Predicting Continuous Integration Build Outcome

Project Report (Draft)

Johannes Kästle
University of Alberta
kaestle@ualberta.ca

Abstract

In order to improve developers productivity and increase development pace, it is tried to predict the outcome of continuous integration builds. For that, $2,500,756$ builds with 37 features were analyzed, using three different machine learning algorithms with over 700 configurations. Generally, all classifiers struggled with predicting failures, but had high sensitivity. At the end, the decision tree classifier could predict the outcome with an accuracy of 76% and performed significantly better than its competitors.

1 Introduction

In modern software engineering using continuous integration has evolved to a best practice. Its goal is to archive higher productivity while developing and reduce the number of faults and bugs. Thanks to its interweaving with GitHub, TravisCI is commonly used for open source projects. Therefore, there is a lot of continuous integration data is available. It has been shown that broken builds delay a project significantly [1], hence the prediction if a build will fail and why it does so can increase development pace. With this information, it is possible to develop tools and techniques which can reduce number of failing builds.

In previous researches it was shown that there are correlations between build metrics and outcome [2], and that cascading classifiers work well for predicting the build outcome [3]. Another research tried to predict build outcome with decision trees, as well, using data from previous builds [4]. This work wants to try a simpler approach and see if it is possible to predict the build outcome only using the current's build information and its associated commits. For this, the TravisTorrent [5] dataset is analyzed in order to classify the build outcome. Then, three different algorithms, naive Bayes, neural networks and decision trees, are used to classify this dataset and it is evaluated if one of them is able to accurately learn this problem.

2 Background

TravisCI is a continuous integration tool to automated compile, build, integrate and test GitHub projects.

2.1 Dataset

The TravisTorrent[5] dataset was created for the MSR 2017 challenge. This study uses the version from January 11, 2017 which contains $3,702,595$ samples. A detailed explanation of all the features can be found on the homepage of TravisTorrent[1]. Because not all features are useful for this study, the number of features was restricted to some features. A list of all 37 used features can be found under table A1. Some examples for features are the language the project is written in, the number of people who actively commit to this project, how many files were added, how long did it take to build the project, the number of words in the pull request, or how many tests ran or failed.

Only builds that were either successful, erroneous or failed were considered. Successful builds were labeled as class 1, or true, while the other builds were marked as failed, i. e. class 0, or false. Aborted or canceled builds were ignored, therefore only $2,500,756$ builds were taken into the final dataset.41.6% of the builds did fail in the final set.

2.2 Algorithms

A mean classifier is used as baseline which all algorithms need to compare to. It labels all data to the most frequent class. Then, three supervised classification algorithms are used. The first one is Naive Bayes which assumes every feature to be Gaussian data. This is a very strict assumption and it is known that not all features will satisfy this. Therefore, it is assumed that this algorithm will not perform as good as the other two.

The second one is a neural net with one hidden layer and the logistic activation function. Adam[6], an optimized stochastic gradient descent, is used as solver for weight optimization. The batch size is determined automatic, the maximum epochs are set to 200 and the dataset is shuffled between each epoch. The neural network is chosen because it is currently used in a lot of modern learning and, hence, to see if it can live up to its expectations.

We use a decision tree as third algorithm. A decision tree uses subsequent rules to classify the data. For example, the first node could be if the programming language is Java. If that is the case, the sample goes to the left child, otherwise to the right child. This is done until the sample is classified at a leaf node. The criterion for splitting is set to gini impurity, for performance it takes the best random split. This algorithm is chosen because previous research has found it useful.

3 Methodology

The dataset is stored in a MySQL database. From there all samples are extracted and stored in a matrix. The build outcome is represented in the vector y, with value 1 if the build was successful and 0 otherwise. The 'language' and 'previous build' features are stored as a vector; for example the language 'java' is encoded into $[1, 0, 0]$, while 'javascript' is encoded to $[0, 1, 0]$. The date features are split up into month, day, daytime in seconds and weekday as integers.

Since not every sample contains all features, the missing features are set to their respective mean, so it does not impact the classification. 7 features miss each about 20% data. For

[1] `https://travistorrent.testroots.org/page_dataformat/`

another 4 features there is data for only 15% of the samples, whereof 3 features can only occur in pull requests and the fourth feature is the build duration. Categorical features did not have missing data.

The actual learning is divided into two parts. First, the best parameters for the neural network and the decision tree have to be learned, and afterwards the algorithms need to be compared. Because the dataset is big enough, it is split up in two equal halves, one for every of the two tasks. The split and all following splits are done stratified. This complete split is used so that the parameter search does not bias the inter algorithm comparison.

The used scripts and results can be found in the author's GitHub repository[2].

3.1 Parameter

To look for the best hyper parameters, the first half of the dataset is used for an exhaustive grid search. This cross validation learns a model for every possible combination of the parameters. This is done on a K-Fold with $k = 5$. A list of all tried parameters can be found in tables B1 and B2. The parameters were chosen in a way, that they either consider a large range of the possible values (e. g. beta can range from 0 to 1, and we chose 0.1, 0.5 and 0.9) or have worked well in previous test runs on a smaller set. A detailed description of all possible parameters can be found in the documentation of sklearn for the decision tree[3] and the neural network classifier[4]. Accuracy is used as scoring method.

3.2 Cross Algorithms

The other half of the data is used to compare the different algorithms. The reasoning for this split is to have an unbiased estimation for the cross algorithm comparison. The learning of the hyper parameters is done completely separate from the comparison and, hence, cannot influence it. This half of the data is sampled into ten random sets, where the proportion of test data is 20%. We run each of the three and the baseline algorithm on every split. Thus we get 10 different runs to account for lucky runs and over fitting. For evaluation, the confusion matrix is saved for every run. Then the means and standard deviation over the runs is computed for every algorithm.

3.3 Statistics

The algorithms are compared using Welch's t-test, because the variances of the algorithms are different. A t-test is satisfactory to show if the algorithms have significant changes in their performance measurements. The baseline algorithm has, as expected, a variance of 0, whereas in test runs the neural networks had generally a high standard deviation. As comparing metrics accuracy, sensitivity (recall) and specificity are used, in which the latter ones are equivalent to the AUC of the ROC curve. Those three metrics are used to give a general sense of how good the algorithms are performing and then in detail how good

[2]`https://github.com/jodokae/cmput551-mini-project/`

[3]`http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html`

[4]`http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html`

they are at predicting successful and failing builds separately. Precision is not considered because we are more interested in failing builds (condition negative) than in the positive values. Therefore, sensitivity and specificity are better measurements how good we can predict a build in its correct class.

The two tailed t-test is done pairwise for every algorithm with $H_0 : \mu_0 = \mu_1$ and $\alpha = 0.05$ to produce a ranking between the three (plus baseline) algorithm.

4 Results

The methodology was applied to the full dataset. The following results were found:

4.1 Parameters

When running on the first 50% of the dataset, 243 different parameter combinations were fitted on five folds for the neural nets. This resulted into 1215 different learnings. With the highest accuracy of 70.78%, the parameters with the highest number of hidden units and the smallest alpha were chosen. The learning rate was with 0.001 the median value, as well as the beta_1 parameter, whereas beta_2 was given its highest value.

For the decision tree classifier, 486 different parameter sets were tried. On five folds, which were not necessarily the same as for the neural net, this resulted into 2430 different decision trees. The best parameters found were with a maximum depth of 100, the split and leaf ratios both at $1e{-}5$ and the maximum leaf nodes as $10,000$. As maximum features parameter, the logarithm was taken. This setting achieved an accuracy of 77.63%.

Apart from max_features all parameters were chosen so that they give the least constraints. This gives the algorithm the most freedom to produce the best possible tree, but has the highest probability of over fitting. Only for max_features a higher constraint resulted into better results.

Table 1: Confusion matrices

(a) Baseline Algorithm

		Actual		
		S	F	$\sum$
Predicted	S	176620 ± 0	73456 ± 0	250076
	F	0 ± 0	0 ± 0	0
	$\sum$	176620	73456	250076

(b) Naive Bayes

		Actual		
		S	F	$\sum$
Predicted	S	173879 ± 386	71402 ± 240	245281
	F	2741 ± 286	2054 ± 240	4795
	$\sum$	176620	73456	250076

(c) Decision Tree

		Actual		
		S	F	$\sum$
Predicted	S	167435 ± 1617	50522 ± 4288	217957
	F	9185 ± 1617	22934 ± 4288	32119
	$\sum$	176620	73456	250076

(d) Neural Network

		Actual		
		S	F	$\sum$
Predicted	S	174003 ± 1766	71159 ± 1441	245162
	F	2617 ± 1776	2297 ± 1441	4914
	$\sum$	176620	73456	250076

Table 2: Metrics

Algorithm	Accuracy	Sensitivity	Specificity
Baseline	70.6%	100%	0%
Naive Bayes	70.4%	98.4%	2.80%
Decision Tree	76.1%	94.8%	31.2%
Neural Network	70.4%	98.5%	3.12%

4.2 Cross Algorithms

On the other half of the dataset, the algorithms were compared. Every algorithm was run with their optimized parameters on every of the ten random splits. The averaged results can be found in table 1.

Obviously, the baseline algorithm just predicted every element to the class with the highest probability, which was the 'Success' class. Therefore, it has never predicted 'Failure'. The decision tree classifier has got very high standard deviation for both classes, especially the prediction for samples, which were failures, deviate at around 1.7%. This is explained due to the hardly constraining features.

As can be seen in table 2 the algorithms don't seem to differ that much in terms of accuracy, apart from the decision tree which is ahead with almost 6 per cent points. This is probably the case because the decision tree is the only algorithm which has an acceptable specificity rate. Although 31% is high in comparison, it is still a low value.

To confirm the feeling that the decision tree is the best algorithm in this case, a two tailed t-test was executed for all three metrics pairwise between all algorithms. The null hypothesis, that the algorithms have the same mean, are therefore equal, is tested with $\alpha = 0.05$. The full results can be seen in tables 3 - 5. Since the baseline algorithm achieves a full score in sensitivity and zero score at specificity by its nature, the t-test for those metrics is only used for comparison of the algorithms under study.

According to accuracy (table 3) the naive Bayes classifier performs worse than the baseline, the neural net performs equally good and only the decision tree can achieve a higher accuracy than the mean predictor. Neural net and Naive Bayes are not statistically different in direct comparison. In terms of sensitivity (table 4) the tree performs the worst. Neural net and Bayes do not differ again, this does not change for specificity (table 5), as well. The decision tree excels considerably.

With the results of the t-tests, the decision tree classifier algorithm is considered as winner over the others. Despite losing in sensitivity, the difference is only 4%, it wins in classifying failures where it is 10 times more accurate than the other two algorithms. This is confirmed by the t-test of accuracy. The other two algorithms are not worth using, given that they do not outperform the baseline algorithm.

5 Threats to validity

In this project not all possibilities which influence the strength of the results could be checked. First of all, because of lack of time (one run of the complete methodology takes

Table 3: t-test accuarcy (t,p)

	Baseline	Naive Bayes	Decision Tree	Neural Network
Baseline	$(0, 1)$	$(12.77, 4.52 \times 10^{-7})$	$(-13.69, 2.49 \times 10^{-7})$	$(1.009, 0.339)$
Naive Bayes	$(-12.77, 4.52 \times 10^{-7})$	$(0, 1)$	$(-14.35, 1.56 \times 10^{-7})$	$(-1.145, 0.28)$
Decision Tree	$(13.69, 2.49 \times 10^{-7})$	$(14.35, 1.56 \times 10^{-7})$	$(0, 1)$	$(13.36, 4.82 \times 10^{-8})$
Neural Network	$(-1.009, 0.339)$	$(1.145, 0.28)$	$(-13.36, 4.82 \times 10^{-8})$	$(0, 1)$

Table 4: t-test sensitivity (t,p)

	Baseline	Naive Bayes	Decision Tree	Neural Network
Baseline	$(0, 1)$	$(21.29, 5.22 \times 10^{-9})$	$(17.04, 3.71 \times 10^{-8})$	$(4.445, 0.00161)$
Naive Bayes	$(-21.29, 5.22 \times 10^{-9})$	$(0, 1)$	$(11.63, 3.84e \times 10^{-7})$	$(-0.2063, 0.841)$
Decision Tree	$(-17.04, 3.71 \times 10^{-8})$	$(-11.63, 3.84 \times 10^{-7})$	$(0, 1)$	$(-8.229, 1.73 \times 10^{-7})$
Neural Network	$(-4.445, 0.00161)$	$(0.2063, 0.841)$	$(8.229, 1.73 \times 10^{-7})$	$(0, 1)$

Table 5: t-test specificity (t,p)

	Baseline	Naive Bayes	Decision Tree	Neural Network
Baseline	$(0, 1)$	$(-25.65, 1 \times 10^{-9})$	$(-16.05, 62.7 \times 10^{-8})$	$(-4.785, 0.000995)$
Naive Bayes	$(25.65, 1 \times 10^{-9})$	$(0, 1)$	$(-14.59, 1.35e \times 10^{-7})$	$(-0.4998, 0.629)$
Decision Tree	$(16.05, 6.27 \times 10^{-8})$	$(14.59, 1.35 \times 10^{-7})$	$(0, 1)$	$(13.69, 2.96 \times 10^{-8})$
Neural Network	$(4.785, 0.000995)$	$(0.4998, 0.629)$	$(-13.69, 2.96 \times 10^{-8})$	$(0, 1)$

over ten hours on a standard laptop), the complete project, i.e. finding parameters with cross validation and compare algorithms on different splits, was only done once and not on separate splits of the complete dataset. To compute more splits would take multiple days. This was also necessary, because different runs would produce various optimal parameters, because a dissimilar training set can be learned better with other parameters, which would then harden the conclusion of the comparison. Because the dataset is quite large, this issue is considered to be acceptable.

Another runtime related problem is that not all possible parameters were considered. For example, this work restricts itself to the gini impurity for the decision tree. Hence, it cannot be generalized that decision trees are strictly better than neural networks for this dataset, but only under the tested parameters. The terms 'better' and 'winner' are only regarding accuracy, sensitivity and specificity and do not consider runtime or other metrics at all. Because this work only wants to compare the possibility of predicting this type of data at all, runtime was considered to be a less important aspect.

Because the dataset only is a small snapshot of all CI projects out there, especially restricted to only three programming languages and open source projects, it cannot be generalized that build outcomes are predictable. But it is believed, that the considered features do not differ significantly between closed and open source projects. Furthermore, the dataset contains missing data for some features, which may impact the outcome of the analysis. This was mitigated due to preprocessing of the data. In future, it might be interesting to see if the accuracy of the prediction can be improved using feature selection.

6 Conclusion

In order to predict build outcomes of continuous integration builds, it was found, that a decision tree classifier can predict the outcomes with an accuracy of 76%. Although it has high variance in its learning, it still outperforms naive Bayes and neural networks with one hidden layer, by far. That Naive Bayes could not outperform the baseline was expectable, because the dataset does not fulfills all assumptions. But that a neural net did equally bad, is a big surprise. Of course not all possible parameters were tested, but it gives a general idea in which direction should be looked at. The biggest problem is optimizing the prediction of failing builds, which have only a specificity of 31%.

References

[1] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 41–50, Sept 2014.

[2] M. R. Islam and M. F. Zibran, "Insights into continuous integration build failures," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 467–470, May 2017.

[3] A. Ni and M. Li, "Cost-effective build outcome prediction using cascaded classifiers," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 455–458, May 2017.

[4] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, (Washington, DC, USA), pp. 189–198, IEEE Computer Society, 2006.

[5] 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.

[6] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

A Features

Table A1: Feature list

feature	type
gh_lang	[java, javascript, ruby]
gh_num_commits_in_push	int
git_prev_commit_resolution_status	[build_found, merge_found, no_previous_build]
gh_team_size	int
git_num_all_built_commits	int
gh_num_issue_comments	int
gh_num_commit_comments	int
gh_num_pr_comments	int
git_diff_src_churn	int
git_diff_test_churn	int
gh_diff_files_added	int
gh_diff_files_deleted	int
gh_diff_files_modified	int
gh_diff_tests_added	int
gh_diff_tests_deleted	int
gh_diff_src_files	int
gh_diff_doc_files	int
gh_diff_other_files	int
gh_num_commits_on_files_touched	int
gh_sloc	int
gh_test_lines_per_kloc	double
gh_test_cases_per_kloc	double
gh_asserts_cases_per_kloc	double
gh_by_core_team_member	double
gh_description_complexity	int
gh_pushed_at	date
gh_build_started_at	date
tr_duration	int
tr_log_bool_tests_ran	double
tr_log_bool_tests_failed	double
tr_log_num_tests_ok	int
tr_log_num_tests_failed	int
tr_log_num_tests_run	int
tr_log_num_tests_skipped	int
tr_log_tests_failed	int
tr_log_testduration	double
tr_log_buildduration	double
tr_status	[errored, failed, passed]

B Parameter

Table B1: Parameters for Decision Tree Classifier

parameter	values	description
`max_depth`	`[1, 10, 100]`	maximum depth of the tree
`min_samples_split`	`[1e-5, 1e-4, 1e-3]`	minimum fraction of samples needed on both child nodes
`min_samples_leaf`	`[1e-5, 1e-4, 1e-3]`	minimum fraction of samples per leaf
`min_weight_fraction_leaf`	`[1e-5, 1e-4, 1e-3]`	minimum weighted fraction of samples per leaf
`max_features`	`['sqrt', 'log2']`	maximum amount of features considered per split, either $\sqrt{\|features\|}$ or $\log_2 \|features\|$
`max_leaf_nodes`	`[100, 1000, 10000]`	maximum number of leafs in tree

Table B2: Parameters for Neural Network Classifier

parameter	values	description
`hidden_layer_sizes`	`[(10,), (50,), (100,)]`	number of hidden nodes
`alpha`	`[1e-4, 1e-3, 1e-2]`	l2 regularization parameter
`learning_rate_init`	`[1e-3, 1e-2, 1e-1]`	step size
`beta_1`	`[0.1, 0.5, 0.9]`	adam specific parameter
`beta_2`	`[0.1, 0.5, 0.9]`	adam specific parameter