2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)

Cost-effective Build Outcome Prediction Using Cascaded Classifiers*

Ansong Ni, Ming Li
National Key Laboratory for Novel Software Technology
Nanjing University
niansong1996@gmail.com, lim@lamda.nju.edu.cn

***Abstract*—Software developers use continuous integration to find defects in the early stage and reduce risk. But this process can be resource and time consuming, which decreases the efficiency of development. In this work, we adopt cascaded classifiers to predict the build outcome and study what kinds of attributes are potentially useful for this process. We emphasize on the "failed" instances which bring more cost. Our experiments reveal that our approach outperforms other commonly used classifiers. It reduces 51.7% of the waiting time and server workload while identifying 85.2% of the defective builds.**

***Keywords*-continuous integration; software mining; ensemble learning; cost-sensitive learning; software engineering**

I. Introduction

Continuous integration (CI) is a widely used practice in modern software engineering to reduce risks by finding defects in the early stage [1]. However, such a process can be time consuming and requires huge computing resources. For large projects, the build certification may take hours or even days to complete [2][3]. This is detrimental for team collaboration because other developers can not decide whether to pull the code before the outcome of the build which severely reduces the efficiency.

Suppose we can correctly predict the success build, we can save the time and resources by just label it as "success" and if our prediction implies that it may be failed, we shall actually run the build to see where the defect lies. So researchers have tried to use machine learning methods to learn from building history and use the commit information as features to predict the outcome of the build. Hassan *et al.* used different groups of attributes of commits to predict the build certification with a high accuracy [3] by decision trees, and Finlay *et al.* also yielded high accuracy (75%) using code metrics of a certain build as features [5].

Though previous works have succeed in predicting the build outcome with high overall accuracy, but the performance on the "failed" builds is actually very disappointing. Some researchers also noticed that it is much harder to predict failure than success [5]. Since commmon projects usually have more succeeded builds than failed ones, failure is a minor class in the whole data set, which explains why classifiers do horribly on failed class can still yield a good overall performance. Unfortunately less quantity does not necessarily mean less importance, it is actually the opposite in this particular case. Finding broken builds is exactly the meaning of CI practice, one hidden defect demands much more time to locate over time which will cost the developers dearly. In this way, it is clearly a cost-sensitive problem which requires high recall on the failed class and reduce the waiting time and server workload at the same time.

Our contributions are in two folds. 1) We tackle this problem with the cascaded classifiers [6] and greatly reduce the waiting time and workload of the CI server while capturing most of the failed builds (FB). The cascaded classifiers consist of multiple layers of base learners. It diminish the chances of making mistakes at each level to refine the final result and emphasize on the minor class at the same time, so it is a ideal method in our problem settings. Moreover, users can customize the performance tradeoff with their own demand, lower recall on the failure leads to less workload on the server and saves more time. 2) We designed lots of practical features which are easily accessible and computable and perform a study to evaluate groups of features with potential effectiveness, which can be referred for future researches on this topic.

II. Methodology

Our proposed approach consists of 3 phases: 1) Pull the commits' information from VCS (Version Control Systems) such as Git or SVN; 2) Extract features from the commits with respect to multiple perspectives; 3) Train the cascaded classifiers with extracted features and a manually set recall on the failed build (FB) to predict future cases.

A. Data Collection

Travis-ci (https://travis-ci.org) provides free CI services for open source projects on *Github* and over 300,000 projects are running CI on travis. Thanks to *TravisTorrent* [7], we are able to synthesis the information from *Travis-ci* and *Github* to do a correlated analysis. We download the *travistorrent_6_12_2016* dataset and label a build as "success" if all its jobs succeeded or "failed" otherwise. Then we discard the projects with $LOC < 1000$ or the projects with a shorter history of than 200 builds to guarantee proper scale of the chosen datasets. We also leave alone the projects who have no test code or never perform tests to make sure that the CI practice is properly used. For simplification, we only analyze the master branch of the projects and ignore other branches. Though the data from TravisTorrent is quite rich, we query the *Github API (https://developer.github.com/v3)* by the SHA1 of a commit

*Research supported by NSFC (61422304, 61272217) and NCET-13-0275.

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.26

to get more information about the committers and committed files. By the end, we have 365,565 builds from 532 projects using Java or Ruby, with full commit and build information as our dataset for the next step.

B. Feature Extraction

Table I shows the list of different groups of features we use to represent a push which consists of several commits and triggers the build. For simplification, we do not distinguish active push and pull requests.

TABLE I: Groups of Features We Studied

(a) Connection to Last Push

Feature Name	Explanation and Rationale
last_build_result	The result of last build("passed", "failed" or "errored")
no_src_edited	The Cartesian product of last_build_result and a boolean to indicate if the there are source code files (*.java or *.rb) edited (added, modified or deleted) in this commit
no_config_edited	Similar to no_src_edited with a different boolean to indicate if the there are configuration files (*.xml, *.yml, etc) edited in this commit
no_src_config_edited	Take "AND" operation on the two boolean variables mentioned above
same_committer	The Cartesian product of last_build_result and a boolean to indicate whether the committer is the same as last push sequentially
time_elapse	take a $log()$ on the days since last push sequentially

(b) Historical Statistics

Feature Name	Explanation and Rationale
committer_history	The fail rate of the pushes by the current committer of the history before this build on this project
committer_recent	Similar to committer_history, but measuring only his last five pushes
project_history	The fail rate of the all the project's previous build
project_recent	Similar to project_history but using only last five
gaussian_threat	Model the build failure of the project as gaussian distribution to measure the threat to current build
committer_exp	The logarithmic of the number of pushes the committer made in the project before this build

(c) Facts on the Current Push

Feature Name	Explanation and Rationale
commit_num	# of commits contained in this single push
total_files_pushed	# of files edited in this push and take a $log()$
avg_file_committed	Avg. # of files edited in the commits being pushed and take a $log()$ on it
file_added	# of files created in this push and take a $log()$
file_modified	# of files modified in this push and take a $log()$
commit_msg_length	Avg. length of the commit messages of this push and take a $log()$ on it

Detailed Explanation on features are as follows. The magnitudes of the numerical features are fairly large, for instance most pushes contain several files but there are cases when thousands of them are included. So we normalize the numerical data by applying logarithmic functions to restrict the values to a relatively small interval. For feature *gaussian_threat*, we try to model the effect of certain failed build (FB) as a Gaussian distribution with the intuition that a build closer to a FB or more FBs has greater chance of failing [8]. So we use equation (1), where $\mathbf{F}$ is the sets of failed builds in the history and f_t refers to number of builds happened after that failure f to measure the "distance" from now. Parameter σ_f is used to indicate the range of builds the failure f can have affect on and here we equally set all σ_f.

$$T = \sum_{f \in \mathbf{F}} \frac{1}{\sqrt{2\pi}\sigma_f} \exp\left(-\frac{f_t^2}{2\sigma_f^2}\right) \tag{1}$$

By this equation, we can measure the threat for the current build from all broken builds of the project's history.

C. Prediction Model

1) Cascaded Classifiers: Cascaded classifiers is brought up by P. Viola and M. Jones [6] to build a face detector. As figure 1 shown, each layer of cascade is a base learner with its own threshold to classify the instances that passed along from the previous one. The basic principle of cascaded classifiers is dropping the high-confidence negative instances and keep suspicious ones for the next layer of cascade to determine. Algorithm (1) describes the training algorithm we derive from the original work [6] and adapte for the build outcome prediction problem. Note that in the following descriptions, we will refer the *failed* class (FB) as *positive* and *passed* as *negative* instances.

Algorithm 1: Training the Cascaded Classifiers

input : $\mathcal{P}$ = A set of positive(failed) instances;
$\mathcal{N}$ = A set of negative(passed) instances;
R_{target} = Target overall positive recall;
K = Number of the layers of the cascade;
f = maximum false negative rate per layer
output: Trained Cascaded Classifiers

Initialize all K base learners;
$R_{layer} \leftarrow \sqrt[K]{R_{target}}$;
for $i \leftarrow 1$ **to** K **do**
  Train the base learner of the i-th layer L_i with $\mathcal{P}, \mathcal{N}$;
  Adjust the threshold t_i so that the positive recall of $L_i = R_{layer}$;
  $\mathcal{P}^* \leftarrow$ False Negative ;
  $\mathcal{N}^* \leftarrow$ True Negative ;
  $f_{layer} \leftarrow \mathcal{P}^*/(\mathcal{P}^* + \mathcal{N}^*))$;
  if $f_{layer} > f$ **then**
    for $j \leftarrow i$ **to** K **do**
      $t_i \leftarrow 1$;
    end
    break
  end
  $\mathcal{P} \leftarrow \mathcal{P}/\mathcal{P}^*$;
  $\mathcal{N} \leftarrow \mathcal{N}/\mathcal{N}^*$;
end

By using cascaded classifiers, we carefully control the possibility of being wrongly classified as passed which bring us much cost by setting a high confidence threshold, meaning only if the classifier has full confident that it is a success build will the classifier label it as "passed", or it will send it to the next layer for more refined analysis. In another word, given a trained cascade of classifiers, the overall positive recall R-FB is determined by equation (2) where r_i refers to the positive recall of the i-th classifier.

$$\text{R-FB} = \prod_{i=1}^{K} r_i \tag{2}$$

After each layer, the training set is more balanced because more negative instances were removed. This utilized base learner's learning process and emphasizes more on the positive class to perform cost-sensitive learning imperceptibly.

We can also observe the fact that after each layer, the number of training instances is reduced. It enables the next

layer to do relatively faster training. Similar for predicting algorithm, if an instance is labelled as "passed" which is the major case, it is very likely that it does not need to go through all the K layers and the classification process will be completed when it is above the threshold of some layer in the early stage. In this way, the cascading schema is much faster than other ensemble methods which relies on all training instances and all base learners to perform prediction.

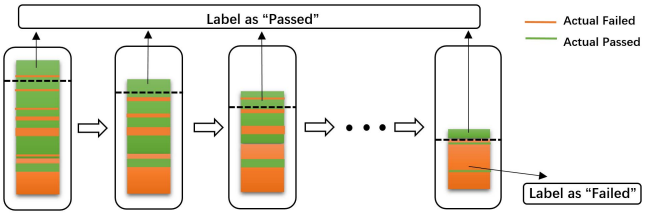


Fig. 1: Structure of Cascade

2) Adaboost: Adaboost is the most famous boosting algorithm brought up by Freund and Schapire [9]. It is an ensemble learning method that greatly improves the performance from it is base learners (usually weak learners) by re-weighting the training set and putting more weight on the wrongly classified instances for the next base learner. In our work, we use DecisionStump [10] as our base learner, which is a single-node decision tree using only one feature. For each base learner we compute the loss by equation (3) where w_t is the weight vector for the t-th layer.

$$\epsilon_t = \sum_{i=1}^{N} p_i^t |h_t(i) - c(i)|, \quad p^t = \frac{w^t}{\sum_{i=1}^{N} w_i^t} \tag{3}$$

Finally, as shown in equation (4), the prediction is made by the weighted-voting of all its base learners, given weight $\log \frac{1}{\beta_t}$ on base learner $h_t(\cdot)$ with error ϵ_t, where $\beta_t = \epsilon_t/(1-\epsilon_t)$.

$$h(i) = \begin{cases} 1, \sum_{t=1}^{T} (\log \frac{1}{\beta_t}) h_t(i) \geq \frac{1}{2} \sum_{t=1}^{T} \log \frac{1}{\beta_t} \\ 0, \text{otherwise} \end{cases} \tag{4}$$

The intuition of the weighed-voting is rather simple, base learner with less error gets more weight in classifying future instances. Since we use DecisionStump which is feeded by only one feature as the base learner, the weights also give us a quantitative analysis on effectiveness of features.

III. Experiment and Result Analysis

In the experiment, we want to answer the following research questions:

RQ1. What kind of features are beneficial for predicting build outcome?
RQ2. Can cascaded classifiers have high *Recall-FB*, high *Accuracy*, high *Cut Ratio* and high *Gain* simultaneously?
RQ3. Does cascaded classifiers outperform other frequently used classifiers in build outcome prediction?

A. Evaluation Metrics

We use accuracy, recall and AUC as the evaluation measures. We also define two specific measures for the build outcome prediction problem.

1) Cut Ratio: If a build is predicted as "passed", we should just label it as "passed" and do not actually build it, predicting more "passed" gives us more cut in actually build quantities which reduces the time and resource consumption. So we use *Cut Ratio* to denote **the proportion of instances predicted as "passed"**.

2) Gain: We also need to access the negative effect brought by false negative instances which are those "hidden defects". So we use a metric to balance between the time and resources we saved and give a penalty on those wrongly classified as "passed" by formula (5).

$$Gain = CutRatio - \lambda \cdot \frac{\#\ of\ False\ Negative}{\#\ of\ Test\ Set} \tag{5}$$

B. Experiments and Results

In the experiment, we used first 50% of the dataset as training set and last 50% instances as test set. We used the AdaBoostM1 from *WEKA* [10] as our implementation for Adaboost and we implemented the cascade framework in Java.

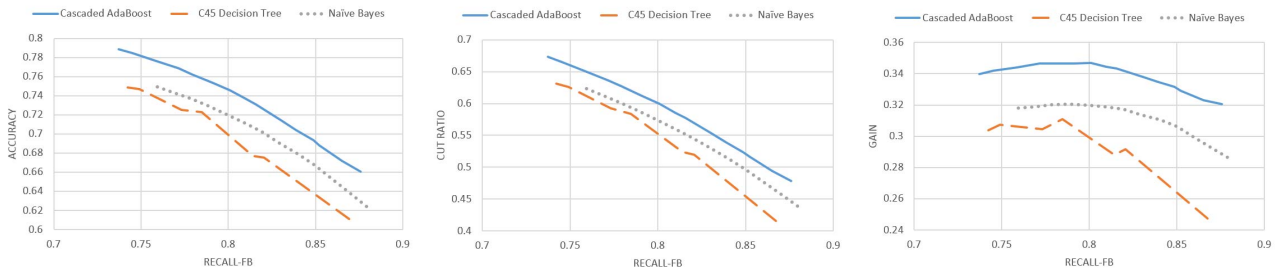
1) Finding Best Features: Here we compare the performance of cascaded classifiers on different sets of features: *Connection to Last Push*(I), *Historical Statistics*(II), *Facts on the Current Push*(III) and their combinations. We set the target positive recall at 80%, the iterations of Adaboost to be 30 and $\sigma_f = 3$ for *gaussian_threat* and evaluates accuracy, AUC and Cut Ratio. Table II shows the result of the experiment, "FB" denotes the class of "Failed Build".

TABLE II: Effectiveness of Combinations of Feature Groups

	I	II	III	I+II	I+III	II+III	I+II+III
Recall-FB	0.689	0.737	0.691	**0.742**	0.727	0.737	0.737
Cut Ratio	0.595	**0.641**	0.337	0.636	0.547	0.640	**0.641**
Accuracy	0.684	**0.753**	0.427	0.750	0.654	0.752	**0.753**
AUC	0.686	0.747	0.518	**0.748**	0.680	0.747	0.747

This implies that *Historical Statistics* are the most useful features in predicting the build outcome. Especially, *gaussian_threat* is the feature with the greatest power after further investigation. *Connection to Last Push* is also quite beneficial while *Facts on the Current Push* is of little use. We also try some other features not included in this experiment such as code chunk metrics (LOC, added_lines, etc), social factors [3] (time, day, week of the push, etc) and commit messages [4]. However, these features are even worse than the *Facts on the Current Push*.

2) Comparison Between Classifiers: As the result of the previous experiment, we use *Connection to Last Push* and *Historical Statistics* (I+II) to compare the performance of different classifiers. Here we use C4.5 decision tree [12], Naive Bayes [11] as contrast methods, with implementation in WEKA [10] as *J48* and *NaiveBayes*, respectively. We set different target Recall-FB to reach different Recall-FB and used CostSensitiveClassifier+J48/NaiveBayes in WEKA to do the same. We evaluate different classifiers by accuracy, CutRatio and Gain against different Recall-FB values. We set $\lambda = 5$ for *Gain* as suggested in [5]. Comparison between the performance of the classifiers are shown in Figure (2a), (2b),



(a) Accuracy vs. Recall-FB (b) CutRatio vs. Recall-FB (c) Gain vs. Recall-FB

Fig. 2: Comparison Between Classifiers

(2c). From those results, we can observe that in high Recall-FB domains, Cascaded Adaboost dominates other contrasting methods. Take positive recall = 80% as an example, the accuracy of Cascaded Adaboost can still reach 74.6% while C4.5 only has 66.7% and NaiveBayes with 72.0%. The performance gap is larger when we evaluate by the *gain*. When C4.5 and NaiveBayes give 27.8% and 32.0%, Cascaded Adaboost can reach 34.7%. This suggests that Cascaded Adaboost has better performance when we need high Recall-FB, which is essential in build outcome prediction.

C. Discussion

By comparison, it can be observed that *Historical Statistics* and *Connection to Last Push* are two sets of helpful features while others are not. This shows that history can convey more information so that history for the committers or projects can better indicate the quality of the current push. Code metrics may not be helpful because commits are dynamic, incremental changes rather than the stable, complete code base, so it is hard to have meaningful code metrics on them. Also, open source projects' contributors may have very flexible schedule with global cooperation, which makes social factors useless in accessing the quality of the commit. That is a possible explanation of our findings being different from the work of Hassan *et al.* [3] on this issue. Studying performance of the classifiers on different programming languages (Java, Ruby) will be an interesting future work for us.

For different classifiers, the results suggest that the our approach outperforms other commonly used classifiers in build outcome prediction such as C4.5 and NaiveBayes. The cascaded structure controls the false negative rate on each level and passes along a more balanced training set, it decreases the chances of paying great loss layer by layer, making it more suitable for cost-sensitive learning. And we can quantitatively analysis the effect of the features by looking into the Adaboost and check the weights of the DecisionStumps, which provides us with good interpretation on the learning process and benefits future research.

IV. Conclusion

In this section, we conclude our work by answering the research questions brought up in the experiment section.

RQ1. What kind of features are beneficial for predicting build outcome?

Answer1. Historical Statistics on the committer and project are the best indicators for build outcome prediction while seeking for connections to last push can also lead to good attributes. Simple statistics about current push such as code, files and commit messages are of little use in this task.

RQ2. Can cascaded classifiers have high *Recall-FB*, high *Accuracy*, high *Cut Ratio* and high *Gain* simultaneously?

Answer2. Yes. Cascaded classifiers can have *Positive Recall* of 80.1%, *Accuracy* of 73.7%, *Cut Ratio* of 60.0% and *Gain* of 34.7% simultaneously.

RQ3. Does cascaded classifiers outperform other frequently used classifiers in build outcome prediction?

Answer3. Yes. When we are targeting a recall for build failure, the cascaded classifiers significantly outperforms other commonly adopted classifiers such as C4.5 decision tree and NaiveBayes.

References

[1] P. M. Duvall, *Continuous integration*. Pearson Education India, 2007.

[2] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An analysis of travis ci builds with github. In *No. e1984v1. PeerJ Preprints*, 2016.

[3] A. E. Hassan, and K. Zhang. Using decision trees to predict the certification result of a build. In *21st IEEE/ACM International Conference on Automated Software Engineering*, 2006.

[4] E. A. Santos, and A. Hindle. Judging a commit by its cover: correlating commit message entropy with build status on travis-CI. In *Proceedings of the 13th working conference on mining software repositories*, 2016.

[5] J. Finlay, R. Pears, and A. M. Connor. Data stream mining for predicting software build outcomes using source code metrics. In *Information and Software Technology* 56.2 (2014): 183-198.

[6] P. A. Viola, and M. J. Jones. Robust Real-Time Face Detection. In *International Journal of Computer Vision* 57.2 (2004): 137-154

[7] M. Beller, G. Gousios, and A. Zaidman. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th working conference on mining software repositories*, 2017.

[8] S. Kim, T. Zimmermann, and E. J. Whitehead Jr.. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, 2007.

[9] Y. Freund, and R. E. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, 1995.

[10] I. H. Witten, and E. Frank. *Data Mining: Practical machine learning tools and techniques with Java implementations*. Morgan Kaufmann, 1999.

[11] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. In *Machine learning* 29.2-3 (1997): 131-163.

[12] J. R. Quinlan. *C4. 5: programs for machine learning*. Morgan Kaufmann, 1993.