ARTICLE REPRINT

Reprinted with permission from CIGRE, August 2021
Copyright © August 2021

S&C ELECTRIC COMPANY
Excellence Through Innovation

Artificial Intelligence Application in Grid Edge Asset Health Monitoring

Q. GUO, R. ZHANG, V. OSTAPCHUK,
W. WATTS, Y. SHARON, D. GUINN, M. GAO
S&C Electric Company
USA

SUMMARY

The problem: utilities and industrial applications have expensive equipment where quickly resolving or predicting malfunctions could increase revenue, reduce operations and maintenance costs plus save significant amount of time and effort in resolving issues. Machine learning is commonly done in the cloud but transporting the necessary volume of data to a server is both cost prohibitive and a cyber security issue.

This paper describes a process with proven results showing trained AI models for device health, predictive maintenance applications, and detecting manufacturing defects using supervised learning. In addition, the paper shows how to measure an AI model's performance with meaningful metrics, iteratively improve the model's performance and lastly deploy them to a constrained edge device.Pulseclosing Technology Applications

As established in the previous sections, the essence of PulseClosing Technology is the ability to detect the presence of a fault without stressing the system or disconnecting power to customers. Applying the concept not only at the device level, but at the system level presents new ways to evaluate the use and benefits of PulseClosing Technology.

KEYWORDS

Artificial Intelligence, Machine Learning, Edge Computing, Grid Edge Asset Health

1. Problem Statement / Introduction

The electric grid has gone beyond just generation and transmission/distribution infrastructure, the need to operate a safe, reliable, and sustainable grid has led to exponential growth of grid edge control devices, both the amount of them and the level of complexity have increased over the past years and this trend continues. The complexity of the control itself poses challenges to monitor overall control system health, requiring multiple components and factors to be considered at the same time to determine system status, and it is often the case where system effect of multiple factors cannot be deduced in a straight-forward way by engineering looking at discrete data points. On the other hand, these complex control devices often come with high computation power, good data sampling and storage capacity, which opens the door for applying data driven health status monitoring to detect manufacturing defects before deployment or malfunctions while deployed.

Machine learning can be a solution to these types of problems, but can be prone to mistakes at every step, requiring:

- Data collection
- Data filtering and normalization
- Labeling the datasets
- Iteratively training and improving the models
- Deploying them on to the grid edge devices

This paper investigates how to collect data, train the model while measuring the model's performance then deploy the model. In the end, the process and resulting model(s) can be used for Artificial Intelligence to be applied within a grid edge device to monitor its own health, to increase device owner's capability to quickly grasp events and anomalies in the network so prompt and appropriate action can be taken.

2. Data Acquisition

a. Simulated data vs real-world

Data is paramount to training models for machine learning and one of the first big challenges is how to obtain data?

There are two types of data: simulated data is generated and collected in a controlled lab environment. The second type, real-world data, is gathered from the 'natural habitat' where the edge device or product is being deployed. Simulated data can be generated without end; however, it cannot fill the reality gap since not every situation can be mimicked accurately. Simulated conditions can only be as good as the extent of

what is covered by the programmer. Mining real-world data would provide a more accurately trained model which covers all the edge cases. This does not mean simulated data is not necessary. Real-world data is sometimes dangerous and costly to collect but some AI algorithms are focused mainly on predicting those 'rare' events; if these rare events are hard to come by but simulating them is possible, it would open up the possibility to choose how many of these rare events you want to simulate. The model created from simulated data is necessary and is very helpful and will come close to the real-world data trained model, but it is also necessary to mix real-world data to train a model for more reliability and accuracy. Regardless of the source of data, they are all in essence just data from the perspective of the model, the data will help cover all the cases including edge cases and improve the model.

In many cases, including ours, we tested the edge devices in different environments, conditions, and edge cases in a lab, collecting the different data from the sensors by a microcontroller to use for training and testing the models.

b. Challenges Filtering Data

Depending on the source of the data and the quality of the data, it may need to be filtered and/or cleaned. Real-world data can be missing or incomplete, and if used as-is to train a model, will result in a model with low accuracy: therefore, we may have to fill in or remove data. When tests are run back-to-back, such as with automated tests to gather simulated data or a long collection of real-world data, potentially transitioning between different scenarios, it's important to separate the data to be used for training data sets. If the transition data is included in the training and/or testing, it could cause the model to expect data that may never happen in the real-world, resulting in a model that performs with poor accuracy.

c. Data splitting for training and testing

Once the training data has been filtered and cleaned, the next step is dividing the data into 2 subsets. The first set to determine how much data to train the model, the training data. The second set, what data will be used to test the performance of the model, the test data. The reason all data should not be used solely to train the model is it makes it difficult to evaluate the performance of the model because the model was trained with all the data, so it has seen all of it.

3. Models

The next step is to determine what machine learning algorithm to train and use, in this case, we will be looking at SVM and Neural Networks.

a. SVM

SVM stands for Support Vector Machine, a supervised machine learning algorithm. The idea for SVM is to separate labeled data as much as possible, for example, if the data is labeled as two classes, the goal is then to draw a separation line that maximizes the distance between itself and the closest data points from either class. SVM can handle data with multiple classes and those that are not linearly separable as well.

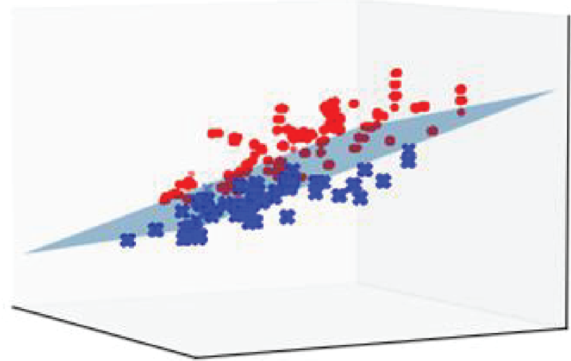


Figure 1. Our SVM Model trained with 5 features, 3D plot for 3 of the 5 features.

SVM is a convex optimization problem which means the global optimum solution is guaranteed to be found, unlike neural networks, which can be stuck in local optimum solutions as explained in the next section. Stochastic process is not required for training SVM models, therefore training with the same training data always results in the same consistent model. SVM can handle thousands of training data and many features yet is efficient. SVM has fewer hyperparameters to tune, which makes it easier to develop a model compared to neural networks.

While SVM can handle data with classes that are not linearly separable, its performance with nonlinear data may not be as good as neural networks. A SVM model may suffer from overfitting if not tuned carefully. Overfitting is the phenomenon where the model performs well on training data, but not on testing data. The reason is that the model is trained to fit the training data so well that it also picks up the noises in the training data.

b. Neural Network

Neural networks (NN) are a subset of machine learning and is similar to the behavior of a human brain, it allows the computer program to recognize patterns and solve common problems with deep learning algorithms. A human brain has a network of billions of neurons, a neural network is comprised of a multi-layered network of many neurons. The more neurons and more layers, the higher the complexity of the model.

Neural networks are great for fitting complex models e.g., non-linear models. They are also great for models with large datasets of hundreds of thousands to millions. It is reliable in an approach of tasks involving many features. It works by splitting the problem of classification into a layered network of simpler elements. Neural Networks work best with more data points. It can be also extremely flexible in the types of data they can support; they do a decent job at learning the important features from basically any data structure, without having to manually derive features.

However, with advantages come some disadvantages, it requires large amounts of data compared to other machine learning algorithms, such as SVM, as it needs at least hundreds of thousands if not millions in labeled samples. This leads to overfitting and generalization problems, in other words, the model might be tuned too perfectly to the data it sees and perform poorly in predicting data it has not seen before. Neural networks are also more computationally expensive and takes more time to train compared to other algorithms, such as SVM. The training time can take a few minutes to hours to days, it depends on the size of the data, the complexity of the neural network, and the hardware doing the computation. Randomness and indeterministic nature in the training of the Neural Network sometimes leads to getting stuck in local minima when searching for the optimal global minimum resulting in inconsistent and unreliable model generation using the exact same training data; thus, it requires multiple runs to train a model that has good validation and testing results and not overfit the model. Since there are no specific rules for determining the structure of a neural network, it requires a lot of effort in trial and error to find the optimal network structure to achieve the best performance.

4. Metrics

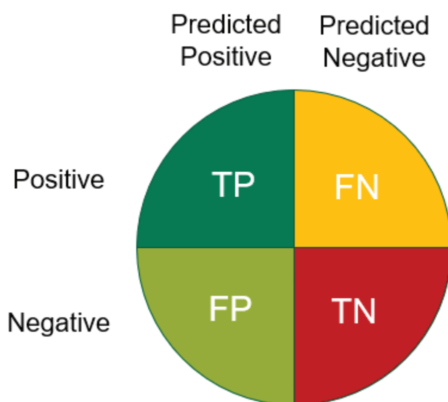


Figure 2

a. Accuracy

Accuracy gives an overall rating on all the correctly identified results from all the data points. One thing we learned quickly when working with machine learning is Accuracy is not the end all be all metric. On one of our abnormal conditions for Device Health, the first trained model had an 85% accuracy, an excellently performing model on the first try. After looking into it further though, we realized the data set had 85% of the first abnormal condition and so the model was just reporting everything had that abnormal condition, resulting in a lot of false positives. Based on our understanding of why the accuracy was so high, we concluded we needed additional metrics to ensure we could measure the performance of our models.

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

b. Precision

Precision shows how precise the model is out of those predicted to be positive, how many of them are actually true positive. The reason this is important is because the costs of a false positive can be incredibly high. The previous example showed 15% of the results were false positives, imagine if a device frequently had false positives for its health, that could result in a lot of requests for maintenance on a device/product that didn't need it, resulting in wasted truck rolls, therefore wasted time and money.

$$\text{Precision} = \frac{TP}{TP + FP}$$

c. Recall

Recall on the other hand calculates how many of the actual positives are successfully identified by the model. Recall is useful in cases where a False Negative would be costly, so in this case an abnormal condition is not correctly identified, for example, a component that will no longer work in 6 months is classified as operating normally, this may eventually cause a product to not function correctly which could cause an unexpected outage resulting in an emergency truck roll instead of scheduling maintenance to resolve the issue.

$$\text{Recall} = \frac{TP}{TP + FN}$$

d. Neural Network vs SVM

One major difference between SVM and Neural Networks is SVM models can create highly accurate models with much smaller training data, thousands of datasets compared to hundreds of thousands to millions needed for Neural Networks. SVM models requiring smaller datasets allow rare

conditions or shorter time to collect data yet resulting in highly accurate models to be trained faster and with less effort. The following figures demonstrate our initial efforts to classify the edge device's health and detect manufacturing defects:

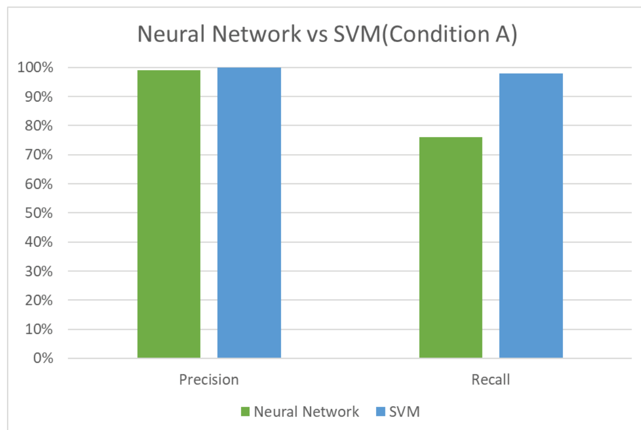


Figure 3

Figure 3 shows using training data containing 26000 datasets, comparing the results for abnormal condition A, the neural network was able to catch most the abnormal condition A (99% precision), but showed a lot of false negatives: it had only a 77% recall and an overall accuracy of 94%.

On the other hand, the same training data on SVM, resulted in 100% precision for abnormal condition A, with significantly fewer false negatives (recall of 98%) and an overall accuracy of 99.5%. These results show with just thousands of datasets, SVM can train accurate models compared to the more computationally complex Neural Networks.

Some abnormal conditions needed larger datasets; the following figure 4 had about 500000 datasets with 4 features for detecting abnormal condition B:

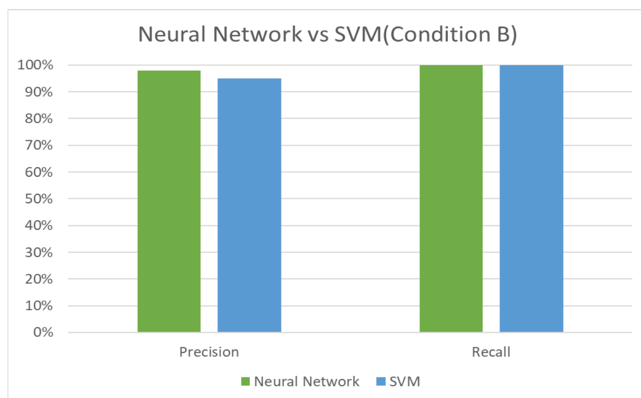


Figure 4

Figure 4 shows the Neural Network demonstrated a 97.8% precision at identifying the abnormal condition B compared to the SVM's 95%, a small but meaningful improvement.

5. Training Model

a. Steps to train model

Preparation of the data is a big and necessary step before feeding it into the model. You must structure the data into a consistent and compatible format, convert or compute the features, remove data that has problems or should be ignored, remove duplicate data, convert the data to the same units, and finally label the data. Data is then normalized over the combined dataset which is explained in the normalization section.

After the data is processed, the machine learning model is trained using that data. Hyperparameters need to be configured and selected for training. The standard and simplest way to configure a Neural Network is to set the number of layers, number of epochs, number of hidden nodes, and the learning rate. Depending on your model, you could also specify additional hyperparameters to handle overfitting such as Dropout, L2 Regularization, and Batch Normalization. In SVM, a kernel needs to be selected for the model. Common kernels include linear, polynomial, and RBF. All kernels have a common hyperparameter C that needs to be selected. C specifies the model's error tolerance, a smaller C means smaller error but if C is too small, the risk of overfitting increases as discussed in Lessons Learned. For RBF kernel, there is an additional hyperparameter γ that needs to be selected. Once the model is configured properly, the training data is then fed into the model, the result is a trained model which can be used to run your inferencing on your target or on a computer/server.

b. Evaluate model / Example applying metrics to model

We evaluate the model by looking at the training results as well as the testing results. In those results, we see how good the training results were; generally, there are no issues there, however, based on the testing results we might infer from the training results that it was 'too perfect' which suggests that the model might be overfitted and thus it's necessary to investigate possibilities of overfitting. The testing results will provide us with the accuracy, precision, and recall, which will tell us how well the trained model classifies on the testing data that it never has seen before. Based on these results, we are able to evaluate how good is the trained model.

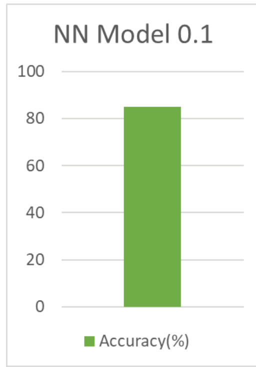


Figure 5

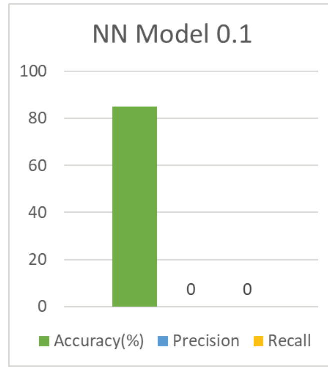


Figure 6

As previously stated, the first neural network model trained had an excellent accuracy right off the bat of 85% for detecting abnormal condition A as shown in figure 5. As seen in figure 6, when adding in the metrics precision and recall from the perspective on the edge devices running normally with no abnormal conditions, it shows both precision and recall were 0% meaning the model was overfitted and not performing well at all.

c. Iteration / How to improve the model

You can improve the model by adding more features and collecting more data for training. The variety of data could also improve your model such as data collected under different conditions or, if there is a feature which you did not include, it would provide the model with an additional N number of data points alongside the other N numbers of data points for other features. This additional information to the model can improve the results of the model prediction. If you're constrained by the amount of data, then the model can only be improved by how you tweak the hyperparameters of the model. If you're under performing in the testing then it could be due to overfitting, and you can address that by building a less complex model or address overfitting issues through other techniques. Other cases for under performance could also be due to noise, techniques such as applying filters to the data or normalizing the data could help negate these outliers. Originally our model with a smaller set of features:

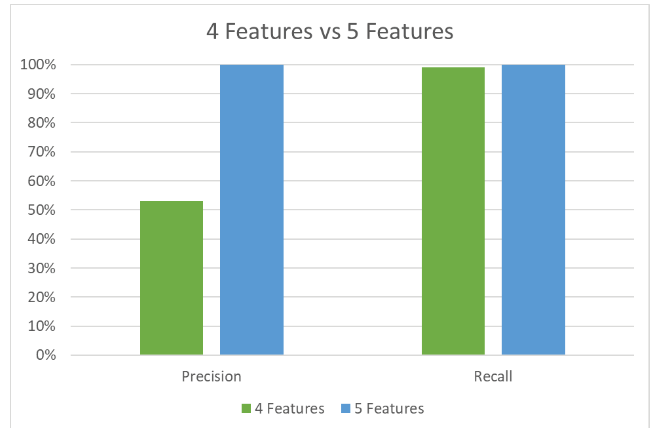


Figure 7. 2 Neural Network models trained and tested for detecting abnormal condition C on the same set of data except one had 4 features and the other had an additional feature.

As can be seen from figure 7, the model with 4 features in green showed a precision of 53%, indicating many of the normal edge devices were incorrectly flagged with an abnormal condition C (which would have resulted in unneeded maintenance). The model with 4 features also showed 1% false negatives, showing some of the abnormal condition C were not successfully detected. The new model trained by adding a 5th feature increased the precision to 100% and recall to 100% meaning there were no false positives and no false negatives.

By comparing the metrics, precision, recall, and accuracy, we were able to clearly demonstrate adding the new feature improved everything across the board for precision, recall and accuracy, eliminating false positives and false negatives and hence perfect accuracy for abnormal condition C.

6. Deploying to Target

a. Export model

Once a model has been trained, the next step is to deploy it to the edge device. For each type of machine learning algorithm, there are different ways to deploy the models depending on the hardware from processors to programmable logic (CPLD/FPGA). By deploying the model to run on the edge device, it can use the locally generated data to detect device health issues or predictive maintenance applications without the need to send data to a server. In our case, both NN and SVM models were deployed to the edge device (target).

b. Tools (TensorFlow Lite) to run

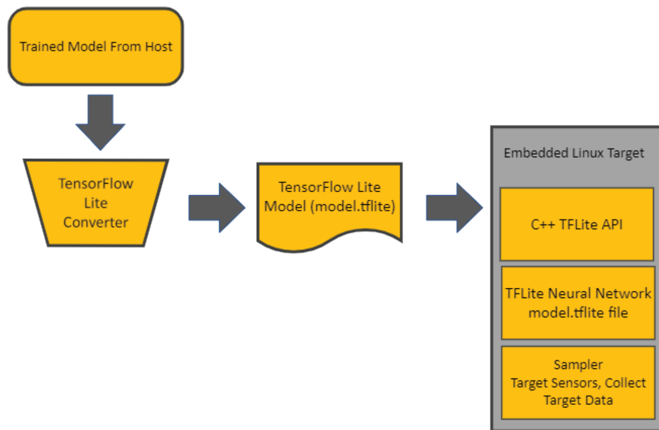


Figure 8

NN inference program for the target was written utilizing TensorFlow Lite's C++ library. The TensorFlow Lite (TFLite [1]) flow on the target works as follows: load model that is exported from host, sample-convert-normalize feature data from target, utilize TensorFlow Lite API to run inference with model, interpret output from model's inference. The model is loaded with a TFLite API from an exported model.tflite file that resides on target. The data samples are collected from the target's sensors via a kernel module that enables C++ program access. The TFLite API is used to feed normalized sample data into the model for inference. TFLite methods return a list of predictions mapped to confidence values. A C++ function produces a model prediction based on the label associated with the highest confidence value.

c. SVM sklearn-porter conversion for target

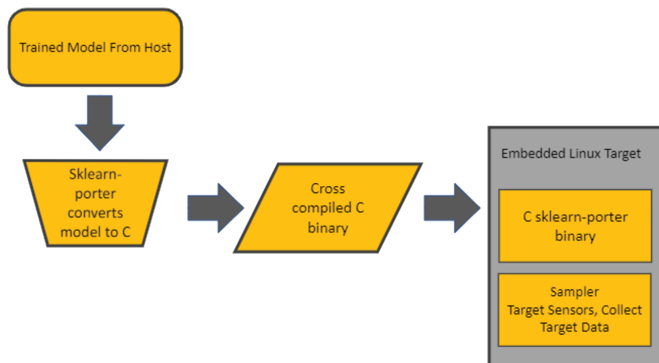


Figure 9

The SVM inference program for target was written utilizing an open-source library sklearn-porter[2]. The model exported with the sklearn-porter library is auto generated C header and source files. The C program can be modified in order to include functions needed to query sensors or data sources from the target. The C program is then cross compiled for the target and deployed. The SVM application flow on target works as follows: sample-convert-normalize features data from target, utilize SVM functions to run inference, and the SVM inference functions returns a prediction. There is no need to load the model as detailed in the TFLite NN deployment flow because the SVM model is already compiled into the application binary.

d. CPU/memory metrics

CPU and Memory usage was characterized with the reported memory from the memory proportional set size metric (PSS). PSS is the total RAM usage of a process and includes the memory shared with one or more processes. The shared memory can include the part that the SVM or NN network use of a particular shared library, such as the proportion of libpthread that a process uses in conjunction with other processes using libpthread.

CPU utilization of each machine learning process was measured for our NN and SVM applications: sampled values from the target, calculated features, and ran inference engine. The sampling-inference cycle occurred every second. The CPU usage of each application reached peak CPU during this cycle. Our target system is running on a quad core ARM 1GHz processor. The following table shows the Linux process stats - Memory, CPU, Inference Time - for the Neural Network and Support Vector Machine applications.

App	USS	PSS	RSS	CPU	Inference Time
NN	2.9 M	3.4 M	5.0 M	0.7%	45 µs
SVM	648 K	1.1 M	2.7 M	0.7%	380 µs

e. Challenges

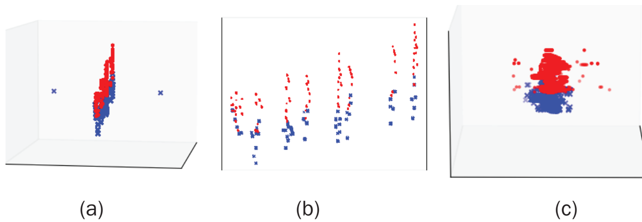
There were the following challenges associated with deploying NN and SVM applications on an embedded system target: optimizing SVM application for faster inference time; reducing TensorFlow Lite NN application RAM footprint; decoupling SVM application model for ease of deployment; and adding configuration ability for feature normalization factors.

We measured an inference time of 380 µs on the SVM application which was much slower than the NN inference time. The SVM sklearn-porter model functions could be optimized to improve inference time. TFlite library is compiled from many opensource libraries and reducing RAM would require determining which of these components could be removed or memory optimized. The SVM model should be decoupled from the binary in order to allow for easier deployments. Our current SVM model requires that the binary be cross compiled and redeployed every time the model is adjusted, a step that is avoided when deploying a TFLite model. Features fed into the model often require normalization factors. Adding these normalization factors to a configuration file, e.g., yaml, or json, would speed up the deployment process if normalization factors need to be changed due to new model deployment.

7. Lessons Learned

a. Data visualization

Throughout the investigation, we found that data visualization helped. It helps investigate data irregularities, select the appropriate machine learning model, and evaluate input features being used. By plotting the samples with a subset of input features on a 2D or 3D space, it is possible to spot the irregular samples that are far from all other samples, which means those samples are anomalies which could be discarded; it is also possible to check if the samples are linearly separable, which helps with the model selection (linear vs. nonlinear). Figure 10 shows how visualization helps identify two samples with anomalies (figure 10 (a)) and samples that are not linearly separable (b):



(a) (b) (c)

Figure 10

How visualization informs machine learning

By plotting the samples using different subsets of features, it is possible to see what features are more significant in deciding the classification. If no feature is significant in deciding the classification, then the data might not be separable at all with the features being selected, and more features might need to be considered before training any machine learning model. Figure 10 (c) is an example of samples where the feature along the horizontal axis may not be significant for the classification.

b. Training Data

Although it may seem obvious, it is not always easy to make sure that the training data collected is representative of the possible scenarios. It is always beneficial to consider as many scenarios as possible when collecting training data. It is the nature of machine learning algorithms that they may not know what to do with new input that is not similar to any training that the model has seen.

Another lesson about training data collection is not to discard features in an early stage just because they are not obviously relevant to the problem. In our experiment, some features that seemed not relevant to the device's health conditions being identified turned out to be useful and helped improve the performance of our models, as shown in figure 7.

c. Overfitting

How to detect and overcome overfitting?

One way we came to detect overfitting is just by looking at the complexity of the model, training results, and trial and error. First plot the data points on a multi-dimensioned graph to visualize the contour line or plane of the model then see whether the model tries to, for example, fit data that are outliers which leads to incorrectly classified data, see figure 11 for an example.

In Neural Network there are several techniques which can help with overfitting, the easiest technique is early termination, which means stopping the training as soon as the cost function has stabilized at a reasonable level. L2 Regularization, dropout, and batch normalization are additional techniques in Neural Network to address overfitting.

For SVM, we mitigated overfitting by tuning the hyperparameters. Specifically, we compared the performance of different hyperparameter combinations on a set of validation data that the training process has no visibility to.

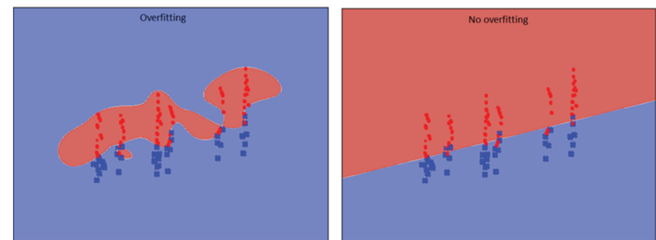


Figure 11. Data showing overfitting vs not overfitted

d. (Consistent) Data Normalization

In machine learning, the input features are usually normalized so they are on similar scales. If an input feature X is uniformly distributed, we normalize it as follows:

$$\bar{x}_i = (x_i - \mathbf{mean}_i\,(x_i)) \,/\, \mathbf{max}_i\,(x_i),$$

where x_i is the value of feature X for the ith sample.

The data normalization should be consistent across training and deployment. In other words, the mean and maximum calculated from training data and used by the training process should be passed to deployment without any change. If there is a discrepancy between the normalization factors used for training and those used for deployment, the inferencing process on target will give incorrect results. The best practice is to pass the normalization factors from training to deployment automatically when exporting the model for target.

To demonstrate the need to normalize data, while training our neural network model to detect our edge device's abnormal conditions with the raw data without any normalization, it resulted in the following model:

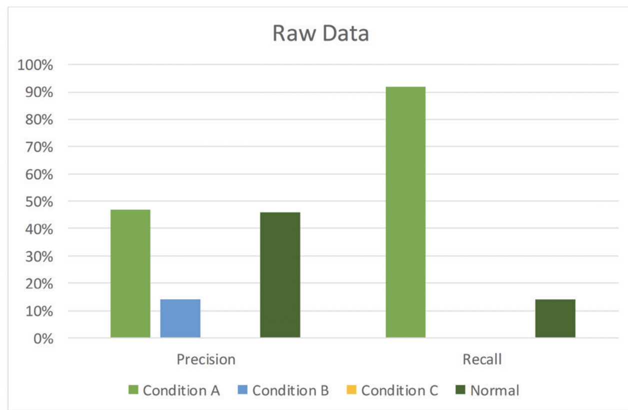


Figure 12. Model trained and tested with raw data

The model in figure 12 is trained and tested with raw data, resulting in an overall accuracy of 25%, demonstrating Condition C could not be detected at all (0% precision). The other conditions only being detected 46% for Condition A and 14% for Condition B therefore doing a categorically poor job at detecting abnormal conditions with the edge device.

Figure 13 demonstrates when we used normalized data to train the model but fed raw data into the model:

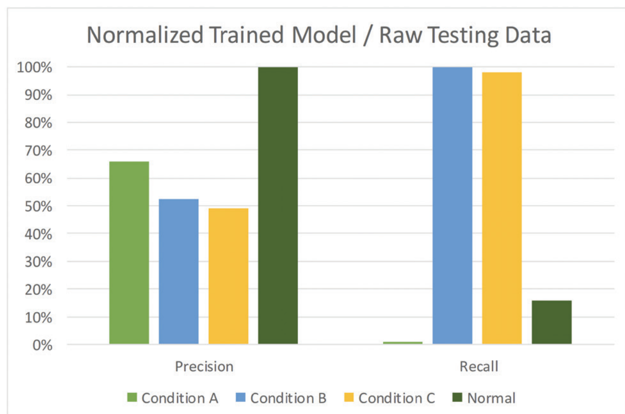


Figure 13. Model trained with normalized data but tested with raw data

The new model trained with normalized data but tested with raw data showed a drastic improvement, with all the abnormal conditions detected to some degree, as demonstrated by the precision between almost 50% to 66% of the abnormal conditions successfully classified. The model shown in figure 13 resulted in an accuracy of 53%.

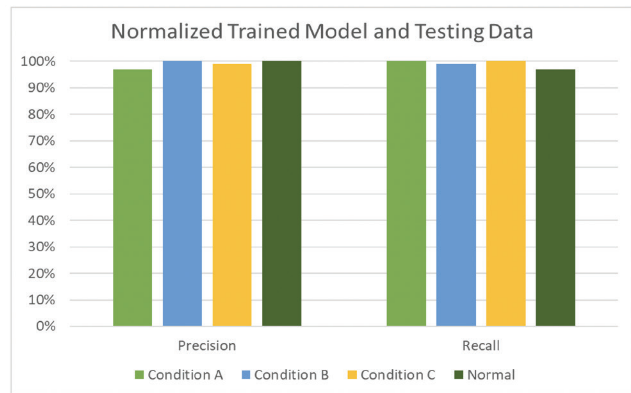


Figure 14. Model trained and tested with normalized data

Last, the model in figure 14 is trained with normalized data running on the edge device, which normalized all the sensor data resulted in catching all the abnormal conditions between 97 to 100% of the time with virtually no false positives as demonstrated by the Recall. The model's accuracy was 99.2%.

8. Conclusion

Monitoring overall system health of electric grid edge control device is a challenging task, due to complexity of control itself. Leveraging the computation power, data acquisition and storage capability of these devices, AI and machine learning can be an effective tool to this task, in that it can help capture potential system level issues that are not obvious to human inspection. Although machine learning process itself requires large amount computational resource to train highly accurate models, our results show by combining the data collection and metrics for evaluating the quality of the models results in models with high accuracy, low to no false positive or false negative conditions. Further as demonstrated by our models running on our edge devices, the model inferencing on target only takes a limited amount of computational resource to run and is thus feasible for most grid edge devices.

BIBLIOGRAPHY

[1] https://www.tensorflow.org/lite/

[2] https://github.com/nok/sklearn-porter