ArduCode: Predictive Framework for Automation Engineering

Arquimedes Canedo, Palash Goyal, Di Huang, Amit Pandey, and Gustavo Quiros

Abstract— **Automation engineering is the task of integrating, via software, various sensors, actuators, and controls to automate a real-world process. Today, automation engineering is supported by a suite of software tools, including integrated development environments (IDEs), hardware configurators, compilers, and runtimes. These tools focus on the automation code itself but leave the automation engineer unassisted in their decision-making. This can lead to longer software development cycles due to the imperfections in the decision-making, which arise when integrating software and hardware. To address this problem, this article addresses multiple challenges often faced in automation engineering and proposes machine learning-based solutions to assist engineers tackle these challenges. We show that machine learning can be leveraged to assist the automation engineer in classifying automation code, finding similar code snippets, and reasoning about the hardware selection of sensors and actuators. We validate our architecture on two real data sets consisting of 2927 Arduino projects and 683 programmable logic controller (PLC) projects. Our results show that paragraph embedding techniques can be utilized to classify automation using code snippets with precision close to human annotation, giving an F_1-score of 72%. Furthermore, we show that such embedding techniques can help us find similar code snippets with high accuracy. Finally, we use autoencoder models for hardware recommendation and achieve a p@3 of 0.79 and p@5 of 0.95. We also present the implementation of ArduCode in a proof-of-concept user interface integrated into an existing automation engineering system platform.**

Note to Practitioners—**This article is motivated by the use of artificial intelligence methods to improve the efficiency and quality of the automation engineering software development process. Our goal is to develop and integrate intelligent assistants in existing automation engineering development tools to minimally disrupt existing workflows. Practitioners should be able to adapt our framework to other tools and data. Our contributions address important practical problems: 1) we address the lack of realistic data sets in automation engineering with two publicly available data sources; 2) we make the reference implementation of our algorithms publicly available on GitHub for other practitioners to have a starting point for future research; and 3) we demonstrate the integration of our framework as an add-on to an existing automation engineering toolchain.**

Index Terms— **Computational and artificial intelligence (AI) - machine learning, computer-aided software engineering, software engineering.**

I. Introduction

INDUSTRIAL automation is undergoing a technological revolution referred to as the fourth industrial revolution [1], [2]. The first revolution was the mechanization of production enabled by steam and water power. The second revolution was the mass production enabled by electricity. The third revolution was automated production enabled by electronics and information technologies. The fourth revolution is smart production enabled by recent breakthroughs in intelligent robotics, sensors, big data, advanced materials, edge supercomputing, the Internet of Things, cyber–physical systems, and artificial intelligence (AI). These systems are currently being integrated by software into factories, power grids, transportation systems, buildings, homes, and consumer devices.

Automation engineering software (AES) integrates various sensors, actuators, and control with the purpose of automating real-world processes [3], [4]. AES development has several challenges that distinguish it from general-purpose software development. In this article, we focus on two of the most prominent challenges: 1) AES software development is done by automation engineers, not by software experts and 2) AES software interacts with the physical world by sampling sensors and writing outputs to actuators in well-defined time intervals. These challenges have important implications in AES engineering. On the one hand, it is time-consuming to develop the automation code. This is partly because the automation code is often not engineered for reusability. Thus, similar functionality is often developed from scratch. On the other hand, the interaction with the physical world requires automation engineers to understand the hardware configuration that defines how sensors, actuators, and other hardware are connected to the digital and analog inputs and outputs of the system. This often requires an iterative engineering approach between the hardware and the software since any change in the hardware (e.g., a change of a component) has an impact in the software (e.g., input/output mappings), and vice versa. The tight coupling between the hardware and the software produces longer development cycles in AES.

The lifecycle of industrial automation systems (IASs) is divided into two phases: engineering and runtime. Engineering

Manuscript received December 15, 2019; revised March 30, 2020; accepted June 9, 2020. Date of publication July 21, 2020; date of current version July 2, 2021. This article was recommended for publication by Editor B. Vogel-Heuser upon evaluation of the reviewers' comments. *(Arquimedes Canedo, Palash Goyal, and Di Huang contributed equally to this work.) (Corresponding author: Arquimedes Canedo.)*

Arquimedes Canedo, Amit Pandey, and Gustavo Quiros are with Siemens Corporate Technology, Princeton, NJ 08540 USA (e-mail: arquimedes.canedo@siemens.com).

Palash Goyal and Di Huang are with the Information Sciences Institute, University of Southern California (USC), Los Angeles, CA 90292 USA.

Color versions of one or more of the figures in this article are available online at https://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TASE.2020.3008055

refers to all activities that occur before the system is in operation. These engineering activities include hardware selection, hardware configuration, automation code development, testing, and simulation. Runtime, on the other hand, refers to all activities that occur during the system's operation. These runtime activities include control, signal processing, monitoring, prognostics, and so on. Applications of AI in industrial automation have been primarily focused on the runtime phase due to the availability of large volumes of data from sensors. For example, time-series forecasting algorithms have been very successful in signal processing [5], [6]. Planning and constraint satisfaction are used in controls and control code generation [7], [8]. Anomaly detection algorithms are becoming very popular in cyber-attack monitoring [9], [10]. Probabilistic graphical models and neural networks for prognostics and health management of complex cyber–physical systems [11] have been deployed for systems, such as wind [12] and gas turbines [13].

The use of machine learning in the engineering phase, on the other hand, has remained relatively unexplored. There may be several reasons for this. First, engineering data are very scarce because of its proprietary nature [14]. Second, the duration of the engineering phase is short compared with the runtime phase; some IASs are in operation for more than 30 years. Therefore, the engineering phase is often considered less critical than the runtime phase. Third, acquiring human intent and knowledge is difficult. Capturing engineering know-how in expert systems is time consuming and expensive.

This article introduces the use of machine learning methods in AES to address three key tasks: code classification, semantic code search, and hardware recommendation. First, we demonstrate code classification on two real AES data sets. We learn representation of AES code via document embedding methods, using different artifacts, such as function calls, includes, comments, tags, and the code itself. Then, we train classifiers on the code embeddings to categorize code projects. Our results show that our approach captures code structure, and it is comparable to human annotation prediction performance. Second, using the resulting code embeddings, we demonstrate a semantic code search capability for AES code capable of finding syntactic and structurally equivalent fragments of code. Third, we develop a hardware recommendation system to autocomplete partial hardware configurations. Our results show a $3\times$ higher precision than the baselines. The original contributions of this article are as follows.

1) The introduction of three AES tasks where AI has a big impact on potential: code classification, semantic code search, and hardware recommendation.
2) An unsupervised learning AES code embedding approach based on natural language processing suitable for code classification and semantic code search.
3) The comparison of two hardware recommendation approaches using Bayesian networks and autoencoders.
4) The evaluation of our AI models in two real AES data sets consisting of 2927 Arduino projects [15] and 683 programmable logic controller PLC) programs [16].
5) The ArduCode reference implementation in Python,[1] and the data sets for advancing the AI research in automation consisting of the following.
   a) AES source code and metadata.
   b) An expert evaluation of code structural and syntactic similarity for 50 code snippets.
   c) A manually curated silver standard for hardware recommendation systems with two levels of granularity.

This article is organized as follows. Section II frames ArduCode in terms of the state of the art in AES and recent developments in code learning. Section III gives an overview of the two types of AES systems: IASs and maker automation systems (MASs). Section IV presents the proposed ArduCode architecture and methodology. Section V evaluates ArduCode's three AES learning tasks. Section VI describes a proof-of-concept implementation of these tasks in an AES tool. Section VII describes future directions of AI in automation engineering. Section VIII provides the concluding remarks.

II. Preliminaries and Related Work

To the best of our knowledge, we are the first to investigate the use of machine learning in AES. However, there is a large body of work on AES. In this section, we motivate the use of machines to assist AES tasks and frame our contributions relative to the state of the art in AES and related fields.

Over the last few years, manufacturing is transforming itself from centralized mass production into a distributed lot size one production. One of a kind product, uniquely customized by customers, is being produced on demand. This shift is creating an unprecedented need for innovation in the engineering phase. Today, despite being short in duration (relative to the runtime phase), we have estimated that the engineering phase contributes with about 50% of the total cost of automation. Thus, using AI in the engineering phase is an important technological development for lowering the cost of production. In mass production, the engineering phase is done upfront (at the beginning of the lifecycle) and only once for a particular product. In distributed lot size one production, engineering is done in parallel with the runtime as the production system must be adapted to satisfy all the variability associated with one of a kind product. While flexible machines and autonomous production systems can help realize a lot size one production, the engineering phase will become intertwined with the runtime phase in the future.

A. Code Classification

As production demands change rapidly, there is a need to efficiently integrate new functionality into production. Today, AES engineers invest a significant amount of time creating functional libraries to organize code according to its functionality. Publicly available examples of such libraries are the Arduino Library [17], PLCOpen [18], and OSCAT [16]. On the other hand, the majority of automation code functions

[1] https://github.com/arducode-aes/arducode

are not neatly organized in libraries. In these cases, automation engineers rely on cloning code by copying and pasting [19]. Cloning code solves an immediate problem because it allows the software development to make progress, but it creates a long-term maintenance problem because engineers quickly forget what a code function does. To address this problem, Thaller *et al.* [19] present a text-based system to detect code clones for the IEC 61131-3 programs commonly used in PLC programming. Although they broadly identify clone classification as one of the tasks in clone analysis, their system seems to be primarily focused on identifying code clones. In general, the lack of code classification tools in the automation domain motivates our work. This article introduces the use of automated code classification to reduce the effort of creating and maintaining functional code libraries. AI-driven code classification can be used to organize code snippets according to their functionality. That is, code snippets can be automatically labeled according to what they do, e.g., signal processing, signal generation, robot motion control, or any organization-defined functionality. Code classification can be integrated into an engineering tool in such a way that as soon as a new function is released by an engineer, it is automatically classified into a category of a library. The engineer can be in the loop to confirm or correct the classification.

B. Semantic Code Search

Frequent reconfigurations of the production system demand a much higher degree of automation code assurance. This puts automation engineers under higher pressure to produce code that works as intended in much shorter engineering cycles. This problem can be broken down into two steps: 1) identifying potential errors and 2) coming up with an alternative solution to solve these errors. In the automation context, static analysis tools have been used to identify AES programming errors and defects. These defects are often referred to as code smells or technical debt indicators [20]. For example, Prähofer *et al.* [21] present a tool to detect issues in the IEC 61131-3 programs. Their approach uses pattern-matching on program structures; control-flow and data-flow analysis; and call graph and pointer analysis. Similarly, Biallas *et al.* [22] present Arcade. PLC, a framework for the verification and analysis of PLC code, combines model-checking and static analysis. Unfortunately, these tools are primarily focused on solving the first half of the problem. Therefore, this motivates the development of new approaches for assist automation engineers in identifying alternative implementations that can fix the problems found by the static analysis tools.

Broadly, recent advances in code learning have shown that semantic code search is viable using machine learning. Code learning can be divided into two categories [23]: 1) language-specific models and 2) language-independent models. Language-specific models use knowledge of the languages used in the code to generate low-dimensional representations. For example, code2vec [24] constructs abstract syntax tree from the code for Java language for the purpose of predicting a method's name from its content. It deconstructs the tree into several paths and learns code embedding by aggregating the representations of these paths. func2vec [25] uses control flow graphs to generate embeddings of functions in C language. They utilize such representations to detect function clones. Similarly, Deeprepair [26] uses a combination of word2vec on tokens and recursive encoder on abstract syntax tree for Java token embedding. They use the representation to automatically repair programs with bugs. Several other works, such as DeepFix [27], use language-specific code learning to identify bugs and programming errors in codes. DeepTyper [28] uses recurrent neural networks to perform type inference in dynamically typed languages, such as Javascript and Python. On the other hand, language-independent models focus on syntactic representation learning. For example, [29] utilizes word2vec directly on tokens from code to learn their representations. They show that their model can help predict software vulnerabilities. Reference [30] utilizes a similar approach to the task of automated program repair. Allamanis *et al.* [31] introduced a syntactic model based on log-bilinear contexts to generate new method names using these embeddings. Such models that do not use language syntax to learn code representations are less widely used compared with language-specific models and often do not perform as well. However, in this article, we show that our proposed language-independent model achieves high accuracy in automation engineering tasks.

C. Hardware Recommendation

Automation engineering is the task of integrating various hardware components in software to achieve a production goal. The selection of hardware components occurs early on in the engineering process [14]. There are two tasks associated with hardware configuration: 1) the selection of a specific hardware component (e.g., temperature sensor model A) and 2) the configuration of that hardware in terms of inputs and outputs for the automation software to interact with it. Depending on the complexity of the project, the selection of components may be done by the mechanical engineering department. However, the configuration of inputs and outputs is always done by the automation engineers. Also, note that the input and output configurations are tightly coupled with the hardware selection. An automation program written for given hardware is not guaranteed to work for another hardware selection. Therefore, any hardware configuration change triggers a reengineering process [32].

Today, AES engineers use an iterative process that is repeated several times because either the hardware was incomplete or the hardware selection was wrong. Thus, reducing these hardware configuration iterations motivates the need for hardware recommendation systems. The goal of hardware recommendation systems is to predict a full hardware configuration from a partial hardware configuration. Recommendation or recommender systems are widely deployed in a variety of areas, such as social media, video streaming, music streaming, news, dating, and consumer products [33]. Unfortunately, recommender systems in the context of automation have remained a relatively unexplored area. Hildebrandt *et al.* [14] presented RESCOM, a multirelational recommender system

for an industrial purchasing system. This system assists users in selecting the hardware for complex engineering solutions based on shopping basket statistical patterns and semantic information. The focus on purchasing makes RESCOM more suitable for solving the hardware selection subproblem. To the best of our knowledge, we are the first to propose hardware recommender system for solving the hardware configuration for inputs' and outputs' subproblems.

III. Automation Engineering Software Overview

There are two types of automation systems: IASs and maker automation systems. This section introduces both. Despite some key differences, the most important aspect in common is the underlying computational model to interact with the physical environment. Most automation systems work on a periodic task model. Every period, or cycle, is composed of three steps. The first step reads inputs from the hardware (e.g., rpm of a motor). The second step executes a task every T seconds. A task is similar to a thread and that executes a user-defined automation program composed of one or more functions. The third step writes outputs to the hardware (e.g., a control signal). These three steps realize the classic closed-loop control system configuration.

In addition to the computation and memory capabilities, an important measure to compare automation systems is the number of input/output (I/O) pins. The I/O capacity determines how many hardware devices can be wired to the automation system. I/O pins can be of type analog and digital. Automation systems can be interconnected to form industrial networks. Although communication can be done through the I/O pins, modern automation systems provide dedicated communication ports supporting Ethernet and industrial protocols, such as Profinet [34] and OPC UA [35].

AES programming is done through an integrated development environment (IDE) referred to as the engineering system. This engineering system provides tools to support the AES development, including programming language editors, hardware configurators, library managers, static analysis checkers, debuggers, compilers, and build systems. Improving the engineering systems with AI is the main focus of this article.

A. Industrial Automation Systems

These systems control industrial processes; many of these are safety-critical. IAS is real-time and must guarantee a response within a specified timing constraint (i.e., cycle). IAS is designed to operate for decades, without downtime, in very harsh environments of extreme temperatures, pressure, humidity, and vibration. Variations of these systems are developed by different manufacturers and are targeted to different industries. For example, discrete manufacturing use PLCs, process plants use distributed control systems (DCSs), and power systems use remote terminal units (RTUs).

Over the years, IAS has been programed using a variety of programming languages. Most of these programming languages were conceived for automation engineers, not software engineers. Some of these languages, e.g., ladder logic, adopted domain-specific and graphical notations that were used to design relay racks in manufacturing and process control. Thus, by giving a syntax that automation engineers were familiar with, they were able to adopt these languages and write the AES code themselves. Today, these programming languages are standardized by the IEC61131-3 standard [36]. Most vendors provide programming language interoperability, and an automation program can contain functions written in different languages. This provides high flexibility to the automation engineers.

B. Maker Automation Systems

The MAS market has been enabled by low-cost electronics, microcontrollers, sensors, and actuators. Makers, people who form a "do it yourself" (DIY) community and culture, have been the primary adopters of MAS. Arduino [15] is one of the most popular MAS. Its open-source hardware and software are used by thousands of students and hobbyists around the world to develop DIY automation projects in robotics, home automation, entertainment, and wearables [37].

MAS, today, is considered nonreal time. However, there are a few recent advances to bring real-time operating systems (RTOSs) to Arduino [38], [39]. Despite this important difference with respect to IAS, MAS boards come with integrated analog and digital I/O pins and follow the same computation paradigm used for automation systems. In the case of the Arduino IDE [40], automation programs are referred to as sketches. Sketches can be written in Arduino code files (INO), C, or C++. There is an extensive library of functions to work with hardware, manipulating data, and using control algorithms.

MAS code is written by hobbyists with very diverse backgrounds and skillsets. Comparing the quality of MAS and IAS code is not straightforward because they are subject to different programming environments. For example, Arduino's C/C++ environment provides the user with full control over memory addressing, whereas PLC environments constrain it. However, we noticed that code cloning [19] is a common pattern in the MAS community. The high availability of code for full projects makes it easy for hobbyists to copy and paste useful snippets. Many of these programmers give attribution to the original source through the use of comments in the code. A more in-depth analysis of MAS versus IAS code is an interesting future research direction.

IV. ArduCode: Automation Engineering Software Learning

In this section, we introduce our predictive framework for automation engineering (see Fig. 1). First, we provide our data collection methodology. Then, we describe our technical approach for each of the three automation engineering tasks: 1) code classification; 2) semantic code search; and 3) hardware recommendation.

A. Data Collection

To validate our approach, we collected two real data sets representative of MAS and IAS. The following sections summarize the two data sets.

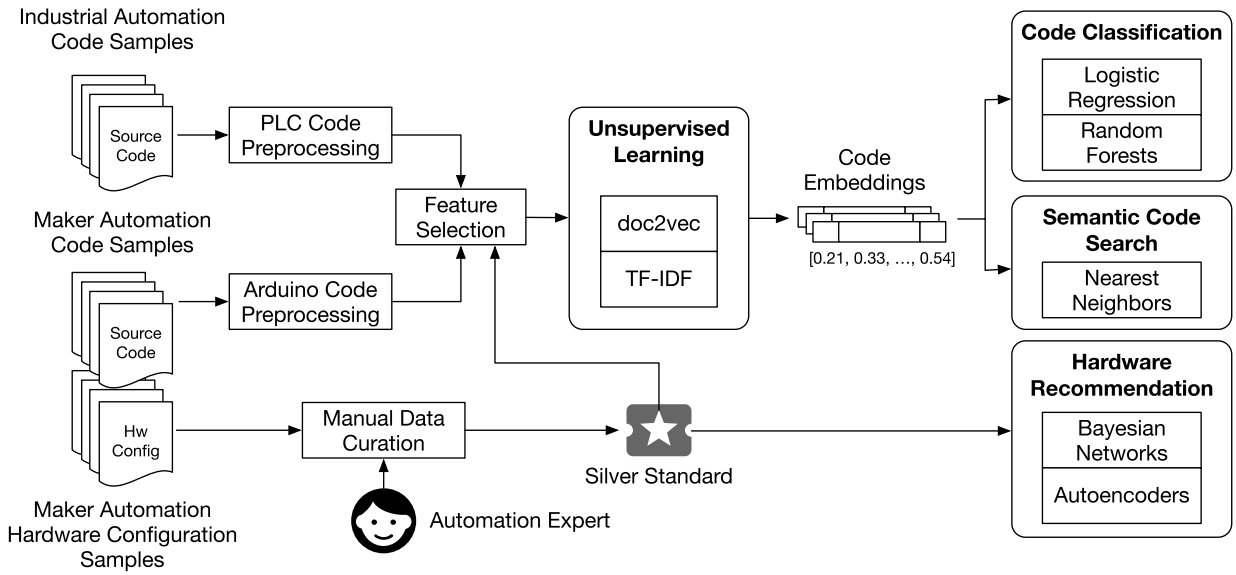

Fig. 1. AES code learning architecture.

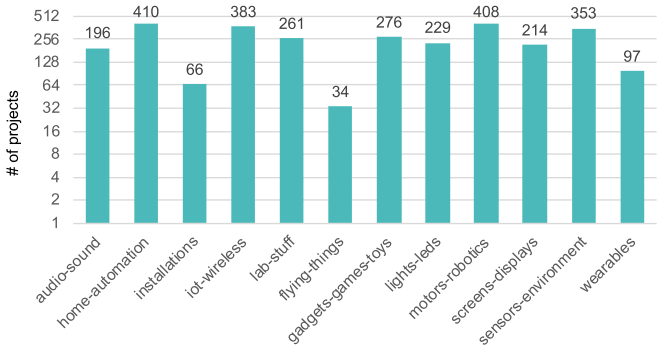

Fig. 2. Hand-curated level-1 Arduino project categories.

1) Arduino Code: We collected the source code and textual metadata from 2927 Arduino projects from the Arduino Project Hub [41]. The textual metadata consists of the project's category, title, abstract, tags, description, and hardware configuration. Each project is labeled by one category. In total, there are 12 categories, as shown in Fig. 2. We use these categories as the labels to predict in the code classification task. Makers are well known for helping and fostering collaboration in the DIY community. The documentation associated with the Arduino projects is extensive. Therefore, the project's title, abstract, tags, and description metadata provide an upper bound baseline for label classification using human annotations.

The hardware configuration is a list of components required to build the project. In the 2927 projects, there are 6500 unique components. After manual inspection, we observed that different authors name the same component differently, e.g., "resistor 10k" and "resistor 10k ohm." To clean the data, we manually curated the hardware configuration lists and renamed the 6500 components according to their functionality. An important contribution of this article is the definition of two functional levels of abstraction for the hardware: level-1 and level-2 functionalities. Our level-1 functionality consists of nine categories: actuators, Arduino, communications, electronics, human–machine interface, materials, memory, power, and sensors. Our level-2 functionality further refines the level-1 into a total of 45 categories: Actuators.{acoustic, air, flow, motor}, Arduino.{large, medium, other, small}, Communications.{ethernet, optical, radio, serial, wifi}, Electronics.{capacitor, diode, relay, resistor, transistor}, Human–Machine Interface.{button, display, input, led}, Materials.{adapter, board, screw, solder, wiring}, Memory.{solid}, Power.{battery, regulator, shifter, supply, transformer}, and Sensors.{accel, acoustic, camera, encoder, fluid, gps, misc, optical, photo, pv, rfid, temp}. We use these two levels of hardware configuration to validate our hardware recommendation algorithm.

2) PLC Code: The OSCAT library [16] is the largest publicly available library of PLC programs. The OSCAT-LIB is vendor-independent, and it provides reusable code functions in different categories such as signal processing (SIGPRO), geometry calculations (GEOMETRY), and string manipulation (STRINGS). These categories are extracted from the comment's section of each file marked by the line "FAMILY: X," where "X" is the category associated with that function. This line is eliminated from the data set during training. In total, the OSCAT-LIB basic version 3.21 contains 683 functions and 28 category labels. Fig. 3 shows the label distribution. The OSCAT-LIB does not contain hardware configuration, and therefore, it is only suitable for the tasks of code classification and semantic code search. All the code is written in the SCL language.

B. Code Classification

Given a code snippet, the task of code classification is to predict its label. A label is a category associated with the code snippet, such as the level-1 and level-2 Arduino project categories, or the OSCAT library function categories. Our machine learning pipeline consists of four steps: preprocessing, feature selection, code embeddings, and classification.

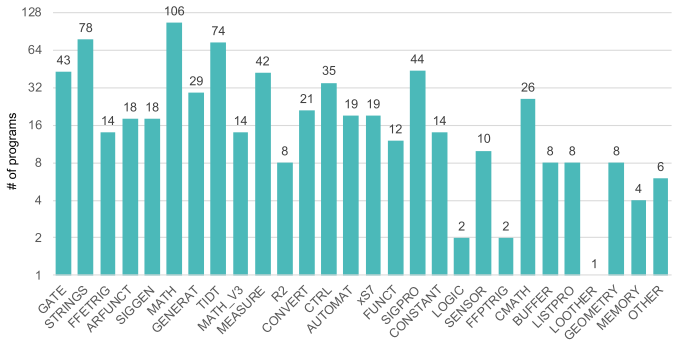

Fig. 3. PLC function block categories.

TABLE I
FEATURES EXPOSED BY PREPROCESSING

Feature	Arduino	PLC	Description
Includes	✓	–	C/C++ includes
Functions	✓	✓	Function names
Comments	✓	✓	Comments in code
Tokens	✓	✓	All code tokens
Code	✓	✓	Code keywords
LOC	✓	✓	Lines of code
Tags	✓	–	Project tags
Title	✓	–	Project title
Descriptions	✓	–	Project descriptions
Labels	✓	✓	Labels to predict
Components	✓	–	Hardware configuration

1) Preprocessing: We preprocess the automation projects and code snippets to expose the various features shown in Table I. The Arduino data set contains more features than the PLC data set. Therefore, not all features are available in the PLC data set. For example, the PLC code does not have includes, and project data, such as tags, title, descriptions, and components, are not available.

2) Feature Selection: The purpose of feature selection is to provide the ArduCode framework with a feature space exploration mechanism to compare the performance of different code representations in the task of code classification and semantic code search. The quality of the code embeddings is expected to vary according to the provided features. Therefore, the feature selection generates different experiments by combining different sets of features. For example, code can be represented by different combinations of includes, functions, comments, tokens, and keywords. Code documentation can be represented by combinations of tags, titles, and descriptions. Alternatively, code representations and code documentation features can be combined to generate richer feature vectors for the code embeddings.

3) Code Embeddings: In machine learning, the projection of a high-dimensional input into a lower dimensional representation space is called an embedding. The next step is to embed the textual representations generated by the feature selection into a learned vector representation suitable for classification. We compare the performance of the embeddings generated by gensim doc2vec [42] with the embeddings generated by the term frequency-inverse document frequency (tf-idf). Doc2vec is a state-of-the-art method for distributed representations of sentences and documents, i.e., tokens and code. Tf-idf, on the other hand, is a statistical measure that evaluates the relevance of a word is to a document, i.e., tokens and code. Tf-idf is typically used as a baseline for validating other machine learning models. Both doc2vec and tf-idf generate an n-dimensional vector representation of the input. The doc2vec's hyperparameters of interest are the embedding dimension and the training algorithm (distributed memory and distributed bag of words). We run all our experiments with a negative sampling of 5; this draws five noise words to help the model differentiate the data from noise.

4) Classification: The final step is to train a supervised model for code classification using the code embeddings as the input samples and the code labels as the target values. We compare the performance of logistic regression (LR) and random forest classifiers using the F_1-score metric. LR and random forest are two of the main workhorses for supervised learning classification. The F_1-score is the harmonic mean of the precision and recall. An F_1-score $= 1$ represents perfect precision and recall.

C. Semantic Code Search

Given a code snippet, the goal of a semantic code search is to find similar programs. For automation engineering, the similarity is defined in terms of syntax and structure. Syntax similarity helps engineers find useful functions in a given context, and structural similarity informs engineers on how other automation solutions have been engineered.

As shown in Fig. 1, semantic code search uses the code embeddings generated by doc2vec to identify similar documents. Doc2vec attempts to bring similar documents close to each other in the embedding space. For a given code embedding of a code snippet, the nearest neighbors are expected to be similar, and this distance metric is used as the basis for our semantic code search. While this approach is intuitive for syntactically similar documents, it is unclear whether the functional structure is captured in the embeddings.

D. Hardware Recommendation

Given a partial list of hardware components, the task of the hardware recommendation is to give a prediction of other hardware components typically used in combination with the partial list. The hand-curated silver standard described earlier is used to learn the joint probability distribution of the hardware components. The hardware recommendation task is then to compute the conditional probability of missing hardware components, given a partial list of components.

We compare two approaches for the hardware recommendation task. Our baseline consists of the predictions given on random hardware configurations. First, we learn a Bayesian network where the random variables are the hardware component categories. Bayesian networks are a probabilistic graphical model that represents a set of variables (e.g., hardware components) and their conditional dependencies (e.g., 80% chance that a sensor of type X is connected to

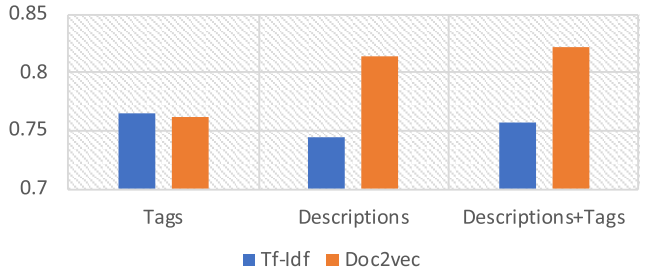

Fig. 4. Human annotation prediction performance.

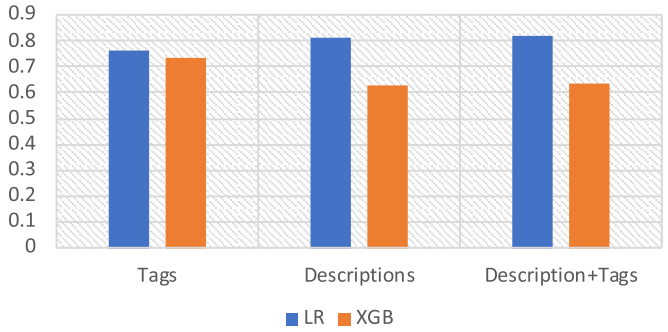

Fig. 5. Performance variation with respect to the classifier.

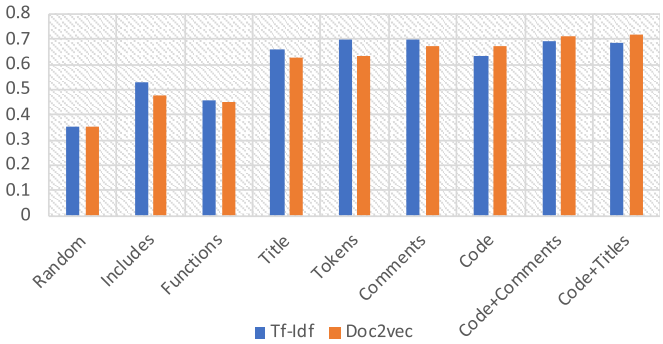

Fig. 6. Code label prediction using code.

a controller of type Y) via a directed acyclic graph. We use the Pomegranate Python package [43] to learn the structure of the Bayesian network and fit the model with 70% of the hardware configuration data. The Bayesian network for level-1 components consists of nine nodes, and the one for level-2 components consists of 45 nodes. However, we were only able to fit the level-1 Bayesian network as the initialization of the Bayesian network takes exponential time with the number of variables. Typically, this cannot be done with more than two dozen variables due to the superexponential time complexity with respect to the number of variables, and the level-2 hardware configuration consists of 45 variables. To overcome this limitation, we use an autoencoder implemented in Keras [44]. An autoencoder neural network is an unsupervised learning algorithm that tries to learn a function to reconstruct its input. In our autoencoder, the encoder learns a lower dimensional representation of the hardware configuration data, and the decoder learns to reconstruct the original input from the lower dimensional representation. Overfitting is when a statistical model is tailored to a data set and is unable to generalize to other data sets. To avoid overfitting in the autoencoder, we use L1 and L2 regularizers.

V. Results

This section evaluates the performance of the three AES tasks in ArduCode: code classification, semantic code search, and hardware recommendation. Our results establish the baselines for these tasks in the AES domain.

A. Code Classification

First, we established the lower and upper bounds for code label classification. The lower bound is given by training the code label classifier using random embeddings. The upper bound is given by training the code label classifier using human annotations. The Arduino data set provides human annotations in the form of tags and descriptions that can be combined in three configurations: tags, descriptions, and descriptions+tags. We first embed these three configurations using tf-idf and doc2vec and compare the label classification performance using the F_1-score. As shown in Fig. 4, doc2vec yields better performance than tf-idf. The embedding dimension for doc2vec was set to 50, and the tf-idf models generated embedding dimensions of 1469 for tags, 66 310 for descriptions, and 66 634 for descriptions+tags. Intuitively, the descriptions+tags configuration provides the upper bound F_1-score of 0.8213.

Fig. 5 compares the performance of a LR classifier from scikit-learn [45] and a random forest classifier (XGB) from XGBoost [46] using the 50-D doc2vec embeddings of tags, descriptions, and descriptions+tags. Our implementation in these libraries was driven purely by the convenience of their software interface and their popularity in the machine learning community. An implementation using alternative libraries should yield similar results to the ones obtained in our implementation. Our results show that the LR classifier provides significantly better performance than the XGB.

We establish the lower bound by generating 50-D random embeddings and predicting the labels using the LR classifier. Fig. 6 shows that with both tf-idf and doc2vec, the lower bound is 0.3538. After establishing the upper and lower bounds, we use different code features to predict labels. Fig. 6 shows that embedding includes and functions provide slightly better performance than the random baseline due to the very limited amount of information contained in these: 1.82 includes and 4.70 functions on average. Other code features improve classification accuracy significantly. For example, tokens and code are similar representations and give a similar F_1-score of 0.63 and 0.67. These results also show that comments contain valuable information that can be used to predict the code label with a score of 0.67. Embedding code+comments and code+titles yield the highest F_1-scores of 0.71. These results show that the prediction performance with code feature embeddings is comparable to human annotation embeddings with improvements of 2.03× and 2.32× over the random baseline, respectively.

The classifier's confusion matrix for code feature embeddings using doc2vec is shown in Fig. 7. Although the matrix

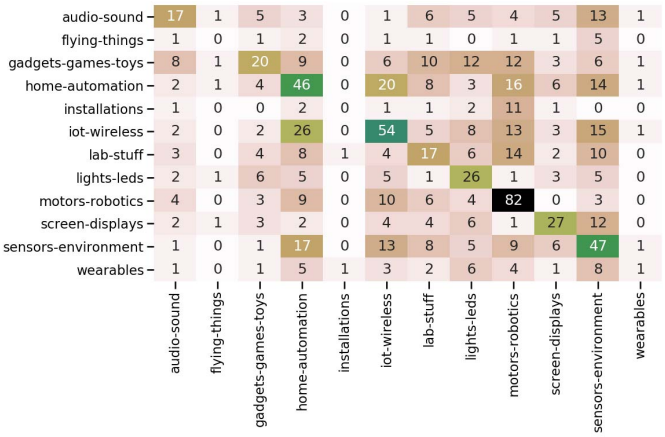

Fig. 7. ArduCode classifier's confusion matrix.

is diagonally dominant, the data set is imbalanced in terms of the number of samples per class, as shown in Fig. 2. In particular, the classifier's poor performance in the categories flying-things, installations, and wearables corresponds to their small number of samples in the data set.

Since the PLC data set does not have any human annotation features, we can only compare the performance of code feature embeddings (against the random baseline. The F_1-score for the code embeddings is 0.9024 and, for the random baseline, 0.2878, a 3.13× improvement. Compared with the Arduino data set, the PLC data set has fewer samples (683 versus 2927), more category labels (28 versus 12), and fewer lines of code per file on average (55 versus 177). These are the three factors that influence the higher prediction accuracy of ArduCode on the PLC data set.

B. Semantic Code Search

To validate the quality of our code embeddings, we randomly sampled 50 Arduino code snippets and tasked a group of six software engineers to score the similarity of each code snippet to its top-three nearest neighbors. For every code snippet pair, two similarity ratings for code syntax and code structure are given. A rating of 1 represents similarity, and a rating of 0 represents the lack of similarity. Code syntax refers to the use of similar variables and function names. Code structure refers to the use of similar code arrangements, such as if-then-else and for loops. In addition, every rating has an associated expert's confidence score from 1 (lowest confidence) to 5 (highest confidence) that represent the expert's self-assurance during the evaluation. During the expert evaluation, we eliminated five out of 50 samples where one of the top-three nearest neighbors was either an empty file, or it contained code in a different programming language. We found three code snippets written in Python and Javascript.

Table II shows the average code syntax and code structure similarity scores given by experts. We only report the high confidence ratings (avg. confidence ≥ 4.5) in order to eliminate the influence of uncertain answers. We also measure the experts' agreement via the Fleiss Kappa. These results show that the similarity scores for both syntax and structure are high for the top-one neighbors (0.68 and 0.61, respectively) but

TABLE II

AVERAGE CODE STRUCTURE AND CODE SYNTAX SIMILARITY AND FLEISS KAPPA VALUES FOR HIGH CONFIDENCE RATERS

	Nearest neighbors		
Similarity	**Top 1 (κ)**	**Top 2 (κ)**	**Top 3 (κ)**
Syntax	0.61 (0.75)	0.48 (0.70)	0.32 (0.61)
Structure	0.68 (0.53)	0.40 (0.44)	0.33 (0.66)

TABLE III

CODE EMBEDDING COSINE SIMILARITY FOR SIMILAR AND NOT SIMILAR CODE SNIPPETS

	Nearest neighbors		
	Top 1	**Top 2**	**Top 3**
Similar code snippets			
#2696	0.8768	0.7527	0.7642
#547	0.8719	0.8705	0.8506
#2815	0.9465	0.9445	0.9126
#54	0.8513	0.8056	0.7815
Not Similar code snippets			
#4512	0.5967	0.5497	0.5643
#4345	0.5415	0.4175	0.5192
#1730	0.5970	0.5035	0.5511

reduce significantly (under 0.50) for the top-two and top-three neighbors. The experts are in substantial agreement ($0.61 \leq \kappa \leq 0.80$) in their syntax similarity scores and in moderate agreement ($0.41 \leq \kappa \leq 0.60$) in their structure similarity scores. While these results confirm that doc2vec code embeddings capture syntactic similarity, they also show that some structure similarity is captured in the top-one neighbor. After their individual assessment, the experts gathered as a group to discuss their findings. Something that quickly became clear is that the experts' background contributed to the deviation in the ratings. Half of the experts with an automation background had additional insights that made them more confident and congruent in their structural similarity assessments. On the other hand, the other half of the experts without an automation background were less confident and congruent in their ratings. With additional context from the automation domain, two of the three nonautomation experts expressed that this information would have made their assessment more confident and congruent.

To further gain insight into our experiment, we selected four similar and three not similar code snippets and measured the cosine similarity of their embeddings, as shown in Table III. The selected code snippets have a strong agreement among the experts and high confidence in the similarity and lack of similarity across the top-three nearest neighbors. These results confirm that the code snippets considered very similar by the experts are close to each other in the embedding space. On the other hand, code snippets considered not similar are far apart in the embedding space.

Fig. 8 shows two similar Arduino code snippets (#54 and #2689) produced by ArduCode. There are similarities between these two programs at different levels. First, all Arduino programs are required to have the `setup()` and `loop()` functions to initialize the program and to specify the

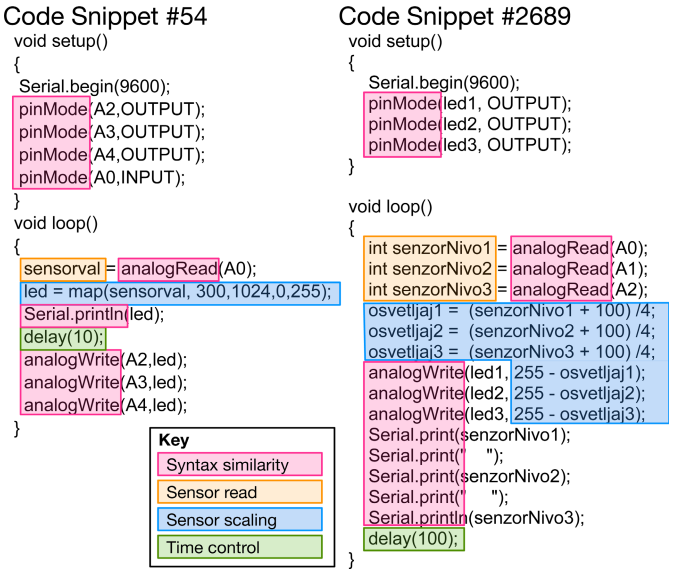

Fig. 8. Arduino semantic code search result.

control logic executed on every cycle. Syntactically, the two programs use the same standard functions: `pinMode()` to configure the Arduino board pins where the hardware connects as inputs or outputs; `analogRead()` to read an analog value from a pin; `Serial.print()` to print an ASCII character via the serial port; `delay()` to pause the program for the amount of time (in ms) specified by the parameter; and `analogWrite()` to write an analog value to a pin. Semantically, the two programs read sensor values (only one value in #54 and three values in #2689) scale the sensor value to a range (from 300–1024 to 0–255 using `map()` in #54 and to $(x+100)/4$ in #2689), print the scaled sensor value via the serial port, write the analog value to an LED (a single LED in #54 and three LEDs in #2689), and pause the program (10 ms in #54 and 100 ms in #2689). Note that the order in which these operations are scheduled is different in the two programs. Functionally, the two programs perform the same task of creating a heat map for a sensor value using LEDs, while there are some syntactic similarities, or semantic code search is also able to capture semantic and structural similarities.

C. Hardware Recommendation

In the hardware recommendation, we are interested in recommending the top-k hardware components. Therefore, we evaluate our models in terms of $precision@k$. $Precision@k$ is the portion of recommended hardware components in the top-k set that are relevant. For each hardware configuration in the test data, we leave one hardware component out and measure its $precision@k$. Table IV shows the results for the random baseline, the Bayesian network, and the autoencoder. As expected, the performance of the random baseline improves linearly from $p@1 = 0.1$, $p@3 = 0.32$, and $p@5 = 0.54$ to $p@9 = 1$ for the level-1 hardware predictions. The Bayesian network also improves linearly from $p@1 = 0.32$, $p@3 = 0.59$, and $p@5 = 0.79$.

TABLE IV
$p@k$ RESULTS FOR LEVEL-1 HARDWARE PREDICTIONS

$p@k$	Random Baseline	Bayesian Network	Autoencoder
$p@1$	0.10	0.32	0.36
$p@3$	0.32	0.59	0.79
$p@5$	0.54	0.79	0.95
$p@9$	1.00	1.00	1.00

TABLE V
$p@k$ RESULTS FOR LEVEL-2 HARDWARE PREDICTIONS

$p@k$	Random Baseline	Autoencoder
$p@1$	0.02	0.21
$p@3$	0.06	0.34
$p@5$	0.11	0.45
$p@10$	0.21	0.69

The autoencoder provides both the best performance and the best improvements from $p@1 = 0.36$, $p@3 = 0.79$, and $p@5 = 0.95$. Note that the autoencoder's $p@3$ is the same performance as the Bayesian Network's $p@5$ and 0.79. Furthermore, the autoencoder achieves >0.95 precision at $p@5$ and the Bayesian network at $p@8$.

Learning a Bayesian Network for level-2 hardware components is computationally unfeasible. Therefore, we rely on an autoencoder to accomplish this task, and the $p@k$ results are reported in Table V. The overall $p@k$ performance of the autoencoder for level-2 is comparatively lower than for level-1. The reason is that the level-2 hardware configuration is sparser than level-1. On average, level-2 configurations have 4/40 components, and level-1 configurations have 4/12 components. However, the improvement over the random baseline is of $10\times$ for $p@1$, $5\times$ for $p@3$, $4\times$ for $p@5$, and $3\times$ for $p@10$.

VI. PROOF-OF-CONCEPT IMPLEMENTATION—COGNITIVE AUTOMATION ENGINEERING SYSTEM

This section presents our experience in developing a proof-of-concept system that attempts to transition this research into an application. Our goal is to explore concrete implementation ideas to assist the AES engineer during the development process in such a way that the AI-driven methods do not disrupt the original AES workflows.

Fig. 9 shows our implementation. The system is centered on an engineering knowledge base, which is built in two complementary ways: by analyzing past engineering projects and by direct input and curation of engineering know-how by experts. We leverage semantic web technologies and represent the engineering knowledge in a Resource Descriptor Framework (RDF) triple store. Engineers with sufficient expertise can edit the graph directly, and we aim to develop interfaces that simplify the formalization of engineering knowledge by domain experts. Note that the past engineering projects and expert know-how are specific to the organization where the system would be deployed, and therefore, the ArduCode's models would reflect that prior expertise. Leveraging data from past projects is a very powerful approach because it

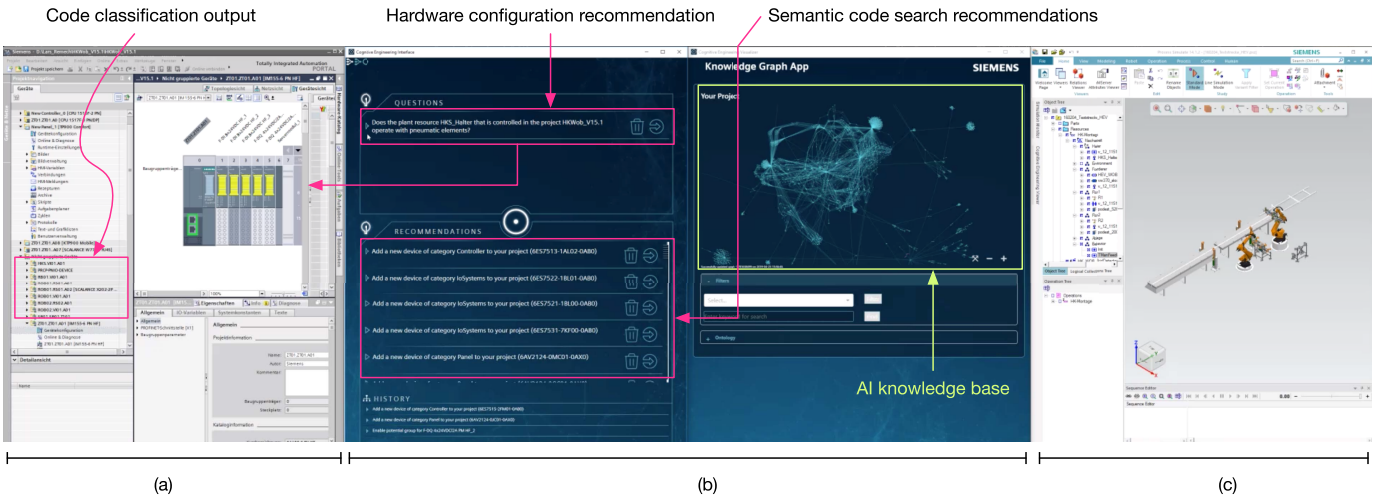

Fig. 9. Cognitive automation engineering system. (a) AES engineering system platform. (b) Cognitive automation engineering system. (c) Simulation and validation.

allows the integration of existing libraries and unstructured repositories of functions into ArduCode's learning process. Thus, code classification, semantic code search, and hardware recommendation can be continuously improved over time as more data becomes available from the regular engineering process. On the other hand, the direct expert input allows the resolution of issues when ArduCode's confidence is low in a prediction, and the direct expert curation can enable the improvement of ArduCode's models over time.

We use an existing AES engineering system as a platform [see Fig. 9(a)]. This approach has multiple advantages. First, an existing AES engineering system provides well-known graphical user interfaces and workflows that engineers are familiar with. Second, we can leverage the APIs and open interfaces from the AES engineering system to push and pull data from the current and other automation projects. Pulling data is critical for building a robust knowledge base for the AI methods to learn from. Pushing data is equally critical because it allows the AI methods to assist the engineer within the well-established workflows and interfaces. Third, it provides a development environment for plug-ins where novel concepts for AI-assistant user interfaces can be developed and tested with real users.

The proof-of-concept user interface plug-in that we developed is shown in Fig. 9(b). The goal of this user interface is to provide the AES engineer an assistant that is continuously monitoring the state of the AES project and providing recommendations from code classification, semantic code search, and hardware recommendation. The assistant system reads the user's automation project containing hardware configuration and program code. Multiple analysis components are then executed on the data and generate recommendations based on this available information. As the user edits configurations or automation code, the code classifier, semantic code search, and hardware recommendation are invoked, and the user interface collects the results of the analyses and presents them to the user in an organized manner based on the context of the corresponding engineering task. In this user interface, the main interaction occurs through questions and recommendations. The AI-assistant presents questions to the user, which the user can answer providing new information to the system. Questions to the user are triggered by information known to be missing in the knowledge model. For instance, if an analysis depends on the value of an attribute of the project (e.g., safety integrity level), and this attribute is missing in the current project, then the assistant will issue a question to the user asking for the value of this attribute (e.g., What is the safety integrity level of this project?). Based on the information gathered from the engineering tool and from user answers, the assistant generates recommendations that the user either accepts or rejects. When the user provides the project's SIL level as an answer, the system may trigger additional analyses using this information (e.g., determine hardware to recommend for the required safety level, classify the code in the automation project to separate the safety and nonsafety code, and reorganize the project structure accordingly, or find library code suitable for the safety application and recommend this code to the user). When the user accepts a recommendation, an action is taken, and the project is modified accordingly. When the user rejects or ignores a recommendation, the system simply continues its operation. Note that these interactions are also useful for refining the AI models. These interactions can be used in the future as data for a lifelong learning system that self-improves over time. To keep the user informed on what the AI-methods do, we provide an interface to the AI knowledge base in a graphical form. We use this view to highlight the nodes and edges related to a given recommendation.

Just like in the traditional AES development process, validation through simulation plays an important role. We show that the simulation and validation loop can be tightly coupled to the proposed cognitive automation engineering system. After a recommendation is accepted, and the change is pushed to the AES engineering system, this can be validated in simulation. Fig. 9(c) shows a production line simulation that the AES project controls. This implementation shows that AI-assistants can be nondisruptive to the existing AES development process.

The assistant system has been used for empirical evaluation of the approach in a lab setting. Our plan is to evaluate this tool in a real production environment.

VII. Discussion

ArduCode is the starting point for predictive automation engineering tasks. This section discusses its known limitations and motivates several directions for future work.

A. Capturing Code Structure

While doc2vec captures some structural code similarity as confirmed by our set of experts, there are other recent approaches, such as code2vec, that are likely to better capture code structure and improve the code classification and semantic code search tasks. However, parsing C++ code requires full access to all the library dependencies. In the case of most Arduino programs, resolving the include paths requires a major manual effort. In the future, we expect to develop some automation to generate abstract syntax trees and embed the code using models, such as code2vec.

B. Hardware–Software Gap

ArduCode's hardware recommendation is limited to hardware components. This task would be even more useful if it incorporated software elements, such as library or API recommendations. We obtained poor results with a supervised learning approach using hardware configuration as the input samples and includes and function names as the target values. Without contexts such as descriptions or titles, this is a hard task even for experts because the software references hardware only through nondescriptive variables (e.g., A0 and A1). To bridge this gap, one promising idea is to model software elements as random variables in the Bayesian network and use expert know-how to define their conditional probabilities. Another potential direction is to take into consideration wiring diagrams that describe how the hardware components are connected to the controller and, based on this information, determine the connection to the software. However, these diagrams are often not synchronized to the software view and could introduce many inconsistencies. A second challenge is that these diagrams are often created using *ad hoc* methods by the software developers, and their syntax and semantics are not consistent.

C. Continuous Integration of New Knowledge

In the current architecture, ArduCode's model is updated via retraining. This means that new knowledge is not automatically integrated into ArduCode's knowledge base. The new knowledge is manually integrated through a retraining step. Retraining takes time and requires a machine learning expert. However, with the high availability of cloud computing and graphic processing units (GPUs), we do not anticipate retraining being a bottleneck for the automation domain. An interesting direction for future work is to extend ArduCode into a lifelong learning architecture [47]. Lifelong learning is the ability of a machine learning system to sequentially retain learned knowledge and to transfer that knowledge over time when learning new tasks and improve its capabilities. Such an approach would continuously integrate new knowledge as it becomes available, instead of being limited to discrete retraining steps.

VIII. Conclusion

In this article, we introduced and studied three automation engineering predictive tasks. First, we showed that our code classification approach based on doc2vec code embeddings and LR achieves F_1-scores of 72% and 90% on two real data sets. Second, a group of six experts validated the semantic code search task by assessing the syntax and structure similarity of 50 code snippets. Third, we demonstrated $p@3$ of 79% and $p@5$ of 95% for the hardware recommendation task using an autoencoder. In addition, we implemented these tasks in a proof-of-concept implementation of a cognitive automation system. This system has been used for empirical evaluation of ArduCode in a laboratory setting.

Future research directions are as follows. Evaluate ArduCode's doc2vec approach against recent approaches, such as code2vec, that are likely to better capture code structure and improve the code classification and semantic code search tasks. In addition, ArduCode's hardware recommendation is limited to hardware components. This task would be even more useful if it incorporated software elements, such as library or API recommendations. One promising idea is to model software elements as random variables in the Bayesian network and use data mining techniques on existing projects to define their conditional probabilities.

Acknowledgment

The authors would like to thank Evan Patterson, Jade Master, and Georg Muenzel for their valuable input and discussion.

References

[1] M. Hermann, T. Pentek, and B. Otto, "Design principles for industrie 4.0 scenarios," in *Proc. 49th Hawaii Int. Conf. Syst. Sci. (HICSS)*, Jan. 2016, pp. 3928–3937.
[2] R. Drath and A. Horch, "Industrie 4.0: Hit or hype? [Industry Forum]," *IEEE Ind. Electron. Mag.*, vol. 8, no. 2, pp. 56–58, Jun. 2014.
[3] V. Vyatkin, "Software engineering in industrial automation: State-of-the-art review," *IEEE Trans. Ind. Informat.*, vol. 9, no. 3, pp. 1234–1249, Aug. 2013.
[4] B. Vogel-Heuser *et al.*, "Challenges for software engineering in automation," *J. Softw. Eng. Appl.*, vol. 7, no. 5, pp. 440–451, 2014.
[5] S.-J. Huang and K.-R. Shih, "Short-term load forecasting via ARMA model identification including non-Gaussian process considerations," *IEEE Trans. Power Syst.*, vol. 18, no. 2, pp. 673–679, May 2003.
[6] D. Tulone and S. Madden, "PAQ: Time series forecasting for approximate query answering in sensor networks," in *Proc. Eur. Workshop Wireless Sensor Netw.*, 2006, pp. 21–37.
[7] M. S. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2006.
[8] M. Dogar, A. Spielberg, S. Baker, and D. Rus, "Multi-robot grasp planning for sequential assembly operations," *Auto. Robots*, vol. 43, pp. 649–664, Mar. 2019.
[9] C. Feng, T. Li, and D. Chana, "Multi-level anomaly detection in industrial control systems via package signatures and LSTM networks," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2017, pp. 261–272.
[10] C.-W. Ten, J. Hong, and C.-C. Liu, "Anomaly detection for cybersecurity of the substations," *IEEE Trans. Smart Grid*, vol. 2, no. 4, pp. 865–873, Dec. 2011.

[11] K. R. McNaught and A. Zagorecki, "Using dynamic Bayesian networks for prognostic modelling to inform maintenance decision making," in *Proc. IEEE Int. Conf. Ind. Eng. Eng. Manage.*, Dec. 2009, pp. 1155–1159.

[12] G. D. N. P. Leite, A. M. Araújo, and P. A. C. Rosas, "Prognostic techniques applied to maintenance of wind turbines: A concise and specific review," *Renew. Sustain. Energy Rev.*, vol. 81, pp. 1917–1925, Jan. 2018.

[13] Y. G. Li and P. Nilkitsaranont, "Gas turbine performance prognostic for condition-based maintenance," *Appl. Energy*, vol. 86, no. 10, pp. 2152–2161, Oct. 2009.

[14] M. Hildebrandt, S. S. Sunder, S. Mogoreanu, I. Thon, V. Tresp, and T. Runkler, "Configuration of industrial automation solutions using multi-relational recommender systems," in *Machine Learning and Knowledge Discovery in Databases*. Cham, Switzerland: Springer, 2019, pp. 271–287. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-10997-4_17

[15] Y. A. Badamasi, "The working principle of an Arduino," in *Proc. 11th Int. Conf. Electron., Comput. Comput. (ICECCO)*, Sep. 2014, pp. 1–4.

[16] OSCAT. *Open Source Community for Automation Technology*. Accessed: 2020. [Online]. Available: http://www.oscat.de/

[17] Arduino. *Arduino Libraries*. Accessed: 2020. [Online]. Available: https://www.arduino.cc/en/reference/libraries

[18] (2020). *PLCOpen*. [Online]. Available: https://plcopen.org/

[19] H. Thaller, R. Ramler, J. Pichler, and A. Egyed, "Exploring code clones in programmable logic controller software," in *Proc. 22nd IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2017, pp. 1–8.

[20] S. Bougouffa, Q. H. Dong, S. Diehm, F. Gemein, and B. Vogel-Heuser, "Technical debt indication in PLC code for automated production systems: Introducing a domain specific static code analysis tool," *IFAC-PapersOnLine*, vol. 51, no. 10, pp. 70–75, 2018.

[21] H. Prähofer, F. Angerer, R. Ramler, and F. Grillenberger, "Static code analysis of IEC 61131-3 programs: Comprehensive tool support and experiences from large-scale industrial application," *IEEE Trans. Ind. Informat.*, vol. 13, no. 1, pp. 37–47, Feb. 2017.

[22] S. Biallas, S. Kowalewski, S. Stattelmann, and B. Schlich. *Static Analysis of Industrial Controller Code Using Arcade.PLC*. Accessed: 2014. [Online]. Available: https://cs.au.dk/~amoeller/tapas2014/tapas2014_1.pdf

[23] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," 2019, *arXiv:1904.03061*. [Online]. Available: http://arxiv.org/abs/1904.03061

[24] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," in *Proc. Program. Lang. (ACM)*, vol. 3, Jan. 2019, pp. 1–29.

[25] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to error-handling specification mining," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2018, pp. 423–433.

[26] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2019, pp. 479–490.

[27] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common C language errors by deep learning," in *Proc. AAAI Conf. Artif. Intell.* San Francisco, CA, USA: AAAI, 2017, pp. 1345–1351. [Online]. Available: https://dl.acm.org/doi/10.5555/3298239.3298436

[28] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 1345–1351.

[29] J. A. Harer *et al.*, "Automated software vulnerability detection with machine learning," 2018, *arXiv:1803.04497*. [Online]. Available: http://arxiv.org/abs/1803.04497

[30] Z. Chen and M. Monperrus, "The remarkable role of similarity in redundancy-based program repair," 2018, *arXiv:1811.05703*. [Online]. Available: http://arxiv.org/abs/1811.05703

[31] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proc. 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, 2015, pp. 38–49.

[32] S. Feldmann *et al.*, "A comparison of inconsistency management approaches using a mechatronic manufacturing system design case study," in *Proc. IEEE Int. Conf. Autom. Sci. Eng. (CASE)*, Aug. 2015, pp. 158–165.

[33] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez, "Recommender systems survey," *Knowl.-Based Syst.*, vol. 46, pp. 109–132, Jul. 2013.

[34] J. Feld, "PROFINET-scalable factory communication for all applications," in *Proc. IEEE Int. Workshop Factory Commun. Syst.*, Sep. 2004, pp. 33–38.

[35] T. Hannelius, M. Salmenpera, and S. Kuikka, "Roadmap to adopting OPC UA," in *Proc. 6th IEEE Int. Conf. Ind. Informat.*, Jul. 2008, pp. 756–761.

[36] *Standard IEC61131-3*, Standard IEC 61131-3 2013, International Electrotechnical Commission (IEC), 2020.

[37] M. Banzi and M. Shiloh, *Getting Started With Arduino: The Open Source Electronics Prototyping Platform*, 3rd ed. Sebastopol, CA, USA: Maker Media, 2014.

[38] B. Greiman. (2019). *FreeRTOS-Arduino*. [Online]. Available: https://github.com/greiman/FreeRTOS-Arduino

[39] P. Buonocunto, A. Biondi, M. Pagani, M. Marinoni, and G. Buttazzo, "Arte: Arduino real-time extension for programming multitasking applications," in *Proc. 31st Annu. ACM Symp. Appl. Comput. (SAC)*, 2016, pp. 1724–1731.

[40] Arduino. (2020). *Arduino IDE*. [Online]. Available: https://www.arduino.cc/en/main/software

[41] Arduino. (2020). *Arduino Project Hub*. [Online]. Available: https://create.arduino.cc/projecthub

[42] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 1188–1196.

[43] (2020). *Pomegranate*. [Online]. Available: https://pomegranate.readthedocs.io/

[44] (2020). *Keras: The Python Deep Learning Library*. [Online]. Available: https://keras.io/

[45] (2020). *Scikit-Learn: Machine Learning in Python*. [Online]. Available: https://scikit-learn.org/

[46] (2020). *XGBoost: Scalable and Flexible Gradient Boosting*. [Online]. Available: https://xgboost.ai/

[47] D. L. Silver, Q. Yang, and L. Li, "Lifelong machine learning systems: Beyond learning algorithms," in *Proc. AAAI Spring Symp., Lifelong Mach. Learn.*, vols. 5–13, 2013, pp. 49–55.