Practical Considerations in Deploying AI: A Case Study within the Turkish Telecommunications Industry

Ayşe Tosun[1], Burak Turhan[1], Ayşe Bener[1], Tim Menzies[2]
ayse.tosun@boun.edu.tr, turhanb@boun.edu.tr, bener@boun.edu.tr, tim@menzies.us
[1]*Softlab, Department of Computer Engineering, Bogazici University, Turkey*
[2]*Lane Department of Computer Science and Electrical Engineering, West Virginia University*

Abstract

We have conducted a study in a large telecommunication company in Turkey to employ a software measurement program and to predict pre-release defects. We have previously built such predictors using AI techniques. This project is a transfer of our research experience into a real life setting to solve a specific problem for Trcll: to improve code quality by predicting pre-release defects and efficiently allocating testing resources. Our results in this project have many practical implications that managers have started benefiting: using version history information, developers only need to inspect 31% of the code to find around 84% of the defects with 29% false alarms, compared to 60% inspection effort with 46% false alarms without using the historical data. In this paper we also shared in detail our experience in terms of the project steps (i.e. challenges and opportunities).

1. Introduction

Telecommunications is a booming industry in Turkey and its neighbouring countries. For example, Trcll operates in Azerbaijan, Kazakhstan, Georgia, Northern Cyprus and Ukraine with a customer base of 53,4 million. This booming industry is highly competitive. Trcll is under constant pressure to launch new and better campaigns in limited amount of time with tight budgets. Accordingly, the company seeks to lever any opportunity for improving their software construction.

Previously [1], we have implemented extensive automated defect prediction programs at NASA using data mining. In this paper, we describe our experience in translating those programs to Turkey. In summary, while Turkish software development practices are very different to NASA, the automated AI methods ported with relatively little effort. Also, we found we could reuse some of the NASA data to enhance defect predictors on Turkish software. However, in order to achieve this goal, we spent much time learning the organization and working in with their current practices. Our conclusion is simple: AI is easy, but integrating that AI with an organization requires much effort.

In this respect, we present our experience on the practical applications of AI techniques to solve the problems of Trcll in their software development processes. The results of our study showed that we can perceive the benefits of a learning-based defect predictor on the allocation of resources, bug tracing and measurement in a large software system. Using code metrics and version history, we managed to keep high stable detection rates, on the average 84%, up to 100%, while decreasing false alarms from 46% to 29% in sample of experiments. Those improvements in the classification accuracy of our model have also affected the inspection costs to detect defective modules. Our cost-benefit analysis presented, on the average, 50% improvement in terms of LOC, which is required for detecting defective modules in projects of Trcll.

2. Goals of the Project

Trcll has grown very rapidly and successfully since its inception in 1994. Their software systems have millions of lines of code that needs to be maintained. As the technology changes and the customers require new functionalities, they keep developing code faster than ever. Similar to other software development companies, testing is one of the most critical stages in their development cycle [2, 14, 15].

This project in Trcll aimed at measuring software artifacts, managing defect rates as well as the testing phase more efficiently and cost effectively.

Previously, software development team did not measure source code attributes such as LOC, complexity, etc. or store any version history with defect

data. Developers did not record any defect information during unit testing due to tight schedule and/ or other priorities in their work load. However, senior management and the project managers strongly believed that they should change their development processes to increase their software quality as well as to make efficient resource allocation.

Our project kick-off meeting was attended by the R&D manager, project managers and development team (both coders and testers). In this meeting, we jointly agreed on the goals of the project and we decided on the roles and responsibilities. We clearly explained to them that, at the end of the project, what they will have and what they will not have. We all agreed that we would build a *code measurement, bug tracing/ matching program and a defect prediction model*. We also aligned the goals of the project with Trcll's business goals as seen in Table 1. In order to track the progress of the project, we decided to make monthly meetings with the project team, and quarterly meetings with the senior management to present the progress and discuss on the next step. Senior management meetings were quite important either to escalate the problems or to get their blessing on some critical decisions we had to make throughout the project. During the life of the project Softlab researchers were on-site on a weekly basis to work together with the coders, testers and quality teams.

Table 1. Goals of the project in line with company objectives

Goals of the project	Management objectives
Construction of a defect prediction model to predict defect prone modules before testing phase.	Decrease their lifecycle costs such as testing effort.
Code measurement and analysis of the software system.	Improve code quality
Guiding developers with a predictor through defect prone parts to fix them before testing phase.	Decrease defect rates
Storing a version history and bug data from completed versions.	Measure/ control the time to repair the defects

Initially, we have planned our work in four main phases. In the first phase, we aimed to measure static code attributes at functional/ method level from their source code. In the second phase, we planned to match those methods with pre-release defects. In the third and fourth phases, we planned to build and calibrate a defect prediction model assuming that we would have collected enough data to train our model. However, the outcomes of every phase have led us to re-define and extend the original scope and objectives in the later stages. We will explain what happened at the beginning, what was unsatisfactory about our plan and our proposed solutions for these problems in detail in the next sections.

3. Phase I: Code Measurement and Analysis

3.1. How Things Went at the Beginning

In the first three months of the project, we aimed to analyze the company's coding practices and to conduct a literature survey of measurement and defect prediction in the telecommunications industry. At the end of this phase, we aimed at having agreed on the list of static code attributes and to decide on an automatic tool to collect them. We also hoped to have collected the first set of static code attributes from the source code in order to make a raw code analysis. In order to do that, we would choose the sample projects that we would collect data from.

Initially, we decided that static code attributes, such as McCabe, Halstead and LOC metrics [4, 5], can be easily used to point out current coding practices, since they can be easily collected through automated tools [1, 3]. Static code attributes are also accepted as reliable indicators of defective modules in the software systems, and they are widely used in many defect prediction studies [1, 3, 6, 8, 15, 16]. Therefore, we have defined the set of static code attributes from NASA MDP Repository [12] as the metrics that needs to be collected from the software in Trcll.

3.2. What Went Wrong During Phase I

Trcll did not have a process or an automated tool to measure code attributes. Our suggestion was to buy a commercially available automated tool to extract metrics information easily and quickly. However, due to budget constraints and concerns for adequacy of functionalities of the existing tools, senior management did not want to make an investment on such a tool. Besides, Trcll's software systems contain source codes and scripts written in different languages such as Java, Jsp, PL/SQL. Therefore the management was not convinced to find a cost effective single tool that would

embrace all languages and extract similar attributes easily from all of them.

Trcll has the standard 3-tier architecture with presentation, application and data layers. However, the content in these layers can not be separated as distinct projects. Any enhancement to the existing software somehow touches all or some of the layers at the same time, making it difficult to identify code ownership as well as to define distinct software projects.

Another problem we have come across was related with data collection process. When we observed the software development process with coding practices of the development team, it seemed that collecting static code attributes in the same manner with NASA datasets, i.e. in functional method level, is almost impossible due to lack of an available automated mechanisms that would match those attributes with the defect data.

3.3. How The Team Changed To Repair This

Metric Extraction: In Softlab, we have developed a new metrics extraction tool, *Prest* [9]. Different than available commercial products, Prest is able to parse C, C++, Java, Jsp projects and extract static code attributes at package/ file/ functional method level. It can also form a dependency matrix based on caller-callee relations between modules to examine the complexity and design of software systems. Currently, we are extending this tool to collect PL/ SQL metrics as well. Using Prest, we have collected 29 static code attributes [1, 12] from 25 critical projects.

Project Selection: This was one of our first critical decisions in this project: to define the scope and to choose projects and/ or units of production that would be under study. Although project managers initially wanted to focus on presentation and application layers due to the complexity of the architecture, we jointly decided to take 25 critical and highly interconnected Java applications embedded in these layers. We refer to them as projects during this study.

Level of granularity: As we mentioned in the previous section, we were unable to use method-level metrics in Trcll software system, since we could not collect method-level defect data from the developers in such limited amount of time. Therefore, we have also converted our method-level attributes into the file-level to make them compatible with defect data.

4. Phase II: Bug Tracing/ Matching

4.1. How Things Went At the Beginning

The second phase of our study was originally planned to store defect data, i.e. test and production defects. If things had their normal course of action in Phase I, we would have collected metrics from the completed versions of 25 projects and matched the bugs as well.

In this phase, we have completed the coding of our Java parser in the Prest metric collection tool. We were able to extract static code attributes from 25 projects. We have pointed out some problems in the coding practices of the company's development team. We have taken best practice coding standards of NASA MDP Repository [12] and compared them with Trcll's measurement. Based on our analysis, we have seen that there are two fundamental issues in the coding practices:

- Developers rarely write comments to the code. We suggested increasing the ratio of comments to code to make the source code easy to read and understand by other developers.
- The number of operators and operands used in the applications, i.e. vocabulary in the software, is very limited. This brings a conclusion that the system is designed to be highly modular. This may also be due to the fact that this code mostly comes from the application layer, and hence, many Jsp codes make it seem too modular.

Moreover, as an alternative analysis, we have also conducted a rule-based code review process based on only static code attributes. Our aim here was to decide what amount of code should be reviewed and how much testing effort is needed to inspect defect-prone modules [10].

In the rule-based review, we have simply defined rules for each attribute, based on its recommended minimum and maximum values [10]. These rules are fired, if a module's selected attribute is not in the specified interval. This also indicates that the module could be defect-prone, therefore, it should be manually inspected. The results of the rule-based model can be seen in Figure 1, where there are 17 basic rules with corresponding attributes and two additional rules derived from all of the attributes. Rule #18 is fired if any of 17 rules is fired. This rule shows that we need to inspect 100% LOC to find defect-prone modules of the overall system. Besides, Rule #19 is fired if all basic rules, but the Halstead rules, are fired. This reduces the firing frequency of the former rule such that 45% of the code (341655 LOC) should be reviewed to detect potentially problematic modules in the software.

Rule No	Metric	Module	%	LOC	%
Rule 1	Intelligent Content	8245	17	507344	66
Rule 2	Maximum Nesting Depth	1307	3	155696	20
Rule 3	Volume	31260	65	345399	45
Rule 4	Total Operators	44117	92	530882	70
Rule 5	Time	143	0	53368	7
Rule 6	Difficulty	83	0	29545	4
Rule 7	Vocabulary	40442	84	444212	58
Rule 8	Effort	1626	3	234039	31
Rule 9	Unique Operands	41699	87	528542	69
Rule 10	Unique Operators	44086	92	464262	61
Rule 11	Total Operands	42774	89	507471	67
Rule 12	Architectural Complexity	1217	3	196641	26
Rule 13	Level	3270	7	28678	4
Rule 14	atio Of Comment To Cod	47062	98	729896	96
Rule 15	Length	525	1	122541	16
Rule 16	Cyclomatic Complexity	1735	4	223773	29
Rule 17	Structural Complexity	1036	2	112470	15
Rule 18	Any	47995	100	763025	100
Rule 19	Any*	6488	14	341655	45

Figure 1. Rule-based analysis [10].

We have seen that rule-based code review process is impractical in the sense that we need to inspect 45% of the code in order to detect, on the average, 70% of defective modules [10]. So, it is obvious that we need more intelligent oracles to decrease testing effort and defect rates in the software system.

4.2. What Went Wrong In Phase II

This phase took much longer than we anticipated. First of all, there was no process for bug tracing. Secondly, test defects were not stored at method level during development activities. Thirdly, there was no process to match bugs.

The project team has realized that the system is very complex such that it needs too much time and effort to match each defect with its file constantly. Additionally, developers did not volunteer to participate in this process, since keeping bug reports would have increased their busy workloads.

4.3. How the Team Changed To Repair This

To solve these issues we called for an emergency meeting with senior management as well as with the heads of development, testing and quality teams.

As a result, Trcll agreed to change their existing code development process. They built a version control log to keep changes in the source code done by the development team. These changes can be either bug fixing or new requirement request, all of which are uniquely numbered in the system. Whenever a developer needs to check out the source code to their version control system, he/ she should provide additional information about "which file has been modified because of which problem", i.e. id of either test defect or requirement request. Then, we would be able to retrieve those defect logs from the history and match them manually with the files of the projects in that version. Since we were able to collect defect data in the source file level, we have also changed our code measurement practice. We have collected static code attributes in the method level and converted them to file level by taking minimum, maximum, average and sum of the methods in each file [17]. Then we classified files as defective or defect-free by assigning 1 or 0 to them. This process change also enabled Trcll to establish code ownership.

5. Phase III: Defect Prediction Modeling

The original plan in this phase was to start constructing our prediction model with the data we had been collecting from Trcll projects. We had planned to test the performance of our model with the ones in the literature. We would try different experimental designs, sampling methods, and AI algorithms to build such a model.

In this phase, we rigorously monitored the bug tracing and matching process together with the quality team. Whenever there was a slow down in the process, we escalated the issues to senior management for them to step in.

5.1. How Things Went At the Beginning

We planned to build a learning-based defect predictor for Trcll. We have agreed to use Naïve Bayes classifier as the learner of this model, since in our earlier research we have shown that Naïve Bayes is a simple algorithm and yet it gives the best results compared to other machine learning methods [1]. We also agreed on the performance measures of the defect predictor we would be building. We decided using three measures: probability of detection, *pd,* probability of false alarm, *pf*, and balance from signal detection theory [13]. *Pd* measures the percentage of defective modules that are correctly classified by the predictor. *Pf,* on the other hand, is a measure to calculate the ratio of defect-free modules that are wrongly classified as defective with our predictor. In the ideal case, we expect to see {100%, 0} for {pd, pf} rates, however, the model trigger more often which has a cost of false alarms [1]. Finally, *balance* indicates how close our estimates to the ideal case by calculating the Euclidean distance between the performance of our model and the point {100, 0}. All of these measures can be easily

calculated from a confusion matrix (Table 2) using the formulas below:

$$pd = A / (A + C)$$
$$pf = B / (B + D)$$
$$bal = 1 - \sqrt{(0 - pf)^2 + (1 - pd)^2} \Big/ \sqrt{2}$$

Table 2. Confusion Matrix

predicted	actual	
	defective	defect free
defective	A	B
defect free	C	D

In order to interpret our results to business managers we also agreed to construct a cost-benefit analysis based on Arisholm and Briand's work [18]. We would simply measure the amount of LOC or the number of modules that our model would predict as defective. Then we would compare this with a random testing strategy to measure how much we gain from testing effort by using our predictor. In a random testing strategy, it is assumed that we need to inspect *K* percent of LOC to detect *K* percent of defective modules [18]. Our aim is to decrease this inspection cost with the help of our predictor.

5.2. What Went Wrong In Phase III

Although we started collecting data from completed versions of the software system, we have realized that constructing such a dataset would take a long time.

We have seen that building a version history is an inconsistent process. Developers could allocate extra time to write all test defects they fixed during the testing phase due to other business priorities. In addition, matching those defects with corresponding files of the software can not be automatically handled. We could not form an effective training set for a long time. Therefore, we were not able to build our defect prediction model.

5.3. How the Team Changed To Repair This

Instead of waiting for a complete dataset, we have come up with an alternative way to move ahead. In our previous research, we had suggested companies like Trcll to build defect predictors with other companies' data which we call cross-company data [6].

5.3.1 Using Cross-Company Data

Cross-company data can be used effectively in the absence of a local data repository, especially when special filtering techniques are used:

- Selecting similar projects from cross-company data using nearest neighbor sampling [8].
- Increasing the information content of data using dependency data between modules [11].

In our study, we have selected NASA projects as the cross-company data. NASA projects [7] contain more than 20.000 modules, of which we used 90% randomly as the training data to predict defective modules in Trcll projects [6]. From this subset, we have selected a subset of projects that are similar to those in Trcll data in terms of Euclidean distance in the 17 dimensional metric spaces [8]. The nearest neighbors in this random subset are used to train a predictor, which then made predictions on the Trcll data. We repeated this procedure 20 times and raised a flag for modules that are estimated as defective at least in 10 trials [8].

Figure 2 shows the results from the first analysis. The estimated defect rate is %15 that is consistent with the rule-based model's estimation. However, there is a major difference between the two models in terms of their practical implications. For the rule-based model, estimated defective LOC corresponds to 45% of the whole code, while module level defect rate is 14%. On the other hand; for the learning-based model, the estimated defective LOC corresponds to only 6% of the code, where module level defect rate is still estimated as 15%.

PROJECT	ESTIMATED DEFECT RATE	ESTIMATED DEFECTIVE LOC	TOTAL LOC	%LOC FOR INSPECTION
Trcll 1	0.05	242	6206	0.04
Trcll 2	0.18	4933	80941	0.06
Trcll 3	0.08	2664	45323	0.06
Trcll 4	0.13	322	5803	0.06
Trcll 5	0.16	3834	53690	0.07
Trcll 6	0.14	193	4526	0.04
Trcll 7	0.28	305	5423	0.06
Trcll 8	0.12	4779	79114	0.06
Trcll 9	0.24	801	10221	0.08
Trcll 10	0.15	2747	61602	0.04
Trcll 11	0.26	140	2485	0.06
Trcll 12	0.12	555	9767	0.06
Trcll 13	0.13	216	5425	0.04
Trcll 14	0.17	196	2965	0.07
Trcll 15	0.09	1568	36280	0.04
Trcll 16	0.32	3108	42431	0.07
Trcll 17	0.09	359	6933	0.05
Trcll 18	0.22	646	10601	0.06
Trcll 19	0.17	393	6258	0.06
Trcll 20	0.06	175	3507	0.05
Trcll 21	0.09	106	1971	0.05
Trcll 22	0.05	9624	215265	0.04
Trcll 23	0.04	1548	51273	0.03
Trcll 24	0.29	627	10135	0.06
Trcll 25	0.11	331	4880	0.07
SUM		40412	763025	
AVG.	0.15			0.06

Figure 2. Results for cross-company analysis: Analysis I [10]

This significant difference is occurred because rule-based model makes decisions based on individual metrics and it has a bias towards more complex and larger modules. On the other hand, learning based model combines all 'signals' from each metric and estimates defects located in smaller modules [1, 10].

We have added one more analysis using cross-company data to increase the information content by adding dependency data between modules of the projects. Our previous research shows that false alarms can be decreased from 30% to 20% using a call graph based ranking framework in embedded software data collected from a white-goods manufacturer [11]. Therefore, we have also calculated caller-callee relations between modules of NASA and Trcll to adjust code metrics with this framework. The results of the second cost-benefit analysis using cross-company data can be seen in Figure 3 for 22 Trcll projects. It is observed that developers need to inspect only 3% of the code to predict 70% of defective modules in their software systems. When we compare this with the rule-based model, we can once more see the benefits of a learning-based model to decrease testing efforts by guiding testers through defective parts of the software. It is important to mention that this analysis was completed in the absence of local data. Therefore, we have used the results to show the tangible benefits of building a defect predictor to the managers and development team in Trcll.

PROJECT	ESTIMATED DEFECT RATE	ESTIMATED DEFECTIVE LOC	TOTAL LOC	%LOC FOR INSPECTION
Trcll 1	0.02	99	6206	0.02
Trcll 2	0.03	1035	45323	0.02
Trcll 3	0.08	163	5803	0.03
Trcll 4	0.06	85.00	4526	0.02
Trcll 5	0.05	1130	53690	0.02
Trcll 6	0.13	138	5423	0.03
Trcll 8	0.18	505	10221	0.05
Trcll 9	0.09	1509	61602	0.02
Trcll 10	0.09	44	2485	0.02
Trcll 11	0.08	303	9767	0.03
Trcll 12	0.08	119	5425	0.02
Trcll 13	0.06	65	2965	0.02
Trcll 14	0.05	746	36280	0.02
Trcll 15	0.18	1476	42431	0.03
Trcll 16	0.04	140	6933	0.02
Trcll 17	0.10	246	10601	0.02
Trcll 18	0.07	137	6258	0.02
Trcll 19	0.03	82	3507	0.02
Trcll 20	0.03	28	1971	0.01
Trcll 21	0.19	369	10135	0.04
Trcll 22	0.07	168	4880	0.03
Trcll 24	0.10	2458.00	80941	0.03
SUM		**8587**	**336432**	
AVG.	**0.08**			**0.03**

Figure 3. Results for cross-company analysis: Analysis II [10]

6. Phase IV: Defect Prediction Model Extended

This phase did not exist in our original plan, since we had underestimated the time and effort that was necessary to build a local data repository.

In this phase, we were able to collect local data for three versions of the software. We have started with a total of 10 projects from three completed versions. Projects vary at each version, since they have been developed or modified according to the content of new campaign at a specific release.

6.1. How Things Went At the Beginning

Different than the first cross-company analysis, we have collected within-company data at file level. We have applied the micro-sampling approach, proposed in our previous study [3], to decide on the ratio of defective vs. defect-free modules in the training set. Based on the results of our previous analysis, we have decided to form a set containing M defective and N defect-free modules, (i.e. M=N) such that the model would not improve with additional data [3]. Then, we have used the training set to predict defective files of the test set, i.e. *n*th version of the selected project. We have also conducted another experiment that treats all projects as one software system at the *(n-1)*th version to build the training set [14]. Then we have estimated the

defective files for a specific project at the *n*th version of the software system [14]. Figure 4 shows a sample of two versions and two projects in our experiment results. It is observed that both of the approaches produced high pd rates in the range between 78% to 100%.

Release number	*Appl. Name*	*1st experiment with 8 appl.*			*2nd experiment with Trcll1 or Trcll2*		
		pd	*pf*	*bal*	*pd*	*pf*	*bal*
2.32	**Trcll1**	100	67	53	85	34	68
	Trcll2	78	75	44	80	66	51
2.33	**Trcll1**	92	51	60	100	36	75
	Trcll2	81	63	45	90	71	44

Figure 4. Results for version- vs. project-level prediction [14].

6.2. What Went Wrong

The results of this analysis, in Figure 4, show that we further increased the false alarms by selecting the

Table 5. Results of our local defect prediction model with cost-benefit analysis.

			pd	pf	bal	# Modules	# Required Modules	# Inspected Modules	Verification Effort(%)
2.32	**Trcll3**	**Model I**	0.50	0.49	0.50	249	137	132	4
		Model II	**0.50**	**0.19**	**0.62**		137	53	**61**
	Trcll2	**Model I**	1.00	0.31	0.76	245	221	88	60
		Model II	**1.00**	**0.18**	**0.86**		221	49	**78**
	Trcll1	**Model I**	0.80	0.75	0.45	388	326	291	11
		Model II	**0.80**	**0.62**	**0.54**		326	238	**27**
2.33	**Trcll2**	**Model I**	1.00	0.22	0.84	247	247	72	71
		Model II	**1.00**	**0.15**	**0.89**		247	43	**83**
	Trcll1	**Model I**	1.00	0.69	0.42	389	233	226	3
		Model II	**1.00**	**0.61**	**0.53**		233	173	**26**
	Trcll4	**Model I**	1.00	0.08	0.95	27	27	8	70
		Model II	**1.00**	**0.00**	**1.00**		27	2	**93**
2.34	**Trcll2**	**Model I**	0.75	0.75	0.42	389	292	293	0
		Model II	**0.75**	**0.53**	**0.55**		292	208	**29**
	Trcll5	**Model I**	0.70	0.41	0.65	493	355	219	38
		Model II	**0.70**	**0.10**	**0.75**		355	69	**81**

training data from previous versions of various projects inside the software system, compared to the results, when training data is selected from the previous versions of the specific project only.

6.3. How the Team Changed To Repair This

In order to decrease the false alarm rates, we wanted to increase the information content of the inputs of our model by focusing on a specific project and build training set only from the previous version of this project.

In Figure 4, sample of two versions and two projects are selected to compare version-level vs. project-level prediction. Third column (the first experiment) presents the prediction performance when we choose our training set from all projects of the previous version to predict defective modules of projects, Trcll1 and Trcll2. The last column (the second experiment), on the other hand, shows the performance of our predictor when we use only the previous version of Trcll1 or Trcll2 to predict defective modules of the selected project in the current version. From the results, we have concluded that project-level defect prediction is better although we still have high false alarm rates.

7. What Happened at the End

We have successfully built our defect predictor for Trcll using local data and presented our results to the project team. The results of the project showed that the Trcll's business goal of decreasing testing effort without compromising the level of product quality can be achieved with intelligent oracles. We have used several methods to calibrate the model for Trcll in order to get the best prediction performance for them. We have seen that file-level call graph based ranking (CGBR) method did not work due to their transition to Service Oriented Architecture (SOA). SOA did not allow us to capture caller-callee relations through simple file interactions. Moreover, we have currently used static code attributes from Java files to build our model. However, there are many PL/ SQL scripts that contain very critical information on the interactions between application and data layers. Thus, a simple call-graph based ranking in file-level could not capture the overall picture and hence fail to increase the information content in our study.

As the final step, we have discussed on the reasons of high false alarms in monthly meetings with Trcll project team and found that we need to clean files that are not changed since January 2008 from the version history. For this, we built a simple assumption on defect-proneness of a module: *It is highly probable that a module is defect free if it has not been changed since January 2008*. Then, we have added a flag to each file of the projects that indicates whether the file is actively changed or passive since January. Our model controls each of its predictions by looking at the history flag of these files. If the model predicts a file as defective, although it has not been used since January, then it is re-classified as defect-free. Sample results for this experiment can be seen in Table 5. Model I represents

our predictor without using version history, whereas Model II represents a combination of our predictor and the version history to predict defective modules of Trcll data. We can clearly observe that using version history improves the predictions significantly in terms of pf rates. Our model succeeded in decreasing false alarms, on the average from 46% to 29% using version history. The change in pf rates vary in terms of projects in the range of {7%, 31%} due to discontinuities in the changed projects throughout version history.

Additionally, we managed to have stable high pd rates, on the average 84%, while reducing pf rates successfully. Besides, we have spent less effort to detect 84% of these defective modules. Our results show that the improvement in testing effort is around 50% when we used Model II instead of I. This indicates that we inspected fewer LOC, only 30% instead of 61%, and still found the best prediction performance on the projects. Therefore, the cost-benefit analysis shows that using a defect prediction model enables developers allocate their limited amount of time and effort to only defect-prone parts of a system. Managers can also see the practical implications of such decision making tools which reduces testing effort and cost.

8. Lessons Learned

During this project we had many challenges to overcome, and we constantly re-defined our processes, and planned for new sets of actions. In this section, we would like to discuss how we feel about this study: what can be used as best practices, and what needs to be avoided next time. We hope that this study and our self evaluation would shed some light for other researchers and practitioners.

8.1. Best Practices

Managerial Support: From the beginning till the end of this work, we had full support of senior management, and mid level management. They were available and ready to help whenever we needed them. We believe that without such a support a project like this would not have been concluded successfully.

Project planning and monitoring: One of the critical success factors was that we had a detailed project plan and we rigorously followed and monitored the plan. This enabled us to identify problems early on and to take necessary precautions on time. Although we had many challenges we were able to finish the project on time achieving and extending its intended goals. These meetings also brought up new and creative research ideas. As a research team we mapped the project plan and its deliverables to the deliverables of two masters and one PhD thesis. Our research deliverables were to publish papers as well as to finish thesis works of three Softlab students. We have successfully achieved these research goals at the end.

Multiplier Effect: One of the benefits of doing a research in a live laboratory environment like Trcll is that researchers can work on-site, access massive amounts of data, conduct many experiments, and produce a lot of results. The benefit of this amateur attitude to a commercial company is that they can get five times more output than originally planned. It is definitely a win-win situation. Although we had started a measurement and defect prediction problem focusing only the testing stage, we have extended the project to be able touch whole software development life cycle: 1) *the design phase* by using dependencies between modules of the software system, 2) *coding phase* by adding static code measurement, raw code analysis and rule-based model, 3) *coding phase* by employing a sample of test-driven development, 4) *testing phase* by building a defect predictor to decrease testing efforts, and finally 5*) the maintenance phase* by examining the code complexity measures to evaluate which modules need to be re-factored in the next release.

Existence of Well Defined Project Life Cycles and Roles/ Responsibilities: The development lifecycle in Trcll has been arranged such that all stages, i.e. requirements, design, coding, testing and maintenance are separately assigned to different groups in the team. Therefore, segregation of duties has been successfully operated in the company. We have benefited from this organizational structure while we were working on this project. It was easy to contact test team to take defect data, as well as the development team to take measurements from the source code.

8.2. Things to Avoid Next Time

Lack of Tool Support: Automated tool support for measurement and analysis is fundamental for these kinds of projects. In this project we have developed metrics extraction tool to collect code metrics easily, however, we were unable to match defects with corresponding files. Therefore it took too much time to be able to construct local defect prediction model. Therefore, our next plan would definitely be initiating an automated bug tracing/ matching mechanism with Trcll. We now highly recommend that before a similar project starts an automated tool support for bug collection and matching is employed.

Lack of documentation and architectural complexity: Large and complex systems have distinguishing characteristics. Therefore, proper documentation is paramount to understand the complexities especially when critical milestones are defined at every stage of such projects. This has caused us to face with many challenges as we moved along. We had to change our plans several times.

9. Conclusion

AI has been tackling the problem of decision making under uncertainty for some time. This is an everyday business problem that managers in various industries have to deal with. In this project and its related research we have built an oracle for software development managers to help them decide "how much testing is enough?" and "when to stop testing and go live?".

The research we undertook during the course of this project has been at the intersection of AI and Software Engineering. We had the opportunity to use some of the most interesting computational techniques to solve some of the most important and rewarding questions in software development practice. Our research was an empirical study where we collected data, designed experiments, presented and evaluated the results of these experiments. Contrary to classical machine learning applications we focused on better understanding the data at hand. The project in Trcll provided the "live laboratory" environment that was necessary to achieve this.

We have seen that implementing AI in real life is very difficult, but it is possible. As always both sides (academia and practice) need "passion" for success. Our empirical results showed that a metrics program can be built in less than a year time: as few as 100 data points are good enough to train the model [3]. In the meantime the company can use cross company data to predict defects by using filtering techniques (NN and CGBR). In the case of Trcll in cross company experiments we showed that the estimated testing effort is only 3% to catch around at least 70% of the defects. But we could not measure the classification accuracy in cross-company analysis. Finally, once a local repository is built and version history information is used the testing effort decreased up to 50% in certain projects, from 60% to 31%.

As a future direction we are extending Prest to extract metrics from PL/ SQL. We would also like to develop an automated tool to match defects seamlessly to the coders. If such a tool can be developed we would be able to match at a lower granularity level (i.e. function level instead of file level) so that we can have more accurate prediction results.

8. Acknowledgment

This research is supported in part by Turkcell A.S.

10. References

[1] T. Menzies, J. Greenwald and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors", *IEEE Transactions on Software Engineering,* January 2007, Vol. 33, No. 1, pp. 2-13.

[2] T. J. Ostrand, E. J. Weyuker and R. M. Bell, "Predicting the Location and Number of Faults in Large Software Systems", *IEEE Transactions on Software Engineering,* April 2005, Vol. 31, No. 4, pp. 340-355.

[3] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic and Y. Jiang, "Implications of Ceiling Effects in Defect Predictors" in *Proceedings of PROMISE 2008 Workshop (ICSE) 2008.*

[4] M.H. Halstead, *Elements of Software Science,* 1977, Elsevier, New York.

[5] T. McCabe "A Complexity Measure", *IEEE Transactions on Software Engineering,* 1976, Vol.2, No.4, pp. 308-320.

[6] T. Menzies, B. Turhan, A. Bener, and J. Distefano, "Cross- vs. Within-Company Defect Prediction", *Technical Report,* WVU, USA, 2008.

[7] G. Boetticher, T. Menzies, T. and Ostrand, PROMISE Repository of empirical software engineering data http://promisedata.org/ repository, West Virginia University, Department of Computer Science, 2007.

[8] B. Turhan, A. Bener, and T. Menzies, "Nearest Neighbor Sampling for Cross Company Defect Predictors (Abstract Only)", in *Proceedings of the 1st International Workshop on Defects in Large Software Systems (DEFECTS'08 Workshop in ISSTA'08)*, pp. 26, 2008.

[9] Prest, *Dept. of Computer Engineering, Bogazici University*, Available from: http://svn.cmpe.boun.edu.tr:8080/svn/softlab/prest/trunk/Executable, 2008.

[10] G. Kocak, B. Turhan and A.Bener, "Predicting Defects in a Large Telecommunication System", in Proceedings of the *3rd International Conference on Software and Data Technologies (ICSOFT)*, 2008, Portugal, pp.284-288.

[11] G. Kocak, "Software Defect Prediction Using Call Graph Based Ranking (CGBR) Framework", MS Thesis, Boğaziçi University, 2008, Turkey.

[12] NASA WVU IV & V Facility, Metrics Data Program, http://mdp.ivv.nasa.gov, 2004.

[13] D. Heeger, “Signal Detection Theory,” available at http://white.stanford.edu//~heeger/sdt/sdt.html, 1998.

[14] A.Tosun, B. Turhan and A. Bener, "Direct and Indirect Effects of Software Defect Predictors on Development Lifecycle: An Industrial Case Study", to appear in Proceedings *of the 19th Interntational Symposium on Software Reliability Engineering (ISSRE'08 Industry Track)*, 2008.

[15] A. Tosun, B. Turhan and A.Bener, "Ensemble of Software Defect Predictors: A Case Study", to appear in
.

Proceedings of the *2nd International Symposium on Empirical Software Engineering and Measurement (ESEM'08 Short Paper),* 2008.

[16] B. Turhan and A.Bener, "Weighted Static Code Attributes for Software Defect Prediction", in Proceedings of the *20th International Conference on Software Engineering and Knowledge Engineering (SEKE'08)*, pp.143-148, 2008.

[17] A.G. Koru and H.Liu, “Building effective defect prediction models in practice”, *IEEE Software*, pp.23-29, 2007.

[18] E.Arisholm and C.L. Briand, “Predicting faults prone components in a Java legacy system”, in Proceedings of the *ISESE’06*,1-22, 2006.