Challenges

* AI and ML systems are non-deterministic – this means that they tend to show different behaviours for the same input.
* Steps need to be taken to ensure data is adequate and accurate – preparing training data is very important in ML models.
* There is a massive level of difficulty when extracting specific attributes – for example trying to explain why a model predicted something wrong.
* Human bias is typically present in training and testing datasets. It should be identified and removed in AI model test situations.
* The behaviour of ML models may change each time the data training is updated.

Constraint Programming Testing

* Cannot be easily handled by existing approaches because of the two following reasons: firstly, constraint programs are instrinsically non-deterministic as they represent sets of solutions and conventional definitions of conformity do not apply. Secondly, the refinement process of constraint programs is specific to CP.
* Constraint solvers are notoriously difficult to debug – in large part this is due to the difficulty of pinpointing the source of an error in the vast searches that solvers perform, since the effect of an error may only come to light long after the error is made.
* A major source of error in constraint solvers is the complex constraint propagation errors algorithms that provide the inference that controls and directs the search.
* Given an input for a system, the challenge of distinguishing the corresponding desired, correct behaviour from potentially incorrect behaviour is called the “test oracle problem”. The use of oracles involves comparing the output(s) of the system under test, for a given test-case input, to the output(s) that the oracle determines that product should have.
* One way of testing constraint solvers is by doing *metamorphic testing*.
* Some other issues include the following
* Constraints with a large domain of input requires huge amounts of memory.
* Only one solution is generated at a time.
* There is no guarantee uniformity in randomization – e.g suppose we have a constraint 0 < X < 11. If we randomize X a number of times and each time it may return the value 0, this meets the constraint, but we are interested in different values for X. So if we repeat the randomization process several times it doesn’t guarantee that a new solution will be generated each time (if there is more than one solution).
* One property based testing tool for CP solvers is outlined in the following paper - <https://www.info.ucl.ac.be/~pschaus/assets/publi/cp2019-solvercheck.pdf>.
* This tool is called SolverCheck, and adopts the so-called *property-based testing* paradigm which tackles the weaknesses of the classical *example-based testing* methodology.
* Example based testing relies on a tester to describe concrete situation – for example, with actual variables and domain instantiation. By combining many such examples, the tester creates a broad test suite covering a large number of potential problems.
* Property-based testing addresses some of the weaknesses of this approach by a combination of fuzzing and formal specification. With PBT a tester must express the general *properties* that must hold for all executions of a given software rather than manually crafting lots of *test cases*. These properties are expressed in a high-level declarative language which abstracts away the details of actual test cases.

Metamorphic Testing of CP Systems

* Property-based software testing technique, which can be an effective approach for addressing the test oracle problem and test case generation problem – i.e the difficulty of determining the expected outcomes of selected test cases or to determine whether the actual outputs agree with the expected outcomes.

There is a lack of standard literature on the methodology or approach to be adopted while testing.

* The actions and responses of the models in question differ over time, based on input data and thus are less predictable than traditional IT systems.
* Traditional testing techniques are based on fixed inputs and produce certain fixed outputs.
* Given the changing environment, tests such as unit testing of individual components and end-to-end tests of running systems, though valuable, are not enough to provide evidence that a system is working as intended.
* Issues also occur with code coverage – in ML the more relevant metric is parameters of the model executed.
* Live monitoring of system behaviour in real time combined with automated response is critical for long-term reliability.
* Significant focus should be put on accuracy-based testing of trained models, the data sets used to train and test these models, and the testing techniques used.

Black box testing techniques on ML models

* Model performance testing – testing with test data sets and comparing the model performance using metrics such as f-score, confusion matrix.
* Algorithm ensemble – multiple models using different algorithms are built and predictions from each of them are compared, given the same input data set.
* Coverage guided fuzzing – test all of the feature activations – e.g for a neural network activate each of the neurons/nodes.

**Testing ML models**

Stages of building ML model

Collecting data – where is it coming from? Where is it being stored, and how costly is this? Is the data streamed. Understand the data journey through the pipeline.

Preparing – training step needs data that is in good shape, collected data may not be ready for training. We need to check what shape the data needs to be in (check), what data types are used, are the missing values? Testing at this stage is crucial.

Training – experimentation, analysis and validation happens here – tune weight and parameters. At this stage exploratory and automated tests should be undertaken.

Testing the models – (even thought testing happens at each step) – how well does model perform against data set it was *not* trained on?

Deployment – at this stage its important to check where the model is hosted. If it’s a REST or GaphQL service, we can use API testing tools and techniques.

Observe and iteratie – monitor the model, measure the model, learn from the model etc.

Where do testers fit in ML models

Lessons learned from badly made models – poor data quality, easy to game the system, too dependent on the ML system, bias from selective focus, risk from external factors.

MLOps – a set of practises that aims to deploy and maintain ML models in production reliably and efficiently.

1. Collect data
2. Prepare data
3. Train the model – experiment, measure, validate
4. Test models – performance and behaviour
5. Deploy – CI/CD, reliability
6. Observe (learn and iterate) – monitoring, insights, usage

High quality MLOps requires good testing techniques

Adversarial attacks

A big part of AI Quality is reliability, robustness, security, privacy. This may cause inconvenient behavior like self driving cars fooled into thinking that a stop sign is actually a speed limit sign.

Data scientists and ML Engineers need to prevent, mitigate, and defend against these attacks and weaknesses. However, that’s easier said than done. Having a tester that looks at these quality attributes and can explore and design tests and attacks would be invaluable on the team.

Even without much ML/AI knowledge, you can still perform tests and attacks like Black Box Attacks. Nose and semantic attacks are simple, yet effective, but you can build on top of these and come up with even better strategies and techniques. As a tester you can play a crucial part in Threat Modelling and exposing these risks and weaknesses. Be creative with your exploration, experiments, tests and attacks.

Attack categories

* Black Box Attack – attacker has no information about the model.
* White box attack – attacker has complete access to the parameters and the gradients of the model.

Each of these have two subcategories

* Targeted attack – attacker perturbs or manipulates the input to try and make the model predict a specific target class.
* Untargeted attack – the attacker perturbs or manipulates the input to try and make the model predict the wrong class.

Different attack techniques

* Noise – adding random pixels to make part of the image meaningless.

Behavioral testing

We can apply a lot of the same types of testing that we are familiar with from software testing and engineering

* Unit Testing
* Risk Analysis
* Heuristics
* Testing Behaviors

Minimum Functionality Tests (MFT)

* Mainly testing for functionality and correctness – these are the things that must be true for the model to be considered good enough to progress.
* For example we may predict the following sentence “Soundtrack and animations were out of this world!” to always be “Positive” and within the 30-70% range (for a movie review model).

Invariance tests (INV)

* Same predictions after removing, adding or editing words – in other words, changing a word for a word of the same sentiment shouldn’t change the classification (using example of movie review model).
* This is more about robustness.
* E.g “soundtrack and animations were out of this world!” and “music and animations were out of this world!”.

Directional Expectation Tests (DIR)

* Sentiment should decrease or increase as expected – in other words, adding a word to the setnece should move the sentiment in the expected direction given its weight.
* As an example, if “soundtrack and animations were out of this world!” predicts 50% positivity, “Soundtrack and animations were amazing and out of this world!” should predict more than 50% positivity.

Automating tests

* Eventually as questions are answered through preliminary tests, you and your team will define baselines, benchmarks and ranges or boundaries for these expectations. This is where you can select which tests should be automated so they can quickly be executed against new iterations of models.
* Tests have to account for the lack of determinism in predictions – instead of writing an assertion that a prediction will be 0.45718038 which will fail too often, we can assert that a value stays or falls between a certain range.

How do we test this from a performance testing perspective

How do we generate data for predictive models

How do we train the model

How do we reset the model

What considerations are in place for testing these sorts of solutons

What monitoring solutions exist

It is a well-accepted practise to monitor software systems so that we can understand performance characteristics and react quickly to system failures. When deploying machine learning models, we still have the same set of concerns discussed above.

We’d also like to have confidence that our model is making *useful* predictions in production. A model could make a “successful” prediction from a service perspective but the predictions will likely not be useful. Monitoring our machine learning models can help us detect such scenarios and intervene.

Three classes of metrics we want to collect

* Model metrics – prediction distributions, feature distributions, evaluation distributions.
* System metrics – request throuput, error rate, request latencies, requestand response body size
* Resource metrics – CPU utilization, memory utilization, network data transfer, disk I/O.

Tools that exist

* LoudML – a Grafana python application that provides functionality for building, training and monitoring models.
* Grafana + Prometheus was used in the ML model made for this project.