2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)

Using Machine Learning to Recommend Correctness Checks for Geographic Map Data

Abhaya Parthy, Leo Silberstein, Emily Kowalczyk, John-Paul High, Adithya Nagarajan, Atif Memon
Apple Inc.
Cupertino, USA
{aparthy, lsilberstein, ekowalczyk, jhigh, nadithya, atif_memon}@apple.com

***Abstract*—Developing an industry application that serves geographic map data to users across the world presents the significant challenge of checking the data using "data correctness checks." The size of data that needs to be checked—the entire world—and data churn rate—thousands per day—makes executing the full set of candidate checks cost prohibitive. Current techniques rely on hand-curated static subsets of checks to be run at different stages of the data production pipeline, These hard-coded subsets are uninformed of data changes, and cause bug detection to be delayed to downstream quality assurance activities. To address these problems, we have developed new representations of map data changes and checks, formally defined "check safety," and built a recommender system that dynamically and automatically selects and ranks a relevant subset of checks using signals from latest data changes. Empirical evaluation shows that it improves (1) efficiency by eliminating 65% of checks unrelated to changes, (2) coverage by recommending and ranking change-related checks from the full set of candidate checks, previously excluded by the manual process, and (3) overall visibility into the data editing process by quickly and automatically identifying latest fault prone parts of the data.**

***Keywords*-automated data validation, data check recommender, safe test/check selection**

I. Introduction

Correctness of large software systems such as Apple Maps (or simply Maps) relies on the combined correctness of its services' code, its vast and diverse underlying data, and the interactions between them. For example, for "routing" to work correctly, Maps routing service code as well as the road network—the underlying data—must be correct. Missing roads, improperly labeled one- and two-way road segments, and incorrect road elevations are a few examples of data errors that may produce incorrect routing output [1]–[3]. Not only must the services code be tested using conventional code-based test cases, the data too must be validated using *data correctness checks* [4]. For example, checks may be written to verify that all adjacent road segments have the same elevation, or to compute the reachability of the road network by modeling it as a directed graph, or checking that buildings are not floating in the air. In general, data correctness checks verify that map data follows the rules of the physical world [5].

Verification of map data is non-trivial because of its vastness and diversity [6], [7]. The number of data elements runs into the billions; their variety and interactions makes their impact on services' code unpredictable. Thousands of changes are made to the data every day. Data dumps from 3rd-party sources and transit agencies need to be conflated with the master data repository. Data validation checks are run billions of times for an Apple Maps release; some individual checks can run for hours on massive compute resources. It is impossible to run all checks for each data change.

The Maps team has made the data validation process tractable by controlling it along two dimensions. First, they have created subsets of checks to run, called check groups, instead of running them all. And second, they run checks on batches of changes, instead of running on each change individually. More specifically, hand-curated check groups are hard-coded—irrespective of the type of data changes—to be executed at certain hand-selected points (data editing, pre-merge, post-merge, vendor conflation, nightly, pre-release, etc.) in the data production pipeline. We have observed that this change-agnostic hard-coding creates two problems: (1) many of the checks in these groups are unrelated to the changes, and hence miss faults, and (2) other checks that are in fact related to the changes end up being excluded. This causes errors in the data to go undetected until a pre-release *catch-all* run of all checks on the entire data.

It is desirable to detect errors as early in the data curation process as possible. As we move from region to region, mapping new countries and continents, the rules change – addresses follow different formats; turn restrictions vary; drivers use different sides of the road; some roads/streets have no names. Consequently, checks developed for one region start to fail in other regions. We need ways to quickly understand factors in our data that lead to new failures so we can take corrective actions quickly.

In this paper, we describe how we advance the software engineering of our Maps product in two ways. First, we develop a dynamic check recommender to select and rank only those checks that are relevant to the latest data changes, helping us to detect failures faster and save significant compute resources. Second, we quickly percolate the latest failure-causing features of our underlying data, giving us unprecedented visibility into our overall maps development process that can inform our data editors and check writers. We formally model our data into features that form the elements of maps (roads, buildings, etc.) and their fields (elevation, direction, etc.). We model the set of changes made to maps data at various stages of maps creation (data editor, pre-merge, post-merge, etc.). And we model the data validation checks in terms of the features/fields

978-1-7281-1760-7/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-SEIP.2019.00032

they validate. We use these models to formalize "safety" for check selection; intuitively, a subset of checks is safe if it is guaranteed to detect all defects detected by the full set. Our safe subset of checks also serves as a "filtering step" for subsequent machine learning, which ranks the checks in terms of their likelihood of detecting defects, and outputs the features/fields mostly responsible for latest failures.

Past research in data validation and software testing has helped to inform our current work. We borrow the notion of "safety" from selection of code-based test cases [8]–[11], and ranking of checks from prioritization of code-based tests [12]. Our modeling of maps features and checks is related to mapping points/areas in a continuous space and spatial dependencies between features [4]. We leverage concepts from test selection that aim to test changed parts of a software's code, although they are between code elements (lines, functions, files, etc.) and tests, e.g., static dependency graphs [13]. We, on the other hand, are creating mappings between changes to data elements and data validation checks.

We have implemented our check recommender and deployed it in production. Our results thus far (we discuss results of 465,500 pre-merge requests over several months in Section IV) have shown that, on average, we can reduce the set of checks to run in pre-submit by 65%. Our formal modeling revealed gaps in the metadata associated with dozens of check definitions, which have now been fixed. Our safe selection revealed gaps in the coverage of hard-coded, hand-curated check groups. Our Machine Learning (ML) models quickly and automatically identified the root cause of new failures that were due to differences in postal codes between US, Canada, and Great Britain.

Our paper makes the following contributions: (1) developing "map data diff" and using it to dynamically compute a relevant subset of validation checks to run, (2) formalizing maps data features/fields, checks, and mapping data changes to checks, (3) conceptualizing "check safety" for data validation and coverage, (4) using ML for ranking checks and identifying root cause of failures, and (5) empirically evaluating our approach on the vast Apple Maps data.

In the next section we provide more background and a discussion of related work. In Section III we discuss our formal models and definition of check safety. In Section IV we present our experimental results. Finally, in Section V, we conclude with a discussion of ongoing and future work.

II. Related Work

We now discuss prior research in data validation and software engineering most closely related to our work on selecting data correctness checks relevant to changes, ranking checks based on fault detection, and using the models for insights into failures in a large industry data production pipeline.

A. Data Validation

At Apple Maps, we work primarily with geographic data, i.e., data with a location component and attributes such as descriptions, measurements, or classifications. Geographic data has unique properties such as the correspondence of a record to a point/area in a continuous space and spatial dependencies between items/features [4]. These properties have resulted in unique internationally defined quality criteria for geographic data, such as the ISO 19157:2013 standard that identifies elements of quality including logical consistency and completeness as well as positional, thematic, and temporal accuracy [5]. Numerous researchers have proposed validation techniques for geographic data including metrics and topological rules for consistency checking [6], [7]. Additional challenges have also been identified including validating features at various scales [1] and stitching data from multiple sources [2], [3].

The broader field of data validation has focused on data validation challenges and techniques, far less on optimizing validation approaches for practical use in industry. To the best of our knowledge. the closest work in this area to ours is primarily theoretical, which seeks to improve the efficiency of integrity checks by generating checks based on a database's newest updates [14], [15]. Numerous methods for evaluating and assuring data quality and correctness have been proposed [16]–[19], including automatically checking consistency and accuracy using conditional functional dependencies [18], generating data correctness rules [20], and applying outlier detection for error discovery in temporal data [21], [22].

B. Test Case Selection & Prioritization

Our approach of creating the "safe subset of checks" builds upon prior work on test safety [8]–[11], which identify and link database dependent statements in code and tests using static analysis. Ranking tests by most likely to fail is related to test case prioritization in software testing, which finds an ordering of tests to maximize early fault detection. Various approaches have been proposed including ranking by estimated ability to find faults [23], [24], code coverage [12] and multi-objective approaches [25]. Memon et al. study the commit and test result history for Google [13] and find tests farther away from the commit's modified code are less likely to detect breakages, while frequently modified code and certain languages/editors/tools increase likelihood. Yoo and Harman [25] provide a survey defining the test case selection and prioritization problems and outlining various approaches, while Kazmi et al. [26] provide a more recent examination of test selection techniques exclusively.

We rank checks using gradient boosted trees; our work is related to ML-based test case prioritization in software regression testing, which typically use ML models to rank tests fault detection ability or predict buggy code [27]–[29]. Most recently, Spieker et al. [27] prioritize tests likely to reveal faults based on past test results. Busjaeger and Xie [28] use more extensive features (code change, change/test text similarity, test histories, and test age) to prioritize fault-revealing tests. Feature engineering techniques using deep learning have also been proposed. Wang et al. [29] use Deep Belief Networks to learn semantic features from tokenized abstract syntax trees, which can be used in trained models for test prioritization and fault prediction. Azizi and Do [30] present an item-based



Fig. 1. An Example Map Region

collaborative filtering recommender system which uses user interaction data and application change history to develop a test case prioritization technique.

While these techniques are conceptually similar to our work, they differ in several ways, including the nature of the items selected (tests vs. data checks), the ability to construct an accurate mapping (tests to code vs. checks to data), and type of system under test (code vs. data). Unlike previous work, we present a practical, multi-purpose system that not only selects and prioritizes, but also provides insight on failures to assist triage or data check maintenance efforts.

III. Formally Modeling Check Selection/Ranking

We now formally model the mapped world, data validation checks, and the coverage relationship between the two. We then model a "data diff" that represents changes to the map data, and use it to formally define safe check selection. Finally, we discuss how we leverage this representation to build our machine learning models for check ranking.

A. The World as Data

Consider a map shown in Figure 1. It shows a very small area, less than one square mile, yet it is still incredibly dense with information. We have one- and two-way roads, intersections, traffic signals, buildings, parks, water bodies, etc. When creating such a map, we need it to follow certain rules that reflect our understanding of the world and the relationships between its constituent individual elements, e.g., road segments, buildings. For humans these rules are so intuitive, they almost seem like axioms: if a road overlaps with a water body, it must be a causeway/bridge/tunnel; an intersection with 4 one-way streets cannot have them all going towards (or away from) the intersection; in a 3D map, a building must not be floating in air. A human looking at a map that violates one of these rules is likely to notice such issues very quickly. Computers lack this intuition, which is why we need to precisely encode these rules for data validation.

We now present a unified representation of the elements of our map and their relationships, with examples of how a single entity (e.g., road intersection) may be represented by one or more *features*. Intuitively, a feature represents a physical entity in the world, or a relationship between features. For example in our map in Figure 1, we break up the roads into multiple *node* features, each of which represents a point on the road; at the same time we represent the buildings as individual features. Each feature in the map is assigned an *id* and a set of *tags*. *Tags* are key-value pairs describing geographic attributes of the given feature. The tags that are needed vary depending on the type of feature it is, how it is situated, and its complexity. Consider a tiny part of our map of Figure 1 that contains the *Barrack Square* road and its intersection with *Barrack Street*. We model the roads as a collection of features: first we have a single feature representing the entire road; this contains a *name* tag, as well as reference to all the points on that road (assigned the tag *type: nodes*). Additionally each feature contains a number of other fields: latitude (lat) and longitude (lon), $subtype$, the $next$ node, etc. In the raw data representation, the elements in the intersection may be modeled as follows, with the meanings of the tags described in Table I.

$$\text{Barrack Street} = \left\{ \begin{array}{rl} (\text{id}, & 1), \\ (\text{type}, & \text{road}), \\ (\text{name}, & \text{Barrack Street}), \\ (\text{nodes}, & [\ldots, 6, 3, 7, \ldots]), \\ & \ldots \end{array} \right\}$$

$$\text{Barrack Square} = \left\{ \begin{array}{rl} (\text{id}, & 2), \\ (\text{type}, & \text{road}), \\ (\text{name}, & \text{Barrack Square}), \\ (\text{nodes}, & [\ldots, 4, 5]), \\ (\text{oneway}, & \text{true}), \\ & \ldots \end{array} \right\}$$

$$\text{Intersection} = \left\{ \begin{array}{rl} (\text{id}, & 3), \\ (\text{type}, & \text{node}), \\ (\text{subtype}, & \text{signal}), \\ (\text{lat}, & -31.95858), \\ (\text{lon}, & 115.85841), \\ (\text{paths}, & \{(5, 7), (6, 7)\}), \\ & \ldots \end{array} \right\}$$

$$\text{Barrack Square Node 1} = \left\{ \begin{array}{rl} (\text{id}, & 4), \\ (\text{type}, & \text{node}), \\ (\text{lat}, & -31.95863), \\ (\text{lon}, & 115.85829), \\ (\text{next}, & \{5\}), \\ & \ldots \end{array} \right\}$$

$$\text{Barrack Square Node 2} = \left\{ \begin{array}{rl} (\text{id}, & 5), \\ (\text{type}, & \text{node}), \\ (\text{lat}, & -31.95860), \\ (\text{lon}, & 115.85834), \\ (\text{next}, & \{3\}), \\ & \ldots \end{array} \right\}$$

TABLE I
TAGS AND WHAT THEY MEAN.

Tag	Required	Description
id	true	The unique identifier associated with a given feature. Can be used by other features to reference this feature.
type	true	The type of feature, used to group together features with similar semantics.
subtype	false	Subtype of a feature.
name	false	Display name of a feature.
nodes	false	Ordered list of nodes that contribute to this feature.
oneway	false	Whether a road is one way or not.
lat	false	The latitude of the feature.
lon	false	The longitude of the feature.
paths	false	Set of paths through an intersection, given as pairs of the form (node in, node out).
next	false	The set of nodes that can be traveled to after this one.

$$\text{Barrack Street Node 1} = \left\{ \begin{array}{rl} (\text{id}, & 6), \\ (\text{type}, & \text{node}), \\ (\text{lat}, & -31.95861), \\ (\text{lon}, & 115.85839), \\ (\text{next}, & \{3\}), \\ & \ldots \end{array} \right\}$$

$$\text{Barrack Street Node 2} = \left\{ \begin{array}{rl} (\text{id}, & 7), \\ (\text{type}, & \text{node}), \\ (\text{lat}, & -31.95851), \\ (\text{lon}, & 115.85840), \\ (\text{next}, & \{8\}), \\ & \ldots \end{array} \right\}$$

More formally, we represent a feature F as a set of ordered pairs: $F = \{(f, v) \mid f \in \mathcal{F}\}$, where v is some value associated with tag f and all features of interest have a unified schema in the form of a set of tags, $\mathcal{F}$. For those features that do not contain any value for a given tag we associate $\varnothing$ to that tag.

Having modeled features allows us to model a map, $\mathcal{M}$. If the universe of all possible features is the powerset $\mathbb{U} = P(F)$, then any map is $\mathcal{M} \subset \mathbb{U}$, i.e., we represent a map as a set of features, where each feature is itself a set of key-value pairs where the keys are drawn from $\mathcal{F}$. For convenience we place the additional constraint that there must be a unique identifier, id, for each feature associated with the tag. We will also adopt code like notation and say that for a given feature F the value of some tag x can be referenced as $F.x$.

We model a map $\mathcal{M}$ as a matrix of dimension $|\mathcal{M}| \times |\mathcal{F}|$, where each row represents a feature in the map and each column represents a tag $f_i \in \mathcal{F}$. For convenience we will refer to the tags as a vector $\vec{f} = [f_0, f_1, \ldots, f_{k-1}]$. Then each entry (i, j) in the matrix is the value $F_i.f_j$, where F_i is the feature represented by row i.

The reader will note that our modeling approach is incredibly generic, allowing us to represent arbitrarily complex map elements, with infinite possible tags. This is by design, allowing us to add new elements to our maps as we enhance the Apple Maps product with new—yet unbeknownst to us—feature types. In practice, however, there is small set of tags that is actually used and is useful to the map. For our example above (for only those features explicitly called out) we have: $\vec{f} = [id, type, subtype, name, nodes, oneway, lat, lon, paths, next]$. Hence, our small subset of the example map is represented as the matrix shown in Figure 2.

B. Modeling Data Checks

As mentioned earlier, our model of a map is overtly flexible so as to allow all types of real-world map entities to be represented. In practice, additional constraints must be applied to model only *sensible* worlds. For example, in our world, buildings all sit on the ground. While you could have a map with buildings that float, we can be pretty confident that this doesn't reflect the real world. We call this set of rules that we expected entities to follow our *data correctness checks*. Each check is designed to enforce some specific rules about how features behave or relate to each other in the real world. Such a check can be represented as a boolean function $t : \mathbb{U} \rightarrow \{true, false\}$ where the function returns *false* if the rule is not met. For example to validate that an intersection has at least one path through it, we can verify this with the following check, $t_{intersection}$:

$$\begin{array}{|l} t_{intersection}(M) = \textit{false} \text{ iff} \\ \exists F_i \in M \text{ such that} \\ node \in F_i.type \wedge signal \in F_i.subtype \wedge |F_i.paths| < 1 \end{array}$$

that asserts for each feature, if it is an intersection (type $node$, subtype $signal$) then there must be a non empty *paths* tag for that feature. If we instantiate this check to our example map, we see that it succeeds (returns $true$ by default) because our intersection does indeed have paths through it (two in this case) and no other features in the map have the *signal* subtype.

C. Modeling Data Check Coverage

We see that for the above example there are a lot of unused fields, and this is true in general as well. For each check we only require a subset of the possible tags for the feature. We define the set D_j to mean the *dependencies* of the check t_j. The set D_j reflects the tags that are *covered* by t_j in order for it to produce a result. For example, our $t_{intersection}$ will return the same results on the above map if we removed all tags except for *type*, *subtype* and *paths*; hence, we say that $t_{intersection}$ *covers type*, *subtype*, and *paths*.

Our notion of check coverage is analogous to that of code coverage from software testing. In testing, we would use code coverage tools to identify that $t_{intersection}$ covers these tags. Typically this is determined by instrumenting the code to record when a given code artifact (line, branch, etc.) is executed. To do this for our checks, we consider an imaginary "program" which contains code representations of all the features in our map as data objects; with accessor methods for each of their tags. Our checks are just code that runs

$$\begin{pmatrix} 1 & \text{road} & \varnothing & \text{B. St.} & [\ldots,6,3,7,\ldots] & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ 2 & \text{road} & \varnothing & \text{B. Sq.} & [\ldots,4,5] & true & \varnothing & \varnothing & \varnothing & \varnothing \\ 3 & \text{n} & \text{signal} & \varnothing & \varnothing & \varnothing & -31.95858 & 115.85841 & \{(5,7),(6,7)\} & \varnothing \\ 4 & \text{n} & \varnothing & \varnothing & \varnothing & \varnothing & -31.95863 & 115.85829 & \varnothing & \{5\} \\ 5 & \text{n} & \varnothing & \varnothing & \varnothing & \varnothing & -31.95860 & 115.85834 & \varnothing & \{3\} \\ 6 & \text{n} & \varnothing & \varnothing & \varnothing & \varnothing & -31.95861 & 115.85839 & \varnothing & \{3\} \\ 7 & \text{n} & \varnothing & \varnothing & \varnothing & \varnothing & -31.95851 & 115.85840 & \varnothing & \{8\} \end{pmatrix}$$

Fig. 2. Formal Matrix Representation of Our Small Map. Street names have been abbreviated.

against this program and returns a result. If we instrument this program and run a check t_j on it, we can record which of the accessor methods are invoked by t_j and assign the associated tags to the set D_j. Note that because many checks will have *short circuit* conditions (for example $t_{intersection}$ short circuits on tags: *type* and *subtype*) we also need the condition that our imaginary program represents sufficient data that the checks themselves will have complete code coverage. In practice, this is how we determine D_j; because the size of the map is very large, and checks are written to address features already in the map, it is guaranteed that the code coverage approach will fully capture all the tags the check inspects.

We are really interested in the subset of tags that may cause a given check to fail. This is strictly a subset of what we get from check coverage (since if a tag is not covered it can't possibly influence the result). Hence, we redefine D_j mathematically in terms of the ability of a check to detect issues with the map. We say that a tag f is included in D_j if and only if there exists some pair of maps $\mathcal{M} \subset \mathbb{U}$, $\mathcal{M}' \subset \mathbb{U}$, with features $F \in \mathcal{M}$, $F' \in \mathcal{M}'$ and there exists some value, v such that $\mathcal{M}' = (\mathcal{M} \setminus \{F\}) \cup \{F'\})$ and $F \triangle F' = \{(f, v)\}$ where $t_j(\mathcal{M}) = true$ and $t_j(\mathcal{M}') =$ *false*. By this stricter definition we can assert that for a given check, t_j, if there were no changes to tags in D_j for a given map, then the result of t_j *cannot* change.

D. Modeling Map Modifications

Intuition tells us that the changes that are made to the map should provide some insight into what checks have a chance of failing. For example if one changes only water bodies, then it is likely that checks that focus on water bodies will find issues; however, checks that focus on shopping malls are unlikely to reveal any issues. As such we seek to model the changes to a map in a way that is easy to work with. Consider two versions of a feature F and F' where F' represents an updated version of F. Formally we can say the changed tags C are defined as $C = \{f \mid (f, v) \in (F \triangle F')\}$, i.e., the tags that are in the symmetric difference of the two versions of the features.

In practice we change more than one feature at a time. Let us consider a set of changes that are applied to a map, $\mathcal{M}$, represented as matrix M, in order to generate the new map, $\mathcal{M}'$, represented as matrix M'. We can represent all the changed fields as a matrix $\mathcal{C}$ of width $|\mathcal{F}|$ whose rows are the C_i (as defined above) formed into vectors based on the indices of the tags in $\vec{f}$. Since our definition of C requires two versions of a feature to be computed we need to adjust our maps to handle added or removed features. We do this by adding empty rows to M' where features were deleted, and adding empty rows to M for any new features added. Note that these empty rows don't actually propagate to the map, they just help us with our calculation of $\mathcal{C}$. Then we can define $\mathcal{C}$ by the elementwise comparison of the two matrices:

$$\mathcal{C}_{i,j} = \begin{cases} 1 & \text{if } f_j \in C_i, \\ 0 & \text{otherwise} \end{cases}$$

where C_i is the changed tags of the feature represented by row i, and f_j is the jth element of the tags vector $\vec{f}$.

E. Defining Data Check Selection Safety

We are ready to define "check selection safety." Intuitively, a check selection algorithm is deemed safe if it selects, from the original set of checks, every check that may reveal a fault in the modified map. More formally, for a given suite of checks T that run on a map M, we define a set of checks, $\mathcal{T} \subseteq T$, to be *safe* if $\mathcal{T}$ contains every check in T that can expose faults in a modified version of the map M.

Since we want to be able to run a whole suite of checks at once we need a way of describing all the dependencies for our suite. Let the matrix $\mathcal{D}$ describe the dependencies that are required by all our checks where each check is assigned a row and the tags are assigned the columns corresponding with their location in $\vec{f}$. We define $\mathcal{D}$ by its entries as follows:

$$\mathcal{D}_{i,j} = \begin{cases} 1 & \text{if } f_j \in D_i, \\ 0 & \text{otherwise} \end{cases}$$

where D_i is the dependencies for check t_i as defined above, and f_j is the jth element of the tags vector $\vec{f}$.

For changes C and check dependencies D_j we observe $(C_i \cap D_j = \varnothing) \Rightarrow (t_j(F_i) = t_j(F_i'))$. This follows directly from the definitions as D_j represents the set of tags to which changes have the *possibility* of causing a change in the results. This means that if we have already computed $t_i(F)$, and $C \cap D_i = \varnothing$, then we already know the result of $t_i(F')$.

With the two matrices $\mathcal{C}$ and $\mathcal{D}$ defined we can extend this observation as follows:

$$\begin{aligned} &\text{let } J \text{ be a matrix of 1's of dimension } 1 \times |M'| \\ &\text{let } \sqcup = J \times \mathcal{C} \times \mathcal{D} \\ &= [\sqcup_0, \sqcup_1, \ldots, \sqcup_{k-1}] \\ &\text{then } \sqcup_i = 0 \\ &\Rightarrow \forall i, t_i(F_i) = t_i(F_i'), \end{aligned}$$

where F_i and F'_i are the features associated with index i in M and M' respectively.

At first glance it may seem like this misses a certain type of check, namely those that inspect the relationships between features (for example, if $node_3$ refers to $node_4$, then $node_4$ must exist). This problem is actually handled by our earlier adjustment to deal with added and deleted fields. Because a feature must contain appropriate tags with which it can be referenced (typically via id) and any check that tests a relation *must* have a dependency on that tag, then by replacing the row for that deleted feature with an empty one, C_i for that row will also contain tag being referenced.

The matrix $\sqcup$ is actually a partial representation for our *safe* set for the changes from $\mathcal{M}$ to $\mathcal{M}'$. To complete the safe set we simply include those checks marked in $\sqcup$ for which $t_i(\mathcal{M}) = \textit{false}$. We know that for map $\mathcal{M}'$ running only those checks in the *safe* set will expose all issues that running the full suite of checks could expose. We can in fact opt to run only the subset of these represented in $\sqcup$ and simply join them with the results from our previous run, overriding only the old values for the new check runs. This does mean that for each change under consideration we need to compute $\sqcup$, however the cost of doing so is small. This is because the changes themselves are already submitted to our system as a *diff* so we simply need to convert that *diff* to a matrix and perform our multiplication. As for computing $\mathcal{D}$, it need only be recalculated when checks change which is a *much* lower frequency than data changes.

F. Leveraging the Data Model for ML

Formalizing the above model of our data and checks gives us big gains by allowing for an easy method—using static analysis of the checks—to calculate the *safe* set of checks; however, it also opens the doors to more in depth data processing. A key example of where having such a model comes in handy is the realm of building ML models to make predictions about the results of checks. While such predictions are no longer *safe*, they do offer significant potential benefits in prioritizing checks to run. Most ML models require well-structured data in order to operate. By simply joining our change matrices $\mathcal{C}$, our check dependency matrix $\mathcal{D}$, and any additional metadata associated with the change, we get a very good structure from which to build ML models. Using this data we seek to identify causes for failures, isolate root causes of failures, and rank checks in order to minimize the time it takes to find a failure.

IV. Evaluating Check Recommendation & Ranking in Production

We now empirically evaluate the effectiveness of safe check recommendation, gaps in the coverage of the checks we currently run, and performance of our machine learning implementation of the recommender system.

A. Research Questions

Specifically, we address the following research questions:

RQ1: How *effective* is the safe subset of checks with respect to the hand-curated static set?

We evaluate effectiveness via two metrics: (1) *efficiency*, which we measure as the ratio between the sizes of the hand-curated static set of checks and the recommended subset; and (2) *recall*, which we measure as the ratio between the number of recommended checks that had a *fail* outcome and the total number of checks in the hand-curated set that had a *fail* outcome. Because the hand-curated and recommended sets vary per merge request (discussed later), we compute these metrics on a per request basis.

RQ2: Does our new concept of check coverage and safe selection help to identify gaps in the current manual check selection (using check groups) approach?

To address this question, we will use the *universal* set of candidate checks, compute a safe subset for this universal set, and determine the number of checks that were missed by the current manual selection process. We do this on a per request basis as well.

RQ3: How accurate is our machine learning model in ranking the checks by likelihood to fail?

We measure accuracy via several standard performance metrics of the machine learning model, namely:

- *recall* - the percentage of the failed checks that were predicted by the model;
- *precision* - the accuracy of the model for predicted check failures;
- *FPR* - the false positive rate which is the number of checks incorrectly predicted as a failure divided by the number of checks that passed; and
- *AUC* - area under the receiver operating characteristic curve which is an aggregate measure of overall classifier performance.

RQ4: Can our machine learning results provide actionable insights into which edits are causing regressions?

We examine the decision-making parts of our learned models to identify their most discerning signals and use these to determine which contributed to the most recent check outcome transitions from *pass* to *fail*.

B. Study Process

In order for the reader to better appreciate the context in which we executed this empirical study, we briefly describe the maps data curation process. Our maps data is created/edited by a large team of editors who make changes that eventually get merged into a *master* data repository. The editors make changes on small geographic regions in separate *branches*. When their edits are complete they attempt to merge these back into the master repository. Prior to the merge, checks are executed as part of a validation job (*pre-merge*) to ensure that the merging of the changes do not cause regressions. The large number of edits, and frequency of merges make this pre-merge stage extremely critical and expensive, which is why our approach operates on this stage of the pipeline.

The checks are statically grouped into *check groups* and the groups are typically manually pre-selected by quality

assurance staff based on the type of project on which the editor is working. The actual checks (we call these the hand-curated static set of checks) executed from the chosen check groups typically range in number across validation jobs.

For our study, we examined 465,500 pre-merge requests over several months. Of these, 450,000 requests passed all checks and were successfully merged, and 15,500 had at least one check failure meaning that the merge was aborted and sent back to the editor to fix before they attempted to merge again. Detailed data on both the changes that were made and the reasons for check failures were logged. 21% of the checks failed at least once with the remainder never failing. The execution results for all but one check were heavily skewed towards pass even when only considering jobs where there was at least one check failure. In this set of results, one check had a 52% failure rate while all the remaining checks had a failure rate of less than 10%.

For clarity of presentation, we will report results of the 15,500 merge requests that had at least one failed check. We consider this data to be representative of the data from all the validation jobs because only a very small percentage of checks fail in each validation job.

We have implemented and put into production all parts of our check recommender. Our study process involves computing the diff (changes to the data being merged) and using it to inform safe check selection, which we implemented using static analysis of the checks. We rank the checks in this safe subset using a gradient boosted decision tree model and deep neural networks.

C. RQ1: Safe Check Selection

As mentioned earlier, we use efficiency and recall to address **RQ1**. Efficiency is important as there is a cost to making a recommendation and we would like to ensure that the gains realized by eliminating checks is significantly greater than the cost of making a recommendation. Recall is important as this method attempts to maximize the number of failed checks that are recommended, i.e., ideally, all of the failed checks, to ensure that no regressions are merged into the master repository.

We first present the results for efficiency. The static analysis based check selection recommended 35% of the all the checks that were executed. That is, on average, 65% of checks were eliminated for each validation job. When considering all 465,500 validation jobs that were executed in the time period, a significant number of check executions were eliminated. The recommendation took on average 500 milliseconds which is significantly less than the average validation job duration of 35 seconds.

Figure 3 shows the fraction of checks that were selected and the fraction of checks that were ignored. The sum of those two are the fraction of checks that were actually executed. The y-axis is normalized by all the checks that are available to be executed. The figure shows that static analysis based test selection reduces the percentage of checks executed by 65%.

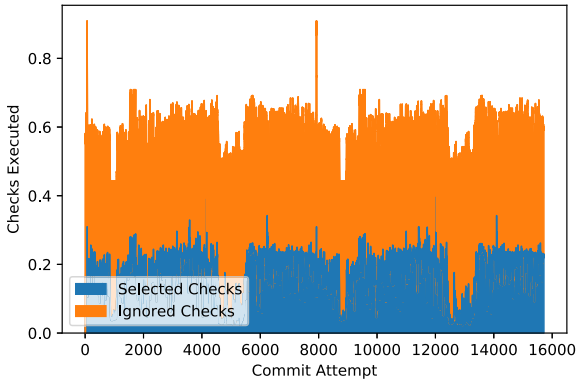


Fig. 3. Fraction of checks selected and the fraction of checks ignored by static analysis based check recommendation. The sum of which is the total fraction of checks executed in each commit attempt. The y-axis is normalized by all the checks that are available to be executed.

Furthermore, analysis showed that the recall rate, i.e., the percentage of failed checks that were recommended for execution, was at 97.5%. The majority of the 2.5% of checks that failed but were not recommended were mostly due to: 1) inaccuracies in the check metadata which identifies the object types and fields they are validating, and 2) pre-existing errors that were already in existence in the repository when it was branched and were not due to edits made by the editors. The 2.5% of checks that failed but weren't recommended was a small number of checks. Recall was at 100% when only checks with accurate metadata were considered for execution and repositories with pre-existing errors were ignored. If checks with inaccurate metadata need to be executed they can be recommended for validation jobs regardless of the changes present in the repository with minimal impact to efficiency.

Figure 4 shows that the fraction of recommended checks plotted against the fraction of fields that were edited. Both are plotted normalized by the maximum value of their respective axes. The relationship between recommend checks and edited fields follows intuition that as more edits are made, more checks will need to be executed to prevent regressions. This relationship also indicates that most checks have a narrow scope and are targeted at detecting specific regressions.

The results for **RQ1** demonstrate that static analysis based check selection to recommend a safe set of checks to be executed produces high recall with significantly improved efficiency.

D. RQ2: Improving Check Coverage

We now present results showing that our safe check recommendation can improve check coverage beyond the current manual, static check grouping. To clarify, the *universal* set of available checks is much larger than the hand-curated static set that is executed per merge request. However, this universal set is never run per merge because of time and resource constraints. For **RQ2**, we applied static analysis based

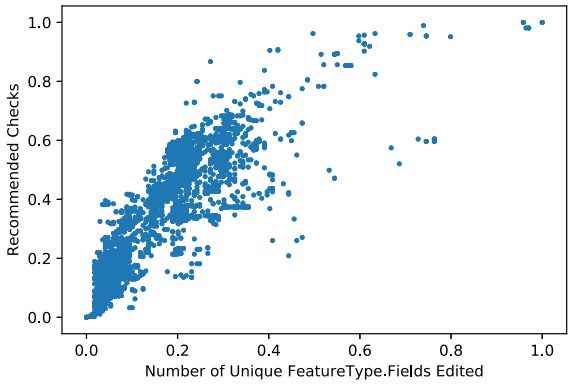


Fig. 4. Fraction of recommended checks plotted against the fraction of fields that were edited in each repository. Both are plotted normalized by the maximum value of their respective axes.

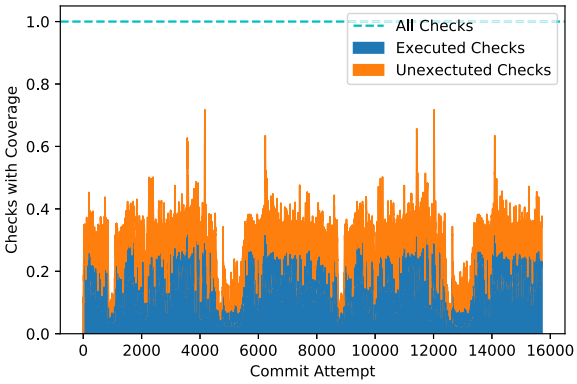


Fig. 5. Improvement in check coverage when considering the full set of checks for safe selection. The *y*-axis is normalized by all the checks that are available to be executed.

check recommendation to the universal set of checks to see if the manual check grouping had inadvertently excluded some applicable checks.

Figure 5 shows the improvement to coverage when considering the universal set of checks for static analysis based recommendation. The dotted cyan line shows the universal check set that are available for selection. The blue lines show the subset of checks executed from the set that was typically chosen for each validation job and the orange lines show the number of checks that were executed when considering the universal set of checks. All values are presented normalized by number of checks in the universal set being considered. This figure shows that it is possible to both increase coverage while at the same time achieving gains in efficiency. The total number of recommended checks is smaller than the size of the set that was typically executed for each validation job.

The results of **RQ2** show that static analysis based recommendation can increase check coverage, help identify gaps in current manual check grouping, while still achieving gains in efficiency.

E. RQ3: Machine Learning Based Ranking Results

Machine learning models were built using deep neural networks and gradient boosted decision trees to predict the likelihood of check failure for a given set of map data changes made by an editor. These two modeling methods were chosen because of their ability to express complex relationships and because they have been proven to work well across many industry applications [31], [32]. The likelihood of failure is used to prioritize the checks for ranking and for selection. Pass and fail samples were collected for each check from the 15,500 validation jobs described earlier. For each check the pass and fail samples were split into a training set consisting of 80% of the samples and a test set consisting of 20% of the samples. Machine learning models were built for each check that had both pass (positive) and fail (negative) samples in the training data. In total, 21% of the checks had both deep neural network and gradient boosted decision tree models built for them.

The deep neural networks were feed-forward networks with 3 densely connected hidden layers consisting of 64, 32 and 16 nodes respectively. All the nodes had ReLU activation functions and were built using the TensorFlow machine learning framework [31]. The gradient boosted decision trees were built using the Turi machine learning framework which leverages the XGBoost library for training [32]. There were 430 input feature columns used to build each model. Each column represented a unique field that was changed by an editor. The performance of the models built using the deep neural networks and the gradient boosted decision trees were almost identical across all measured metrics. We present the performance results for the gradient boosted decision trees. The decision tree models have an advantage in that they are interpretable and we can gain insights from their structure which we will present and discuss for **RQ4**. We conjecture, drawing from past experience, that the deep neural networks will have significantly better performance than the decision tree models as the training data size increases substantially.

Table II shows the performance of the gradient boosted decision tree models built for the 10 checks with the highest number of fail samples in the training data. The total number of pass and fail samples available for each check is given in Columns 2 and 3 of the table. The testing data set was a random sample of 20% of this data and was not used in model training. The model performance was evaluated with the testing data set using metrics on recall, precision, false positive rate (FPR) and area under the ROC curve (AUC). The decision threshold of the models were adjusted so that recall was at least 0.97 or 97%. To interpret the table, for example, the first row is the results for Check ID: C01. A total of 2,293 pass and 2,095 fail samples were collected for this check. The models for this check were trained using a random selection of 80% of those samples, or 1,834 pass and 1,676 fail samples. The models were evaluated using the remaining 20% of samples,

TABLE II

PERFORMANCE OF THE DECISION TREE MODELS FOR THE 10 CHECKS WITH THE HIGHEST NUMBER OF FAILURES. CHECK ID IS AN INTERNAL IDENTIFIER FOR CHECKS, PASS SAMPLES IS THE NUMBER OF CHECK PASS SAMPLES IN THE COLLECTED DATA SET, FAIL SAMPLES IS THE NUMBER OF CHECK FAILURE SAMPLES IN THE COLLECTED DATA SET, RECALL, PRECISION, FALSE POSITIVE RATE (FPR), AND AREA UNDER ROC CURVE (AUC) ARE COMPUTED FROM THE TEST SET.

Check ID	Pass Samples	Fail Samples	Recall	Precision	FPR	AUC
C01	2293	2095	0.972	0.918	0.080	0.974
C02	10948	404	0.971	0.545	0.026	0.994
C03	8416	271	0.989	0.065	0.499	0.953
C04	10492	250	0.974	0.064	0.351	0.964
C05	5994	196	0.982	0.072	0.394	0.959
C06	10190	177	0.981	0.342	0.032	0.992
C07	3061	175	0.981	0.323	0.121	0.975
C08	435	171	0.981	0.531	0.346	0.881
C09	1278	150	0.980	0.653	0.069	0.986
C10	5971	147	0.978	0.276	0.066	0.983

TABLE III

TOP-10 MOST IMPORTANT INPUT FEATURES ORDERED BY THE SUM OF OCCURRENCES OF THAT INPUT FEATURE AS A BRANCHING NODE FOR THE GRADIENT BOOSTED DECISION TREE MODEL FOR A CHECK ON SUB-POSTAL CODES. THE FIRST COLUMN IS THE PATH TO A SPECIFIC FIELD OF A FEATURE IN THE BASE-MAP. THE SECOND COLUMN IS THE THE SUM OF OCCURRENCES OF THAT INPUT FEATURE AS A BRANCHING NODE FOR THE GRADIENT BOOSTED DECISION TREE MODEL AND IS A PROXY FOR THE IMPORTANCE OF THAT FEATURE IN EVALUATING THE OUTPUT OF THE MODEL.

Input Feature	Branching Nodes
address_component.name.code_type	26
address_component.is_manually_curated	17
history.source_key	14
address_component.name.unparsed_name	11
address_component.territory_type	10
CAN	10
GBR	10
address_point.radius_of_influence	9
address_component.address_component_type	8
address_component.name.size	7

or 459 pass and 419 fail samples. The recall indicates that of the 419 fail samples the model was able to correctly predict a fail for 97.2% of those or 407 fail samples. The precision indicates that the model was correct in 91.8% of cases when it predicted a failure. The FPR indicates that 8.0% of the pass samples or 183 out of 2293 samples were incorrectly labeled as a fail. The AUC of 0.974 indicates a very high classifier performance overall. Similarly all models showed very high performance in the AUC metric which allows for relatively accurate check ranking. For models with low precision, such as those for C03, C04 and C05, the FPR increases significantly resulting in a significant number of additional check executions that do not fail. Although the AUC is high the FPR rate has increased since the models have been tuned to have a recall such that at least 97% of all failures are recommended.

The results for **RQ3** demonstrate that the models built using machine learning were successful at ranking checks by likelihood of failure for a given set of changes. The metrics in Table II show that we could predict true failures with low false positive rates.

F. RQ4: Actionable Insights Derived from ML Models

The decision tree models built for predicting check pass and failure also provide other valuable insights into the data. One of these is feature importance. Feature importance is a measure of the importance of an input feature to the model in predicting the output. The measure of importance of an input feature is determined by the sum of all the occurrences of that feature as a branching node in all trees that are built as part of the ensemble of decision trees for the gradient boosted decision tree models.

Table III shows the top-10 most important input features ordered by the sum of occurrences of that input feature as a branching node for the gradient boosted decision tree model for a specific check on sub-postal codes. The sub-postal code is the additional numbers or letters at the end of a standard postal code in some countries. In the United States for an example postal code of 94301-1445, the sub-postal code is the 4-digits at the end, i.e, 1445. Each row in the table represents a path to a unique field which is a member of a feature. A large collection of features comprise the base-map. These fields contain properties of the features and are edited by the editors. Edits to these fields are used to train the models. We can therefore use the importance of these fields in the models to understand which edits are causing checks to fail.

Most of the important features are related to standard fields within the base-map, however, two input features that have provided valuable insight into the how the editors edit data are the input features denoting that an edit has been made to a sub-postal code in Canada or Great Britain, denoted by the country codes CAN and GBR. Canada and Great Britain have sub-postal codes that are of a different format to those in the United States. The trained model shows a strong correlation between edits to the sub-postal code in Canada and Great Britain and likelihood of a pass or fail on the sub-postal code check, indicating that this feature played an especially important role in the check's failure, and hence, warrants closer examination. This information was conveyed back to check writers and editors, who confirmed sub-postal codes to be causes of failures in CAN and GBR.

The results for **RQ4** show that the models trained for machine learning can quickly provide actionable insights into those edits that are causing regressions by analyzing input feature importance.

V. CONCLUSION AND FUTURE DIRECTIONS

A major challenge in developing any large-scale industry application that is used by billions of end users is its validation, which becomes even more complex if the correctness of the application depends on a large amount of underlying data. Apple Maps is one such application; we spend significant resources validating the constantly changing maps data to provide the best possible experience to our end users.

We presented the problem of validating the underlying data used in Apple Maps and described the current hand-curated static style of selecting data validation checks. We

presented a new concept of check coverage and used it to define *check selection safety*. We have implemented a safe check recommender that combines static analysis of the data checks with a representation of the latest data changes, and a ranking approach using machine learning.

Our study on several months of production data demonstrated that we are able to significantly reduce the number of check executions both through static analysis based selection, and ranking based on machine learned models. We identified problems in the meta-data of several checks and have been able to influence future check development.

Work is ongoing to collect more training data to improve the performance of the ranking models and to use the recommender system to expand the coverage of checks.

ACKNOWLEDGMENT

We thank Benoit Dupin for helping to define the vision and scope of this project; Stefan Bengtsson, Dinkar Gupta, Krishna Kadali, Ravi Valmikam, and Ivan Ukolov for providing insights into the maps data validation process; Ajay Vaddadi, Rujuta Ranade, Maitri Verma, and Praveen Krishnan for providing infrastructure support; and Alex Braunstein for his constant support of our work.

REFERENCES

[1] J. S. Longo and C. B. Medeiros, "Providing multi-scale consistency for multi-scale geospatial data," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013, p. 8.

[2] D. Sheeren, S. Mustière, and J.-D. Zucker, "A data-mining approach for assessing consistency between multiple representations in spatial databases," *International Journal of Geographical Information Science*, vol. 23, no. 8, pp. 961–992, 2009.

[3] E. Xavier, F. J. Ariza-López, and M. A. Urena-Camara, "A survey of measures and methods for matching geospatial vector datasets," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, p. 39, 2016.

[4] M. F. Goodchild, "Geographical information science," *International journal of geographical information systems*, vol. 6, no. 1, pp. 31–45, 1992.

[5] I. ISO, "19157: 2013: Geographic information—data quality," *ISO-standard and Swedish SIS standard*, 2013.

[6] H. Senaratne, A. Mobasheri, A. L. Ali, C. Capineri, and M. Haklay, "A review of volunteered geographic information quality assessment methods," *International Journal of Geographical Information Science*, vol. 31, no. 1, pp. 139–167, 2017.

[7] A. L. Ali and F. Schmid, "Data quality assurance for volunteered geographic information," in *International Conference on Geographic Information Science*. Springer, 2014, pp. 126–141.

[8] R. A. Haraty, N. Mansour, and B. A. Daou, "Regression test selection for database applications," in *Advanced Topics in Database Research, Volume 3*. IGI Global, 2004, pp. 141–165.

[9] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso, "Regression testing in the presence of non-code changes," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 21–30.

[10] D. Willmor and S. M. Embury, "A safe regression test selection technique for database-driven applications," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 421–430.

[11] R. H. Rosero, O. S. Gómez, and G. Rodríguez, "Regression testing of database applications under an incremental software development setting," *IEEE Access*, vol. 5, pp. 18 419–18 428, 2017.

[12] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

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

[14] J.-M. Nicolas, "Logic for improving integrity checking in relational data bases," *Acta Informatica*, vol. 18, no. 3, pp. 227–253, 1982.

[15] H. Christiansen and D. Martinenghi, "On simplification of database integrity constraints," *Fundamenta Informaticae*, vol. 71, no. 4, pp. 371–417, 2006.

[16] D. M. Strong, Y. W. Lee, and R. Y. Wang, "Data quality in context," *Communications of the ACM*, vol. 40, no. 5, pp. 103–110, 1997.

[17] C. Batini, C. Cappiello, C. Francalanci, and A. Maurino, "Methodologies for data quality assessment and improvement," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 16, 2009.

[18] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, "Improving data quality: Consistency and accuracy," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 315–326.

[19] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang, "Data cleaning: Overview and emerging challenges," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 2201–2206.

[20] S. Kolb, S. Paramonov, T. Guns, and L. De Raedt, "Learning constraints in spreadsheets and tabular data," *Machine Learning*, vol. 106, no. 9-10, pp. 1441–1468, 2017.

[21] C. Mayfield, J. Neville, and S. Prabhakar, "Eracer: a database approach for statistical inference and data cleaning," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 75–86.

[22] M. Gupta, J. Gao, C. C. Aggarwal, and J. Han, "Outlier detection for temporal data: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 9, pp. 2250–2267, 2014.

[23] S. Wang, J. Nam, and L. Tan, "Qtep: quality-aware test case prioritization," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 523–534.

[24] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*. IEEE, 2016, pp. 266–277.

[25] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[26] R. Kazmi, D. N. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: a systematic literature review," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, p. 29, 2017.

[27] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 12–22.

[28] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 975–980.

[29] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 297–308.

[30] M. Azizi and H. Do, "A collaborative filtering recommender system for test case prioritization in web applications," *arXiv preprint arXiv:1801.06605*, 2018.

[31] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[32] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794.