Accelerate Functional Coverage Closure Using Machine-Learning-Based Test Selection

Jakub Pluciński, Nokia, Kraków, Poland (jakub.plucinski@nokia.com)

Łukasz Bielecki, Nokia, Kraków, Poland (lukasz.bielecki@nokia.com)

Robert Synoczek, Nokia, Kraków, Poland (robert.synoczek@nokia.com)

Emelie Andersson, MathWorks, Gothenburg, Sweden (eanderss@mathworks.com)

Antti Löytynoja, MathWorks, Espoo, Finland (aloytyno@mathworks.com)

Cristian Macario, MathWorks, Munich, Germany (cmacario@mathworks.com)

***Abstract*— As designs become more complex, constrained-random verification has become insufficient to validate the design quickly and efficiently. This technique has serious limitations, and it has been proven it may be difficult to achieve functional coverage closure with complex designs in a reasonable amount of time. The lack of appropriate techniques to cope with complex modern designs turned researchers' attention to machine learning-based solutions. These solutions excel at finding data patterns and effectively automating the test generation process, making it more effective and less expensive. This paper explores using neural network-based machine learning for functional coverage and, more explicitly, how autoencoders can help select appropriate stimuli to reduce the number of simulations needed. The approach has been evaluated on a channel estimation block, which is part of a 5G radio receiver. The results show it achieves up to 2x speedup in coverage closure compared to the traditional constrained-random test selection. Additionally, the approach is shown to improve overall verification quality by reducing the number of redundant test cases.**

Keywords—Constrained Random Verification; autoencoder; Machine Learning, AI; coverage-based verification

I. INTRODUCTION

One of the major challenges of functional verification is producing the most stressful tests with the greatest stimuli variety. While randomization can aid in this challenge, the transactions comprising a test cannot be entirely random, as certain combinations might be illegal. Verification engineers widely use constrained random testing techniques to solve this problem. This involves creating a set of constraints that define design input space and generate random test cases satisfying them, which are then applied to validate given properties in functional simulations. Engineers use the functional coverage metric to measure and understand how much of the functional design has been exercised, providing a quantitative measure of the completeness of the verification process.

The concept of constrained random testing in FPGA and IC design has its roots in the broader field of software engineering constrained random testing. While specific methodology origins are difficult to trace, it is believed that this technique was developed in the 1990s as a response to increasing software system complexity and the need for more efficient and effective testing methods. This methodology was then adopted in the field of FPGA and IC verification. There has been an exponential increase in the total number of transistors in a chip due to Moore's law over the years. This translates to more functionality in the same die area and increased design complexity. These trends became a serious challenge as the time to verify the design started to increase drastically, making it difficult to ensure the design was bug-free. Constrained random verification may not be effective at identifying certain bugs in these conditions, such as those requiring a specific sequence of inputs to trigger. In fact, recent studies have shown that constrained random verification leads to some coverage points being hit extremely frequently and others being hit very rarely [1].

The time-to-market pressure for the FPGA- & ASIC-based products is high, meaning verification must be completed quickly and efficiently. This poses the question of whether constrained random verification in its current shape really applies to modern design verification. We present a state-of-the-art, joint co-simulation and machine learning enhancement to the standard constrained random verification approach in this paper that accelerates the closure of functional coverage. This approach has been developed based on the hypothesis that dissimilar tests tend to hit dissimilar functional coverage events [2]. In the following sections, we will go through various approaches we have exercised and describe their application and results.

II. DUT and Constrained Random Verification Environment

The machine learning model was connected to a verification environment prepared for the Physical Uplink Shared Channel (PUSCH) IP estimation block as part of a 5G radio receiver. Demodulation Reference Signal (DMRS) processing is the main part of its operation. The base station generates a sequence sent to the User Equipment (UE). UE sends it back, so the base station can estimate the composite propagation channel by comparing received and transmitted DMRS [3]. Estimated channel parameters are needed for the correct processing of the transmitted data.

In this project, we used a UVM-based testbench utilizing a 5G processing simulator written in MATLAB® as a reference model. The simulator is connected to the testbench via the C/C++ interface layer, taking advantage of the SystemVerilog DPI-C on one side and MATLAB Data API for C++ on the other. The testbench is responsible for creating random test configurations and scenarios. It sends them to the model, which produces stimulus and reference data based on received parameters. The data then goes back to the SystemVerilog part, where drivers use the stimulus to feed the DUT and reference data. The data is compared in scoreboards with the information gathered by multiple monitors.

Many parameters can be used to generate the test cases, and most interact with each other. This is why simple randomization is not enough, as it would result in an extremely low probability of generating a parameter set that will meet all the requirements the simulator sets. Several dozens of constraints had to be implemented to ensure that randomization gives reliable results. Three major factors were considered while preparing the constraints—the DUT capabilities, requirements of the reference model, and the 5G standard.

III. AI Technologies

Many terms are circulating in Artificial Intelligence (AI), leading to misunderstandings and incorrect conclusions. In this chapter, we will introduce AI on a high level to avoid this and describe the different AI methods this study explores. The autoencoder is covered in more detail at the end of this chapter as the method selected for implementation.

Artificial intelligence is an umbrella term referring to any technique enabling machines to mimic human intelligence. **Machine Learning** (ML) is a sub-category of AI that refers to statistical methods allowing machines to learn (or mimic) behaviors and tasks directly from data without their explicit programming. Several ML algorithm families have been developed over the decades, such as Support Vector Machines (SVM) [4], Neural Networks [5], Bayesian Learning [6], Fuzzy Logic [7], Ensemble Learning [8], etc. “AI” and “ML” are often used interchangeably as terms, even though AI also includes techniques not based on ML.

Deep Learning (DL) is in turn a sub-category of machine learning. It refers to neural networks, typically consisting of multiple or even hundreds of hidden layers, resulting in a “deep” network architecture. The autoencoder implemented in this study is an example of a deep learning model. The relationship between AI, ML, and DL is illustrated in Figure 1.

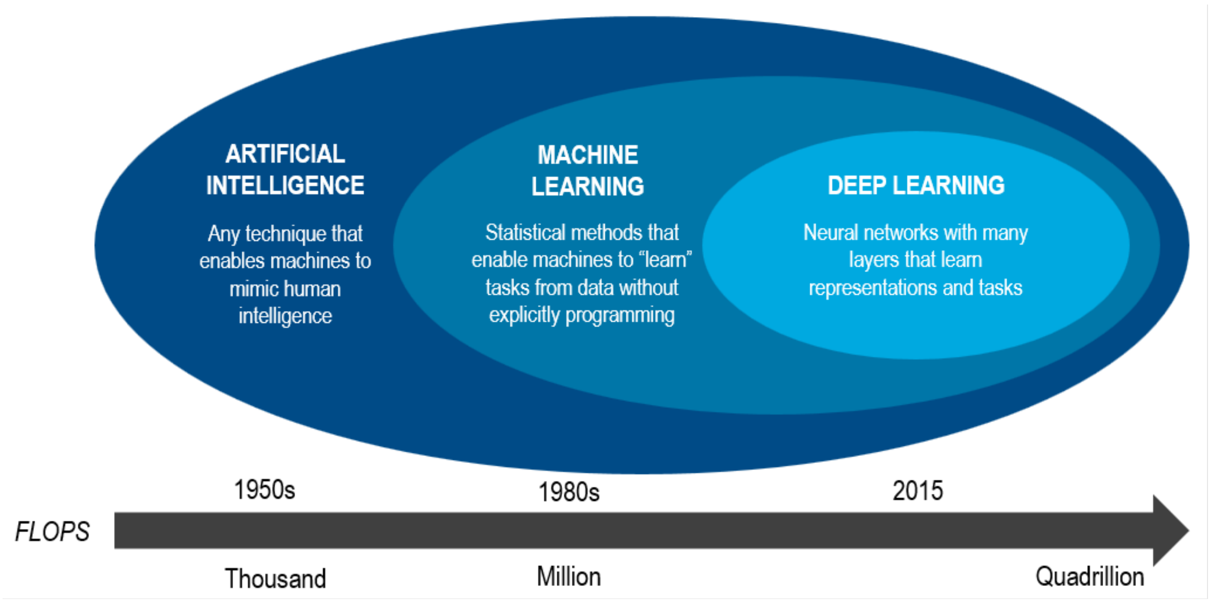
ARTIFICIAL INTELLIGENCE
Any technique that enables machines to mimic human intelligence
MACHINE LEARNING
Statistical methods that enable machines to "learn" tasks from data without explicitly programming
DEEP LEARNING
Neural networks with many layers that learn representations and tasks
1950s
1980s
2015
FLOPS
Thousand
Million
Quadrillion

Figure 1. AI Overview

Furthermore, ML consists of two main branches: **supervised learning** and **unsupervised learning**. Supervised learning is the most popular branch and refers to model learning input to output mapping. Once the ML model has learned the mapping from a potentially large number of examples of input and output data, the model can be used to *predict* the output, given a new observation of input data. The model learns a *classification* task if the output is a discrete value or a category. A trained model can then predict a new input data class. The model learns a *regression* task if the output is a continuous value. For example, we can train a regression model to predict electricity demand based on input data such as weather forecasts, time of the year and day, past demand, etc. The major challenge with supervised learning is that it can be very difficult to obtain enough input-output data to train accurate models depending on the task. There is usually plenty of input data, but the output data associated with each sample of input data often needs to be provided by humans. Therefore, it can be very time-consuming to build such a dataset.

In contrast, unsupervised learning is a technique used with data that does not include associated outputs. We can find patterns, structures, and relationships in the input data using unsupervised learning without prior knowledge about data groupings and relationships. Typical use-cases of unsupervised learning are clustering, anomaly detection, and dimensionality reduction. However, we do not necessarily know what these clusters or relationships mean since unsupervised learning does not use labeled output data, such as knowledge of the associated classes of the data points.

The assumption in this study was using ML could reduce the number of simulations required to reach a certain functional coverage by identifying the relevant test stimuli that would increase the coverage. Supervised ML techniques were explored first. The goal was to develop a classifier model that could label the test inputs into two categories: inputs that would likely increase the functional coverage and inputs that would not. As supervised techniques eventually did not work, the problem was reframed as an unsupervised machine learning problem. A special type of neural network—*Autoencoder* [9] —was found to provide promising results.

An autoencoder learns compressed or latent data input representations, as illustrated in Figure 2. It can be used for data dimensionality reduction, data denoising, feature extraction, and anomaly detection. An autoencoder model consists of two parts: an encoder and a decoder. The encoder compresses the input data to a latent representation, and the decoder reconstructs the input data from the latent data. The model learns how to reconstruct the input at its output during the training. The model can be used to reconstruct new input data after training. If the input data and the data used in training the model are similar, it can reconstruct the input data accurately with a small reconstruction error. However, the model does not know how to reconstruct the input if the input data is dissimilar to the input data used during training, resulting in a measurable difference between the input and the output.

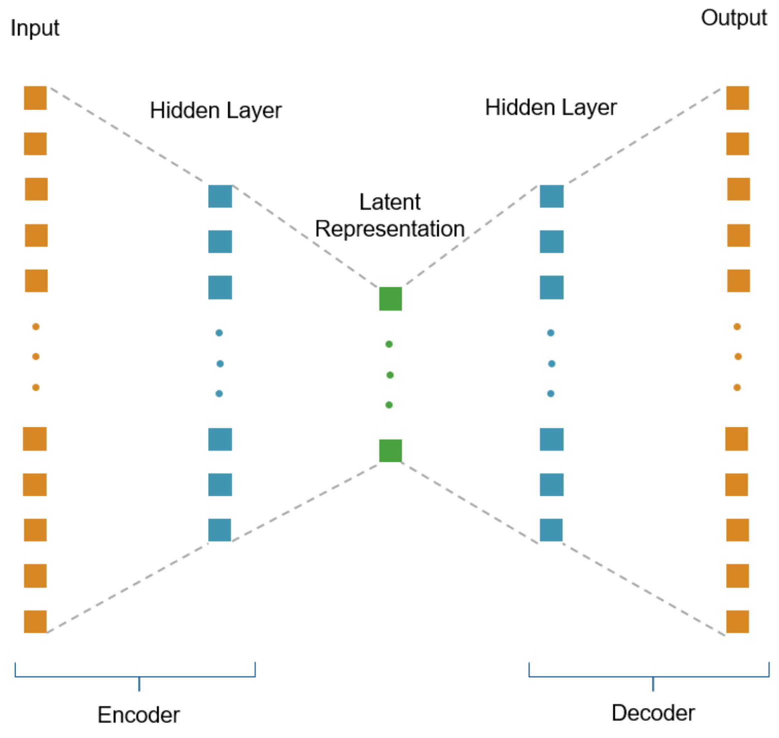

Figure 2. Structure of an autoencoder.

By examining this autoencoder reconstruction error, we can determine if the input data is anomalous or somehow different from the training data. The autoencoder was used to determine how dissimilar the newly generated test vectors are from what the design has already been exposed to in this study.

IV. Applying Machine Learning methods to Constrained Random Verification testing

This chapter covers the presented solution as well as some previous failed experiments to avoid publication bias in accordance with [10]. The goal was to develop an ML model to predict whether a given test would increase the functional coverage of the DUT. Any tests that did not meet these criteria would be considered redundant and discarded, allowing for new test generation. Tests that could increase the functional coverage would be kept and simulated. Both supervised and unsupervised methods of ML were tested.

A. *Supervised Approaches*

A small 1800-sample dataset was created for the supervised learning methods. Each sample consisted of a set of parameters: the test generation input and later model input. Every set of parameters was later used to generate the appropriate test that was then simulated with simultaneous functional coverage. A human labeled this set based on a non-sequential assessment. Each test was labeled as "interesting" if it increased the functional coverage or "non-interesting" if it didn't. A wide range of supervised learning methods were evaluated, notably:

- Support Vector Machine
- Decision Trees [11]
- Random Forest [12]
- Simple Neural Networks [5]

These methods did not effectively classify any test series. The poor training results stem from technically training the network on a single series of randomly assorted tests.

A sequence classification approach was taken next. A script for randomization and labeling was created. It randomized the sequence order of 1800 tests from the previous dataset, re-labeled them based on which test covered a new bin, and saved the test series as a CSV file. A new 2000 test series dataset was created based on this script, and the use of Long short-term memory (LSTM) [13] networks were tested. This approach proved unfruitful, as all different network configurations have learned to label all tests after a set number of them as “non-interesting” resembling a step function. All tests before the cutoff point were seen as increasing the coverage (i.e., “interesting”), and the rest were classified as “non-interesting.” This resulted in no functional coverage increase after reaching the cut-off point. This is a logical generalized conclusion, as the beginning of each test series contains most of the coverage-raising tests. The number of tests that increase the functional coverage gets exponentially smaller with each one simulated. Therefore, the LSTM neural network learns a cutoff point.

B. *Unsupervised Approaches*

Further research determined that the problem of finding interesting tests within a test set based on their simulation input parameters was an unsupervised anomaly detection problem. An unsupervised approach was also seen as beneficial due to the difficulty of creating and labeling the dataset. Omitting the data collection and dataset creation step would also allow this method to be easily scalable with growing DUT sizes and ever-changing parameter sets.

A combination of dimension reduction and clustering was used at the beginning. The test parameter space dimensions were lowered to 2 or 3 features, using Factorial Analysis of Mixed Data (FAMD) [14], t-distributed Stochastic Neighbor Embedding (t-SNE) [15], or Uniform Manifold Approximation and Projection (UMAP) [16]. All these methods have proven to give highly overlapping clusters for the previously labeled data and thus were seen as unfit for this solution.

Based on [2] and previous experience with sequential data classification, a batch-based autoencoder anomaly detection approach was finally tested. The parameter randomization and test generation flows were divided into batches of a few dozen tests. The first batch was used in its entirety as the dataset base and was used to train the autoencoder. The next batches were processed using the following algorithm:

1. Process the batch of data using the autoencoder.
2. Calculate the quality of decoding (difference between input and output) using a chosen metric.
3. Sort the tests according to the decoding quality, where the lowest quality test comes first.
4. Cut off a given percentage of tests with the highest decoding quality.
5. Add the remaining tests to the dataset.
6. Fine-tune the autoencoder using transfer learning and the new dataset.

This process allowed the autoencoder to approximate the functional coverage space and detect the tests that appeared less in the random generation. The autoencoder architecture is shown in Table 1. In most cases, it is assumed that parameters and their combinations not seen previously should cover a new bin in the functional coverage. Transfer learning on the aggregated test dataset also allowed to strengthen the model, as the knowledge of tests that it saw in the beginning was reinforced by repeated exposure. The cutoff threshold was chosen based on two methods: Fixed and Moving Mean Square Error (MMSE).

The Fixed method of cutoff was based on a simple percentage. For example, only 50% of the tests from the new batch with the highest MSE would be saved and simulated if a 50% threshold was chosen. The MMSE method was based on the average mean square error from training. Any tests with an MSE lower than the average mean square error from the previous transfer learning step would be discarded.

Table 1. Autoencoder layers

Type	Activation sizes [Channel x Batch]	Learnable Parameters
Feature input	Number of Features x N	-
Fully Connected	Number of Features / 2 x N	Weights + Bias
Leaky ReLU	Number of Features / 2 x N	-
Fully Connected	Number of Features / 4 x N	Weights + Bias
Leaky ReLU	Number of Features / 4 x N	-
Fully Connected	Number of Features / 2 x N	Weights + Bias
Leaky ReLU	Number of Features / 2 x N	-
Fully Connected	Number of Features x N	Weights + Bias
Regression Output	Number of Features x N	-

V. INTEGRATING MACHINE LEARNING WITHIN THE VERIFICATION FLOW

We created a script for running the whole verification flow in the proof-of-concept project, including the ML model. We used ExecMan for test simulation and functional coverage reporting and MATLAB for data preprocessing, model training, and prediction. A separate Python script handled data flow and program execution. The flow was divided into the following steps:

1. Take the target functional coverage score and a maximum number of regression iterations as user input (Python).
2. Make testbench generate constrained random sets of test parameters (Synopsys ExecMan).
3. Provide generated dataset to the ML model (MATLAB).
4. Perform RTL simulation using tests based on parameter sets recommended by the ML model (MATLAB, Synopsys ExecMan).
5. Merge functional coverage results (Synopsys ExecMan).
6. Go back to Step 2 unless the target functional coverage score or maximum number of regression iterations was reached.

VI. RESULTS AND FUTURE ENHANCEMENTS

The system was tested on different batch sizes and threshold types. Four batch sizes were used: 25, 50, 75, and 100. Both proposed thresholding methods were tested. The fixed-level thresholding was set at 25% and 50%. Each configuration was run five times. The coverage goal was kept at 67%, as it was the maximum possible for the current test randomization constraint configuration. The initial autoencoder training was done on 300 epochs, and transfer learning was done on 700 epochs in further iterations. The MSE loss function was used during training. Each approach was measured on total regression time, number of simulated tests, and number of prediction iterations. Figures 3, 4, and 5 present the averaged results for each batch size and thresholding type.

As expected, each thresholding type for each batch size has lowered the total number of simulated tests compared to the baseline. The worst-performing thresholding type was the fixed 50% threshold. The fixed 25% threshold gave the best results for lower batch sizes, lowering the total number of tests simulated by a maximum of almost 43%. The MMSE method described in Section IV yielded better results with bigger batch sizes, outperforming the baseline by a maximum of 37% and the fixed 25% threshold by a maximum 14%. A lower number of tests to simulate lessens the simulation load, allowing for lower CPU allocation and lower power consumption.

The number of iterations compared to the baseline is comparable. The fixed 25% threshold uses more iterations in smaller batch sizes to achieve the same coverage goal, as it is more prone to missing interesting tests at the regression's beginning.

The average total regression time was not improved due to the parallel nature of running tests and the overhead caused by the repeated simulation environment starting and closing. The regression time was doubled in the worst case. This could be improved by running the simulation environment in the background and generating the test batches as requested by the simulation server. This could greatly improve the regression time.

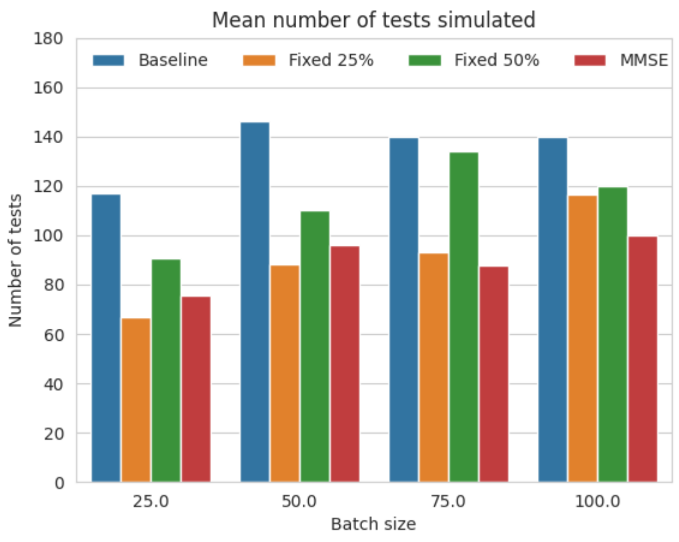

Figure 3. Mean number of tests simulated based on threshold type and batch size.

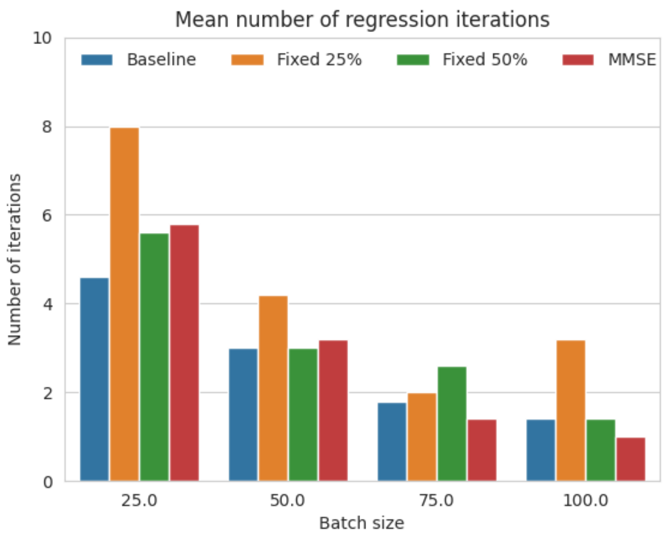

Figure 4. Mean number of regression iterations based on threshold type and batch size.

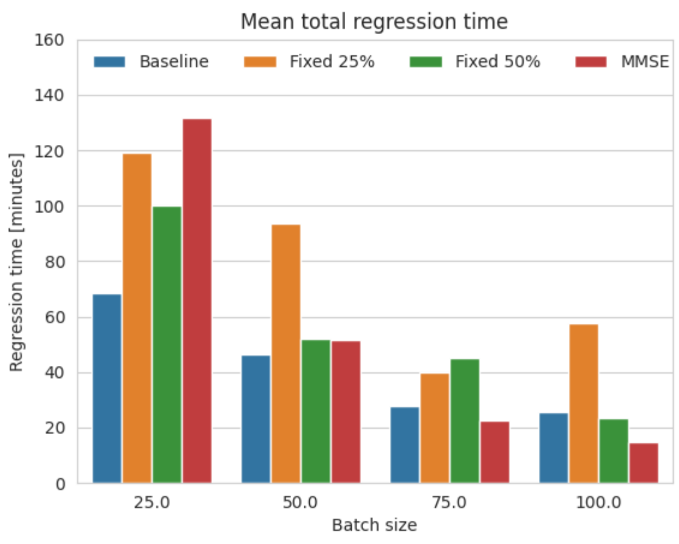

Figure 5. Mean total regression time based on threshold type and batch size.

Other thresholding techniques and further hyperparameter optimizations could improve regression performance, lowering the number of tests simulated even lower. A fixed scheme for lowering or a moving average threshold scheme could be employed in further research.

VII. Conclusion

The solution presented is the first state-of-the-art, neural network, and co-simulation-based functional coverage closure acceleration approach successfully used on a commercial IP. It allows for a generic, unsupervised novelty test selection system to be created. It can be adapted for use in any hardware and software verification and testing environments that employ a random test generation scheme.

References

[1] S. Sokorac, "Optimizing Random Test Constraints Using Machine Learning Algorithms", DVCon U.S. 2017, February 27-March 2, 2017

[2] Xuan Zheng, Kerstin Eder, Tim Blackmore, "Using Neural Networks for Novelty-based Test Selection to Accelerate Functional Coverage Closure", ACM Conference'17, July 25-27, 2017

[3] Johnson, C. 5G New Radio in Bullets. Farnham, England, 2019

[4] Cortes, Corinna, and Vladimir Vapnik. "Support-vector networks." Machine learning 20 (1995): 273-297.

[5] Schmidhuber, Jürgen. "Deep learning in neural networks: An overview." Neural networks 61 (2015): 85-117.

[6] Bishop, M. Christopher. "Pattern Recognition and Machine Learning." Springer (2006).

[7] Zadeh, Lotfi Asker. "Fuzzy sets as a basis for a theory of possibility." Fuzzy sets and systems 1.1 (1978): 3-28.

[8] Hansen, Lars Kai, and Peter Salamon. "Neural network ensembles." IEEE transactions on pattern analysis and machine intelligence 12.10 (1990): 993-1001.

[9] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning internal representations by error propagation." (1985).

[10] Kay, Dickersin, "The existence of publication bias and risk factors for its occurrence", Jama 263.10 (1990): 1385-1389, March 9, 1990

[11] Quinlan, J. Ross. "Induction of decision trees." Machine learning 1 (1986): 81-106.

[12] Breiman, Leo. "Random forests." Machine learning 45 (2001): 5-32.

[13] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

[14] Davidow, Matthew, and David S. Matteson. "Factor analysis of mixed data for anomaly detection." Statistical Analysis and Data Mining: The ASA Data Science Journal 15.4 (2022): 480-493.

[15] Hinton, Geoffrey E., and Sam Roweis. "Stochastic neighbor embedding." Advances in neural information processing systems 15 (2002).

[16] McInnes, Leland, John Healy, and James Melville. "Umap: Uniform manifold approximation and projection for dimension reduction." arXiv preprint arXiv:1802.03426 (2018).