An Analysis of Travis CI Build Failures

Sara Ghaemi

I. Introduction

In the past few years, continuous integration has become very famous since it automates the process of software testing and helps identify bugs faster. Build failures in such systems are a signal that there has been a problem in the development process. These failures usually need rework, and they may cause a delay in the project process. As a result, finding the reasons that continuous integration builds fail can improve the whole process. Moreover, the result of such research can help developers grow habits that in the long term reduces their development time. Among continuous integration tools, Travis CI is used in most open-source projects since it works well with GitHub and it is open-source itself.

In this project, we focus on Travis CI build jobs to find the main causes of failure in them. The goal is to identify these causes and help developers grow habits that result in fewer build failures. Although others have also tried to find these causes, this project is unique as it works on a large dataset and tries to answer questions that have not yet been answered. These research questions are as following:

- **RQ1:** What are the factors that influence Travis CI build status in general?
- **RQ2:** Does the average time between commits have a relationship with build failures?
- **RQ3:** Does the number of lines of codes that are changed influence the build status?
- **RQ4:** Is the size of a team related to build failures?
- **RQ5:** How are each of these factor influencing build failure?

To answer these questions, we use TravisTorrent [1] dataset which is an open database of Travis CI [2] build jobs. We use two prediction models, logistic regression and random forest, to learn the behaviour of the dataset and find the most important features influencing the build failure by analyzing the statistical features that are provided to us by these models. After finding the influential features, we use bootstrapping and different kinds of plots to find a meaningful relationship between build status and each feature.

Following are the main contributions of this project:

- Building an accurate prediction model which can both help us better understand the build failures, and be used to predict whether a job is going to fail or not.
- Finding whether the average time between commits, number of lines of code that are changed, and team size influence the build failure.
- Providing some insight on how to prevent build failures.

II. Related Work

Other researchers have also worked on different factors that influence the build failure. Souza et al. [3] have considered the commit messages of each build in Travis CI to find a relationship between the developers sentiment and build failure. Santos et al. [4] found that the unusualness of a commit is correlated with build failure. To find the unusualness of a commit, they have used cross-entropy of its message with respect to a language model. Rebouças et al. [5] focused on finding a relationship between contributors involvement in the project and build failures. They found that in 85% of cases there is no indication that a casual developer causes more failed builds. Jain et al. [6] evaluated the influence of team size on build failures. They found that an increase in team size causes an increase in build failure. Islam et al. [7] have studied the impact of the complexity of a task, size of a project and size of a team on build failures. They have found that complexity of a task influences the build status but project and team size has no impact on it. This result is against what was proposed in [6]. This is one reason we are interested in studying the impact of team size on build status. In [8], Luo et al. focus on finding which features most affect the build result. They have concluded that the number of commits, number of files changed and the density of tests are the most influential features. This paper is the closest study to what we have done in this project. However, they have only considered 45 projects and we study about 1200 projects.

III. Background and Terminology

Continuous integration(CI) is the practice of frequent commits to a source repository, after which automated builds and tests are run. In other words, in this practice, developers are encouraged to commit their changes more often to the central repository, and the builds and tests are automated. A CI server runs the tests and builds, and reports the status of every commit. Although the developers should commit often, they should try not to commit broken code and run tests locally before committing to the central repository. CI helps developers find bugs faster and easier which in long term results in a higher quality production code. Moreover, since this process is automated on a server, it reduces the assumptions about the code environment. [9] This practice also helps teams deliver updates faster since team members should commit changes more often.

Many continuous integration tools are available to developers among which Jenkins [10], Team City [11], and Travis CI [2] are some of the most famous ones. Since an open database containing about 3.7 million Travis CI build jobs' information

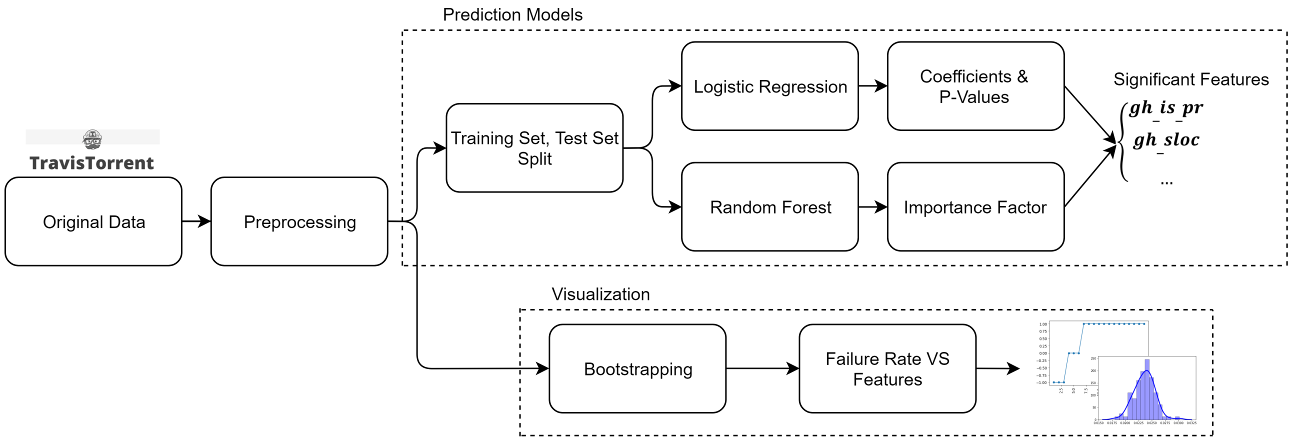

Fig. 1. An overview of the methodology

named TravisTorrent [1] is available online, we focus on Travis CI builds in this project.

Some principal methodology used in Travis CI which is going to be used in this paper are as follows:

- Phase: The sequential steps of a job. (e.g., install, script, deploy)
- Job: The process of automatically cloning a repository into a virtual machine and going through some phases.
- Build: A group of jobs.

IV. Dataset

The dataset used in this project is the latest version of TravisTorrent [1], released on Feb 8, 2017. This dataset was gathered for the MSR'2017 Mining Challenge by combining Travis CI API data, an analysis of the build logs, and repository and commit data from GitHub or GHTorrent [12], a dataset containing GitHub repository information. Each feature's name in this dataset starts with git_, gh_, or tr_ which represent the source of the feature being GitHub, GHTorrent, and Travis CI respectively.

This dataset has about 60 features for about 3,702,595 Travis CI build jobs which belong to 1283 different projects. Some of the most important features included in this dataset are the job and build ID, team size, the branch that was built, production source lines of code, etc. Explanation on which features are used in this project and why can be found in section V-A.

V. Methodology

This project has three main parts, preprocessing, prediction models, and visualization. The five research questions are answered mainly based on prediction models and visualization. Figure 1 shows an overview of the methodology of this project. This project is available on GitHub[1].

[1]https://github.com/cmput402-w19/project-saraghaemi

A. *Preprocessing*

Since the dataset mentioned above has lots of problems, a large proportion of the development time was spent on preprocessing. Most of the preprocessing step is done in *Python3* using *Pandas* and *scikit-learn* libraries, and the last step is done in R. Before running any preprocessing, there are 3,702,595 rows of data in the dataset. In each step, some data points are removed, and the next step works on the new dataset. Figure 2 shows an overview of the preprocessing steps and how the data size is changed in each step.

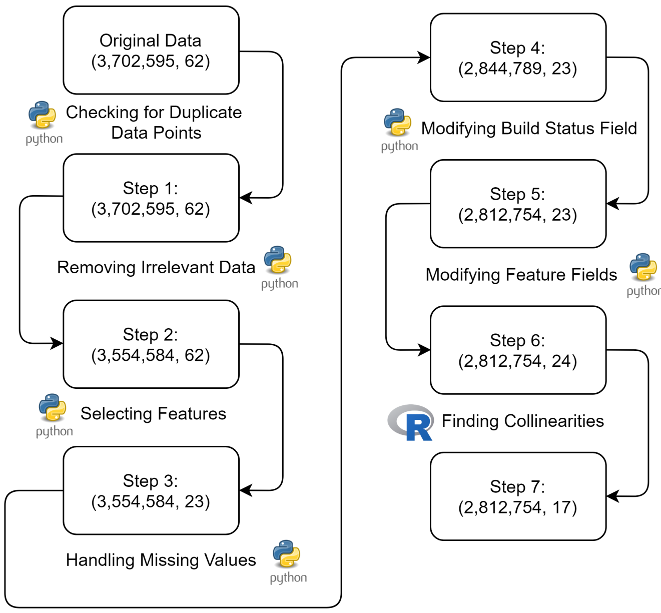

Fig. 2. An overview of the preprocessing steps

1) Checking for Duplicate Data Points: In the very first step of the project, we should check if there are any duplicate data points in the dataset. To do so, we look at the unique values of Travis CI build job ID field and compare this number to the total data points. Since these two numbers are equal, we can conclude there are no duplicate rows in the dataset.

2) Removing Irrelevant Data: We should consider removing rows of data that are not relevant to this project, or seem to be an error in data collection. The number of data points with each characteristic reported below is the total number of rows with that characteristic in the whole dataset.

In this project, we would like to have build jobs in the dataset that have a build duration time of more than zero. Projects that have a negative build duration time are errors in data collection and those with zero time do not give the prediction model any information about build status. There are 920 rows of data with build duration time less than or equal to zero.

Moreover, data points with a team size less than or equal to zero are definitely due to some errors in data collection. There are 4,855 rows with such a characteristic.

There is a feature in the dataset that shows the duration of running the tests, in seconds, extracted by build log analysis. We use this feature to remove the data points which have a running test duration of less than or equal to zero. Similar to build duration, rows with duration less than zero are errors in data collection and the ones with duration equal to zero are not valuable to the prediction model. 142,281 data points are found with running test duration of less than or equal to zero.

After removing the irrelevant data, there are 3,554,584 remaining data points in the dataset.

3) Selecting Features: Not all the features in the dataset are useful for this project. In order to find the reasons builds fail, all features that may influence the build status are manually selected from the dataset. In this step, 23 features are selected which later would be checked for collinearity.

4) Handling Missing Values: There are a couple of strategies to deal with missing data in a dataset such as removing them from the dataset, replacing them with the average value of the column, replacing them with the median of the columns or finding the distribution and replacing the missing value based on that. Depending on the dataset size, the feature that has the missing value and the project, each of these methods may be suitable. After selecting the features, there are 709,795 data points with missing values in two features: time of the push that triggered the build and the number of commits included in the push that triggered the build. In our case, replacing the missing values may cause a major problem since one of the features is time and cannot be replaced with either the mean or the median of the values. Therefore, in this project, all missing values are removed. After this step, we end up with 2,844,789 data points.

5) Modifying Build Status Field: In TravisTorrent dataset, build status field can be *passed*, *failed*, *errored*, or *cancelled*. Builds that are *cancelled* would not provide the learning algorithm with any specific information. Moreover, *failed* and *errored* are not different from the perspective of this project. We would like to know what causes the build to either fail or receive an error. As a result, data points with *cancelled* build status are deleted, and those with *errored* build status are labeled as *failed*. At the end of this step, there are 2,812,754 rows in our dataset, and the target value is either *passed*, or *failed*.

6) Modifying Feature Fields: Some feature fields also need some modification. The time of the push that triggered the build is a string which is converted to timestamp. The branch that was built is the name of the branch which is not important to us. The only thing that may influence the build status is whether the master branch is built or another branch. Therefore, we delete the branch feature and add another one named is_master instead. In addition, all non-numeric features are converted to numeric values using one-hot encoding, and all integer fields are converted to float. Finally, a new random feature is added to the dataset. This feature will later be used for comparing the importance of features. Any feature with smaller importance value than random feature is not significantly important.

7) Finding Collinearities: Having features that are linearly dependant on each other can mess with the learning process. For instance, in feature importance calculations, if there are two features linearly dependant on each other, only one would have a correct significance value, and there is no way of understanding which one is correct. As a result, it is important to spot the linearly dependent features and remove one of them.

This can easily be done in R programming language by *car* package. The logistic regression model in R has a characteristic called variance inflation factor (VIF) which shows the collinearity of each feature with other features. The VIF number for each feature shows how much worse the standard error is compared to what it would be if the feature was independent of other features.

In this project we iteratively fit a logistic regression to the dataset, find the collinearities, delete the linearly dependant features and do this process again. We keep monitoring the VIF values untill none of the features are linearly dependant on others. After doing so, we have 16 independent features which are described in Table I.

B. Prediction models

In this project, we need a prediction model for a binary classification task. We would like this model to both be a good prediction model, and accurately report the most important factors influencing the build status. As a result, two prediction models, random forest, and logistic regression are used. Most of the prediction model codes and the generated results are written in Python except for logistic regression P-values and coefficients.

Random forest is used because it is capable of learning complicated relationships based on decision trees. Since we have a huge dataset with features that may have a complicated impact on the results, random forest is a good choice. As we will see in sections V-B1, this model reaches a high classification accuracy and also has a promising F1-Score,

TABLE I
FINAL FEATURES

Feature	VIF	description
gh_lang	1.102	Dominant repository language.
gh_sloc	1.276	Number of executable production source lines of code.
gh_is_pr	1.192	Whether this build was triggered as part of a pull request.
gh_pushed_at	1.055	Time of the push that triggered the build (GitHub provided), in UTC
gh_num_commits_in_push	1.122	Number of commits included in the push that triggered the build.
gh_num_commit_comments	1.009	The number of comments on all the commits that were built for this build.
gh_team_size	1.169	Number of developers that committed directly or merged PRs from the moment the build was triggered and 3 months back.
git_diff_src_churn	1.144	Number of lines of production code changed in all the commits that were built for this build.
git_diff_test_churn	1.098	Number of lines of test code changed in all the commits that were built for this build.
gh_diff_doc_files	1.06	Number of documentation files changed by all the commits that were built for this build.
gh_diff_other_files	1.073	Number of files which are neither source code nor documentation that changed by the commits that were built.
gh_asserts_cases_per_kloc	1.082	Test density. Assert density. Number of assertions per 1000 executable production source lines of code.
gh_by_core_team_member	1.093	Whether a core team member authored this commit or not. A core team member is someone who has committed code at least once within the three months before this commit.
gh_num_commits_on_files_touched	1.216	Number of unique commits on the files touched in the commits that were built for this build that triggered the build from the moment the build was triggered and 3 months back. It is a metric of how active the part of the project is that these commits touched.
git_prev_commit_resolution_status	1.203	When walking backwards the branch to find previously built commits, what is the reason for stopping the traversal? Can be one of: build_found: when a previous build is found, or merge_found: when traversal is stopped at a merge point or no_previous_build
is_master_branch	1.102	Whether this commit was on the master branch or not.
random	1	Random numbers generated to compare with significant features.

and area under ROC curve[2] value. However, random forest finds such complicated relationships that we might not be able to understand them easily. Although the significant features reported by random forest are important for prediction model, we might not be able to interpret them to understandable results and use them to draw conclusions on how developers should behave to prevent CI build failures.

Because of the aforementioned limitations of random forest, we have also used logistic regression to find important features. Since logistic regression is a simple linear model, its prediction accuracy would not be as good as random forest. However, since logistic regression finds simple relationships between features and targets, the relationships are more interpretable as opposed to random forest.

As a baseline, a stratified classifier has also been implemented. This classifier generates predictions with respect to the training sets class distribution.

To compare the prediction models, three main evaluation metrics are used on test set: accuracy, F1-score, and AU-ROC. Accuracy is the ratio of the number of correctly predicted data points to the total number of data points. To understand F1-Score we first need to see what precision and recall are. Precision is an evaluation metric that shows when the model predicts positive, how often is it correct. Precision is calculated based on Equation 1. On the other hand, recall shows how many of the actual positives the model has been able to capture compared to the total actual positives. Recall is calculated based on Equation 2. F1-score is a combination of these two factors based on Equation 3, and shows how good the model is in both precision and recall.

[2]Receiver Operating Characteristic curve which is the true positive rate versus false positive rate

$$Precision = \frac{True\ Positive}{True\ Positive + Flase\ Positive} \tag{1}$$

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{2}$$

$$F1-score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3}$$

AU-ROC or area under ROC curve is another evaluation metric for classification models which shows how well the model is able to separate the two classes. This metric is the area under the true positive rate (TPR) versus false positive rate (FPR) plot. TPR is the recall, and FPR shows how often the model gives a false alarm, and it is calculated based on Equation 4 .

$$Recall = \frac{False\ Positive}{False\ Positive + True\ Negative} \tag{4}$$

In all of the mentioned evaluation metrics, the closer the number is to one, the better the model is predicting. Table III shows a comparison between the models used in this project. Explanation for each of the models can be found bellow.

1) Random Forest Classifier: To decide on which learning parameters to use for random forest prediction, we have manually tested a few variations and finally decided on the learning parameters shown in Table II. N_estimators is the number of trees in the decision tree which is one of the parameters that show how complex the model should be. Max_depth is the maximum tree depth that the model should go. Max_features is the number of features the model considers when looking for a best split. We have selected this value to the square root of the total number of features so that the model fits faster. If class_weight is set to *balanced*, the model automatically sets the weights associated with each class. This feature can be used when the dataset is skewed and does not have the same number of rows belonging to each class. In the case of this project, the dataset is very skewed and contains more data points in the class *passed*.

TABLE II
RANDOM FOREST LEARNING PARAMETERS

n_estimators	1000
max_depth	30
max_features	'sqrt'
class_weight	'balanced'

Random forest classifier fitting process is heavy and time-consuming specially for large datasets. To address this issue, we increase the number of data points used in the fitting gradually instead of using the whole dataset. In each step, we consider the accuracy of the model, F1 score and area under ROC curve on the test set. We run random forest with 1,000, 10,000, 100,000, and 1,000,000 data points randomly sampled from the whole 2.8 million data points. The results are show in Table III. It can be seen that with 1 million data points, random forest reaches an accuracy of 0.95 percent as well as an F1-score of 0.96 and AU-ROC of 0.93. Thus, there is no need to run the random forest model with larger datasets.

TABLE III
PREDICTION MODELS' EVALUATION METRICS

Model	Accuracy	F1	AU-ROC	Time (S)
Stratified	0.58	0.7	0.5	2.83
Logistic Regression	0.69	0.82	0.5	6.01
Random Forest 1k	0.68	0.8	0.51	2.23
Random Forest 10k	0.73	0.83	0.59	14.16
Random Forest 100k	0.82	0.88	0.74	165.63
Random Forest 1M	0.95	0.96	0.93	2674.7

Now that we have the random forest prediction model, we can look for the most important features in our model. The random forest model in scikit-learn library has an attribute called feature_importances_. When comparing the importance values, features with higher values are more important. However, there is no threshold given for this metric to say features with an importance value of less than that threshold are not important. As a result, there is no way to distinguish between the important and unimportant features. To address this issue, as mentioned in section V-A6, we have added a new random feature. Features with an importance value of more than random are categorized as important and the ones with an importance value of less than random are categorized as unimportant. Figure 3 shows the important and unimportant features based on random forest predictor. Six features are reported important in this model. Although the git_diff_test_churn feature has a lower importance value than random, since it is very close to random's value, we can say that this feature is also important. Therefore, we have extracted seven features as important from random forest classifier.

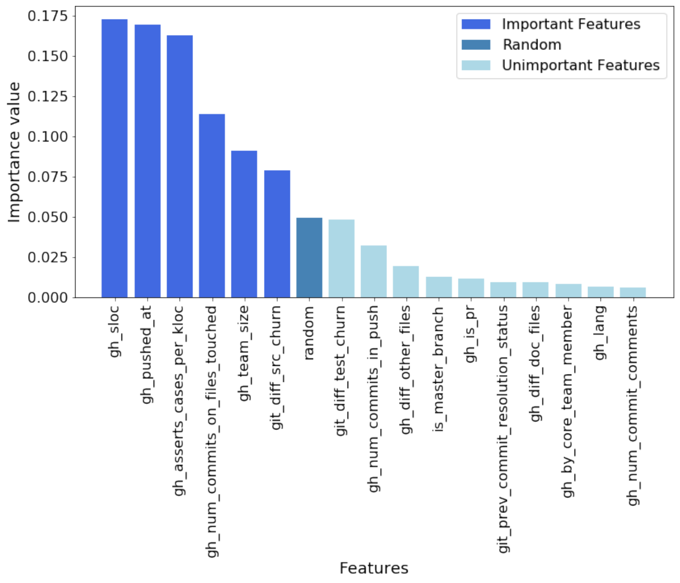

Fig. 3. Importance value of the features based on random forest

2) *Logistic Regression Classifier:* A logistic regression model with default learning parameters in Python scikit-learn library is used for evaluating the prediction. Table III shows the evaluation metrics of this model. The code is run on a computer with 12GB RAM, and Inrel(R) Core(TM) i7-6700HQ CPU with 2.6 GHz speed. It can be seen from the results that in terms of prediction, logistic regression is not as good as random forest. However, it might be able to report important features that can be easily be interpreted to conclusions for developers.

To use logistic regression for extracting the important features, we need to find the P-values and coefficients (weights). Since extracting these values can easily be done in R, we have used R with the complete dataset (2.8 million rows). To be able to compare the weights of the features, we have first standard scaled each column of data so that it has mean and variance equal to zero and one respectively. The P-values and coefficients of logistic regression are reported in Table IV.

As can be seen in the table, the P-values reported are almost unusable. Most of the features have a P-value of zero which is due to the high number of data points in the dataset [13]. However, we can still find some information based on the weights of the features. Since we have standard scaled the data, these weights are comparable.

3) *Bootstrapping:* In the prediction models part, we were able to identify the important features for each of the prediction models. However, to conclude how developers should

TABLE IV
LOGISTIC REGRESSION VALUES

Feature	**P-Value**	**Coefficients**
gh_lang	0	-0.128
gh_sloc	0	-0.191
gh_is_pr	0	0.075
gh_pushed_at	0	0.155
gh_num_commits_in_push	0	-0.062
gh_num_commit_comments	0	-0.04
gh_team_size	0.4	0.001
git_diff_src_churn	0	-0.06
git_diff_test_churn	0.25	0.003
gh_diff_doc_files	0.0003	-0.008
gh_diff_other_files	0.00013	-0.0059
gh_asserts_cases_per_kloc	0	-0.033
gh_by_core_team_member	0	0.08
gh_num_commits_on_files_touched	0	0.099
git_prev_commit_resolution_status	0	0.19
is_master_branch	0	0.052
random	0.569	0.0007

behave to minimize the chances of build failure, we need more concrete relationships. In this part, we use the bootstrapping technique to find, verify and visualize these relationships. In statistics, any test that relies on random sampling with replacement is called bootstrapping. We will describe the technique we have used with an example: we would like to see how gh_is_pr influences the build status. We follow the steps bellow:

- **Step 1:** Divide the dataset into two datasets, one containing all data points that are pull request, and another containing all data points that are not pull request.
- **Step 2:** Randomly select 100,000 rows of data from the two new datasets.
- **Step 3:** Calculate the failure rate in each of these sampled data. (After this step, we would have the failure rate for pull request data points and not pull request data points.)
- **Step 4:** Calculate and save the difference between the two failure rates.

$$PR\,Failure\,Rate - Not\,PR\,Failure\,Rate$$

- **Step 5:** Repeat Step 1 to 5 for 100 times.
- **Step 6:** Plot the histogram of the saved differences. Figure 4 shows an example of such a histogram.

If all the differences are negative, we can say that not PR failure rate is more than PR failure rate meaning there is a higher chance for the not PR builds to fail compared to PR ones. On the other hand, if all the differences are positive, we can say that PR failure rate is more than not PR failure rate. If there are both positive and negative values in the plot, we are not able to draw any conclusions.

The example above is showing how we use this method for binary features. For numeric features, we should slightly change the method. We use the same concept but with a threshold. Take git_diff_src_churn for example; we would like to see how this feature is influencing the build status. We would follow the steps bellow:

- **Step 1:** Choose a threshold. (e.g., One)

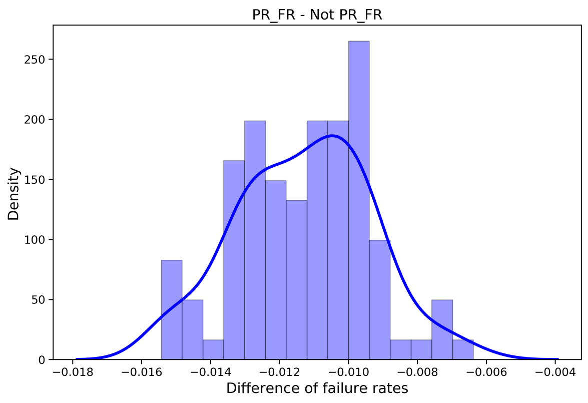

Fig. 4. Bootstrapping on gh_is_pr feature

- **Step 2:** Divide the dataset into two datasets, one containing all data points that have a git_diff_src_churn of less than the threshold, and another containing all data points with a git_diff_src_churn of more than the threshold.
- **Step 3:** Follow steps 2 to 6 from the previous method. Except, instead of plotting the results, check whether all values are negative or positive. If all values are negative, store -1 as the result of this threshold, if all of them are positive, store 1 as the result, and if there are both positive and negative values, store zero.
- **Step 4:** Repeat steps 1 to 4 for all the thresholds needed.
- **Step 5:** Plot the results stored for all thresholds. Figure 5 shows an example of such a plot.

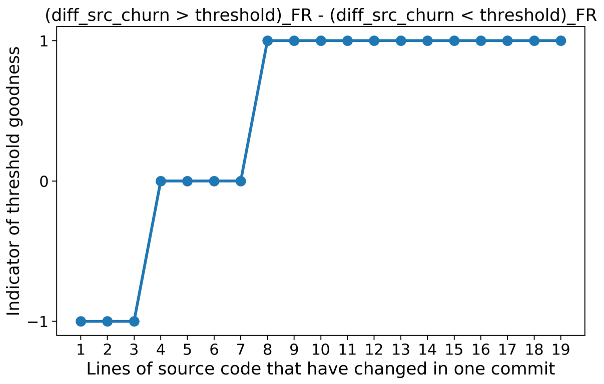

Fig. 5. Bootstrapping on diff_src_churn feature

The plot would show us which threshold is better. In Figure 5, it can be seen that failure rate for $git\_diff\_src\_churn < 3$ is more than $git\_diff\_src\_churn > 3$ meaning it is better to have commits that in which we at least change 3 lines of source code. In addition, failure rate for $git\_diff\_src\_churn > 8$ is more than $git\_diff\_src\_churn < 8$. Based on these two observations, we can conclude that if we keep the lines of source code changes in each commit between 3 and 8, there is a lower chance for the build to fail.

VI. Results

In this section, we use all the different methods used in the project to answer the five research questions proposed in the introduction.

A. *RQ1: What are the factors that influence Travis CI build status in general?*

Based on the random forest model, the most important factors are shown in Figure 3 which are 6 features. Moreover, based on logistic regression, all values in Table IV with P-value zero are important features which are 12 features.

RQ1 Summary: The factors that influence the build status based on random forest model and logistic regression model can be found in Figure 3, and Table IV respectively.

B. *RQ2: Does the average time between commits have a relationship with build failures?*

In both logistic regression and random forest prediction models, the time of the push that has triggered the build is reported as a very important feature. The logistic regression weight suggests that commits that have been pushed later in time are less probable to fail. However, this value is not representing the time between commits. To find whether this feature is influencing the build status, we calculate the average time between commits for each project in the dataset. One problem that we faced in this part was some missing commits in the dataset. Since we were not able to identify these missing commits, we assume there is not missing commits and sort all the build jobs based on their timestamp. Then we calculate the difference between each commit time and its previous one and find the average of these differences. Therefore, we have 1,272 rows of data showing the average time between commits for each project and the failure rate of the project. We then use bootstrapping on this information by doing sampling with replacement with the size of 1272 for 100 times which is shown in Figure 6. However, since the number of datapoints is small, each time we run the bootstrapping code it generates a slightly different graph. This plot is not showing any obvious relationship between the development time and failure rate.

RQ2 Summary: No, there is no easily interpretable relationship between time between commits and build status.

C. *RQ3: Does the number of lines of code that are changed influence the build status?*

The number of lines of codes that are changed in each commit is represented by git_diff_src_churn in the dataset. As we saw earlier, random forest reports this feature as important and logistic regression has a weight of -0.06 for this feature. This weight suggests that an increase in the git_diff_src_churn feature results in a higher failures probability. We have also seen in Figure 5 that this feature is, in fact, influencing the failure rate. The results suggest that commits that change between 3 to 8 lines of source code have a lower build failure probability.

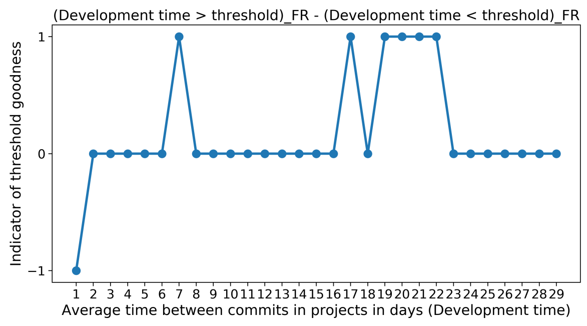

Fig. 6. Bootstrapping on development time feature

RQ3 Summary: Yes, all the three methods suggest a relationship between this feature and build status. Commits that change between 3 to 8 lines of source code have a lower build failure probability.

D. *RQ4: Is the size of a team related to build failures?*

Team size is a really interesting feature. In the random forest model, it has been reported as one of the most important features. On the other hand, in logistic regression model, not only it has a very low weight, it also has a very large P-value (0.4) suggesting that this feature is not at all important in the build status. Since our complex model shows a relationship and our simple model does not show a relationship, we expect this feature to have a complex influence on failure rate. The bootstrapping results are shown in Figure 7. If we were to look at teams with less than 8 members (small teams), the ones with size 5 to 7 have the lowest failure probability. On the other hand, if we were to look at teams with size more than 8 (large teams), the ones with size 9 to 12 have the lowest failure probability. We can strongly say that teams larger than 20 have a higher build failure probability.

RQ4 Summary: Yes, although complex, there is a relationship between this feature and build status.

E. *RQ5: How are each of these factor influencing build failure?*

We have already answered this question for RQ2, RQ3, and RQ5. In this part, we will use bootstrapping on some other interesting features. Figure 8 shows the results on gh_by_core_team_member feature. These results suggest that when commits are pushed by a core team member, there is a lower chance the builds fail. The logistic regression weight also verifies these results as it is positive.

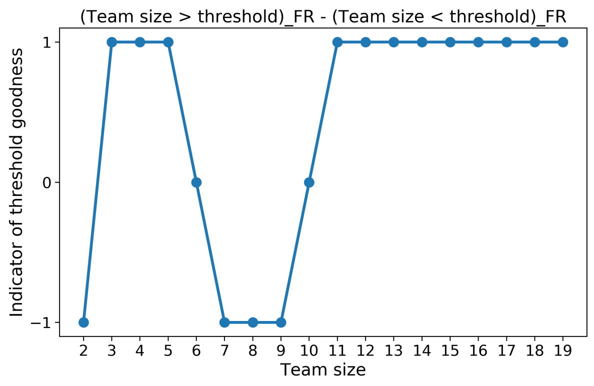

Fig. 7. Bootstrapping on team size feature

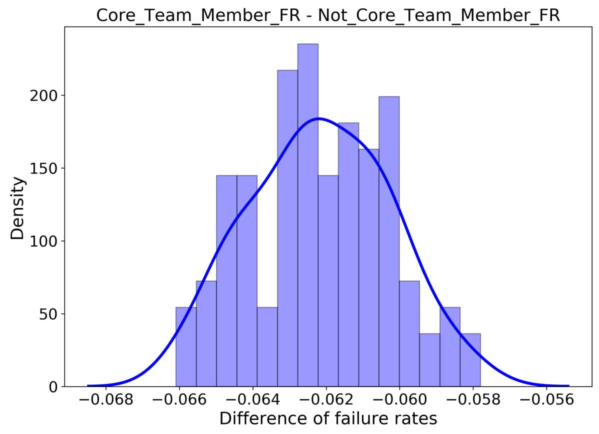

Fig. 8. Bootstrapping on core team member feature

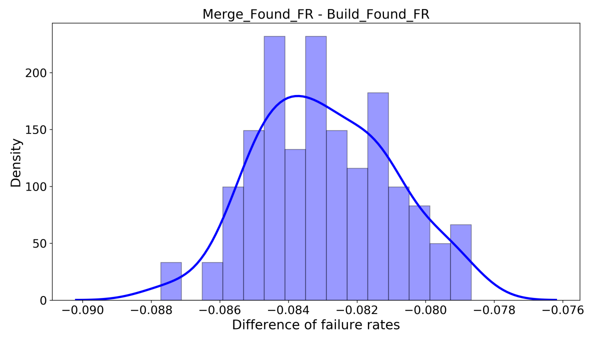

Fig. 9. Bootstrapping on previous commit resolution status feature

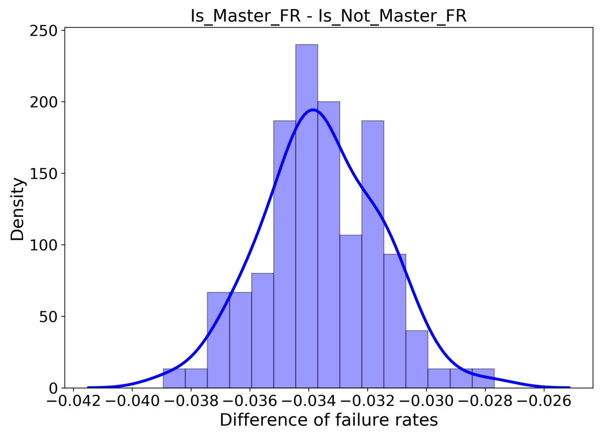

Fig. 10. Bootstrapping on is_master_branch feature

The next feature that we investigated is git_prev_commit_resolution_status. In our final dataset, this feature is either build_found or merge_found meaning this commit was either from another branch or not. Figure 9 shows the results for this feature. It can be seen that commits that are merge have a lower failure probability than those from the same branch. This is probably because commits from other branches have already been tested. Its logistic regression weight also confirms this result as it is positive and the weight is relatively high which means it is an important factor. This interesting finding suggests that developers should try to work on non-master branches and only commit to the master branch when they have tested the results.

The result for is_master_branch feature in Figure 10 shows us that developers are actually following the aforementioned rules. The commits on master branch have lower failure rate than the ones not on master branch.

Feature gh_num_commits_on_files_touched shows us how active this part of the project is. If the number of commits in this part of the project is high, this part is regularly changed. On the other hand, if this number is low, this part of the project is hardly ever changed. Figure 11 shows the bootstrapping results for this feature. These results suggest that when developers commit on files that have been in 20 to 50 commits, there is a lower failure probability compared to other files. Although this result is interesting, it does not give us any useful insight.

RQ5 Summary: Being a core team member and using branches in development lower the chance of build failure

VII. Threats to Validity

Although we have tried our best to minimize the errors in this project, there are still some factors that may influence the results. We have not considered the fact that some of the projects in the dataset may be Test Driven Development (TDD) based. In these projects, failure in the build might not mean a problem in the development process. In contrast, the goal is first to have a failed test and then try to make the tests pass. One possible approach for identifying these projects in the dataset is to look at the number of lines of production code, and test code changed in the consecutive commits and compare these two values to find a TDD trend in the project. However,

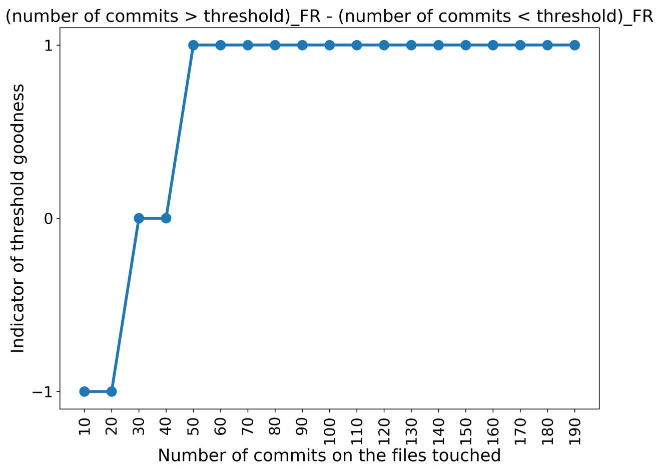

Fig. 11. Bootstrapping on gh_num_commits_on_files_touched feature

the dataset does not contain all the consecutive commits of all projects. Therefore, we were not able to identify the TDD based projects.

Another important threat is the size of the dataset. This dataset is huge and as we saw in the previous sections, this could cause some problems in the course of the project. Moreover, the dataset only has information of about 1200 projects which is not enough to draw project based conclusions.

Finally, this dataset has some outliers in almost all features. Finding and removing these outliers is very hard in such a huge dataset. We have not removed any of the outliers because we did not want to remove any information. However, this could itself cause some problems in the prediction models.

VIII. Conclusion and Future Work

In this project, the goal was to find reasons continuous integration builds fail and help developers grow habits that result in fewer build failures. In order to find these reasons, we used three different methods and combined the results: logistic regression prediction model, random forest prediction model, and bootstrapping. The following are the main findings of this paper:

- The average time between commits does not seem to have any interpretable relationship with build status.
- Commits that change between 3-8 lines of source code have a lower build failure rate than others.
- Team size influences the build status but with a complicated relationship.
- Using branches other than master and committing the tested code to master branch lowers the build failure rate.
- It is better that core team members commit the results to master branch as they have a lower build failure rate.

As future work, one suggestion is working on finding the TDD based projects and removing them from the dataset. We may be able to do so either by analyzing the dataset in more details or manually. Moreover, since this dataset is huge, it would be useful to optimize the code to use less memory and make the code faster to run. Finally, working on finding and removing the outliers in the dataset could improve the prediction model and bootstrapping results.

References

[1] 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.

[2] "travis-ci." ,https://travis-ci.org/, [Online; Accessed on 13 Apr 2019].

[3] R. Souza and B. Silva, "Sentiment analysis of travis ci builds," in *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pp. 459–462, IEEE, 2017.

[4] E. A. Santos and A. Hindle, "Judging a commit by its cover," *MSR, New York, NY, USA*, 2016.

[5] M. Rebouças, R. O. Santos, G. Pinto, and F. Castor, "How does contributors' involvement influence the build status of an open-source software project?," in *Proceedings of the 14th International Conference on Mining Software Repositories*, pp. 475–478, IEEE Press, 2017.

[6] R. Jain, S. K. Singh, and B. Mishra, "A brief study on build failures in continuous integration: Causation and effect," in *Progress in Advanced Computing and Intelligent Engineering*, pp. 17–27, Springer, 2019.

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

[8] Y. Luo, Y. Zhao, W. Ma, and L. Chen, "What are the factors impacting build breakage?," in *Web Information Systems and Applications Conference (WISA), 2017 14th*, pp. 139–142, IEEE, 2017.

[9] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 805–816, ACM, 2015.

[10] "Jenkins." ,https://jenkins.io/, [Online; Accessed on 13 Apr 2019].

[11] "Teamcity." ,https://www.jetbrains.com/teamcity/, [Online; Accessed on 13 Apr 2019].

[12] G. Gousios, "The ghtorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, (Piscataway, NJ, USA), pp. 233–236, IEEE Press, 2013.

[13] M. Lin, H. C. Lucas Jr, and G. Shmueli, "Research commentarytoo big to fail: large samples and the p-value problem," *Information Systems Research*, vol. 24, no. 4, pp. 906–917, 2013.