Faster Feedback Through Lexical Test Prioritization

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Falco Dürsch
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
falco.duersch@student.hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Immediacy and continuity of feedback are desirable properties during programming. Automated tests are a widely used practice to gain feedback on whether test authors' expectations are consistent with an implementation. With growing test suites, feedback becomes less immediate and is obtained less frequently because of that. The objective of *test prioritization* is to choose an order of tests that catches errors as early as possible, ideally within a time frame that we can consider *live*.

Research in test prioritization often relies on dynamic analysis, which is expensive to obtain. Newer approaches focus on most recently edited source code locations and propose IR (information retrieval) approaches that regard a *change* to the software as *query* against a collection of tests.

We study the capability of the IR approach to reduce testing time in the presence of faults using the example of open-source Python projects, identify trade-offs in classical TF-IDF-based IR frameworks, and propose different approaches that consider lexical and semantic context of a change, including topic modeling.

We conclude that even simple IR strategies achieve *immediate* error detection, especially when tests themselves were edited alongside program code. We further discuss applications of this approach in live programming environments, where change granularity does not leave sufficient time to run a test suite entirely.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; Software version control; • **Computing methodologies** → *Information extraction*; • **Information systems** → Document topic models.

KEYWORDS

feedback, testing, topic models, information retrieval

ACM Reference Format:
Toni Mattis, Falco Dürsch, and Robert Hirschfeld. 2019. Faster Feedback Through Lexical Test Prioritization. In *Companion of the 3rd International Conference on Art, Science, and Engineering of Programming (Programming '19), April 1–4, 2019, Genova, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3328433.3328455

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Programming '19, April 1–4, 2019, Genova, Italy
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6257-3/19/04…$15.00
https://doi.org/10.1145/3328433.3328455

1 INTRODUCTION

Immediacy and continuity of feedback are desirable during programming activities. Automated tests, often manifesting themselves as *unit tests*, are a best practice to receive feedback on whether the test authors' expectations are consistent with the implementation at hand. With growing test suites and, consequentially, longer execution times, feedback becomes less immediate and is obtained less frequently by programmers. As a result, the benefits of frequent testing begin to cease, among them the cost benefits of early-caught errors, and the psychological benefits of perceiving causality between change and test failure.

Test prioritization. The goal of *test prioritization* is to choose a better order in which tests are executed – in our case, the test that is most likely to fail should run first. (Alternative ranking criteria include covering the source code as fast as possible or selecting *n* tests that are most likely to detect a fault together, but these are not considered here.)

While ranking tests does not speed up the test suite as a whole, it reduces the time programmers wait for test *failures* (if any are present) and increases the chance of fault detection under time constraints. Examples where this can be helpful include:

- Live programming environments [6] that rely on immediate feedback as primary part of their experience
- Programming environments that run tests on every change but are not given sufficient time to run the full test suite between subsequent edits
- Continuous integration services where a "red" build needs to be discovered and resolved quickly as it impedes progress
- Continuous delivery infrastructures with multiple stages, where the most frequently updated stage is tested against a small, but fast subset of tests, while more stable deployments are exercised with larger and larger test suites the closer they move to production.

Change-based Test Prioritization. Our focus on feedback during programming activities demands that we set our scope to approaches targeting the most recent change to a program. Our objective will be to find the tests that likely detect an error introduced in the immediate past and do so fast enough that prioritization itself does not significantly increase testing overhead.

Lexical Test Prioritization. Prioritizing tests based on the vocabulary they share with the most recent change is an approximate, but fast strategy to discover relevant tests. State of the art in lexical test prioritization [11] follows a standard IR approach, where lexical features in each test are TF-IDF (term frequency–inverse document frequency)-weighted, scored against a query computed from the

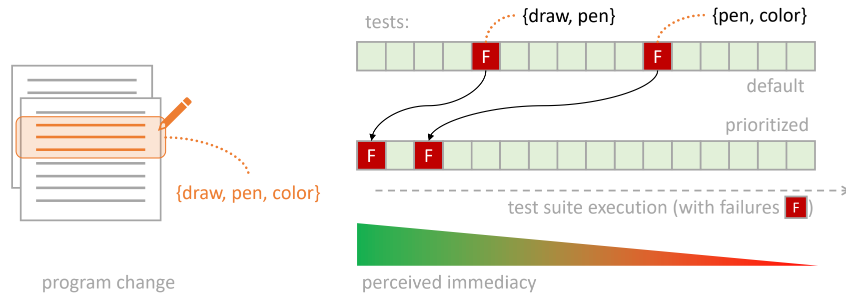


Figure 1: Lexical text prioritization aims at running tests first that share vocabulary with the most recent change.

last change, and returned in descending order with respect to this similarity score.

The IR approach to test prioritization does not yet include the module or concept structure of the system at hand, which makes them unable to detect semantically related changes that have no lexical overlap with the failing test. Examples where this happens include tests that probe an abstraction, while the underlying implementation is changed, or tests that use synonyms or semantically related words, but not the exact vocabulary used by the tested implementation.

In the scope of this work, we show that even without the inclusion of latent semantic information, modern TF-IDF-based retrieval strategies can save significant time, and explore the trade-offs involved in adding structural information or switching to a *topic model* that does not compare lexical features directly, but the more general *concepts* they belong to. We present a fault-seeding strategy that simulates faulty changes made by programmers and generates a large quantity of example test runs.

We then re-rank testing results based on the widespread IR approach BM25 (Okapi best match 25) and the LDA (latent Dirichlet allocation) topic model, and study their fault detection capacities with regard to *immediate feedback*. We further discuss how these approaches can be integrated into *live programming* environments to balance immediacy of feedback with the benefits of a test suite.

2 LEXICAL TEST PRIORITIZATION

Ranking tests based on how well their lexical features (e.g., identifier names) coincide with those affected by the most recent change is an inexact, but quick strategy to prioritize the tests that are semantically related to the change.

From an IR perspective, our tests play the role of documents to be retrieved in response to a query. The relevant variation points of an IR strategy are the conversion of a change to a query, the selection of lexical features, and the similarity metric used to compare tests with the query.

2.1 Background and State of the Art

State of the art in IR-based test prioritization relies on extracting variables, methods, classes, comments, and other programmatic concepts affected by a change, normalizing the features (e.g. by splitting camel case identifies, stemming, etc.), and assigning TF-IDF weights to them. Given a change to the code base, a query vector can be analogously computed from the features occurring in edited lines of code. The query vector is compared to each test's feature vector, usually using an inner product or variation thereof. In this section, we shortly describe BM25 and the LDA topic model.

TF-IDF. The idea behind the TF-IDF scheme is to construct per-document (in our case, per-*test*) vectors in which a weight is assigned to each feature (e.g. word or identifier) proportional to how often it appears within that document, called TF (term frequency), while discounting words that occur in a large proportion of documents by multiplying with the feature's IDF (inverse document frequency).

Additionally, the resulting weight can be adjusted by document length such that a feature occurring more often due to being in a longer block of test code is not at an unfair advantage. TF-IDF-vectors are sparse, the weight of a non-occurring feature can be implicitly regarded as 0.

BM25. The Okapi BM25 model is a widespread variation [8] of TF-IDF to compute a document d's score $S(q, d)$ with respect to a query q:

$$S(q,d) = \sum_{f \in q} \text{IDF}(f) \cdot \frac{\text{TF}(f,d)(k_1+1)}{\text{TF}(f,d) + k_1 \left(1 - b + b\left(|d|/\hat{d}\right)\right)} \tag{1}$$

Here, $\text{IDF}(f)$ of a feature f is defined as $\text{IDF}(f) = log\frac{N - n_f + 0.5}{n_f + 0.5}$ with N as the total number of documents and n_f the number of documents containing feature f. $\text{TF}(f,d)$ is the number of times feature f appears in document d, $|d|$ is the document length and $\hat{d}$ the average document length. k_1 is a free parameter that determines how fast term frequencies saturate, and the value of b determines how much the frequency is scaled with respect to the document length.

LDA. The LDA topic model [1] groups semantically related features into topics. Instead of reporting the frequency $\text{TF}(f,d)$ of a feature in a document or query, it reports the probability $p(t|d)$ of that document being concerned with a topic t and the probabilities $p(f|t)$ that each feature f belongs to a topic t. It is important to note that the topics and thus $p(f|t)$ are shared across all documents.

In the LDA model, the term frequency $\text{TF}(f,d)$ is considered a sample from the multinomial term probability $p(f|d)$ over all topics t:

$$p(f|d) = \sum_t p(f|t)p(t|d) \tag{2}$$

Two parameters, α and β control the sparsity of $p(t|d)$ and $p(f|t)$ respectively, i.e., for higher α more topics are allowed to emerge per document, while higher β allows more features to participate in a topic and thus increases overlap between topics.

Typically, the number of topics ($\approx 10-20$ in small programs) is much smaller than the number of features ($\approx 1000-1500$), hence dealing with $p(t|d)$ and $p(f|t)$ for every combination of input parameters is effectively a lossy compression of the full document-feature-matrix spanned by $\text{TF}(f, d)$.

Using LDA is a two-step process: First, topics need to be computed over a large training set of documents. This requires the full program to be broken down into documents (e.g. individual methods). Second, the test features need to be "compressed" to topic vectors, i.e., instead of storing $\text{TF}(f, d)$ for each feature, the topic vector stores $p(t|d)$ for each topic. The same applies to the query.

The scoring function needs to compare two topic vectors. Since they are probabilities that sum up to 1, the square root is taken to obtain a vector whose sum of squares is 1, i.e., a unit vector. This geometric representation can now be compared using cosine similarity:

$$S(q, d) = \sum_t \sqrt{p(t|d)p(t|q)} \quad (3)$$

More sophisticated comparisons between two probability distributions exist, e.g., KL divergence, or in the special case of probabilistic models like LDA, the conditional probability of a test given the query topics, but comparing these with respect to test prioritization is beyond the scope of this work.

2.2 Tradeoffs in IR-based Test Prioritization

IR-based test prioritization as implemented through TF-IDF retrieval models offers a number of trade-offs. The most important ones in the scope of this work are:

Context By using only the change as reported by a Unix-diff-like algorithm, the approach remains largely language agnostic, but the lexical context within the program structure (e.g. the surrounding method or class) is lost. Often, tests call a method or instantiate a class under test, while the change only affects the implementation thereof. The lexical context, e.g., the class or method name in which a change occurred could reveal which test is affected. In this work, we will explore the influence of including context in the query.

Structure TF-IDF weighting prioritizes features that have high information content based on their frequency, but not on their importance in source code. Related work has shown that differentiating between features from comments, class or method names, temporary variables, etc. can improve retrieval performance in fault localization tasks [10]. However, as shown by the *REPiR* project [11], the resulting test prioritization does not improve significantly.

Latent semantics/abstractions A TF-IDF model would only consider exact matches of features. Synonyms (e.g. *count / number*, *str / string*, ...) and semantically related words (e.g. *draw & color*, *file & read*, ...) are regarded as distinct. For example, a change in a *color*-related method may affect *drawing* tests, but their relation is not reflected yet. On closer consideration, semantic relatedness is often *asymmetric:* Drawing routines use colors, hence the *drawing* test may fail more likely when color-related code is modified, but no *color* test should fail when *drawing* routines are updated. One feature (*drawing*) is at a higher level of abstraction, and *color* is an implementation detail. In the scope of this work, we investigate the use of LDA-based topic modeling on ranking performance, and leave the asymmetric case for future work.

Updatability A simple TF-IDF-based model can be cheaply updated when a new test is introduced or an existing one is changed, as only IDF scores of the affected features need to be recomputed, as well as TF scores of the new/changed test. Any model involving latent semantics or abstractions is likely much larger as it is concerned with semantically related features beyond those found in tests. These models would need to be updated after any code change, or fully re-trained.

In our experimental setup, we will primarily focus on context and latent semantic relations, since they are still underexplored in related work.

3 CHANGE-BASED FAULT SEEDING

If historical code changes and subsequent testing reports from a program were available, we could evaluate test prioritization strategies without user involvement. Simulating the impact a "treated" test order would have had on the report and comparing metrics (e.g., position of first failure) helps estimating how much faster the desired feedback would have been obtained if the treatment was in place.

Unfortunately, realistic examples of changes causing test failures rarely leave traces in publicly available data. Most programmers ensure all tests are green before committing their change to a public repository, and only in infrequent cases did a continuous integration (CI) server (e.g. TravisCI) produce a detailed log of a failed build. In the light of this data scarcity, we see the need to synthesize faulty changes and corresponding test results.

A strategy used by Saha et al. [11] is focused on sampling *regressions* by running the regression test suite of an old version against a newer version of the program. This way, the old test suite lacks tests for new functionality, but the changes that lead to test failures represent real programming activity. In contrast, our approach aims at keeping test suite and program in sync, while relaxing the requirement that the fault is caused by the change.

Change-based fault distribution. To obtain synthetic, yet realistic, faulty changes and associated test results, we propose a fault seeding strategy based on the actual edit history of the program. We compare each version, e.g., a Git commit, against the previous version and only changed or inserted lines are considered for seeding faults. We exclude non-code files (documentation, configuration, resources, build scripts, etc.). Moreover, test code is excluded from fault seeding, because compromising one test should not cause any test other than the faulty one to fail if best practices have been followed.

Our fault seeding tool is designed for and written in Python. The approach is general enough to be applied to other dynamic

languages but is not guaranteed to generate a valid program under static type systems.

Mutation operators. The actual fault seeding relies on operators known from *mutation testing*. In our case, an operator is an AST (abstract syntax tree) transformation that consists of two rules: A definition to which AST nodes it applies, and a transformation that takes an AST node and returns a new one replacing the originally matched node.

We make use of similar operators as used in the Javalanche [12] tool. A notable difference is that we are working on ASTs rather than bytecode for practical reasons: since the parser keeps track of the origin of each node, it is easier to link AST nodes with modified lines of code in a Git commit. In contrast to Java infrastructure, our test runner operates on source directories rather than compiled artifacts, hence a source-to-source transformation makes sense.

Negate Branch Condition: Applies to ASTs nodes in the role of an if-condition. Mutation returns the ASTs node wrapped into a unary negation node.

Omit Method Call: Applies to Call nodes. Mutation returns the `None` literal. If the call was a statement, it has no effect anymore. If it was part of an expression, it simulates forgetting to return a value.

Swap Arithmetic Operator: Applies to ASTs nodes of binary operator type. Mutation swaps `+` with `-` and `*` with `/`. For simplicity, we ignore logical operators (their frequent use in conditions is already covered by Negate Branch Condition), rarely used bit-operations, and the modulo (`%`) operator which is primarily used for string formatting in Python.

Modify Number: Applies to ASTs leaves representing numeric literals. Mutation increments them by 1 with the intent of causing off-by-one or indexing errors. However, many numbers have non-functional roles, such as buffer sizes and timeouts, or are exchangeable by design, such as port numbers or error codes.

In a first step, each operator is applied to the program's ASTs and all suitable locations in the code are collected, no modification happens yet. For each operator, its set of locations is then intersected with the lines changed in the respective version, yielding a set of *candidate locations*.

If the candidate set is empty, this particular version is uninteresting and we proceed to the previous version (i.e., the parent Git commit). This usually happens when documentation or configuration fixes are being committed, but no change in program logic.

For each candidate location, we apply the matching operator once to create a *mutant* with a single seeded fault. This way, we obtain multiple mutants per program version, each representing a different fault that could have happened while the programmer was implementing this particular change.

Collecting test results. The collection process starts with the most recent version of a software repository and then iterates backwards through the main line of changes. This way, we can use an up-to-date test framework that works with recent versions and stop the iteration once a version is old enough to be incompatible to the current set of dependencies.

The test suite is version-controlled alongside the program, so we run the test suite corresponding to this particular version for the unmodified program (*control result*) and all mutants (*single-fault results*). For further processing of single-fault results we only consider tests that passed in the control result and were executed in the presence of the fault, too.

4 EXPERIMENTAL RESULTS

To assess the effectiveness of different variations of lexical test prioritization, we ran an exploratory experiment on three GitHub projects written in Python. We investigate the following prioritization strategies:

UNT The native (untreated) order of the test suites as chosen by the test runner.

BM25 Tests ranked according to their relevance score assigned by the BM25 model with $k_1 = 10$ and $b = 0.5$.

BM25C The same as BM25, except the change query includes lexical context, i.e., class and function names where the change occurred in.

LDA Tests ranked according to their topical similarity given by an LDA model trained on the most recent version with 20 topics, $\alpha = 0.05$, and $\beta = 0.1$.

RAND The tests in random order.

Our setup aims at assessing the following questions:

(1) To which degree do the three strategies improve test prioritization over the native ordering and a random order?
(2) Do inclusion of contextual features or a switch to a topic model improve prioritization?
(3) Regarding immediacy of feedback, how likely can a prioritized test suite detect faults when execution time is limited?

4.1 Implementation

Project selection. With Python, we have chosen an interpreted language that avoids build overhead before test runs and between different program versions. The *PyTest* test runner is easy to instrument to obtain the desired data about test runs, failures, and timings. For our projects, we expected them to be testable with PyTest directly from their download location given their dependencies are installed, and belong to the top 100 in popularity on GitHub (measured by number of stars after selecting those marked as Python) at the time of writing. Unexpectedly, almost none of the projects on GitHub turned out to be testable out of the box without compiling separate libraries, requiring a modified test runner, or a specific platform to run the tests. This left us with three popular projects to sample from[1]:

Flask After Django (which uses its own test runner) the second most popular Python web framework

Requests Library for web requests

Sphinx Documentation generator

Fault seeding. For each project and each Git commit, we perform a control run of the test suite (usually located in the `tests` directory) with Python's import path set to the checked out working tree of

[1]The PyTest project itself fulfills these requirements but had to be dropped. Our setup could not sufficiently isolate the colliding namespaces of the PyTest instance we use for testing and the fault-seeded PyTest version under test

Table 1: Number of commits, total number of faults seeded, and average number of tests executed per test suite run

Project	Commits	Faults	Tests per fault
Flask	74	322	364.7
Requests	38	55	505.6
Sphinx	30	355	1364.4

that commit. If the control run was able to report at least one successful test, all `*.py` files that have changed since the previous Git commit are gathered (via the *GitPython* library) and the differing ranges of lines are computed using Python's `SequenceMatcher`.

Subsequently, the files are parsed using the `ast` module to obtain their ASTs. Mutation operators, implemented as `NodeTransformer` subclasses (a mutating visitor pattern), are then applied in a "dry run" to gather candidate mutations. Candidates are identified by a mutant ID (concatenation of file path, operator name, line number, and column offset) and discarded when their line is not within one of the previously computed diff ranges.

For each remaining candidate, the operator that generated it is applied to the exact location only, the resulting ASTs is written back into the Git working tree, and PyTest executed in a separate process. PyTest is instrumented using a plugin that writes (test, success, duration)-tuples to a file that is identified by commit hash and mutant ID. Individual tests that did not succeed in the control run, and full test suite runs that match their control run exactly are discarded, since they did not detect failures caused by the seeded fault.

Flask and Requests began to show incompatibilities to the installed dependencies (we opted not to re-install dependencies for each version, since this would increase sampling time by orders of magnitude) and the sampling process for Sphinx exceeded 48 hours.

The number of selected commits, seeded faults (which is equivalent to the number of obtained test suite runs), and the average number of tests per run is listed in Table 1.

Test indexing. For each test run, we parse the source files containing the executed tests and extract the following features using a `NodeVisitor` subclass:

- Name of the test method
- Names of fixtures and parameters (in parametrized testing) given to the test
- The test method's documentation ("docstring")
- Identifiers (ASTs nodes of type `Name`)
- Strings

Composite features are broken down and normalized to lower case, e.g. `CamelCase` results in `camel` and `case`, `HTTPServer` becomes `http` and `server`, and any non-alphabetic character is considered a delimiter.

Change query construction. The query against the set of tests is constructed analogously to the feature extraction described for tests. As discussed above, the context of a change is not reflected in line-based diffs but might be important. To be able to assess the value of the outer lexical context, we provide two ways to construct a query from a change:

Without Context Used in BM25. The visitor, although it scans the whole ASTs, only emits features when the ASTs nodes lie within the diff to the previous commit.

With Context Used in BM25C. The visitor carries class and function names along and adds them to the list of features when a change affects the respective class/function.

LDA. To use the LDA topic model, we first need to train it on a set of documents. This set is constructed by collecting features as described for test indexing, except that all source code from the most recent version of the program is processed. We choose functions as units of granularity, i.e., features in the same function constitute a document. Surrounding features, e.g. those at class or module level, get their own document each. Definitions are duplicated to appear in both their implementation and the surrounding document, as the following example illustrates:

Listing 1: Example Python code

```
class C:
    v1 = v2
    def f1(self):
        g(); h()
    def f2(self):
        i(); j()
k = C(); l = C()
```

Listing 1 generates the following documents:

- Function *f1*: `f1, self, g, h`
- Function *f2*: `f2, self, i, j`
- Class C: `C, v1, v2, f1, f2` (Function names duplicated)
- Module: `C, k, C, l, C` (Class name duplicated)

To make sure that LDA training does not cause detrimental overhead, we use a low-level implementation of a Gibbs sampler. For this evaluation, we used a conservative number of Gibbs sampling iterations (500) that completes in under 6 s on the Flask project[2]. The *perplexity* metric that measures how well the topic model approximates the test suite only improves marginally after 50 iterations, which leaves room for trade-offs if faster training is desired. The topic model is never re-trained between runs, making training overhead a constant warm-up cost.

4.2 Results

To quantify the effectiveness of a prioritization strategy, we employ the APFD metric. We then compare the percentage of detected faults (recall) each strategy achieves when run under time constraints and measure the average time to detect the seeded failure over all test suite runs.

APFD. The APFD metric is frequently used to assess the effectiveness of a prioritized test suite. It is defined over a set of faults $F = \{f_1 \ldots f_m\}$ and a sequence of tests $t_1 \ldots t_n$ as:

$$\mathrm{APFD}(F; t_1 \ldots t_n) = 1 - \frac{\sum_{f \in F} \mathrm{argmin}_i\, t_i(f) = \mathrm{fail}}{n \cdot m} + \frac{1}{2 \cdot m} \quad (4)$$

[2]Compiler: `rustc` 1.31.1, Platform: Intel Core i7-8650U, 16GB DDR4, Windows Build 17763.253

Table 2: APFD (average percentage of faults detected) scores (first test failure only)

Project	UNT	BM25	BM25C	LDA	RAND
Flask	0.8036	0.9894	0.9945	0.9918	0.9366
Requests	0.8672	0.9983	0.9968	0.9791	0.8987
Sphinx	0.9600	0.9876	0.9878	0.9955	0.9831

Table 3: Percent of faults detected within 0.1 and 1 second of running tests

	UNT		BM25		BM25C		LDA		RAND	
Project	0.1 s	1.0 s	0.1 s	1.0 s	0.1 s	1.0 s	0.1 s	1.0 s	0.1 s	1.0 s
Flask	8.1	88.2	78.4	91.9	**79.3**	92.5	77.2	93.4	49.9	**96.0**
Requests	6.8	6.8	72.9	91.5	72.9	93.2	**78.0**	**98.3**	25.4	45.8
Sphinx	0.0	40.1	18.7	**63.8**	18.7	**63.8**	**21.2**	63.5	7.9	38.9

where $t_i(f)$ represents the result of the i-th test in the presence of fault f. The sum aggregates the positions of the first failing tests per fault. Tests vary slightly across commits, so we fix n as the maximum number of tests.

Since our fault seeding produces single-fault subjects, our APFD scores are not directly comparable to related work that computes scores form multi-fault test suite runs, where some faults can remain undetected for longer and reduce the score. The reported number is the APFD over all test suite runs with respect to all faults.

Table 2 summarizes the APFD score averaged over all test runs compared to the untreated (UNT) score that PyTest has chosen.

Detection probability. One objective of test prioritization is to run a smaller set of tests that still reveal relevant faults. Considering liveness as our goal, we use timeouts of 0.1 s and 1 s second respectively and measure the percentage of faults detected within this timeout in Table 3. The majority of faults can be detected within one second, and the two faster test suites achieve above 70 % fault detection rate within 0.1s. In the Flask project, a randomly ordered test suite outperforms all other strategies in the (1.0 ± 0.2) s range.

All prioritization strategies show an improvement compared to the untreated order of tests, with context-augmented BM25 and LDA performing slightly, but not significantly better than the baseline BM25 model with respect to both APFD and detection probability under time constraints.

Average time to detection. APFD scores only refer to the position of tests in the test suite, so we compare execution times until the first test failure in Figure 2, and also plot how the number of detected faults evolves over time when running the test suite. We assume the durations of individual tests are insensitive to their ordering, hence we are re-using durations measured during the untreated test run. The highly I/O-sensitive set-up time needed to discover and load tests from the file system is not included.

Measurements show strong variability (outliers omitted). Although average times to detect the fault are consistently lower with all prioritization strategies, variability suggests that prioritization can, in some cases, move tests to the end of the test suite when the unmodified test runner would run them earlier. Differences between the individual strategies are mostly insignificant given the variability.

4.3 Discussion

Our preliminary experiments, although only taken from three well-tested Python projects, are consistent with related work on regression testing and provide additional evidence that lexical test prioritization is able to shorten fault detection times up to a level suited for immediate feedback, a criterion not yet evaluated by previous research. Studying programs in other programming environments with qualitatively different test suites would be required to better support generalizability of the findings, for example live programming environments like Smalltalk.

The apparent problem addressed by topic modeling, i.e., the use of semantically related but not exactly matching names is *not* supported by this data, as TF-IDF prioritization performs so well that LDA could not add significant benefits except in a few test suite runs. However, the inclusion of contextual features has shown minor improvements. We cannot conclude from this experiment that the benefits of LDA or even the context-augmented BM25 model, are worth the added complexity yet.

Upon further investigation, it is common in open-source projects to not only modify program logic, but also tests within the same commit. Even if these tests are disjoint from the set of failing tests, they introduce vocabulary into the change that is used by the relevant tests as well (e.g. fixtures). A follow-up study could investigate how much improvement is explained by programmers modifying their tests compared to modifications to program code only.

Limitations. The scope of our conclusions is limited given the small selection of projects in a single language. Moreover, they are well tested compared to projects with less users and contributors, and mostly relate to web-centered topics where the Python ecosystem sees widespread use. The size of the test suites is less than 2000 tests (including parametrized tests) and they run in less than 10 minutes on consumer hardware, so we cannot generalize this to larger test suites yet.

A property of the PyTest framework is that it loads dependencies and scans the full project directory upfront, which is a constant

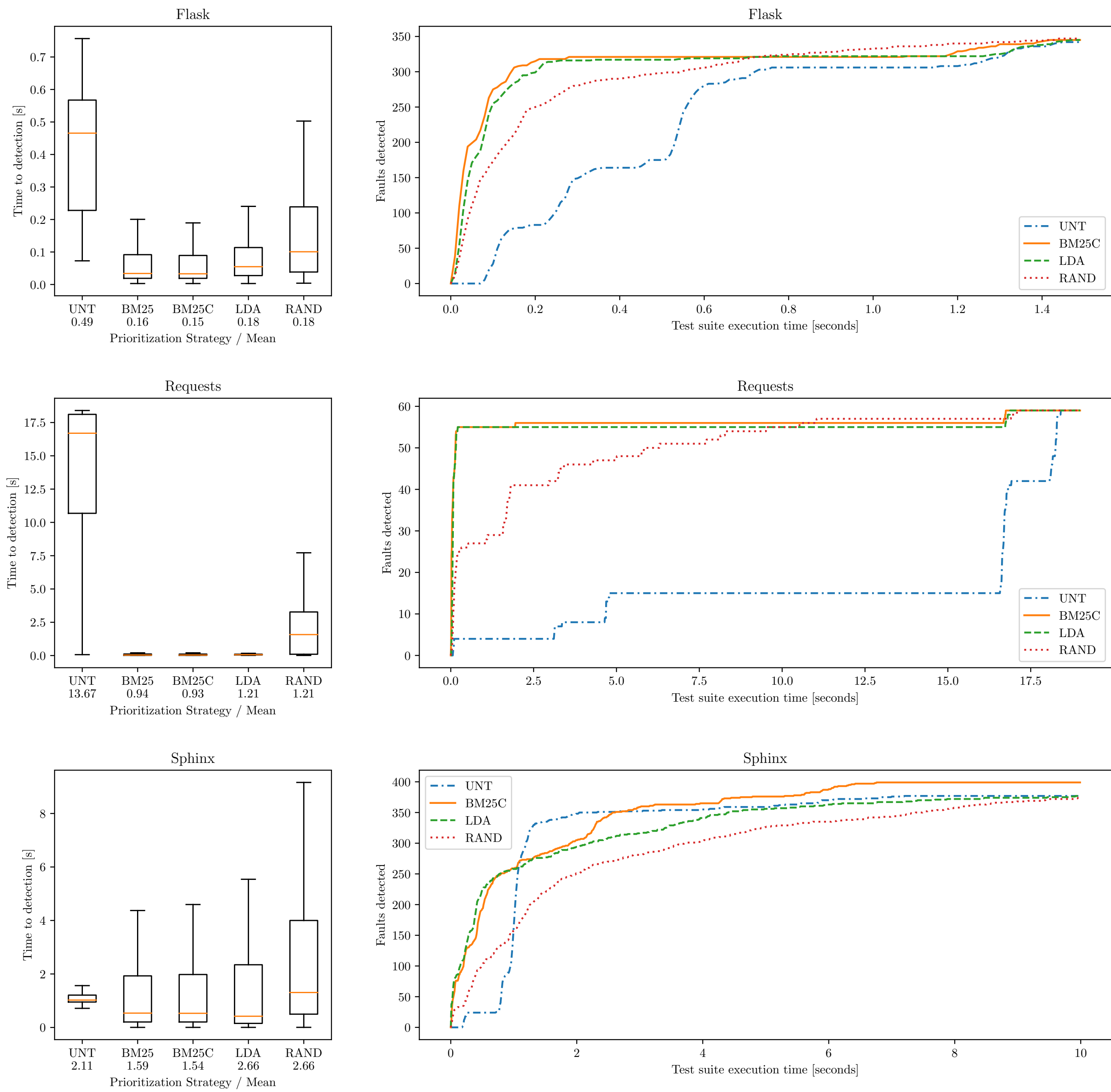


Figure 2: Time to detect a fault and cumulative faults detected over time for the example projects

overhead we did not measure. Similar cold-start behavior should be expected from most test runners, except those that never shut down the execution environment in between test runs (e.g. SUnit in Smalltalk).

Regarding the used algorithms, we are dealing with hyperparameters that might not be optimally tuned, i.e., k_1 and b in the Okapi formula, and α, β, and the number of topics in LDA. The Okapi parameters originate from from related work [7], while LDA parameters were determined by searching a limited section of the parameter space and choosing the values that managed to represent the test suite of the most recent commit with lowest perplexity.

Usage of LDA for source code is notoriously difficult since the way code is broken down into documents leaves much room for variation, hyper-parameters are rarely transferable between software projects, and document sizes are rather small. Hence, we expect that there are better configurations of LDA, code-oriented topic

models, such as [3], or more recent software similarity metrics, such as those derived from kernel-based learning [9], which are worth studying in future work.

On one hand, the fact that we trained LDA only once per project and re-used it limits its ranking performance when confronted with tests from distant versions. On the other hand, updating a topic model for every change incurs substantial run-time costs. Viable trade-offs involve re-training the model occasionally when the programming environment is idling, or re-training it externally by the continuous integration infrastructure, but more research is needed on the trade-offs of incrementally updating topic models used in development tools.

5 OUTLOOK: LIVE TESTING

Our exploratory results show that shared vocabulary between tests and production code allows to identify tests that are likely to fail, and manage to detect faults within (almost) immediate time frames with high probability. This gives rise to further exploration in the context of live programming environments.

AutoTDD. Our proposed tool integration targets live programming environments, such as *Squeak/Smalltalk*[3]. We propose a workflow in which tests are executed after every modification, i.e., saving (via `ctrl + s`) in Squeak. Existing testing frameworks, such as *SUnit* in connection with edit-triggered test runners (*AutoTDD*[4]) already provide this functionality. They can be extended to collect edit features and prioritize tests accordingly. Additional priority can be given to recently edited tests and recently failed tests.

Continuous testing. A yet unexplored challenge emerges when test runs are restarted after every edit, leaving not enough time for low-priority tests. A *continuous testing* approach must balance the probability of catching an error against the uncertainty associated with not running a test for a long time. Relevant insights could be gained by studying the granularity and time intervals in which changes are saved in a live programming environment, how these relate to test execution times, and how much of an asynchronous delay programmers are willing to accept to maintain the impression of causality and continuity.

Concept-aware programming environments. Combining live programming environments with automated topic/concept mining capabilities, like proposed in [4], would also associate test cases with concepts. Instead of using lexical features only, test ranking would operate similarly to what we did using LDA, except the underlying topic model would learn from a wider range of programming activities, e.g. co-changed code locations. Since LDA does not seem to perform worse than TF-IDF-style IR, a model that not only learns the vocabulary distribution of a program, but also auxiliary information is a promising candidate for test prioritization.

Live examples. A novel approach to integrate example data with source code has been designed by Rauch et al. [5]. Especially in live programming environments, tests and examples serve a similar role in providing immediate feedback. Both are conceptually interchangeable, i.e., (failing) tests can provide explorable examples while an example with an assertion/expectation becomes a test case. If a large set of examples is present, the same prioritization strategies that help selecting important tests may help selecting live examples as well. Their goal could be to provide the user with examples that are most relevant in the context of the current editing task.

[3]https://squeak.org/, retrieved 2019-01-30
[4]https://github.com/hpi-swa-teaching/AutoTDD, retrieved 2019-01-30

6 RELATED WORK

Saha, Zhang, Khurshid, and Perry studied the effectiveness of IR-based regression test prioritization with their *REPiR* project [11]. With a focus on regression testing, the authors generate test failures by running new versions of Java projects against the test suite of the previous version, thereby obtaining regression faults. Their approach uses the Okapi TF and IDF weights on both query and document features. A notable extension of classical IR is their strategy to classify features according to their role in the source code and to the type of change they appear in, which allows comparing different roles (e.g. added/deleted fields, added/deleted methods) separately. This approach has been successfully used for bug localization in the *BLUiR* system before [11]. Due to differing foci on regression testing and live unit testing respectively, and different fault generation strategies, our results are not directly comparable to *REPiR*.

Static test case prioritization operates without change data and tries to spread similar tests to cover many concepts (diversity) or most of the program (coverage) upfront [2]. Topic models have been used by Thomas, Hemmati, Hassan, and Blostein [13] to address this problem from a lexical viewpoint.

ACKNOWLEDGMENTS

To the HPI Research School for Service-Oriented Systems Engineering for supporting this research, to Stefan Ramson and Patrick Rein for comments on the approach, to Tobias Pape for editorial and typographic advice, and to all reviewers and participants of the *PX/19 Writers' Workshop* for insightful feedback and discussions.

CONCLUSION

Unit testing and immediate feedback are not mutually exclusive as long as running the test suite yields the most relevant feedback upfront. We explored the field of lexical test prioritization and conducted a preliminary study that demonstrates how unit test suites of several hundreds of tests can detect faults in under a second when prioritized by the amount of vocabulary they share with the change containing the fault. The results highlight that simple models, which are likely the least expensive in terms of implementation and run-time costs, already offer competitive performance.

Motivated by these results, we look forward to integrating test prioritization in live programming environments and studying the interplay between short edit cycles, feedback generated by unit testing, and the capability to interact with live objects and examples during programming activities.

REFERENCES

[1] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3 (March 2003), 993–1022.
[2] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A Large-Scale Empirical Comparison of Static and Dynamic Test Case Prioritization Techniques. In

Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016). ACM, New York, NY, USA, 559–570. https://doi.org/10.1145/2950290.2950344

[3] Toni Mattis. 2018. Mining Concepts from Code Using Community Detection in Co-Occurrence Graphs. In *Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming (Programming'18 Companion)*. ACM, New York, NY, USA, 232–233. https://doi.org/10.1145/3191697.3213797

[4] Toni Mattis, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2017. Towards Concept-Aware Programming Environments for Guiding Software Modularity. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience (PX/17.2)*. ACM, Vancouver, BC, Canada, 36–45. https://doi.org/10.1145/3167110

[5] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-Style Programming. *The Art, Science, and Engineering of Programming* 3, 3 (Feb. 2019), 9:1–9:39. https://doi.org/10.22152/programming-journal.org/2019/3/9

[6] Patrick Rein, Stefan Lehmann, Toni Mattis, and Robert Hirschfeld. 2016. How Live Are Live Programming Systems?: Benchmarking the Response Times of Live Programming Environments. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop (PX/16)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/2984380.2984381

[7] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (April 2009), 333–389. https://doi.org/10.1561/1500000019

[8] S. E Robertson, S Walker, and M Beaulieu. 2000. Experimentation as a Way of Life: Okapi at TREC. *Information Processing & Management* 36, 1 (Jan. 2000), 95–108. https://doi.org/10.1016/S0306-4573(99)00046-1

[9] Amir Saeidi, Jurriaan Hage, Ravi Khadka, and Slinger Jansen. 2019. Applications of Multi-View Learning Approaches for Software Comprehension. *The Art, Science, and Engineering of Programming* 3, 3 (2019), (to appear).

[10] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. 2013. Improving Bug Localization Using Structured Information Retrieval. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Piscataway, NJ, USA, 345–355. https://doi.org/10.1109/ASE.2013.6693093

[11] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 268–279.

[12] David Schuler and Andreas Zeller. 2009. Javalanche: Efficient Mutation Testing for Java. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. ACM, New York, NY, USA, 297–298. https://doi.org/10.1145/1595696.1595750

[13] Stephen W. Thomas, Hadi Hemmati, Ahmed E. Hassan, and Dorothea Blostein. 2014. Static Test Case Prioritization Using Topic Models. *Empirical Software Engineering* 19, 1 (Feb. 2014), 182–212. https://doi.org/10.1007/s10664-012-9219-7

APPENDIX

Uncalibrated APFD scores. To make our prioritization curves quantitatively comparable to related work, we also report *uncalibrated* APFD scores in Table 4. We use the same formula as above, but assume each test failure constitutes an individual fault. This assumption is not compatible with our single-fault seeding strategy, but can be made when information on the underlying number of faults cannot be obtained.

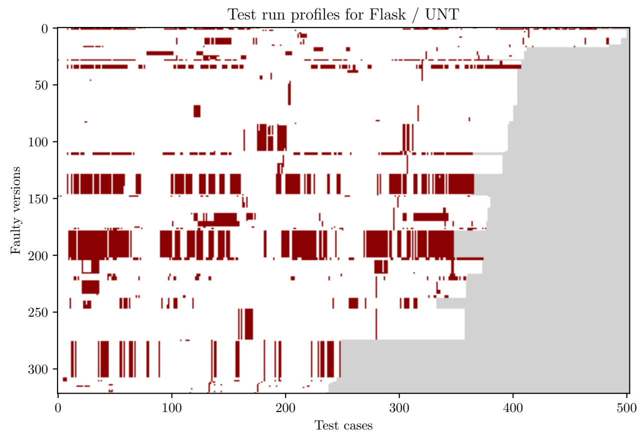
Uncalibrated APFD scores are much lower, as test failures occurring later in the test suite are considered as well, and exhibit much higher variability between test runs. The reported range spans 95% of the test runs (not a confidence interval).

Profiles of test runs. To visualize the raw data on which the metrics computed above are based, we plotted each test run from the Flask project in Figure 3. Each one-pixel line constitutes a full run of the test suite on an individual faulty version, test failures are marked. The gray area is padding, as test suites had fewer tests in earlier revisions.

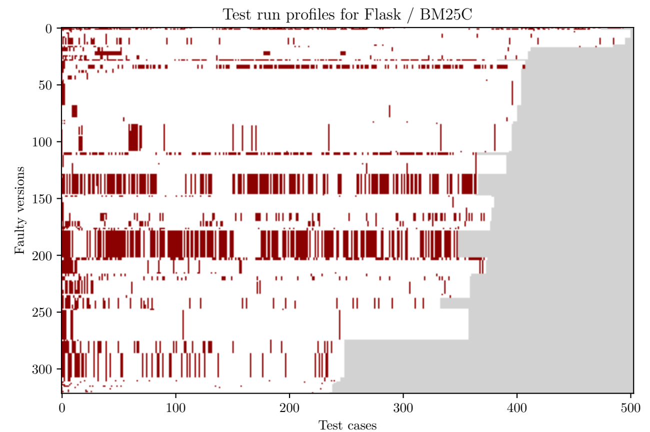
In the untreated version, clustering can be seen as failures often co-occur within the same test class or module, and tests within the same unit of modularity are executed together. When ranked deterministically, e.g. using BM25 with context, the "bar code" pattern intensifies as individual test cases are moved towards the beginning of the test suite. If seeding different faults in the same Git commit causes the same tests to fail, BM25 would re-order them in the same way, as the query vector remains the same for that particular commit. This is the reason that similar lines in the untreated profiles remain similar in the ranked version. The "bar code" dissolves with the LDA topic model as estimating the topics from the change uses a probabilistic algorithm. Since we re-compute topics each time, even if the faulty version is based on the same change, a minor variation in ranking can be observed.

Table 4: Uncalibrated APFD scores (all faults as failures)

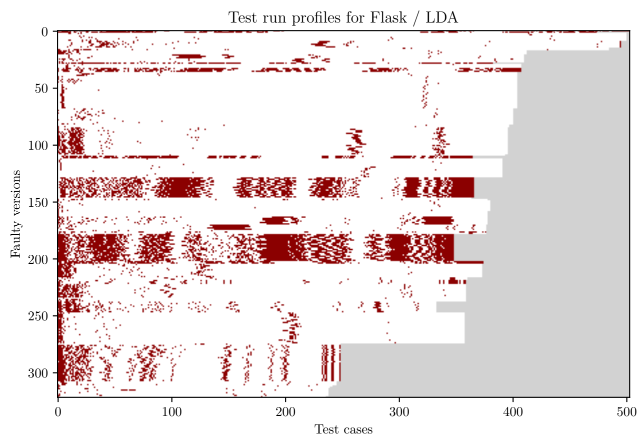
Project	UNT	BM25	BM25C	LDA	RAND
Flask	0.477 ± 0.291	0.679 ± 0.238	0.675 ± 0.242	0.667 ± 0.252	0.507 ± 0.202
Requests	0.627 ± 0.270	0.701 ± 0.227	0.690 ± 0.211	0.695 ± 0.191	0.485 ± 0.205
Sphinx	0.489 ± 0.225	0.586 ± 0.222	0.590 ± 0.233	0.574 ± 0.198	0.500 ± 0.087



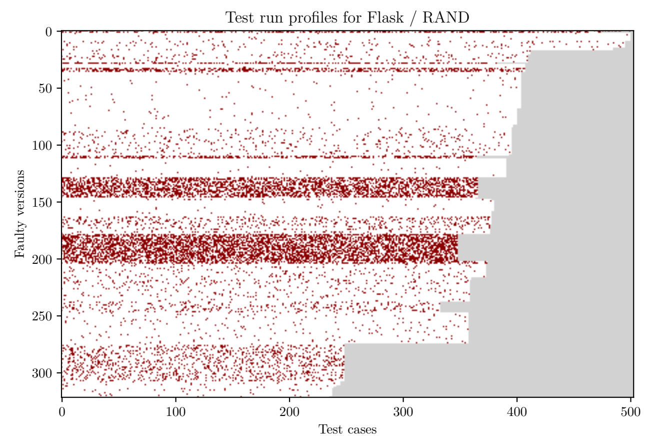
(a) Untreated test runs (Flask)



(b) BM25C-ranked test runs (Flask)



(c) LDA-ranked test runs (Flask)



(d) Shuffled test runs (Flask)

Figure 3: Test run profiles, one line per test suite run, each pixel representing a test success (blank) or failure (dark red)