AI-Driven Test Generation: Machines Learning from Human Testers

Dionny Santiago, Tariq M. King, Peter J. Clarke

dsant005@fiu.edu, tariq_king@ultimatesoftware.com, clarkep@cis.fiu.edu

Abstract

Although recent test automation practices help to increase the efficiency of testing and mitigate the cost of regression testing, there is still much manual work involved. Achieving and maintaining high software quality today involves manual analysis, test planning, documentation of testing strategy and test cases, and development of test scripts to support automated testing. To keep pace with software evolution, test artifacts must also be frequently updated to stay relevant. Current test automation is unable to generalize across applications and is unable to mimic human intelligence. As such, there is a significant gap between the current state of practice and a fully automated solution to software testing.

This paper investigates a practical approach that leverages artificial intelligence (AI) and machine learning (ML) technologies to generate system tests based on learning testing behavior directly from human testers. The approach combines a trainable classifier which perceives application state, a language for describing test flows, and a trainable test flow generation model to create test cases learned from human testers. Preliminary results gathered from applying a prototype of the approach are promising and bring us one step closer to bridging the gap between human and machine testing.

Biography

Dionny Santiago is a Principal Quality Architect at Ultimate Software, a leading cloud provider of human capital management solutions. Dionny provides expertise and leadership in software testing and contributes in various capacities including hands-on testing, technical problem-solving, training, and internal tools development. Test automation being one of his primary areas of interest, one of Dionny's goals is to advance the current state of the art; he is focused on research and development efforts to apply artificial intelligence and machine learning to software testing. Dionny is also currently pursuing his M. Sc. in Computer Science at Florida International University.

Tariq M. King is the Senior Director and Engineering Fellow for Quality and Performance at Ultimate Software. With more than fifteen years' experience in software testing research and practice, Tariq heads Ultimate Software's quality program by providing technical leadership, people leadership, strategic direction, staff training, research and development in software quality and testing practices. Tariq is a frequent presenter at conferences and workshops, has published more than thirty research articles in IEEE and ACM sponsored journals, and has developed and taught software testing courses in both industry and academia.

Peter J. Clarke is an associate professor in the School of Computing and Information Sciences at Florida International University. His research interests are in the areas of software testing, model-driven software development, and computer science education. He has published over 75 research papers in various journals and conferences, and has received research grants from the NSF and Ultimate Software Group Inc. He has served as the program co-chair for one IEEE conference, the co-chair for 11 faculty development workshops, and on the technical program committees for more than 50 conference and workshops. He is a member of: AAAS, ACM, AISTA, ASEE, AST and the IEEE Computer Society.

1 Introduction

Although test automation practices provide critical efficiency benefits to the software development process, there are many accompanying problems with the current state of the art. Test scripts do not generalize across applications, and may easily break in the presence of changes to the underlying application. Outside of model-based testing (discussed later), test scripts are hand-crafted by humans, and test execution and logging of results are the only fully automated parts of the testing process. Also, automated test script oracles are limited and can only detect defects based on the path and assertions that were coded explicitly in the test script. Consequently, current approaches do not provide for fully automated software testing and substantial manual effort is still required.

There exists a significant gap between machine testing and human-level performance. Human testers can perceive the state of an application, can act intelligently, and can observe resulting application state to uncover defects. To improve efficiency and reduce the cost of quality, there is a need for improving software testing and test automation, and for introducing more intelligent automated testing behavior that is capable of mimicking human behavior.

Advances in AI and ML have shown that machines are capable of matching or surpassing human performance across various problem domains (Bojarski et al. 2016, Moyer 2016, Xiong et al. 2017). As a result, we are witnessing the new uprising of self-driving cars. These systems are capable of perceiving the environment and surroundings of a vehicle, and are capable of mimicking intelligent human driving behavior. We are also seeing many intelligent systems that can communicate with humans and answer questions asked in spoken natural language, and even systems capable of beating top human players at classic board games such as Go.

As a testing community, we are lagging behind other software engineering fields when it comes to innovation. While there has been extensive research and work into semi-automated testing techniques, such as model-based test generation, these techniques do not: (1) mimic the thought and learning process of human testers; and (2) seamlessly generalize across applications and application domains. There is a need for building more intelligent software testing approaches akin to that of self-driving cars and other comparable intelligent systems.

This paper aims to stimulate research and innovation in software testing through the use of AI and ML. We present an example of our journey and experiences of applying existing AI research to testing. We also strive to provide critical takeaways for how quality professionals can get the necessary foundation to break into the AI world and start contributing to this ripe emerging field of intelligent automated software testing.

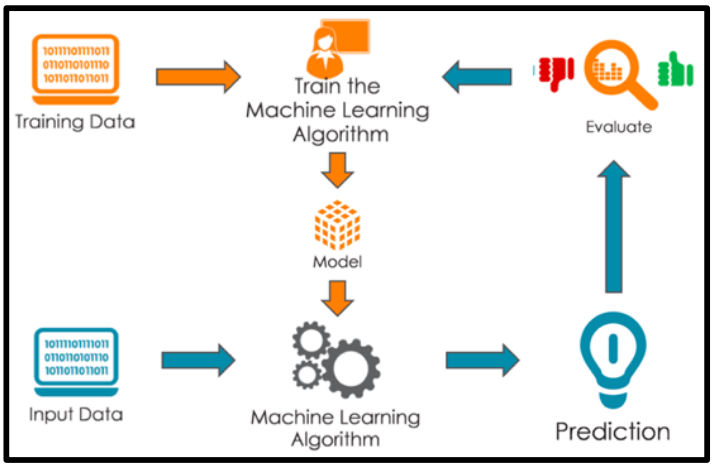
2 AI for Software Testing

Thanks to the continuing advances in AI and ML, the idea of being able to build intelligent systems that can test software, and that can improve their ability to test by learning from humans, may be closer than we think to being a reality. In this section, we provide an overview of ML, and we also establish a mapping between the software testing problem and ML.

2.1 Overview of Machine Learning

Machine learning is a sub-field of AI and is the science of getting computers to act without being explicitly programmed (Stanford 2018). Two major applications of ML are referred to as **unsupervised learning** and **supervised learning**. In unsupervised learning, unlabeled data is fed into a training algorithm with the goal of discovering patterns and relationships. Examples include clustering algorithms that attempt to organize data points into groups. Supervised learning, in contrast, involves **human-labeled training data**. A supervised learning workflow typically involves multiple iterations of constructing labeled training

data, extracting features from the data, choosing an algorithm, training a model, evaluating the model using test or input data, and improving the model. The diagram below portrays this workflow:



A common ML system is the artificial neural network (ANN). ANNs are computing systems vaguely inspired by biological neural networks, and are based on a collection of connected units called artificial neurons. There are many different types of ANNs, including convolutional neural networks (CNNs) and recurrent neural networks (RNNs). CNNs are commonly used for image recognition problems, and RNNs are commonly used for natural language processing (NLP) problems.

Another application of ML is **reinforcement learning**. This learning approach involves creating algorithms that are capable of learning by using reward systems. Useful actions are positively rewarded, whereas less useful actions may be penalized.

2.2 Relationship Between ML and Testing

There is a direct mapping from the software testing problem to a machine learning solution. The testing problem involves applying a test input to an application or function, then comparing the output to an expected result. This is precisely what machine learning does. A set of inputs (or **features**) is supplied to a training algorithm. In supervised learning, the correct answer is also supplied to the training algorithm with each set of inputs. The job of the machine learning system is to iteratively *(slightly)* reconfigure the "internal brain", each time getting better and better at providing the correct answers based on the provided input sets. Therefore, all of the existing and ongoing research and development that has gone into building these ML systems are providing a direct benefit towards further automating the software testing problem.

Self-driving cars and the technology that powers them can be mapped to the problem of software testing. Just as humans can teach learning algorithms how to drive a car, we can envision building a system capable of learning from a user's testing journey. Similarly, NLP and natural language generation research may map to test case generation. Lastly, game theory and reinforcement learning may map well to the problem of discovering a system and hunting for bugs. For example, reinforcement learning may be used to reward an intelligent system for uncovering a system crash or an exception.

Researchers and practitioners realize the potential for AI and ML to be leveraged to help bridge the gap between the testing capabilities of humans and those of machines (AIST 2018, AISTA 2018). Existing work on applications of ML to software testing has explored applying supervised ML to the testing problem (Arbon 2017). There is also an abundance of emerging research on new AI and ML techniques and algorithms. It is imperative that the testing community make a concerted effort to keep up with AI research and look for ways to innovate within the testing field.

3 Applying AI to Testing

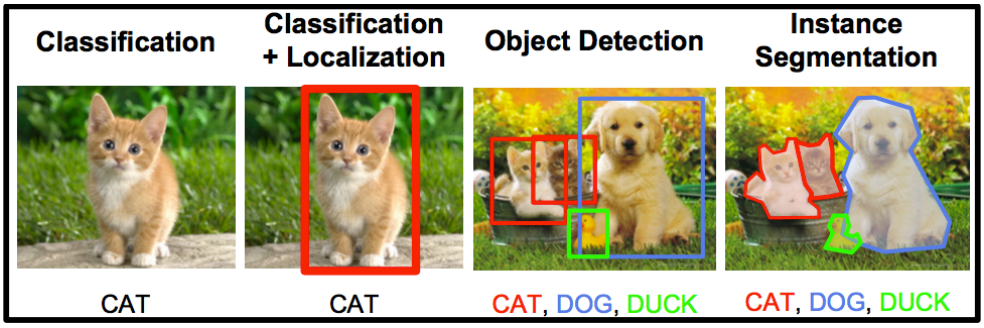
In this section, we set out to provide a journey of how we have been able to leverage existing AI research and map it to specific software testing problems. We present two examples. For each example, we first focus on an AI research area; then we explore our experiences when trying to map the research to testing. Our goal is to let our experiences serve as a framework and as motivation for the reader to follow a similar approach.

3.1 Research: Object Recognition

There exist several problems and areas of research surrounding object recognition within images. Image classification involves visually inspecting an image and deciding whether the image belongs to a particular class. For example, given an image of a cat as input, an image classifier may return the label "cat" as an output.

In some cases, classifying an image with a single label is not sufficient. Suppose the input image contains both a cat and a dog. While assigning either label would be correct, we may instead want to produce two output labels for the single input image. Research efforts have been focused on ML algorithms that can produce multiple outputs. Lastly, in some cases, we are not just interested in generating classes for a given input image. Depending on the problem at hand, we may want to build a system capable of detecting the location of the object within an image.

The following image illustrates several object recognition research problems:



To get a feel for how the different training and prediction processes work, we encourage you to experience these algorithms for yourself. There are several image recognition tutorials and videos freely available online (see **Section 5**). We recommend starting with image classification tutorials that use the TensorFlow (Abadi et al. 2016) framework. TensorFlow is a popular and highly active open source ML framework developed and maintained by the Google Brain team.

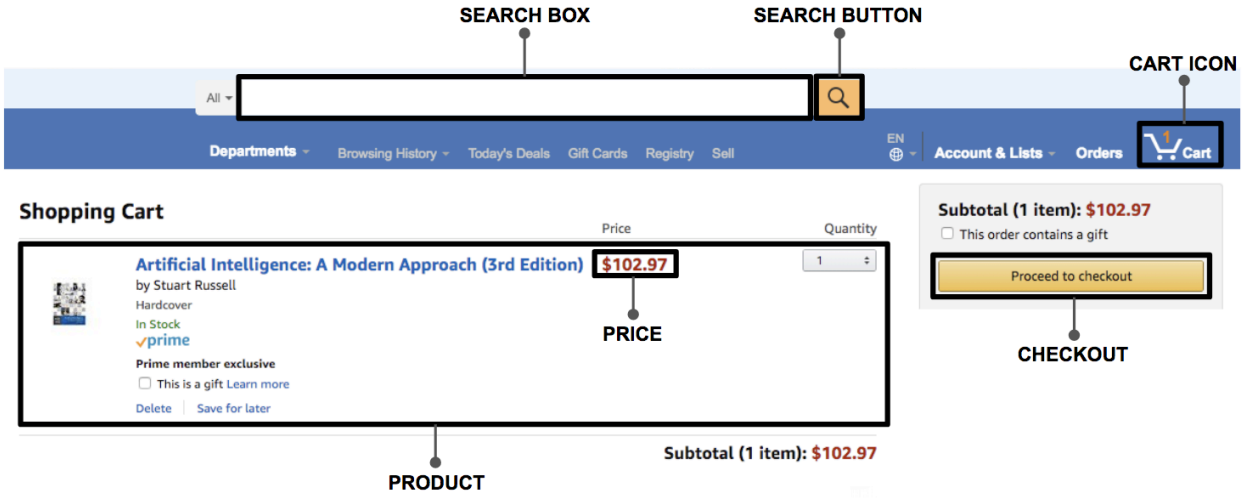
3.2 Application: Perceiving Web Application State

The question we will explore in this section is: How can image classification and object detection be leveraged to aid in software testing?

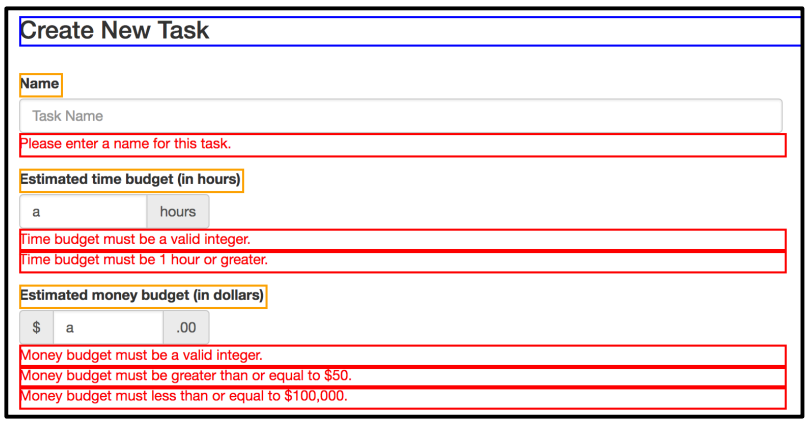
To generate and execute test cases against a web application, the current state of the art involves constructing page objects that are coupled to the implementation details of a specific web application. Page objects often include references to document object model (DOM) and stylesheet information for a specific system under test (SUT). It is beneficial to shift towards an approach that leverages ML to raise the level of abstraction by which generated test cases interact with webpage components. Rather than interacting with elements using information that is SUT-specific, the goal is to be able to leverage ML to identify web components based on models that have been trained across various SUTs.

Having the ability to perceive web application state and recognize objects using ML-based approaches moves us one step closer to being able to write test cases devoid of any SUT-specific implementation details. Armed with enough example training data, we can leverage existing AI research and techniques for object recognition to raise the level of abstraction by which our automated test cases refer to objects. There are several benefits to this approach, including the ability for test scripts to self-heal if the SUT changes. For instance, while a change in the way the SUT renders a shopping cart button would likely break test scripts that leverage traditional DOM-based element selection strategies, a sufficiently trained ML model may be able to locate the new shopping cart button, allowing a test script that leverages ML-based element selection to continue execution. Also, by raising the level of abstraction, it becomes possible to reuse test cases across different SUTs.

With enough training data, an ML model could be trained to recognize various components of a web application state. An example of webpage decomposition and object recognition is below:



The diagram below shows another example of an actual ML-based classification system recognizing various components (Page title, widget labels, and error messages) on an arbitrary web page:



Picking up from the exercise we recommended in **Section 3.1**, we encourage readers to continue practicing by collecting many images of webpage components, labeling the images, and then building an ML system capable of classifying webpage components. Images can be collected and labelled using tools such as LabelImg (LabelImg 2018). Frameworks like TensorFlow and Keras (Chollet 2015) can then be used to train ML models using the generated training set. During our research, we have also found it worthwhile to collect information from the webpage DOM alongside with the component images. Since ML systems thrive on data, being able to extract more features from the raw collected data generally

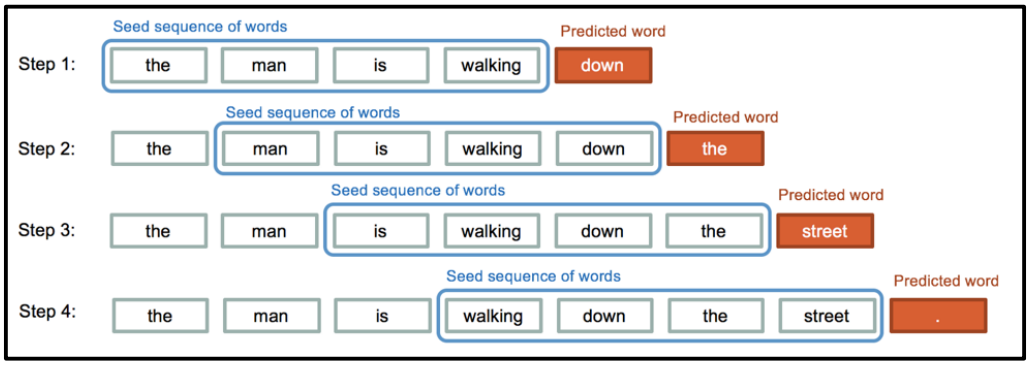
allows for more useful experimentation towards increasing the accuracy of the ML system's performance on the training data.

3.3 Research: Text Generation

Much research exists on generating sequential information using ML-based techniques. In the field of natural language processing (NLP), several approaches to text generation have been studied. A conventional approach is to treat text generation as a **sequence-to-sequence** problem. A sequence of text is fed into a trainable algorithm that in turn outputs a sequence of text. The training goal is that the concatenation of both sequences results in a plausible sentence. Since machines do not understand words, a common practice is to map words to integer values, resulting in what we may refer to as a **word encoding**. This means that both the input and output of the sequence-to-sequence problem are a sequence of integers. Word encodings are challenging to create and maintain as there are many possible words that may appear, especially when different languages are used.

An alternative approach is to create a **character encoding**, where each character (for example, each character from A-Z) is mapped to an integer. Although this simplifies the text-to-machine mapping process, it creates additional complexity from a training standpoint. There are many valid (and invalid) arrangements of character sequences that can first form words and eventually form sentences. In contrast, there are much fewer unique arrangements of complete words. ML-based text generation techniques that leverage character-based encoding typically require more training to reach stability and feasibility-of-use over word-based encoding approaches.

An example of an ML approach to sentence generation using word encodings is below:



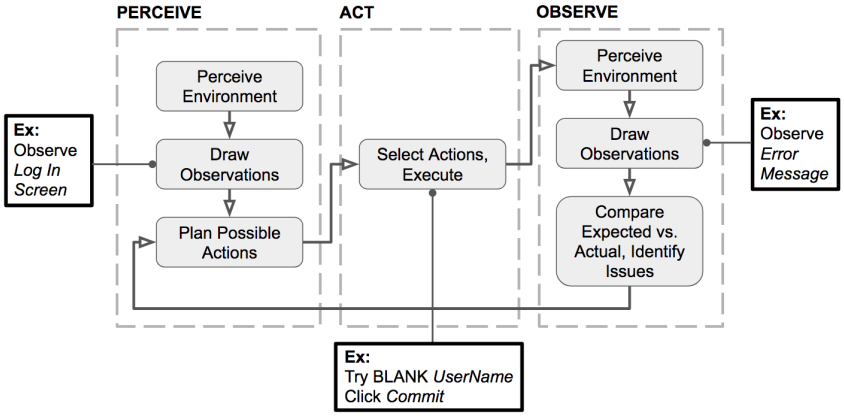
Research has shown that specialized ANNs are capable of being trained to handle sequence-to-sequence problems (such as text generation). Long Short-Term Memory (LSTM) recurrent neural networks (RNNs) are a particular type of neural network capable of generating sequential data with long-range structure (Graves 2013). For the interested reader, Jason Brownlee provides a great step-by-step tutorial that leverages tools such as Keras and TensorFlow to build LSTMs capable of generating plausible sentences for a problem domain based on training from classical texts (Brownlee 2017).

3.4 Application: Text Generation for Web App Testing

The question we will explore in this section is: How can text generation be leveraged to aid in software testing?

In **Sections 3.1** and **3.2**, we explored image recognition research and practical applications for software testing. Raising the level of abstraction by which we interact with objects in a web application sets the stage for the next step. Having the ability to recognize objects, the next task of interest is learning important test flows from human testers. A **test flow** is a sequence of actions performed onto a SUT,

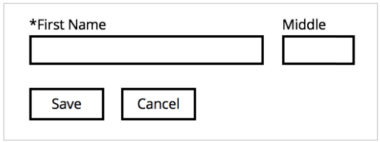
followed by a set of expected observations. The steps that a human tester follows while executing test flows may be modeled as follows:



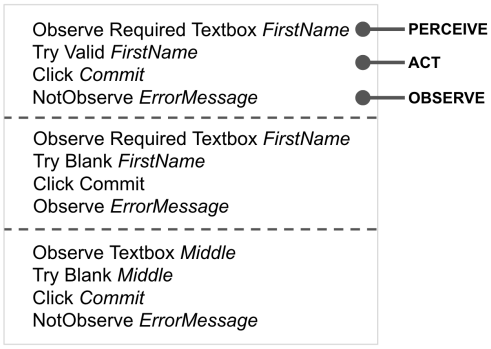
At a high level, we present the test flow execution process as a sequence of steps where each step belongs to one of three categories: **Perceive**, **Act**, **Observe**. Steps belonging to the **Perceive** category focus on establishing preconditions for a test flow, and involve drawing observations from the environment, and action planning before selecting an appropriate set of actions. Steps belonging to the **Act** category focus on selecting and executing appropriate actions based on the previously constructed plan. Finally, steps belonging to the **Observe** category focus on comparing expected vs. actual SUT behavior, and deciding on correctness. We support the continuity property of the testing process by allowing the results of the **Observe** steps to initiate a new **Perceive** stage, thus forming a cycle.

Presenting the test flow process using this framework is the first step towards the goal of being able to represent the process in a machine-understandable format. Next, we define a **language** that may be used to express concrete test flows that fit into our framework. The language must be expressive enough to allow covering essential test cases, yet constrained enough to reduce the data complexity of ML-based techniques to test flow generation. The language must also promote the use of webpage component abstractions that utilize the system described in **Section 3.2** in place of SUT-specific information. This supports the generality of the approach.

The main building blocks of the language are **components**, **actions**, and **observations**. Components represent elements on a web page. Observations represent information about components that can be perceived from a given web page. Finally, actions are interactions that may be performed onto components. By interleaving observations and actions, the language allows for the specification of test flows. The language supports the use of abstract learned objects, instead of using specific input values and observed text values that may only be pertinent to a single SUT or only pertinent to a specific domain of software applications. The figure below shows how test flows (b) may be created for a simple form (a) by utilizing the language:

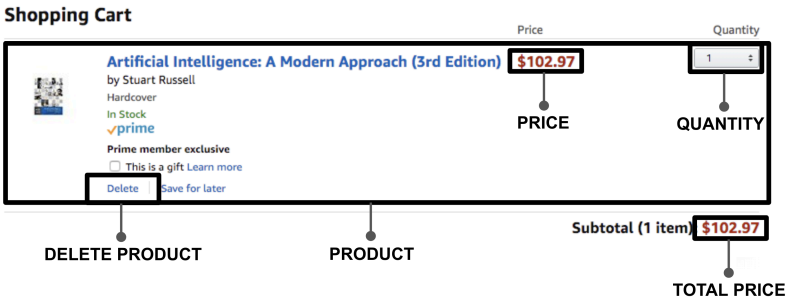


(a)



(b)

Going back to the shopping cart example from **Section 3.2**, the figure below shows how the language could be utilized for more complex scenarios:



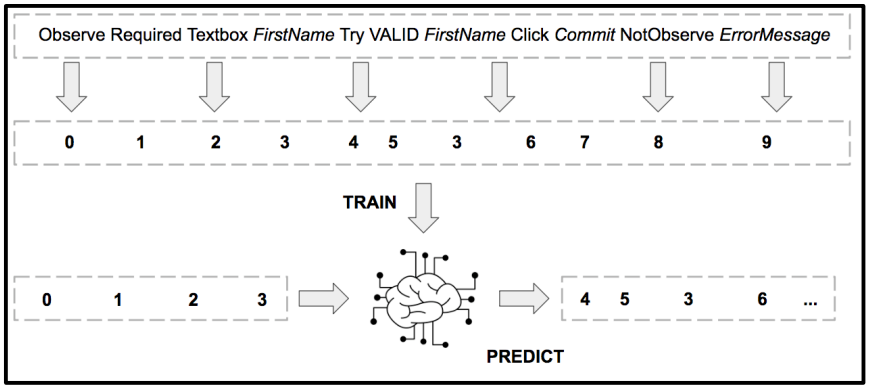
Observe Screen *ShoppingCart*
Focus *Product* In *Collection*
Try Click *Delete Product*
Observe *Product* Not In *Collection*

Observe Screen *ShoppingCart*
Focus *Product* In *Collection*
Try Increase *Product Quantity*
Observe Increase In *Total Price*

It is important to note that the test flows are described at a higher level of abstraction than the specific underlying SUT that motivated their creation. For example, the equivalence class "VALID" is used in place of any specific First Name. Additionally, the "Commit" element class is used instead of referencing a specific "Save" button. This allows learning generalized test flows that may be used to generate test cases across various SUTs. As an example, the test flows presented in the shopping cart example are reusable across different e-commerce web apps.

Having defined a language capable of expressing test flows, we can now use the techniques we learned about in **Section 3.3**. Since we now have a mechanism of expressing test flows using strings belonging to a language, we map the problem of text generation to that of generating strings that belong to the language that we have defined.

Using the **word encoding** technique for representing text sequences, below is an example of an ML-based training and prediction approach to test flow generation:



To use test flows as training data, a **sliding window** data augmentation technique is used to expand the training data. We frame the test flow learning problem as a supervised learning problem by taking each test flow from the training set and expanding it one word at a time, creating a new training example with a more complete sentence at each iteration. For each generated training example, the next step of the sentence is the correct assigned label.

Being able to train an ML system to generate test flows means we can now leverage humans not just to train a system to detect objects on a web page, but also to train a system on how those objects should interact in ways that are meaningful to testing.

4 State of the Art in AI for Testing

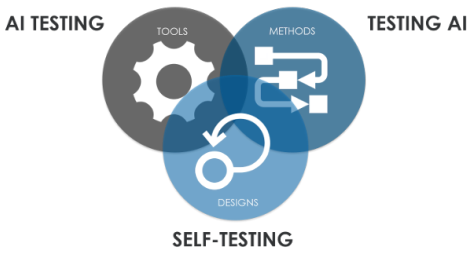
Companies such as test.ai (http://test.ai/) are making a significant impact in the field of AI for Testing. Built on top of AI-based approaches to mobile object recognition, reinforcement learning, and an abstract intent test language for expressing high-level test cases, the test.ai system is capable of automatically crawling through many mobile apps and is capable of reusing general test cases across concrete apps. In addition to test.ai, other products are leveraging ML to increase the efficiency of software testing. For example, mabl (https://www.mabl.com/) can self-heal test cases by relying on ML approaches to detecting web page objects at runtime. Applitools (https://applitools.com/) uses an AI-powered visual testing approach capable of detecting rendering issues and visual differences between app builds. Eggplant AI (https://eggplant.io/) is able to intelligently navigate applications and predict where quality issues are most likely to manifest themselves.

At Ultimate Software, quality research and development teams have worked on building tools that leverage AI for software testing. One such effort involved creating a system discovery and testing platform capable of leveraging ML to explore a SUT and discover states and interactions automatically. Reinforcement learning is used to seek interactions that yield high reward (discovering a new state, discovering a significant state variant). Image recognition and natural language processing techniques are also used to extract information encountered on web pages, such as error messages.

The platform enables the creation of agents that are tailored to perform specialized testing tasks. For example, an agent may be created that automatically explores a SUT looking for exceptions, stack traces, etc. A separate agent may explore a SUT while actively resizing the web browser, looking for rendering defects along the way. Using the techniques described in **Section 3.2**, a system may be trained to classify web pages as being improperly rendered under different screen sizes.

A significant problem that AI for Testing systems encounter is the domain knowledge gap problem. The knowledge gap problem stands in the way of being able to achieve tasks such as filling out complicated forms and being able to understand the expected behavior of applications with complex workflows. Without knowledge of expected system behavior, appropriate test oracles cannot be constructed. At Ultimate Software, ongoing research is being expended towards the creation of an AI-based Domain Expert (AIDE). Leveraging existing research on expert systems, we have been able to start building knowledge bases that aim to reduce the domain knowledge gap problem encountered by AI for Testing systems.

The Artificial Intelligence for Software Testing Association (https://www.aitesting.org/) defines three important problems in the space of AI and testing. *AI for Testing* focuses on applying AI in order to identify software quality issues, apply test inputs, validate outputs, and emulate users or other conditions. *Testing AI* focuses on methods for testing software where AI is a major component of functionality or purpose. *Self-testing* in the context of AI is a new area of research focused on how to enable systems to test themselves. While this paper focuses on exploring the *AI for Testing* problem, it is important to be aware of the ongoing work in each of the aforementioned areas.



5 Breaking into AI

If you are ready to take the next step and start learning more about AI to start contributing towards the true automation of software testing, there are many excellent resources to start from.

Several tutorials and massive open online courses (MOOCs) are available that range from covering foundational material to covering deeper domain-specific AI problems. Blogs such as Jason Brownlee's machine learning mastery blog can also be good resources for finding many applied ML examples.

The AISTA is another resource for learning more about this exciting space, and for getting involved. AISTA is focused on applying, and extending AI to the world of software quality in all forms, and hopefully obviating the need for human testing activities. Genuinely automating software testing is not just a fundamental problem, but also a challenging problem. AISTA serves as an open community that fosters collaboration towards the common goal of achieving more intelligent automated software testing. One of the goals of AISTA is to provide a way for the community to contribute towards building datasets that may be used for training ML systems. Also, AISTA strives to provide a platform for sharing knowledge on different AI approaches to various testing problems.

The following presentations/tutorials also provide insightful information into the world of AI and testing:

- Jason Arbon, "AI and Machine Learning for Testers", PNSQC 2017
- Tariq King, Keynote "Rise of the Machines: Can Artificial Intelligence Terminate Manual Testing?", StarWest 2017
- Paul Merrill, "Machine Learning & How It Affects Testers", Quality Jam 2017
- Geoff Meyer, Keynote "What's Our Job When the Machines Do Testing?", StarEast 2018
- Angie Jones, Keynote "The Next Big Things: Testing AI and Machine Learning Applications", StarEast 2018
- Jason Arbon and Tariq King, "Artificial Intelligence and Machine Learning Skills for the Testing World", StarEast 2018

The following books cover many topics in great detail for readers that desire to attain a deep understanding of AI and ML:

- "Artificial Intelligence: A Modern Approach", Peter Norvig and Stuart J. Russell
- "Python Machine Learning", Sebastian Raschka
- "Deep Learning with Python", François Chollet
- "Deep Learning", Ian Goodfellow, Yoshua Bengio, Aaron Courville

The following resources are freely available online:

- Book: "Neural Networks and Deep Learning", Michael Nielsen, http://neuralnetworksanddeeplearning.com/
- YouTube: A full Stanford course on Machine Learning, taught by Andrew Ng

The following resources cover topics specifically surrounding AI and testing:

- AI Summit Guild (https://aisummitguild.com) has presentations from many of the leaders in the field
- Jason Arbon provides many additional resources on AI and testing (https://www.linkedin.com/pulse/links-ai-curious-jason-arbon/)

6 Closing Remarks

This paper aimed to stimulate research and innovation in software testing through the use of AI and ML. We presented an example of our journey and experiences of applying existing AI research to testing. We have also provided critical takeaways for how quality professionals can get the necessary foundation to break into the AI world and start contributing to this ripe emerging field of intelligent automated software testing. We have also highlighted a few practical applications of AI research, as well as related work being done by several companies. Along the way, we have also raised several important and challenging outstanding problems.

Current web test automation approaches rely heavily on the construction of page objects that are coupled to the implementation details of the SUT. The research and practical applications presented in this paper show that it is possible to raise the level of abstraction for working with webpage components. We also defined test flows and a well-defined language capable of expressing test flows. Also, we leveraged ML-based text generation techniques to learn how to recall and generate strings that belong to the language. It follows that it may be possible to use a combined learning approach to automatically generate and execute test cases against a SUT.

The application of AI and ML to testing is not going to happen in the future. It is already happening right now. As we discussed in **Sections 4 and 5**, more and more companies are starting to pay closer attention to this problem. As research and development continues and more data is collected to be able to train smart AI-driven testing bots, we will get closer to reaching a new level of test automation. As we shift closer to “true automation”, human testers will need to adjust, and our roles will certainly be redefined. Within the next 5-10 years, it is quite possible that we will be experiencing a shift in testing culture and in testing responsibilities. Instead of spending time manually crafting automated scripts, we will be spending that time collecting and annotating more training data to make the bots even smarter. With thousands and thousands of these bots at our command doing the work we’ve taught them to do, it will help free us from the burdens of test automation scripting and maintenance, and allow us to focus on more important aspects of testing, such as exploratory testing and finding defects.

If nothing else, we hope that after reading this paper, the reader has increased awareness of how significant these problems are, and how important it is to start getting involved. We hope that by sharing our experiences in the fashion by which it was presented, it establishes a framework for others to follow and start contributing to these problems.

Appendix

This section presents our research and preliminary results for an *AI for Testing* approach that combines a trainable classifier that perceives application state, a language for describing test flows, and a trainable test flow generation model to create test cases learned from human testers.

In **Section 3.2**, we discussed the application of image-based classification to the problem of perceiving web application state. Aside from collecting and labeling images, as part of our research we collected and labeled structural information from web pages. To leverage structural information from a web page, we start by collecting a render tree. A render tree contains information on the DOM structure and about styling. A Computed Render Tree (CRT) representation of the webpage is then constructed. In contrast to a standard render tree, a CRT extends browser render trees by collecting the render tree only once the web browser has finalized rendering, and by calculating additional information such as: (1) element positions; and (2) element sizes.

Using the CRT representation, a feature synthesis step is then performed. An element-wise pass is done through all of the elements in the CRT, synthesizing several features for each element. Although the synthesis is done at the local level for each element, the full global context (information on the entire set of elements) is used to compute several features. The feature synthesis results in the generation of

training data that can be annotated for the purpose of training ML models. A set of example synthesized features is described in the following table:

Feature	Description
HTML Tag	The tag for the given element.
Parent HTML Tag	The tag for the given element's parent.
"For" Attribute	The existence of a value for the HTML "For" attribute.
Num. Children	The number of HTML nodes that are children of the given element's node.
Num. Siblings	The number of HTML nodes that are siblings of the given element's node.
Depth	The depth of the given element's node within the ADOM tree.
Horizontal Percent	The relative horizontal position (in percentage) of the given element.
Vertical Percent	The relative vertical position (in percentage) of the given element.
Font Size	The relative (normalized against the full set of elements) font size of the given element.
Font Weight	The relative (normalized against the full set of elements) font weight of the given element.
Is Text	Describes whether a given element is a text node.
Nearest Color	The closest color computed using CIEDE2000 algorithm.
Nearest Background Color	The closest background color computed using CIEDE2000 algorithm.
Distance from Input	The relative (normalized against the full set of elements) distance to the closest input widget from the given element.
Text	The actual text associated with the given element.

In order to evaluate the web classification approach, CRTs were collected and hand-labeled for 7 different SUTs, comprised of 95 web pages and 17,360 web elements. Among the labeled data were 122 elements labeled as page titles, 384 elements labeled as widget labels, 91 labeled as error messages, and 16,762 noise data points. For each of the three component classes analyzed, an experiment was done to compare the performance of the following ML classifiers: (1) random forests, (2) J48 decision trees; (3) k-nearest neighbor; (4) support vector machines; and (5) Bayesian networks.

The evaluations were done using percentage split and cross-validation. Performance was measured using accuracy, precision, recall, and F1 score (Goutte 2005). For the problem of classifying widget labels, the results show that the random forest algorithm performed best with an F1-Score of **96.3%**. When classifying error messages, the random forest technique also performed the best, resulting in an F1-Score of **99.4%**. The data suggests page title classification to be the most difficult of the classification problems that were investigated. The K-Nearest Neighbor algorithm performed the best with an F1-Score of **71.6%**.

	Label Candidates				Error Messages			
Classifier	**Accuracy**	**Precision**	**Recall**	**F1-Score**	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Random Forest (100 Estimators)	**99.83%**	**98.9%**	**93.8%**	**96.3%**	**99.99%**	**100.0%**	**98.9%**	**99.4%**
J48 Decision Tree	99.64%	95.0%	88.8%	91.8%	99.94%	91.8%	98.9%	95.2%
K-Nearest Neighbor (K=3)	99.61%	92.7%	89.6%	91.1%	99.98%	97.8%	98.9%	98.4%
SVM	99.29%	99.2%	68.8%	81.2%	99.97%	95.7%	98.9%	97.3%
Bayesian Network	99.13%	78.2%	84.1%	81.1%	99.88%	83.2%	97.8%	89.9%

	Page Titles			
Classifier	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Random Forest (100 Estimators)	99.52%	67.3%	62.3%	64.7%
J48 Decision Tree	99.56%	79.5%	50.8%	62.0%
K-Nearest Neighbor (K=3)	**99.64%**	**82.8%**	**63.1%**	**71.6%**
SVM	99.41%	85.7%	19.7%	32.0%
Bayesian Network	97.32%	17.0%	72.1%	27.5%

In **Section 3.4**, we mentioned the creation of a **language** that could be used to express test flows. In order to validate both the designed language, as well as test flow generation, a prototype was constructed. An Extended Backus-Naur Form (EBNF) (McCracken 2003) language parser implementation was developed using the Lark library available in Python (Lark 2018). A total of 250 test flows utilizing all of the features of the language were crafted and parsed successfully using the custom Lark parser. Each of the 250 crafted test flows captured real test cases that could feasibly be performed by a human tester against a web application. Using the sliding window data augmentation technique, a dataset containing a total of 2610 labeled examples for the purposes of using ML for test flow generation was created. The Keras framework was used to build the neural networks for the prototype.

The performance of the LSTMs were measured using the built-in categorical cross-entropy loss function available in the Keras neural network programming framework. Training set accuracy was based on whether or not a predicted test flow was a valid test flow sub-sequence in the training data. In addition, the output predictions were only considered valid if the resulting test flow was able to be parsed by the custom Lark parser. Different network topologies, optimization algorithms, and regularization technique settings were compared. The following table shows a detailed breakdown of the performance of each configuration that was evaluated:

Epochs	Layers	Units	Dropout	Optimizer	Accuracy	Epochs	Layers	Units	Dropout	Optimizer	Accuracy
50	1	32	No	Adam	63.62%	**500**	**1**	**256**	**No**	**RMSProp**	**93.85%**
50	1	32	Yes	Adam	39.86%	**500**	**1**	**256**	**Yes**	**RMSProp**	**93.85%**
50	1	64	No	RMSProp	84.17%	50	2	64+64	No	RMSProp	88.17%
50	1	64	Yes	RMSProp	82.63%	50	2	64+64	Yes	RMSProp	89.03%
50	1	64	No	Adam	81.57%	50	2	64+64	No	Adam	73.06%
50	1	64	Yes	Adam	84.92%	50	2	64+64	Yes	Adam	81.06%
50	1	128	No	RMSProp	88.55%	50	2	128+128	No	Adam	86.67%
50	1	128	Yes	RMSProp	82.63%	50	2	128+128	Yes	Adam	86.67%
50	1	128	No	Adam	88.68%	50	2	128+128	No	RMSProp	89.98%
50	1	128	Yes	Adam	85.64%	50	2	128+128	Yes	RMSProp	85.57%
50	1	256	No	RMSProp	59.59%	50	3	128*3	No	RMSProp	91.49%
50	1	256	Yes	RMSProp	87.15%	50	3	128*3	Yes	RMSProp	86.50%
50	1	256	No	Adam	91.52%	**500**	**3**	**128*3**	**No**	**Adam**	**93.74%**
50	1	256	Yes	Adam	86.19%	**500**	**3**	**128*3**	**Yes**	**Adam**	**93.85%**

Each trained LSTM model was used for generating a total of 2925 test flows. When evaluating each LSTM configuration, accuracy was measured by dividing the number of generated test flows that were valid strings in our language by the total number of generated test flows. Experiments included varying the number of training iterations (epochs), the number of LSTM layers, and the number of LSTM units per layer. The Adam and RMSProp optimizers were also evaluated, as well as dropout regularization. Based on the results, the highest accuracy (**93.85%**) was achieved when training a single LSTM layer network for 500 epochs, using the RMSProp optimization algorithm. Equivalent accuracy was observed when using multiple stacked LSTM layers. Generally, increasing the number of layers and units improved accuracy; however, training time was also observed to increase. Stacking LSTM layers appears to be a viable option for increasing accuracy as more training data is added. While dropout regularization generally reduced training set accuracy, it did not greatly affect accuracy when training for longer durations or when using larger networks.

References

Abadi, Martín, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Je rey Dean, Matthieu Devin, et al. 2016. “TensorFlow: A System for Large-scale Machine Learning.” In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 265-283. OSDI'16. Savannah, GA, USA: USENIX Association. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

AIST. 2018. *IEEE International Workshop on Automated and Intelligent Software Testing.* Http://paris.utdallas.edu/AIST18/ (Accessed March 18, 2018).

AISTA. 2018. *AI for Software Testing Association.* Https://www.aitesting.org/ (Accessed March 18, 2018).

Arbon, Jason. 2017. *AI for Software Testing.* In *Pacific NW Software Quality Conference. PNSQC, 2017.* Http://uploads.pnsqc.org/2017/papers/AI-and-Machine-Learning-for-Testers-Jason-Arbon.pdf (Accessed July 18, 2018).

Bojarski, Mariusz, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, et al. 2016. “End to End Learning for SelfDriving Cars.” *CoRR* abs/1604.07316. arXiv: 1604.07316. http://arxiv.org/ abs/1604.07316.

Brownlee, Jason. 2016. *Text Generation With LSTM Recurrent Neural Networks in Python with Keras.* Https://machinelearningmastery.com/text-generation-lstm-recurrent-neuralnetworks-python-keras/ (Accessed April 4, 2018).

Goutte, Cyril, and Eric Gaussier. "A probabilistic interpretation of precision, recall and F-score, with implication for evaluation." In *European Conference on Information Retrieval*, pp. 345-359. Springer, Berlin, Heidelberg, 2005.

Graves, Alex. 2013. “Generating Sequences With Recurrent Neural Networks.” *CoRR* abs/1308.0850. arXiv: 1308.0850. http://arxiv.org/abs/1308.0850.

LabelImg. 2018. *A graphical image annotation tool.* https://github.com/tzutalin/labelImg (Accessed August 10, 2018).

Lark. 2018. *A modern parsing library for Python.* https://github.com/lark-parser/lark (Accessed August 10, 2018).

McCracken, Daniel D., and Edwin D. Reilly. "Backus-naur form (bnf)." (2003): 129-131.

Moyer, Christopher. 2016. *How Google's AlphaGo Beat a Go World Champion Inside a man-versus-machine showdown.* Https://www.theatlantic.com/technology/archive/2016/03/theinvisible-opponent/475611/ (Accessed April 4, 2018).

Ng, Andrew. 2018. *Machine Learning (free online course on Coursera).* Http://www.andrewng.org/courses/ (Accessed March 10, 2018).

Stanford. 2018. *Machine Learning.* Https://online.stanford.edu/course/machine-learning1/ (Accessed March 18, 2018).

Xiong, Wayne, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. 2017. “The Microsoft 2017 Conversational Speech Recognition System.” In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP),* 5255-5259. doi:10.1109/ICASSP.2017.7953159.

Thanks to the following people for their help editing this paper: Rick D. Anderson, Keith Stobie, and Keith Briggs. Thanks to the following people for their contributions to AI for testing efforts at Ultimate Software: John Maliani, Robert Vanderwall, Michael Mattera, Brian Muras, Keith Briggs, David Adamo, Justin Phillips, Philip Daye, and Patrick Alt.