Machine Learning in Network Analysis and Software Testing

Teemu Kanstrén
VTT
Oulu, Finland
firstname.lastname@vtt.fi

***Abstract*—This is a review of some recent works in applying machine learning in network analysis and software testing. Originally done to support work in building more advanced use cases for domain experts and partners to give some ideas on how ML might be applied in different contexts. Not to copy from, but the given an idea of the potential, properties of application domains, data, and other related aspects when looking at what benefits ML could bring.**

***Index Terms*—network, software, testing, qos, qoe, machine learning**

I. Introduction

This is a brief review of paper on applications of machine learning (ML) in network analysis and software testing. Network analysis is reviewed to provide a basis for applications in test scenarios in test environments such as the Finnish 5G test network (5GTN). Software testing is reviewed to provide a second domain with example applications, to give a broader idea of application of ML across different domains. The connection between testing networked systems, network elements, and software is also quite close, and as such the review can support people working in both areas to find synergies. For example, many use cases run on top of a test network such as 5GTN are also likely to require similar tuning of the algorithms and approaches to their domain as in software testing.

The reviewed works are described in terms of their application domains, analysis goals, applied techniques, and the feature sets (ML training data) applied.

The rest of the paper is structured as following. Section II gives a brief overview of machine learning concepts discussed in this paper. Section III reviews applications of ML in network analysis and testing. Section IV reviews applications of ML in software testing. Sections V and VI discuss these applications overall, and conclude the paper respectively.

II. Background Concepts

Machine learning is defined here as the ability of a computer system to automatically learn (complex) patterns from data. In networking, this data can be, for example, data packets, quality of service (QoS) statistics (e.g., throughput, jitter, delay), or flow related metrics (such as connection times). In telecommunication systems this can further include domain specific extensions such as call detail records (CDR) or control signaling (call setups, handovers, etc.). In networked systems, service specific data can be included, such as node resource use, transaction times, and application specific metrics (e.g., video streams, user types and counts, messages sent).

For example, in a test network such as 5GTN, we can measure detailed and accurate QoS measurements between different end points, or for specific users, using specific tools. In operational networks such specific measurement deployments are not possible. A potential application of ML could be to evaluate the possibility to estimate such QoS values from generally available network parameters (e.g., base stations and core network), enabling such estimates in operational networks and using those as input for network management.

A related term is artificial intelligence (AI), which here is defined as using the ML results and other available information as a basis for applications and actions. In other cases, the ML results can be used for understanding the system, such as which features in the data best describe the differences observed in the network/system during different test scenarios. In this paper these applications of the ML algorithms are briefly discussed with their description.

There are many different machine learning techniques, and in this paper we will not go in depth into their definitions. The techniques applied in the works discussed in this paper are mentioned as reference for the interested reader. These include traditional machine learning methods such as Linear Regression (LNR), Logistic Regression (LGR), Naive Bayes (NB), Decision Trees (DT), Random Forests (RF), and Support Vector Machines (SVM), as well as neural networks (NN) and their deep learning (DL) variants. Besides these, many variants of ML algorithms exist. For example, [2] studies 179 classifiers from 17 different classifier families, and find, out of all these, specific variants of RF, SVM, DT, and NN consistently perform best across their datasets. Thus, this list is at least a good starting point.

Beyond classifiers, other ML techniques also exist. Natural language processing (NLP) is a set of techniques that can be used to process a set of natural language documents as input for ML algorithms. The NLP techniques cited in this paper are term frequency with inverse document frequency (TF-IDF) and latent dirichlet allocation (LDA). TF-IDF aims to highlight words that are especially relevant for specific documents. Term frequency (TF) in a document is used to highlight terms most relevant for one document, and inverse document frequency (IDF) is used to down-weight those terms

that appear in many documents. The end result is intended to highlight terms (words) that are important (common) in a document, but are not generally common in all documents.

LDA uses a statistical process to group words in documents into specific topics. A "topic" is a set of words, and each word in each document is assigned to a topic. Typically there are multiple topics in each document, and each unique word can be assigned to different topics in different sentences. The topics do not necessarily have a meaningful human interpretation (although often one can be assigned by human), but can still be useful to perform automated actions such as finding documents covering similar topics.

In addition to the different ML algorithms mentioned, another concept mentioned in some of the reviewed works in ensemble learning. This uses several variants of ML algorithms trained to solve the same problem. The predictions from these different variants are then combined using techniques such as averaging or feeding the results of each separate algorithm as features to another ML classifier. The goal is to have a more accurate classification than provided by any single algorithm alone.

III. Network Analysis and Testing

Running test scenarios in test networks such as 5GTN requires building the required network architectures and configurations, running the test scenarios, collecting data about the overall networks and services that are part of the test scenarios, and analyzing the data. The goal of these data analytics is to understand the test scenarios, what is happening, what are the impacts on different elements, how do the different configurations effect the operations. What is analyzed, and how this is done, depends on the goal of the test scenario. In a test network, those scenarios are often related to different aspects of networks, which can also include aspects of applications and services running on top of the networks especially from different non-functional viewpoints, such as reliability, performance, and security. This section reviews applications of ML to networking related data analytics to support such test scenario analytics.

A. Examples of Use Cases

Example cases of big data, and its use, in telecommunications are presented in [3]. The datasets are classified under signaling data, traffic data, location data, radio waveform data, and heterogeneous data. Examples of signalling data analysis include identifying coverage holes by analyzing handovers. Examples of traffic data analysis include analyzing traffic volumes over times of the day to provide input for network management, failure detection, performance analysis, and security management. An example of location data is given in using CDR data to movements of groups of people during large events (sports fans, emergency situations, etc.). An example of analyzing radio waveform data is given in using it to estimate user movement speed. Heterogeneous data use refers to combining these datasets together and with other existing datasets, and finding overall patterns for goals such as anomaly detection for cyber security analysis.

Big data use cases for mobile operators are considered also in [5]. Many of these use cases go beyond what is related to communication networks themselves, such as providing route maps and plans, vehicle and person tracking, analysis of billing data, and optimizing service pricing based on sentiments.

The more network focused use cases in [5] include real-time failure root cause analysis, self-organizing network automation such as provisioning, configuration, and commissioning, input to intelligent marketing campaigns, fraud detection, and informing customer support about caller issues in real-time. Input data (features) in these cases include CDR records, more extensive data records (xDR), traffic data, network event data, location data, subscriber data, customer care data, and historical data.

B. QoS and QoE Prediction

A genetic programming (GP) based approach for time series prediction of QoS values is presented in [9]. Input data features used can be any selected QoS metrics (in this case they seem to use server response time), and a genetic programming is used to generate a function to predict near-future values for the same QoS metric. The N previous values of the chosen QoS metric(s) are used as the input for this prediction and for learning the function. In [9], N is basically 15, although can be smaller depending on what function the algorithm ends up with. The resulting function is then used to predict QoS values from previous measurements, and evaluated against past measurements during training (when generating the prediction function).

The generated function consists of a combination of multiple basic functions, such as: +, -, *, /, mod(), min(), max(), pow(), sin(), cos(), tan(), abs(), floor(), ceil(), round(), log(), and exponent(). Features in the formula are time series current value (Yt) and previous values from previous step (Yt-1) up to 15 steps in the past (Yt-15). These functions and features are combined using various genetic programming operators such as crossover and mutation. Their comparison against methods such as LNR and NN shows GP can provide good results in some cases as well.

Machine learning to identify network interference is described in [28]. Channel Quality Indicator (CQI) values for connected UE's is collected throughout the network. As CQI is based on a 3GPP standard, all UE's are expected to implement it similarly under similar conditions, making it a useful overall measure to monitor and analyze. By comparing the UE CQI values against their neighbouring cells, the aim is to optimize spectral efficiency and minimize interference. As a basic approach, correlations between neighbouring cells having high throughput values, at the same time as a cell has low CQI values reported for it, are used as a basis to identify optimization opportunities. A cell is considered to interfere with another if a high throughput for that cell correlates with a low CQI for the other cell.

Beyond this, when applying ML, a single end-user quality metric (CQI here) is taken as the target variable to predict. Similarly, a measure for interference of neighbouring cells is selected, such as physical resource block (PRB) utilization rate, or data volume. The input features for ML are these selected interference values for N neighbouring cells, and the target variable is the chosen quality metric (e.g., average CQI). An ensemble of several simple learners is used to produce the final estimate of predictor for the target variable, with the use of multiple learners targeting to better identify relations between multiple interacting cells. The resulting prediction value(s) from ML are used to give importance scores of how much interference each neighbouring cell is giving to a problematic cell.

Optimizations applied as a result of identifying interfering cells include antenna tilt changes and modifications to reference signal transmission power (boosting or de-boosting). As a benefit, this is also seen as a means to replace expensive and non-real time drive testing of networks, with more automated, constant and close to real-time views and optimizations of operational networks. The results in [28] indicate the resulting optimizations to increase throughput while data volume remains the same. Radio signal de-boosting was found to be a more efficient optimization technique in the cases investigated in [28]. They measure success using metrics such as radio resource control (RRC) and enhanced radio access bearer (ERAB) setup success and drop rates.

End user application QoE performance prediction is investigated in [1]. To get labeled training data for training ML classifiers, a set of users is surveyed performing given tasks. For example, tasks can be to watch a 2 minute high-definition (HD) video clip on YouTube, view Facebook streams, or browse cities in Google Maps. Each surveyed user had an app installed on their mobile device that they used after each task to report their subjective QoE experience on levels from 1 (bad) to 5 (excellent). These user reported values are then used as the ML training targets.

In addition to the manual reporting tool, an automated network traffic monitoring tool was applied, associating network flows to specific applications. The flow information includes start time, duration, direction (up/downstream), and throughput metrics (up/downstream). Additionally, connection metrics such as signal strength, operator id, cell id, radio access technology (LTE/3G/2G/...) were collected and used. The automated collection is done locally and periodically transferred to backend servers for post-processing. A session is defined as a set of joined (in time) flows for a specific application for a user. The user provided QoE evaluation is associated to these sessions, along with a set of KPI values calculated from the set of automatically collected network metrics. These KPI's simply summarize the session metrics into single values, typically statistics such as average, minimum and maximum (e.g., average throughput in session).

Machine learning classifiers are trained on these values, with the user given QoE estimates as target labels, and the metric KPI's as input values. The base algorithm used in decision trees, to allow easy interpretation of results with reasonably robust classifier. Additionally, RF, SVM, and NB are evaluated, showing DT scoring equally well to other classifiers on this dataset.

To select subsets of features, [1] uses correlation-based feature selection. Selected variables have highest correlation with the target variable, and lowest with each other. Finally, each feature is ranked by the decision tree weights, according to the positive/negative effect it has on the QoE for a specific app. A separate model is built for each app, where different metrics are found to be more important for different apps. A deeper investigation into specific metrics per app is also shown to provide further insights. For example, longer YouTube sessions were reported to have generally lower QoE, which was found to be due to stalling video causing delays.

Root causes for network performance variability are investigated in [7]. A mobile application on an Android device constantly monitors and records both passive and active measurements. Passive measures include local signal strength, network provider, used wireless access technology, and users current location. Active measurements include round-trip time (RTT), trace-route and throughput. RTT from the UE to the measurement servers is measured periodically, and a throughput measurement is executed when the user activates it. Data is locally collected and later transmitted to backend services for deeper analysis.

RTT is measured using the Android ping command, and throughput with the iPerf application. A server in the authors host site is used for ping backend target. Additional HTTP "ping" is measured as time-to-first-byte when performing a HTTP request for a subset of the top sites listed on Alexa website ranking list.

A number of different measurement campaigns are executed to collect a diverse dataset. A measurement tool is made available in the Google Play store for Android, and the data collected and provided by the users of this tool forms one subset. A group of five students is recruited as one subset to perform specific measurements during their daily activity. Finally, a stationary set of measurements was collected using a set of devices deployed in a specific location and kept there for the whole duration of the measurement campaign. This was aimed to minimize the change of parameters during measurement. The stationary dataset was also used as a reference to capture time-based variation not caused by other parameters. There were 3 stationary locations, each selected to give different profiles. Two in different residential areas, and on in an office building next to a central park. Time of experiment and measurement was 4 weeks.

Analysis of collected data includes isolating mobile network operator effect from the data by looking at the trace-route data, identifying users potentially assigned point of presence (PoP) in the network. Markov-chains and decision-tree based machine learning are used to predict PoP assignments, and PoP assignments are shown to contribute a high amount to user perceived QoS values. The Markov-chain is used to predict transitions between PoP states, and is hypothesized to provide

a basis to predict the assigned PoP when a device attaches to the network. The decision tree is used to predict a users PoP state at given time. Features used to train the DT here include the day of the week, the hour of day, and the user ID value. For example, HTTP "host" field, SSL client and server "hello" packets, UDP ports, and TCP flow sizes are used.

The DT evaluation here gives over 97% accuracy, but this is not explored in more depth in the paper. This seems high for simply time-based prediction, highlighting interesting results, while it seems a deeper investigation would be needed to more broadly adopt any results. For example, it may be that the observed people have very specific routines, leading to good prediction accuracy due to predictable movements during days. The paper concludes with analysis that PoP assignments, in this case, indicate random PoP assignment by the operator, showing large potential improvements.

C. Application Identification

Application flow identification is studied in [10]. Test automation tools such as GUITAR and Appium are used to drive application use scenarios on mobile devices. A deep packet inspection (DPI) tool is used to capture the network traffic during this time. The flows are first classified as HTTP, SSL, TCP or UDP flows. Based on this classification, specific protocol fields are extracted as features for the ML algorithms. To clean the dataset of unrelated flows (such as advertisement and tracking flows on websites), exclusion lists are used. These exclusion lists are built by monitoring all the captured sessions for shared flows, indicating generic flows to ignore (e.g., adversing or tracking).

Different fields such as HTTP and SSL header fields are collected as monitoring data. Patterns of shared byte-sequences are considered if no suitable fields are found (e.g., in lower-level raw TCP and UDP dumps). Statistical properties of the flows are also collected, such as sizes, average, standard deviations. These statistical properties are further clustered to provide more refined features. These features of application-internal flows are used to classify application specific flows further. An example of three different flows for the WhatsApp application is used. By clustering the application flows based on the WhatsApp collected data packet statistics, K-means clustering is used to identify control-flow and data-flow specific flows. A third category is also used as "irrelevant flows", which are a small subset of anomalies not fitting in the control- or data-flow categories.

The "machine learning" part in [10] is learning to separate different flows this way, and to identify the patterns related to different applications. These for the set of rules used to classify flows into specific applications.

Application identification from network data is also studied in [4]. Examples of potential application areas for this are given as malware detection, content caching, application specific QoS optimization, and traffic control. The approach in [4] uses two types of features for training. Statistical features are collected as "non-payload" metadata from network packets. This includes features such as source and destination IP addresses and ports, statistics (averages, standard deviations) of packet lengths and arrival times, SYN packet window sizes, and ratios between source and destination traffic sizes.

The second set of features are captured using DPI methods. These features are formed by analyzing 2-grams of bytes in packet data, where unique 2-grams form features (2-gram here refers to a sequence of two bytes). To limit the size of the feature set, the set of top 100 2-grams (top 100 most useful) are selected as these features. This top set refers to the 100 best features for use in the chosen ML algorithm. Unfortunately, [4] does not describe how they select the top 100 features, but rather refers to future work for more extensive study on feature reduction. The DPI packet capture is focused on start of the data-flow, as 5-20 first packets in each flow. This choice is based on most likely containing header data in the first packets, which will likely vary much less that the actual payload data itself across users (for same application).

The training dataset in [4] is collected from a set of modified (rooted) Android devices. These have been modified to add an application identifier to each SYN packet starting a flow. This provides a set of labeled training data for the machine learning classifier. The classifier used is a random forest. A learning period of 5 days is found to give 92% accuracy with DPI enabled. In their previous studies, the same authors achieved 80% accuracy without DPI.

Again, application identification from network flows is presented in [12]. They identify network flows based on a combination of source IP, destination IP, source port, and the protocol identifier. The ML input features used are packet and connection statistics, such as maximum and minimum packets lengths in connections and flows. The target prediction label is the recorded application during specific flows. A large feature set of such potential statistics is referred to but not described in detail. As labeled training data, they use both an existing application dataset from the internet, and an additional dataset generated by themselves by having users use specific applications while all traffic is recorded.

As the overall set of possible features listed in [12] is described as large, feature selection algorithms (from the Weka machine learning library) are used to trim it to smaller. These algorithms are correlation-based and chi-squared based selection. The machine learning algorithms applied are decision trees, random forest, k-nearest neighbours, and bayesian nets. For the two datasets used for the study, k-NN is found to perform better for one and RF for the other, with RF being more consistent across both datasets although K-NN also doing well on both.

A more general approach to traffic classification is presented in [11]. Instead of looking purely at application specific flow optimization, they look to identify groups (or categories) of applications, and to optimize flows in those groups. However, this is still based on first identifying a set of applications and putting these to a set of four categories: voice/video conferencing, interactive data, streaming, and bulk data transfer.

The approach presented in [11] first identifies a set of "elephant flows", which are simply flows that are taking 1-10%

of the total bandwidth at a network edge. Using DPI, the first 20 packets of these flows are collected for analysis. The set of features used for ML include packet length entropy, destination and source ports, average and median packet lengths, and packet counts and interactivity degree from each flow. An SVM classifier is trained based on the identified properties (features) of the specified flows.

The architecture applied in [11] is especially tailored for SDN environments. The trained algorithm is deployed on an SDN controller node to manage network traffic and categorize the ”elephant flows”. Categorization is based on identifying specific types of flows based on their shared statistical properties. The features are not described fully in [11], rather a subset of final selected features from the full set is shown, and only a name is given not a full description. Openflow data capture functionality is used throughout the network to capture flow data and DPI metrics. The overhead of this is distributed throughout the SDN elements the different flows traverse through.

D. Anomaly Detection

Call detail record (CDR) analysis using clustering to find anomalies is explored in [8]. Input features for the clustering are taken directly from the CDR values as the (caller region) square id, timestamp, inbound call duration, outbound call duration, inbound SMS count, and outbound SMS count. Both K-mean and hierarchical clustering are applied to collect a set of clusters from the data. The clusters with a small number of data points are classified as anomalous. These are then provided for further investigation. In [8] the examples of findings are identifying special events such as football matches, and special locations such as hospitals.

Another application area suggested in [8] is to cleanse data of anomalies, or the separate the data into anomalous and ”normal” data, and use the resulting dataset as a labeled training set for a supervised classifier. This could then be applied to find more generally new types of anomalies, or to identify specific types of anomalies in operational networks, and to use those for specific actions such as real-time network management algorithms.

Generally potential application domains for big data analysis for networks given in [8] are providing visibility into the network, automated self-coordination of network functions and entities, assessing long-term network dynamics, optimizing for faster and more proactive networks, optimizing for smarter and more proactive network caching, optimizing energy efficiency, and unified performance evaluation of networks.

IV. Software Testing

Software testing aims to verify functional (e.g., conformance to specification) and non-functional (e.g., performance) properties of software. In relation to network testing, similar functionality may be applied from the viewpoint of networked systems. More generally, looking at ML applications for software testing also gives a second application domain to see some examples of broader application of ML to different problems. This section reviews applications of ML to software testing related data analytics to support different aspects of testing processes.

A. Defect Prediction

One of the main applications of machine learning in software testing is for defect prediction. This is likely due to the obvious applicability of ML to defect prediction. Generally, defect information (bug reports) are collected for almost every system. This provides a natural set of labeled data to use as input for training a ML classifier. An overall review of ML applications to defect prediction is given in [13], showing a clear rise of interest over past years.

The set of most applied ML algorithms as listed in [13] for defect prediction is similar to those presented in Section II; DT, RF, NB, SVM, and NN. As input data for ML, the studies referenced in [13] focus on use of source-code related metrics. These metrics are the potential input features for the ML algorithm training for defect prediction, while the target labels are the parts and versions of the code where faults have been found. Different studies find different sets of these metrics useful for different projects. To select between these metrics, [13] lists several used feature selection methods, and cites the most commonly used feature selection method as co-relation based feature selection (CFS). This is described as eliminating ”noisy and redundant” features, and retaining the features that are highly correlated with the target attribute. Finally, the trained classifiers are used to predict the most likely parts of the code to contain faults, which are provided as input for quality assurance to potentially focus more testing on.

An example of applying neural networks for fault-proneness prediction is from [14]. To determine the NN parameters (layers, unit count, learning rate), a coarse sampling of the NN parameter values is performed, and each variant is evaluated on the training set. That is, a set of parameter configurations for the NN are tried with a small set of different parameters, and the set that performs best on the training set is selected as the parameters for the classifier.

For feature selection, [14] uses sensitivity analysis. Values of each feature are perturbed (modified) by a given amount, and the accuracy of the model is evaluated. If modifying a feature causes large changes in the model accuracy, the feature is considered important and retained. Features used in [14] include various source code metrics, such as lines of code, number of branches, and cyclomatic complexity.

To estimate overfitting, [14] uses mean squared error estimation of both the training and validation sets. While the training set error keeps reducing over all iterations, the validation set error starts to increase at some point. This is used as a stopping criterion to stop overfitting for the training set. At the same time, the graphs in [14] show how the overfitting starts to mainly occur after the biggest gains over iteration counts in the training set have been achieved (i.e., the following gain in accuracy is much smaller).

As a final evaluation, [14] compares the effectiveness of NN vs SVM on predicting the fault-proneness. The NN approach

is found to give an accuracy of about 73%, and SVM about 87%. This shows the need to finely tune NN based approaches, and the usefulness of "traditional" approaches such as SVM in many cases.

Another approach to predicting fault-proneness of software with NN is presented in [23]. This one uses convolutional neural networks (CNN), along with traditional source code metrics, to build the predictor. To turn the source code into suitable input features for the CNN, the tokens in the source code are analyzed to produce the initial set for input. The selected set of source code tokens includes method declarations and invocations, and control flow node types (e.g., if/while statements). These lists are turned into integers, which are fed to a word-embedding layer to produce vectors of word embeddings as input features for the CNN.

The overall NN for defect prediction uses the CNN layers as an initial set of layers to produce abstracted feature vectors from the textual source code. These are merged with the traditional source code features (metrics, similar to previously discussed source code features) for the final assessment using a dense layer as single valued output (probability of fault). In the input data, a source code element (file) is considered as buggy if there has been a bug reported against it. The training set is described as unbalanced in containing more "clean" than buggy files. Sampling from the clean set is used to achieve class balance. As CNN also requires fixed length feature vectors, and the source code files differ in size, the feature vectors for different files are given equal length by appending zeros as needed. The resulting classifiers are applied to several projects, with a result of about 60% accuracy. A comparison against using the source code metrics for LN and NN is also made, showing the ML algorithm version with CNN to do better in the cases presented.

Use of ensemble learners for defect prediction is investigated in [26]. Several different configurations of classifiers such as Naive Bayes, Decision Trees, and K-nearest Neighbours are used to form the ensemble, and a final classifier is used to combine the predictions of all these for the final prediction. Features for the algorithms again include the source code metrics.

To build a more optimal set of classifiers for the ensemble, the diversity of classifiers is measured, and the set of selected classifiers aims to maximize this diversity. The diversity in this case is measured as the fraction of cases where one of two classifiers makes the correct prediction while the other one does not (in relation to sum of all predictions). The evaluation in [26] compares diversity based ensembles against the best performing single classifiers, showing the ensembles to produce improvements in the evaluation metrics. However, in most cases they show increased number of true positives (defects predicted correctly), but also increased number of false positives (defects predicted where none exist). In this case, the authors argue it is better to find more places that correctly need more testing focus (faults found in the study), even while providing some places where it can be useful but not critical (e.g., no fault was found in the study). That is, ensembles seem useful but needing tuning and domain consideration with regards to all evaluation metrics.

B. Automated Debugging

To ease debugging of a set of failed test cases, [18] clusters similar failed test cases together. NLP based techniques are used to analyze and cluster the test cases by treating each test execution as a document. The words and sentences in these "documents" are formed by recording the executions of the test cases, and collecting the source code lines executed by each test case. These source code lines are the split using camel-case splitting to form the words. Clustering is based on semantic similarity of executed test cases. The semantic similarity is defined using cosine similarity measures based on TF-IDF scores of the test cases. The intent is to make debugging easier by grouping similar failing tests together, to enable the expert to analyze these together for potentially shared faults.

In a similar approach for bug localization (finding the cause/location of failure in code) using textual features, Latent Dirichlet Allocation (LDA) is used in [15]. Source code and bug reports are used as input data for the ML algorithms. The source code is split into sections, which are considered documents for the topics modelling algorithms, based on some criteria, such as package level or method level splitting. Source code is preprocessed for the LDA model as well as the document topic assignments by stemming, stop word removal, and camel-case splitting. The features used in [15] are thus the words parsed and processed from the source code. For LDA analysis, each source code method is considered a document.

A topic model is generated based on this generated document set, source code tokens/words are assigned to these topics, and each bug report is evaluated with regards to which documents would be most likely to generate this bug report. These parts of the source code are then ranked highest in the resulting report for debugging the bug reports. The results across several projects are evaluated, and for the available set of bug reports with matching fixes available, the relevant method where the bug is located, is in most cases returned in less than 1% of all methods. However, in practice this can be thousands of methods still for very large projects.

C. Testing ML Algorithms

Testing the implementations of a machine learning algorithms is described in [27]. A basic approach is to compare results from two different libraries implementing the same algorithm for the same dataset. A more detailed approach is described as based on metamorphic testing. A set of relations (called metamorphic relations according to metamorphic testing) are defined that should hold for different classifiers. Input and output data can be modified based on these metamorphic relations and associated functions. Metamorphic relations for specific algorithms can be excluded if not considered relevant for that algorithm. A list of generic relations is provide in [27], such as adding or removing training data elements. However, it remains unclear how well this approach could work due

to possible issues such as resulting in overfitting the data for duplicated elements, or other similar issues.

More recently, verification and validation of a machine learning framework application for image analysis is studied in [19]. The application focuses on classifying biological cells in medical images. A technique called correlation-based feature selection [20] is used for feature selection for the ML algorithm itself. This technique evaluates which features most correlate with the target variable, and least with each other. These features are then taken as the subset for the chosen machine-learning algorithm. To actually test the machine learning algorithms, an image based classifier is first trained using a training set as usual. A set of modifiers for the data is defined, and data that is classified in one way by the classifier is modified by these different classifier. The results of classifying the modified input data is then observed and used to evaluate the robustness of the classifier with known type of changes in the input data.

The description and evaluation of the approach in [19] is somewhat limited. The authors have access to an image generation tool that can be parameterized to generate images with desired features. However, if the "metamorphed", or generated images, end up producing better training for classifiers, and how well this would generalize to actual real-world data and images remains open. However, morphing input data to provide additional input data for ML is also more generally applied. For example, re-scaling, re-sizing, flipping, rotating, or adding noise, can be used as transformations to make an image recognition ML algorithm more accurate and robust [29]. When such transformation can be defined (and the more advanced they can be made), they can be potentially very useful for training and evaluating a classifier to be more robust.

The idea of using machine learning models as partial test oracles is briefly discussed in [17]. In [17], this is simply described as an idea for future research, but still provides an interesting potential application domain for ML in software testing. It describes a concept of machine learning models providing us with means of evaluating partial correctness of a system. Since inferring perfectly accurate specifications of a systems behaviour, or having those models fully accurately describe the systems expected behaviour is seen as unlikely, the idea in this case is to modify the expectations and assume that these might be describing the correctness only partially. The models would then be used as support for building more complete descriptions of the system behaviours.

Since these techniques are described as targeting automatically generated programs, they can be seen as another way of looking at testing machine learning algorithms (or applications build on top of them for specific purposes as AI based systems). As such, these questions can be related to the above approaches to testing ML algorithms and their applications, but asking the question of how to apply them to evaluate overall functionality vs producing training and evaluation data. To generate and train ML algorithms to reason about correctness in a specific domain.

D. Analyzing Test Artefacts

To help plan and schedule test execution effort, [16] uses a NN to predict the test execution effort based on a set of test and source code metrics. Source code metrics/features are similar to other ML techniques discussed above. The test metrics related features used include number of test cases and test steps, execution counts, and similar test metrics. The target is not to accurately estimate the effort needed, but rather to bias the model to overestimate it, as the authors see overestimation of effort as less of a problem than underestimation. This is a generally useful concept to keep in mind, how the training goal as evaluation metrics for ML algorithms may vary across different domains.

Automated linking of software artefacts using machine learning techniques is investigated in [21]. Recurrent neural networks (RNN) are used to give a probability of two artefacts being linked. The raw input to this system are the words in each artefact, which are first pre-processed using the usual NLP preprocessing methods (lower-case, removal of non-standard characters, stemming, stop-word removal). Each word is turned into its word2vec word-embedding vector to use as input for the neural network. These words are fed as input to the RNN, which provides as output a semantic vector for that input sequence. By running the RNN on both the source artefacts and the target artefacts two semantic vectors are acquired. The direction and distance of the vectors are then used as input to an integration layer in the neural network to assess the probability of a link existing.

To get the input features for this task, [21] train a word2vec model using both a generic set of words and sentences from Wikipedia, as well as a domain specific set acquired from their project partners. This is used to transform the input words into their word-embedding vector representations. From these, the domain specific model is found to be more effective than the combination of a general model (Wikipedia) and the domain model. A large set of predefined links between artefacts is used as a labeled training data set for the neural network. Since the dataset has large class imbalance (there are many more non-linked pairs than linked pairs), this is balanced by randomly sampling a subset of non-linked items to match training set label sizes. The results of the probability classification are presented to the human expert as input in the form of possible or most likely links. The human expert can then make the final decision, and their input can be used to produce further test and training data.

E. Application State Modelling

Using ML to automatically identify application state of web applications is explored in [22]. This is based on collecting the contents of a pre-specified set of the document object model (DOM) elements from a web-page. The DOM is a data-structure used to describe web-pages for the browser to render them, and thus contains information such as the page text, and page field identifiers. The words collected from the DOM are preprocessed using bag of words, TF-IDF, and latent semantic indexing. A training set is constructed by manually labeling

DOM elements with tags such as ”last name”, ”password”, and so on. To label new samples, the distance of the word-vector for each element is calculated, and the closest match is suggested as a label. For example, to identify login pages when crawling a web application. In practice, it seems one would do better to just define simple rules for what is the login page of ones application. However, for more complex crawling approaches, and model-based test-generation, some aspects of web-application models and mapping them together could benefit from using such ML approaches and their labels as one input.

F. Test Prioritization

Test case prioritization aims at reducing test cost by executing the tests that are most likely to find faults first. In [24], the likelihood of a test to find a failure is predicted using an SVM-based model. Features used are the number of lines of code executed by the test in changed files, the file paths of changed code vs tests, the textual content of change file vs tests, test failure history, and test age. The textual elements (path and contents) are transformed into TF-IDF vectors, and their cosine similarity is measured. Failure history is used to weight more recently failed tests higher, based on the likelihood of re-failure (regression). Test age gives a higher weight to newer tests, based on likelihood of recent changes introducing new faults.

To build a training set for the SVM classifier, every failed test set is added to the training set with these features. Snapshots of the tests and code where the test have failed and passed are added to the training set. The resulting classifier is used to give a probability of failure for test cases, and this probability is used to rank the tests. Unsurprisingly, the best single feature is found to be the test vs code path, as Java code is assigned to packages, and each package has a file path matching the package name. If the test and source code paths match, that usually means the test is exercising that code as well (code under package ”payments” and tests under package ”payments” are typically linked). The evaluation in [24] shows high gains for their dataset compared to previously applied approaches for that dataset. For example, to detect 75% of failures in that dataset, only 3% of the test set was needed, while the previous approaches using single metrics (e.g., code coverage, text matching, code path matching) required 44%.

An analysis of test and fault data, what properties in the given dataset influence defects, and how they might be used to help developers and testers focus testing across large sets of projects at Google is presented in [30]. This analysis starts by looking at the elements in the datasets, and their most likely interesting relations. In this case this focuses on the changes made, and how these propagate in the system through known dependencies.

The properties identified are discussed as potential features for future applications of ML to make the defect prediction more accurate [30]. These properties include how much overlap is between changed source code and package units, how often each individual or automated tool causes test failures, how far in dependency graphs failures typically are found regarding the changes, and how often specific file types or programming languages produce failures. These are used to provide basic guidance where and for whom more testing and review effort is suggested, and as feature for future application of machine learning to produce automated classifiers for more effective defect prediction.

G. Performance Testing

Feedback driven learning with software testing producing the input is described in [25]. As test scripts are executed, traces of program execution paths are collected. These traces are ranked (clustered) according to their execution time, in relation to the average execution time of all traces. Since the goal is to find performance bottlenecks, traces with higher than average execution times are labeled as good, and lower than average as bad. A rule-based learner is applied on the resulting set of bad traces and associated test inputs, aiming to identify test inputs that cause low performance. Method sets in clusters of bad and good traces are used to provide candidate methods to check for performance issues. Methods in good sets may be removed from the bad set. This approach in general seems applicable also for network testing, as well as several types of resources besides execution time.

V. Discussion

While network analysis and software testing are different domains, we look at some potential links between the two in this section.

From network analysis viewpoint, some of the software performance testing techniques can also be applied. Often the goal is to analyze the performance of networks in different configurations, with different loads, applications, or traffic types. In such cases, data needs to be collected for these different configurations to analyse the network performance. Techniques similar to those used in software performance testing could be applied to guide the generation of traffic in networks, with the help of ML techniques or otherwise.

As listed previously, collecting issue report and mapping them to part of the source code is a common application are for ML in software testing. For the network analysis viewpoint, similar approaches could be applied by collecting issues in networks, and using these as input for the ML algorithms to better identify when such issues might occur and what causes them.

Manipulation and selection of features is a common trend across ML application in software testing, network analysis, and many other domains. Image manipulations as described in [19] and [29] are an example of transforming input data for more robust classifiers. Many testing related ML algorithms preprocess basic metrics and features to provide higher level features (e.g., [21], [23]). These are typically various NLP transformations, as much of testing related data is in natural language form. Question is, if similarly shared higher-level transformations for network-related data could be identified.

Much as also discussed for defect prediction and test prioritization, basic data analytics of the network scenarios

could be performed as a basis for building the actual use cases, features, and contexts for more advanced ML applications in networks. The example given for software testing was to take the passing and failed test cases, and other data available about the testing process, analyze this, and to build feature sets to study the hypothesis for which tests should be prioritized first. Here the high-level use case is known to start with, and this is followed by setting a set of smaller sub-hypothesis, analyzing these, and using the information learned from those hypothesis to build more practically useful and advanced, supporting, ML approaches. Similar approach for network analysis could be done by defining the higher level goal, building sub-hypothesis, reviewing available data, and building ML algorithms and feature sets from the learned knowledge as required for the high-level goal. This could be useful if integrated as part of a test methodology or process as part of test services in test networks such as 5GTN.

VI. Conclusion

This is a review of small subset of work from recent years, and from the viewpoint of potential applications in 5GTN or software testing. Broader review could be done as needed, and as any new ideas of useful applications surface in different use cases and test scenarios.

References

[1] P. Casas, A. D'Alconzo, F. Wamser, M. Seufert, B. Gardlo, A. Schwind, P. Tran-Gia, R. Schatz, "Predicting QoE in Cellular Networks using Machine Learning and In-Smartphone Measurements", International Conference on Quality of Multimedia Experience (QoMEX), 2017.

[2] M. Fernández-Delgado, E. Cenadas, S. Barro, D. Amorim, "Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?", Journal of Machine Learning Research, vol. 15, 2014.

[3] Y. He, F. R. Yu, N. Zhao, H. Yin, H. Yao, R. C. Qiu, "Big Data Analytics in Mobile Cellular Networks", IEEE Access, vol. 4, no. 99, 2016.

[4] T. Iwai, A. Nakao, "Adaptive Mobile Application Identification Through In-Network Machine Learning", Asia-Pacific Network Operations and Management Symposium (APNOMS), 2016.

[5] R. I. Jony, A. Habib, N. Mohammed, R. I. Rony, "Big Data Use Case Domains for Telecom Operators", IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity), 2015.

[6] A. B. Letaifa, "Adaptive QoE monitoring architecture in SDN networks: Video streaming services case", International Wireless Communications and Mobile Computing Conference (IWCMC), 2017.

[7] F. Kaup, F. Michelinakis, N. Bui, J. Widmer, K. Wac, D. Hasheer, "Assessing the Implications of Cellular Network Performance on Mobile Content Access", IEEE Transactions on Network and Service Management, vol. 13, no. 2, 2016.

[8] M. S. parwez, D. B. Rawat, M. Garuba, "Big Data Analytics for User Activity Analysis and User Anomaly Detection in Mobile Wireless Network", IEEE Transactions on Industrial Informatics, vol. 13, no. 4, 2017.

[9] Y. Syu, Y-Y. Fanjiang, J-Y. Kuo, J-L. Su, "Quality of Service Time-series Forecasting for Web Services: A Machine Learning, Genetic Programming-Based Approach", Annual Conference on Information Science and Systems (CISS), 2016.

[10] U. Trivedi, M. Patel, "A Fully Automated Deep Packet Inspection Verification System with Machine Learning", IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS), 2016.

[11] P. Wang, S-C. Lin, M. Luo, "A framework for QoS-aware Traffic Classification Using Semi-Supervised Machine Learning in SDNs", IEEE International Conference on Services Computing (SCC), 2016.

[12] B. Yamansavascilar, M. A. Guvensan, A. G. Yavuz, M. e. Karsligil, "Application Identification via Network Traffic Classification", Workshop on Comuping, Network and Communications (CNC), 2017.

[13] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction", Applied Soft Computing Journal, vol. 27, 2015.

[14] I. Gondra, "Applying machine learning to software fault-proneness prediction", Journal of Systems and Software, vol. 81, no. 2, 2008.

[15] S. K. Lukins, N. A. Kraft, L. H. Etzkorn, "Bug localization using latent Dirichlet allocation", Information and Software Technology, vol. 52, 2010.

[16] D. G. Silva, M. Jino, B. T. de Abreu, "Machine Learning Methods and Asymmetric Cost Function to Estimate Execution Effort of Software Testing", International Conference on Software Testing, Verification and Validation, 2010.

[17] W. B. Langdon, S. Yoo, M. Harman, "Inferring Automatic Test Oracles", IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST), 2017.

[18] N. DiGiuseppe, J. A. Jones, "Concept-Based Failure Clustering", International Symposium on the Foundations of Software Engineering (FSE), 2012.

[19] J. Ding, X-H. Hu, V. Gudivada, "A Machine Learning Based Framework for Verification and Validation of Massive Scale Image Data", IEEE Transactions on Big Data, 2017.

[20] M. Hall, "Correlation-Based Feature Selection for Machine Learning", Ph.D. Dissertation, The University of Waikato, Hamilton, New Zealand, 1999.

[21] J. Guo, J. Cheng, J. Cleland-Huang, "Semantically Enhanced Software Traceability Using Deep Learning Techniques", IEEE/ACM 39th International Conference on Software Engineering, 2017.

[22] J-W. Lin, F. wang, P. Chu, "Using Semantic Similarity in Crawling-Based Web Application Testing", IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017.

[23] J. Li, P. He, J. Zhu, M. R. Lyu, "Software Defect Prediction via Convolutional Neural Network", IEEE International Conference on Software Quality, Reliability, and Security (QRS), 2017.

[24] B. Busjaeger, T. Xie, "Learning for Test Prioritization: An Industrial Case Study", ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2016.

[25] Q. Luo, D. Poshyvanyk, A. Nair, M. Grechanik, "FOREPOST: A Tool For Detecting Performance Problems with Feedback-Driven Learning Software Testing", 38th International Conference on Software Engineering Companion (ICSE), 2016.

[26] J. Petrić, et al., "Building an Ensemble for Software Defect Prediction Based on Diversity Selection", ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2016.

[27] X. Xie, et. al., "Application of Metamorphic Testing to Supervised Classifiers",9th International Conferencee on Quality Software, 2009.

[28] P. Chiu, J. Reunanen, R. Luostari, H. Holma, "Big Data Analytics for 4.9G and 5G Mobile Network Optimization", IEEE Vehicular Technology Conference (VTC Spring), 2017.

[29] H. Hosseini, B. Xiao, M. Jaiswal, R. Poovendran, "On the Limitation of Convolutional Neural Networks in Recognizing Negative Images", 16th IEEE International Conference on Machine Learning and Applications (ICMLA), 2017.

[30] A. Memon, et al., "Taming Google-Scale Continuous Testing", 39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017.