Towards Language-independent Brown Build Detection

Doriane Olewicki
Polytechnique Montréal
Montréal, Canada
doriane.olewicki@polymtl.ca

Mathieu Nayrolles
Ubisoft Montréal
Montréal, Canada
mathieu.nayrolles@ubisoft.com

Bram Adams
Queen's University
Kingston, Canada
bram.adams@queensu.ca

ABSTRACT

In principle, continuous integration (CI) practices allow modern software organizations to build and test their products after each code change to detect quality issues as soon as possible. In reality, issues with the build scripts (e.g., missing dependencies) and/or the presence of "flaky tests" lead to build failures that essentially are false positives, not indicative of actual quality problems of the source code. For our industrial partner, which is active in the video game industry, such "brown builds" not only require multidisciplinary teams to spend more effort interpreting or even re-running the build, leading to substantial redundant build activity, but also slows down the integration pipeline. Hence, this paper aims to prototype and evaluate approaches for early detection of brown build results based on textual similarity to build logs of prior brown builds. The approach is tested on 7 projects (6 closed-source from our industrial collaborators and 1 open-source, Graphviz). We find that our model manages to detect brown builds with a mean F1-score of 53% on the studied projects, which is three times more than the best baseline considered, and at least as good as human experts (but with less effort). Furthermore, we found that cross-project prediction can be used for a project's onboarding phase, that a training set of 30-weeks works best, and that our retraining heuristics keep the F1-score higher than the baseline, while retraining only every 4-5 weeks.

1 INTRODUCTION

Producing high-budget video games ("AAA games") takes a lot of effort and organization. Modern AAA games are composed of tens of millions of lines of code, scattered across hundreds of thousands of files and tens of thousands of code changes created by hundreds of developers. Furthermore, modern AAA games' developers need to manage additional complexities due to the multidisciplinary teams (i.e., artists, devs, physics experts, data scientists), very different from code-only projects, and to be compatible and scale across a multitude of platforms (various consoles, PC and mobile devices).

The combination of online imperatives and the multiplicity of platforms has made reliable Continuous Integration (CI) pipelines paramount to producing AAA games with a limited number of bugs, if not bug-free. Whenever a code change is submitted for review, it is forwarded to the CI pipeline, which automates compilation, testing and other required activities (e.g., static analysis) [22]. If all the steps are successful, the CI status is *green* and the change is integrated into the project. Otherwise, it turns *red* and the developer needs to debug/fix the code change based on the CI's build log. [1]

Unfortunately, the CI build results are not always a reliable indication of a code change's quality, since builds can fail because of factors related to the build process [16, 17, 38] (e.g., missing dependencies or network/disk access on the build machines), or because of flaky tests [19, 30, 33] (i.e., tests with non-deterministic outcomes).

We define brown builds as a build failure that changes to a success on at least one build rerun without changing the build setup or source code. For instance, a build could fail if the communication between the CI pipeline and physical game consoles is interrupted. A test could be failing on an under-provisioned/overused CI job worker (due to which operations are executed in a different order than the test expects), but pass otherwise. Simply rerunning such a "brown build" (on the same code change) could make the build failure disappear.

In our experience at one of the world-leading AAA game producers, we saw that brown builds hinder the confidence of developers in their CI, and impact the productivity of developers and testers alike. Instead of immediately investigating the source code upon a build failure, the potential presence of brown builds tends to push developers to manually trigger reruns of builds, just to be sure. On six large industrial projects analyzed in this paper, 31% (3.5k/11.4k) of commits had at least one manually rerun build job, and 15% (27k/179k) of build jobs were rerun at least once. This is not only a waste of hardware resources for CI, but also of developers' productivity since they have to wait for the reruns to finish, while the CI pipeline is blocked. While the rerun could still cost less time than manually checking the source code in vain, it adds to an already congested CI pipeline and delays even further the manual testing process of games, since testers wait for a green build.

Hence, there is a strong need for pragmatic approaches that can distinguish real build failures from brown builds. At a minimum, such approaches provide a second opinion that could confirm developers' suspicions about a build failure, restoring their trust in CI results. One would also expect the approaches to be integrated into the CI pipeline, for example to automatically re-run a subset of the brown builds, or to perform other automated resolution techniques,

---

[1]In recent years, the term "CI" has started to refer to these pipelines instead of to the original, agile practice. The rest of this paper will do the same.

making better use of CI resources. While approaches have been proposed for flaky test detection, based on code dependencies [11, 25], dynamic code analysis [18], or test-smell [10], flaky tests are only part of the problem, since brown builds can be due to the build process itself, as mentioned before. Furthermore, industrial software projects feature a variety of programming languages and tools, making adoption in practice of existing flaky test models hard. Finally, brown builds are an issue for both young and old projects, yet only the latter have sufficient historical build information to build models (the so-called "cold-start" problem [26]).

To address these shortcomings, this paper presents a language-independent approach to identify brown builds that leverages the build logs produced within the CI. We extract and filter vocabulary from the build logs, transform the resulting words into a vector-based representation using TF-IDF [36], then train boosted tree-based classifiers [8] to predict if a job is brown or not. We empirically evaluate the classifiers on six industrial projects of a leading AAA-games developer and one open-source project.

This paper addresses the following research questions:

RQ1. *Can we accurately detect brown builds in a language/ project-agnostic way ?* Our best models got an F1-score of 84% (precision of 76%, recall of 94%), with F1-score on par with experts' prediction (-4% to +17%), suggesting that our models are pragmatic.

RQ2. *Can a brown build prediction model be used on another project ?* A model trained on a project can be used for a new project's prediction during its onboarding phase, but a project-specific model should be used as soon as onboarding is over.

RQ3. *How long can a model stay relevant without being retrained ?* We found that we can schedule the retraining of the model every 4-5 weeks depending on its performance evolution and age. Data older than 30 weeks does not significantly improve the F1-score (< 0.01%), and even harms the model for some projects.

Our major contributions are a language-independent brown build detection approach with F1-score two to three times higher than baseline models for 7 large projects (and on par with human experts), as well as the empirical evaluation of heuristics to counter the impact of concept drift over time. A replication package is available online [3].

2 BACKGROUND & RELATED WORK

A Continuous Integration (CI) server [22] automatically rebuilds and retests the source code of a project whenever a developer pushes a code change to their version control system, in order to detect faults and merge conflicts as soon as possible. A typical CI pipeline like Jenkins or TravisCI consists of a sequence of build stages (e.g., "compile" followed by "test"), each of which are composed of one or more parallel build jobs (e.g., "compile" jobs on Linux and Windows). The behavior of such a build job is specified via build scripts in a domain-specific language like GNU Make, Maven or Gradle. Such scripts typically transform source code into an executable program by invoking configuration tools, preprocessors, and compilers, they automate the execution of test harnesses and/or can even deploy the produced build artifacts [9, 22].

While conceptually, a CI server is thought of as performing one build for each new code change, in practice its role is much more complex. First, the build dashboards of large open source organizations like Mozilla's treeherder [7] or OpenStack's zuul [2] show a multi-dimensional matrix that tries to summarize results for dozens of CI pipelines and build jobs, ranging from classic compilation and unit test execution to deployment or even static analysis. Furthermore, each such pipeline is run multiple times for a given code change, since, in each build stage, multiple jobs should be run to cover the major feature and environment configurations the code base is expected to run on. If a project has 10 features and should support 5 operating systems, ideally 10 x 5 build jobs should be scheduled in each build stage. Since a typical project has a much larger number of features, and its environment comprises of not only different operating systems (versions), but also different devices, processor models, library dependencies, supporting databases and web servers, each code change potentially yields a combinatorial explosion of build jobs to run. Of course, if a build is deemed to fail, all builds would need to be repeated for the proposed code fix.

While this "build inflation" phenomenon [38] increases confidence in build results, it brings a number of major disadvantages as well. First, it increases the build infrastructure (and energy) cost for organizations. Google, for instance, performs 800k builds per day, which schedule 150M test runs [32]. Google's breakneck code velocity of one commit per second coupled with its linearly increasing test corpus implies a quadratically growing need in build resources [32]. Similar to many other organizations, including OpenStack, they have moved to CI scheduling algorithms that group multiple code changes (e.g., all changes arriving within 45 minutes) before starting a new build on the entire group, instead of executing separate builds for each code change. While successful builds at the group-level leave out many builds at the individual level, failures at the group-level do require additional follow-up builds to determine the individual code changes responsible for the failures. Furthermore, while interpretation of build failures is the biggest challenge of CI users [20], the large number of builds generated by build inflation makes build failures harder to interpret. For example, Gallaba et al.'s analysis of 3.7 million GitHub build jobs [16] found that 12% of passing CI builds contain failing or skipped build jobs that the CI system was asked to ignore by developers, with 2 out of 3 breakages occuring more than ones. Furthermore, 44% of the studied build failures were environment-dependent, i.e., only occurred for some environments.

Even worse than the presence of noise and build inflation due to different environments is the ambiguity of build results caused by so-called brown build jobs. These are build jobs that fail inconsistently due to issues with asynchronous calls, multithreading, or test order dependencies [16, 17, 30]. Only by repeating such build jobs a sufficiently large number of times, we could determine for sure whether a build job really failed or succeeded. In the meantime, the code contribution pipeline conservatively would be locked, basically preventing other teams to merge in their contributions. If one could predict that the build is truly brown, i.e., not a real build failure, the pipeline could remain open, and one could also avoid propation of brown build results. This is because, in a typical organization, different software components are reused across different libraries and products, and the (apparent) success of a new build is considered as a "go" signal for dependent products to adopt the new release of the component, potentially inheriting brown-ness.

While certain build tools like Maven have support for rudimentary flaky test detection through build repetition, this is not efficient, nor accurate. For this reason, existing work in this area [10, 11, 24, 30, 35] has focused on empirically understanding the causes of flaky tests, as well as ways to detect such tests for specific programming languages. Other reasons of brown builds than flaky tests are not considered, nor approaches that are independent of programming language. Ironically, there are often resources available to optimize and fix the build environment or other resources code responsible for flaky builds, yet those need to be briefed with concrete starting points, which currently are unknown. A recent language-independent approach, proposed by Lampel et al. [27], will be discussed in Section 4.2.

3 LANGUAGE-INDEPENDENT BROWN BUILD DETECTION APPROACH

This section will present each step of our methodology for language-independent brown build detection, as well as the research process according to which the approach was designed.

3.1 Vocabulary extraction from log files

We first extract the vocabulary from the log file produced by each build job. To reduce the dimensionality of this vocabulary, we applied the following series of rules:

Rule 1 All URL and file paths, identified by a regular expression, are replaced by a known string.

Rule 2 Commit IDs (series of characters containing at least one letter and one number) are replaced by a known string.

Rule 3 Any non-letter characters are removed.

Rule 4 Camelcase notations are split.

Rule 5 English stop words [1] are removed.

Rule 6 A stemming algorithm [6] extracts the root of words.

On average, these rules reduced the size of the log files by 50% on the dataset studied. We then split the text into words and perform n-gram extraction, since using n-grams (sequences of n words) convey more meaning [13]. In this study, we applied n-grams with $N = 1$ and $N = 2$ (see section 4.2.1 for the hyper-parameter tweaking). We discarded our experiments with $N = 3$ and $N = 4$, because they were computationally too expensive and did not significantly improve the results. As output of this step, each build job is represented as a dictionary of features (words or series of words) to the number of their occurrences in the analyzed log file. The resulting dictionaries vary in size depending on the size and variety of the vocabulary in each file. We also keep track of metadata surrounding each CI build job such as the date on which it was submitted, the job ID, the commit ID and the number of retries.

3.2 Vectorization

In this step, we create a uniform representation of the data, where each build job is represented by a vector of relevant features. The features consist of the TF-IDF computation of textual build job data (words and series of words), and other build-related metrics.

3.2.1 TF-IDF computation. We used TF-IDF to represent each textual feature in order to reduce the impact of large log files on the vocabulary word counts [36].
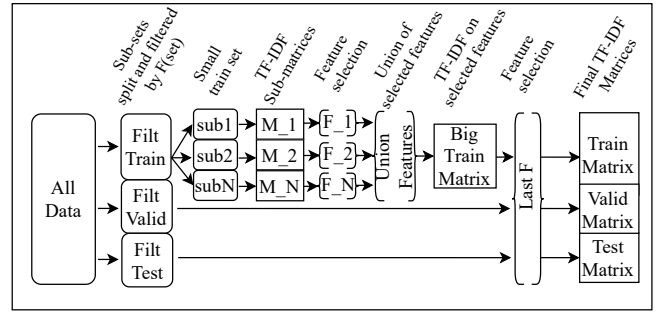

Figure 1: Iterative vectorization approaches.

3.2.2 KBest feature selection. Despite the filtering applied on the log, the number of unique (series of) words used in the vector representation of our data is still significantly larger than the number of observations (CI build log files). Consequently, we used feature selection to further reduce the number of features used. For this, we used the SelectKBest feature selection from the sklearn package [5] in Python, with ANOVA F-value score function.

Other algorithms considered for feature selection (i.e., infogain [14], correlation computations [12]) had similar results as KBest in terms of selected features, but were computationally taxing.

3.2.3 Iterative computation. While KBest's time and memory complexities are acceptable for most cases, our dataset's dimensionality and size made it impractical to apply to the entire data set at once. Consequently, we perform SelectKBest on subsets of our dataset, then the union of the best features of each subset is analyzed again as summarized in the middle of Figure 1.

In particular, we split the training set into sub-matrices of 1,000 build jobs, then perform TF-IDF and KBest on each sub-matrix individually. Afterwards, the extracted sets of selected features F_i are united into one large matrix. The KBest feature selection algorithm is finally applied one last time on the resulting union set F_{uni}, yielding the final matrix with selected features F_{final}.

For the iterative approach to be acceptable, we need to validate that the size of the union-ed matrix F_{uni} is similar to the size of the individual F_i sets ($|F_{uni}| \simeq |F_i|$). We confirmed that in our case study a sub-matrix size of 1,000 led to the size of the F_{uni} set exceeding K by less than 10%, which we considered to be acceptable.

3.2.4 Other metrics. Apart from the build log vocabulary, we also considered a number of other features related to the life cycle of CI jobs and to the position of the build job in that cycle. First, we computed the number of prior reruns, fails, and successes for each build job. Also, we compute the number of commits since the last brown job (#commit_since_brown), to control for temporal information about when brownness was found previously.

3.3 Classification

As shown in Figure 1, the dataset is split into training/test/validation sets. Vectorization and feature selection is done on the training set, then applied to the two other sets, such that the model is not contaminated by either validation or test set. We also avoid set-contamination by gathering, for each commit, all its jobs in the same
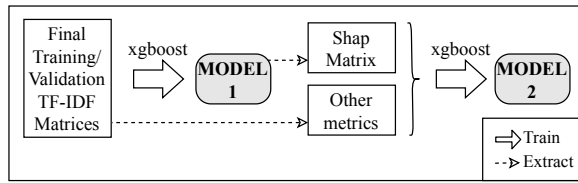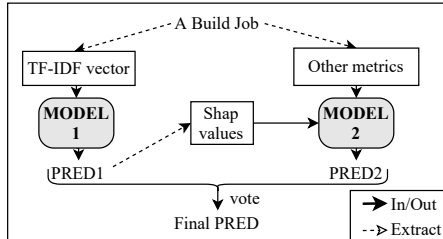
Figure 2: Two-step model training.

Figure 3: Prediction with two models. A vote is computed between both model's prediction.

set. The test set will be used for the model's performance estimation, while the validation set is used to optimize the classification model.

Our models classify failed jobs, since successful builds do not block the CI pipeline, and hence cannot be brown. However, successful jobs might still bring information about brownness, and thus helps the model identify brown features to be used on failed jobs' predictions. We defined a filter to be applied on the dataset, to identify from which set we filter out all the successful jobs. The filter application is shown in Figure 1, with the notation $F(set)$.

- *None*: no filter applied.
- *Train*: filter applied to training set;
- *All*: filter applied to training and validation set.

We use the XGBoost algorithm [8] (eXtreme Gradient Boosting) to predict the brownness of each build job, using a two-step training process (Figure 2). XGBoost is a directed classification algorithm based on random forest. Each prediction is a real value between 0 and 1 (where 1=brown and 0=safe). The model also generates so-called Shap values [4], which provide the impact of each feature on the training and validation sets' predictions.

We opted for a two-step training process, since the resulting composite model allows to first focus only on data from the job's build log, then to add the CI lifecycle-related information (Section 3.2.4). Such a two-step model has been used before[23]. The first step of our training process only considers the vocabulary-related features, while the second one considers the other metrics and combines those with the Shap values of the first model. Then, we pass both predictions through a vote in order to have a final classification of the build, before making the final classification based on a threshold β. The vote is the weighted sum of both predictions, as follows:

$$\text{PredF} = \frac{\alpha}{100}\text{Pred1} + \frac{100-\alpha}{100}\text{Pred2} \quad (1)$$

$$\text{Classification} = \begin{cases} Brown, & \text{if PredF} > \beta \\ Safe, & \text{otherwise} \end{cases} \quad (2)$$

Both hyper-parameters α and β were chosen experimentally during the validation phase and will be discussed in the section 5.1.

3.4 Research Process

The brown build detection approach presented in this section was developed using a design-science process [34]. In particular, we performed the following activities:

inferring objectives consultations with several teams to establish KPIs for model accuracy and concept drift;

design and development exploring build features, then iterating over different build log vectorization approaches on a pilot project (Project A of Section 4.1), eventually adding a second model to our approach (CI metrics);

demonstration of pilot to teams;

evaluation on projects B-F and OSS (Section 4.1) to validate generalizability.

4 CASE STUDY SETUP

4.1 Projects studied and data extraction

We gathered data from six large projects of our industrial partner and of one Open-Source project. These projects were different in language, purpose (i.e., code analysis, cross-platform computation, animation and path finding), and size, in order to reduce the threats to external validity of our approach. Table 1 shows the characteristics of the seven projects, with the closed-source projects' names elided for confidentiality reasons, and project OS being the open source project "Graphviz"[2]. We chose to work with Graphviz because it is a sizeable, multi-language open-source project with available (brown) build data.

We extracted data about build jobs from the 2 CI/CD platforms used by the 7 projects: Gitlab[3] and TeamCity[4], see Table 1. Using the REST APIs provided by both CI/CD platforms, we are able to extract the logs produced by each build job, which forms the input of our approach (see Section 3.1).

In order to obtain labeled data for our study, we leverage development guidelines adopted by the analyzed software projects regarding brown build jobs. In the absence of prediction models, the developers and other contributors of the six industry projects have to rerun a failed build job if it is suspected to be brown. This proportion is shown in the second column of Table 2. Out of the failed build jobs that were rerun, those that changed build outcome (without any change to the build setup or source code) are considered to be brown builds by our industrial partner, and hence by this paper (third column of Table 2). Other build jobs are labeled as true-result/safe.

The Brown Failure Ratio *BFR* is the percentage of brown job failures over the total number of job failures, including the reruns. Table 1 shows that the projects vary in their BFR among the failed jobs, from 58% (E) and 35% (A, B) down to 13% (OS), 10% (F) and 5% (C, D). While Graphviz does not have an explicit policy to rerun suspicious builds, we notice that its brown failure ratio (BFR) of 13% is close to the median brownness of the other analyzed projects (15%), which suggests its oracle is representative.

---

[2]Graphviz Gitlab link: https://gitlab.com/graphviz/graphviz
[3]Gitlab: https://about.gitlab.com/
[4]TeamCity: https://www.jetbrains.com/teamcity/

Table 1: Information on the projects studied.

Proj	Full project history				Scraped data					
	#Contri-butors	#commit	Main Lan-guages	Age [y]	DevOps Plat-form	#months	#jobs	#failing jobs	Brown failure ra-tio (BFR)	Mean job du-ration [mm:ss]
A	15	3k	Go, JS	3.5		18	23k	3008	30%	03:27±04
B	29	2k	C#	4	Gitlab	29	63k	8100	37%	03:14±14
C	64	7k	C++, C	3		17	22k	5986	5%	15:49±30
D	47	3k	C++,C#	1		3	88k	8711	6%	09:28±19
E	45	4k	C++	5	TeamCity	2	53k	1310	58%	03:07±07
F	47	3k	Py,JS,C#	5,5		22	9k	1705	10%	05:34±46
OS	20	14k	C,C++	17	Gitlab	43	47k	1237	13%	06:12±17

Table 2: Information on the projects studied regarding the brownness labeling.

Proj	#failed_rerun/failed	#brown/failed_rerun	#reruns (only for brown cases)				max #reruns
			0	1	2	3+	
A	34%	76%	1417	506 (371)	202 (163)	122 (94)	14
B	35%	74%	3360	702 (464)	268 (173)	802 (680)	29
C	8%	44%	5077	294 (126)	86 (36)	45 (25)	38
D	6%	84%	8032	497 (414)	48 (45)	7 (4)	9
E	65%	86%	313	79 (62)	191 (137)	318 (309)	24
F	18%	49%	1119	153 (78)	54 (21)	39 (21)	32
OS	13%	82%	984	104 (88)	26 (22)	16 (10)	12

4.2 Validation approach

4.2.1 Model building.

Hyper-parameter tweaking. In the methodology section 3, we identified a list of hyper-parameters to tweak. Those are gathered in the following list, with a summary of their purpose and the range of values that were chosen.

- $F(set)$: filters to apply on the sets.
  (Range: *None* (no filter), *Train* (only fails in the training set) and *All* (only fails in all sets))
- N: Number Ngram to consider.
  (Range: [1], [2], [1, 2])
- K: Number of features to be chosen by the feature selector.
  (Range: 100 to 300, by 25)
- α: Weight of the first model's prediction in the final predic-tion of Eq. (1).
  (Range: 0 to 100, by 10)
- β: Threshold for the classification, see Eq. (2).
  (Range: 10 to 90, by 10)

For all studied projects, we trained models with all hyper-parameter combinations. We used cross-validation to validate the results on the data set obtained in the previous subsection.

Cross-validation settings. To do the cross-validation, all build jobs were randomly given a group number from 0 to 9 in order to obtain 10 folds. The group separation respects the constraint that all builds related to a given commit ID are in the same data set group. Furthermore, the cross-validation is stratified, conserving the same proportion of brown jobs in each fold.

Then, for each iteration i, the following sets are defined:

- Train set: all folds but fold i (90% of dataset);

Table 3: Baselines (BFR is brown failure ratio per project).

Baseline name	Proba. brown pred	Brown			Safe	
		F1	Pre	Rec	Pre	Rec
Random50	50%	$\frac{BFR}{1+2BFR}$	BFR	$\frac{1}{2}$	1−BFR	$\frac{1}{2}$
RandomB	BFR%	$\frac{BFR}{2}$	BFR	BFR	1−BFR	1−BFR
AlwaysBrown	100%	$\frac{BFR}{1+BFR}$	BFR	1	NA	0

- Valid set: half of fold i (5% of dataset);
- Test set: the other half of fold i (5% of dataset).

Cross-validation is performed once for each hyper-parameter combination, with the train set used to vectorize (Section 3.2), the validation set used to optimize XGBoost's internal parameters on the trained models, and the test set to predict the model on an unseen data set. The validation/evaluation is done twice so that both halves of the subgroups are used once as a validation set, then test set, resulting in 20 models being trained per cross-validation.

Performance metrics. We use the commonly known precision ($\frac{TP}{TP+FP}$), recall ($\frac{TP}{TP+FP}$), F1-score ($\frac{2}{pre^{-1}+rec^{-1}}$) measures [15, 21, 28] to evaluate our models. We calculate precision and recall sepa-rately for the "brown" and "safe" labels. These metrics are used to compare our models to the baselines in Table 3, as well as to deter-mine the optimal configuration of hyper-parameters to use for our models. We computed local and global optimizations of the hyper-parameters. The local optimization $LocOp$ is the hyper-parameter combination c that optimizes the F1-score of the prediction for each given project p. As such, the optimal local hyper-parameter combi-nation may be different for each project ($LocOp_p = \arg\max_c F1(c)$).

In contrast, the global optimization is the hyper-parameter combination c that minimizes the sum of the squared differences between the F1-score of a hyper-parameter combination and the local optima across all projects ($GlobOp = \arg\min_c \sum_{p \in proj}(LocOp_p - F1(c))^2$).

Baselines. The baselines we use are based on random prediction models whose theoretical performance is calculated based on a given percentage of predictions being brown or real failures, as shown in Table 3. Since the projects we studied are multi-language in nature, it was difficult to apply existing, language-dependent approaches as baselines. For example, DeFlaker [11] covers flaky test detection in Java. Furthermore, Pinto et al. [35] predict flaky test cases based on the tests' tokens, yet brown builds do not necessarily relate to test cases (or even code).

Another approach for brown build detection, language-independent as well, was proposed by Lampel et al. [27], who leverage additional resource metrics related to execution time and CPU usage. The choice for these metrics is based on observations at Mozilla that brown builds would typically take longer to finish than real build failures. While promising, most of the metrics required by this approach were unavailable from our industry partner's CI. This is simply due to the CI system being deployed in a cloud, which makes accurate readings of execution time and CPU usage non-trivial because of (1) multi-tenancy and (2) unknown changes to the cloud's underlying hardware. As such, a project might have a low BFR running on slow hardware, then start to exhibit a large BFR on better hardware.

To validate this hypothesis, we scraped our 7 projects' build duration data (the only resource metric tracked by our industry partner) for build failures from 2021. We only included build job configurations with at least one brown build, then performed a Mann-Whitney test between the build duration distributions of (non-)brown failures ($\alpha = 0.01$). Results are available online in our replication package [3].

We found that, at least on the studied projects, multi-tenancy and evolving infrastructure impact the applicability of resource metrics. In particular, only projects A and F showed a significant difference (small Cliff's Delta effect size), confirming Lampel et al.'s hypothesis. For those projects, we then split the data chronologically into 5 groups, comparing the build duration of brown and real failures within each split. For project A, 4 out of 5 groups show a longer build duration for brown builds (2x large effect, 2x small). For project F, build duration differences alternate over time between (non-)significance (2x small).

To conclude, in the context of language-independent brown build prediction on the studied projects, we were only able to use random prediction models as baselines.

4.2.2 Manual validation. We also compare our model to experts' prediction and decision time. To do so, we asked experts of two projects to answer a survey related to their project. These experts comprise developers of the projects who are knowledgeable of what the code changes are doing and have been exposed to CI feedback. Per project, the experts were split into two groups (project A has one expert per group, project B two experts per group).

Each survey contained 40 build jobs, selected randomly among the dataset using the following constraints: (1) at most one build job per commitID was selected and (2) TP/TN/FP/FN jobs from our globally optimal model's prediction were equally represented ($\frac{1}{4}$ of the set of jobs in the survey for each category). While each group of experts received the 40 build jobs to evaluate, half of the jobs were provided with our model's prediction (either correct or incorrect) and the other half without. The jobs coming with predictions for Group 1 did not come with predictions for Group 2, and vice versa.

Each group was asked to evaluate for each given build job if they would label it as brown, based on the associated commit (ID, name, diff), build log, other metrics like #rerun and #commit_since_brown (see Section 3.2.4), and (for half of the jobs) our model's prediction.

4.2.3 Cross-project validation. Cross-project predictions means training the model using a training and validation set of a given project, then using it to predict the results on another project. If the results are satisfying, during the early stages of a new project (onboarding phase), models built on other projects could be used instead of having to wait until enough builds would have been run for the new project. In our evaluations, we apply each project's models on the other projects to evaluate cross-project performance.

4.2.4 Concept drift. Our concept drift validation aims to evaluate (1) how long the training set data and the trained model should stay up, (2) when the model should be retrained, and (3) with which part of the data. This is important to keep the performance competitive, since new cases of brown builds might be missed, while old types of brown builds might never reappear once corrected, making models obsolete at some point.

Intuitively, we might think that the more data we gather, the better results we will get. However, training on a large amount of data can be time- and resource-consuming. Furthermore, old data could be outdated, with given data patterns never reappearing in the newest build jobs' traces (e.g., when fixing a flaky test or the build machine).

Furthermore, we evaluate the question of when to retrain a model. On the one hand, predicting after each new build job is computationally expensive and the benefit of adding a single new build job to the training set is relatively small. On the other hand, using the same model forever disregards any new data. As such, new types of brown builds might never be identified by the model.

First, we want to evaluate how our approach is impacted by concept drift. Second, we propose a number of switching heuristics, to decide when to retrain the model. For this validation, we used the hyper-parameters that get the best global optimization.

Regarding the impact of concept drift on our approach, we needed to split the data sets into sub-data sets per period of time, which will be referred to as groups: we chose to split the data into weekly groups (Sunday to Monday). We then need to choose a training window size, which will be the number of consecutive groups that are used as the training data set. The window is then shifted across the whole data set to simulate a model being retrained each week: each retraining is referred to as a drifting step (see Figure 4). The three sets needed for our model computation and validation (training, validation and test sets) are computed respecting time isolation (**Iso-H**), by selecting consecutive ordered groups:
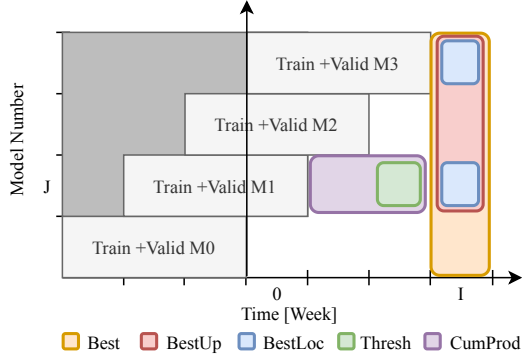
Figure 4: Switching heuristics.

Iso-H time isolation: The data-set groups are ordered such that if Group i comes before Group j, all the jobs in Group i precede all the jobs in Group j.

The sets are then defined for each drifting step i:

- Train set: {build ∈ group j | $j ∈ [i, i + win_{train} − 1]$}
- Valid set: {build ∈ group j | $j = i + win_{train}$}
- Test set: {build ∈ group j | $j > i + win_{train}$}

By varying the size of the training window, we can evaluate how far in the past data must be retrieved to achieve relevant prediction performance.

To evaluate how the model ages across time, we evaluate the performance on the test sets for each drifting step: we can then observe the concept drift impact on the Brown-Detector approach by observing the performance of each drifting step by group (week), and by the number of weeks since the model was trained.

We also observe the drift of features to analyze whether there is an evolution in the set of selected features in the vectorization step. For this, we measure the weekly feature surviving ratio, where model i's feature surviving ratio at week j is the percentage of features selected at week i that still are selected in the model of week j. If the surviving ratio decreases consistently over time for all models, then the feature selection and the model are impacted by the concept drift and a retraining would be relevant. We show the median feature surviving ratio of models after x weeks of existence.

4.2.5 Model switching heuristics. Based on the evolution of the performance of each model over time, we evaluate when to change from one model ot another in order to obtain the best performance. For this, we define different switching heuristics that will be used to decide when to retrain and switch models. Those algorithms will either use *a priori* or *a posteriori* heuristics. *A priori* heuristics choose to switch to the model of week$_i$ based on information about all weeks$_j$ before i ($j < i$). *A posteriori* heuristics choose to switch to the model of week$_i$ based on information about all weeks$_j$ before i and week$_i$ included ($j ≤ i$). The latter are unrealistic models since they need information about the current week before it even happened, but can be used as baseline for the a priori heuristics.

For the definition of the model switching heuristics, let us assume that the drift parameters are $win_{train} + win_{valid} = W$ (with $win_{valid} = 1$ the window size of the validation set) and that we have $W + N$ weeks of data, and that for each week w_i with $i ≥ 0$ there

is a model m_i created with the data from weeks $[w_{i−W−1}; w_{i−1}]$, with $i ∈ [0, N]$. We also define the model m_{s_i} as the model selected by a switching heuristic at w_i. The performance of model m_j on week w_i is $perf(i, j)$. Figure 4 shows an example with $W = 3$.

For the following switch heuristics description, we suppose that we are currently starting week w_i and that the previous chosen model $m_{s_{i−1}}$ is m_j.

Best: a posteriori algorithm where the model at w_i is chosen to be the model m_k with $0 ≤ k ≤ i$ with the highest performance $perf(i, k)$, as shown in orange on Figure 4.

BestUp: same as best, but with $s_{i−1} ≤ k ≤ i$ (red on Figure 4).

BestLoc: same as best, but with $k = s_j$ or $k = i$ (blue on Figure 4).

Diagonal: a priori algorithm where the model at w_i is m_i, without looking at other models' performance (switch every week to the newest model).

Fix: a priori algorithm where the model is switched every fixed number of weeks to the most recent model. If week w_i needs to switch (because the model has been used for the fixed number of weeks), the chosen model is m_i. Otherwise, keep m_{s_j}, without looking at the performances.

Thresh: a priori algorithm where the chosen model at w_i is m_i if the performance $perf(i − 1, s_{i−1})$ of m_{s_j} during week $w_{i−1}$ was lower than a threshold T, as shown in green on Figure 4.

CumProd: a priori algorithm where the model chosen at w_i is m_i if the cumulative product $\prod_{j=curr}^{i−1} perf(j, s_j)$ of the performance of the models of week w_j with $s_j ≤ j < i − 1$ was lower than a threshold T. This is shown in purple on Figure 4. We use a cumulative product such that the model ages over time, since the multiplication will reduce the value over time.

Algorithms **Best**, **BestUp** and **BestLoc** are weekly upper-bounds for our analysis, representing three ways to choose the best models a posteriori. While **Best** will always outperform the other two a posteriori algorithms, we included the latter to improve understanding of the findings. On the a priori side, **Diagonal** and **Fix** are heuristics based only on a model's age, whereas **Thresh** and **CumProd** are based on a model's evaluated performance and age.

To compare model switching heuristics, we aggregate the performance over the whole period (weeks w_0 to w_N), for each week's selected model, by summing up the weekly confusion matrices into one overall confusion matrix, i.e., counting up the true positives, false positives, etc. Averaging the weekly performance would not have worked due to weeks with very few or no brown jobs at all. We will as well compare those switching heuristics with **Always-Brown**, which is a random prediction based on the brownness ratio introduced in Section 5.1 and defined in Table 3. Finally, each switch heuristic has a life expectation (*LifeExp*), which is the median number of weeks a model is used before a new model is trained.

5 RQ1: CAN WE ACCURATELY DETECT BROWN BUILDS IN A LANGUAGE/ PROJECT-AGNOSTIC WAY ?

5.1 Hyper-parameter optimization.

Motivation. The purpose of this first validation is to identify the best hyper-parameters for our model and to evaluate if the classifiers' performance is relevant in practice.

Table 4: Model performance.

Project	Brown-Detector										Baseline														
	Local opti [%]					Global opti [%]					Random50 [%]					RandomB [%]					AlwaysBrown [%]				
	Brown			Safe		Brown			Safe		Brown			Safe		Brown			Safe		Brown			Safe	
	F1	Pre	Rec	Pre	Rec	F1	Pre	Rec	Pre	Rec	F1	Pre	Rec	Pre	Rec	F1	Pre	Rec	Pre	Rec	F1	Pre	Rec	Pre	Rec
A	67	56	85	92	71	62	55	72	86	75	19	30	50	70	50	15	30	30	70	70	23	30	100	na	0
B	73	69	76	85	80	67	57	81	85	64	21	37	50	63	50	19	37	37	63	63	27	37	100	na	0
C	38	41	35	96	97	36	35	38	97	96	5	5	50	95	50	3	5	5	95	95	5	5	100	na	0
D	35	32	38	96	95	24	21	27	96	94	5	6	50	94	50	3	6	6	94	94	5	6	100	na	0
E	88	84	93	88	74	84	76	94	88	58	27	58	50	42	50	29	58	58	42	42	37	58	100	na	0
F	52	51	53	95	94	46	38	58	95	89	8	10	50	90	50	5	10	10	90	90	9	10	100	na	0
OS	63	61	66	91	94	52	47	57	91	91	10	13	50	87	50	6	13	13	87	87	12	13	100	na	0
Median	63	56	66	92	94	52	47	58	91	89	10	13	50	70	50	6	13	13	87	87	12	13	100	na	0
Mean	59	56	63	91	86	53	47	61	91	81	13	22	50	72	50	11	22	22	77	77	16	22	100	na	0

Approach. For this RQ, we use all 7 projects, thus including the open-source project, using cross-validation to select the best set of hyper-parameters per project (local optimization) and for the seven projects at once (global optimization), see Section 4.2. The best hyper-parameter values for both local and global optimizations are gathered in Table 5.

Results. **Our models are two to three times better than the best random baseline *AlwaysBrown* in terms of F1-score for each studied project.**

Table 4 shows the performance per project of the local and global optimization of hyper-parameters, in comparison with the baselines. We observe that performance varies substantially from one project to another. This is due to the unbalance of brownness in build jobs across the studied projects: The higher the ratio of brownness in the failed build job, the higher the F1-score will be. For instance, project E has the best F1-score (88%) and the highest brown failure ratio among the studied projects ($BFR = 58\%$), while project C has the second-worst F1-score (38%) and the lowest brown failure ratio ($BFR = 5\%$).

When switching from local to global optimization, the F1-score decreases between 2% for project C and 11% for projects D and OS when choosing a globally optimal set of hyper-parameters.

The baseline that gets the best F1-score is *AlwaysBrown*. Yet, the mean F1-score of this baseline is more than three times (53% compared to 16%) worse than the global optimization of our models. Furthermore, the OS project shows results close to project F, which is the project with the closest *BFR*, providing initial evidence that the results could be generalized to open-source projects.

In addition to F1-score, Table 4 also shows the other performance metrics. The median precision (recall) goes from 56% (66%) for local optimization to 47% (58%) for global optimization. Having precision close to 50% can seem problematic, however, let us not forget that the dataset is highly unbalanced, and that the values are higher than the baselines. Furthermore, 89% of the true failures (non-brown) are correctly identified by our model and 91% of failure predictions are actually true failures, which meets our objective of making the CI results more reliable.

Regarding feature selection, we analyzed the features with significant tf-idf values of the OS project, to better understand what drives the models. We observed that occurrences of "read_databas"

Table 5: Optimal hyper-parameter values per project in terms of F1-score.

Optimization		Hyper-parameters				
Type	Project	F	N	K	α	β
	A	Train	[1]	300	0%	10%
	B	All	[1]	275	40%	20%
	C	Train	[2]	300	90%	20%
Local	D	All	[1, 2]	100	50%	10%
	E	All	[1]	300	50%	30%
	F	All	[1]	225	100%	10%
	OS	Train	[1]	200	0%	30%
Global	All	All	2	300	70%	10%

in jobs tend to be related to brownness and of "return_error" tend to be related to true failures, which are intuitive to understand. However, less intuitive observations were seen with features such as "build_object" or "occur_dure" that can be related to both labels, depending on the other terms in the logs.

5.2 Manual validation

Motivation. While the previous subsection compared our models to intuitive baselines, this section compares our models' performance to human experts from our industrial partner to validate the extent to which our prototype can perform as well if not better than experts in terms of predicting brownness quicker.

Approach. The manual validation was done on projects A and B only. Each expert is given 40 failed build jobs and must identify if they are true failures or brown. The final expert's performance is computed respecting the original confusion matrix of our model prediction. For project A, this contains TP:22%, TN:52%, FP:18%, FN:9% and for project B TP:30%, TN:40%, FP:23%, FN:7%.

Table 6 shows the mean prediction results for the experts in each project (A and B) by category. We use the mean because there were only respectively 2 and 4 experts in the projects and only 10 jobs per category, making outliers unlikely.

Results. **Project A's experts manually predict brown builds with mean F1-score 4% better than our models, while project B's experts manually identify jobs as brown with mean F1-score 17% lower than our models.**

Table 6: Manual validation's mean prediction [%] of the experts by category.

Project	Expected to be brown (100%)		Expected to be true-failure (0%)		Mean evaluation time by build job [mm:ss]
	TP	FN	TN	FP	
A	83	73	18	54	2:44
B	68	68	61	65	2:55

Table 7: Manual validation's performance results.

Project	Comparison	Brown			Safe	
		F1	Pre	Rec	Pre	Rec
A	Pred vs Oracle	62	55	72	86	75
	Expert vs Oracle	66	56	80	89	73
B	Pred vs Oracle	67	57	81	85	64
	Expert vs Oracle	50	39	68	38	37

From Table 6 (i.e., the results for project A), we observe that experts of project A have predicted TP jobs correctly in 83% of the cases and TN in 82% (100-18), whereas our model, by definition, identifies both groups correctly in 100% of the cases. However, the experts correctly identify 73% of FN and 46% (100-54) of FP, whereas our model misclassifies all the jobs. Project B shows less variation from one category of results to another, showing that the developers on that project globally identify jobs as brown in 61-68% of the cases, without significant differences between categories. For both projects, no significant difference was found in the survey (see Section 4.2.2) for predicting jobs when we provided our prediction or not.

In Table 7, we evaluated the same metrics as for the hyper-parameter validation, this time comparing the global optimization of the hyper-parameters for projects A and B with their respective experts' performance. The metrics are computed by weighing the different categories (TP/TN/FP/FN) with the real ratio of each category from the globally optimal model. We observe that experts from project A have better results than our model, with an F1-score of 66% (4% higher than our model). For project B, we see that our model has better results, with an F1-score of 67% (17% higher than the experts).

Even though our models' precision and recall are not perfect, we observe that they are similar to or better than experts' performance. Our model can thus be used to improve the CI system, by detecting brown builds before the developer has to rerun them manually, saving the experts' precious time. In fact, our model, once trained, only takes seconds to predict the status (brown or safe) of a new build jobs, reducing the effort needed by experts, since the experts of projects A and B took on average 2min44 and 2min55, respectively, to interpret the brown-ness of the analyzed build jobs.

6 RQ2: CAN A BROWN BUILD PREDICTION MODEL BE USED ON ANOTHER PROJECT ?

Motivation. One of the hypotheses we defined from the beginning was that a classifier have to be trained for each project. This is a limitation as each project needs a cold-start period to gather data

Table 8: Cross-project F1-score. The project "Pred" is predicted by the a model trained on the project "Train". (Diagonal is the global optimization and "★" is *AlwaysBrown*.)

Train Pred	F1-score							
	A	B	C	D	E	F	OS	★
A	**62**	32	3	45	49	1	5	23
B	15	**67**	27	47	39	13	4	27
C	2	16	**36**	16	12	17	8	5
D	8	12	12	**24**	11	7	4	5
E	53	80	0	35	**84**	5	23	37
F	10	20	na	6	18	**46**	29	9
OS	11	12	21	12	12	4	**52**	12

before being able to provide predictions, while brown build detection would be required from the start. To challenge this hypothesis, here we build and evaluate cross-project prediction models.

Approach. For this validation, we used each project's model to predict brown builds on other projects (which we call "1-to-1 approach"). The models use the hyper-parameter values that yielded globally optimal performance across the projects (for within-project prediction). We also considered applying a leave-one-out approach (training with all datasets but one, then testing on the latter), but this approach would be computationally much more expensive than the 1-to-1 approach (due to the very large training set) [29].

In this section, we will refer to "cross-project prediction" when a project is predicted by a model trained on another project, and will refer to by-project prediction otherwise (see section 5.1).

Results. **F1-score of cross-project prediction is between 4 and 84% (median of 12%) lower than the prediction by-project for all studied projects. Provided the right training project(s) can be identified, cross-project prediction could be a viable alternative, at least until a project-specific model is available.**
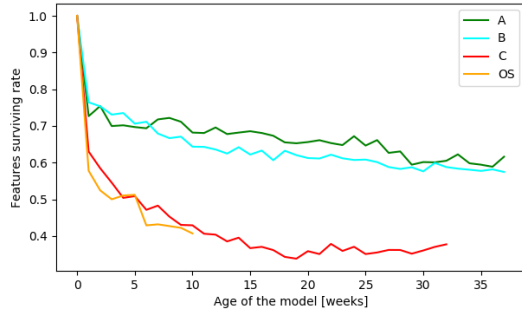
In Table 8, we gathered the performance of cross-project validation for the seven studied projects (for proj_x and proj_y in studied projects where $x \neq y$, proj_x is predicted by a model trained on proj_y). It can be seen that the cross-project prediction for all projects is 4 to 84% lower than the global hyper-parameter optimization. Some cross-project combinations still give results close to the by-project prediction. For instance, project E predicted by project B gives an F1-score of 80% (4% lower than by-project prediction). Other combinations do not perform prediction correctly at all, for instance, project F's prediction by project C always identifies failure as true-failure, yielding an "na"result.

One of the authors, who is an expert on project A, manually analyzed a sample of TP (true brown) predictions for project A. We extracted the corresponding TP predictions of each cross-project combination. From the 25 sampled predictions, 14 corresponded to cases the author was aware of, while 11 identified new brown build cases due to issues with the file system and network.

Furthermore, the F1-score of cross-project results are not consistently higher than the *AlwaysBrown* baseline, varying from an improvement of 43% to a loss of 37% on the F1-score (with a median of 3%). This validates that a project-specific model is essential as soon as historical data is available, but that, given the right training

Table 9: Switcher comparison with a win_{train} size of 30. The table vertically is split into a posteriori, a priori heuristics and baselines. Bold indicates the line with the best (F1-score,life expectancy) of a project per category.

Switcher	Project A						Project B						Project C						Project OS					
	Brown			Safe		Life	Brown			Safe		Life	Brown			Safe		Life	Brown			Safe		Life
	F1	Pre	Rec	Pre	Rec	Exp	F1	Pre	Rec	Pre	Rec	Exp	F1	Pre	Rec	Pre	Rec	Exp	F1	Pre	Rec	Pre	Rec	Exp
best (post)	**69**	**70**	**69**	**87**	**87**	**1**	**74**	**79**	**70**	**89**	**83**	**1**	**32**	**59**	**21**	**99**	**96**	**2**	**36**	**28**	**48**	**95**	**98**	**4**
bestup (post)	65	65	64	85	85	2	71	78	65	89	81	2	30	57	20	99	96	3	36	28	48	95	98	4
bestloc (post)	63	64	62	85	84	2	70	77	63	89	80	2	22	45	14	99	95	6	21	24	19	97	97	8
first	51	56	46	84	78	43	62	70	56	86	77	43	14	35	9	99	95	38	32	27	39	96	97	16
diagonale	56	59	54	84	81	1	67	75	61	88	79	1	22	45	15	99	95	1	**44**	**44**	**45**	**98**	**98**	**1**
fix (2 weeks)	57	59	54	84	81	2	66	73	61	86	78	2	**27**	**50**	**19**	**99**	**96**	**2**	39	40	39	98	97	2
fix (5 weeks)	59	60	57	84	82	5	66	73	61	86	79	5	24	44	16	99	95	5	42	39	45	97	98	5
fix (10 weeks)	56	59	54	84	81	10	67	71	64	85	79	10	20	40	14	99	95	10	38	33	45	96	98	8
fix (15 weeks)	56	59	53	84	81	15	66	70	62	84	78	15	22	42	15	99	95	15	34	29	42	96	98	8
cumProd ($T = 0.01$)	**60**	**61**	**60**	**84**	**83**	**6**	**67**	**73**	**62**	**86**	**79**	**5**	22	39	15	99	95	5	23	24	23	97	97	8
cumProd ($T = 0.02$)	59	61	58	84	82	4	66	74	60	87	78	5	20	42	13	99	95	4	23	24	23	97	97	8
cumProd ($T = 0.05$)	60	60	59	83	83	4	66	74	60	87	78	4	20	38	14	99	95	3	29	29	29	97	97	5
thresh ($T = 0.7$)	59	60	58	84	82	1	67	74	60	87	79	1	26	52	18	99	96	2	35	35	35	97	97	2
thresh ($T = 0.8$)	59	60	57	83	82	1	67	74	60	87	79	1	26	52	18	99	96	2	35	35	35	97	97	2
AlwaysBrown	23	30	100	na	0		27	37	100	na	0		5	5	100	na	0		12	13	100	na	0	
Best Hyper-param	**62**	**55**	**72**	**75**	**86**		**67**	**57**	**81**	**64**	**85**		**36**	**35**	**38**	**96**	**97**		**52**	**47**	**57**	**91**	**91**	

Figure 5: Feature surviving ratio per project with a win_{train} size of 30.

project(s), cross-project prediction could be a feasible, temporary solution.

7 RQ3: HOW LONG CAN A MODEL STAY RELEVANT WITHOUT BEING RETRAINED ?

Motivation. This RQ evaluates how well models can deal with and mitigate concept drift. In particular, we aim to (1) evaluate the size of the data needed for a model to achieve optimal prediction results and (2) to evaluate when to change models (and thus retrain).

Approach. For this validation, we used hyper-parameter values of the globally optimal models, see Table 5. Since we are evaluating the prototype over time, we needed datasets with a long history and a representative number of builds by month. This is why we selected projects with at least one year of data and at least 1k build jobs per month, leaving us with projects A, B, C and OS. In the case of OS, only the last 48 weeks of data were considered to respect the condition of 1k build jobs per month.

Results. **A window size of 30 weeks of data is sufficient to have significant performance, since adding more weeks does not considerably increase the performance ($< 0.1\%$) and , in the case of project A, using more than 40 weeks even harms the performance.**

We plotted the median performance (F1-score, precision, recall) of the models over time, depending on the project and win_{train} training window size. For space concerns, we added these plots to the replication package. These plots showed how the curves of the three metrics decrease between 0 and 10% per week, for each project. This decrease in performance over time indicates that retraining a new model at some point would be beneficial. We also find that the smaller the win_{train} size is, the lower the F1-score is. However, as we increase the win_{train} size, the median improvement on the F1-score becomes $< 0.1\%$. For Project A, a closer analysis showed that win_{train} sizes 45 and 50 obtain an F1-score worse than 40. This shows that using too old data harms the model, since the data is not coherent with more recent data. Closer analysis showed that a win_{train} size 30 gets the best results for all studied projects. Project OS was ignored for this experiment since the number of weeks was 48 and we needed up to 50 weeks of data in the training set.

The feature surviving ratio drops down to 58-76% after one week, then decreases consistently over time for all project.

Figure 5 shows the training feature surviving ratio for a win_{train} size of 30 weeks (i.e., the optimal value). Projects A and B show similar values, dropping quickly to a median feature drifting rate of 72% and 76% respectively, then reducing linearly to 60% at week 38. Project C and OS both have a median feature surviving ratio drops to 63% and 58%, then reduce to 35% at week 30 and 40% at week 10 respectively. The first drop for the three projects seems to be due to features local to the week of the training, only relevant to the prediction of the first few (1 to 2) weeks after training. The consistent decrease after that shows that other features get rejected over time, even if they stay relevant longer.

A priori switching heuristics achieve an F1-score about 7-10% lower than a posteriori heuristics, while keeping results 2-to-4 times higher than the *AlwaysBrown* baseline.

Table 9 shows all performance metrics that we used, as well as the life expectation *LifeExp* (in number of weeks) when using a particular switcher. We observe that the a priori switching heuristics have an F1-score 7-10% under the a posteriori heuristic *Best*, except for the project OS. For the latter project, the *Best* heuristic chooses the best model each week (performance ≥ diagonal), yet has a lower overall F1 score because the number of true/false positive and false negative fluctuates across the studied weeks (known as Simpson's paradox[31]). Compared to *diagonal*, *CumProd* gets the same F1-score for project B and C and a higher score for project A, while needing less training computation (the median *LifeExp* for project A is 6 and for project B and C is 5). Project OS's F1-score is 2% lower than the diagonal with the *Fix (5 weeks)* heuristic.

When comparing our switch heuristics with the *AlwaysBrown* baseline, we observed that our F1-score is two to four times better. When compared with the best hyper-parameter combination's performance, the F1-score performance loses 0-10% depending on the project when including the time constraint and using a priori switch heuristics instead of posteriori heuristics. Our prototype, once extended with a switch heuristic, has thus results significantly higher than the baseline. The model can adapt through time, retraining when it is needed to have the best performance possible.

8 THREATS TO VALIDITY

External validity. The approach targets projects involving multiple programming technologies, of which gaming projects are an extreme case, since builds involve code, AI, physics, 3D models, sound objects, etc. on dozens of platforms, each with their own SDK versions. Our algorithm was evaluated on seven large projects with various BFRs and build activity. We observed that the recall and precision performance decrease as the brown failure ratio decreases.

Hence, the approach is expected to generalize to projects with build logs with default verbosity and a reasonable BFR (i.e., balanced, unlike projects C/D). A project having a brown failure ratio lower than 1% might not justify the need for our algorithm, since the performance might not be sufficient.

We considered only one open-source project, Graphviz, among the 7 studied projects. Further analysis on other projects would be relevant to analyze the generalization of our results.

Our approach focuses on the identification of brown builds, but does not propose any solution on how to fix the identified brown builds (rerunning builds does not target the root cause of brown builds). Similar to the related field of flaky tests (subset of brown builds) [11, 37], fixing flakiness is an ongoing research area, especially since brown builds are even more challenging to deal with, due to the occurrence of different languages and build technology.

Construct validity. The oracle used for evaluating the brown build identification models relies on a heuristic, i.e., if a job is re-run at least once and changes results for the same commit-ID, it is identified as brown. However, most of the build jobs are run only once for a given commit-ID, and hence are considered to be true build failures or successes. If a developer forgot to re-run a brown build, our oracle would have missed it. We believe the risk of losing such builds in our oracle is limited because brown builds are relatively rare, and the developers of the 6 commercial projects have as practice to rerun the known cases of brown build to check if the build's status switches to success.

In our evaluation, we optimize the models for F1-score, which maximizes both precision and recall. However, depending on the use case, organizations adopting our models might prefer to tweak the model for better precision (less false alarms) at the expense of recall (missing brown builds), or vice versa.

Internal validity. Internal validity refers to alternative explanations of our results. The ground truth has been "labeled" (decision to re-run a build failure due to suspicion of being brown) by build experts of the corresponding project right after the build finished. Finding a better expert or time to do the labeling seems implausible.

Regarding the user study, the participating experts were experts in the project they were asked to evaluate builds for, but they were shown 6-month old builds they may not have been involved with. This design was used to counter potential learning effects.

9 CONCLUSION

Developers regularly experience brown builds, i.e., build failures not due to code changes, test cases or build logic, but due to factors outside their control. Our empirical study on build results of 7 multi-language projects, 6 developed by one of the leading AAA-game producers and one open-source project (graphviz), observed that between 5% and 58% of failed build jobs were brown, depending on the project, highlighting the need to address this brown build problem and propose a detection algorithm.

Our brown build detection algorithm is language- and project-agnostic, and obtained a median F1-score of 52%, more than two times higher than the *AlwaysBrown* baseline (for all projects), and similar to experts' F1-score (-4% to +17%), while reducing the effort needed by those experts. We showed that cross-project prediction can be a workaround for on-boarding new projects, but their performance is not consistently higher than the baselines. We recommend switching to a project-specific model as soon as possible.

Our study of the impact of concept drift on the models shows that our approach in its current form is sustainable over time, and forms a solid base for future research on brown builds. While models and data age over time and impact the performance, we found a sweetspot (30 weeks) in terms of the size of the training set and model change frequency (4-5 weeks), and we proposed promising heuristics for deciding about switching to a new version of a model.

In terms of implications for practitioners, we have shown how our language-independent models perform at least as well as human experts (RQ1), and also function in a predictive setting (RQ3); with the right training project, cross-project prediction can bootstrap a new project (RQ2). In terms of research implications, we have shown how build logs are sufficiently rich to predict brown builds in a real setting, independent from programming technologies.

These implications open new research directions. Apart from validating the approach (and future incarnations) on other systems, we believe that future work could focus on specialized models for identifying different subsets of brown builds (e.g., due to time-out vs. hardware failure), as well as on strategies to fix brown builds, once identified.

REFERENCES

[1] List of english stop words. Accessed February 2020. [Online]. Available: http://xpo6.com/list-of-english-stop-words/

[2] Openstack zuul ci dashboard. Accessed February 2020. [Online]. Available: http://zuul.openstack.org

[3] Replication package. [Online]. Available: https://github.com/ubisoft/ubisoft-laforge-brownbuild

[4] Shap values documentation. Accessed February 2020. [Online]. Available: https://shap.readthedocs.io/en/latest/

[5] Sklearn selectkbest python package. Accessed February 2020. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest

[6] Stemmer golang package. Accessed February 2020. [Online]. Available: https://github.com/caneroj1/stemmer

[7] Treeherder. Accessed November 2019. [Online]. Available: https://treeherder.mozilla.org/#/jobs?repo=mozilla-inbound

[8] Xgboost python package. Accessed February 2020. [Online]. Available: https://xgboost.readthedocs.io/en/latest/python/index.html

[9] B. Adams and S. McIntosh, "Modern release engineering in a nutshell–why researchers should care," in *IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 5. IEEE, 2016, pp. 78–90.

[10] A. Ahmad, O. Leifler, and K. Sandahl, "Empirical analysis of factors and their effect on test flakiness-practitioners' perceptions," *Journal of Software: Testing, Verification and Reliability*, 2021.

[11] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 433–444.

[12] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[13] W. B. Cavnar, J. M. Trenkle *et al.*, "N-gram-based text categorization," in *3rd annual symposium on document analysis and information retrieval (SDAIR)*, vol. 161175, 1994.

[14] R. Y. Chen, J. Schulman, P. Abbeel, and S. Sidor, "Ucb and infogain exploration via q-ensembles," *arXiv preprint arXiv:1706.01502*, vol. 9, 2017.

[15] A. Fujino, H. Isozaki, and J. Suzuki, "Multi-label text categorization with model combination based on f1-score maximization," in *3rd International Joint Conference on Natural Language Processing (ICNLP): Volume-II*, 2008.

[16] K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh, "Noise and heterogeneity in historical build data: an empirical study of travis ci," in *IEEE 33rd International Conference on Automated Software Engineering (ASE)*. ACM, 2018, pp. 87–97.

[17] T. A. Ghaleb, D. A. da Costa, Y. Zou, and A. E. Hassan, "Studying the impact of noises in build breakage data," *IEEE Transactions on Software Engineering*, 2019.

[18] J. Ha, J. Yi, P. Dinges, J. Manson, C. H. Sadowski, and N. Meng, "System to uncover root cause of non-deterministic (flaky) tests," 2016, US Patent 9,311,220.

[19] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *37th International Conference on Software Engineering (ICSE) - Volume 2*. IEEE Press, 2015, p. 39–48.

[20] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *IEEE 31st International Conference on Automated Software Engineering (ACM)*. ACM, 2016, pp. 426–437.

[21] H. Huang, H. Xu, X. Wang, and W. Silamu, "Maximum f1-score discriminative training criterion for automatic mispronunciation detection," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 23, no. 4, pp. 787–797, 2015.

[22] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[23] W. M. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A. E. Hassan, "Should i contribute to this discussion?" in *7th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2010, pp. 181–190.

[24] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: a study of java projects using continuous integration," in *11th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 2017, pp. 821–830.

[25] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *12th IEEE Conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 312–322.

[26] X. N. Lam, T. Vu, T. D. Le, and A. D. Duong, "Addressing cold-start problem in recommendation systems," in *2nd international conference on Ubiquitous information management and communication (ICUIMC)*, 2008, pp. 208–211.

[27] J. Lampel, S. Just, S. Apel, and A. Zeller, "When life gives you oranges: detecting and diagnosing intermittent job failures at mozilla," in *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 1381–1392.

[28] Z. C. Lipton, C. Elkan, and B. Narayanaswamy, "Thresholding classifiers to maximize f1 score," *Machine Learning and Knowledge Discovery in Databases*, vol. 8725, pp. 225–239, 2014.

[29] C. Liu, D. Yang, X. Xia, M. Yan, and X. Zhang, "A two-phase transfer learning model for cross-project defect prediction," *Information and Software Technology*, vol. 107, pp. 125–136, 2019.

[30] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 643–653.

[31] G. Malinas and J. Bigelow, "Simpson's paradox," 2004.

[32] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *39th International Conference on Software Engineering: Software Engineering in Practice Track (SEIP)*. IEEE, 2017, pp. 233–242.

[33] A. M. Memon and M. B. Cohen, "Automated testing of GUI applications: Models, tools, and controlling flakiness," in *35th International Conference on Software Engineering (ICSE)*. IEEE Press, 2013, p. 1479–1480.

[34] K. Pfeffers, T. Tuunanen, C. E. Gengler, M. Rossi, W. Hui, V. Virtanen, and J. Bragge, "The design science research process: A model for producing and presenting information systems research," in *1st International Conference on Design Science Research in Information Systems and Technology (DESRIST)*, 2006, pp. 83–106.

[35] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, *What is the Vocabulary of Flaky Tests?* Association for Computing Machinery, 2020, p. 492–502.

[36] J. E. Ramos, "Using tf-idf to determine word relevance in document queries," 2003.

[37] S. V. V. Subramanian, S. McIntosh, and B. Adams, "Quantifying, characterizing, and mitigating flakily covered program elements," *IEEE Transactions on Software Engineering*, 2020.

[38] M. Zolfagharinia, B. Adams, and Y.-G. Guéhéneuc, "Do not trust build results at face value-an empirical study of 30 million cpan builds," in *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 312–322.