold or leave-one-out cross validation [8].

The second problem is when to stop the learning process. Stop criteria are all based on monitoring the Mean Square Error (MSE). The curve of the MSE as a function of time is called the learning curve. The most frequently used criterion is a fixed number of iterations, but the final error can also be preset. Since, each of these solutions has its own drawbacks, a compromise is to set a minimum incremental learning value. If between two consecutive iterations, the error is not reduced by at least a given amount, training should be terminated. Another possibility is to monitor the MSE for the test set, as in cross validation. The learning should be stopped when the error in the test set starts to increase. This is where the maximum generalization takes place. To implement this procedure, the network must be trained for a certain number of iterations, the weights frozen, and the test set performance checked. The training process is then continued and the test set will be checked again until an error increase is attained.

4. Experiments

This section presents our empirical experiments, designed to evaluate the proposed framework. The major research questions to be answered through these experiments are as follows:

- **RQ1:** How does the training data range impact the prediction quality?
- **RQ2:** How much effort will be required to build and tune NNTCP?
- **RQ3:** How far in time can NNTCP predict with an acceptable performance?

To answer the above questions, first of all, the threats to validity of our approach are discussed. The experimental setup and the supporting case studies are presented. This is then followed by a discussion on obtained results. Finally, the conclusion reveals answers to the research questions and research contributions are highlighted.

4.1. *Threats to Validity*

In order to analyze the achievements of this study, it is required to describe the construct, internal, and external threats to the validity of this study, and the approaches used to limit the effects of these threats.

Construct Validity The general threats in this facet are mono-operation and mono-method biases. These threats refer to the cases where only one measure, one

case study, or one single method are used for proof of concept. In this article, the temporal changes of two large-scale case studies with NNTCP are predicted. One may raise the question as to why other prediction methods are not considered. The answer is, that although other prediction models (e.g. statistical models) can be used for time series prediction, in dealing with a dynamic prediction model for each hot spot, the manual formulations of those approaches will be difficult. On the other hand, the notable strength of neural networks lies in their ability to represent both linear and non-linear relationships of input-output sets. Neural networks have the ability to learn these relationships directly from data, which is not the case in statistical prediction methods.

Internal Validity There could be several other factors which influence the results and affect our inferences. The tools developed/used to collect data can potentially have defects. Efforts to restrict this threat were dealt with by applying these tools to several case studies. Another noteworthy point is that locating hot spots in the NNTCP framework is based on Pareto law. This assumption would lead to lower quality results in cases that do not completely follow Pareto law. In such cases, one solution would be to use a combination of development and product metrics.

External Validity There are also threats to the external validity of the experiment. The cases being studied in this article are quite large (750k-6000k lines of code), relatively old (6-11 years), and are all open source software. Other industrial software with different characteristics may be subject to different concerns. Finally, the tools used in this study are prototypes, and therefore may not reflect tools used in a typical industrial environment. Controlling most of these threats can be achieved by applying additional empirical studies to various software projects, preferably with different change patterns.

4.2. *Case Studies*

For proof of concept, fairly large-scale case studies with rich CVS history and available source code are needed. In order to choose appropriate case studies, the CVS history of several large-scale open source applications are analyzed. Finally, Mozilla and Eclipse, two open source projects from different application domains, are selected as the case studies. More details on these projects are available in Table 4.2.

Table 1. Case studies

Project Name	Domain	Since	Total LOC	Files	Revisions	Authors
Mozilla	Internet suite	1998	6,272k	111,093	1,033,041	837
Eclipse	IDE	2001	2,272k	38,164	308,555	128

4.3. *Experimental Setup*

According to NNTCP architecture, the prediction process is based on three main stages. In the first stage, the hot spots are identified and selected. In the case studies, this task is performed using the Maximum Likelihood Estimation (MLE) model. The MLE can be treated as a development metric that uses counts from the sequences to estimate the distribution. In the MLE model, the relative frequency of each new event is computed (predicted) based on the preceding sequence. Therefore, the calculated MLE value for each entity is a good indicator of possible hot spots in software projects. The MLE probability distributions is computed using the following formula [23]:

$$P_{MLE}(f_i) = \frac{Count(f_i) + 1}{N + d} \tag{3}$$

where f_i is an entity (e.g. a file) from a domain D ($f_i \in D$), N is the size of an event sequence, $Count(fi)$ is the number of event occurrences of f_i, and d is the size of D. The number of times a certain event (change of f_i) occurs is called the relative frequency of the event.

The entities are sorted by their P_{MLE} values. The top 20% entities with the highest P_{MLE} values will be marked as hot spots based on Pareto law. After detecting the hot spots, a training data set is generated based on the change history (revision data) of each hot spot. The training set can cover the complete revision history or just a selected segment. The length of the history data to train the prediction model is arbitrary, but as previously discussed, enough data is required to successfully train the network, . For performance evaluation, 5% of the recent samples of the training data set are reserved as testing data. In a non-empirical setting, the complete history data should be used as a training data set.

4.4. *Obtained Results*

The quality of prediction results depend heavily on the quality of the training data set. Therefore, before performing experiments, an appropriate range of the revision history should be determined for the prediction task. The range of the training data should be large enough to train the neural network. However, a training data set that is too large may inject unwanted complexity to the prediction framework, which may result in poor network training or overfitting. In the cases presented in the study, each hot spot from the Mozilla and Eclipse projects comprise several hundred revisions. Using a prediction length of 2, and the single hidden layer TLFN (see Figure 3), the size of the training data using all available change history seems appropriate. The prediction results in Table 2 support this argument for the top 5 hot spots of Mozilla and Eclipse. However, to show that this rule of thumb is generalizable, Figure 4 and Figure 5 present the normalized trend of the average change rate in Mozilla and Eclipse for all files versus the selected hot spots. As observed in these two curves, the change rate trends are fairly similar. This issue also validates the Pareto distribution of revision changes in Mozilla and Eclipse.

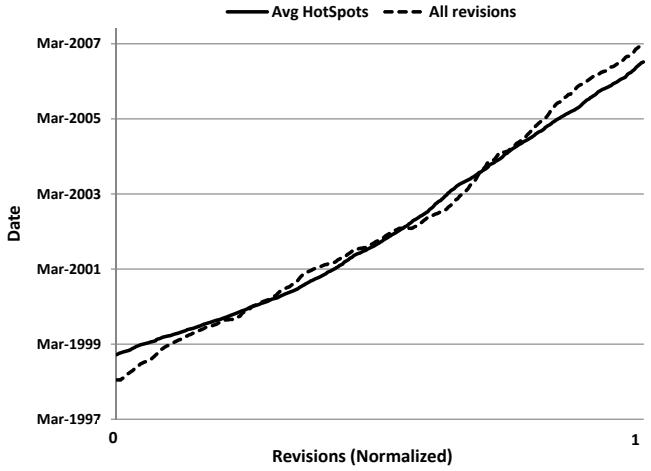

Fig. 4. Mozilla Change Rates for Average Hotspots & All Entities

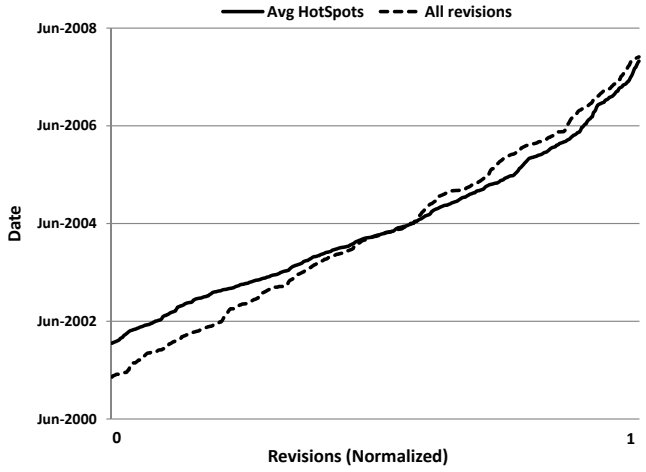

Fig. 5. Eclipse Change Rates for Average Hotspots & All Entities

Table 2. Prediction Results for the Top 5 Hot spots of Mozilla and Eclipse

Case Study	Name	Revision Head	Total Rev.	P_{MLE} (10^{-3})	Train Time (sec)	Pred. Bias	Future Change	Pred. Change	Pred. MSE (Day)	Pred. Next Rev. Err
Mozilla	nsCSSFrameConstructor.cpp	1.1317	1473	2.92	72	13.90	17	36	2.66	4.82
	nsCSSFrameConstructor.cpp	1.1023	1316	2.61	62	26.03	21	36	6.45	1.48
	nsGlobalWindow.cpp	1.911	1273	2.52	61	49.79	13	19	4.33	-0.48
	nsHTMLDocument.cpp	3.713	913	1.81	40	26.23	15	18	8.72	8.63
	nsPresShell.cpp	3.796	962	1.91	46	22.71	18	25	4.40	1.77
	Average	NA	1187.40	2.35	56	27.73	16.80	26.80	5.31	3.24
Eclipse	OS.java	1.483	536	1.74	34	15.10	9	13	7.71	4.74
	Workbench.java	1.447	530	1.72	40	2.34	12	15	6.90	6.63
	Control.java	1.354	382	1.24	38	57.42	3	4	5.66	2.72
	Table.java	1.451	489	1.59	19	43.09	16	19	8.47	1.94
	WorkbenchWindow.java	1.390	478	1.55	54	21.20	5	7	11.30	4.74
	Average	NA	483	1.57	37	27.83	9.00	11.60	8.01	4.15

Table 2 lists the top five detected hot spots of Mozilla and Eclipse by P_{MLE} value. The prediction window is set to one month for all hot spots. This means that the change history data of the last month is reserved as test data. This range of test data enables us to count the number of actual versus predicted future changes. Therefore, this model can be used to predict the change rates in a specific time range. The *Future Change* column of Table 2 shows the actual number of changes for each hot spot using a one-month prediction window. The values of this number are comparable to the values of the *Prediction Change* column, which shows the predicted number of changes for each hot spot. It is observed that in all cases, the predicted number of changes is greater than the actual number of changes.

In this table, the prediction window is set using a short time range, because in trying to predict the change dates over a longer time span, the errors will accumulate after each prediction iteration. This observation is evident as iterative prediction is used, and the prediction is based on the previously predicted values, which may contain errors. To monitor and control these errors *Prediction's MSE* and *Prediction's Next Revision Error* are calculated as seen in Table 2. The *Prediction's MSE* is the average MSE of a predicted change date relative to its actual change date with identical change numbers. The *Prediction's Next Revision Error* is the average error of the next change date predictions. An acceptable prediction error limit can be set by project managers.

The training and testing data have a one-day accuracy. The network training results show that for all hot spots, the training MSE is less than one day accurate. This indicates that the prediction models for all cases were successfully trained. However, the trained time series function does not produce exact values when compared with the actual time series. This results in a shift error between the actual prediction's starting point (i.e. current date) and the prediction model's starting point. In order to fix this shift error, a constant bias of this difference is added to all predicted change dates. The *Prediction Bias* value indicates the perception accuracy of the model from the current date, and does not affect the predicted change pattern.

The prediction results achieved using all available change history data for each hot spot are promising. However, the same prediction performance might be achieved or even improved by using a selective training data set. To study these situations, the prediction models of the most critical hot spots in Mozilla and Eclipse are trained and tested for various data ranges. The results of the change date prediction using several training data ranges are illustrated in Figure 6 and Table 3 for Mozilla's hot spot, and in Figure 7 and Table 4 for Eclipse's hot spot. Similar to previous experiments, the change data of the last month is reserved as test data for evaluation. The desired change prediction trends in Figures 6 and 7 represent this data. The process starts with a full data range using the entire revision history, and then the data range is reduced to half until there is not enough data to train the network. This technique is arbitrary; the limits of the data ranges can coincide with something more meaningful, such as the release dates of the software project.

Table 3. Prediction Results for 'mozilla/layout/base/nsCSSFrameConstructor.cpp' Entity

Training Data Range	Train Data Count	From Rev.	Train MSE (10^{-4})	Pred. Bias	Pred. MSE (Day)	Pred. Next Rev. Err	Valid Pre-dic-tions	% Valid predic-tions
3 Month	54	1.1278	15.35	4.11	16.0	0.03	3	18.75
6 Month	115	1.1110.6.40	6.41	2.98	13.0	0.48	8	50
12 Month	235	1.1186	2.49	2.37	7.0	0.98	8	50
24 Month	405	1.1036.6.2	2.69	0.47	4.7	1.40	15	93.75
48 Month	632	1.824	0.74	3.22	2.5	2.23	16	100
96 Month (All)	1455	1.1	2.50	13.90	2.7	4.82	16	100

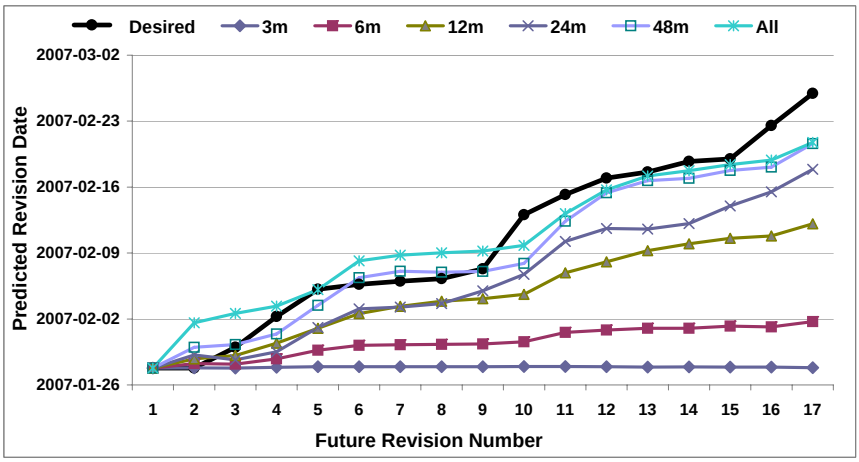

Fig. 6. Prediction Results for Mozilla's Hottest Spot Using Various Training Data Ranges

In this scenario, the objective is to predict the next month's change dates. The best performance is achieved with the training data of all previous revisions for both Mozilla and Eclipse. In the case of the hot spot studied in Mozilla, as the prediction window is shortened from one month to one week, it is observed that the models based on shorter training sets perform better. It is inferred that there is an optimal training data range size that minimizes the prediction error for any given prediction window size and for each entity. Although, as long as the required prediction performance is met, this range size is set to the pre-advised values described in the previous section. In the case of Eclipse's hot spot studied, the results are divided into two distinct categories. For 'All', '12 month', and '24 month' training data ranges, the prediction results are quite similar as observed in Figure 7. Therefore, in this special case, it is concluded that the training data set can be reduced to the

Table 4. Prediction Results for '.../eclipse/ui/internal/Workbench.java' Entity

Training Data Range	Train Data Count	From Rev.	Train MSE (10^{-4})	Pred. Bias (Day)	Pred. MSE (Day)	Pred. Next Rev. Err (Day)	Valid Pre-dic-tions	% Valid predic-tions
3 Month	15	1.399	64.34	7.22	10.7	1.38	6	54.55
6 Month	20	1.394.4.1	14.64	1.99	10.2	2.47	6	54.55
12 Month	55	1.371.2.1	12.85	-1.16	6.2	2.55	7	63.64
24 Month	153	1.283	2.95	-2.23	6.0	2.38	7	63.64
52 Month (All)	495	1.1	1.71	26.65	6.4	1.24	7	63.64

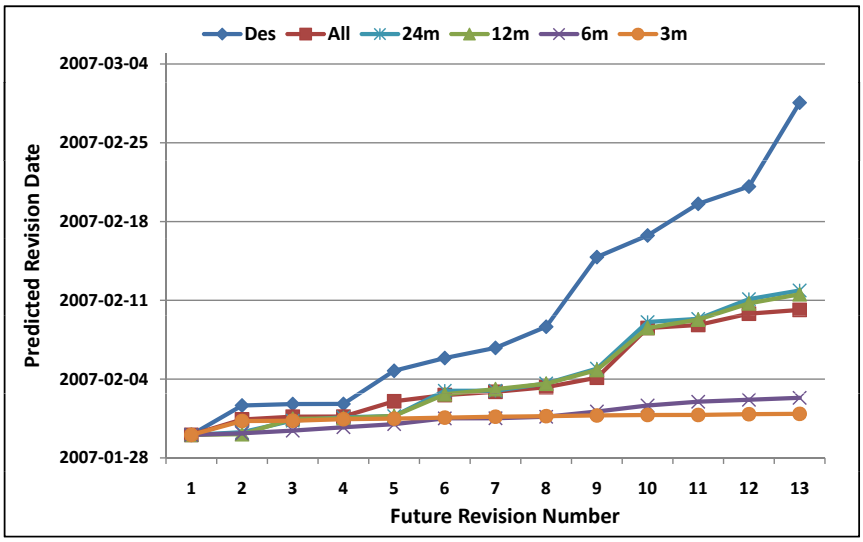

Fig. 7. Prediction Results for Eclipse's Hottest Spot Using Various Training Data Ranges

latest 12 months history data without significantly affecting the prediction's performance. However, this conclusion can not be made for all other Eclipse hot spots since a slight difference in change history trend could highly impact the prediction results.

The calculated number of 'Valid Predictions' and '% Valid predictions' are based on a one-week threshold value. This means that the predicted change is considered to be valid if the predicted revision time is within the same working week as the actual revision. Both case studies show that reducing the training data range decreases the percentage of valid predictions, and the best results are achieved using all of the available training data.

4.5. *Research Contributions*

The main contribution of this work is looking at a new aspect in predicting software changes: "when" an entity is expected to change based on knowing its change log history. This article suggests that this information can help managers plan effectively for maintenance and evolution tasks. At the time of writing this paper and based on the literature review conducted, this issue has not been previously addressed to a great extent in this domain. The proposed framework builds prediction models for selected hot spots (i.e. change-prone entities) of a software system. These models predict future changes for hot spots up to a limited time span.

The proposed research questions relating to the training data, the prediction time span, and the effort required for building the models are discussed as follows:

- RQ1 - *How does the training data range impact the prediction quality?* Although the answer to this question depends on the sample data of a case study, it is acceptable even if some underlying assumptions are not valid. For example one of these assumptions is that change history follows the Pareto distribution.
- RQ2 - *How much effort will be required to build and tune NNTCP?* Based on the data results in Table 2, the training times of the prediction model is not remarkable for a single entity. The building of each model happens only once (based on Figure 3) with different parameters. Therefore, the building time is constant. The most time consuming task in this work was preparing the data set from CVS, because a semi-automatic approach was used. For the case studies in this article, approximately five hours was spent extracting the entire data from change log and preparing data to be fed into the prediction models. Therefore, the training time and even the building time were much less than the data preparation time. Moreover, as the framework suggests, each hot spot has its own prediction model and should be trained with its respective data. The total number of entities selected as hot spots (top 20%) depends on the size of software. However, the total training time for all hot spots can be improved by eliminating the mentioned internal and external threats to validity.
- RQ3 - *How far can NNTCP predict with an acceptable performance?* Based on the results obtained in both case studies, it is found that NNTCP is able to predict several upcoming revisions. However, the prediction error can increase with time. In addition, it is observed that the larger training data set performs better over longer periods in the future. This is due to the fact that the larger time frame selected from the change history of an entity would result in a better trained network with both low and high frequency change patterns. It is noted that the shorter training data set contains only the high frequency change patterns that are solely usable for short-term future predictions.

5. Conclusions and Future Works

This article presented the Neural Network-based Temporal Change Prediction (NNTCP) framework used to predict future change dates of software entities. The

prediction approach was performed on a set of selected hot spots based on change-proneness. This selection process relied on the fact that entities, which have been most recently changed, were considered for potential modifications in the near future.

The proposed framework generates a set of prediction models that augments the time dimension to the information related to hot spots. The framework was evaluated using two large-scale software projects, Mozilla and Eclipse. This is similar to the time-series prediction model used in weather forecasting and stock-market behavior predictions. The predicted temporal information revealed the pattern of changes, which can guide managers to plan better for future maintenance and evolution changes. This would result in more efficient task prioritization and resource allocation.

Similar to the weather and climate forecasts, for short and long-term predictions respectively, different predictors and parameters would be required. The obtained results clearly indicate this phenomenon for short-term and long-term predictions. Another notable point is that a major re-engineering/refactoring stage has a severe impact on the time-based prediction. Therefore, temporally predicted facts, in their current form, would not be useful in comparison with a regular stage of maintenance/evolution. One possible direction for future work is to automatically determine the range of training data based on given samples.

Hot spots can be selected using both development and product metrics. Numerous studies have considered product metrics for measuring the change-proneness of OO software entities. For instance, Koru *et al.* show that size, including lines of code and the number of attributes/methods, have a remarkable impact on change-proneness [16]. One potential extension to this research is filtering the current set of hot spots using product metrics or other development metrics. For example, a Request Tracking System (RTS) provides useful information about the reported bugs and their priority. This information can be refined as a development metric set, which can be used instead of the one employed in this research. This can be particularly useful in cases that do not follow Pareto distribution.

References

[1] Mehdi Amoui, Mazeiar Salehie, and Ladan Tahvildari. Temporal software change prediction using neural networks. In *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE'07)*, pages 380–385, 2007.

[2] Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

[3] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, 2000.

[4] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution: Research articles. *J. Softw. Maint. Evol.*, 18(3):207–236, 2006.

[5] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[6] Ahmed E. Hassan and Richard C. Holt. The top ten list: Dynamic fault prediction. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 263–272, 2005.

[7] Ahmed E. Hassan and Richard C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Softw. Engg.*, 11(3):335–367, 2006.

[8] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 1998.

[9] Xishi Huang, Luiz F. Capretz, Jing Ren, and Danny Ho. A neuro-fuzzy model for software cost estimation. In *Proc. of Int. Conf. on Quality Software (QSIC)*, page 126, 2003.

[10] Standard for software maintenance - IEEE 14764-2006 - ISO/IEC 14764, 2006. URL = http://ieeexplore.ieee.org/iel5/11168/35960/01703974.pdf.

[11] T. M. Khoshgoftaar, R. Shan, and E. B. Allen. Using product, process, and execution metrics to predict fault-prone software modules with classification trees. *hase*, 00:301, 2000.

[12] Taghi M. Khoshgoftaar, Edward B. Allen, K. S. Kalaichelvan, and N. Goel. Predictive modeling of software quality for very large telecom. systems. In *Proc. IEEE Int. Conf. on Communications*, 1996.

[13] Taghi M. Khoshgoftaar, Edward B. Allen, Xiaojing Yuan, Wendell D. Jones, and John P. Hudepohl. Preparing measurements of legacy software for predicting operational faults. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1999. IEEE Computer Society.

[14] Taghi M. Khoshgoftaar, A. S. Pandya, and H. B. More. A neural network approach for predicting software development faults. In *Proc. of Int. Symposium on Software Reliability Eng.*, pages 83–89, 1992.

[15] Taghi M. Khoshgoftaar, Vishal Thaker, and Edward B. Allen. Modeling fault-prone modules of subsystems. *issre*, 00:259, 2000.

[16] A. Gunes; Koru and Hongfang Liu. Identifying and characterizing change-prone classes in two large-scale open-source products. *J. Syst. Softw.*, 80(1):63–73, 2007.

[17] Paul Luo Li, Jim Herbsleb, and Mary Shaw. Finding predictors of field defects for open source software systems in commonly available data sources: A case study of openbsd. In *Proc. of IEEE Int. Software Metrics Symposium (METRICS)*, page 32, 2005.

[18] Audris Mockus, Ping Zhang, and Paul Luo Li. Drivers for customer perceived quality. In *Proc. of Int. Conf. on Software Eng. (ICSE)*, 2005.

[19] George Stephanides Nikolaos Tsantalis, Alexander Chatzigeorgiou. Predicting the probability of change in object-oriented systems. *IEEE Trans. Softw. Eng.*, 31(7):601–614, 2005.

[20] D. Patterson. *Artificial Neural Networks*. Prentice Hall, Singapore, 1996.

[21] S. Ramanna. Rough neural network for software change prediction. In *Proc. of Int. Conf. on Rough Sets and Current Trends in Computing (TSCTC)*, pages 602–609, 2002.

[22] Martin Shepperd and Gada Kadoda. Comparing software prediction techniques using simulation. *IEEE Trans. Softw. Eng.*, 27(11):1014–1022, 2001.

[23] F. Stulajter. *Predictions in Time Series Using Regression Models*. Springer Verlag, 2002.

[24] Peter Weissgerber and Stephan Diehl. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005. Student Member-Thomas

Zimmermann and Member-Andreas Zeller.

[25] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proc. of IEEE*, 78(10):1550–1560, 1990.

[26] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. on Software Eng.*, (6):429–445, 2005.

[27] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). In *Proc. of Int. Workshop on Principles of Software Evolution (IWPSE)*, page 73, 2003.