Automatically Authoring Regression Tests for Machine-Learning Based Systems

Junjie Zhu, Teng Long, Atif Memon
Apple Inc., Cupertino, USA
{jason.zhu, teng.long, atif.memon}@apple.com

***Abstract*—Two key design characteristics of machine learning (ML) systems—their ever-improving nature, and learning-based emergent functional behavior—create a moving target, posing new challenges for authoring/maintaining functional regression tests. We identify four specific challenges and address them by developing a new general methodology to automatically author and maintain tests. In particular, we use the volume of production data to periodically refresh our large corpus of test inputs and expected outputs; we use perturbation of the data to obtain coverage-adequate tests; and we use clustering to help identify patterns of failures that are indicative of software bugs. We demonstrate our methodology on an ML-based context-aware Speller. Our coverage-adequate, approx. 1 million regression test cases, automatically authored and maintained for Speller (1) are virtually maintenance free, (2) detect a higher number of Speller failures than previous manually-curated tests, (3) have better coverage of previously unknown functional boundaries of the ML component, and (4) lend themselves to automatic failure triaging by clustering and prioritizing subcategories of tests with over-represented failures. We identify several systematic failure patterns which were due to previously undetected bugs in the Speller, e.g., (1) when the user misses the first letter in a short word, and (2) when the user mistakenly inserts a character in the last token of an address; these have since been fixed.**

***Keywords*-ML testing, ML-based testing, spelling correction**

I. Introduction

End-to-end regression testing of Machine Learning (ML) software has disrupted the way we think about functional testing [1], [2]. Traditional functional tests are of the form *(input, expected_output, assertion())*, where *input* is supplied to the software under test (SUT), and the *test oracle* (*expected_output* and *assertion()*) verifies whether the SUT functioned as expected [3]. Testers strive to develop a test suite that provides adequate *coverage* of software features [4].

Regression testing of ML systems casts aside the 3 traditional tenets of functional testing: *input*, *expected_output*, and *coverage* in multiple ways. First, the input spaces of ML-based systems are extremely large [5] (think about all the situations to which an autonomous vehicle must react), which is why these systems are, by design, optimized for their most common inputs. Indeed, they may not always return correct outputs for all uncommon inputs. Developers may not even know all the uncommon/corner cases [6] at design time, neither would the testers during in-house test development [7]. The software's eventual functional behavior is not pre-defined; rather it emerges as it learns and evolves.

Second, imperfect understanding of the input space upsets the traditional role of functional testing, which is to use functional boundaries/partitions and corner cases to ensure that the system behaves as intended within—and at the boundaries of—each partition. Consequently, test authors are unable to determine whether they have an adequate test suite that covers all functional boundaries. All their hard-coded *inputs* in a test suite may be distributed over an initial guesstimated set of partitions but may, over time, end up in quite another set, causing the inputs to become less important or even irrelavent. Moreover, because much of the ML decision logic is typically encoded mathematically, e.g., in a deep neural network or a logistic regression model, there is no control-flow-graph, and hence traditional coverage also does not directly apply [8].

Third, ML systems, by their very nature, are designed to better serve the most prominent inputs and constantly improve their outputs over time by learning from new training data [7]. A traditional test oracle will quickly become obsolete as its hard-coded *expected_output/assertion()* turns stale with respect to the software's new improved output.

Finally, another distinction in regression testing of ML vs. conventional systems is that individual test failures for ML systems may not be indicative of a bug. Recall that ML systems are optimized for certain classes of common inputs – they may not work for uncommon inputs; hence failures on such inputs may be perfectly acceptable. Instead, of interest to the ML-system developer are systematic test failures as well as patterns of failures that assist in software/model debugging. This shift creates new challenges for test authors, who must now create a large number of tests to reveal such patterns. Moreover, test failure triage is not always useful when looking at individual isolated failures; rather, groups of failing tests need to be examined to provide a more holistic picture of what went wrong with the ML software.

Other researchers have also recognized emerging challenges posed by ML systems [1], [2], [9] and proposed new solutions. In particular, metamorphic testing [10] and perturbation have been used to generate test suites [11]. White-box testing approaches [12], [13] have aimed to define test coverage based on the state of the neural networks. However, there is no one-size-fits-all methodology that will work for every ML system. On the other hand, Log Replay [14] is a practical general-purpose solution that is widely used for large-scale testing; however, it does not cover all corner cases, including ones that might not occur frequently but are critical for functional testing.

978-1-6654-3869-8/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE-SEIP52600.2021.00049

In this paper, we develop a new methodology aimed at functional regression testing for ML software, and apply it to a context-aware ML-based spelling checker/corrector (Speller). Our results show that the methodology can scale up the test suite to cover a large typo space, and, at the same time, reveal failure cases that can often be masked by other common misspelled inputs. Furthermore, the methodology can cover a multidimensional space with test cases automatically built upon constantly-changing production data. We show that these adaptive test suites can isolate the performance of the ML component from the end-to-end speller system, and keep up with both model and data changes. Instead of reporting individual test failures, we rely on featurizing misspell patterns and spectral clustering to automatically report subcategories of tests that contain higher proportions of defects. In particular, we make the following contributions:

- We keep up with the ML software's evolving input space by automatically revising our test suite using production data to obtain new test cases and delete obsolete ones.
- We learn a coverage-driven perturbation model to generalize existing cases in production data to enrich edge cases that are underrepresented in real training and test data.
- We resolve the obsolete oracle problem by using the relationship between the original data from production and its perturbed counterparts; we determine the expected output of a number of test cases where the consequent feedback is positive and indicates that the users received correct outputs.
- We automatically identify important failure classes by mining patterns of test cases using unsupervised learning and cluster the test cases to identify subgroups with high number of failure cases.

The next section describes our ML-based Speller service that we use as our study subject. We then present our new methodology using our study subject as an example in Section III. Section IV presents our evaluation results. In Section V, we summarize related work, and finally, in Section VI, we conclude with a discussion of ongoing and future work.

II. The ML-Based Speller

We now describe how the aforementioned challenges apply to testing a context-aware ML-based Speller. We highlight how its ML sub-component influences end-to-end behavior leading to stochastic and anecdotal failures (which we refer to as *ad hoc* test failures) that are inherently difficult to triage.

1) Context-Aware Speller: Our Speller takes two parameters as input: (1) a query string, such as an input to a search bar, and (2) contextual information, such as geographic location, device settings, locale, user preferences, etc. It outputs a *ranked* list of strings that each corresponds to a different suggested correction. Similar to typical spelling correction systems [15], the Speller relies on edit-distance-based algorithms to generate multiple correction candidates from the corpus. The Speller utilizes several filters and ranking algorithms to output a ranked list of spelling corrections, e.g., "*car ash*" may yield "*car wash*", "*car dash*".

The Speller we consider not only relies on commonly-used information, such as word frequencies, as is used in traditional language models [15], but also takes the contextual information to improve upon a generic spelling correction system. For instance, if a user is suggested multiple businesses that are spelled similarly, the *location*—a contextual variable—can prioritize the businesses that are in a town where the user is located. For instance, suppose there is a town where a new popular business called "*car dash*" appears. The top suggestion for the misspelled query "*car ash*" can flip from "*car wash*" to "*car dash*" because more users are visiting the town specifically to go to the popular business. An additional user preference (also a contextual variable) indicating that the user does not prefer lower ranked businesses might lead to a completely different, more relevant, result.

Our Speller has a large input space, spanning multiple dimensions. First, the space of query strings spans not only all possible valid English words, but numerous misspelled variants of each word. For instance, misspells of the string "*car wash*", e.g., "*car ash*", "*carwash*", "*car dash*", and etc., can grow exponentially with the alphabet size if we consider any possible misspell. Second, contextual variables such as location coordinates, locale settings, etc., themselves present a multi-dimensional space of inputs. The product of these two dimensions create innumerable possibilities, which make methods such as exhaustive testing impractical.

2) Functional Boundaries of Speller: A traditional implementation of the Speller would have separate logic branches that return different suggestions (e.g., business names) based on the contextual features (e.g., location coordinates), so it would suffice to just cover each branch during testing. However, with such an enormous input space, developers cannot hard-code all the possibilities, which is why we instead rely on ML models to automatically score and rank each suggestion. ML systems are frequently used for ranking context-based query suggestions because they can mine informative patterns from multivariate data sources. When multiple query strings are equally valid spelling corrections, the Speller relies on the ML system to leverage contextual features and order them.

The way the ML model scores each suggestion is fundamentally different from how traditional software is written. ML models rely on mathematical expressions that vary based on trained numerical parameters, so traditional test coverage that rely on logic branches are no longer applicable. Furthermore, because ML decision boundaries are not represented in the software code, we cannot easily determine when the expected values in the output changes due to the parameter updates. In this scenario, a fixed expected output of "*car wash*" will report a failure when "*car dash*" is returned, even though this is acceptable after the ML is updated based on new data points. This boundary behavior of the ML system can hardly be assessed by traditional testing systems because one can only observe changes in the model parameters instead of modifications in the code.

3) Systematic Defects in Spell Correction: From a bird's-eye view of millions of data points, failures that share similar

misspells are intrinsically more valuable than an individual failure. Take the former example of "*car ash*" and suppose that this data point causes a failure because its spelling correction does not match our expectations. If this is a stochastic behavior that changes the next time a model is updated, then it might not require any actionable fixes because each ML system is expected to always perform well on some data points and worse on others in a non-deterministic way, especially if the data points are close to the decision boundary [16]. Thus, this failure alone constitutes an ad hoc failure with a transient result that does not pinpoint defects in the ML model of the software.

On the other hand, we might simultaneously observe a set of failures with similar misspells, e.g., "*ballon*" (expected correction: "*balloon*"), "*seafod*" (expected correction: "*seafood*"), "*notebok*" (expected correction: "*notebook*"), and "*typhon*" (expected correction: "*typhoon*"). These data points collectively indicate something more informative: *the Speller fails more frequently on words with double-letter misspells.* This could be a real software issue where deletions in letters are not corrected, or a model training bias where there is insufficient training data with double-letter deletions. Yet, with only a single failure, we cannot easily come up with such hypotheses or prioritize what to investigate; we instead need a large number of related failures.

III. Methodology to Test ML-based Systems

We have developed a new methodology to address the challenges brought about by ML systems. The key intuition behind our methodology is to leverage the scale of production data to automatically author large numbers of coverage-adequate test cases whose Pass/Fail outcomes reveal systematic patterns that may be indicative of failures in the ML system. More specifically, as shown in Figure 1, we start by mining (A) large volumes of production data, which we then perturb using (B) a coverage-driven model-based test input curator that yields a large number of coverage-adequate test cases, with test inputs and expected outputs. These tests are then executed on the ML SUT, the actual output is obtained, and (C) an automated test oracle determines if the SUT passed or failed the test. Together with features of the inputs, test outcomes, and expected outputs, we use (D) clustering to determine which failures are related, in that all failures in a given cluster stem from a single bug. The results from clustering can also be used to improve the curator to generate more refined test suites along with oracles. We now break down our methodology workflow tailored to the Speller.

1) Production Data: Our methodology is centered around leveraging a massive amount of available production data (Figure 1A) to obtain a large number of test cases that can reveal patterns of failures in the ML software as opposed to a handful of failures that may not easily map to bugs in the ML system or its model. Hand crafting such a large number of tests is cost prohibitive.

For the Speller, production data takes the form of a query input string with contextual information and, among the user's

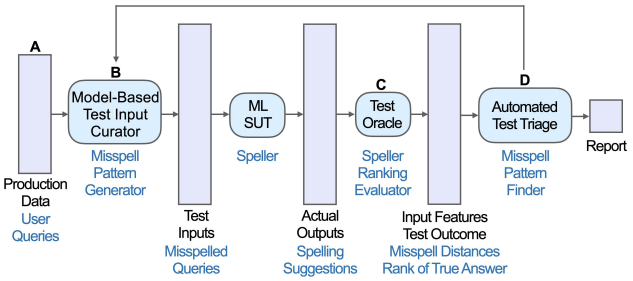


Fig. 1. Our testing methodology applied to the Speller. It (A) selects user queries (with and without misspells) from production data, (B) uses a model-based test curator to automatically generate test cases, executes them, (C) evaluates the outputs with test oracles, (D) and automatically triages failures to find defect classes that represent specific misspell patterns.

search result, an indicator of whether spelling correction was required, as well as 20 ranked spelling suggestions. Crucially, we only select queries where the user successfully interacted with some suggestion in the application. The suggested results each user selected have no misspells because the list of suggestions are generated from the corpus of valid words.

The selected queries are divided into two groups based on whether spelling corrections are required. For the queries that do not require spelling suggestions (determined by the previous Speller software instance), we assume that the original query was correctly spelled. Although it could be possible that a small number of inputs had misspells and were not corrected, yet some users still selected the uncorrected suggestions, we ensure such scenarios are filtered out based on our criteria that the user interacted with the suggestion extensively. For the queries that required spelling correction, we assume that the associated user inputs had some form of typo, since the users selected suggested spellings that were different from their original inputs.

A. Model-Based Test Input Curator

The raw production data is not useful in itself because it lacks the expected output needed for a test oracle; it also lacks a measure of test adequacy as all functional partitions may not be represented. This is why we perturb/mutate the correctly spelled queries and generate new misspelled test cases by developing a coverage-driven model-based test input curator (Figure 1B). The design of such a curator is non-trivial and we custom design it for every domain. We next discuss the steps needed to design the curator for the Speller.

1) Learning the Perturbations for Spelling Domain: We leverage the group of selected *misspelled* queries that require spelling correction to develop our perturbations, i.e., *typo models* for the Speller. Given the user input string with typos and the spelling-corrected string in the result that user selected, we model the difference between the two strings based on three base edits: insertion (ins), substitution (sub), and deletion (del). The edit distance is the minimum number of edits required to mutate the spelling-corrected query string into the one with misspells.

Among the misspelled queries in the production data set, we identified nearly 80% of the queries with edit distance 1 and nearly 20% of the queries with edit distance 2. As one can sequentially edit mutated strings to increment the edit distance, it is sufficient to model the 3 individual base edits using the abundant edit-1 queries. For $e \in \{sub, del, ins\}$, we model the (binned) position of the mutation based on the frequencies in the data according to

$$p(i \mid e) = \frac{C_i^e}{\sum_{j \in \mathcal{B}} C_j^e},\ i \in \left\{ \left[0, \frac{1}{B}\right), ..., \left[\frac{B-1}{B}, 1\right) \right\}, \quad (1)$$

where C_i^e is the number of misspells in production data with event e in position bin i. The positions are normalized between 0 and 1 and discretized into B bins. For $e = sub$ or $e = del$, we use the position of the character swapped or deleted from the correctly spelled string; for $e = ins$, we use the position of the character that precedes the inserted character.

For del, the event position is sufficient to generate the misspell word, and we do not need additional modeling; but for ins and sub, we need to specify more model parameters to generate the character to insert or swap to. For sub, we define a Markov matrix P^{sub}, whose k, ℓ-th element represents the transition from the k-th character to the ℓ-th character in the indexed alphabet set $\mathcal{A}$ (which includes the white space character and other necessary punctuation symbols). Thus, we have

$$P^{sub}_{k,\ell} = \frac{C^{sub}_{k \to \ell}}{\sum_{a \in \mathcal{A}} C^{sub}_{k \to a}}, \quad k, \ell \in \mathcal{A} \quad (2)$$

where $C^{sub}_{k \to \ell}$ is the number of substitutions observed in the production data from the k-th character to the ℓ-th character. For ins, we consider the conditional probability of inserting a specific character k at a position bin i:

$$p^{ins}(k \mid i) = \frac{C^{ins}_{i \rightsquigarrow k}}{\sum_{a \in \mathcal{A}} C^{ins}_{i \rightsquigarrow a}}, \quad k \in \mathcal{A}. \quad (3)$$

where $C^{ins}_{i \rightsquigarrow k}$ is the number of single insertions of the character k at bin i in the production data.

2) Generating New Test Cases & Expected Outputs: Given the probabilistic models developed from the first group of queries *with* misspells, we mutate queries from the second group of queries *without* misspells to generate new test cases to assess the Speller (Figure 1c). Algorithm 1 illustrates an example of how we mutate misspelled queries to be edit distance 1 from the original query. The data input is the group of queries that were selected only when a user had successful interaction with the suggestion that matches the original query input (and so no spelling correction was required). For each query (Line 2), the algorithm randomly selects (Lines 3, 4) one of the three mutation events: *ins*, *sub*, or *del*, and a random position in the word (Line 5). Given the mutation type and the position, it mutates the original query based on the learned probabilistic models. For example, if the event is sub (Lines 6-10), the algorithm takes the character at the selected position and looks up the transition probabilities in Eq. (2) to sample the character to substitute. The algorithm also handles

Algorithm 1: Generate Edit-1 Test Cases with Oracle

```
   Data: Production data with successful user interaction and no misspell queries.
   Result: Test cases with (misspelled) query, context, and expected correction.
1  testCases ← ∅;
2  foreach originalQuery, originalContext in data do
3      event ← Sample(sub, ins, del);
4      Sample i based on Eq.(1) with e = event;
5      position ← Round(i * len(originalQuery));
6      if event == sub then
7          k ← GetCharacterIndex(originalQuery[position]);
8          ℓ ← Sample based on row k in matrix in Eq.(2);
9          mutatedQuery ← SubstituteChar(originalQuery,
             position=position, from=k, to=ℓ);
10     end
11     if event == del then
12         mutatedQuery ← DeleteChar(originalQuery,
             position=position);
13     end
14     if event == ins then
15         k ← Sample based on Eq.(3) with i = i;
16         mutatedQuery ← InsertChar(originalQuery,
             position=position, character=k);
17     end
18     testCases.append({
         query:mutatedQuery,
         context:originalContext,
         expectedCorrection:originalQuery});
19 end
20 return testCases;
```

del (Lines 11, 12) and *ins* (Lines 14-16) appropriately. Then, the newly mutated query and the original context parameters become the new test input, while the original query becomes the correction that we expect in the suggested list of spelling corrections (Line 18). This gives us our fully-automatically obtained expected output, which is used by the test oracle.

In order to generate queries with more typos, one can trivially re-run Algorithm 1 on the mutated query until the edit distance increments to the desired value. For coverage analysis, we also keep track of the mutation types for each test cases: for example, we associate a test case with sub-del if we first generated a substitution and then generated a deletion so that the resulting query is edit distance 2 from the original query. Moreover, note that a correctly spelled word could occasionally be mutated to a valid English word, e.g., "near" mutated as "neat". On the one hand, we can re-mutate the word until it is absent in the English dictionary. On the other hand, we also allow such mutated valid English words to occur in our tests because the Speller suggests multiple candidates regardless of whether or not a word has an exact match in the corpus. For instance, even if both "near" and "neat" are included in the suggested candidates, we would expect the ML ranker to rank the correction "near" higher if the context is related to looking for a business near a location.

As noted earlier, within Algorithm 1, the expected output is automatically defined without requiring additional manual inspection. Manual selection from the production data could be prohibitive, and even strategies such as crowd sourcing incur higher cost and longer turnaround time compared to our strategy. Our expected output clearly leverages the scale of production data as well because we do not expect every single query to lead to successful user interaction in practice.

3) Defining the Coverage Space for the Speller: Our test case generation algorithm not only produces test cases at an increasing scale, but also ensures that the test cases accurately cover a multidimensional input space to reveal the boundaries of an ML system. For the Speller, we chose to cover two major dimensions: *query size* and *typo categories*, because both directly drive the difficulty of spelling correction and the accuracy of the ML ranker. *Query size* accounts for both the number of tokens and the length of the tokens in a production query that is presumed to be correctly spelled. We refer to an input with one token as a *word*, and a *phrase* otherwise. We group the queries into 4 categories: short/long words/phrases. A short word has fewer than 7 chars; and a short phrase has an average length of tokens less than 5 chars. These thresholds were chosen empirically to differentiate the Speller's capability to suggest spelling correction candidates. *Typo categories* builds upon the aforementioned base edits: ins, sub, and del. The 3 base edits alone can combinatorially cover all typo patterns – the space of typos is exponential in the alphabet size. However, it has been shown that around 95% typos in English spelling occur to be of minimum edit distance 2 or less [17] and most spelling correction algorithms do not focus on correcting for errors beyond edit distance 2. As we also observed similar phenomena in our data, we argue that it is sufficient to cover all edit-1 and edit-2 combinations. There are 3 scenarios (sub, ins, del) for edit-1 patterns, and 6 typo patterns (ins-ins, sub-ins, sub-sub, del-ins, sub-del, del-del) for edit-2 patterns, and thus, the typo dimension consists of these 9 broad categories.

B. Test Oracle

Once we run all the test cases (preserving original context information) through the Speller, we collect the ranked outputs from each input and compare each output with the test oracle (Figure 1C). For each test case, we define a boolean measure: k-*hit* to be TRUE if and only if the expected correction is ranked in the top-k spelling suggestions.

When considering any collection of tests (either aggregated over an entire test run or over a specific test coverage dimension), we are interested in the fraction of k-hits. For a fixed number of tests cases, the fraction of 1-hits measures how often the top suggestion matches the expected output; the fraction of 3-hits is a more lenient measure as it only requires the expectation to be among the top 3 suggestions; the fraction of 20-hits is our most lenient measure because the Speller only generates 20 spelling candidates. Consequently, the fraction of 20-hits will always be greater than or equal to the fraction of 3-hits or 1-hits.

C. Automated Test Triage

With millions of test cases, it can be prohibitive to evaluate individual failures. We designed an effective triage process that relies on finer featurization and automated clustering (Figure 1D) to subcategorize groups of test cases that are pre-categorized (e.g., insertions in long phrases) in the original coverage space. Note that the featurization and clustering steps can also take place right after the test inputs are generated as they do not require the test oracle. However, given the output of the test oracle, our workflow can prioritize these clustered subcategories based on the fraction of k-hits for further triaging.

TABLE I
STRING COMPARISON FEATURES USED FOR CLUSTERING

Feature	Description
`leve_dist`	Levenshtein Distance, minimum number of ins, del, and sub to mutate one word to another
`dale_dist`	Damerau-Levenshtein Distance, similar to Levenshtein Distance but counts transpositions as a single edit [18]
`hamm_dist`	Hamming Distance, position-by-position measure of mismatches between two strings
`jaro_dist`	Jaro Distance, measure that factors in the number of matching characters, transpositions, and string length
`jawi_dist`	Jaro-Winkler Distance, similar to Jaro Distance, but favors matches from the beginning [19]
`jacc_dist`	Jaccard similarity for the two sets where each set is the 3-gram representation of each word as elements
`mara_dist`	measure of whether two names are pronounced similarly using the Match Rating Approach algorithm [20]
`len_prop`	the number of characters in the correct word divided by the number of characters in the misspelled word
`ntoken_prop`	the number of tokens in the correct word divided by the number of tokens in the misspelled word
`nunique_prop`	the number of unique characters divided by the number of unique characters among the two words
`sfx_match`	the length of the longest common prefix over the length of the shorter string
`pfx_match`	the length of the longest common suffix over the length of the shorter string

1) Featurization: Prior to testing the Speller instance, each test case was generated with a (misspelled) query and its expected correction. For instance, the queries that share edit-1 ins are randomly generated from Algorithm 1. While the actual position and character are inserted are random, we characterize how the query differs from its expectation based on several numeric features, listed in TABLE I. Each feature requires an ordered pair of strings with an expected correction followed by its misspelled query. Positional features, e.g., `sfx_match` and `pfx_match`, capture the location of typos both from the beginning and from the end. Distance features, e.g, `leve_dist` and `jaro_dist`, offer different ways to count the degree of typos regardless of positions. Other measures such as phonetic difference, e.g., `mara_dist`, account for more subtle sources of spelling mistakes. These features are chosen to capture typo patterns that are independent of the expected correction, such that test cases can be grouped together accounting for mainly their misspell patterns.

2) Clustering: We apply spectral clustering [21] based on the derived features (each normalized to zero mean and unit variance) across all the test cases within a category defined in our coverage space. As an example, we would apply spectral clustering on all short words within the category in ins-sub typos, and determine more fine-grained subcategories that belong to ins-sub. For example, one cluster could define a

subcategory where the insertions are all space insertions that break a correctly spelled query in half. Our spectral clustering relies on a standard n-nearest neighbor graph construction [21], so the only hyper-parameter that we need to select is n, the number of nearest neighbors. Following the graph construction where each node represents a test case, we rely on the graph spectrum to determine the (disjoint) independent components. Here the hyper-parameter n affects the number of resulting clusters: the larger n is the fewer number of clusters. We use $n = 30$ as default, because we found that it empirically gives us a good balance between generality and specificity in each of the subcategories.

3) Feedback to the Input Curator: Clustering results in more refined partition of the original multi-dimensional coverage space. For instance, the coarse dimension of edit-1 deletions can contain single whitespace deletions in the query string. Even though the test curator did not initially sample production data to cover such cases, overrepresented passes or failures in such a cluster may suggest a new dimension to be covered in the next iteration. In general, unsupervised clustering can be useful for revealing new patterns that represent new coverage subspaces to track more nuanced regressions.

D. Developing End-to-End Testing Cycles

Our strategy can be flexibly implemented in practical software testing workflows that operate on different testing cycles. Broadly speaking, there are two scenarios that can trigger the Speller tests: ML updates due to training data updates or model changes, which take place less often, e.g., weekly; and Continuous Integration (CI) due to any code changes, which take place more often, e.g., hourly.

1) ML Updates: The first scenario relies on all components of our complete ML testing strategy, where we leverage production data to generate millions of test cases. Even though the frequency of testing ML is less than that for CI, the frequency of our test data refresh remains significantly higher than that of typical training refreshes for ML modeling and training. When comparing a baseline instance to a new instance of a software, we can rely on the coverage analysis to track the k-hit-based pass rate in each dimension and among different combinations. If any specific pass rate significantly decreases, e.g., more than 5%, then test case clustering can help enumerate similar failures that indicate defect classes.

2) CI: The second scenario of CI focuses on detecting any software regression based on a subset of test cases retrieved from the first scenario. For performance in this scenario, we prefer a minimal set of high confidence test cases which should always pass based on either 100% 1-hits or 20-hits. For test coverage, not only do we have sufficient test cases generated and categorized based on our coverage dimensions, we also have subcategories of test cases that define more refined subspaces in the overall coverage space. Test cases in each cluster share redundant patterns with one other, so we can simply keep one representative per cluster that perform stably as test cases for CI. In practice, the resulting number of test cases may only be a few hundred for such a test run.

IV. Hypothesis and Results

We have implemented our methodology in an automated regression test authoring tool that we now apply on our Speller to evaluate 3 hypotheses presented next.

A. Hypothesis 1: Our strategy detects a large number of previously unknown defects due to an increased scale of test cases

Metrics: For each test case, we rely on the test oracle to compute its two boolean values for the 1-hit and 20-hit status respectively. Recall from Section III-B that 1-hit is TRUE only if the expected correction is ranked first, whereas 20-hit is TRUE if the expected correction is suggested among the 20 candidates regardless of the ranking. Given n test cases, let n_{20} be the number of 20-hits and n_1 be the number of 1-hits, we rely on percentage-based metrics that capture the two boolean status for aggregate reporting: n_{20}/n (20-hits over total), n_1/n_{20} (1-hits over 20-hits), and n_1/n (1-hits over total). The larger the values the better the performance of the Speller. In general, both n_{20}/n and n_1/n_{20} are application-dependent because they reflect candidate generation and are sensitive to the extent of misspell (e.g. edit distance). On the other hand, n_1/n_{20} can be interpreted as a measure of the ML ranking performance because the denominator corresponds to the suggested candidates. If the ranker performed completely randomly, then the expected value for n_1/n_{20} would correspond to 5%. In general, we consider over 80% as robust performance for these metrics for our application.

Procedure: We randomly sampled over one million queries from our production traffic, where the data contain no personal identification information by our privacy policy. These queries had successful user interactions but did not require any spelling correction, thus contained correctly-spelled queries as well as contextual information. The typo generator automatically mutated each of these queries to contain misspells, *ceteris paribus*, so that new tests cases alter the original user spelling but preserve the original context and intent. Then, we tested these new inputs with a new Speller instance, and used the test oracle to summarize the performance metrics.

Results: The aggregated percentages over the one million test cases are summarized below:

	% of total	% of 20-hit
1-hit	61.48%	88.68%
20-hit	69.33%	100.00%

Over 30% tests cases contained some level of defects, as suggested by the 1-hit and 20-hit percentages of the total cases we tested (61.48% and 69.33%, respectively). The typo patterns used to mutate the test cases were designed to drastically exaggerate the percentage of failures that we would see in practice. The main reason that we increase the harder instances, where Speller more frequently fails to provide the correct suggestion or ranking, is to generate enough samples to cover the high-dimensional input space and reveal systematic behavior in automated clustering. (We will defer

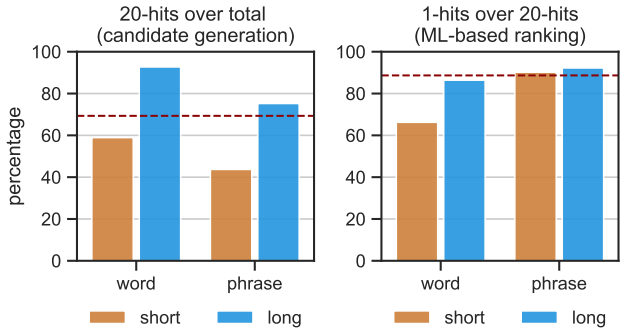


Fig. 2. Breakdown of the measures based on the query size. For each short/long word/phrase combination, the left plot shows the 20-hits over the total tests cases (which measures if the expected correction is suggested among the candidates), and the right plot shows the 1-hits over the 20-hits (which measures if the expected correction is ranked on the top, given that it is among the candidates). The red dashed lines correspond to the aggregated measures (left: 69.33%, right: 88.68%) prior to breaking them down by query size.

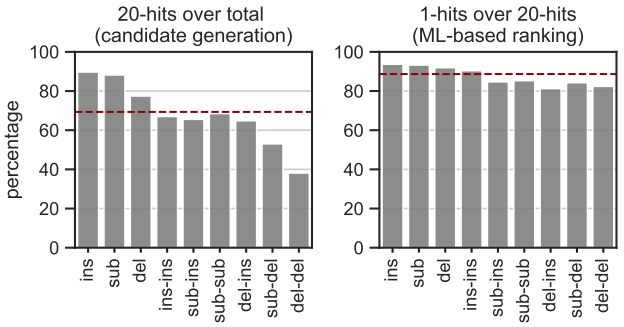


Fig. 3. Breakdown of the measures based on the typo categories. Similar to Figure 2, the left plot shows the 20-hits over the total tests cases, and the right plot shows the 1-hits over the 20-hits. The red dashed lines correspond to the aggregated measures (left: 69.33%, right: 88.68%).

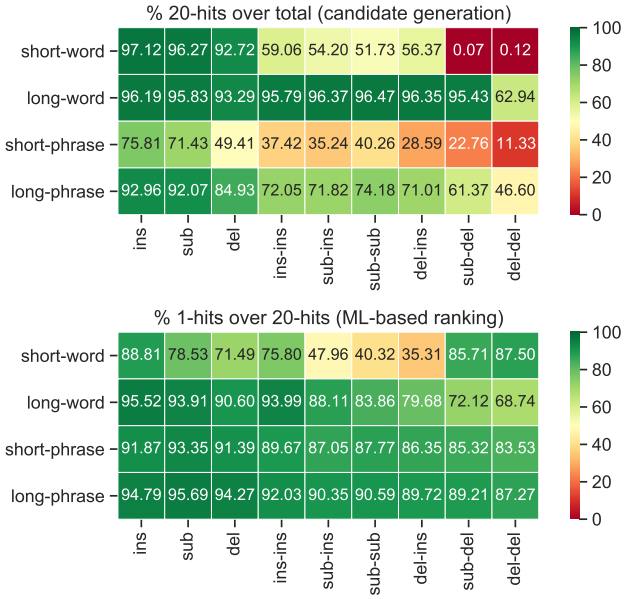


Fig. 4. Performance matrices of the measures for candidate generation and ML-based ranking. Similar to Figs. 2 and 3, the top plot shows the 20-hits over the total tests cases, and the bottom plot shows the 1-hits over the 20-hits. Each cell in the matrices represents test cases that share the same combination of query size and typo category.

the discussion of the coverage and clustering results under the other hypotheses.) We also observed that 88.68% of the 1-hits among the 20-hits, meaning that for the test cases where the expected spelling is suggested, the ML ranker recovered them as the top ranked suggestion 88.68% of the time. We found this percentage consistent with the performance of the trained ML model on its training and test data sets, even though our test cases did not include any data points from either data set.

Discussion: What differentiates our testing framework from existing ones is its ability to detect more failure cases, thanks to both the scale of the production data and the degree of test automation. We remark that generating test cases at this scale is impractical with traditional human-curated tests. Meanwhile, to capture the same number of test cases with defects from production data, one would also need to collect at least 10-times more samples, as we estimated that approximately 10% of the traffic contains real typos (and most of them are often easier to correct than our automatically-generated test cases).

B. Hypothesis 2: Sufficient coverage of cases in a multidimensional space reveals boundary regions of the ML system

Metrics: When evaluating the Speller in a multidimensional input space, we continue to focus on the two metrics: n_{20}/n (20-hits over total) and n_1/n_{20} (1-hits over 20-hits). By viewing these metrics that are aggregated over specific input regions, we can better understand the boundary behavior of the ML system. We first independently divided on each of the two key coverage dimensions: query size and typo categories, defined in Section III-A3. Then, we analyzed the interaction of the two coverage dimensions by considering the metrics in regimes with specific query size and typo category combinations.

Procedure: For our one million data points, we first partitioned all the test cases based on the short and long query sizes to compute the two metrics (resulting in a total of 4 aggregation regimes). The breakdown of the number of test cases over the first dimension of query size as follows:

	word	**phrase**
short	107,416	245,481
long	166,317	593,831

Longer phrases were most frequent and shorter words were least frequent in this production data set. Next, we partitioned all test cases based on the typo categories to compute the same metrics (resulting in a total of 9 aggregation regimes). To exaggerate the typo patterns and provide sufficient coverage in each category, we randomly distributed around 12,000 test cases for each category uniformly. Last, we considered the

performance for every query size and typo category combination, which lead to 36 aggregation regimes, and report the two metrics in two 4×9 performance matrices.

Results: As shown in Figure 2, we observed that the Speller performs better on longer words and phrases than their shorter counterparts because the search space for longer inputs is much narrower. Based on the breakdown along the second coverage dimension in Figure 3, the Speller indeed performed better on edit-1 typos than on edit-2 typos. However, among edit-2 typos, ML-based ranking performed best on ins-ins, whereas candidate generation performed best on sub-sub. We also compared the performance in greater detail by looking at the combinations in performance matrices, as shown in Figure 4. With the multidimensional coverage space, we can track the performance of the Speller's ML behavior in different regimes that capture the intrinsic difficulty of spelling correction. We interpret the collection of test cases from a cell which scores below average (e.g., the edit-2 errors in short words) as classes of boundary cases in the combination space.

Discussion: Comparing the two plots in Figure 2 and in Figure 4, we observed an interesting phenomenon: *candidate generation generally performs better on longer inputs but performs worst on short phrases, whereas ML-based ranking generally performs better on phrases but performs worst on short words.* We reasoned that the ML-based ranking component relies heavily on contextual information as well as query-based features, such as the number of tokens, so when only a short word is given (compared to a phrase of similar size), there is more uncertainty in the ML decision. In contrast, a simple search-based system for candidate generation is more efficient in searching for similar words than phrases because the search space is smaller.

Our metrics identify boundary regions that distinguish the behavior of candidate generation and ML-based ranking for the Speller. Regions with lower performance scores often result from (1) heuristics that limit the search space for computational tractability, and (2) insufficient data in this particular region during model training. Typically, ML models rely on one aggregate measure, such as the F_1 score, across all test or validation samples. Yet, such aggregate measures can mask systematic biases when the model performance improves in one regime but worsens in another. With the built-in granular structure we developed, the performance matrices can not only suggest regimes to sample additional training data for model improvement but also track the performance of different versions of the Speller and alert developers if any regression occurs in any cell.

C. Hypothesis 3: Automatic clustering can triage subcategories of defects within the multidimensional coverage space

Metrics: Previously, we aggregated the two metrics: n_{20}/n (20-hits over total) and n_1/n_{20} (1-hits over 20-hits) over predefined 2-D input spaces, where the inputs are divided into 4×9 regimes. After unsupervised clustering, each regime can be further partitioned based on the clusters, and we can re-evaluate the two metrics in each individual cluster. Because there could be dozens of clusters in one regime, we compare the cluster-specific performance with the aggregated one for this regime to identify outliers. For instance, for the (short-phrase, *ins*) regime that has 75.81% 20-hits over total in Figure 4, if any of the clusters within this regime has less than this value, they are indeed as worse than average. In general, we can prioritize triaging the test cases from clusters with the lowest score, especially if it diverges far from the average.

Procedure: We relied on unsupervised clustering within each cell in the 4×9 performance matrix. The spectral clustering algorithm (based on 30-nearest neighbor graphs) partitioned test cases in each cell such that the cases in the same subcategory share more refined typo attributes (e.g., position and phonetics) Figure 5 illustrates an example of 5 randomly selected samples from 3 representative clusters (out of 79 clusters) under the category of "edit-1 insertions in a long word". Each of these 3 subcategories include a specific type of edit-1 insertion: cluster 1 captures a space insertion in the middle of a word; cluster 2 captures a double-spell insertion near the middle of a word; and cluster 3 captures a random letter insertion in the beginning of the word. These clusters and others can be differentiated because their features capture the insertion positions, unique characters, resulting number of tokens, and etc. Additionally, as shown in Figure 5, some of the distance features (i.e., distance-based measures) are more similar to each other compared to positional features (i.e., prefix and suffix matches).

Results: To visualize aggregate summaries of all clusters/-subcategories simultaneously, we consolidated scatter plots measuring the percentage of 1-hits over 20-hit against the percentage of 20-hits over total within each subcategory in Figure 6. As expected, the Speller's performance on short queries (marked as orange) vs. long queries (marked as blue) are visually distinguishable. Consistent with the previous results, Figure 6 suggests that candidate generation is more variable in phrase inputs whereas ML-based ranking is more variable in word inputs, as the scattered points spread vertically on the top row and horizontally on the bottom row.

Discussion: Unsupervised clustering automatically organizes subcategories that would have been challenging to inspect by manual inspection because each category has over thousands of cases that are highly diverse. With newly identified subcategories where test cases share similar typo patterns, it becomes more convenient for developers to identify defect class.

Among the word inputs, we identified a subcategory (pinpointed in the top row of Figure 6) with 251 test cases of insertions including "*ochi*" (expect correction: "mochi"), "*lubs*" (expect correction: "clubs"), and "*laze*" (expect correction: "blaze"), where the first letter of a short word is deleted. As one can imagine, there are many ways to correct the spelling of these short queries and the ML ranker can only rely on limited contextual information to rank them, so these queries indicate a boundary case with more defects than others. Among the phrase inputs, we identified a subcategory (pinpointed in the bottom row of Figure 6) with 66 tests cases of deletions such as "1 apple park *wpay*" (expected correction: "1 apple park way")

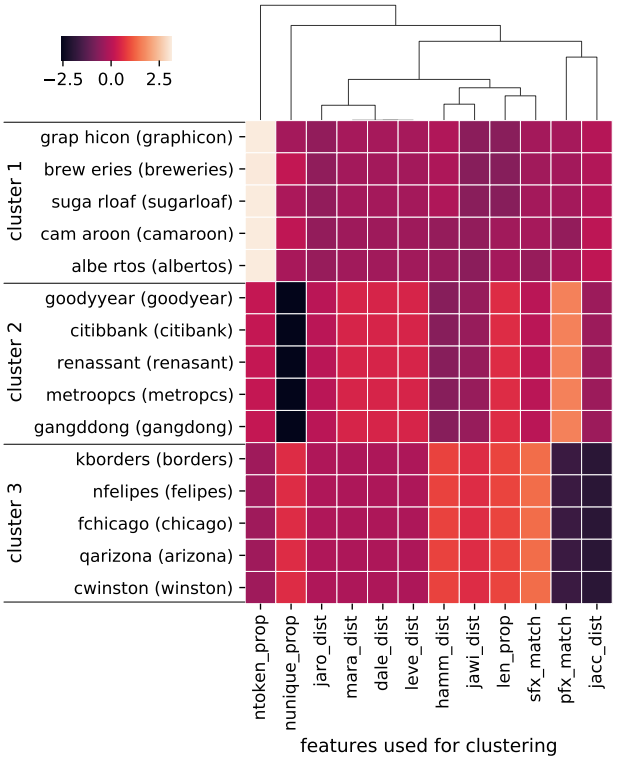


Fig. 5. Example test cases with their clustering features. Five test cases were selected from three representative clusters. Each case is displayed based on the query and its expected correction in the following brackets. Each row of the heatmap corresponds to a test case, and each column is a distinct feature. The color of each cell represents the feature values (in z-score for display purpose). The test cases are ordered by their cluster grouping (based on from our automatic triaging), and the columns are ordered by automatic hierarchical clustering (based on built-in plotting utility function for display purpose).

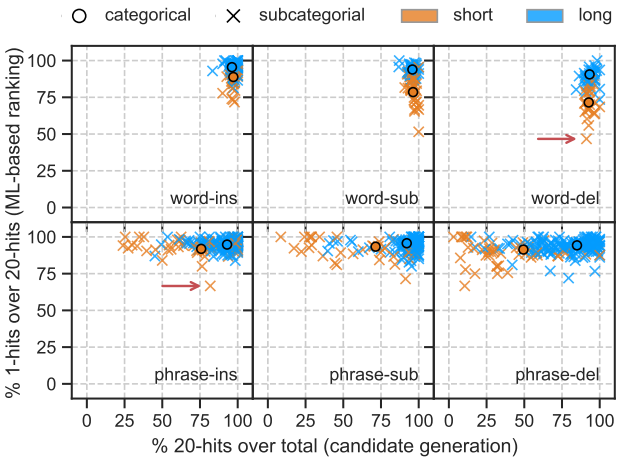


Fig. 6. Scatter plots of performance measures for subcategorized test cases. The rows correspond to word vs. phrase inputs and the columns correspond to different edit-1 typos. (We omitted edit-2 typos due to space limit.) Each 'x' marks a specific cluster/subcategory of test cases, and its position indicates the two percentages computed on the test cases within the subcategory. The percentages aggregated over tests cases across all the clusters in one category are marked as 'o' (previously shown as individual cell values in Figure 4). Short and long query types are distinguished by orange and blue respectively. The red arrows highlight the worst performing clusters of test cases across all word and phrase input types respectively.

and "100 fourth *est*"(expected correction: "100 fourth st"), where the patterns are related to multi-token addresses where last (short) token is modified. We discovered this defect class because when short tokens such as "st", "rd", "way", and etc. are misspelled, multiple corrected addresses can confuse the ML-based ranking system. These edge cases would have been hidden with non-address queries if we only considered them as part of their broader category of "edit-1 insertions within short phrases". Instead, unsupervised clustering allowed us to identify such patterns without requiring manual inspection of all phrases in this broad category.

V. Related Work

We build upon the work of others to develop a practical methodology to address challenges of regression testing for ML software. Here we discuss prior work on testing ML systems, input perturbation, testing spelling correction systems, and limitations of existing metrics to assess the overall correctness of ML systems.

Several researchers have enumerated the challenges of assuring the quality of ML systems. Zhang et al. [2] surveyed a comprehensive landscape of ML testing, linking research topics in robustness, privacy, efficiency, and fairness. Other reviews have emphasized general aspects such as defect detection in ML programs [9] or implementation quality [1].

Crafted **perturbations** in testing trace back to metamorphic testing [10]: it relies on perturbation patterns where the expected change in the output is predictable. Thus, metamorphic testing can effectively alleviate the test oracle problem and the test case generation problem. Xie et al. [22] proposed multiple ways to apply mutational analysis in ML systems including addition or removal of samples, classes, and features. Zhang et al. [11] introduced perturbation into the training data labels, re-trained the model, and measured the decrease of accuracy to assess model fit. Often, the mutations are crafted such that the ML system validation performance should be invariant to the perturbations. For image data, Huang et al. [23] relied on manipulations such as changes to camera angle or lighting conditions to assess the robustness of an individual image classification. Many works have proposed white-box testing approaches that narrow down the focus on specific ML models, specially neural networks. Pei et al. [12] proposed DeepXplore, a differential white-box testing approach for neural networks: it relies on gradient-based search techniques to automatically optimize inputs cases so that it achieves a 'neuron coverage' (the fraction of activated neurons in a neural network for a set of test inputs). Odena and Goodfellow [13] modified coverage-guided fuzzing such that adaptively mutated inputs can be updated to automatically search for diverse neural networks states.

Our Speller developers use **classical metrics**, such as F_1 score to evaluate the goodness of their ML models in isolation.

However, the F_1 score alone cannot fully capture end-to-end software service behavior. The technical reason is that the actual ML code sits among many supporting software components that need to function correctly [24]. When the F_1 score is low, the root cause can be large amount of noise in the data, the choice of a ML model with low predictive power [16], or software issues related to logic condition handling, data conversion, numerical accuracy, and etc. For software testing, we need to rely on additional metrics and processes so that we can distinguish issues that require data collection or ML model improvements from those that require more immediate bug fixes.

VI. Conclusion

We identified several practical challenges for functional regression testing of ML software: the rapid obsolescence of input and expected output parts of test cases; lack of coverage measures; and insufficiency of individual *ad hoc* test failures to reveal systematic bugs. We presented a methodology to overcome these challenges, the salient features of which are coverage-driven perturbation of production data to automatically author a large number of test cases, consisting of both test inputs and corresponding expected outputs, followed by clustering to reveal failure patterns indicative of software/model bugs. We demonstrated our methodology by applying it to a context-aware ML-based Speller software. Our results showed that we can completely automatically "refresh" our test suite in numbers that are large enough to reveal patterns of failures.

Our study that focuses on the ML-based Speller raises some threats to validity due to its empirical nature. We analyzed one large set of production data that reflected specific bugs during clustering. Thus, data captured in other time windows may yield different test cases and clustering patterns. Due to the data constraint in our application, spelling corrections used for very long input strings, such as sentences and text documents, have not been explored. Further, the context variables we consider does not reflect the wide spectrum of all possible contexts available to other spelling correction software.

Nevertheless, the key elements of the Speller testing methodology are generalizable, including (1) leveraging the scale of production data to automatically generate both test inputs and expected outputs using a learned domain-specific perturbation model. These test cases generalize the existing cases in the production data to enrich edge cases that are underrepresented in real training and test data; (2) adopting user interaction data from production to resolve the oracle problem: we determine the expected output of a number of test cases where the consequent feedback is positive and indicates that the users receive correct outputs; and (3) mining the patterns of test cases using unsupervised learning and clustering the test cases so we can automatically detect defect classes based on the subgroups with high number of failure cases. Our next step is to apply the methodology to additional ML systems to automatically author test cases.

Acknowledgment

We thank Elif Aktolga, Yutong Li, and Himanshu Yadava for sharing their insights into the Speller software; Ryan Wang for providing infrastructure support; Emily Kowalczyk and Alex Braunstein offering their feedback and constant support of our work.

References

[1] S. Masuda, K. Ono, T. Yasue, and N. Hosokawa, "A Survey of Software Quality for Machine Learning Applications," in *ICST*, 2018, pp. 279–284.

[2] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE TSE*, 2020.

[3] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for gui-based software applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, p. 4, 2007.

[4] T. Kanij, J. Grundy, and R. Merkel, "Performance appraisal of software testers," *Information and Software Technology*, vol. 56, no. 5, pp. 495–505, 2014.

[5] D. Marijan, A. Gotlieb, and M. K. Ahuja, "Challenges of testing machine learning based systems," in *AITest*. IEEE, 2019, pp. 101–102.

[6] I. Avazpour, T. Pitakrat, L. Grunske, and J. Grundy, "Dimensions and metrics for evaluating recommendation systems," in *Recommendation systems in software engineering*. Springer, 2014, pp. 245–273.

[7] D. Marijan and A. Gotlieb, "Software testing for machine learning." in *AAAI*, 2020, pp. 13 576–13 582.

[8] D. Mondal, H. Hemmati, and S. Durocher, "Exploring test suite diversification and code coverage in multi-objective test case selection," in *ICST*. IEEE, 2015, pp. 1–10.

[9] H. B. Braiek and F. Khomh, "On testing machine learning programs," *arXiv preprint arXiv:1812.02257*, 2018.

[10] A. Nair, K. Meinke, and S. Eldh, "Leveraging mutants for automatic prediction of metamorphic relations using machine learning," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2019, pp. 1–6.

[11] J. M. Zhang, M. Harman, B. Guedj, E. T. Barr, and J. Shawe-Taylor, "Perturbation validation: A new heuristic to validate machine learning models," *arXiv preprint arXiv:1905.10201*, 2019.

[12] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 1–18.

[13] A. Odena and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," *arXiv preprint arXiv:1807.10875*, 2018.

[14] R. Kodre, H. Ziv, and D. Richardson, "Statistical sampling based approach to alleviate log replay testing," in *ICST*, 2008, pp. 533–536.

[15] E. Brill and R. C. Moore, "An improved error model for noisy channel spelling correction," in *Proceedings of the 38th annual meeting on association for computational linguistics*, 2000, pp. 286–293.

[16] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics Springer, Berlin, 2001, vol. 1.

[17] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Comm. of the ACM*, vol. 7, no. 3, pp. 171–176, 1964.

[18] G. V. Bard, "Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric," in *Proceedings of the fifth Australian symposium on ACSW*, 2007, pp. 117–124.

[19] C. William, R. Pradeep, and F. Stephen, "A comparison of string distance metrics for name-matching tasks," in *IIWeb*, 2003, pp. 73–78.

[20] D. Kaur and N. Kaur, "A review: An efficient review of phonetics algorithms," *IJCSET*, vol. 4, no. 5, p. 5068, 2013.

[21] U. von Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, vol. 17, no. 4, pp. 395–416, aug 2007.

[22] X. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *JSS*, vol. 84, no. 4, pp. 544–558, 2011.

[23] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety Verification of Deep Neural Networks BT - Computer Aided Verification," R. Majumdar and V. Kunčak, Eds., 2017, pp. 3–29.

[24] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Advances in neural information processing systems*, 2015, pp. 2503–2511.