AI for Software Testing

Jason Arbon

jarbon@gmail.com

Abstract

Software testing is fundamentally an exercise in applying sample inputs to a system, and measuring the outputs to determine the correctness of the application's behavior. This testing input and output function is very similar to the basic operations of training and executing modern Artificial Intelligence (AI) and Machine Learning (ML) systems. The similarity implies that the field of Software Testing is ripe to benefit from recent advances in AI and ML.

Software Testing and approaches to measure Software Quality have been relatively stagnant over the past decade or so, with only slight technical improvements in test frameworks and methodologies; meanwhile product development, DevOps, and deployment have benefited greatly from increasing availability of compute, network, and storage. Software testing has remained an often linear activity with respect to test coverage--adding tests one at a time or in small batches while products add complexity at a much higher rate, meaning almost no product is tested efficiently. The complexity of products has been growing exponentially, where testing has remained more or less a linear activity with respect to test coverage.

Applying the advancements in AI and ML will help testing 'catch up' with development advances, increase the efficiency, speed and efficacy of software testing, and free testers from many mundane testing tasks. AI and ML, when used to abstract the test input, test execution, and test evaluation problems enable testing at scale, standardization of quality metrics, benchmarking, and a global set of reusable test cases.

Biography

Jason Arbon is the CEO of Appdiff, which is redefining how enterprises develop, test, and ship mobile apps with zero code and zero setup required. He was formerly the director of engineering and product at Applause.com/uTest.com, where he led product strategy to deliver crowdsourced testing via more than 250,000 community members and created the app store data analytics service. Jason previously held engineering leadership roles at Google and Microsoft, and coauthored "How Google Tests Software and App Quality: Secrets for Agile App Teams."

1 Introduction

The software engineering world today is busy applying Artificial Intelligence (AI) and Machine Learning (ML) to solve complex product problems for everything from self-driving cars, smart chatbots, and security. Software quality and testing efforts are also starting to benefit from the early application of AI for classic testing problems, and also waking up to a new world of software testing problems related to testing these new AI-based products.

Software Testing, especially testing applications directly from the user interface, has proven difficult and expensive. So difficult and expensive that many software test automation attempts fail to deliver much value, or fail altogether. In a world of self-driving cars and machines beating the top GO players at their own game, the best software companies and engineers still heavily rely on manual testing, and iterative deploy-identify-fix-deploy loops. Humans are still in this testing loop today because software requires the power of human brain to design, execute and validate complex software systems--but that is about to change thanks to ML.

This paper describes the need for a new approach to software testing, and the problems modern software testers face. It then gives a brief introduction to AI and ML--just enough for the purposes of this paper. Next, several examples of applying ML to generate test input and validation that are more efficient and useful than current methods will be shared. Following that, is an overview of the new software quality and testing problems created by new AI-based products, along with examples of solutions. Lastly, some thoughts on the scope and timing of the impact of these testing approaches and how the testing community can speed up the development and application of AI and ML for testing.

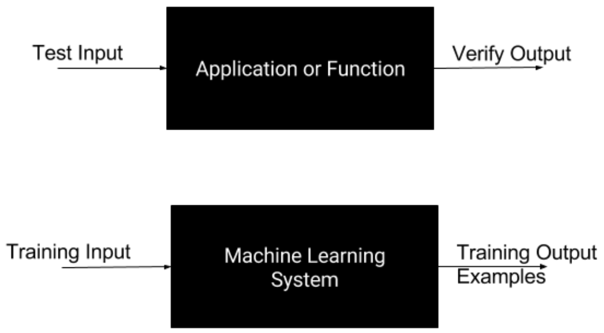
This paper strives to give both an overview of how AI and ML will impact software quality and testing efforts in the coming years, with practical real-world examples from the author's experience.

2 AI and ML Overview for Testers

Artificial Intelligence is the idea of building machines that interact with external data and behave in a cognitive way--actions that appear like human thinking. There are really two branches of AI; Artificial General Intelligence (AGI), which has the goal of building truly generalized thinking, emotive, conscious machines. And then there is the subset of application-focused AI, called Machine Learning, which is what this paper focuses on.

Generally Machine Learning (ML) is used to train a machine to perform a function. In non-AI programming, you design a function thinking of all the if-then and for loops that you need to convert the input data to the correct output data and you write that function. In ML, the idea is inverted -engineers work on gathering many examples of input and output data and show that to a machine which trains itself to build a function that satisfies the Input and Output requirements. This process is called supervised learning, as a human is showing it the correct behavior. Supervised learning is analogous to Test Driven Development (TDD) where the inputs and output tests are defined first, then humans code the functions. In this case the machines 'code' the functions.

Note that this data and training method looks very much like a manual tester, or test automation engineer's job--apply some test input data to the system, and verify:



Another variant of ML is called reinforcement-learning, where humans define a reward system to tell the machine when it has done a good or bad thing. The machine builds all sorts of different variations of itself based on this reinforcement. This is analogous to operant conditioning--similar to how people learn task proficiency

For the curious tester, they may want to know how these ML black boxes work. The ML black boxes can be any of ANNs (Artificial Neural Networks), CNNs (Convolutional Neural Networks, SVMs (State Vector Machines), Decision Trees, etc. For purposes of this paper we don't need to know the details of this ML black box, just know that we give it training data and it attempts to build a function that satisfies all the input and output tests it is trained on.

3 Need for AI in Testing

While software engineering and infrastructure has become far easier and scalable thanks to improvements in development languages, tools, processes and cloud computing - software test engineering hasn't seen similar gains in productivity, efficiency or efficacy. Software Test engineering needs similar attention and breakthroughs in technology.

UI testing frameworks Selenium, Appium, and Webdriver were created back in 2004. Most proprietary testing systems such as TestPlant and HP UFT have been around just as long, or sometimes even longer. Even recent variants such as Google's Espresso or Square's KIF test frameworks look, and are used very similar to, test frameworks from 13 years ago. Over the last 10 plus years, the familiar test automation paradigms of leveraging image identification, record and playback, or hand-coding product selectors based on DOM (Document Object Model) trees or accessibility identifiers have only incrementally improved and have been ported to more modern platforms.

The problem is that while not much has changed in testing, the world of software engineering has sped up thanks to tools and processes such as Agile, and the implementation of Continuous Engineering. which produce more code, with far higher frequency of new builds, all requiring testing.

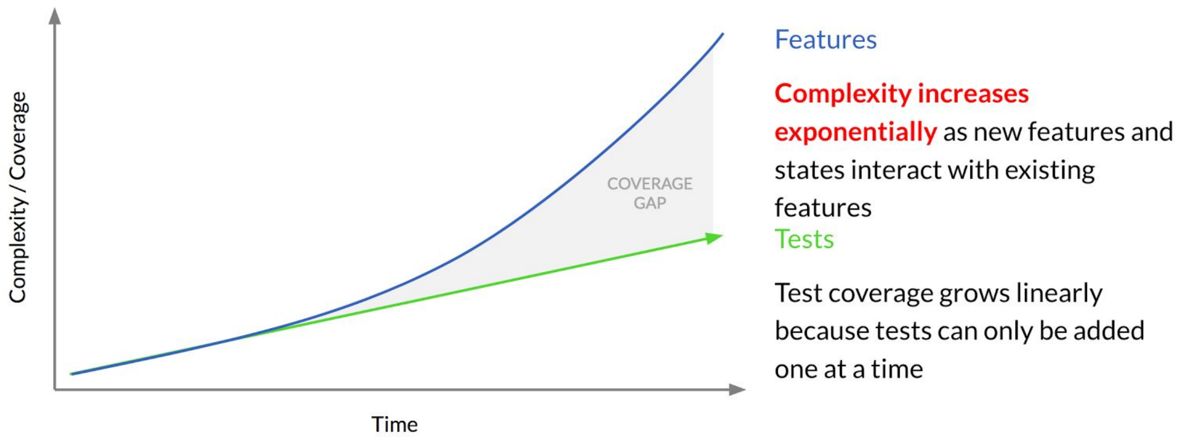
Worse, there is a fundamental disconnect between software engineering and software testing. Writing a simple function or product feature requires a fair amount of testing to verify its behavior. In high-quality applications more engineering time is spent on the testing effort versus the implementation. But, there is an additional test and validation effort as the new feature can affect the behavior of other features. Each new feature can impact some subset of other features.

$$T_{total}(f_n) = T(f_n) + \sum_{i=0}^{n-1} k_i T(f_i)$$

where T is then additional functional test engineering effort for a given feature f.

and ki is the relative risk/impact feature n has on feature i.

Thus, the testing effort for each new feature (f_n) includes the testing required to eliminate the risk to all impacted features. This results in the net testing effort to grow exponentially. Most current testing methods of adding more manual tests, or hand-crafting code for more regression scripts, only adds coverage at a rate linearly proportional to testing effort/cost. Yes, even data-driven tests, delivery linear amounts of test coverage.



This means every software product suffers from an increasing gap between the amount of complexity that needs to be tested, and the ability to add test coverage. Something has to change to enable a far greater ability to generate test coverage, and I will show that Machine Learning is the only test approach that has the possibility of such exponential test coverage.

How does a Machine Learning approach to testing result in exponential levels of test coverage?

1. ML can be trained to generate test input and test validations for arbitrary applications.
2. ML delivers the ability to execute the same test case against multiple applications.
3. ML test generation and execution can run automatically on exponentially improving storage, network, and compute in the cloud.

Additionally, as we'll see later, a common ML generated test execution environment means quality can be measured and benchmarked against other applications, improving our ability to quantify quality itself.

ML-driven test creation and execution has the possibility of closing the test coverage gap.

4 AI for Test Input and Validation

How do you teach a machine to generate test input? Today, to generate test cases, testers consider the application, and generate test inputs they expect from users to deliver positive test coverage. They then draw upon their past experience of what types of inputs might be interesting or break the application under test. We need to teach a machine to think similarly.

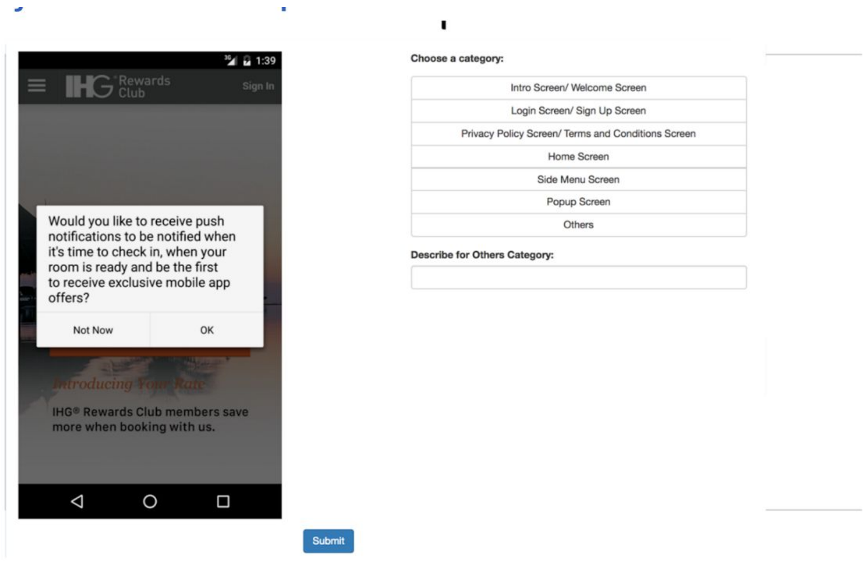
The machine needs to be taught two different things to replicate the human tester's behavior: Recognize the state of the application, and know what inputs are valid or interesting for this application state.

4.1 Training ML: Recognize Application State

Recognizing the application state can be accomplished by giving the machine many screens and labeling the screens as a type of application 'state'. For example, 'login screen', 'search results', 'permissions dialog', etc. If we can teach the machine to recognize what type of state the application is in, much like a tester recognizes the state of the application they are testing, they can intelligently pick what types of inputs to apply to the application.

First, we gather many thousands of screenshots of real world applications. We then consider all the major classes of UI state that applications have. Luckily for machine learning, most apps have similar states, which means we can have many examples to for the ML training.

Labeling is the next step. We need to have humans apply the labels to each screen. A simple application is built which shows one of these screens alongside a list of possible labels. The labeler, a human performing this labeling task, clicks the appropriate labels for each screen in sequence.



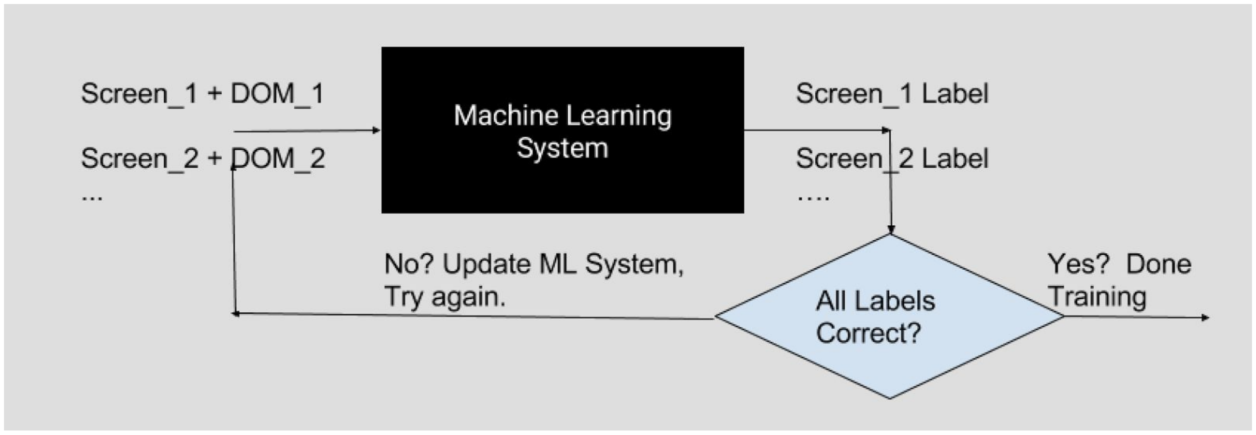
Would you like to receive push notifications to be notified when it's time to check in, when your room is ready and be the first to receive exclusive mobile app offers?

Choose a category:

- Intro Screen/ Welcome Screen
- Login Screen/ Sign Up Screen
- Privacy Policy Screen/ Terms and Conditions Screen
- Home Screen
- Side Menu Screen
- Popup Screen
- Others

Describe for Others Category:

Once all the labels have been saved, there is now a corpus of labeled data with which to train the ML. Again, the detailed mechanics of this work are beyond the scope of this paper, but suffices to say that thousands of pages are shown to the ML program, where the input is a combination of the pixels in the screenshot itself, along with the DOM elements. The training/test data is the labels we got for each screen from the humans.



ML Training Input: Image, DOM element information (e.g. <button>, <span>, etc.)

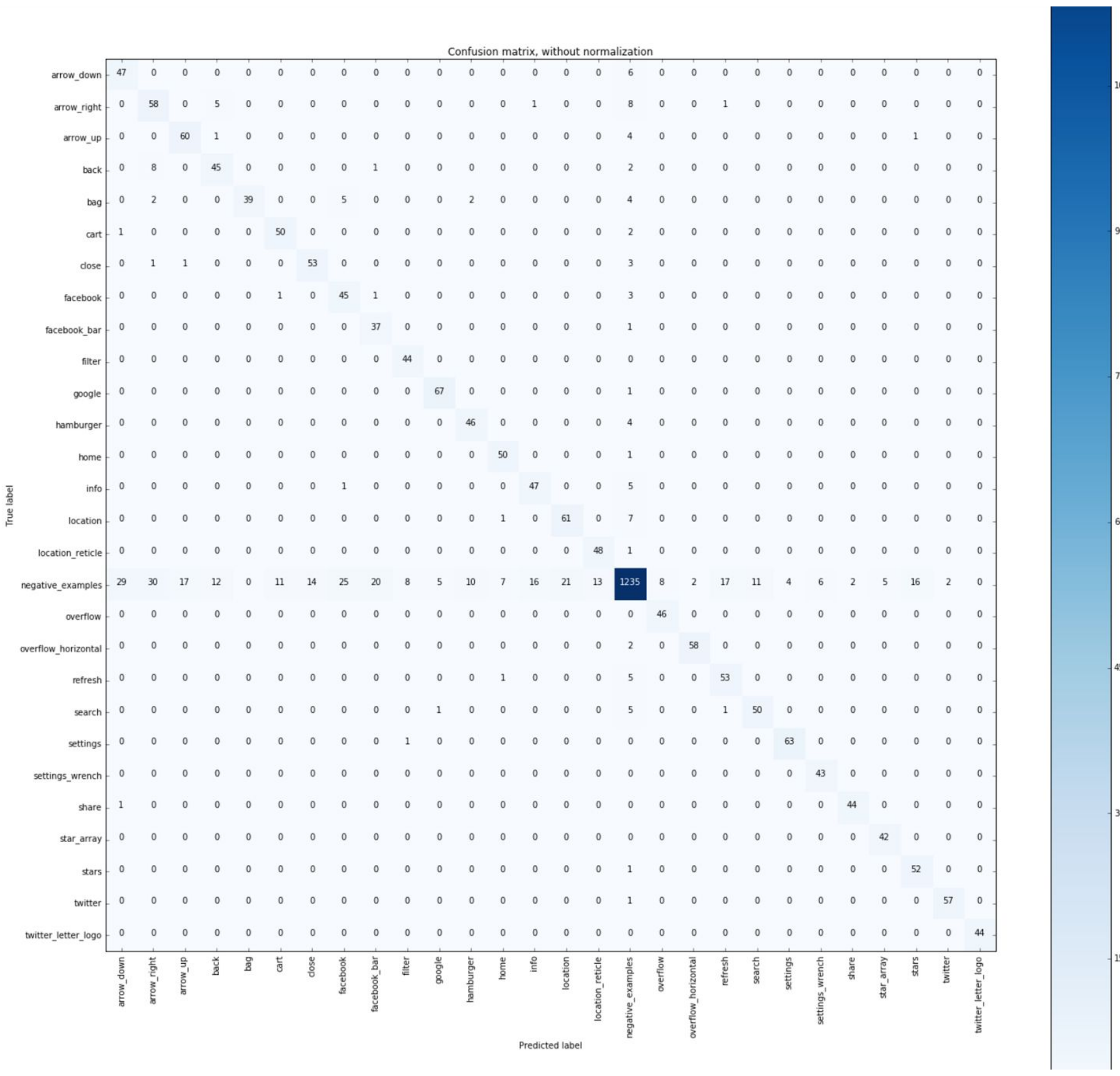
ML Training Expected Output: the correct label(s) per page from humans.

The training process can require hours of computation. With the training system showing the ML machine screenshot after screenshot, determining whether or not the ML machine successfully labeled each screen. Every time the ML gets the label for a screen wrong, it changes its internal structure, and the process is repeated until the ML does the best labeling job it can do.

It is worth noting that training data that is sparse, incorrect, or incoherent can prevent the ML from learning.

Once this process is complete, we now have an ML system that can accept a screenshot and a snapshot of the DOM of the application and not just give the correct label for the screens we have trained it on, but also on screens it has never seen before--just like the human.

When the ML is done training, the quality of the machine labeling is often visualized via a chart that is called a ‘Confusion Matrix’. The confusion matrix simply plots each screen label on each axis and shows how often one screen is confused for another--a mistaken label. The results here are from early work after training with about 6,000 screenshots. This approach works fairly well, as lighter blues and smaller numbers mean great labeling.



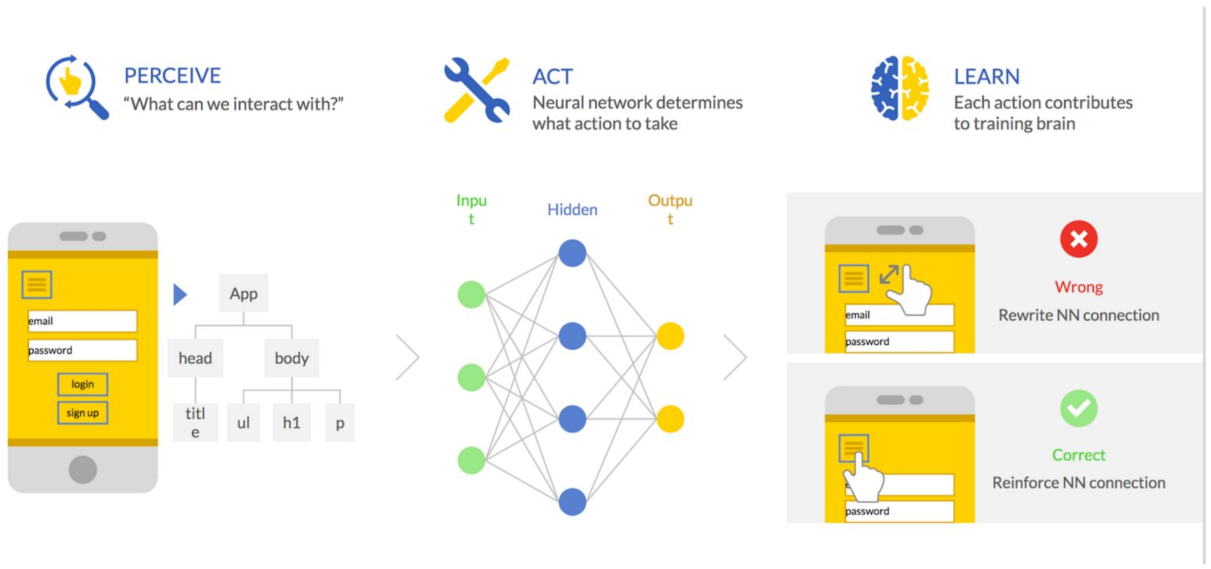
4.2 Training ML: Apply Test Inputs

The next step is to teach the ML how to decide on correct input actions to take based on what screen state the application is in. An individual action is a pair ‘element’ and ‘input’. For example, the element can be an OK button, and the input could be a ‘tap’, or an element could be a search text box, and the action could be entering the text ‘beanie babies’.

Again, we need a large amount of training data, where the input here is the set of all elements on a screen, the label of the screen, and the output is the set of

Input Training Data: [‘screen_label’:’search’, ‘elements’:’button, textbox, image’]

Output Training Data: [‘textbox’:’enter_text’, ‘button’:’tap’, …]



The learning to generate human-like actions on elements is similar to the training label classification. Screen context and a specific element are shown to the network, which guesses at a good action. If the network suggests a strange action, it reconfigures itself and tries again until it starts to behave like a human interacting with the application.

Note that this is a simple version for purposes of this paper, more complex variants deliver more 'intelligent' behaviors if inputs such as previous screens and action taken, or even what 'goal' is being attempted. A goal being a high level intent, such as: 'buy something', or 'search for something', or 'change profile image', etc.

The ML now has the ability to make human like decisions about what action to take on a screen.

It is also worth noting that a general 'fuzz' tester does very poorly when it comes to replicating human like interactions. On average each page has about 150 pairs of plausible element/action pairs. With each new page in a 'flow' of exploring the application there is a branching factor of 150 possible paths. So the total number of possible paths to explore semi-randomly is

num_paths ~= 150^X

where x=number of steps.

To cover all paths in app, say only 35 steps deep, it would require 150^35 = 2^78 test iterations to walk all possible paths.. As there are only ~2^78 atoms in the universe, one can get a feel for the amount of compute it would take a random-walk crawler to explore what an ML trained bot can do in just thousands of iterations. As the ML makes human-like decisions at every step, it prunes the total state space down to something efficient enough for practical applications.

4.3 Executing ML: Real World Apps

Now that the ML can intelligently decide what action to take given an app's state, the question of how the ML 'sees' the app and can take action in the app is worth addressing.

The ML sees, by simply having a bootstrap program that launching the app in a simulator or on a device, takes a screenshot of the application and downloads the current DOM over a debug connection to the app. This can be accomplished with off-the-shelf test automation components and utilities like android 'ADB' utility and executing a command such as 'adb shell uiautomator dump' to get the DOM in xml format, and the command 'adb shell screencap -p /sdcard/screen.png' to get a screenshot. For iOS, the XCUITest framework API allows similar functionality, more generally Appium and Selenium for the Web have this functionality as well. The application driver makes calls to the specific platform driver to request the data and simply passes the information to ML ('brain') to decide what to do next.

Now that the ML has decided what to do next, the output of the ML is a pair of element and action names. The Application driver simply finds the element's location or handle on the screen and based on the recommended action, sends the appropriate tap or text input.

It is worth noting that with only these two pieces of trained ML, we have a system capable to intelligently exploring *any* app. No new training is necessary to exercise human-like paths, and automatically discover crashes or performance issues that would previously only have been done via human manual testing, or humans writing thousands of automated UI test scripts. Best of all, this ML approach is applicable to all apps immediately--no need for humans to manually use the apps, or write test scripts for months.

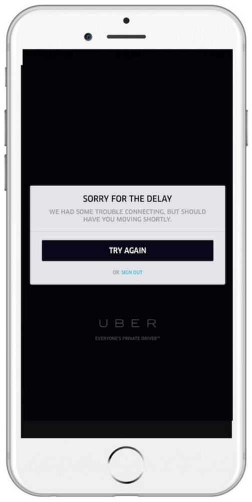
4.4 Training ML: Verifying Behavior

The ML bots can now intelligently drive the application like a human would, but what of the question of verification? How do the these ML bots know the application is behaving correctly. There are three approaches to enable the ML bots to know if the behavior is correct.

1. Automatically check for common, detectable issues.
2. Train ML on examples of bugs.
3. Human review previous flows, then bots notify of any differences in subsequent test runs.

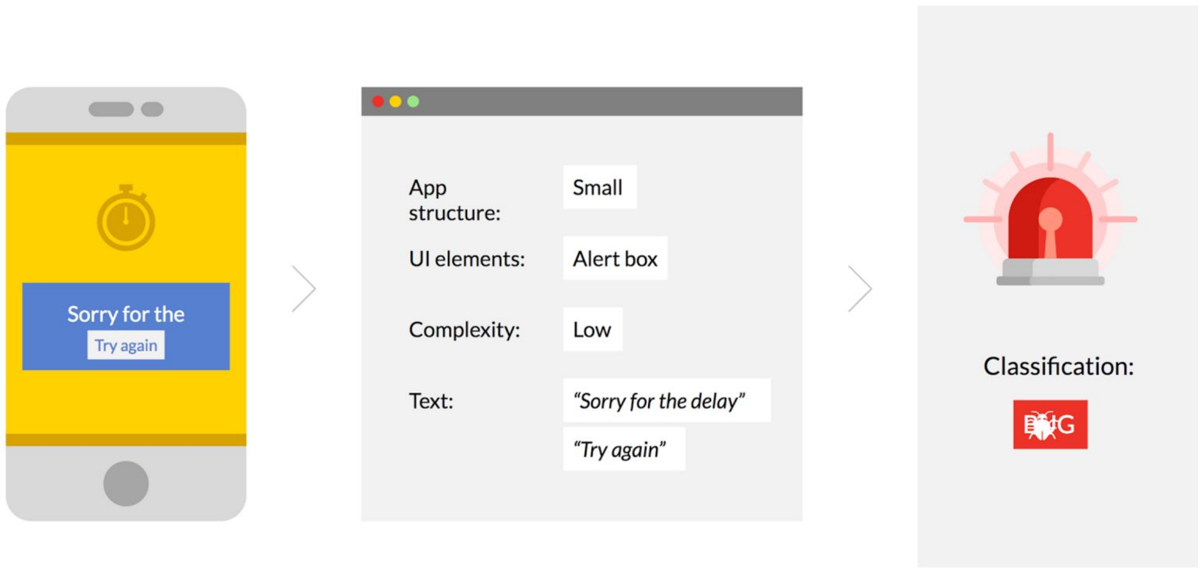
Automatically checking for common functionality detectable issues is the first line of app verification. At each step, the App Driver checks for any application crashes, error dialogs, etc.

Training a new ML on examples of previously found bugs, to auto classify the next screen in a sequence as a bug. This works for common failures where there are many examples to train on (for example error dialogs) like this one from the UBER app.


SORRY FOR THE DELAY

WE HAD SOME TROUBLE CONNECTING, BUT SHOULD HAVE YOU MOVING SHORTLY.

With a fair number of similar examples of failures, the ML quickly learns that condition such as screens with few DOM elements, a single dialog that contains the strings ‘sorry’, ‘oops’, or ‘try again’ are very often bugs in the app.

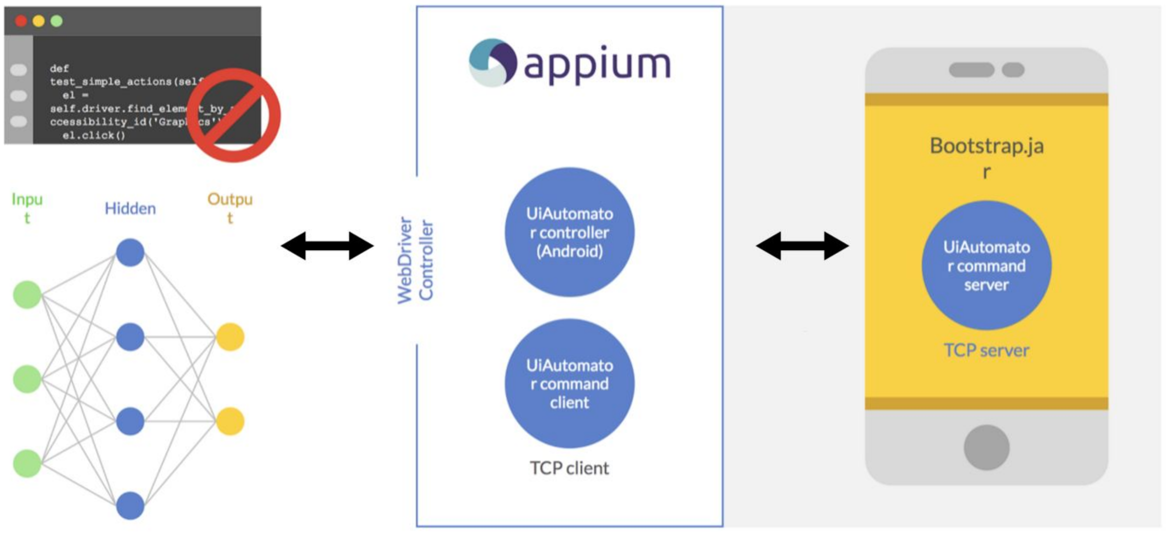


The most powerful method of detecting issues lies in the ability to record every screenshot, DOM and action sequence taken in every test run through the app. Humans then quickly verify these sequences represent good or bad behavior (pass or fail). Then, on subsequent runs, comparing the data between the older run and the newer reveals:

A. The run was identical, the application is still behaving correctly.
B. The run was not identical and revealed a new fault or bug in the applications behavior
C. The run was not identical but discovered a new valid element or path in the application. Humans review for functional correctness, and/or issues.

A subtle but powerful note about the validation that occurs in #3--the ML bots are able to exercise new functionality as soon as it appears the in the app, humans only need review the new behavior, whereas non-ML approaches require the human testers to take note of new functionality or UI in the application, write a manual or automated test cases and execute the test case--much of which in practice just doesn't happen due to the complexity gap, test resourcing and agile practices described above. The generalized ML models on the other hand are robust to changes in the application structure and design, and even recognize new functionality as it is very likely to look similar to that of other apps.in the training set.

ML-based testing system has obviated the need for classical manual or automated regression testing. Human tapping and test code scripts are simply replaced by trained machines that interact with the app via testability APIs.



```
def
test_simple_actions(sel
  el =
self.driver.find_eleme t_by_
ccessibility_id('Grap cs'
  el.click()
```

Figure: Replacing traditional, hand-crafted test code logic, with neural networks.

4.5 Benchmarking and Quantifying Quality

Traditionally quality metrics are often simply measured as the percent of available tests that pass or fail, or the load time of a few select pages. Those quality metrics lack context and often aren't motivating enough for app teams to take action and improve them. As ML test bots can test arbitrary apps with little customization, there is now the opportunity to test every app, The reference bot systems I've worked on have executed on over 30k apps, and with all that data we can now benchmark an apps quality against the rest of the marketplace of apps.

From each app test run, data such as performance, stability, and errors can be collected, and associated with labels and application categories. Benchmarks such as 'average login screen load time' or reliability of all apps in the Shopping category enable app teams to understand their app metrics relative to other apps.

Without benchmarking, app teams discover their performance or stability numbers, but are rarely sure how good is good enough. With benchmarking, app teams can be motivated by knowing their app is in the slowest 20% of all apps in their category.

ML also enables the ability to do competitive analysis of app quality. Most app team's can't keep up with the testing requirements for their own app, let alone their competitors apps. With ML, it is possible to run vast regression suites against direct competitors apps, and compare quality metrics in a way that directly relates to the business and is highly motivating to non-engineers to resolve quality issues when presented in a competitive light.

4.6 Abstract Intent Test Cases

The last mile of applying ML to software testing is the ability to have the ML testing bots execute very app-specific test cases, or import legacy test cases for ML execution. The ML bots know how to get to and from differently labeled portions of the application. We now need a way to orchestrate the ML bots to execute very specific, named, test sequences with exacting input and output verification. There are three capabilities needed to execute these specific regression test cases:

1. Named sequences of steps.
2. Specific test input per app state
3. Verify specific app features or strings in a given state.

To address the definition of test cases at this level of abstraction (labeled states and elements), I propose a new formal test case format specifically designed for ML test execution. Traditional manual test cases are often loosely schematized collections of test case names, test steps, validations, and categorization metadata. Test automation test cases are often either encoded directly in a procedural coding language with very little structure, and/or represented in schemas similar to manual test cases in a formatted file, or formal Test Case Management System. As ML tests are able to execute on arbitrary applications, we want to ensure there is a declarative format that doesn't bind the test case logic to the specific application.

It borrows heavily from Gherkin, but has additional specificity in that it allows for the actions and verification steps to be sequenced, versus unordered in Gherkin. The details are beyond the scope of this document, but an example instance is provided below.

Example AIT:

```
# Demo AIT Test Definition
Test Name: Remove Beanie Baby from item from cart
  Description: Make sure we can remove an item from the shopping cart.
  Tags: cart, remove
  Step: Search for Beanie Baby
```

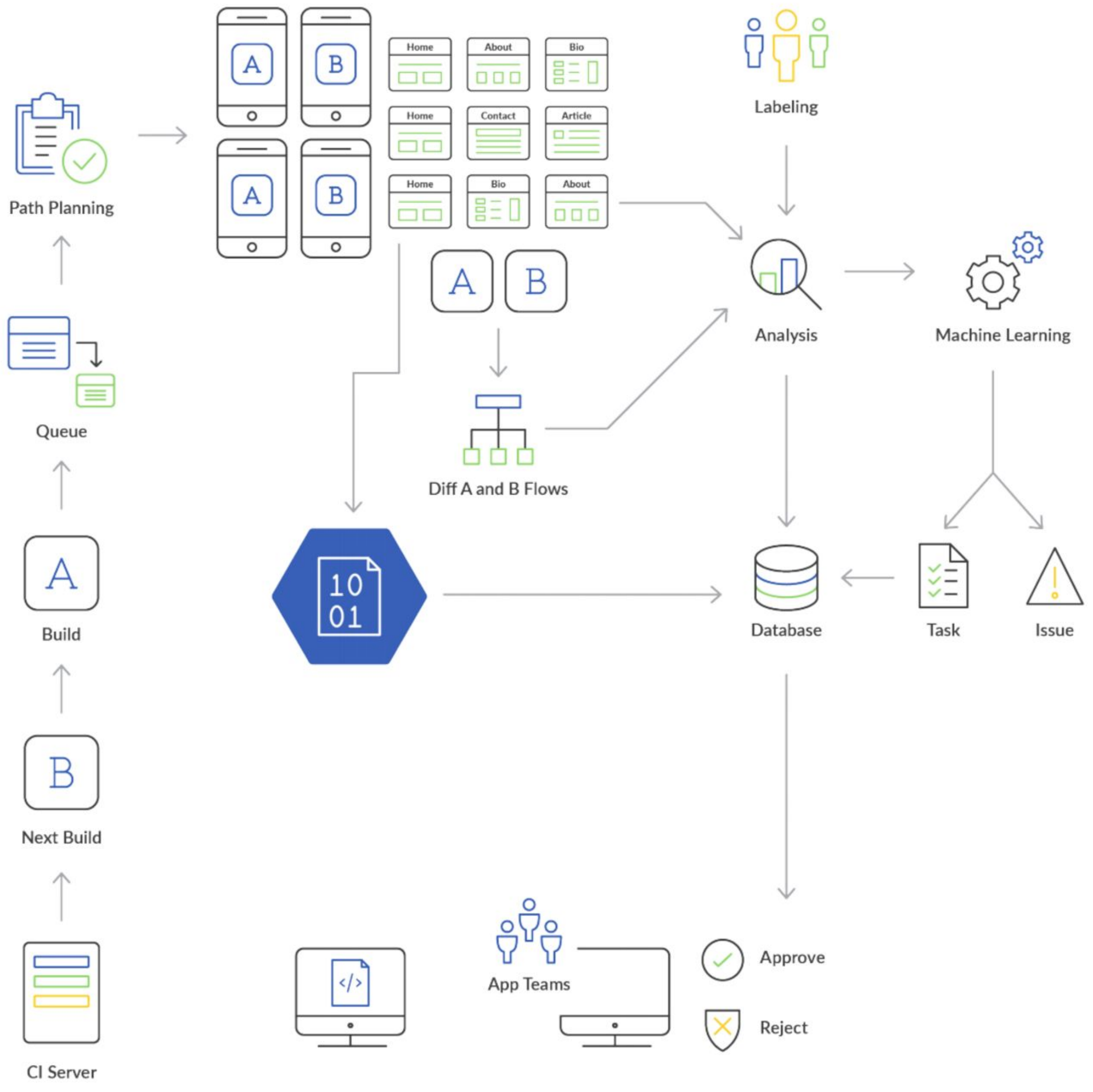
```
    Context: SCREENNAME “Search”
    Input: ACTION SEARCH “Beanie Babies”
  Step: Add Item
  #Step: Add Item Any item will do.
    Context: SCREENNAME “Product” and HASWORD “Beanie Baby”
    Input: ACTION ADDTOCART
  Step: Remove Item
    Context: Cart and HASWORD “Beanie Baby”
    Input: Remove Item
  Step: Verify Item Removed
    Context: SCREENNAME Cart and NOTHASWORD “Beanie Baby”
```

The aspects to note are that this test format allows for repeatable app- and data-specific test input and validation are possible. The value of the AIT format is that it focused on the Abstract Intent of the Test Case, and all I/O is based not on exact steps or sequences in the application, rather they are notes to the execution of the ML testing bots that they need to ‘find’ a given labeled app state, interact with that screen’s labeled element with a specific action.

This label-based approach to test case definition avoids one of the most common sources of test case maintenance pain--changes in the application UI or flow. Traditional frameworks are sensitive to changes in the DOM structure and flow of an app as each test step must be executed in exact sequence and it must find each element for each steps interaction based specific element search criteria. With ML testing bots, the burden of finding the app state and input elements is left to the bot’s ML classifiers from the label training above. If the application’s flow changes, the bots will still search the statespace of the app to find the new, correctly label state for data entry. In the case that an element for input has changed its location, size, color, parent-child relationship in the DOM, etc., the bots have been trained on thousands of applications and can still identify the correct element for input despite severe morphing.

4.7 ML Test Architecture

Here a reference Architecture for an ML-based test generation and execution system is shown.



5 Testing AI Products and Features

Many applications and services are incorporating machine learning. In the current AI renaissance, there is an almost irrational rush to re-design old features with AI/ML, and add new features only possible with the advent of modern AI/ML and cloud compute. Often, this is done with little thought of how to test these new systems.

Versus classically coded functional features, AI/ML apps introduce a new set of software quality and testing challenges:

1. Feature Computation
2. Sampling Quality
3. Outlier Tolerance
4. Labeling Quality
5. Quality Drift
6. Traceability

5.1 Feature Computation

As we have seen for the ML testing methods above, all ML systems need input data for training and during execution. This is generally called Feature Engineering. In the above diagrams, this logic is built into the 'Application Driver'. Feature engineering is the work of transforming real world application or signals into a form that ML can understand. Generally speaking, ML systems can only understand inputs of floating point numbers between zero and one.

An example of feature computation is converting the number of buttons on a given application page, to a value useful for training networks.. The code for this 'feature', asks the platform driver for a list of all elements in the current application DOM, counts how many buttons appear in the DOM and then normalizes this count to a number between zero and one. This feature engineering is often traditional procedural code that needs traditional testing. There are often hundreds of features similar built for every ML system implementation. Each of these needs to be tested carefully because if the data used in the training or runtime is incorrect, the output of the AI feature or product will also be incorrect.

Beyond testing the functional quality of these 'features', it is important to note that the normalization is done correctly. In the example above, if the count of buttons in practice never exceeds 100 per page, if the normalization divides the the number of buttons by 50, some granularity will be lost in the training as pages with more than 50 buttons will be missed. Also, if the normalization includes a division by 1000, most every app will appear to the network to have very few buttons as the input values will always be in the range of [0, .1], leaving 9/10's of the possible values in the range [0, 1] unused.

Feature engineering and selection is an art, but it must be directly tested and questioned.

It is also important to note that the features into the ML are also likely related to the desired output, ideally correlated with the outputs if that is known. Irrelevant features mean the ML training time is more expensive as the ML learns to ignore that signal, might cause spurious outputs, and often results in lower quality ML execution as the ML could often have used the part of it's brain to suppress that bad signal, and used it to better differentiate other aspects of the input.

5.2 Sampling Quality

An often overlooked aspect of ML quality is the sampling of the training data. If the training set has a bad sampling of data, the ML system will appear to function well in test conditions, but in real world inputs it will behave poorly, or even dangerously.

AI-Powered web search can help illustrate this problem. The process of taking the top 50 results returned by the core search index and ranking system, then optimizing the results based on the actual query, who is issuing the query, where the query was issued from, etc. This last ‘dynamic ranking’ step helps put the right blue links at the top and is critical for web search relevance.

What do we need for training data? Lots of queries. We can’t possibly train on all possible search queries, there are too many, and in fact new unique queries never seen before are issued every second. A lazy approach might be to just a random sample of 5,000 search queries from the 100M search logs from yesterday. That sounds reasonable but if we trained the web search ranker on this data:

1. If the training sample was taken from logs on a Wednesday, it likely doesn’t contain, or adequately represent queries that happen on weekends--people are often searching for very different things based on time of day, week, and year.
2. Is the sample size correct? Is 5,000 enough? Probably not. Queries are made up of things like people’s names, places, objects, shopping intent, UPS Shipping Codes, etc. How likely is it that your sample contains enough examples of each subtype of query? If 1 in 10000 queries is a UPS Shipping Code, it is likely your sample doesn’t even have a UPS Shipping Code and the ML training won’t even have a single example to train on.
3. Consider whether the sample represent the targeted or business motivated sample? If you work on a search engine that isn’t the market leader, with a predominantly older and non-technical demographic, your search sample means the search engine you create will great at queries for these folks, not do well on technical, or queries typical of a younger demographic, meaning younger users will find your product subpar and not switch over.
4. Consider whether the sample is too broad? If you have an international, multi-lingual audience and you sample includes all of these languages, the ML can be easily confused, or you need enormous amounts of training data and compute to let the ML sort it all out. Or, you can build a classifier to determine the language of the query, and then route that query to an ML system that trained only on one language with a far smaller language-specific query sample.

In practice, for a production web search ranker, the training and test samples often include queries from all of the last month and year and significant holidays., 30,000 samples turned out to be a good balance between sampling breadth and training time, and yes, one team never realized they were training on demographically-based data.

5.3 Outlier Tolerance

Often, ML systems will perform very poorly outside the range of input they were trained on. When ML systems are deployed into real world environments, the input data can involve very odd outlier inputs to the system. Especially with ML systems, it is important to test for inputs that you cannot predict.

In the web search example, imagine Beyonce’s next child’s is named “UniverseOfAllThingsAmazingChild”. Or, Facebook engineers create a chatbot that says: “balls have a ball to me to me to me to me to me to me to me”. Though that might never happen, the internet may start searching for this unusual and nonsensical phrase.

To test the robustness of a system, it is often not enough to sample from the real world for test and training data. Testing should often include generated, deliberate outlier inputs to see how the trained system performs in strange situations.

5.4 Labeling Quality

As we saw above, a key aspect of training ML to behave intelligently (like humans) is to have human-labeled data which which to train on. The process of labeling is critical to the quality of the resulting ML system.

As with sampling training data, care should be taken when selecting the humans used to label your data. Labels are human-opinions and judgments and can have a strong impact on how the ML learns to behave. In the web search example, human labelers used to rate queries -> result matches. They look at the search query and rate the possible result links on relevance--based on their opinion. So, if a search engine used inexpensive humans and paid them all $10/hour, and that data is used to train the search engine, the resulting ML search ranker will sort results like $10 an hour worker. How intelligently do you think people willing to work for $10/hour will rate the results for a query on the topic of ‘Artificial Intelligence’? What if the labelers were 90% women? Ensuring labelers and the labeling activity matches your desired application can be very important and is often overlooked.

It may not be obvious, but disagreement in labeling is common. People just think and see things differently. If you have only one person label each item, the labeling process will miss any disagreement. High quality labeling systems and processes ensure there is overlap in the ratings, meaning more than one person labels each artifact. In the web search example, if a labeler sees the query ‘Bush’, they will label the result for President Bush high if they are over 30 years of age or well educated. If the labeler is a farmer, or didn’t grow up in the United States, they may label the result for tiny trees much higher. The labeling system must have overlap, and allow for disagreement in the labeling or risk having lopsided training data.

A well-known production web search engine once paid every labeler about $10/hour, and when labelers disagreed with the consensus vote of other labelers--they were fired. Avoid these actions in your own labeling activities to ensure a high quality ML training system.

5.5 Quality Drift

One of the most overlooked quality issues with ML systems is that each new version is a completely new implementation. In classically coded products, only a few lines of code change between each build and the next. Often, the means the regression testing can even rely on this and reduce the number of test (aka Test Selection).to only regress the areas of the app likely impacted those few lines of code.

Most every ML system has a completely differently implemented system after each training session. ML systems often start by randomizing their internal networks--yes, they randomize themselves to start with and that is a fundamental aspect of how they learn. Any small changes not just in the test data, but even the ordering of the training data, or if the training rand function isn’t seeded, or even different training times or threshold configurations of the training system will often result in completely different implementations.

In the web search example, we measure the relevance of the system based on how well the results per query are ordered by the machine, compared to the human labeled suggestions of correct order. 100 would be a perfect search engine--it matches all the human labels. Perfect search engines don’t exist, in fact that sit around 70% correct by most measures today. So, if we have a build from yesterday and it scored a 70, and a new build today also scores a 70, that only means that *on average*, the two builds have the same relevance/quality measure. In practice the two engines might perform very similarly and do well on the same queries and poorly on the same queries. It is often the case though , that any small tweak to the training data, feature set, or because of random initialization in the training process, the two engines do well on completely different subsets of queries, and poorly on another set of subqueries. They are very different search engines with the same average value. The first build may be great at people searches but suffer on technical queries. The reverse could be true of the second build of the engine. If we shipped a new build of the engine every day, searches for ‘Britney Spears’ will be amazing today and be of poor quality tomorrow. This is quality drift.

To ensure your ML product doesn’t suffer from drift, or to at least track it, you must have separate measures of quality for each subset, or subcategory of test input, and track those quality measures separately and decide in the drift build to build is acceptable for your application.

5.6 Opacity and Traceability

I described ML as a black box above--that’s not just true for the context of this paper, but in all practicality it is true for almost every ML-based product. Imagine not being able to debug your code, not having meaningful logs--that is the state of deployed ML. ML systems often have hundreds or thousands of nodes, each with seemingly random weights. Much like the human brain, it is difficult to understand why any given brain made a decision--its just did.

This means many classic approaches to quality such as log monitoring, stepping through code, and code-coverage analysis tools; are all useless in this new world of ML-based application logic.

Generally, the problem of testing and debugging this new wave of AI-based products and features is incredibly difficult. Not only is the problem difficult, but the consequences are grave as these AI systems are embedded in fast-moving vehicles on the road, controlling supply chains, and soon deciding where and when to fire a missile from autonomous drones.

DARPA (the Defence Advanced Research Projects Agency) has deemed this lack of visibility and traceability in modern AI systems to be a DARPA-hard problem as there is no obvious solution, or enough folks working on this problem today and it’s consequences are worth of national security interest. DARPA has defined the XAI project and is starting to fund research in this area.

6 Summary

Artificial Intelligence (AI) and Machine Learning (ML) have the potential to dramatically improve the ability to test software applications. Realizing that ML-training systems and software testing are very similar implies that general ML solutions should readily apply to software testing tasks. The generality of ML-based testing systems that can execute on many different apps without customization greatly improves the re-use of test development efforts. This generality will speed up development of test cases, and improve quality by generating far more coverage through re-use.. ML-based testing solutions has the promise to both disrupt and help the world of traditional software testing.

The sudden increase of application software written using these very same AI and ML techniques will create even more difficult software quality and testing problems. ML-based products are often re-trained from scratch with each new version, difficult to debug, and their relevance and predictive ability is limited to the availability of training data. These issues indicate that quality will suffer relative to simpler, procedural-based applications. These new ML-based products create the need for standardization in the toolsets and best practices for testing such systems.

AI and ML will radically change the nature of software testing and quality in the coming years--far more radically than most expect.

References

Wikipedia contributors, "Machine learning," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Machine_learning&oldid=796373868 (accessed August 20, 2017).

Wikipedia contributors, "Selenium (software)," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Selenium_(software)&oldid=795945653 (accessed August 20, 2017).

Wikipedia contributors, "Test-driven development," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Test-driven_development&oldid=790275486(accessed August 20, 2017).

Wikipedia contributors, "AlphaGo," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=AlphaGo&oldid=795373355 (accessed August 20, 2017).

Wikipedia contributors, "DeepMind," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=DeepMind&oldid=795218131 (accessed August 20, 2017).

KIF, "kif-framework/KIF: Keep It Functional - An iOS Functional Testing Framework", https://github.com/kif-framework/KIF (accessed August 20, 2017).

ESPRESSO, "Espresso | Android Developers", https://developer.android.com/training/testing/espresso/index.html (accessed August 20, 2017).

DARPA XAI Project, "Explainable Artificial Intelligence", https://www.darpa.mil/program/explainable-artificial-intelligence (accessed August 20, 2017).

Wikipedia contributors, "Cucumber (software)," *Wikipedia, The Free Encyclopedia,* https://en.wikipedia.org/w/index.php?title=Cucumber_(software)&oldid=794655871 (accessed August 20, 2017).

XCUITest, "User Interface Testing", https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/09-ui_testing.html (accessed August 20, 2017).

Thanks to the following people for their help editing this paper: Tariq King, Rick D. Anderson, and Chris Navrides.