Artificial intelligence and testing

Written by Esko Hannula

October 2020

Abstract

This paper discusses the challenges and opportunities in the testing of software systems that include or rely on artificial intelligence, particularly machine-learning algorithms.

Contents

Introduction 3
A system that thinks is too complex for the human mind 4
The process of testing 5
Finding test data 6
Dealing with regression: does my software still work? 7
Searching for a recipe 8
Ultimately, artificial intelligence is just algorithms and data 9

Introduction

"Artificial Intelligence" or AI is a loose term that may refer to any computer system that carries out activities that even distantly resemble human thinking. Examples of such activities are logical reasoning, pattern recognition, or proximity calculation.

AI has been a part of computing for decades already. It originated as rule-based artificial intelligence where a computer system is fed with logical reasoning rules and it uses those rules to find answers to questions that require logical reasoning. Such reasoning may be very complex and impossible for the human brain to perform. Still, it is deterministic.

The emergence of big data brought along another type of artificial intelligence, often referred to as "machine learning" or briefly "ML". The mathematical models for machine learning had been around for decades already but became usable only with the easy availability of data and computing power. The idea is to have a software algorithm that is taught with labeled data rather than by rules. The software algorithm learns, for example, what an image of a cat looks like or what a data record for a fraudulent bank transaction looks like. The more cats and transactions the software sees the better it gets at recognizing them. This is why it's called "learning".

The examples above, although rough simplifications, illustrate the state of the of "artificial intelligence" today as well as the opportunities to utilize it. The main opportunities are in complex logical reasoning and in making sense of data.

Software testing and artificial intelligence make an interesting pair. There are a plethora of activities in software testing that can benefit from artificial intelligence. Obvious examples are test case design and analysis of test results that are still today largely human activities.

But there is another, more interesting connection between software testing and artificial intelligence. The whole discipline of software testing is based on the idea that the tester is somehow able to know, or at least guess, the expected result of a test in advance. Usually, the expected result is provided by the specification of the software under test. The specification tells how the software is supposed to behave and the tester aims at finding a deviation between specified behavior and actual behavior.

AI applications can be challenging to test. In all testing, the tester has to find or create relevant input data and then somehow determine the expected result, i.e. what the software under test is supposed to produce as output or result from that particular data. If the task the AI is carrying out would be easy for a human brain, e.g. image or text recognition, the tester's job isn't that difficult. But if the AI is doing something that is overwhelmingly difficult for the human brain, e.g. digging out patterns from consumer data or scientific data, the tester may need to somehow reconstruct the logic of the artificial intelligence to figure out the expected result. Such an effort may be impossible or at least very slow.

Machine learning applications are "trained" with an initial data set. To improve the algorithm one needs to re-train it with more data. New data may, of course, change how the algorithm treats the initial data. Some machine-learning algorithms may even be programmed to learn something new every time they are being used. In all cases, the machine-learning algorithms tend to be stochastic, which means their "thinking" contains a fair amount of randomness. Thus, machine-learning software is not fully deterministic in the way traditional software applications are. Just like the human brain can be influenced by false data and fake news a machine learning application can learn to make wrong conclusions. And just like human beings often fail to explain the logic of their thinking, machine learning applications, too, fall short in their capability to explain how they came to their conclusion. The challenge of testing is to figure out whether changes and improvements made the intelligence better or worse.

A system that thinks is too complex for the human mind

Applications of artificial intelligence do not, of course, think in the way humans do but there is some resemblance to human thinking.

An AI application may carry out logical reasoning using a complex algorithm that, although fully deterministic, can be too complex for a human mind to follow. This is different from the majority of computer algorithms that are simple enough to be executed by human brains. A human being trying to reconstruct a logic of AI is a bit like a human being trying to reconstruct the line of thought of another human being: the outcome is clear, the input data is clear, but the process in between remains largely unexplained.

The algorithm itself, although designed by a human being is itself too complex to be followed and imitated by a human mind. Moreover, the algorithm learns every time it meets new input data. Even though a human being could be able to repeat the algorithm it is not always certain whether the output of subsequent runs of the algorithm remains the same.

Let us consider an AI algorithm that takes information about a credit card transaction as its input and tries to determine how likely that transaction is a fraud. The input may contain data such as the content and location of the transaction, card number, and the card holder's transaction history. The AI algorithm has probably been programmed to identify suspicious transactions and trained with a wealth of data to distinguish between transactions that are fraudulent from those that are not.

Such an algorithm can rarely be 100% reliable because there are always gray areas. People sometimes make purchases very different from their normal purchasing patterns and the bad guys may be very competent in constructing unauthorized transactions that match the buying patterns of the authorized cardholder.

The detection algorithm has to learn all the time to remain effective. This can be achieved by creating a self-learning algorithm, known as unsupervised machine-learning, or re-training the algorithm frequently. Ideally, while processing a transaction, the algorithm would also accumulate its knowledge on how a proper transaction looks like. This is brilliant, of course. However, just like human beings, machine learning algorithms do a notoriously poor job explaining how and why they made their conclusion. Therefore, there is no reliable way to ensure that the learning of the algorithm actually makes it better.

An infamous example is Tay, an experimental Twitter bot published by Microsoft in 2016. Wikipedia summarizes the story of Tay as follows:

Tay *was an artificial intelligence chatterbot that was originally released by Microsoft Corporation 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.[1] 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.*

Learning algorithms, at least those based on reinforcement learning, can learn bad habits, too. Just consider the possible consequences if our credit card checking algorithm, for whatever reason, began to learn that some fraudulent transactions are, in fact, valid.

The process of testing

Testing the functionality of a piece of software is a very simple process. I mean, it's very simple in principle.

1. *Figure out how you can interact with the software.*
   For example, when testing a login screen, one can (usually) interact by typing in a username, a password, and by clicking an OK button or a Cancel button.
2. *Figure out what data you need to interact with the software.*
   On the login screen, you only need usernames and passwords, some of which would be correct and some others incorrect.
3. *Plan a flow of user actions and related input data.*
   Even in a simple application, there are usually various flows of actions. A professional tester looks both at the "normal" flow and exceptional or abnormal flows. You may test clicking Ok without entering any username or password or you may test if it makes any difference whether you enter the password first and the username second.
4. *Figure out what should happen as a consequence of executing the flow.*
   This is where professional testers shine. One should always decide the expected result in advance. If you don't do so, you'll easily accept erroneous behavior as "expected". If one clicks OK without entering a username at all, the expected result is an error message and an opportunity to try again.
5. *Execute the flow.*
6. *Record what happened.*
7. *Compare what happened to what should have happened.*
   In testers' jargon, "error" or "defect" is found if expected and actual outcomes are different. Nowadays professional testers call these differences using a more neutral term "finding".
8. *If you think you have not tested enough, go back to step 3.*

Of these steps, number 4 is the most interesting one. How does the tester know what should happen? There may exist a specification that describes the correct behavior, or there may exist a previous version of the software that is considered to behave correctly. Sometimes, the tester may just rely on his or her experience. The behavior of a login screen can be considered commonly known and there aren't that many alternative behaviors.

Sometimes, the tester first makes a guess about the correct behavior and then assumes most users are like him or her and would thus assume the same. This principle is the basis of a discipline known as exploratory testing.

Any source that defines the correct behavior of the system under test is known as "test oracle" in testers' jargon. When testing AI applications, a test oracle may be hard to find. It is both tedious and difficult to construct credit card transactions and label them as fraudulent or non-fraudulent. Sometimes, of course, it is not that difficult. This is probably why recognizing cats in images is commonly used for testing machine learning algorithms.

Finding test data

A well-defined flow of events through an application, with associated data and expected outcome, is known as a test case. 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, i.e. a specification or some other "source of truth" that knows the expected outcome of the test case.

Sometimes test case design is far from trivial. In our example of credit card fraud detection, there is no explicit specification of what makes a fraudulent or possibly fraudulent transaction. Finding the right outcome is like crime scene investigation: collecting a lot of information, constructing a big picture from small clues, making a verdict, and estimating its probability.

In most machine learning applications, the functional logic of a test case is simple. One feeds in data and gets out an answer, such as "this is, on the probability of 94,2%, an image of a cat". The hard part is to find or create a set of input data that is representative enough and to label it correctly; especially if it is something more complex than an image of a cat. Please note, that while recognizing an image of a cat is easy for a human being it is difficult for a computer algorithm while recognizing a fraudulent credit card transaction may be easier for an algorithm than for a human being.

Testing machine learning software would need a large number of diverse, high-quality test cases but they are hard to build. It is usually possible to have a limited set of known cases, e.g. 10 credit card transactions, that are known to be fraudulent and 10 that are known to be proper. Testing with those 20 cases gives some confidence in the correct behavior of the software and helps, for example, detect if a new version of the software is still working as expected.

Such limited tests help ensure the software still does what it could do before. They tell nothing about whether and how the software has improved or if new problems have been introduced. Lacking a proper test oracle, crafting such new tests may sometimes be very tedious, if not impossible.

There are at least four different approaches to constructing test data for a learning system:

1. Manually crafted test data.
2. Testing with real production data.
3. Combinatorial test data.
4. AI-assisted test data design.

Let's assess these approaches in the context of the credit card fraud detection problem.

It is possible to construct test data manually. Most probably, the test input data comes in the form of transaction records and structured files, for example in XML or JSON format, or may even contain natural language. A human being can certainly construct such data and label it as proper or suspicious. It's just that the effort of doing so is high and may require technical skills.

Testing with real data, if available, is often a great choice. In our case study, this would mean taking real data from old transactions that have later on been proven righteous or fraudulent. While this approach has its merit, it may be illegal in many countries as the privacy of personal and financial data is strongly protected by laws and regulations. There are, however, many other applications where testing with real data is a viable choice.

The combinatorial test data approach may work well for our case study. The idea is to divide the needed test data into separate parts, such as transaction data, card data, purchase location data, purchase history data, etc., and make combinations of those parts so that a large number of data sets can be generated from a reasonably small amount of well-known data. Moreover, one could add random data or randomly mutated data. This is, of course, best done by some kind of software generator rather than a human being. The problem is that those "data atoms" are rarely independent. For example, building test data by combining a card transaction of one person with a purchase history that represents another person and card data that represents a third person may not make a viable test case. This approach works well when test data is complex but can be divided into independent or quasi-independent pieces that each can be labeled reliably.

Finally, AI-assisted test data design means using the beast to tame itself. There exist few, if any, general-purpose solutions for AI-assisted test data design but the idea is simple: use a more or less intelligent algorithm to process past test data and test results and generate new test data based on the old one. In a way, this is like the combinatorial approach with steroids. Explainability may become a challenge, again. The more machine intelligence there is in the test data generation the less capable a human being is of following the logic that was applied. This approach resembles the concept of Generative Adversarial Networks where one algorithm tries to fake real data and the other tries to detect what data is genuine and what is faked.

No matter which approach you choose, the basic challenges remain the same: how to create a large enough amount of diverse test data and how to label it reliably.

Dealing with regression: does my software still work?

A software tester seeks answers to two big questions: does the software do what is expected and does the software do something that is not expected. When testing subsequent versions of the same software, a tester looks at what has changed: there is a piece of new functionality that has never been tested before, there is old functionality that should still work as before, and there is old functionality that used to have defects but has been fixed and should now behave differently.

The older the software gets the more important it is to test that what worked yesterday still works the same way today. Sometimes, especially if the software has a large number of users, it may even be practical to not correct old defects but to verify that they still are there. A well-known example is an environment sensing device that, because of a design error, reports certain temperatures as Fahrenheit while in all other functions it uses Centigrade. Such an error is a big nuisance, but if there are already several people and applications that know about it and have worked around it, correcting the error would actually break the assumptions on the behavior and thus make many applications fail.

Testers use the phrase ”regression testing” to refer to the activity of ensuring that the behavior of the software has not changed unintentionally between subsequent versions. With machine learning applications, regression testing comes with a new twist: how does one know if a change was intentional?

It is a rather common practice to use an older version of the software as a test oracle and compare how the behavior of a newer version differs from the behavior of the older one. This works well when the new software happens to behave the same way as the old. It does not work for new functionality, though. It may also fall short if the test results for the new version are different.

Building on our credit card example again: imagine that a new version of the software classifies a transaction as valid while the previous version classified it as possibly fraudulent. How do we know if this is an improvement or an error? The only way is to make some kind of human judgment. This requires skill and knowledge as the tester needs to go deep in the test data and in the decision-making rules. It may also turn out to be very slow and expensive if such changed regressions are many. Moreover, the fact that the algorithm made a false conclusion on data that is previously had recognized right may not mean the algorithm is now worse. The test results have to be assessed statistically, maybe even applying some kind of cost function. In this particular example, one has to consider how many transactions in total the new algorithm classified correctly compared to the old and how did the cost of erroneously classified transactions change.

There is no general solution to the regression testing challenges. The fundamental question is general, though: has the algorithm evolved for better or for worse.

Searching for a recipe

It would be foolish to seek a single, ideal solution for such a broad challenge as testing of AI software. All viable solutions, however, seem to have a few things in common.

Always test with your initial data set
If your application uses any kind of machine learning it has been trained with some kind of initial data set. This data set is your first test asset and, in the beginning, it may be the only reliable test data you have. Whenever you need to test your software start with this data set.

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. Therefore, it won't take you far.

Accumulate test data continuously
You need more test data. In most AI applications it is the data rather than the logic of the test case that makes the test design a challenge. Therefore, you need to keep accumulating test data. Sometimes it is possible to pick real data from production, sometimes it has to be engineered by the testing team. Sometimes your software may malfunction in production.

The only silver lining in that cloud is that you just found very useful new test data. Any data that the software could not process correctly is potential input for future tests. If possible, use combinatorial techniques to generate more test data from existing test data. Take good care of the quality of your data. Accumulating just any data may make your test set large but not necessarily any better.

Use many test oracles, if possible
Depending on what your software does, it may be very easy or very difficult to determine the expected outcome of each test. If you can manually label your test data, you will be fine. This is the case with the infamous cat pictures, for example. If you cannot, you need to come up with something else. Sometimes you may have an opportunity to use several machine learning algorithms for the same data and compare their outputs. Often, 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 subsequent runs tells that your software has evolved somehow. You may not always be able to determine if the change was for better or for worse - or maybe both at the same time. But you will notice if something has changed and can then start digging deeper. As explainability is the Achilles' heel of most AI applications, be prepared for tedious digging. Any explainability that can be built in the application will help to test and thus accelerate time to value.

Use AI and automation in testing
The world is full of marvelous testing tools that can automate many mundane tasks of software testing, even when testing AI. However, none of them are particularly well equipped for testing AI applications. You may need to complement commercial tools with your tools and scripts that generate suitable test data or automate the analysis of test results. While such general-purpose tools may not yet exist building a tool to suit the specific needs of your application may be a lesser effort. Maybe you can find ways to apply AI to test AI.

Ultimately, artificial intelligence is just algorithms and data

Technically, “artificial intelligence” is not that different from any other software. It is comprised 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 algorithms may change their behavior when they learn from new data. There is a random element in many of those 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 make AI sometimes challenging to test. As we have shown in this paper, the traditional methods of software testing still apply for AI, too. The amount and quality of test data are even more important than with traditional applications. Automation of testing tasks is necessary because many aspects of AI testing are simply too tedious or overwhelming for the human mind. The use of AI to test AI may open completely new avenues in software testing and, at a high probability, will be increasingly used also for testing “traditional” software systems.

Esko Hannula

CEO, Qentinel

esko.hannula@qentinel.com

Qentinel | Robotic Software Testing

info@qentinel.com