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

Built to Last or Built Too Fast? Evaluating Prediction Models for Build Times

Ekaba Bisong, Eric Tran, Olga Baysal
Department of Computer Science
Carleton University, Ottawa, Canada
ekaba.bisong@carleton.ca, eric.tran@carleton.ca, olga.baysal@carleton.ca

***Abstract*—Automated builds are integral to the Continuous Integration (CI) software development practice. In CI, developers are encouraged to integrate early and often. However, long build times can be an issue when integrations are frequent. This research focuses on finding a balance between integrating often and keeping developers productive. We propose and analyze models that can predict the build time of a job. Such models can help developers to better manage their time and tasks. Also, project managers can explore different factors to determine the best setup for a build job that will keep the build wait time to an acceptable level. Software organizations transitioning to CI practices can use the predictive models to anticipate build times before CI is implemented. The research community can modify our predictive models to further understand the factors and relationships affecting build times.**

***Keywords*-Machine learning; continuous integration; builds; build time**

I. Introduction

Automated builds are integral to the Continuous Integration (CI) software development practice. As developers check in code to the shared repository, an automated system picks up the changes and triggers a build. The automated build compiles the code, and ideally, runs a test suite. Build results notify developers about integration problems such as compilation errors or missing dependencies. When combined with unit tests, build results can reveal broken or changed functionality in a software project.

In CI, developers are encouraged to commit their changes early and often. Changes that are smaller and more regularly integrated are easier to debug when something breaks [8]. Thus, build times are very important when integrations are frequent. Long build times can become a bottleneck to the CI process.

Long build times are problematic for developers. Developers can lose focus and become less productive while waiting for a build to finish. For example, developers may work on specific tasks in separate branches using a version control system. If a build fails, it would be cumbersome to switch back to the original branch due to several reasons such as caching, configuration files, and so on. Additionally, developers may experience context switching when changing between different tasks. The cost of context switching can be low when the complexity of tasks is low. Contrarily, when the complexity of tasks is high, context switching can be a costly expenditure of the developer's mental energy [6]. Therefore, it may be easier to stay in the current branch and wait for the build results to finish before moving on to the next task or continuing with the current one.

We are interested in finding a balance between integrating often and keeping developers productive. Our research goal is to build a predictive model that can estimate build time of a job. In our work, we use TravisTorrent [2], a freely available dataset combining features from GitHub and Travis CI for builds of more than 1,000 projects.

II. Related Work

Mokhov et al. [9] reported that build systems face issues when projects have a large number of components, multiple languages, and complex interdependencies. For example, build systems may have executables that depend on libraries or running tools that are generated by the same build system. Also, build systems can be hard to maintain when build environments are not managed. Employing tools such as the Nix package manager can help describe package build actions and their dependencies, allowing the build environments to be produced automatically [4].

The effects of long build times can be negative. Brooks [3] found that long build times can affect the following variables: commit size, commit frequency, build down time, development flow, and developer satisfaction. Negative perceptions about waiting times can be lowered by providing feedback, controlling perceived waiting time, and having different waiting times for different tasks [7].

As a solution to slow builds, Ammons [1] suggested breaking large, all-or-nothing builds into many smaller builds. To demonstrate the technique, Ammons implemented a tool suite called Grexmk. Grexmk contains tools for dividing large builds into mini-builds and tools for executing mini-builds in parallel and incrementally. Moreover, a mini-build is an all-or-nothing build that explicitly lists its output files, source files, dependences on other mini-builds, and build script. Overall, incremental builds were sped up by a factor of 1.2 when using Grexmk. Brooks [3] suggested that a build time of 2 minutes was optimal; build times under 10 minutes were considered acceptable. However, the suggested build times were based on experience reports from the same company. In summary, there is a lack of empirical quantitative research to address optimal build times in a CI environment [7].

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.36

III. Methodology

We now briefly describe our methodology including the dataset we used and the process we followed.

A. Dataset

We selected TravisTorrent [2] as the dataset for our prediction task. TravisTorrent is a synthesis of pull-request commits from software projects hosted on GitHub with TravisCI integrated as a mechanism for continuous integration. The dataset draws features from GitHub for a particular build job in a project, as well as corresponding build features from TravisCI. The data contains over 2 million records spanning 1,000 projects. In this paper, we mine the information from this dataset to investigate the factors that affect the build time of a build job in a pull-request CI development ecosystem. We build a predictive model to estimate the build time of a particular build job given that a specified set of build job features is known.

B. Feature Selection

We used the 30/9/2016 Travis dataset which contained 56 features, of which 34 are numeric, 16 are strings, 4 are booleans, and 2 are in the ISO date format. We selected all but 3 of the integer/numeric features. The excluded features were unique records which indicated the unique identifier for each build job, the pull request number in GitHub, and the build number in Travis. We did not find any of the string and date variables to be useful enough to be included as features in our predictive model. Finally, we considered all the boolean variables (which are coded as factors) in this initial feature selection phase. In total, 35 features were selected for building the predictive model from the original 56.

C. Evaluation Metrics

To evaluate how well our model is performing in predicting build times we used Root Mean Square Error (RMSE) and R^2 (R-Squared). RMSE reports the mean deviation of the predicted value from the original value. The lower the RMSE, the better the prediction accuracy. R^2 gives us a measure of how much of the variation in predicted values is explained by the model. The values of R^2 ranges between 0 and 1. Values close to zero signify that a large proportion of variability in the result is unexplained by the model, while values close to one indicate that most of the variance in the result is accounted for by the model.

D. Rationale for Algorithm Selection

We sample a set of linear and non-linear algorithms that work on regression problems to get a baseline performance on model accuracy. In doing this, we use 10-fold cross validation (CV) with 3 repeats. This CV procedure splits the training set into 10-folds with each selected algorithm running 10 times on 90% of the data, using the remaining 10% to assess model performance. This process is repeated 3 times to produce an unbiased estimate of the algorithm performance. This also prevents over fitting the model by capturing noise from the data, which inherently leads to poor generalizability in predicting an out-of-sample build job time.

TABLE I: Benchmark results.

Algorithm	**RMSE**	R^2
LM	5,839	0.5421
PLS	5,769	0.5519
GLMNET	5,755	0.5539
LARS	5,825	0.5441
CART	4,723	0.7127
SVM	5,281	0.6317
KNN	5,451	0.6066
NNet	10,180	NaN
BCART	5,516	0.5942
RF	4,145	0.7742
SGB	4,374	0.7489
CB	4,052	0.7808
XGBOOST	4,693	0.7134

For linear models, we sample the following algorithms, Linear Regression (LR), Partial Least Squares (PLS), Penalized Linear Regression (GLMNET), and Least Angle Regression (LARS). For non-linear models, we sampled Classification and Regression Trees (CART), Support Vector Machines (SVM) with a radial basis function, k-Nearest Neighbors (KNN), and Neural Network (Nnet). We then spot-check a set of Ensemble methods such as Bagged Classification and Regression Trees (BCART), Random Forest (RF), Stochastic Gradient Boosting (SGB), and Cubist (CB) models.

The linear models were selected due to their tendency to provide good prediction results even when the inherent structural form of the data is non-linear. With enough data entries, linear models perform well, sometimes out-performing or even equalling the performance of their non-linear counterparts on non-linear datasets. The non-linear models were selected because we understand the structure of the underlying relationships between features in the data to be non-linear. With a non-linear structure, a non-linear algorithm will be favoured to perform better in prediction accuracy assessments. Finally, ensemble methods combine the outputs of various algorithms to get a better prediction score on unseen data. We sample a few of them here because ensemble methods are known to give good accuracy measures in various prediction tasks.

IV. Results

In this section, we report the results of our study.

A. Benchmark

To speed up the data processing time, we ran our algorithms on a subset of 10,000 records from the training set to get our initial baseline results on how the learning algorithms are performing. A seed was set to ensure consistency and reproducibility. The results are shown in Table I.

The results demonstrate that Cubist (CB) model has the lowest RMSE of 4,052 seconds, followed by Random Forest (RF) with 4,145 seconds. While Cubist (CB) model and Stochastic Gradient Boosting (SGB) also have the highest and second-highest R^2 value of 0.7808 and 0.7742, respectively. An average baseline difference of 4,052 seconds is the accuracy measure to beat in order to improve the model.

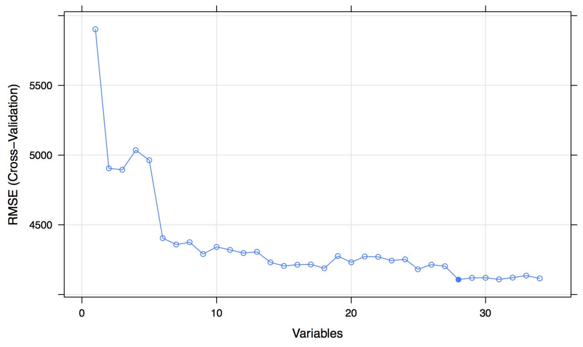

Fig. 1: Recursive feature selection.

B. Model Improvements

Based on our experience, this CV RMSE value is likely to increase when testing on unseen data. We explored other predictive analytic techniques to see if we can get a better prediction model with a lower RMSE and a higher R^2.

a) Eliminating Highly Correlated Features: In machine learning, features with high correlation can have an adverse effect on the predictive model accuracy [5]. Thus, we employ this technique to prune out features that are highly correlated. To do this, we set a correlation cut-off score of 0.7 (i.e., values about 0.7 or below -0.7) to indicate highly correlated variables. We employ the findCorrelation() function of the Caret package to find and remove highly correlated features. We use the subset of 10,000 records from the training set to compute the correlation matrix. The features *gh_src_files*, *tr_tests_ok*, *gh_test_cases_per_kloc*, and *gh_test_lines_per_kloc* had a correlation index greater than 0.7 or less than -0.7, hence they are removed from the model. This reduced the number of features to 31. However, we did not get a noticeable improvement in the prediction accuracy by eliminating these variables.

b) Recursive Feature Selection (RFE): RFE is an automatic method of selecting features for a predictive model based on their relative importance. RFE is defined as a wrapper method, since it samples the features as it analyzes the interactions between increasing subsets of features to determine the relative impact of each feature in the presence of others. This can also be viewed as a brute-force method; it is computationally expensive. We used the Random Forest implementation of RFE to search and identify the best feature space that can improve our model accuracy. The results of implementing this automatic selection method are shown in Figure 1. We hoped to achieve a slightly better model by reducing the feature space to 28 variables. However, there was no statistically significant difference in the evaluation metrics.

C. Prediction Accuracy

Finally, we applied our baseline models on the test set to see how our model performs on out-of-sample data points. Typically, we expect our test set errors to predict slightly worse than our train set error estimates. This is because our train set can sometimes give us overly-optimistic prediction values. The test set was constructed by sampling 10,000 unseen records from the original 791,310 records. We used each of the final

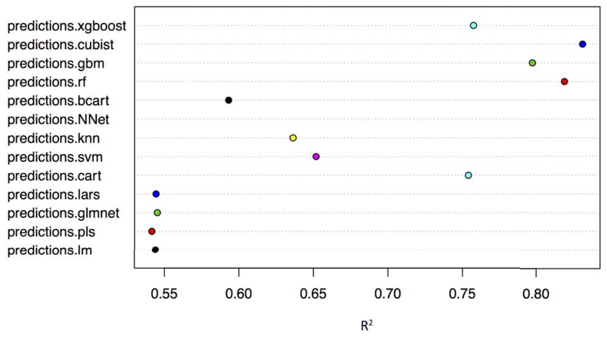

Fig. 2: Test-set prediction accuracy (R^2).

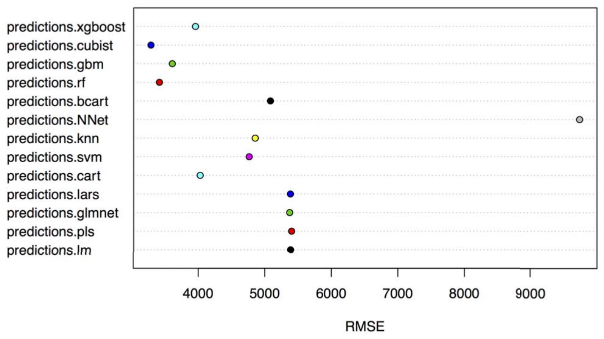

Fig. 3: Test-set prediction accuracy (RMSE).

models to predict the build duration of the test data. The model estimates were compared to the original *tr_duration* value of the test set using RMSE and R^2 as evaluation metrics.

Figures 2 and 3 present the test-set RMSE and R^2. As expected, Cubist (CB) and Random Forest (RF) outperform the others with a lower RMSE and higher R^2. This R^2 metric is particularly very encouraging, because it shows us that a high percentage of variance in the prediction accuracies is accounted for by the model.

V. Discussion and Conclusion

Wallace et al. [10] suggest that as the number of developers increases, the project size and complexity also increases. This, in turn, escalates the wait time of a build job. In addition, frequent integration enables a minimal number of changes in software artifacts (e.g., lines of code changed, files added, files deleted, files modified, etc.) to be constantly integrated into the main codebase.

Developers tend to lose focus and productivity while waiting for code to build [6]. Such distraction has a detrimental effect to the idea of continuous integration that advocates for frequent builds to commit changes to the central codebase fast and early. However, as the projects expands, frequent integration can become a huge impedance to productivity. For example, if a build takes approximately 1 hour or more, integrating more than once a day can become a setback to developer speed and efficiency.

The related work proposes numerous techniques to balance the trade-off between build waiting time and the need to continuously integrate code. Our research project comes in the middle to further balance this trade-off. We developed a predictive model to approximate the build time of a build job in a CI environment.

Developer Productivity. When the approximate time for a build is known, developers can better plan how they spend their time and prioritize their tasks. As a result, developer efficiency and productivity can be increased. For example, if a build is expected to take a long time, then developers may move on to other tasks such as responding to e-mails or code reviews.

Project Management. Project managers can also benefit from knowing the approximate build time beforehand. They can become more strategic on how best to manage the CI process to balance the anticipated build wait time versus the need to continuously commit changes to the central codebase. Such knowledge can be particularly crucial for large projects. Project managers can explore different factors to manage build jobs keeping the build wait time within the needed time frame.

CI Adoption. Software organizations looking to adopt CI can use our prediction model. As organizations transition to CI practices, they may want to estimate their build times. Making process changes that favourably reduce build times may be easier when CI practices are not fully implemented within an organization. The lack of constraints from a CI environment may allow organizations to quickly make changes to reduce build times before CI is fully adopted.

Research and Industrial Relevance. Finally, other researchers can use and modify the proposed prediction model to further improve build times. Researchers can further study the factors and relationships affecting build times. The prediction model can be expanded to include a dataset other than TravisTorrent. Our research has a direct and immediate relevance to practitioners, and further strengthens the concept of continuous integration, which consequently increases continuous delivery in the widely adopted agile development model.

A. Study Limitations

Our study had several limitations that mitigated the improvement of our predictive model. The main obstacle was the sheer computation power required to perform our analysis. The TravisTorrent dataset contains approximately 1.76GB of data with more than 2 million observations.

Additionally, we considered a variety of learning algorithms, which comprised of a combination of linear, non-linear, and ensemble methods. Some of the learning algorithms are computationally expensive: this is especially true of kernel methods such as support vector machines and ensemble methods like random forest and stochastic gradient boosting. These algorithms took over 8 hours each for a single run taking into consideration the fact that our original dataset was subsampled from 1,846,396 records to 10,000 records for the training set.

Typically, the predictive accuracy of a model is improved when more data is available. We could not take advantage of the big data available to train our model due to the lack of computational resources. We strongly recommend the adoption of Apache Spark/Hadoop for further analysis.

B. Demo Application

We hope the feature space can be further reduced for us to implement a tool for practitioners that predicts the build times of various GitHub projects running Travis as a CI platform. We have implemented a demo application as a proof of concept. The application can be viewed at https://goo.gl/kCbrhE. The tool can be implemented as an IDE plugin or as an add-on in a planning/scheduling platform for developers.

In our demo app, we used a trained Cubist model. We formulated a sample test by using as input the values of team size, lines of production code changed, test code changed, files added, deleted or changed and the number of jobs contained in the build. The remaining 30 variables were estimated by using the means of the values in the test data sample. We hope that future studies can reduce the feature space to a small set of relevant features that can minimize RMSE and maximize the R^2 error metric.

VI. Future Work

While our work has several limitations, it provides a foundation for further investigation of build time prediction. Possible extensions of this work are:

- Identify cases in which the build fails. It is helpful for developers to know about these "corner cases" early enough, as this usually has a significant impact on the build time.
- It would be interesting to investigate whether build time predictions vary between different programming languages (i.e., Ruby or Java).
- Further work can harness deep learning methods and tools to automatically learn useful feature representations for build time prediction models.

References

[1] G. Ammons. Grexmk: Speeding up scripted builds. In Proc. of the 2006 Int. Workshop on Dynamic Systems Analysis, p 81-87. ACM, 2006.

[2] M. Beller, G. Gousios, and A. Zaidman. TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration. In Proc. of the 14th working conference on Mining Software Repositories, 2017.

[3] G. Brooks. Team pace keeping build times down. In Agile Conference, p 294-297. IEEE, 2008.

[4] E. Dolstra and E. Visser. The nix build farm: A declarative approach to continuous integration. 2008.

[5] P. Domingos. A few useful things to know about machine learning. Volume 55, p 78-87. ACM, 2012.

[6] P. Kainulainen. The cost of context switching. https://www.petrikainulainen.net/software-development/processes/the-cost-of-context-switching.

[7] E. Laukkanen and M. V. Mantyla. Build waiting time in continuous integration: an initial interdisciplinary literature review. In Proc. of the Second Int. Workshop on Rapid Continuous Software Engineering, p 1-4. 2015.

[8] M. Meyer. Continuous integration and its tools. In IEEE Software, volume 31, p 14-16, 2014.

[9] A. Mokhov, N. Mitchell, S. Peyton Jones, and S. Marlow. Non-recursive make considered harmful: Build systems at scale. In Proc. of the 9th Int. Symposium on Haskell, p 170-181. ACM, 2016.

[10] L. Wallace and M. Keil. Software project risks and their effect on outcomes. Communications of the ACM, 47(4):6873, 2004.