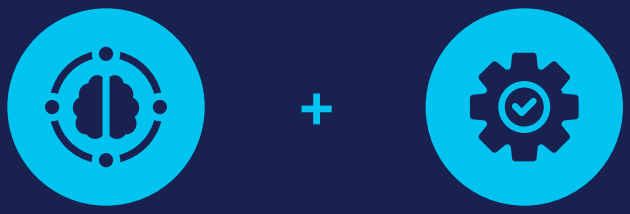


Artificial Intelligence + Software Testing:

How to Harness the Power of Machine Learning

WHITEPAPER

COPADO ROBOTIC TESTING

Abstract

This paper explores challenges and opportunities within the software testing space that rely on artificial intelligence — specifically machine-learning algorithms.

What is Artificial Intelligence?

Artificial Intelligence (or AI) is a loose term for any computer system that mimics human thinking. For decades, technologists have been leveraging AI for things like logical reasoning, pattern recognition and proximity calculation.

Rule-based AI

AI originated as rule-based artificial intelligence — where a computer is fed logical reasoning rules and follows them to track down answers to complex questions. Rule-based AI can quickly achieve things that are impossible for the human brain, but it is still deterministic and dependent on the rules you feed it.

The Rise of Machine Learning

Next, the emergence of big data resulted in the development of another type of AI: machine learning (or ML). While the mathematical models for ML had already been around for decades, the technology only became functional once the availability of data and computing power caught up.

In a nutshell, ML is a software guided by labeled data instead of logical reasoning rules. For example, ML can learn to identify anything from an image of a cat or a data record for a fraudulent bank transaction. The more cats and transactions the software sees, the better it gets at recognizing them.

Today, many innovative companies are harnessing the power of AI and ML to perform complex logical reasoning and make sense of their data. One key area of opportunity is in the realm of software testing.

Software Testing + AI

Software testing and artificial intelligence make an interesting pair. While test case design and test result analysis are still largely performed by humans, more and more businesses are turning to AI to save time and drive quality.

However, the connection between software testing and AI goes far deeper than simple use cases. The discipline of software testing is based on the idea that the tester is able to know — or at least guess — the expected result of a test in advance. Typically, the specifications of the software being tested explain how it's supposed to behave. The tester aims at finding a deviation between specified behavior and actual behavior.

When you throw AI into the mix, it mirrors human brain waves for better or worse. If AI is executing a task that would be easy for a human brain (image or text recognition), the tester's job is simple and straightforward. But if the AI is doing something difficult for the human brain (pinpointing patterns in scientific data), the tester may need to reconstruct the AI's logic to figure out the expected result. Such an effort may be impossible — or at least very slow.

While ML applications are trained with an initial data set, you need to re-train it with more information to improve the algorithm. Keep in mind — introducing new data may impact the way the algorithm treats the initial data set. What's more, some ML algorithms are programmed to learn something new every time they are used.

In each instance, the algorithm's thinking ability is stochastic — it contains a fair amount of randomness. This means ML software isn't fully deterministic like rule-based AI. Just like the human brain, it can be influenced by false data and fake

news. As a result, ML applications can learn to make wrong conclusions.

And just like humans, ML applications can fall short in their efforts to explain how they came to their conclusion — making it a challenge to determine whether changes and improvements made the intelligence better or worse.

A System that Thinks Is Too Complex for the Human Mind

In many ways, attempting to reconstruct the logic of AI is like trying to understand how another person thinks. The outcome is clear. The input data is clear. But the process in between remains murky and mysterious. The algorithm is simply too complex to be followed and imitated by a human mind. In addition, AI learns every time it's fed new data — which may change the output of subsequent runs.

Let's explore a real-life example: an AI algorithm that is fed information about a credit card transaction and determines how likely a transaction is a fraud.

The input data may include the content and location of the transaction, the card number and the card holder's transaction history. The algorithm has also likely been programmed to identify suspicious transactions and trained with a wealth of data to distinguish between fraudulent and legitimate transactions.

In this example, even the most advanced algorithm is unlikely to be 100% reliable. Why? Because human behavior is hard to predict — and identity thieves are often adept at replicating the buying patterns of the cardholder.

To combat cybercrime, you can create a self-learning algorithm (known as unsupervised ML) or retrain the algorithm frequently. In an ideal world, the algorithm would expand its knowledge every time a proper transaction is processed. But in reality, ML algorithms often do a poor job of explaining how and why they arrived at their conclusion. An infamous example of this is Tay — an experimental Twitter bot launched by Microsoft in 2016. But unfortunately, Tay was quickly influenced by the wrong crowd:

> "Tay was an AI chatterbot that was originally released by Microsoft via Twitter on March 23, 2016. It caused subsequent controversy when the bot began to post inflammatory and offensive tweets through its Twitter account, causing Microsoft to shut down the service only 16 hours after its launch. According to Microsoft, this was caused by trolls who "attacked" the service as the bot made replies based on its interactions with people on Twitter.

The Seven Steps of Software Testing

Testing the functionality of a piece of software is a simple process — at least in principle. Here are the seven core steps:

1. **Figure out how to interact with the software.**

   For instance, when testing a login screen, you can (usually) type in a username and a password and click an OK or Cancel button.

2. **Determine what data you need to successfully interact.**

   On the login screen, you only need username and password info.

3. **Plan a flow of user actions and related input data.**

   Even the simplest applications include several action flows. A tester looks at the normal flow as well as exceptional or abnormal flows. You may test the software by clicking OK without entering any username or password or you could check to see if it makes any difference whether you enter the password first and the username second.

4. **Learn what is supposed to happen when you execute the flow.**

   If you don't decide the expected result in advance, you risk accepting erroneous behavior as "expected." If you click OK without entering a username at all, the expected result is an error message and an opportunity to try again.

5. **Execute the flow and record what happened.**

6. **Compare what happened to what should have happened.**

   An error or defect is found if the expected and actual outcomes are different.

7. **Need to test again? Go back to step 3.**

How Do You Determine Correct Behavior?

Number 4 is often the trickiest step. How does the tester know what should happen?

A software specification may describe the correct behavior — or the tester might refer back to a previous version of the software that behaves correctly. While most testers understand how a login screen is supposed to behave, things could get dicey if you're testing a more complex flow.

Sometimes, the tester makes a guess about the correct behavior and assumes most users would follow the same line of thinking. This principle is known as exploratory testing.

Any source that defines the correct behavior of the system being tested is known as a "test oracle." When testing AI applications, a test oracle may be hard to track down. For instance, it is both tedious and difficult to construct credit card transactions and label them as fraudulent or non-fraudulent.

Designing Test Cases

Selecting, designing and maintaining test cases is a tester's most important task. The test case design is always based on the existence of a test oracle that knows the expected outcome of the test case.

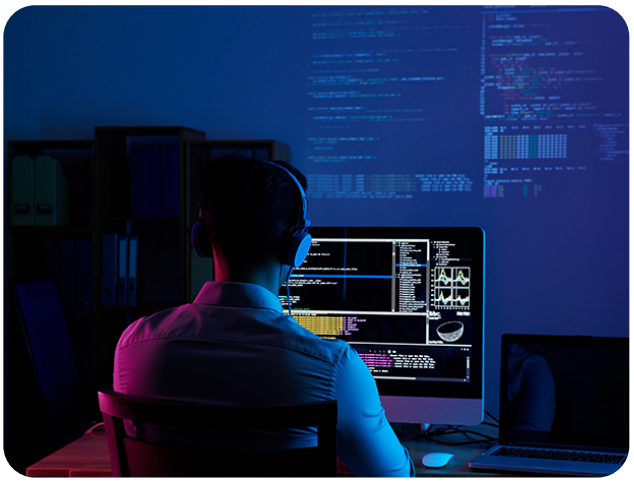
In our example of credit card fraud detection, there is no explicit definition of a fraudulent (or possibly fraudulent) transaction. Finding the right outcome is like investigating a crime scene: you must collect information, use small clues to build the big picture, make a verdict and estimate probability.

In most ML applications, the functional logic of a test case is simple. You feed it data and it spits out an answer: "The probability that this is an image of a cat is 94.2%." The hard part is finding, creating and labeling an input data set that is representative enough: especially when the subject matter is more complex than cat identification.

The Pitfalls of Limited Testing

Testing ML software requires a large number of diverse, high-quality test cases. It is usually possible to have a limited set of cases (10 credit card transactions) that are known to be fraudulent and 10 that are known to be proper. Testing these 20 cases can give you confidence in the correct behavior of the software and helps detect whether a new version of the software is still working as expected.

Unfortunately, these limited tests tell you nothing about how the software has improved or if any new problems have reared their heads. Without a proper test oracle, crafting new tests can be tedious — if not impossible.



4 Ways to Construct Test Data for an ML System

There are at least four different approaches to assemble test data for a learning system: manually crafted test data, testing with real production data, combinatorial test data and AI-assisted test data design.

Manually Crafted Test Data

Let's examine these approaches in the context of our credit card fraud detection example. In all likelihood, the test input data arrives in the form of transaction records and structured files (XML or JSON format). It may even contain natural language. A human can certainly construct such data and label it as proper or suspicious — but the effort may be extremely demanding and technical skills may be required.

Testing with Real Production Data

If real production data is available, testing with it can be a great choice. In our fraud detection example, this would mean taking real data from old transactions that have either been proven legitimate or fraudulent. However, this approach is illegal in many countries with laws and regulations that protect the privacy of personal and financial data.

Combinatorial Test Data

For this approach, you divide the needed test data into separate parts (transactions, card info, purchase locations, purchase history, etc.) and combine individual parts to generate large data sets. In addition, you could have a software generator add random data or randomly mutated data.

There's just one problem with this approach: these "data atoms" are rarely independent. Building test data by combining one person's card transaction with someone else's purchase history and a third individual's card data might not result in a viable test case.

AI-assisted Test Data Design

Finally, with AI-assisted test data design, you use the beast to tame itself.

For this approach, you leverage an intelligent algorithm to process test data and results and generate new test data based on the old one. This is essentially the combinatorial approach on steroids. However, the more machine intelligence in the test data generation, the harder it is for a human to follow the applied logic.

This mimics the concept of Generative Adversarial Networks where one algorithm tries to fake real data and the other tries to detect which data is genuine and which is false.

No matter which approach you choose, the basic challenges remain the same: how do you create enough diverse test data and label it reliably?

Dealing with Regression: Does My Software Still Work?

A software tester seeks answers to two big questions:

1. **Does the software do what is expected?**
2. **Does the software do something unexpected?**

When testing new versions of the same software, a tester looks at what has changed. There might be new functionality that has never been tested before or old functionality that used to be defective but has been fixed. The older software gets, the more important it is to make sure what worked yesterday still works the same way today.

If the software has a large number of users, it may even be practical to leave old defects as-is and verify they're still there. For example, you could have an environment-sensing device that reports certain temperatures as Fahrenheit due to a design error but uses Centigrade across all other functions. This error might be a nuisance, but if numerous people and applications already know about it and have worked around it, correcting it could break the behavioral assumptions and cause many applications to fail.

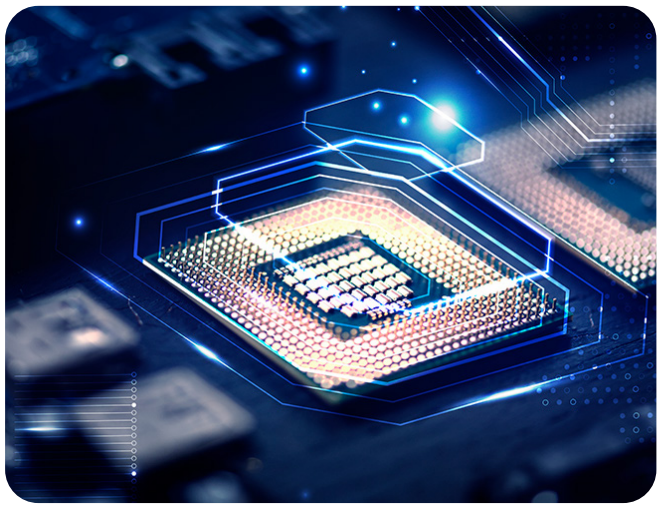
Regression testing ensures that the behavior of the software hasn't changed unintentionally between subsequent versions. But with ML applications, regression testing features a new twist: how do you know if a change was intentional?

It's common practice to use an older version of the software as a test oracle and compare how the behavior of a newer version differs. This is an effective strategy when the new software behaves the same way as the old — but it can't help you out if the new version includes different functionality.

Let's boomerang back to our credit card example one last time.

Imagine that a new version of the software classifies a transaction as valid while the previous version classified it as possibly fraudulent. How do you know if this is an improvement or an error? You'll most likely need to go deep into the test data and decision-making rules to make a judgement call.

To make matters worse, the fact that the software made a false conclusion could mean that the algorithm is less accurate than before. You'll have to consider how many transactions the new algorithm classified correctly compared to the old one — and how the cost of erroneously classified transactions changed.



Searching for a Recipe

There is no one-size-fits-all solution to a challenge as broad as testing AI software. However, all viable solutions have a few things in common.

Always test with your initial data set

If your application uses any kind of ML, it has been trained with an initial data set. This is your first test asset and in some cases may be the only reliable test data you have. Keep in mind — testing with the initial data set can only tell if your software can still do the things it was able to do when it was created.

Accumulate test data continuously

At this point, you need more test data. For most AI applications, collecting enough data is a bigger challenge than figuring out the test case logic. Sometimes you can pick real data from production, in other cases it has to be engineered by the testing team. Sometimes your software may malfunction in production.

Any data the software can't process correctly is potential input for future tests. If possible, use combinatorial techniques to generate more test data from existing sets. And take good care of the quality of your data! Accumulating any old data may make your test set large — but it isn't likely to result in precise outcomes.

Use many test oracles

Depending on what your software does, it may be very easy or very difficult to determine the expected outcome of each test. Can you manually label your test data? You'll be fine. If that's not possible, you'll need to come up with a different plan. You may also have an opportunity to use several ML algorithms for the same data and compare their outputs. Sometimes, the best you can do is to rely on the earlier versions of the same software as the "source of truth."

Compare subsequent test runs

Any change in test results between runs tells that your software has evolved. You may not be able to tell if the change was for better or worse — but you'll notice that something has changed. Explainability is the Achilles' heel of most AI applications, so be prepared for tedious digging. An application with built-in explainability will help to test and accelerate time-to-value.

Leverage AI and automation in testing

The world is full of marvelous testing tools that can automate many mundane software testing tasks. However, none of them are particularly well equipped for testing AI applications. You may need to complement commercial products with custom-built tools and scripts that generate suitable test data or automate the analysis of test results. Perhaps you can find a way to leverage AI to test AI.

Artificial Intelligence Is Just Algorithms and Data

In many ways, AI is no different than any other software. It consists of programmatic algorithms and data that were applied to train those algorithms.

The key differences are in learning, randomness and explainability. Unlike traditional computing algorithms, AI's behavior may change when it learns from new data. There is a random element in many AI algorithms — making them only partially deterministic. And finally, because of learning and randomness, their logic may be hard for a human being to reconstruct or explain.

These characteristics can make AI a challenge to test. While the traditional methods of software testing still apply for AI, the amount and quality of test data are much more important.

Automating testing tasks can drive speed and efficiency since many aspects of AI testing are too tedious or overwhelming for the human mind. As the practice of using AI to test AI evolves, this discipline could open completely new avenues in software testing and will also likely be leveraged to test traditional software systems.

COPADO | #1 Native DevOps for Salesforce

GET A DEMO