

# Enabling machine learning on resource constrained devices by source code generation of the learned models

Tomasz Szydło<sup>1</sup>, Joanna Sendorek<sup>1</sup>, and Robert Brzoza-Woń<sup>1</sup>

AGH University of Science and Technology,  
Department of Computer Science, Krakow, Poland

**Abstract.** Due to the development of IoT solutions, we can observe the constantly growing number of these devices in almost every aspect of our lives. The machine learning may improve increase their intelligence and smartness. Unfortunately, the highly regarded programming libraries consume too much resources to be ported to the embedded processors. Thus, in the paper the concept of source code generation of machine learning models is presented as well as the generation algorithms for commonly used machine learning methods. The concept has been proven in the use cases.

**Keywords:** IoT, Edge Computing, Machine Learning

## 1 Introduction

Due to the development of IoT solutions, we can observe the constantly growing number of network enabled devices in almost every aspect of our lives. It includes smart homes, factories, cars, devices and others. They are sources of large amount of data that can be analyzed in order to discover the relations between them. As a result, they can provide functionalities better suited to the needs, predict failures and increase their reliability.

The data generated by the devices can be used by machine learning algorithms to learn and then make predictions. For example, the historical information of engine behaviors may lead to the machine learning models that can be used to predict in advance failures of other engines and be used to plan appropriate repairing actions. Such an approach is possible because of the virtually unlimited resources in the computational clouds to store and process the data from large number of devices.

Such a concept is extremely important in the industry which is facing the revolution termed Industry 4.0. The main concept is focused on including cyber-physical systems, IoT and cognitive systems in the manufacturing. In the so-called smart factories, every aspect of the manufacturing process will be monitored in real-time and then gathered information will be used by the cooperating systems

and humans to work coherently. At the same time, the machine learning algorithms may gain the quality of the final products and decrease the production costs.

One of the important aspects in the industrial IoT is the response time of the systems. For example, in the factory automation, motion control and tactile Internet the acceptable latency is less than 10ms[8]. It means that the IoT systems using machine learning algorithms in the cloud for that kind of applications are not sufficient due to the fact that Internet routing to the worldwide datacenters introduces significant delays[12].

One of the solutions to circumvent that drawback is to move machine learning algorithms to the edge of the network [10] e.g. to the data center located in the factory and learn only on the local data. As a result, the latency introduced by the communication protocol would be significantly smaller because limited to the local networks, but the gained knowledge would be incomplete. The promising improvement would be to perform machine learning in the cloud environments on a large volume of data and then send learned models to the edge datacenters in order to make predictions locally e.g. in the factories. That approach would increase the accuracy of the predictions due to the variety of sources that data came from in the learning process.

Nevertheless, even with that approach, the devices have to be constantly connected to the local computer network in order to use the machine learning models. Thus, in the research we are moving machine learning models to the embedded devices itself. In our concept, instead of implementing machine learning libraries for embedded devices that can read and interpret the learned models, they are converted to the source code that can be compiled in the device firmware. This enables possibility to embed these models into embedded processors that may have sporadic access to the network.

The concept presented in the paper can be used to design e.g. smart tools in which machine learning models are used to prevent their damages by modifying internal characteristics according to the usage. Such devices during charging could synchronize itself with a cloud by sending the historical usage logs from their memory and download new firmware with updated machine learning models. The process can be automated using mechanisms presented in the paper.

The scientific contribution of the paper is (i) the concept of source code generation of machine learning models (ii) the generation algorithms for commonly used machine learning methods and finally (iii) practical verification of the method.

Organization of the paper is as follows. Section 2 describes the related work in the field of machine learning for constrained devices. Section 3 discusses concept of the proposed method and the algorithms for commonly used ML algorithms. Section 4 describes the evaluation, while section 5 concludes the paper.

## 2 Related work

At the time of writing, numerous machine learning programming libraries are available on the market. They offer a number of algorithms to enable learning with and without supervision. They can be divided into dedicated applications for individual computing nodes (for example Weka, SMILE, scikit-learn, LibSVM) and for high performance computers (cluster/cloud computing e.g. Spark, FlinkML, TensorFlow, AlchemyAPI, PredictionIO). Many large companies offer services which rely on machine learning in public cloud infrastructures. The most popular services of this type are BigML, Amazon Machine Learning, Google Prediction, IBM Watson and Microsoft Azure Machine Learning and the dedicated for IoT such as ThingWorx. These solutions analyze data mostly in the cloud and role of IoT devices comes down to software agents providing data for analysis. Solutions categorized as Big Data Machine Learning and dedicated for cloud computing are a fast-growing branch of machine learning[2].

In the domain of resource-constrained systems we can find many implementations of ML algorithms on mobile and embedded devices that cooperate with the cloud computing. The work of Liu et al. [7] describes an approach to image recognition in which the process is split into two layers: local edge layer constructed with mobile devices and remote server (cloud) layer. In [6] the authors present a software accelerator that enhances deep learning execution on heterogeneous hardware, including mobile devices. In the edge, i.e. on a mobile devices, an acquired image is preprocessed and a segmentation is performed. Then the image is classified on a remote server running pre-trained convolutional neural network (CNN). In [9] the authors propose the utilization of Support Vector Machine (SVM) running on networked mobile devices to detect malware. A more general survey on employing networked mobile devices for edge computing is presented in [11].

There are also implementation of algorithms related to machine learning domain on extremely resource-constrained devices with a few kB of RAM. In [4][5] authors develop extremely efficient machine learning algorithms that can learn on such devices. The problem presented in the paper addresses the same group of devices but is not related to the performing learning process on them but is related to the usage of the models learned elsewhere and used on the devices. It enables possibility to design systems that can perform machine learning in the clouds on a large volume of data and then use the results in the resource-constrained devices.

## 3 Concept of the method

In the IoT domain there are several hardware architectures and sets of peripherals in the processors used in the devices[1]. Generally, they can be classified into two categories - application processors that can run Linux and the embedded ones that can run real-time operating systems such as FreeRTOS or be programmed directly on the bare-metal.

On the devices with application processors such as RaspberryPi, the tuned versions of machine learning libraries such as `Tensorflow` or `scikit-learn` can be executed due to the availability of Java, Python and other programming languages. This means that machine learning models can be directly copied between the cloud environment and the device only if the same libraries are used in both places.

The other approach assumes that the models can be moved between various ML libraries. For that purpose, description languages such PMML[3] has been developed. For example, models can be learned in the cloud using Big Data tools then after export/import operation used by the libraries ported to the embedded devices.

The problem is more complex with the second group of embedded devices such as Arduino with resources constrained embedded microcontrollers (MCUs). In this case, porting the high-level and general purpose machine learning libraries is not possible. In this situation, the implementation of description languages such as aforementioned PMML may consume significant device resources. Thus, the authors propose the approach in which source code of the estimator that expresses the learned model is generated and then compiled into the device firmware. The presented concept of the machine learning model source code generation requires three steps to be performed:

1. analysis of the machine-learning algorithm and the way how it can be expressed in the source code,
2. analysis on how to get details of machine-learning model from the ones generated by the particular software or library,
3. analysis on how the final code can be optimized for the target embedded architecture regarding its resource constraints.

In the next subsections, the source code generation algorithms for the commonly used machine learning methods for the classification problem are presented. Additionally the technical details on how to generate the source code based on the popular `scikit-learn` library is discussed. We have also analyzed how the final code should be generated for AVR and ARM embedded processors.

### 3.1 Bayes Networks generator

Naive Bayes algorithm is the method which applies probability theorem to the machine learning problems, treating input features and output classes as events. The problem of classification - assigning class for the given input features - is reduced to finding output class event which has highest conditional probability, assuming that input features event has occurred. To calculate the conditional probability, Bayes theorem is applied. Therefore, definition for classification problem can be written as:

$$\operatorname{argmax}_y (P(y|x_1 \dots x_N)) \stackrel{\text{Bayes th.}}{=} \operatorname{argmax}_y \frac{P(y)P(x_1 \dots x_N|y)}{P(x_1 \dots x_N)}, \quad (1)$$

where:

- $x_1 \dots x_N$  - input features;
- $N$  - number of input features;
- $P(x_1 \dots x_N)$  - constant probability of input feature event which is the same regardless of output class
- $y$  - element of output classes events.

In order to calculate right side of equation (1), two assumptions are made:

1. Input features are pair-wise independent of each other which allows to calculate probability  $P(x_1 \dots x_N | y)$ .
2. The probability distribution of  $P(x_i | y)$  is normal distribution  $\mathcal{N}(\theta, \sigma)$ .

After applying both of the assumptions to the equation (1) and natural logarithm function to the density function of normal distribution, problem of classifying the set of features can be written as:

$$\operatorname{argmax}_y \left( \log P(y) + \sum_{i=1}^N \left[ -\frac{1}{2} \log 2\pi\sigma_{y,i} - \frac{(x_i - \theta_{y,i})^2}{2\sigma_{y,i}} \right] \right), \quad (2)$$

where:

- $M$  - number of output classes;
- $\sigma, \theta$  - matrices of size  $M \times N$  calculated during the learning phase - those relate to parameters of normal distribution;
- $P(y)$  - prior probability for class  $y$  calculated as the proportionate part of a class occurrences in the training set.

The necessity of calculating natural logarithm, the only part of equation requiring math module in C, can be eliminated by introducing third matrix -  $\sigma_{\log}$  containing element-wise logarithm function applied to matrix  $2\pi\sigma$ .

Therefore, formula (2) can be reduced to:

$$\operatorname{argmax}_y \left( \log P(y) - \frac{1}{2} \sum_{i=1}^N \left[ \sigma_{\log y,i} + \frac{(x_i - \theta_{y,i})^2}{\sigma_{y,i}} \right] \right), \quad (3)$$

which equation will be the base for construction of program evaluating Bayes model for new set of input features. Implementation of such evaluator in C is presented on listing 1.1.

Listing 1.1: Naive Bayes model evaluation in C

```
double sigma[M][N] = <learned values>;
double theta[M][N] = <learned values>;
double log_sigma[M][N] = <learned values>;
double prior[M] = <calculated values>;

double temp_sum;
double class_est[10];
```

```

for (int i = 0; i < M; i++){
    temp_sum = 0;
    for (int j = 0; j < N; j++){
        temp_sum += log_sigma[i][j];
        temp_sum += ((x[j] - theta[i][j]) * (x[j] -
            theta[i][j])) / (sigma[i][j]);
    }

    class_est[i] = prior[i] - 0.5 * temp_sum;
}

return get_max_index(class_est);

```

It can be observed that the evaluator code remains the same as to the structure, regardless of specific learned Naive Bayes model. The program has a structure with declaration part, where matrices  $\sigma$ ,  $\theta$  and  $\sigma_{log}$  are defined, and instruction part which implements formula (3). In case of specific trained model only matrices values has to be set, altogether with  $M$  and  $N$  constants. Therefore, generation process for naive Bayes algorithm may be reduced to using evaluator template and filling it accordingly with trained values. The other approach to generation will be presented in sections 3.2 and 3.3, where not only data declarations but whole program structure relies on trained model.

In scikit-learn, class `sklearn.naive_bayes.GaussianNB` implements aforementioned classifier. Trained instance of model stores values of matrices  $\sigma$  and  $\theta$  in fields `sigma_` and `theta_` respectively and values of prior probabilities for classes in array `class_prior_`. In result, demanded values for `theta` and `sigma` can be retrieved directly from trained model and values for `prior` and `log_sigma` can be calculated.

### 3.2 Decision Trees generator

Decision Tree classifier is based on the algorithm which recursively tries to split training dataset based on the value of one chosen input feature.

Figure 1 presents structure of example decision tree. Each node represents one training data split which corresponds to different condition on chosen feature value. The split condition is created in such a way as to minimize *gini index* in the child nodes. Gini index is calculated as presented on equation (4) and describes how well are output classes distributed through the dataset.

$$gini_{index} = 1 - \sum_{i=1}^M p_i^2, \quad (4)$$

where:

- $M$  - number of output classes;
- $p_i$  - fraction of representatives of class  $i$  in the whole dataset.

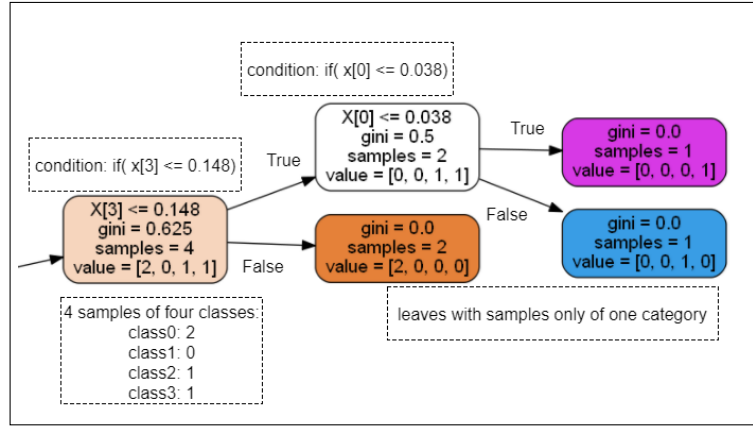


Fig. 1: Example decision tree structure.

Construction of tree is being conducted in learning phase of algorithm, based on training set. Once the tree is constructed, the classification of the new input sample is done by traversing the tree from top to bottom, evaluating conditions in each node and choosing appropriate child of the node until leaf is reached. Such a structure of trained model is equivalent to a set of hierarchical condition instructions and can be unambiguously conversed to such a structure.

In scikit-learn library, tree structure of trained classifier is held in `tree_` property of the classifier object and consists of commonly used pointer representation. Each node has an unique index used to reference its properties in properties arrays:

- `children_left` - array of left children indexes - index -1 means that there is no left child;
- `children_right` - array of right children indexes - index -1 means that there is no right child;
- `feature` - array of input features on which splitting is conducted;
- `threshold` - array of values on which splitting condition is based;
- `classes` - array of arrays holding count for each output class on given data subset.

Listing 1.2 presents pseudocode of algorithm which generates hierarchy of condition clauses based on trained classifier. The tree structure is processed recursively by pre-order traversal, using aforementioned properties arrays. Visiting each node, appropriate *if-else* clause is created which represents one data split.

Listing 1.2: Tree code generation algorithm.

```
generate_statements( tree ):

recurse( node , depth ):
```

```

if node is not leaf:
    indent = get indent for depth
    feature = tree.feature[node]
    threshold = tree.threshold[node]
    return (
        'indent' +
        'if clause' for given feature and threshold +
        recurse(tree.children_left[node], depth + 1) +
        'ending if clause' +
        opening of 'else clause' +
        recurse(tree.children_right[node], depth + 1) +
        closing of 'else clause' )
else:
    result = 'most numerous class for leaf'
    return 'indent' + result

return recurse(0, 1)

```

### 3.3 Neural Networks generator

For the purpose of the authors research and proving concept presented in the article, one class of neural network algorithms has been examined - multilayer perceptron (MLP) which is one of the less complicated neural network methods. MLP aim is to learn the function  $f: \mathbb{R}^N \rightarrow \mathbb{R}^M$ , where  $N$  is number of input features and  $M$  is the number of output classes. The learning process of neural network is out of scope of this paper, but understanding the model evaluation process - execution of function  $f$  used in example - is essential to explain code generation for MLP.

Equation (5) presents schema for function  $f$  execution. It consists of  $H + 1$  consecutive **layer transformations**, where  $H$  is the number of **hidden layers** and is the parameter of method, determined before training phase.  $i$ th layer transformation consists of the following steps:

1. linear transformation based on previous layer result multiplication by `coef[i]` matrix;
2. addition of vector `itc[i]` to the result of previous step;
3. application of the **activation function** which introduces nonlinearity to the method.

Initial vector for the first transformation is the vector of input features. Activation function for each layer apart from last one - for all hidden layers - is *ReLU* function defined as in equation (7). Last layer is activated by application of *softmax* function which enables interpreting last hidden layer result as the probability distribution over set of output classes. Classified output class is the one under index of maximum element in last transformation result vector. In



the schema described, elements learned during training phase are lists `coef` and `itc` holding parameters for steps 1 and 2 of the layer transformation.

$$\begin{array}{c}
 \left[ \begin{array}{c} x_0 \\ x_1 \\ \vdots \\ x_N \end{array} \right]^T \xrightarrow[\text{act.}]{\underbrace{\text{coef}[0]}_{N \times p_0} + \underbrace{\text{itc}[0]}_{1 \times p_0} \text{ReLU}} \dots \left[ \begin{array}{c} a_0 \\ a_1 \\ \vdots \\ a_{H-2} \end{array} \right]^T \xrightarrow[\text{act.}]{\underbrace{\text{coef}[H-1]}_{p_{H-2} \times p_{H-1}} + \underbrace{\text{itc}[H-1]}_{1 \times p_{H-1}} \text{ReLU}} \\
 \underbrace{\hspace{15em}}_{\text{H transformations for each hidden layer}} \\
 \left[ \begin{array}{c} b_0 \\ b_1 \\ \vdots \\ b_N \end{array} \right]^T \xrightarrow[\text{activation}]{\underbrace{\text{coef}[H]}_{p_{H-1} \times M} + \underbrace{\text{itc}[H]}_{1 \times M} \text{softmax}} \left[ \begin{array}{c} y_0 \\ y_1 \\ \vdots \\ y_M \end{array} \right]^T \xrightarrow[\text{k}]{\text{argmax}} y_k
 \end{array} \tag{5}$$

- $H$  - number of hidden layers (indexed as 0...H-1)
- *coef* - matrix of coefficients used to transform layers to different sizes
- *itc* - intercepts matrix
- $y_k$  - result of classification

$$\text{softmax}(v)_i = \frac{e^{v_i}}{\sum_{j=0}^{K-1} e^{v_j}} \quad \text{for } i = 0, \dots, K-1; \tag{6}$$

where:  $K$  - size of vector  $v$

$$\text{ReLU}(x) = \max(0, x) \tag{7}$$

From the description above it follows that model evaluation code for trained classifier could be implemented as a sequence of matrix operations on consecutive layers. Code for generation algorithm is presented on listing 1.3.

Listing 1.3: Multiple layer network evaluator generation.

```

generate appropriate headers
for i in layer_count - 1:
    generate coef matrix for layer i

for each hidden layer:
    generate layer transformation:
        1. declaration for new result vector
        2. loop of matrix multiplication
        3. generate vectors addition sequence
    generate ReLU activation on result vector

generate layer transformation

```

```
generate softmax activation on result vector

generate loop for max index search
```

### 3.4 Source code optimization for embedded processors

Resource constrained embedded microcontrollers (MCUs) may be equipped with different microprocessor cores and peripheral sets. From a software engineer point of view, the main difficulties in programming such MCUs are low computing power and small amount of available memories: both operating and for executable firmware storage. In typical MCUs, the non-volatile flash memory is much larger in storage size than the operating memory, because the latter one generates a higher production cost per storage unit.

The computational performance of resource-constrained embedded platforms is generally low when compared to general-purpose application units. There are only a few methods to increase the performance. For example, depending on a software developers skills, the code can be manually optimized or partially implemented in a low level language. That option may be difficult to implement in automated code generating software and the resulting code may not be easily portable between different MCU architectures. A relatively easy way of controlling a balance between code size and execution speed is to find a correct optimization level. GNU C compilers (GCC) offer various standard optimization levels. Below we list the selected ones.

- With *O0* the optimization is disabled,
- With *O1* the compiler tries to reduce the execution time and the output code size.
- With *O2* the compiler optimizes the code as much as possible without introducing a trade-off between the execution time and the output code size.
- With *O3* the compiler optimizes as in *O2* with a set of additional flags.
- The *Os* is referred to as optimization for size. It makes the compiler optimize the code similarly to *O2* but without increasing the output code size.

Usually embedded microcontrollers may run a relatively simple scheduler or a real-time operating system (RTOS), but do not run an application operating system. In those cases, the memory management relies partly on a software developer. As an example, the AVR 8-bit MCU family has the Harvard architecture in which program and data address spaces are separate. This makes it less convenient to declare read-only variables stored in the microcontrollers program memory. Therefore, the code generator should consider the target MCU architecture. For example, when writing and compiling code for AVR MCUs, a variable with the *const* modifier will be placed in the operating memory. In the case of generating code for previously trained models, we often need a large number of constant values. Storing them in operating memory may quickly cause a shortage of that resource. To store read-only data in the program memory and to retrieve their values the software developer must use a special-purpose

macros which work as additional declaration modifiers or access functions, e.g. *PROGMEM* or *pgm\_read\_float\_near*. That problem is non-existent in newer and more advanced microcontrollers which implement a single and unified address space. Those units do not need additional modifiers for objects in code to store and retrieve them to and from the MCU non-volatile memory. Usually, thanks to their more modern design, they are also equipped with more resources than 8-bit AVR.

## 4 Evaluation

In order to evaluate described code generation methods proposed in the paper, authors have prepared use case demonstrating how trained model could be used for classification on embedded device. The bigger the training set, the more complex and time consuming learning phase is and therefore the advantage of separating it from evaluation phase is the most evident.

For the evaluation purposes, two databases has been used. First one is the **mnist database of handwritten digits**<sup>1</sup> has been chosen. In order to retrieve dataset `fetch_mldata` function from **scikit-learn** library has been used. Dataset fetched this way consists of 70 000 samples, each being vector of length 784 representing one handwritten digit picture. Each picture has dimension of  $28 \times 28$  pixels arranged in row-major order. After choosing and loading described dataset, an instance of each classifier from section 3 has been created and trained on the randomly chosen ninety percent of dataset. For each of them, source code has been generated extracting model evaluation which has been used to classify handwritten digits on touch screen attached to devices. For MLP classifier one hidden layer with 15 neurons has been established.

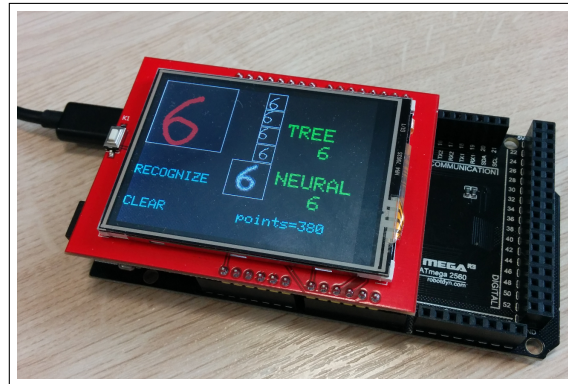


Fig. 2: Digit recognition application for Arduino that uses generated source code of the machine learning models for MNIST dataset

<sup>1</sup> <http://yann.lecun.com/exdb/mnist/> (access for 23 Feb 2018)

As an additional dataset, for comparison purpose, the iris dataset has been chosen which is much smaller than mnist one. The set contains of 150 samples divided into three categories representing variations of iris flowers: setosa, virginica and versicolor. Input features of samples consists of five parameters of iris flowers. Dataset has been divided into training and testing set similarly to mnist - ninety percent assigned for the training and ten percent assigned for the training. The exact same set of classifiers with parameters have been used for this dataset as for the mnist.

Table 1 contains the size of pickled models from `scikit-learn` for selected classifiers. It is worth to notice, that to use that models, the appropriate Python libraries are necessary thus the overall memory requirements are much larger.

The source code generators for machine learning models presented in the paper has been implemented in Python<sup>2</sup>. Based on the aforementioned models learned for the selected databases appropriate source codes were generated. Finally, the concept has been verified on two embedded platforms. First one, depicted in Fig.2 is based on Arduino Mega with ATmega2560 (8kB RAM, 256kB flash) microcontroller and a simple touch screen display. The second platform was STM32F4 Discovery board with ARM STM32F429 (256kB RAM, 512kB flash) microcontroller. Table 1 contains the size of compiled source-code for the learned models. For the Arduino platform, the Bayes model for mnist database was too large to fit into the memory, thus was not evaluated. For the other cases, the size of the memory footprint of the compiled classifiers was small enough to fit in the microcontrollers memory.

Table 1: Size of the serialized scikit-learn model and the compiled source-code of the classifier for the AVR and ARM processors

dataset	method		size of models in bytes						score
			scikit-learn	AVR	ARM				
					O0	O1	O3	Os	
iris	Bayes		771	2298	2352	2004	3440	2028	1.00
	MLP		12247	2360	4768	4004	5184	3936	0.933
	tree(float)		2501	272	592	512	16	480	0.933
mnist	Bayes		126164	—	190712	189980	190872	189956	0.556
	MLP		292984	52000	54088	52444	54992	52280	0.919
	tree	float	1051335	166476	158592	130336	132816	133200	0.874
		integer	1051335	75776	72832	53264	55920	54768	

## 5 Summary

In the paper we have presented the idea of how the machine learning models can be executed on the embedded devices with constrained resources. This allows

<sup>2</sup> <https://github.com/tszydlo/FogML>

developers for example to embed sophisticated failure prediction ML models in the home appliances such as toothbrushes, electric drills, kitchen mixers and others increasing their smartness.

The concept presented in the paper can be extended. We are currently working on two problems. First one is related to the mechanisms of how to combine incremental learning in the cloud from IoT sensors with automatic deployment of the learned models to the devices located in the edge environments. The second one is related to the development of the generator tools for Big Data ML such as TensorFlow or Apache Flink. The latter one would give a greater applicability and usefulness of the presented method.

## Acknowledgment

The research presented in this paper was supported by the National Centre for Research and Development (NCBiR) under Grant No. LIDER/15/0144 /L-7/15/NCBR/2016.

## References

1. Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., Ayyash, M.: Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials* **17**(4), 2347–2376 (2015)
2. Al-Jarrah, O.Y., Yoo, P.D., Muhaidat, S., Karagiannidis, G.K., Taha, K.: Efficient Machine Learning for Big Data: A Review. *Big Data Research* **2**(3), 87 – 93 (2015), big Data, Analytics, and High-Performance Computing
3. Grossman, R.L., Bailey, S., Ramu, A., Malhi, B., Hallstrom, P., Pulleyn, I., Qin, X.: The management and mining of multiple predictive models using the predictive modeling markup language. *Information & Software Technology* **41**(9), 589–595 (1999)
4. Gupta, C., Suggala, A.S., Goyal, A., Simhadri, H.V., Paranjape, B., Kumar, A., Goyal, S., Udapa, R., Varma, M., Jain, P.: ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices. In: *International Conference on Machine Learning*. pp. 1331–1340 (2017)
5. Kumar, A., Goyal, S., Varma, M.: Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things. In: *International Conference on Machine Learning*. pp. 1935–1944 (2017)
6. Lane, N.D., Bhattacharya, S., Georgiev, P., Forlivesi, C., Kawsar, F.: Accelerated Deep Learning Inference for Embedded and Wearable Devices using DeepX. In: *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services Companion*. pp. 109–109. ACM (2016)
7. Liu, C., Cao, Y., Luo, Y., Chen, G., Vokkarane, V., Ma, Y., Chen, S., Hou, P.: A New Deep Learning-based Food Recognition System for Dietary Assessment on An Edge Computing Service Infrastructure. *IEEE Transactions on Services Computing* (2017)
8. Schulz, P., Matthe, M., Klessig, H., Simsek, M., Fettweis, G., Ansari, J., Ali Ashraf, S., Almeroth, B., Voigt, J., Riedel, I., Puschmann, A., Mitschele-Thiel, A., Miller, M., Elste, T., Windisch, M.: Latency Critical IoT Applications in 5G: Perspective

- on the Design of Radio Interface and Network Architecture. *IEEE communication Magazine* (02 2017)
9. Shamili, A.S., Bauckhage, C., Alpcan, T.: Malware detection on mobile devices using distributed machine learning. In: *Pattern Recognition (ICPR)*, 2010 20th International Conference on. pp. 4348–4351. IEEE (2010)
10. Szydło, T., Brzoza-Woń, R., Senderek, J., Windak, M., Gniady, C.: Flow-Based Programming for IoT Leveraging Fog Computing. In: *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. vol. 00, pp. 74–79 (June 2017)
11. Tran, T.X., Hosseini, M.P., Pompili, D.: Mobile edge computing: Recent efforts and five key research directions. *MMTC Communications-Frontiers* **12**(4), 29–34 (2017)
12. Yi, S., Li, C., Li, Q.: A survey of fog computing: concepts, applications and issues. In: *Proceedings of the 2015 Workshop on Mobile Big Data*. pp. 37–42. ACM (2015)