ESTIMATING HANDLING TIME OF SOFTWARE DEFECTS

George Kour[1], Shaul Strachan[2] and Raz Regev[2]

[1]Hewlett Packard Labs, Guthwirth Park, Technion, Israel
[2]Hewlett Packard Enterprise, Yehud, Israel

ABSTRACT

The problem of accurately predicting handling time for software defects is of great practical importance. However, it is difficult to suggest a practical generic algorithm for such estimates, due in part to the limited information available when opening a defect and the lack of a uniform standard for defect structure. We suggest an algorithm to address these challenges that is implementable over different defect management tools. Our algorithm uses machine learning regression techniques to predict the handling time of defects based on past behaviour of similar defects. The algorithm relies only on a minimal set of assumptions about the structure of the input data. We show how an implementation of this algorithm predicts defect handling time with promising accuracy results.

KEYWORDS

Defects, Bug-fixing time, Effort estimation, Software maintenance, Defect prediction, Data mining

1. INTRODUCTION

It is estimated that between 50% and 80% of the total cost of a software system is spent on fixing defects [1]. Therefore, the ability to accurately estimate the time and effort needed to repair a defect has a profound effect on the reliability, quality and planning of products [2]. There are two methods commonly used to estimate the time needed to fix a defect, the first is manual analysis by a developer and the second is a simple averaging over previously resolved defects. However, while the first method does not scale well for a large number of defects and is subjective, the second method is inaccurate due to over-simplification.

Application Lifecycle Management (ALM) tools are used to manage the lifecycle of application development. Our algorithm relies on a minimal number of implementation details specific to any tool and therefore has general relevance. Our implementation is based on the *Hewlett Packard Enterprise* (HPE) ALM tool. We tested the algorithm with projects from different verticals to verify its broad applicability.

One of the advantages of our implementation is that it does not assume a standard data model, but is able to handle the cases where the defect fields available vary between different data sets. We

Dhinaharan Nagamalai et al. (Eds) : CoSIT, SIGL, AIAPP, CYBI, CRIS, SEC, DMA - 2017
pp. 127– 140, 2017. © CS & IT-CSCP 2017 DOI : 10.5121/csit.2017.70413

provide experimental evidence of the significance of including non-standard fields as input features for the learning algorithm.

Our approach supposes that we can learn from historical data on defects that have similar characteristics to a new defect and use this to make predictions on the defect handling time that are more accurate than many comparative approaches. We found that in real life, the available data was often of poor quality or incomplete, and so in implementing our solution we encountered a number of challenges, which we discuss in this paper.

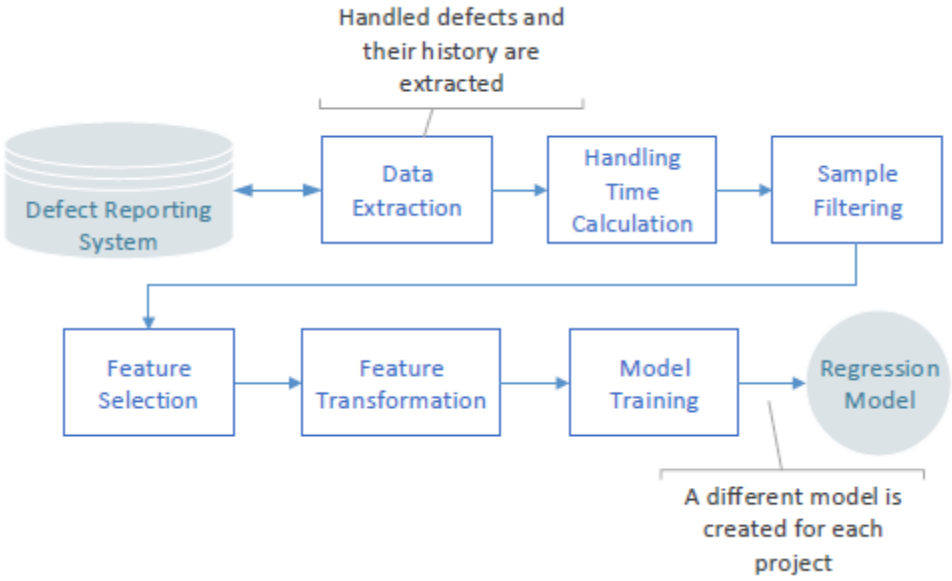

Figure 1: Training process

Our approach is based on a supervised regression machine learning algorithm in which the output is the predicted handling time for defects. The training phase is summarized in Figure 1. It takes as its input (1) a training set of handled defects together with their fields and (2) the history of status changes from these defects, and outputs a learned model. The prediction phase is summarized in Figure 2. It takes as its input (1) the model outputted from the training phase and (2) a set of unhandled defects together with their fields, and outputs the predicted handling time.

2. RELATED WORK

There are many practical methods of defect handling time estimation used in software engineering, as described in [2].

Defect fix effort estimation using neural networks was reported in [3]. The data for this study was extracted from the NASA IV&V Facility Metrics Data Program (MDP) data repository. Their approach is based on clustering the defects using a Kohonen network. Then the known values of defects fix effort were assigned to the found clusters. Given an unseen sample, the fix time is estimated based on the probability distribution of the different clusters.

A text similarity approach for estimating defect resolution time was described in [4]. The title and description of the defects were used to measure similarity between defects using the "Lucene"

engine developed by the Apache foundation [5]. In this study, *k Nearest Neighbours* (k-NN) was used to find the closest k defects already resolved to a given new defect, and the mean was taken as the final estimation.

A work done by Rattiken and Kijsanayothin in [6] investigated several classification algorithms for solving the defect repair time estimation problem. The data set, taken from defect reports during release of a medical record system, contained 1460 defects. They investigated seven representative classification variants of the following algorithm families: Decision Tree Learner, *Naive Bayes Classifier, Neural Networks (NN), kernel-based Support Vector Machine (SVM), Decision Table, and k-NN.*

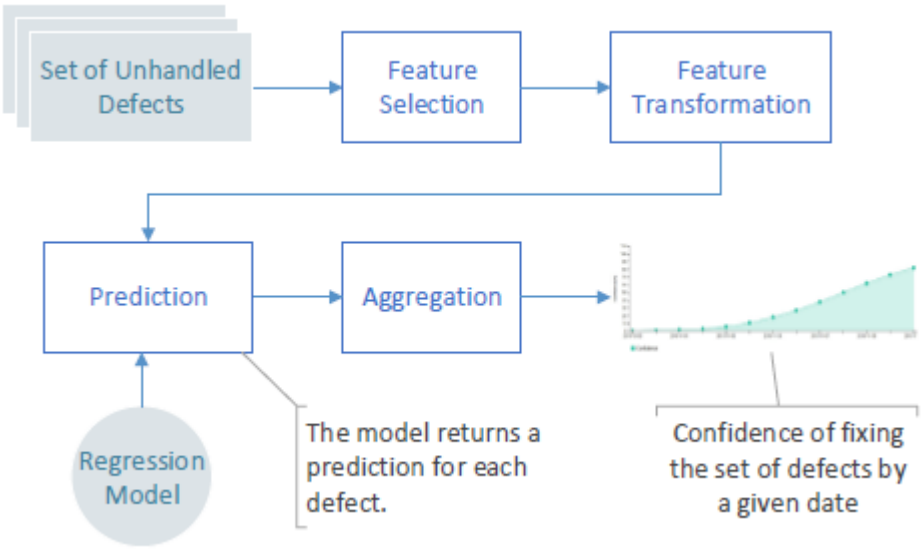

Figure 2: Prediction process

Giger et al. presented a method for predicting the handling time of a new defect using decision trees [7]. The defects in the sample set were classified into two categories: fast and slow defect fixes, based on the median defect fix time of the corresponding project. The data set included 6 open source projects, each containing between 3,500 and 13,500 defects spanning periods between 6 and 11 years.

In a recent work by Zhang et al. in [8], the authors suggested a k-NN classifier for predicting fast or slow defect fixes. A bug was represented as a vector of standard defect attributes and a predefined bag of words extracted from the summary field. They used defect datasets collected from three CA Technologies projects. The results reported in this work is based on a normalized time unit, where 1 unit equals the median time needed to fix a defect.

3. TRAINING

3.1. Data Extraction and Handling Time Calculation

In the first step in the training process, we first extract the defect records, including all their fields and history from the project database. The fields available vary from project to project. Typical

fields include Summary, Description, Assignee, Status, and Severity. However, users generally customize their projects to include user-defined fields. Excluding such fields from the analysis might mean missing valuable features. For example, suppose there is a field such as Estimated Effort, with possible values Low, Medium and High. It seems reasonable that a correlation exists between this field and the actual handling time. Our system does not rely on knowing in advance which fields exist and is able to learn from any number of fields, of various types.We use the history of the Status field to calculate the total time effort invested in fixing a handled defect, refered to as Handling Time. Possible defect statuses vary between projects but generally include New, Open, Fixed, and Closed. For the purposes of our research we define our target value as the total number of days spent in status Open. For the training set, we consider only defects that have reached the end of their lifecycle, i.e. are in state Closed.

In Table 1 we see general information about the projects used in this study. The data sets used for analysis were snapshots of customers' databases.

Table 1: Summary of the projects in the data set

Project	Total # of defects	Observation Period	Industry
1	41,490	Jan. 2010 - Dec. 2013	Banking
2	15,291	Nov. 2004 - May. 2007	Banking
3	3,851	Sep. 2012 - Nov. 2015	Telecom.
4	48,855	Mar. 2001 - Oct. 2006	Software
5	29,425	Sep. 2001 - Dec. 2006	Software
6	2,350	Oct. 2013 - Nov. 2015	Software

3.2. Sample Filtering

We aim to provide managers with a tool to enable better estimation of the time needed to handle defects. Therefore we cannot allow the prediction system to estimate a practically unreasonable time for fixing a defect and we consider it as sampling error. Such a long fixing time can be caused by the fact that in some cases users do not update their current status on working on a defect, and may mark a defect as Open, and after a while switch to another task without updating the defect status.

Although other related work in the field allowed long fixing periods (e.g. more than a month) [7], we observed that it is rare that a defect takes more than 30 days to be fixed and so such defects were filtered out. We encountered defects that remained in state Open for more than a year. Figure 3 shows the accumulative distribution of the defect handling time versus their percentage in the data set. If we allow defects with very large fixing time, the algorithm might find patterns characterizing defects that were "forgotten" in Open state rather than identifying patterns that affect real-life handling time.

In addition, we employ standard extreme value analysis to determine the statistical tails of the underlying distribution using the z-scoring on the handling time field [9]. Samples that are not in the interval $[\mu-3\sigma, \mu+3\sigma]$ are filtered out, where μ is the mean and σ is the standard deviation of all the defects in the data set.

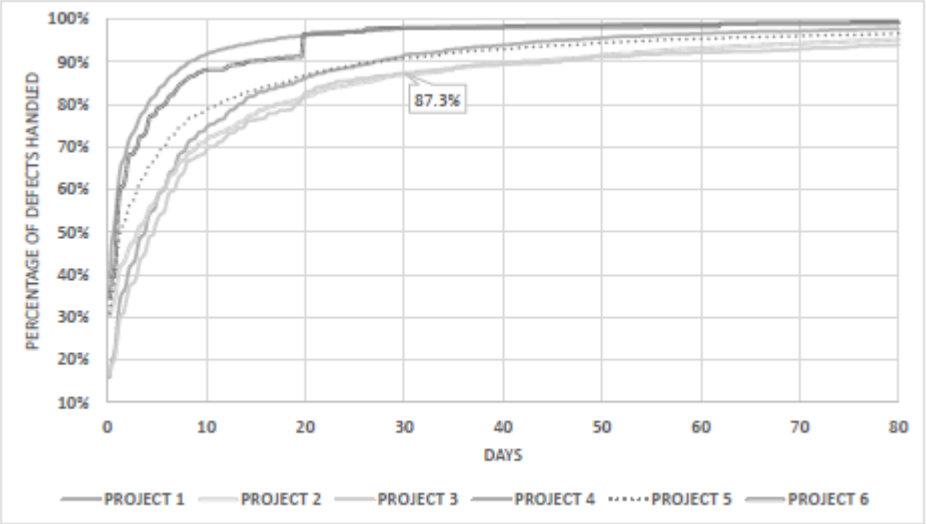

Figure 3: Percentage of defects handled by days

3.3. Feature Selection

Different fields in a data set may have different population rates, where the population rate is the percentage of samples containing a value in a given field. This might be due to the fact that not all fields are required for each defect, or that a field was added to the project at a later stage, and therefore defects opened prior to this field's addition do not have values for the field. To improve the data set stability and the quality of the model, we remove fields with very small population rate (less than 2%). To avoid data leakage, the system automatically filters out attributes that are usually unavailable on defect submission, by comparing the population rate of the attributes in handled and unresolved defects. We also remove non-relevant internal system fields.

3.4. Feature Transformation

A defect is represented by a set of data fields of both qualitative and quantitative data types. However, most machine learning algorithms employ quantitative models in which all variables must have continuous or discrete numerical values. Therefore, certain transformations must sometimes be performed on the given fields before they can be used as features to the learning algorithm.

Numeric fields are generally used as is, directly as features. However, in special cases they are treated as categorical features, as described later in this section.

Categorical fields are transformed using a "one-hot" encoding scheme; a vector of features is assigned to each such field - each feature corresponding to a possible value of that field. For each defect, the feature corresponding to the actual value of the field in that particular defect is assigned the value 1, whereas the rest of the features are given the value 0. We identify a categorical field as a string or number field whose number of distinct values is sufficiently lower than the number of defects with this field populated. To avoid there being too many features ("the curse of dimensionality" [10]) we group the values with low frequencies together under a single value Other, which also captures possible unprecedented values during the prediction process.

String fields whose domain of values is very large, may correspond to free-text fields such as Summary and Description. Although these fields may contain important information, they require additional effort to be appropriately mined and so were discarded in the current implementation.

Date and time fields are projected into time/date cycles such as hour of the week, used to capture the presumed correlation between the day of week for some fields and the handling time. In order to represent this feature radially, we projected each date onto a unit-circle representing the corresponding hour of the week h and took cos(h) and sign(sin(h)), where sign(.) is the signum function, as a new pair of independent variables.

Scarcity of data is a common issue when analysing real-life project defects. Therefore, before starting the training phase, imputation is applied to fill missing values. Empty cells in categorical features are simply treated as another category, as they may have a particular meaning in many cases. However, for numeric features, we filled up empty cells with the most frequent value. In order to avoid data leakage, the imputation is performed just before training the model (after the partitioning to test and train). Replacing empty cells with the mean value of the feature or the median value are two other common numerical imputation schemes which were considered but were empirically found to yield inferior scores in our setting.

3.5. Model Training

Unlike in classification, in which the goal is to identify to which class an unlabelled observation belongs, regression algorithms, given a predictor variable x, and continuous response variable y, try to understand the relationship between x and y, and consequently enable predicting the value of y for a new value of x. The basic idea underlying the Regression Tree learning algorithm is similar to the idea on which the commonly used Decision Tree algorithm is based, but slightly altered to adapt the non-discrete nature of the target field. Random Forests [11] are an ensemble learning method for both classification and regression problems. They operate by building a multitude of decision trees and returning the class that is voted by the majority of the trees in classification, or the mean of the individual trees in regression [10]. Random forests are much less prone to overfitting to their training set compared to decision trees, and so we chose to use them as our machine learning algorithm.

We trained a random forest regression in Python using 80 estimators (i.e. individual decision trees), with a limitation on the minimum number of samples required to split internal node set to 2, and minimum number of samples in a newly created leaf set to 6. The default values were used for the other model parameters. These parameters are constant for all data sets in this work and they were empirically tuned.

4. PREDICTION

While estimating the handling time of a single defect is important, in reality managers are usually interested in making a prediction based on a set of defects (such as the content for a future release) to be handled by a group of developers. Therefore, we have built a system that provides a completion time estimation for any given set of unhandled defects.

As shown in Figure 4, the system presents a graph showing the resulting cumulative distribution in a report which displays the confidence of the defects being closed by any selected date. Based

on the graph, we let the user calculate the release end date given confidence level and vice versa. We also let the user set the number of available developers.

Figure 2 describes the flow for predicting the completion time for a set of defects. Each defect in the given set of unhandled defects passes through the same preprocessing flow as in the learning process, apart from the handling time calculation stage. Then, using the model, the system returns an estimation of the handling time of the given defect, as well as a prediction confidence calculated based on the variance between the answers of the individual regression trees.

To automatically calculate the total time it takes to complete a set of defects by several developers, one would ideally use the optimal scheduling which minimizes the makespan (i.e. the total length of the schedule). However, the problem of finding the optimal makespan in this setup, better known as the Minimum Makespan Scheduling on Identical Machines, is known to be NP-Hard [12]. We employ a polynomial-time approximation scheme (PTAS) called List Scheduling, utilizing the Longest Processing Time rule (LPT). We start by sorting the defects by non-increasing processing time estimation and then iteratively assign the next defect in the list to a developer with current smallest load.

An approximation algorithm for a minimization problem is said to have a performance guarantee p, if it always delivers a solution with objective function value at most p times the optimum value. A tight bound on the performance guarantee of any PTAS for this problem in the deterministic case, was proved by Kawaguchi and Kyan, to be $\frac{1+\sqrt{2}}{2}$ [13]. Graham proved a relatively satisfying performance guarantee of 4/3 for LPT [14].

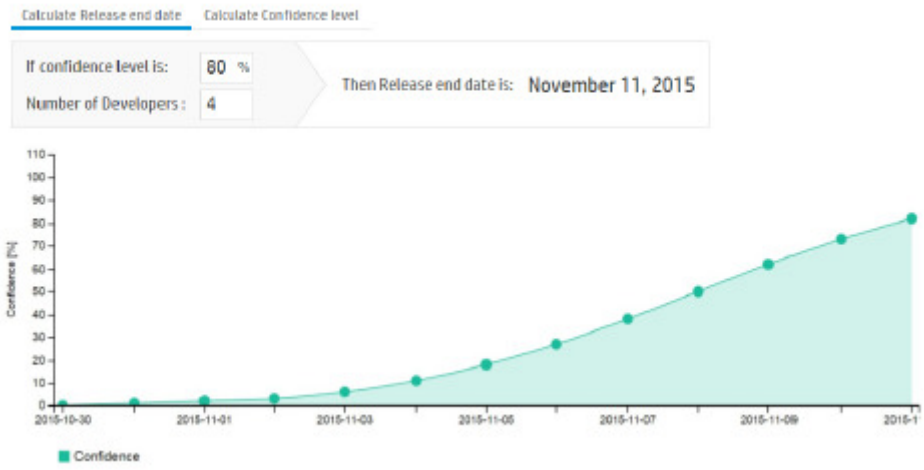

Figure 4: A screenshot of the actual system report.

After computing the scheduling scheme, we end up with a list of handling times corresponding to developers. The desired makespan is the maximum of these times.

Accounting for the codependency between defects is infeasible since it requires quantification of factors such as developers' expertise levels and current availability. It is also subject to significant variance even within a given project. In practice, we observed that treating defects as independent for the sake of this calculation yields very reasonable estimates. By the assumption that defects

are independent, we may treat the completion times as pairwise independent, which yields the Cumulative Distribution Function (CDF) F(t). This value is the probability that the set of defects will be completed until time t.

$$F(t) = Prob(\max_i C_i \leq t) = \prod_i Prob(C_i \leq t) \quad (1)$$

Where C_i is the completion time for developer i. C_i's distribution is not obvious. It is the sum of several independent random variables - a variable whose Probability Density Function (PDF) can be generally computed by convoluting the PDFs of its constituents. However, this requires knowing the distribution of each defect's handling time.

Zhang et al [8] found that the distributions best describing defect handling times are skewed distributions, such as the Weibull and Lognormal distributions. Therefore, we at first used to take the mean and variance values outputted by the predictor, fit a Weibull distribution corresponding to these values, and then apply convolution to achieve the required measures. Our results showed that, in most cases, the distributions of single defects highly resembled the Normal distribution. Moreover, convoluting their PDFs proved to converge very quickly to the Normal distribution, as the Central Limit Theorem guarantees. For these reasons, and to allow fast computation, we simplified the aggregation such that each variable was treated as a Normal random variable. The sum of such variables is also normally distributed and can be easily calculated by a closed formula.

Given F(t), the expected value and variance of the total handling time can be derived using standard techniques.

5. EXPERIMENTAL METHODOLOGY

To facilitate comparison with related work, which mostly discuss handling time for individual defects, and due to the more accurate historical information available for such cases, we focused on the prediction accuracy for handling a single defect. To evaluate the quality of our predictive model, we use the six customer projects introduced in Table 1. After extracting a data set of defects and applying the preprocessing described in Section , we randomly partition the sample set into learning and testing sets, containing 80% and 20% of the data respectively.

The total number of defects in the sample sets of each project is shown in Table 2. The training set is used to construct the model. We use the standard accuracy estimation method, n-fold cross-validation [15] to avoid overfitting.

We employ several performance metrics to evaluate our algorithm. First, we use the *Coefficient of Determination* (denoted R^2) which is a key indicator of regression analysis. Given a set of observations $\{y_i\}_{i=1}^n$ with average value $\bar{y}$, in which each item y_i corresponds to a prediction p_i, R^2 is defined as follows:

$$R^2 = 1 - \frac{\sum_i (y_i - p_i)^2}{\sum_i (y_i - \bar{y})^2} \quad (2)$$

An R^2 of 1 indicates that the prediction perfectly fits the data, while $R^2 = 0$ indicates that the model performs as well as the naive predictor based on the mean value of the sample set. Equation 2 can yield negative values for R^2 when fitting a non-linear model, in cases when the mean of the data provides a better prediction than the model.

Second, we employ the Root Mean Square Error (RMSE), an accuracy metric measuring how well the proposed model fits the data by averaging the distance between the values predicted by a model and the ones actually observed. Last, we utilize a metric proposed in [4], calculating the percentage of predictions that lie within $\pm x\%$ of the actual value y_i. Let e_i denote the absolute difference between the predicted value p_i and the actual value y_i, i.e. $e_i = |y_i - p_i|$.

Pred(x) is then defined as follows:

$$Pred(x) = \frac{|\{i|e_i/y_i < x/100\}|}{n} \quad (3)$$

6. EXPERIMENTAL RESULTS

We performed the following experiments according to the methodology described above on each project independently. The target field, namely the handling time, is calculated in days. In Table 2, we present a general overview of the system performance on the six projects in our data set. For each project the sample size (S. Size) and several accuracy metrics are given. We see that in Project 2 a high value of R2 was achieved, and in Projects 3 and 6, our implementation cut the mean square error by half, compared to the naive algorithm. Comparing our results to the work done in [4], our approach achieved better results, in both Pred(25) and Pred(50), in all of the projects in the data set. In the corresponding work less than 30% of the predictions lie within 50% range of the actual effort on average, whereas our results show Pred(50) = 0:42, a 40% improvement. We see a similar improvement in Pred(25). It is important to mention that the data sample sets' sizes in this study are significantly larger than the projects used in [4]. These results demonstrate that fields other than the Summary and Description should be considered when predicting how long it will take to fix a defect.

Table 2: Summary of performance results

Project	S. Size	R^2	RMSE	$Pred(25)$	$Pred(50)$
1	2659	0.27	4.815	0.195	0.39
2	4000	0.605	4.045	0.405	0.665
3	1088	0.48	4.44	0.32	0.525
4	4000	0.26	2.895	0.16	0.315
5	3794	0.205	4.835	0.145	0.295
6	493	0.48	3.95	0.18	0.325

In Figure 5 we see the learning curve of the projects in our data set. In this particular graph each experiment was conducted three times, i.e. each point represents three experiments done with a given project and a given sample set size, in addition to the cross-validation done in each experiment. We can see that the results are stable when the data set contains more than 1000

defects. Projects 1, 2 and 3 show high values of R^2 for data sets containing more than 500 defects. The differences in results between the projects should be further investigated.

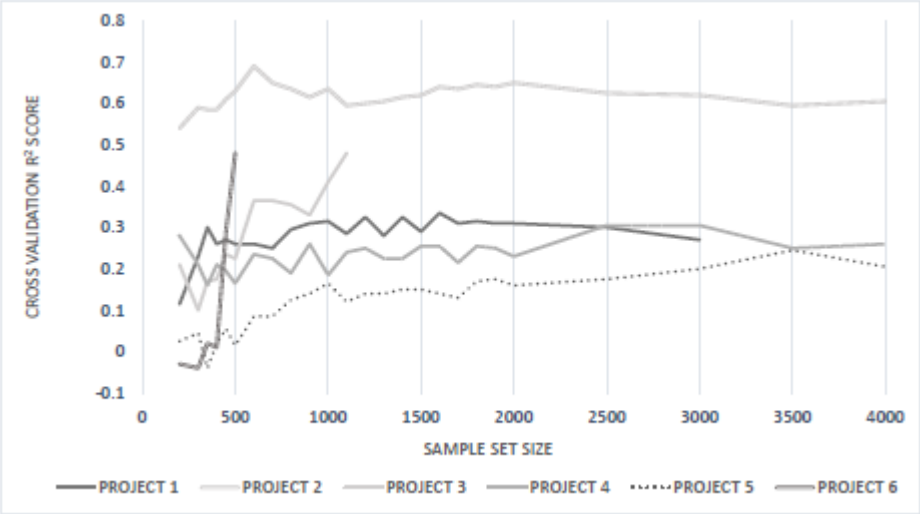

Figure 5: Learning Curve - R^2 as a function of the sample set size

In Figure 6 we compared five sets of data fields for each of the projects used in our experiments. The first set, Base Fields, is a set comprised of the following basic fields: *Detector, Assignee, Priority, Severity and Project Name.* The next couple of field-sets were based upon the base set with additional fields specified in the legend of Figure 6; the fourth set included all standard fields, common between all projects; and the last set contained all fields, including user-defined fields, which vary significantly between different projects. As shown in Figure 6, R^2 scores increased in correspondence with increases in the number of fields available for learning, and particularly when the system was allowed to use user-defined fields. It is important to note that any field may or may not be discarded by the algorithm, either in the learning phase itself or during pre-processing.

While analyzing the results and comparing the different projects, we also examined the feature importances (i.e. the weight of each feature computed by the model). This examination showed that user-defined fields played a crucial role in determining the amount of time to fix a defect. In all projects, at least one such field was ranked as one of the three leading features. We also found that the date a defect last entered an Open state is significant for the prediction. This result supports the assumption mentioned briefly in Section , where we explained how the time a defect was opened or detected may be important.

Examining feature importances also showed that fields which are intuitively thought as significant in handling time prediction of defects, are not necessarily so. Fields such as Severity, Priority, Assignee and Detector were not found to be noticeably important in most projects. These results support those described in [16].

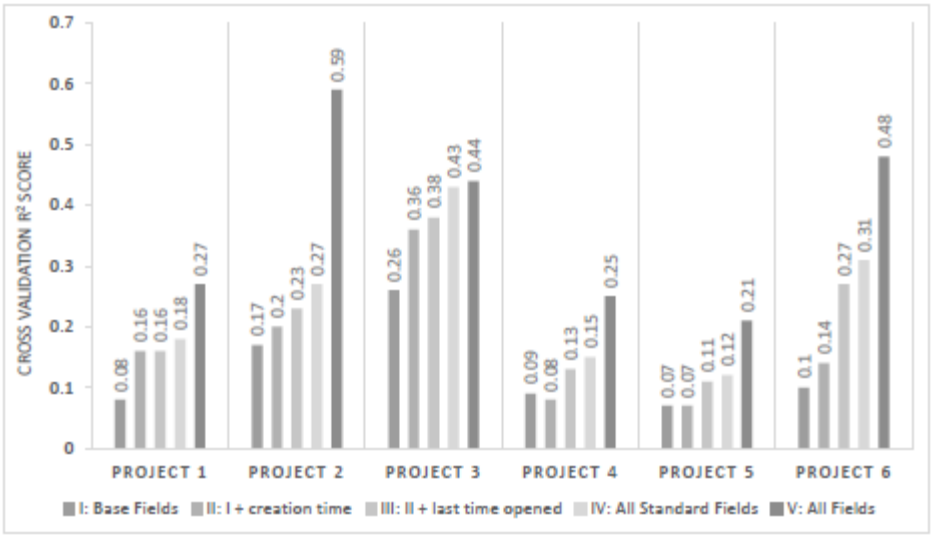

Figure 6: R^2 score as a function of the fields used for learning features

To accurately compare our results to the work done in [7], we ran an experiment in which the target field was calculated in hours and the defects were categorized into Fast and Slow defects using the median of fixing time. Similarly, in this experiment we calculated the median for each project and partitioned the defects into two classes, we used the measures described in [7], and used a Random Forest Classifier with parameters close to those used for our regresssor (described in Section 1). The results presented in Table 3 are the average over all projects in our dataset, and the average results over all projects presented in the compared work when the initial defect data was used. Their data set contained six open source projects, with a similar number of defects to the current paper. Our results show an improvement over the reported results in the compared study.

Table 3: Performance comparison

	Precision	Recall	AUC
Current	0.742	0.668	0.795
Giger et. al [7]	0.635	0.657	0.702

7. VALIDITY

In this section we discuss the validity of our work by addressing the threats for validity of software engineering research proposed by Yin in [17].

Construct Validity. Our construct validity threats are mainly due to inaccuracies in calculating handling times, based on information stored in defect tracking systems. This is due to the "human factor": developers are expected to update the systems, often manually, to reflect the real work process, and have different expertise levels and variable work availability and productivity.

Internal Validity. Methods such as cross-validation were used to make sure the results are as accurate as possible. In cases where a phenomenon had alternative explanations (e.g. comparison

between results), we tried to incorporate the uncertainties to explain the variability and we are not aware of any other factors that could bias the results.

External Validity. Six different projects from different companies and several industries were examined in this study. Our approach can be applied to almost any data set of defects, with a wide range of diversity. However, a possible threat is that the data sets in this study were all taken from the same implementation of ALM. Further studies on different systems are desirable to verify our findings.

Reliability Validity. The dataset used in the work is commercial so cannot be publicly accessible for the moment and therefore this work cannot be replicated by other researchers using the exact dataset. However, we made a significant effort to provide all relevant implementation details.

8. SUMMARY

In this paper, we presented a novel approach for prediction of software defect handling time by applying data mining techniques on historical defects. We designed and implemented a generic system that extracts defect information, applies preprocessing, and uses a random forest regression algorithm to train a prediction model. We applied our method on six projects of different customers from different industries, with promising results. Our system was designed to handle flaws in the data sets common in real-life scenarios, such as missing and noisy data.

9. FUTURE WORK

We currently do not sufficiently exploit the content of the free-text fields. We would like to use text mining techniques to extract key terms that affect the defect handling time.

We are also considering an expanded approach based on state transition models, in which we calculate the probability of a defect transitioning between any given pair of states during a certain time period. A similar idea was described in [18] but we want to expand this idea by computing a separate model for each defect, based on its fields, rather than assuming that all defects behave identically. This could be used to make a large number of predictions about defect life-cycle, for example, to predict how many defects will be reopened. Combining this with methods used for defect injection rates, such as those surveyed in [19], may provide a more realistic prediction for the situation in which new defects are detected within the project time-frame.

Our algorithm can be easily generalized to other entities representing work to be done, such as product requirements and production incidents, and we would like to evaluate its accuracy in these cases. We plan to also generalize our system to extract data from different defect reporting systems.

To make our data publicly available to be used by related research, we plan to obfuscate our data set by removing any identifiable or private information and publish it.

REFERENCES

[1] B. Boehm and V. R. Basili, \Software defect reduction top 10 list," Foundations of empirical software engineering: the legacy of Victor R. Basili, vol. 426, 2005.

[2] S. McConnell, Software estimation: demystifying the black art. Microsoft press, 2006.

[3] H. Zeng and D. Rine, \Estimation of software defects fix effort using neural networks," in Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International, vol. 2, pp. 20{21, IEEE, 2004.

[4] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, \How long will it take to fix this bug?," in Proceedings of the Fourth International Workshop on Mining Software Repositories, p. 1, IEEE Computer Society, 2007.

[5] E. Hatcher, O. Gospodnetic, and M. McCandless, \Lucene in action," 2004.

[6] R. Hewett and P. Kijsanayothin, \On modeling software defect repair time," Empirical Software Engineering, vol. 14, no. 2, pp. 165{186, 2009.

[7] E. Giger, M. Pinzger, and H. Gall, \Predicting the fix time of bugs," in Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, pp. 52{56, ACM, 2010.

[8] H. Zhang, L. Gong, and S. Versteeg, \Predicting bug-fixing time: an empirical study of commercial software projects," in Proceedings of the 2013 International Conference on Software Engineering, pp. 1042{1051, IEEE Press, 2013.

[9] L. Davies and U. Gather, \The identification of multiple outliers," Journal of the American Statistical Association, vol. 88, no. 423, pp. 782{792, 1993.

[10] J. Friedman, T. Hastie, and R. Tibshirani, The elements of statistical learning, vol. 1. Springer series in statistics Springer, Berlin, 2001.

[11] L. Breiman, \Random forests," Machine learning, vol. 45, no. 1, pp. 5{32, 2001.

[12] D. S. Hochbaum and D. B. Shmoys, \Using dual approximation algorithms for scheduling problems theoretical and practical results," Journal of the ACM (JACM), vol. 34, no. 1, pp. 144{162, 1987.

[13] T. Kawaguchi and S. Kyan, \Worst case bound of an lrf schedule for the mean weighted ow-time problem," SIAM Journal on Computing, vol. 15, no. 4, pp. 1119{1129, 1986.

[14] R. L. Graham, \Bounds on multiprocessing timing anomalies," SIAM journal on Applied Mathematics, vol. 17, no. 2, pp. 416{429, 1969.

[15] R. Kohavi et al., \A study of cross-validation and bootstrap for accuracy estimation and model selection," in Ijcai, vol. 14, pp. 1137{1145, 1995.

[16] P. Bhattacharya and I. Neamtiu, \Bug-fix time prediction models: can we do better?," in Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 207{210, ACM, 2011.

[17] R. K. Yin, Case study research: Design and methods. Sage publications, 2013.

[18] J. Wang and H. Zhang, \Predicting defect numbers based on defect state transition models," in Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, pp. 191{200, ACM, 2012.

[19] N. E. Fenton and M. Neil, \A critique of software defect prediction models," Software Engineering, IEEE Transactions on, vol. 25, no. 5, pp. 675{689, 1999.