Improving ML-Based Information Retrieval Software with User-Driven Functional Testing and Defect Class Analysis

Junjie Zhu
Apple Inc.
Cupertino, CA, USA
jason.zhu@apple.com

Teng Long
Apple Inc.
Cupertino, CA, USA
teng.long@apple.com

Wei Wang
Apple Inc.
Cupertino, CA, USA
wwang52@apple.com

Atif Memon
Apple Inc.
Cupertino, CA, USA
atif.memon@apple.com

ABSTRACT

Machine Learning (ML) has become the cornerstone of information retrieval (IR) software, as it can drive better user experience by leveraging information-rich data and complex models. However, evaluating the emergent behavior of ML-based IR software can be challenging with traditional software testing approaches: when developers modify the software, they cannot often extract useful information from individual test instances; rather, they seek to holistically verify whether—and where—their modifications caused significant regressions or improvements at scale. In this paper, we introduce not only such a *holistic approach* to evaluate the system-level behavior of the software, but also the concept of a *defect class*, which represents a partition of the input space on which the ML-based software does measurably worse for an existing feature or on which the ML task is more challenging for a new feature. We leverage large volumes of functional test cases, automatically obtained, to derive these defect classes, and propose new ways to improve the IR software from an end-user's perspective. Applying our approach on a real production Search-AutoComplete system that contains a query interpretation ML component, we demonstrate that (1) our holistic metrics successfully identified two regressions and one improvement, where all 3 were independently verified with retrospective A/B experiments, (2) the automatically obtained defect classes provided actionable insights during early-stage ML development, and (3) we also detected defect classes at the finer sub-component level for which there were significant regressions, which we blocked prior to different releases.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Information systems** → *Test collections*; *Relevance assessment*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558941

KEYWORDS

Machine Learning Testing, Information Retrieval System Testing, Relevance Search, AutoComplete Search, Query Interpretation

ACM Reference Format:
Junjie Zhu, Teng Long, Wei Wang, and Atif Memon. 2022. Improving ML-Based Information Retrieval Software with User-Driven Functional Testing and Defect Class Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3540250.3558941

1 INTRODUCTION

Modern industrial-scale information retrieval (IR) software, such as search engines and recommendation systems, rely heavily on machine learning (ML) to learn the intention and preference of different users based on their queries [3]. Compared to traditional IR methods that used few features (e.g., term frequency and inverse document frequency [21]), modern IR systems have evolved thanks to richer features (e.g., search history, geographic location, and reviews [27]), increasing amount of training data, and better ML techniques (e.g., deep learning methods [8]). However, this technological leap has made system testing of such ML-based IR (ML-IR) software challenging for a number of reasons.

First, ML-IR developers are less concerned with failures on an individual query or test input – rather they want to know if their modification has *holistically* improved or regressed any aspects of the software. In traditional IR software testing, a modification could cause a conventional lookup function $foo(i)$ to fail its release due to a single test input i if the scenario is a blocking issue (also known as a 'P0 failure' within the software industry). On the other hand, if *foo* were a ML-IR software function, then the individual failure for the input i would be acceptable if the modification caused *foo* to perform better on a large number of other inputs. Take a location-aware restaurant recommender system as an example of a ML-IR software, and assume that over 98% of its customer base are young vegetarians. If remote steakhouses start to rank lower due to a code modification or ML-model retraining, we might see recommendation improvement for many vegetarians, despite a very localized regression where only a small fraction of the omnivore users start seeing fewer number of steakhouses nearby.

Second, ML-IR developers wish to gather information-rich testing results for specific classes of inputs that influence the decision

space of the ML components [24]. Suppose the developers of the aforementioned recommendation software learn that the ML is only improving recommendations for vegetarian users at the cost of worsen recommendations for the omnivore users. Compared to just an overall performance score, such refined information can guide the developers to investigate biases in the training data, adapt the ML to improve upon a specific input subspace (defined by age and dietary preferences in this case), and eventually improve the ML to make smarter recommendations. Making these subclasses of inputs tractable can help advance ML development and maintenance, especially as the software's decision space is continuously evolving with changes in the real data, e.g., shifts in user taste and restaurant quality.

Third, to comprehensively understand the behavior of their software, the ML-IR developers need a large number of test cases to cover the diverse decision space of the software, e.g., tests to cover all types of restaurants, all age groups of customers, all dietary preferences. Not only are such large test suites difficult to create, they are even harder to maintain. A conventional system test case is of the form <*input, expected_output*>, where *input* may mimic an end-user, e.g., a query input 'indian food' and, *expected_output* determines if the software is functioning correctly, e.g., an output list of Indian restaurants ranked by distance. Each test brings the software into a certain internal state, applies the test input, and checks the output. A large number of test cases may become obsolete as the ML software frequently improves its output over time by learning from new training data. Consequently, hard-coded *expected_output* in the test cases may turn stale with respect to the software's new improved output.

In this paper, we develop a system testing approach that addresses these challenges. Our approach automates test authoring with appropriate expected outputs, i.e., test oracles [12, 35, 39, 40] by leveraging the scale of production data which reflect both positive and negative user feedback. Moreover, we rely the notions of *holistic measures*, which quantify overall functional behavior of a software to help developers make pass/fail decisions on their modifications, and *defect classes*, which capture patterns in partitioned input spaces that can help developers reason about improvements and regressions.

Crucially, our approach factors in three different levels of evaluation that are typically treated separately: *user-level*, *model-level*, and *engineering-level* [32]. We have implemented the approach and evaluated it on a real Search-AutoComplete (Search-AC) ML-IR system that supports beyond millions of queries per day. Consequently, the results in this paper are organized to illustrate how we can cover all three levels of testing comprehensively.

User-level testing mainly relies on online evaluation based on live production traffic [17]. Such testing, including A/B testing [34], has become the gold standard for industrial-scale IR systems due to the possibility of fielding a large number of users in an end-to-end fashion, as well as the business impact that is associated with user engagement. Yet, online measures do not directly expose bugs at the engineering or model levels, so fixing these bugs is expensive not only because of wasted online traffic but also because of the additional engineering hours needed to understand the root cause, retrain the ML models, and deploy the system.

Our contribution at this level is that we were able to completely automatically reject a number of software modifications, of which we present two along with another modification which was accepted — all these outcomes were confirmed via independent online A/B testing. By relying on the holistic measure of the rate at which the system returns no results or recalls the ones desired by the user (i.e., test oracles automatically derived from production queries), we were able to distinguish improvements and regressions that were consistent with the reaction by a large number of users.

Model-level testing ensures that individual ML models achieve performance gains with increasingly complex model architecture. The models are typically evaluated *offline* (not using live production traffic) via common ML metrics, such as the precision, recall, and the F_1 score [9]. These metrics, however, do not necessarily capture holistic and emergent behavior of the IR systems that can consist of multiple ML models, non-ML tables hard-coded for certain inputs, and code logic that puts these together.

At this level we conducted deeper analysis of the system based on the functional coverage space that captures the ML behavior of the Search-AC system. We were able to identify many defect classes, among which we present three examples that not only covered a broad class of inputs but also helped define new opportunities for ML model improvement.

Engineering-level testing emphasizes correct software implementation, and is typically performed by traditional approaches such as unit tests and simple integration tests [26]. Validating these aspects of the software correctness is undeniably necessary, but, as we illustrated via the restaurant recommendation system example, ML-based software needs significantly more attention at the system level, because that is where its behavior truly emerges.

At this level, we re-purposed the test oracles derived automatically from production data to directly evaluate the multi-task query interpretation component within the Search-AC system. By such sub-component testing, we identified implementation artifacts, of which we present two in this paper to show how the approach can drive the triaging of new regressions and bug fixes.

The main contribution of our work is two-fold. First, we offer a practical methodology to perform system testing for industrial-scale IR systems, which complements popular online testing approaches by providing early improvement and regression signals. To our knowledge, there is limited research on such ML-based IR system testing despite the wide-spread usage of search engines and recommendation systems in industry. Second, our methodology can be generalized to other ML-based systems as it is designed for systems with large input spaces that contain complex functional dimensions. As such, our work contributes a solution to a class of similar applications within the broader ML-testing literature [25, 41].

Following this section, we model our holistic measure, define defect classes, and describe how we automatically obtain test cases. In Section 3, we present our Search-AC application that we use in our experiment, and in Section 4, we provide our hypothesis and results. We outline a methodology for generic ML-based systems in Section 5 after all the case studies, highlight related work in Section 6, and finally conclude in Section 7 with a discussion of future work.

2 MODEL

In this section, we describe the common model used for defining the metrics and designing the procedures that address different research hypotheses in Section 4. Thus, this section focuses on generic approaches to holistically measure ML-IR software using functional testing, as well as strategies to identify defect classes to both detect new regressions and drive improvements.

Without loss of generality, one simplified example we consider in this section is a Search-AC system which completes partial user queries, e.g., the string "hote" may *auto-complete* to "hotel." This AC also supposed to leverage ML sub-components to 'understand' the query intent based on the query semantics and user context, e.g., geographic location, to improve the query completion task; e.g., if a user is in the country Angola, then "hote" may auto-complete to the city name "Hote" as opposed to "hotel." (The specific subject application for which we present the results that address our hypotheses will be described with full details in Section 3.)

2.1 Holistic Functional Testing

A functional test returns the pass/fail outcome of an individual input by comparing the actual output and the expectation, known as the test oracle [12, 35, 39, 40]. Take the Search-AC system as an example, we may expect the oracle 'hotel' to be ranked among the top-3 completion suggestions with input 'hote' for a given context. If this test fails on only the input 'hote' due to a slight ranking change after an ML update, it does not necessarily mean AC has regressed, especially if its ML improves the ranking on many other query inputs. Thus, an individual test case for an ML software cannot fully generalize a systematic behavior unless there is sufficient evidence supported by many other similar inputs.

Here, we present a holistic measure that can easily build upon such existing functional metrics and their aggregated pass/fail outcomes to determine if a failure is systematic. Let $A_X = \{a_0, a_1, ...\}$ be a set of 0-1 boolean values that represent failure or passing outcomes on the test suite indexed by X for a specific function of the system under test (SUT), which we refer to as SUT-A. We define a holistic measure to aggregate these individual outcomes via a weighted average:

$$h_A(X) = \sum_{i \in X} w_i a_i, \tag{1}$$

where $\sum_{i \in X} w_i = 1$ and $w_i \geq 0$ for all i. The weights capture the relative importance of individual test cases. In practice, the weight can default to $w_i = 1/|X|$, where $|X|$ is the size of the test set, then Eq. (1) becomes the arithmetic mean of the pass/fail outcomes, also known as the *pass rate*. The two constraints on the each weight value w_i ensure that the holistic measure equals its maximum value of 1 only when all tests are passing.

Another way to construct the weights is to rely on the value space of one or more specific test dimensions with a mapping function f, such that $w_i = f(v_i)$, where v_i is the value in these dimensions. For instance, if we consider v_i to be the specific *query locale* (e.g., place where the query originated) as a dimension for the Search-AC system, one can systematically assign higher weights to the inputs whose locales correspond to important markets already in production and lower weights to those that correspond to emerging markets under examination.

When we compare the outcomes for SUT-A with those that result from a modified version, which we refer to as SUT-B, with $B_X = \{b_0, b_1, ...\}$, we can directly evaluate the difference between the two SUTs on the same input set X with the holistic measure in Eq. (1) as follows:

$$D_B^A(X) = h_B(X) - h_A(X) = \sum_{i \in X} w_i (b_i - a_i). \tag{2}$$

From this equation, we can deduce that the overall function for SUT-B has improved (with respect to SUT-A) if this difference is positive, or it has regressed if the quantity is negative. The second equality in Eq. (2) that refactors the summation implies that one can still meaningfully evaluate individual cases that improved or regressed based on how they contributed to the overall holistic measure of a function. For a given test case i, the difference between the two outcomes $(b_i - a_i)$ equals 1 if the case has improved from SUT-A to SUT-B, -1 if the case has regressed, and 0 if there is no difference. We will show in Section 4.1 three examples of software modifications, two of which we blocked, and one passed, using this model.

Because randomness of the individual pass/fail decisions can arise from ML fuzziness [14] and infrastructure flakiness [20, 42], the holistic measure is effectively an observed statistic based on the specific test input set. To make it more robust to such randomness, one also has the option to compare the difference in Eq. (2) with thresholds to create decision rules for improvement/regression.[1]

2.2 Defect Class Analysis

We also partition the input set based on functional coverage dimensions to enable efficient triaging [36]. The dimensional analyses can help us identify specific *defect classes*, e.g., the class of omnivore customers who received lower steakhouse rankings in Section 1, to understand the root cause of a holistic regression.

Take the general case where the test suite X can be partitioned as $\{X_0, X_1..., X_{m-1}\}$, we can then compute the metric in Eq. (1) for each partition to obtain sub-metrics

$$h_A(X_0), ..., h_A(X_{m-1}) \text{ and } D_B^A(X_0), ..., D_B^A(X_{m-1}) \tag{3}$$

given SUT-A and SUT-B. Suppose that one of the input classes X_j has a very low value of $h_A(X_j)$ or negative value of $D_B^A(X_j)$, then we refer to X_j as a defect class.

The analyses of defect classes are not only useful for identifying relative regressions, but also helpful for understanding how to improve the current ML-based software with prioritization. We can sort $h_A(X_0), ..., h_A(X_{m-1})$ and prioritize the defect classes that are much worse than the overall average $h_A(X)$. Addressing these worst-performing defect classes would likely yield more noticeable improvements. With any new modifications introduced (say via SUT-B), developers can then rely on the difference metric $D_B^A(X_j)$ to track if the change presents any sign of improvement. Let us revisit the previous AC example where we partition the space of inputs based on the locale dimension. We can measure if the model improvement in a new locale with inputs X_j comes as the cost of worsened performance in another locale with inputs X_k by inspecting the directionality of $D_B^A(X_j)$ and $D_B^A(X_k)$ and further quantifying the trade-offs. In Section 4.2, we will show three examples of

[1] One way to determine such thresholds is to compare two identical SUTs on the same input set and establish what differences are likely insignificant.

defect classes: (1) *Category queries with prepositions and locations*, (2) *Queries with unique places*, and (3) *Users with distant viewports*, all of which helped the developers of our software under test to make improvements.

2.3 Automated Test Oracle

Our methodology for ML-IR software requires sufficiently large input size $|X|$ to ensure that the statistics in Eq. (1) and Eq. (2) do not deviate too far from the true underlying software behavior in production (due to the 'Law of Large Numbers' [19]). When we want to additionally cover specific functional dimensions, each partition (e.g., $\{X_0, X_1..., X_{m-1}\}$) of the input space also requires sufficient number of inputs so that a sub-partition can provide abundant evidence of a specific defect class. Thus, both the holistic measures and the defect class analyses heavily depend on practical solutions to generate a large number of test cases automatically.

Test automation relies on resource and data availability, and more importantly, deriving accurate expectations, which is known as the 'test oracle problem' [40]. To detect improvement or regressions in ML-IR software, we solve the oracle problem with production data that already captures some real user feedback based on the existing baseline version in production. For instance, if many users have tapped and interacted with the result 'hotel' returned from AC when typing 'hote', outside of Angola, we can trust the suggestions 'hotel' as a reasonable oracle for each of the user scenarios; within the context of Angola, the suggestion "Hote" is a better oracle if most end users have tapped and interacted with this result. The data captured by the test oracle is highly application-specific, so appropriate test oracles that offer insights into the core functions are the key to providing actionable feedback for development.

The benefit of using production data for ML-based software is multi-fold. First, it does not require case-by-case inspection and authoring by test authors. Instead, it only requires automated logging of the relevant query input, context, and user-selected output. Next, it is well-suited for software that can have fuzzy or complex outputs, such as AC outputs with ranked suggestions. The expected output from an AC can be elusive because the result is data dependent and can alter with slight variation of the data dependencies. Nevertheless, if a suggestion that a user previously liked or selected starts disappearing or getting much lower ranks, the user is likely to have a worse experience with the software. If many users experience the same behavior, then the holistic measure will reflect the degree of this regression systematically. Last but not least, these measures can offer meaningful business insights if they reflect the user impact of the software, such as common search metrics like user clicks. In Section 3.3, we will show how we can concretely leverage user interactions to automatically derive test oracles for a specific Search-AC system.

2.4 Stratified Sampling

The test inputs from production data we consider are often observational, in that we do not have control over the user traffic. Therefore, it is often necessary to balance the size of the inputs for different categories for critical dimensions, so that the associated functionality can be evaluated *fairly* [18]. For instance, suppose that an AC system applies three different ML models for short, medium, and long input queries, which are distinguished by 5- and 20-character cutoffs. (Here the input query length would be a coverage dimension with 3 categories.) If short queries are heavily represented (e.g., 99%) in production traffic, then this imbalance may cause insufficient evaluation of the two models that rely on longer inputs. In other cases where there is only a single ML, it is still useful to partition the input space based on one or more coverage dimensions to understand if there are any ML biases that distinguish different outputs.

To optimize the input test size, we use the sampling strategy in Algorithm 1 to stratify such biases in production, and construct X with a bottom-up approach. The algorithm utilizes parallelization and built-in functions under common map-reduce frameworks [7]. The function `ComputeSampleRatio` determines the sampling ratio for a each category in the coverage dimension based on `n`, the desired number of counts per category. If the counts of a specific category is less than `n`, then the sampling fraction is set to 1, so that all existing samples will be used. The function `sampleByKey` implements a default sampler without replacement via Poisson sampling which does not generate exact sample sizes of `n` but is highly performant. A notable property of this algorithm is that one can iteratively stratify another dimension on a given stratified output, so that the final output can contain fair representations of multiple dimensions.

We will apply Algorithm 1 based on specific input dimensions of the Search-AC system and its ML sub-component in Section 3.4

Algorithm 1: Distributed Stratified Sampling

Input: Parallelized input data: `input` with attribute `attr` and target size `n`.
Output: Stratified data `output` with approximately `n` samples for each of category.

```
Function ComputeSampleRatio(countMap, n):
    fractionMap ← emptyHashMap;
    foreach entry ∈ countMap do
        fraction = min(1, n / entry.getValue());
        fractionMap.put(entry.getKey(), fraction);
    end
    return fractionMap;
End Function;
pairs ← input.map(x => (x.get(attr),x));
countMap ← pairs.countByPair();
fractionMap ← ComputeSampleRatio(countMap, n);
sampledPairs ← pairs.sampleByKey(fractionMap);
output ← sampledPairs.map(x => x._2);
```

3 SUBJECT APPLICATION

We now revisit Search-AC and demonstrate how the elements in Section 2 apply to this heavily-used application. Such Search-AC systems are commonly available and the interested reader may

easily find details of their design and implementation.[2,3,4] In this section, we first abstract a generic workflow of the Search-AC system from the end-user's perspective, and explain notions of search- and AC-conversions, which are the prerequisites for solving our test oracle problem. Next, we highlight specific functions of a geo-aware AC application that relies on its Query Interpretation (QI) sub-module to perform multiple ML tasks. Finally, we illustrate how we derive test oracles for the AC and QI SUTs and the corresponding input sets.

3.1 Search and AC Conversions

AC performs query completion in relevance search, and enhances the user experience by displaying results given only partial inputs. Figure 1(A) provides an overview of a typical user session. AC is invoked while the user is typing the query, as shown on the right branch of Figure 1(A). For each character typed, a list of suggestions is displayed in real time. The user can directly tap on a desired result among the AC suggestions. If the user selects any specific AC suggestion and interacts with it, we refer to this session as an AC-conversion session. Broadly speaking, a 'conversion' refers to any desired user interaction, and is often a key metric in most relevance search applications [30].

Instead of selecting any of the AC recommendations, the user can, at any time, switch to the 'click-search' workflow, shown on the left branch of Figure 1(A). This typically requires a user to type a full query; and click-search may return multiple results. If the user selects one of the results and follows up with positive interactions, we refer to this session as a search-conversion session.[5] Such clicks or conversions are a common industry standard used for evaluating and optimizing search engines [13].

AC-conversion sessions are often desired over search-conversion sessions because the former means the user got to their desired result faster with fewer input characters. However, compared to click-search, AC has to resolve the prefix to predict the full query (and often needs to meet more stringent latency requirements). For instance, a three character prefix: 'hot' can be interpreted in numerous ways by AC and return different acceptable rankings of multiple results.

3.2 A Geo-Aware Search-AC System

To handle the inherent ambiguity in what one should expect from AC, many AC software have gradually specialized towards custom applications and user contexts. Here, we consider a geo-aware Search-AC application that allows users to look up specific geographic (GEO) results or place-of-interest (POI) results. Meaningful interactions with a specific click-search or AC result can include continued exploration with the result, such as setting it as a destination to navigate to. Hence, the general concepts of search- and AC-conversion sessions concretely apply to this application.

This system also relies on ML to utilize the user context to 'understand' the user intent. The contexts include user preference

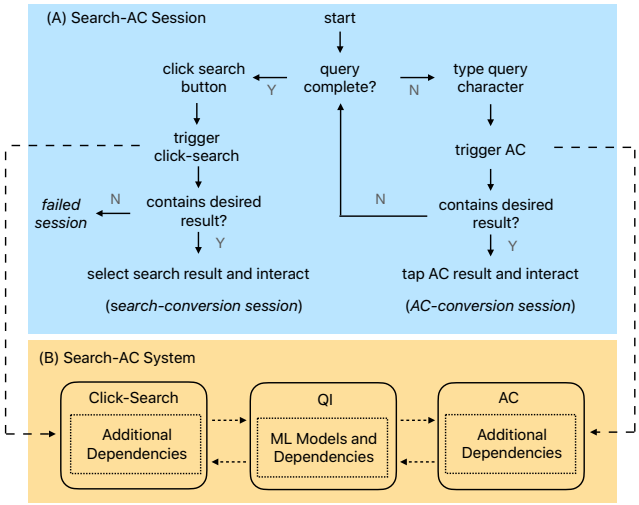


Figure 1: (A) Workflow and (B) Abstraction of the Search-AC System.

settings, the user location, and the viewport (which is the rectangular geographic region of interest to the user). Well-understood user intents can intuitively narrow down the search space and present more accurate and relevant results to the user. To achieve this, both the click-search and AC can employ core ML models that interpret the meaning of the query based on the user context as well as the way the query is typed. We refer to the modules that host such ML models as QI sub-modules, shown in Figure 1(B).

Let's revisit the earlier example of 'hot'. The geo-aware AC application can rely on popular search queries to infer that the user is searching for a 'hotel'. It can also utilize the viewport to determine if the user is looking for a hotel near their location or remotely. Next, AC can either prompt the user to trigger a click-search result with the query 'hotel' or present specific hotel POIs in the viewport. In the latter case, the user may directly navigate to the POI or book a room from a link (without having to view other results shown in click-search).

Consider another user query 'new york hot', where the user provides more text. QI can directly infer from the language pattern that the user might be searching for a hotel in New York City, and return hotels within (or near) the specific city. Alternatively, the user might intend to search for 'new york hotdogs', in which case, they would likely select POIs that serve 'hotdogs' (a food keyword) in 'new york'. Thus, AC can rely on QI to perform multiple tasks as follows.

(1) *Query Segmentation* [16]: segmenting the query, as well as inferring potential completions of the last entity, e.g., breaking the example query 'new york hot' into two entities, including segement 'new york' for the first entity, and 'hotel' and 'hotdog' as possible (completed) segments for the second entity.
(2) *Query Tagging* [37]: labeling the query segements, e.g., tagging the first entity 'new york' as a GEO type and second as POI category for 'hotel', and POI keyword for segment 'hotdog'.
(3) *Entity Recognition* [28]: fully resolving what each tagged segment specifically means, e.g., returning both the entity *New York*

[2]https://blog.google/products/search/how-google-autocomplete-works-search/
[3]https://www.algolia.com/blog/ux/what-are-predictive-search-and-autocomplete/
[4]https://baymard.com/blog/autocomplete-design
[5]In some cases, AC may complete the query and trigger click-search, but the user still needs to select results presented by click-search.

State or the entity *New York City* for possible interpretations of 'new york', and the unique entity identifiers for 'hotel' and 'hotdog' respectively.

For query tagging and entity recognition, QI can utilize indices which contain the mapping from segments to the entities (and tags), or the reverse map of different ways users refer to the same entity, e.g., the city of New York can be referred as 'nyc', 'new york', 'big apple', etc. In this example, QI outputs four distinct interpretations (as each entity yields two different interpretations) to AC. AC then relies on its own dependencies to retrieve specific results (e.g., hotel or restaurant POIs) before pruning and ranking the suggestions presented to the user.

3.3 Test Oracles for AC and QI

Next, we provide details of how we adopt production data to derive test oracles for both the geo-aware AC and its QI sub-component, following the strategy in Section 2.3. Each uses a different SoT (shown in Table 1) in this paper to focus on different aspects of the context-aware AC functions.

Table 1: Relevant Specification for Subject Applications. 1: Unique Places; 2: Category; 3: Brand; 4: Preposition; 5: Geo

SUT	SoT	Test Oracles	Tested Query Classes
AC	search-conversion sessions	expected place of interest (ePOI)	1_5, 2_5, 3_5, 1_4_5, 2_4_5, 3_4_5, 5_1, 5_2, 5_3.
QI	AC-conversion sessions	expected segmentation, tags, and entity ids	1, 2, 3, 1_5, 2_5, 3_5, 1_4_5, 2_4_5, 3_4_5, 5_1, 5_2, 5_3.

3.3.1 AC End-to-End Tests. When considering the test oracles for AC, we cross-reference search-conversion sessions to highlight ways to improve AC (with click-search as a reference).[6] This approach builds upon the insight that a user who applied click-search to obtain search-conversion potentially did not find satisfactory results in AC when typing the query. For such a user session, we chose the selected POI in click-search as the expected POI (ePOI), i.e., the test oracle, for AC. As the input to click-search is the full query, we can truncate it to only include a prefix of the full query as the input for the AC SUT and replay realistic user scenarios with preserved context. The truncation mechanism is application-dependent, and to test our geo-aware AC, we mainly considered a minimum number of input characters to trigger AC: we only allowed 3 characters[7] (excluding numbers and special characters) to be included for the last entity in the query. Such inputs enabled us to detect boundary conditions that could highlight new improvements if the geo-aware AC leveraged the context, despite the ambiguity in the input alone.

3.3.2 QI Component Tests. Among the experiments in this paper, we mainly use the ones pertaining to QI to demonstrate how we detect regressions and triage the root causes that impact the end-to-end AC system. Thus, we only focus on AC-conversion sessions as the SoT for QI tests. Note that QI is responsible for query understanding, so its oracles include expected segmentation, tags, and entity ids, which each correspond to its three main functions (as opposed to the ePOI oracle used for AC). These three variables are logged as intermediate outputs when an AC-conversion is generated. Not only do we have these multi-task test oracles, we also have the relevant inputs to QI, such as the query and relevant contexts, which are part of the production data. Then, we can rely on specific assertions to determine if the interpretation generated for an AC-conversion query is still returned when we compare a modified SUT vs. the one in production.

3.4 Input Space Partition for AC and QI

Here we describe the common key coverage dimension for both the geo-aware AC and its QI sub-component[8]: the query class based on user intents inferred by the ML. Each of these classes are represented by the query tag patterns that can be generated by QI, but each classified tag pattern triggers a different code logic in the end-to-end AC system.

Generally, there are single-entity queries, including: 1: unique places (e.g., 'the Hollywood Sign'); 2: category (e.g., 'hotels'); and 3: brand (e.g., 'starbucks'), and others (not shown in this paper) that cover the entire intent space for AC and QI. There are also multi-entity queries which are ordered combinations of single entities, such as location followed by a category (e.g., 'new york hotel'), denoted as 5_2, where 5 represents a geo/location entity, and ones with prepositions (e.g., 'starbucks in new york'), denoted as 3_4_5, where 4 represents a proposition entity, and additional combinations supported by the geo-aware AC. In this paper, we limit the scope of query classes to those shown in Table 1 for the two SUTs.

Note that all the AC query classes contain both POI (tagged by 1, 2 or 3) and GEO (tagged by 5) entities. We refer to such inputs as 'mixed intent queries' (MI queries). As opposed to single-entity queries, these queries have specific ePOIs test oracles associated with the POI component but are constrained by the GEO component, and thus, often require special treatment by the ML or dedicated ML modules. The query classes for QI are a superset of those for AC, as we will showcase how we use QI to detect regressions outside of mixed intent queries. For both QI and AC experiments, the inputs for each SUT were sampled based on production data stratified by the query classes described in Table 1 according to Algorithm 1.

4 HYPOTHESIS AND RESULTS

Following the defect class model and holistic metrics defined in Section 2, and the subject application of the Geo-aware Search-AC system described in Section 3, we present our hypotheses and experiment results as follows.

[6] We also use AC-conversion sessions to ensure queries that previously generated AC-conversions can continue to present the desired result to the user, but we omit the discussion and the results from this paper as this approach is used for catching regressions similar to those found by QI testing here.

[7] For the specific geo-aware AC system, requiring at least 3 characters moderately limits the number of results that match the query prefix, while we can still expect the system to interpret the query under a reasonable level of ambiguity.

[8] Additional dimensions specific to functionalities for AC and QI are described in specific experiments in Section 4.

4.1 Hypothesis 1: Our Strategy Captures Holistic Improvements and Regressions of ML-Driven Software with Large-Scale Inputs

Metrics: We first wanted to evaluate if our holistic measures can determine if a new software change leads to improvement or regression on specific functional metrics. To demonstrate with our geo-aware AC (the application described in Section 3.2), we considered two different metrics: (i) the *suggestion rate*, which is the fraction of tests that return at least one suggestion, and (ii) the *recall rate*, which is the fraction of tests whose suggestions contain the search-conversion ePOIs (the oracle defined in Section 3.3). We tested the software on a targeted input set of MI queries (mixed intent queries, one of the more complex query classes described in Section 3.4), as they are inherently complicated to evaluate, and hence lend themselves to automated analysis.

Procedure: We considered three real MI feature modifications (MOD-1, MOD-2, MOD-3), and compared them to their respective baselines using a targeted test set of over 300,000 MI queries. The queries were obtained by our stratified approach in Section 2.4, such that each of the 9 MI patterns contain over 30,000 test cases. For each modification (to be compared with its baseline), we calculated the suggestion rate difference and the recall rate difference, respectively based on the formula in Eq. (2). Meanwhile, each modification was independently evaluated by online user traffic in A/B experiments. If a modification generated more AC-conversions compared to the baseline, it would be accepted and merged into the new baseline; otherwise, it would be rejected.

Results: Table 2 shows the differences between the modified version (SUT-*B*) and its baseline version (SUT-*A*) for each metric (suggestion rate difference and recall rate difference), along with the A/B experiment outcome. The fractions are converted to percentages for display purpose, and the sign of the values correspond to negative and positive outcomes. The modification that contains negative values in our targeted test suite coincides with the outcome of the independent A/B experiments.

Table 2: Relative Changes for Different Modifications

	Sugg. Rate Difference	Recall Rate Difference	A/B Exp. Outcome
MOD-1	-5.57%	-5.1%	Rejected
MOD-2	-0.78%	1.9%	Rejected
MOD-3	0.09%	0.1%	Accepted

Discussion: Both MOD-1 and MOD-2 were rejected modifications, which attempted to improve the recall rate on MI queries. Both suffered in the experiments as more users were seeing no results in A/B experiments. The root causes agreed with the negative suggestion rate difference determined from our test suite. Interestingly, when we observed the ML improvements in MOD-2 via the increased recall rate on some queries, we were also able to catch emergent timeout issues that led to no results on other queries, which ended up having a stronger impact on the user, and led to experiment to be rejected. For MOD-3, we observed alignment in the positive outcomes between our tests and the experiments, even though the scale of the improvement does not directly correspond to the increase in AC-conversion rate measured via online experiments.

4.2 Hypothesis 2: Defect Class Analysis Identifies Systematic Issues Instead of Individual Failures for Holistic Improvement

Metrics: We investigated the current production baseline to determine if there are other areas that could be improved for AC MI queries. Here, we consider not only the recall rate but also the ranking behavior. Given a test set X_j on a specific class of inputs, let q_j be the number of tests that recall any POIs, and r_j^k be the number of tests that recall the search-conversion ePOI among the top-k suggestions. The ranking precision is defined as r_j^k/q_j if $q_j > 0$, and if $q_j = 0$, the precision is set to 0. For the following analyses, we set $k = 6$ as it corresponds to the number of suggestions presented to the user which does not require additional scrolling actions in the AC system we studied. In addition to the coverage dimension of intent types (described in Section 3.4), we also include a second coverage dimension: a distance dimension measured by the average travel time from the viewport centers to the ePOIs. We selected this dimension because this distance strongly influences the ranking due to a mix of rule-based and ML-based logic in the SUT. The distance dimension is split into 7 sub-categories based on 10-min, 30-min, 2-hr, 10-hr, or 1-day driving time as thresholds to reflect the spectrum of user scenarios.

Procedure: Taking the same target set of 300,000 MI queries, we ran all the test cases against the production version of the software to perform defect class analysis (described in Section 2.2): we partitioned the inputs into 9 subsets based on the intent patterns (shown in Table 1 for the AC SUT) to compute the recall rate and ranking precision for each subset based on Eq. (1), and compared which pass rates were systematically lower than the rest for each metric. For each of the 9 subsets, we further partitioned the queries based on the distance dimension and repeated the computation of the same metrics for each sub-partition of inputs.

Results: We visualize the 9 query patterns in the full MI input space with a 3×3 grid in Figure 2, where each row corresponds to an ePOI subtype: unique places, category, or brand, and each column corresponds to a specific query composition: location modifiers in the suffix (location suffix), after preposition (preposition), or in the prefix (location prefix). The values in the heatmap were initially recall rates, but we scaled and centered the values around 0 to highlight the contrast across different query patterns, to distinguish the ones below average and the ones above average. After normalization, we also use the color spectrum from red to green to indicate the relative score from worst to best in Figure 2 and subsequent heatmaps.

In Figure 3, each of the 9 cells were further divided into 7 subcategories based on the distance dimension. Each value in the heatmap corresponds to the ranking precision, which are then scaled and centered like Figure 2.

Discussion: Based on the results, we identified defect classes that were relatively worse than average.

1) *Category queries with prepositions and locations*: This class has the lowest score in the center cell of Figure 2. It indicated that prepositions can often confuse AC when it is followed by incomplete location modifiers because ones like 'on', 'in' can also

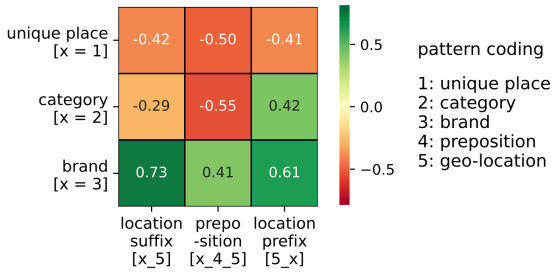


Figure 2: The heatmap displays relative recall rates for the 9 different query patterns among 300,000 MI queries for a production SUT. Rows correspond to different POI subtypes, and columns correspond to different query compositions. Values are centered at 0 and scaled to magnitude 1. Red represents overall worse cases and green represents overall better cases.

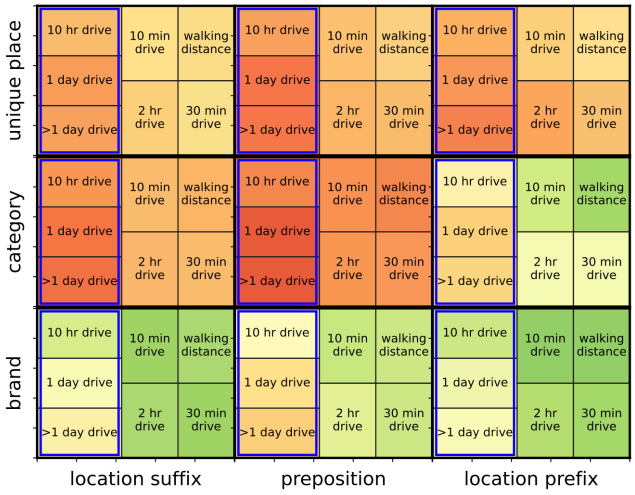


Figure 3: This heatmap contains the identical 9 query patterns on the same MI queries in Figure 2, but further subdivides each cell into 7 distance categories. The values in this heatmap are the relative ranking precision (instead of the recall rate), but each value is centered, scaled, and colored according to Figure 2: the spectrum from red to green also indicates the relative score from worst to best.

be seen as abbreviations of entities that are part of an address; and ones like 'at', 'by' are often followed by specific types of queries.

2) *Queries with unique places*: These correspond to any MI queries with unique places, shown the top row in Figure 2, such as 'hollywood sign california', 'disney land in florida', and 'washington park bellevue'. With the complete query, a user was able to directly see and select the result from click-search, but the AC failed to recall the result. Our metrics quantified the gap between unique places and other query classes (category and brand), and suggested that there is still potential for the system to leverage the uniqueness of these entities to disambiguate the location modifiers.

3) *Users with distant viewports*: For the cases where the distance was more than a 10-hour drive (highlighted by the blue bounding boxes in Figure 3), we observed that POIs in the viewport were returned and ranked much higher frequently. This means that the system has a higher tendency to favor the viewport over the geo-related text hints in the MI query. Consequently, the ePOI closer to the user location was not ranked high enough, compared to other POIs. This is a more complex behavior where the user intent can be ambiguous (because they can either prefer something close-by or in a remote view port), and the ranker needs to discern the intent based on additional query or contextual features.

In comparison, our previous test suites, which relied on ad hoc test scenarios and individual pass/fail use cases, were unable to identify these defect classes because (1) the tests lacked holistic measures for proper comparisons across sub-groups, and (2) they did not contain the scale of inputs to obtain sufficient number of cases for each defect class.

4.3 Hypothesis 3: Our Strategy Applies to Sub-Component Testing and Can Drive Triaging of New Regressions

Metrics: We also considered how testing subcomponents that perform specialized tasks can accelerate the triage process with our holistic approach. The QI subcomponent relies on its own ML models, and performs simultaneous query tagging, completion and entity recognition (defined in Section 3.2). Thus, we designed the following three recall-based metrics based on the three corresponding QI-specific oracles (described in Section 3.3):

1) *Tagging*: whether there is an interpretation whose entity tags (corresponding to the generic classification of the segmented query) equal those in the oracle.

2) *Completion*: whether there is an interpretation whose segments (where each can contain query rewrites, and last one can contain any necessary completion) equal those in the oracle.

3) *Resolution*: whether there is an interpretation whose entity ids (corresponding to a specific entity in the database, e.g., a specific brand or category) equal those in the oracle.

Procedure: To guarantee the functionality of QI and focus on catching regressions, we partitioned and stratified the inputs according to the derived intent patterns from AC-conversion data (shown in Table 1 for the QI SUT). The resulting data covered all supported intent patterns with at least 10,000 queries and had been running in all testing environments comparing every new SUT with the current SUT in production. For each intent pattern, we required all three metrics, computed based on the holistic difference in Eq. (2), to be greater than -1% (instead of 0%) for each evaluation. We allow for this 1% margin to account for minor variations in the production data such as changing business names that can lead to a small number (false positive) failures.

Results: Over the course of a few months after it was developed, the QI test suite caught multiple regressions and blocked them from being released to production. Here we highlight two real scenarios that were identified (and we only show the query classes that are relevant to the regressions we demonstrate). Each row corresponds to a different metric, and each column corresponds

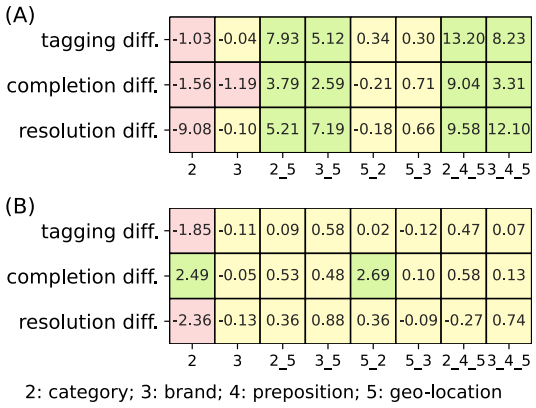


Figure 4: The heatmaps display two modifications: (A) and (B) respectively. Each modification is compared with its baseline counterpart in production over QI-specific functional metrics (rows) and query classes (columns). The values of each cell correspond to the percentage difference between the modification and the baseline based on Eq. (2).

to a different intent pattern. The highlighted values indicate the relative regressions (red) and improvements (green).

Discussion: First, we identified a candidate generation artifact corresponding to Figure 4(A), which improved MI performance, but caused regressions on simpler queries, especially category queries. The intended improvement aimed at increasing the category and brand recalls of specific businesses, but ended up creating an increasing number of noisy entity ids for the simple queries. This artifact overrode the ones that match the oracles especially for category (tag: 2) queries, resulting in -9% difference in the resolution metric. Additionally, those that yielded an extreme number of entity ids ended up not returning any interpretations at all (captured by $> 1\%$ in both tag and completion difference metrics).

Second, corresponding to Figure 4(B), we caught a data change that removed noisy tokens (e.g., those with misspells) such that QI could reduce the occurrence of unintuitive query completions. Indeed, we noticed completion improvements in two classes of queries: category (tag: 2) and location + category (tag: 5_2). However, many of the entity ids and tags (which were originally mapped to the removed tokens) could no longer be retrieved, resulting in a loss in resolution and tagging in the same query class. This blocked the particular release sign-off and led to a fix.

5 GENERAL METHODOLOGY

Even though we used the Search-AC application as our running subject, we have identified opportunities to apply similar methodologies to other applications of ML-based software.

One of our recent works [45] offers an example of an ML-based spelling correction software, where the test oracles can be automatically derived from production data. Specific to the spell correction functionality, the test dimensions were captured by edit-distance-based input typo spaces, and the functional evaluation relied on recall and ranking measures. We note that these measures are special cases of the holistic metrics described in our current work, and hence, the defect class analysis applied to the Search-AC application should also be able to systematically identify defects, including the ones reported in the original paper.

Taken together, both subject applications prescribe a general methodology that applies to testing the emergent behavior of generic ML software.

The methodology consists of the following steps:

(1) Define the functional behavior of the SUTs, including input-output relations that capture ML behavior.
(2) Determine the Source of Truth (SoT) that can determine the expected outputs (i.e., the test oracle).
(3) Specify coverage dimensions that reflect core functions and sample test cases according to the dimensions.
(4) Evaluate the SUT based on the overall holistic metric to determine the degree of improvement or regression.
(5) Repeat the evaluation of the metrics on sub-dimensional inputs and determine if there are defect classes.

It is crucial in Step 1 to distinguish the black-box software behavior from the internal ML tasks, so that the tests reflect the integrative influence of the ML (or multiple MLs). For instance, consider a speech recognition software that performs three functions: speaker identity recognition, language detection, and transcript generation. The black-box input we consider could be an audio recording along with user configurations, and the output could be the transcribed text along with interpretations. On the contrary, an internal ML may only rely on 'cleaner' pre-processed data for its task and the output from the ML needs to be post-processed based on other configurations.

Solving the oracle problem in Step 2 requires an SoT that contains the content needed for functional assertions. For the speech recognition software, this means knowing the expected outputs associated with each individual function, such as how the user is addressed, the desired displayed language, and the output text with appropriate highlighting. The testers can take the user feedback as the SoT to extract these outputs validated by the user either from explicit rating by the user, or from positive subsequent interactions captured by the software. There are generally two sources of oracles: in-house evaluation and crowdsourcing, which mirrors the process of data labeling for ML. This example and our proposed solution in Section 2.3 both fall under crowdsourcing from production data, a type of semi-automated labeling [10]. However, if production data is not available, one can still derive the oracle from the existing SoT dataset being served, e.g., all entries in a knowledge graph, since they can be converted into some form of expectation at the software output.

Specifying coverage dimensions in Step 3 is a common procedure in traditional software testing which ensures sufficient use cases to cover all software functions. This notion applies to our framework as well. However, because the space of inputs (e.g., text of audio inputs) cannot be fully enumerated, we resort to stratified sampling on each key dimension in Section 2.4 to ensure sufficient representation within each dimension. The reasoning follows from label-balancing used for ML training [15]. For instance, to ensure all the languages supported by the speech recognition software are

covered, the tester can define 'language' as an attribute of each test case, whose value is derived from the user preference or from the output language (with positive feedback), and then balance the test cases along this dimension according to Algorithm 1.

Once test cases are generated, Steps 4 and 5 allow the tester to evaluate the software systematically: they can rely on the holistic metric in Section 2.1 to evaluate a modification compared to a baseline to see if the change incurs an improvement or regression not only for each of the functional metrics but also for each subspace within each dimension for defect class analysis as explained in Section 2.2. With the speech recognition software for example, one can rely on the specific metrics to answer different questions about a modification: e.g., whether there is correlated improvements in all three functionalities; whether there is a trade-off between language detection and others; whether a regression in language detection is specific to only a subset of languages. Taken together, these signals can allow the testers and developers to understand how to improve the software and prioritize the modifications systematically for timely evaluation [36].

6 RELATED WORK

Our work builds upon a rich body of literature involving automating software testing at scale and addressing challenges of ML software testing. Here we discuss prior work on testing IR and ML systems.

Evaluation of search and similar IR systems (e.g., other variations of Search-AC systems) has been an active research area in automated software testing. Several works have proposed user-driven black-box testing approaches to solve the test generation and oracle problems that we also faced for their applications: Zhou et al. [44] adopted metamorphic testing to create tests for web search engines; Olteanu et al [29] used crowdsourcing to identify, categorize and understand problematic search completion suggestions. In the work of Shokouhi et al [33], the researchers designed a time-sensitive query autocomplete service, and evaluated it against query sets with daily, weekly, monthly and annual cycles. While these works focus on specific classes of inputs that are difficult to evaluate in search systems, our case studies take a global view of ML-IR systems based on our holistic metrics to determine how functional regressions need to be fixed and provide patterns of defects to prioritize triaging. Our results for AC and QI show that our strategy can adapt to complex ML-IR systems by considering several regression metrics in a multi-dimensional input space.

Assuring the functionality and quality of ML-based systems has also been proven complex and challenging by previous research. Zhang et al. [41] surveyed a comprehensive landscape of ML testing, summarizing topics in robustness, privacy, efficiency, and fairness. Lewis et al. [22] studied faults caused by mismatches among data science, software engineering, and operations in ML-enabled systems. Santhanam et al. [31] and Devanbu et al. [11] discussed the challenges of assessing deep learning systems. Other reviews have emphasized specific aspects of evaluating ML systems, such as defect detection in ML programs [2] or implementation quality [25].

Practitioners in industry have also echoed the level of complexity of testing ML systems. Lwakatare et al. [23] reviewed challenges and solutions of testing large-scale ML systems. Amershi et al. [1] conducted a study over the development workflows of multiple ML-based systems in Microsoft, and emphasized the importance of continuously evaluating the both the ML and non-ML parts of ML-based systems end-to-end. To detect faults in neural machine translation systems, Zheng, et al. [43] derived properties between input and output which can be checked systematically, and applied it to test WeChat (a messenger app with over one billion monthly active users). Chen et al. [4, 5] focused on continuous triaging of large-scale online service systems. Washizaki et al. [38] surveyed good/bad software-engineering design patterns for ML techniques in practice. Chen et al. [6] studied the challenges in the deployment process of ML systems with a focus on Deep Learning based software. Even though these approaches do not prescribe a generic system testing workflow like ours, they pose the multitude of existing industrial challenges, including scalability, applicability, and acceptability, and inspired us to propose our practical strategy.

7 CONCLUSION

As ML-IR systems get increasingly complex, consisting of multiple ML models and non-ML decision-making code, they continue to challenge conventional software testing. We presented a new approach to test such systems, thereby addressing these challenges, by: (1) leveraging functional test cases to inform ML-IR developers holistically about the impact (regression/improvement) of their latest code and data modifications, (2) measuring system regression at the level of a defect class – a partition of the input space on which the ML-IR software does measurably worse for an existing/new feature, and (3) drilling down deeper to sub-components to understand the causes of regressions. Our deployment on a production Search-AC system demonstrated the effectiveness of our new approach by successfully detecting multiple regressions, and identifying defect classes at the system as well as sub-system level to ease debugging.

Meanwhile, defining holistic metrics from individual functional test failures, together with identification of defect classes, is a powerful and general approach for continuous iterative improvement of any ML system. In the near future, we plan to study how defect classes evolve in AC and QI over time as new features are added, systems are re-designed, and new usage patterns start emerging. In the medium term, we will apply the same approach to other ML systems, including ones for which we do not have access to production data, and hence we will need to develop test cases using other generative models. In the long term we plan to build frameworks to generalize this approach, understand its strengths and weaknesses, and implement general tools that others can leverage to test their ML systems.

ACKNOWLEDGMENT

We would like to thank Alex Braunstein for constant support of our work, Emily Kowalczyk for feedback on the manuscript, and Archana Bhattarai for sharing insights into our subject application.

REFERENCES

[1] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[2] Houssem Ben Braiek and Foutse Khomh. 2018. On testing machine learning programs. *arXiv preprint arXiv:1812.02257* (2018).

[3] Hsinchun Chen. 1995. Machine learning for information retrieval: Neural networks, symbolic learning, and genetic algorithms. *Journal of the American society for Information Science* 46, 3 (1995), 194–216.

[4] Junjie Chen, Xiaoting He, Qingwei Lin, Hongyu Zhang, Dan Hao, Feng Gao, Zhangwei Xu, Yingnong Dang, and Dongmei Zhang. 2019. Continuous Incident Triage for Large-Scale Online Service Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 364–375. https://doi.org/10.1109/ASE.2019.00042

[5] Yujun Chen, Xian Yang, Hang Dong, Xiaoting He, Hongyu Zhang, Qingwei Lin, Junjie Chen, Pu Zhao, Yu Kang, Feng Gao, Zhangwei Xu, and Dongmei Zhang. 2020. *Identifying Linked Incidents in Large-Scale Online Service Systems*. Association for Computing Machinery, New York, NY, USA, 304–314. https://doi.org/10.1145/3368089.3409768

[6] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A Comprehensive Study on Challenges in Deploying Deep Learning Based Software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 750–762. https://doi.org/10.1145/3368089.3409759

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113. https://doi.org/10.1145/1327452.1327492

[8] Li Deng and Dong Yu. 2014. Deep learning: methods and applications. *Foundations and trends in signal processing* 7, 3–4 (2014), 197–387. https://doi.org/10.1561/2000000039

[9] Leon Derczynski. 2016. Complementarity, F-score, and NLP Evaluation. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*. European Language Resources Association (ELRA), Portorož, Slovenia, 261–266.

[10] Michael Desmond, Evelyn Duesterwald, Kristina Brimijoin, Michelle Brachman, and Qian Pan. 2021. Semi-Automated Data Labeling. In *Proceedings of the NeurIPS 2020 Competition and Demonstration Track (Proceedings of Machine Learning Research, Vol. 133)*, Hugo Jair Escalante and Katja Hofmann (Eds.). PMLR, 156–169. https://proceedings.mlr.press/v133/desmond21a.html

[11] Prem Devanbu, Matthew B. Dwyer, Sebastian G. Elbaum, Michael Lowry, Kevin Moran, Denys Poshyvanyk, Baishakhi Ray, Rishabh Singh, and Xiangyu Zhang. 2020. Deep Learning & Software Engineering: State of Research and Future Directions. *CoRR* abs/2009.08525 (2020). https://doi.org/10.48550/ARXIV.2009.08525

[12] Daniel Di Nardo, Nadia Alshahwan, Lionel C Briand, Elizabeta Fourneret, Tomislav Nakić-Alfirević, and Vincent Masquelier. 2013. Model based test validation and oracles for data acquisition systems. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 540–550. https://doi.org/10.1109/ASE.2013.6693111

[13] Georges E. Dupret and Benjamin Piwowarski. 2008. A User Browsing Model to Predict Search Engine Click Data from Past Observations.. In *Proceedings of SIGIR '08* (Singapore, Singapore). Association for Computing Machinery, New York, NY, USA, 331–338. https://doi.org/10.1145/1390334.1390392

[14] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. 211–224. https://doi.org/10.1145/3395363.3397366

[15] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2001. *The elements of statistical learning*. Vol. 1. Springer series in statistics Springer, Berlin. https://doi.org/10.1007/978-0-387-84858-7

[16] Matthias Hagen, Martin Potthast, Benno Stein, and Christof Bräutigam. 2011. Query Segmentation Revisited. In *Proceedings of the 20th International Conference on World Wide Web*. 97–106. https://doi.org/10.1145/1963405.1963423

[17] Katja Hofmann, Lihong Li, and Filip Radlinski. 2016. Online Evaluation for Information Retrieval. *Found. Trends Inf. Retr.* 10, 1 (jun 2016), 1–117. https://doi.org/10.1561/1500000051

[18] Max Hort, Jie M. Zhang, Federica Sarro, and Mark Harman. 2021. Fairea: A Model Behaviour Mutation Approach to Benchmarking Bias Mitigation Methods. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. 994–1006. https://doi.org/10.1145/3468264.3468565

[19] Pao-Lu Hsu and Herbert Robbins. 1947. Complete convergence and the law of large numbers. *Proceedings of the National Academy of Sciences of the United States of America* 33, 2 (1947), 25.

[20] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A Large-Scale Longitudinal Study of Flaky Tests. *Proc. ACM Program. Lang.* OOPSLA, Article 202 (nov 2020), 29 pages. https://doi.org/10.1145/3428270

[21] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2020. *Mining of massive data sets*. Cambridge university press.

[22] Grace A. Lewis, Stephany Bellomo, and Ipek Ozkaya. 2021. Characterizing and Detecting Mismatch in Machine-Learning-Enabled Systems. *CoRR* abs/2103.14101 (2021). https://doi.org/10.48550/arXiv.2103.14101

[23] Lucy Ellen Lwakatare, Aiswarya Raj, Ivica Crnkovic, Jan Bosch, and Helena Holmström Olsson. 2020. Large-scale machine learning systems in real-world industrial settings: A review of challenges and solutions. *Information and Software Technology* 127 (2020), 106368. https://doi.org/10.1016/j.infsof.2020.106368

[24] Dusica Marijan, Arnaud Gotlieb, and Mohit Kumar Ahuja. 2019. Challenges of testing machine learning based systems. In *AITest*. IEEE, 101–102. https://doi.org/10.1109/AITest.2019.00010

[25] S Masuda, K Ono, T Yasue, and N Hosokawa. 2018. A Survey of Software Quality for Machine Learning Applications. In *ICST*. 279–284. https://doi.org/10.1109/ICSTW.2018.00061

[26] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.

[27] Ph Mylonas, David Vallet, Pablo Castells, Miriam Fernández, and Yannis Avrithis. 2008. Personalized information retrieval based on context and ontological knowledge. *The Knowledge Engineering Review* 23, 1 (2008), 73–100. https://doi.org/10.1017/S0269888907001282

[28] David Nadeau and Satoshi Sekine. 2007. A survey of named entity recognition and classification. *Lingvisticae Investigationes* 30, 1 (2007), 3–26. https://doi.org/10.48550/ARXIV.2109.11406

[29] Alexandra Olteanu, Fernando Diaz, and Gabriella Kazai. 2020. When Are Search Completion Suggestions Problematic?. In *Computer Supported Collaborative Work and Social Computing (CSCW)*. ACM. https://doi.org/10.1145/3415242

[30] Khalid Saleh and Ayat Shukairy. 2010. *Conversion optimization: The art and science of converting prospects to customers*. " O'Reilly Media, Inc.".

[31] P. Santhanam, Eitan Farchi, and Victor Pankratius. 2019. Engineering Reliable Deep Learning Systems. *CoRR* abs/1910.12582 (2019). https://doi.org/10.48550/ARXIV.1910.12582

[32] Tefko Saracevic. 1995. Evaluation of evaluation in information retrieval. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*. 138–146. https://doi.org/10.1145/215206.215351

[33] Milad Shokouhi and Kira Radinsky. 2012. Time-Sensitive Query Auto-Completion. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Portland, Oregon, USA) *(SIGIR '12)*. Association for Computing Machinery, New York, NY, USA, 601–610. https://doi.org/10.1145/2348283.2348364

[34] Dan Siroker and Pete Koomen. 2013. *A/B testing: The most powerful way to turn clicks into customers*. John Wiley & Sons.

[35] Foivos Tsimpourlas, Ajitha Rajan, and Miltiadis Allamanis. 2021. Supervised learning over test executions as a test oracle. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 1521–1531. https://doi.org/10.1145/3412841.3442027

[36] Kristen R Walcott, Mary Lou Soffa, Gregory M Kapfhammer, and Robert S Roos. 2006. Timeaware test suite prioritization. In *Proceedings of the 2006 international symposium on Software testing and analysis*. 1–12. https://doi.org/10.1145/1146238.1146240

[37] Xuanhui Wang. 2020. Query Segmentation and Tagging. In *Query Understanding for Search Engines*. Springer, 43–67.

[38] Hironori Washizaki, Hiromu Uchida, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2019. Studying Software Engineering Patterns for Designing Machine Learning Systems. In *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. 49–495. https://doi.org/10.1109/IWESEP49350.2019.00017

[39] Elaine J Weyuker. 1982. On testing non-testable programs. *Comput. J.* 25, 4 (1982), 465–470.

[40] Qing Xie and Atif M. Memon. 2007. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.* 16, 1 (2007), 4. https://doi.org/10.1145/1189748.1189752

[41] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE TSE* (2020). https://doi.org/10.1109/TSE.2019.2962027

[42] Wei Zheng, Guoliang Liu, Manqing Zhang, Xiang Chen, and Wenqiao Zhao. 2021. Research Progress of Flaky Tests. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 639–646. https://doi.org/10.1109/SANER50967.2021.00081

[43] Wujie Zheng, Wenyu Wang, Dian Liu, Changrong Zhang, Qinsong Zeng, Yuetang Deng, Wei Yang, Pinjia He, and Tao Xie. 2019. Testing Untestable Neural Machine Translation: An Industrial Case. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 314–315. https://doi.org/10.1109/ICSE-Companion.2019.00131

[44] Zhi Quan Zhou, Shaowen Xiang, and Tsong Yueh Chen. 2016. Metamorphic Testing for Software Quality Assessment: A Study of Search Engines. *IEEE Transactions on Software Engineering* 42, 3 (2016), 264–284. https://doi.org/10.1109/TSE.2015.2478001

[45] Junjie Zhu, Teng Long, and Atif Memon. 2021. Automatically authoring regression tests for machine-learning based systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 374–383. https://doi.org/10.1109/ICSE-SEIP52600.2021.00049