A Machine Learning Based Approach to Application Landscape Documentation

Jörg Landthaler(✉), Ömer Uludağ, Gloria Bondel, Ahmed Elnaggar, Saasha Nair, and Florian Matthes

Software Engineering for Business Information Systems, Department of Informatics, Technical University of Munich, Boltzmannstr. 3, 85748 Garching bei München Germany
{joerg.landthaler, oemer.uludag, gloria.bondel, ahmed.elnaggar, saasha.nair, matthes}@tum.de

Abstract. In the era of digitalization, IT landscapes keep growing along with complexity and dependencies. This amplifies the need to determine the current elements of an IT landscape for the management and planning of IT landscapes as well as for failure analysis. The field of enterprise architecture documentation sought for more than a decade for solutions to minimize the manual effort to build enterprise architecture models or automation. We summarize the approaches presented in the last decade in a literature survey. Moreover, we present a novel, machine-learning based approach to detect and to identify applications in an IT landscape.

Keywords: Software Asset Management · EAM · Machine Learning.

1 Introduction

Traditional enterprise architecture management (EAM) uses enterprise architecture (EA) models to support enterprise analysis and planning, in particular in IT-intensive organizations. A standard EA model, e.g. based on the ArchiMate meta-model, comprehensively models many different aspects of an organization from roles via processes through to applications, software components, and IT infrastructure components. The creation of EA models is an error-prone, difficult and labor-intensive manual task [2, 10, 12]. The field of EA documentation (EAD) seeks to automate the creation of EA models [6]. However, the automated creation of EA models is a challenging task, because not all information is easily available such as the relationship between business processes and software components or due to required high-level semantic information [6].

With the advent of digitalization, IT-landscapes grow in size and complexity [13]. Moreover, elements in the EA become more and more interwoven even across organizational boundaries. As a consequence, another increasingly pressuring challenge is to manage organizational IT-landscapes at runtime, which requires to capture dynamic aspects such as failures of servers and errors in applications [1]. The question arises whether (automatically generated) EA models

can be used to support the operation of large IT-landscapes, for example, to support root cause analysis or business impact analysis. This requires a high-quality of the automatically generated EA models. Fast and automatically generated high-quality EA models could support the initial creation of (as-is) EA models. It could foster the comparison of manually created EA models with the actual EA. Moreover, the progression of the implementation of a target (to-be) EA state could be measured. In the last decade, different approaches to automate EAD have been proposed that vary in their degree of automation and coverage of EA meta-models like ArchiMate. Buschle et al. [2] proposed to use the manually collected information stored in an enterprise service bus and Holm et. al. [10] presented an EAD tool that uses information acquired by network sniffers.

The automated creation of full EA models is a major endeavor that incorporates the detection and identification of many elements of different types and their relationships. In this paper, we focus on the application component elements as defined in the ArchiMate meta-model (application components include applications). In particular, we propose an envisioned approach to automatically document the application landscape of standard software in an organization that is supported by machine learning techniques. Our approach is based on the classification of binary strings of the application executables that are present on a target machine.

Machine learning on application binary strings is used for example in antivirus software. Our goal is to identify applications, which results in challenges that are very interesting from a machine learning point of view. The machine learning problem has many classes and eventually only few examples. To our surprise, the binary strings of executables vary even for the same application on different devices (with same application version, device type and OS versions). While the problems for the machine learning approach are challenging, we expect the effort to manually create a similar knowledge base solely based on rules to achieve the same goal to be even larger.

Our key contributions for this exploratory paper encompass the presentation of an envisioned approach to EAD based on machine learning, a small literature survey of approaches to EAD proposed so far and the evaluation of the basic technical feasibility of our approach regarding the machine learning aspect using a dataset of applications collected from devices at our research group.

The remainder of this work is organized as follows: In Section 2 we present our envisioned approach. We present a literature review of published approaches to automated EAD in Section 3. The technical feasibility of our approach is evaluated in Section 4, followed by a critical reflection and limitations in Section 5. Section 6 concludes the paper with a summary and presents future work.

2 Approach

In this Section, we motivate and describe a machine learning based approach to identify standard software in (possibly large) IT-landscapes and integrate it into a larger picture. The identification of installed or running applications

in large IT-landscapes is a major challenge, because many equal, similar and different types of applications are spread over several hardware devices. It is not uncommon that an IT-landscape in a larger company contains several thousand applications. It is desirable to get an overview of all applications present in an IT-landscape to create an inventory (Software Asset Management [5] and License Management), to manage the operations of applications or as part of dynamically or continuously built EA models.

Machine learning, in general, helps to solve repetitive problems. It is applicable if inputs vary in the nature of their contents [4]. Particularly for supervised learning (i.e. classification), a sufficient amount of labeled training data is required. The problem of identifying applications in an IT-landscape is challenging, because of the sheer amount of applications and because of the many possible smaller differences among individual installations such as installation directories and configurations. Supervised machine learning is a promising approach because of the repetitive characteristics, the large manual effort of the problem at hand and the varying nature of features. However, in contrast to typical problems solved by supervised machine learning, the problem at hand is a very challenging task for classifiers, because of the large number of different applications, resulting in a classification problem with many classes. However, parts of our experiments in Section 4 are sufficiently promising to merit investigating this approach more deeply.

Existing approaches detect running applications in an indirect way from the outside (i.e. without placing an agent on the server) for example with port scanners or by investigating traces that applications leave behind, for example network communication. Another, conventional approach to detect applications that are not executed is to create a knowledge base of rules that enables an agent installed on a server to find all (relevant) installed applications. The knowledge base can be either shipped with the agent or stored centrally and queried by agents. There are two challenges to this approach: one is the diversity of application characteristics and the other is the large manual effort to create and maintain rules for hundreds of different applications, even if central registries are available. However, there are commercial providers that maintain such knowledge-bases, for example Flexera[1].

The major difference between our approach and a conventional, strictly rule-based approach is to place an agent on a server that identifies all executables (which can be done efficiently and effectively) and classifies the executables as different applications. In our envisioned approach the result of the classification of executable binaries helps to identify applications present on a device or server. We believe that a rule-based refinement of the results will still be necessary. To our surprise, we observed that sometimes applications differ greatly in their binary strings even for equal versions across different devices, which imposes an additional challenge to our approach and that we explore in Section 4. The benefit of our approach over the rule-based approach is that we identify all applications, even when they are renamed or installed in non-standard

[1] Flexera FlexNet, https://www.flexera.com, last accessed in November 2017

directories, which is often the case on servers. Our evaluation dataset does not incorporate renamed files but the evaluation is still valid because the machine learning based approaches input are merely the binary strings of the applications executable files.

There are several design decisions that one has to make for a real-world application. For example, if the agent consumes a service that provides the classification functionality or if the agent ships with the trained model (and eventually needs to be updated often). A service-based solution might interfere with data protection requirements, but application binaries usually do not hold information worth high protection.

3 Literature Study & Related Work

In this Section publications related to the research are reviewed and summarized. The commonalities and differences to our approach are summarized in Table 1. Farwick et al. [7] automatically integrate various runtime information of the cloud infrastructure into the open-source EAM tool Iteraplan. The automatically integrated information is synchronized with a project management tool to distinguish between planned and unplanned changes of the cloud infrastructure.

Holm et al. [10] aims to map automatically collected information with the network scanner NeXpose to ArchiMate models. The approach collects IT infrastructure and application data. Buschle et al. [2] have the goal to evaluate the degree of coverage to which data of a productive system can be used for EA documentation. In order to do so, the database schema of SAP PI is reverse engineered based on its data model and conceptually mapped to the ArchiMate model and the CySeMol and planningIT tools.

Hauder et al. [9] aim to identify challenges for automated enterprise architecture documentation. They map the data model of SAP PI and Nagios to Iteraplan in order to extend Iteraplan models for identifying transformation challenges [9].

The goal of Välja et al. [15] is to automatically create enterprise IT architecture models by collecting, processing and combining data from more than one information sources, in particular from the NeXpose and Wireshark network scanners and by enriching the P^2CySeMoL security meta-model with the collected data. Farwick et al. [8] provide a context-specific approach for semi-automated enterprise architecture documentation. Farwick et al.'s approach consists of several configurable documentation techniques, a method assembly process, as well as an accompanying meta-model to store necessary meta-data for the process execution.

Johnson et al. [12] automatically create dynamic enterprise architecture models. The models leverage Dynamic Bayesian Networks to predict the presence of particular entities of an enterprise IT architecture over time.

Next, we investigate approaches for the automated population of EA models that have actually been implemented and evaluated. Using this inclusion

Table 1. Comparison of published approaches of automated EAD to our approach.

	Commonalities	Differences
Farwick (2010)	- Both approaches use agents in order to collect relevant data.	- The focus of Farwick (2010) lies on IT-infrastructure data, whereas we focus on standard software data from servers or clients. - Farwick (2010) focuses on collecting data from cloud specific information sources.
Holm (2014)	- Both approaches are using primary information sources for the automated collection of data. - Both approaches require access on the investigated devices to identify application components. - Both approaches can only identify a subset of application components.	- Holm (2014) can only collect data of applications that have an open interface to the outside of the server. - Holm (2014) supports multiple entities of all EA layers, whereas our approach supports only the collection of application component data. - Holm (2014) also uses an indirect way to collect data by using unauthenticated network scans (from the outside), but is not able to collect information about application components.
Buschle (2012)	- Both approaches collect data on application components (ArchiMate application layer)	- The proposed approach needs to formulate transformation rules in order to propagate data from SAP PI to other modelling tools. - The proposed approach needs a manual effort for creating data in SAP PI wheras our approach automatically collects installed software information from the running devices. - The proposed approach aims to maximize the model coverage of all ArchiMate layers. - The primary information source of our approach are is the automatically collected data from running devices, whereas the primary information source of Buschle (2012) is the ESB (SAP PI).
Hauder (2012)	- Both approaches collect data on application components (ArchiMate application layer).	- Hauder (2012) extends existing manually created EA models automatically with data from SAP PI and Nagios, whereas our approach uses a single source, namely the running devices themselves, to collect data. - Hauder (2012) classifies data with the help of transformation rules (manual task), whereas we use machine learning in order to automate the classification. - Hauder (2012) supports multiple EA layers, namely business, application and infrastructure layers, in contrast to our approach application components are collected manually or need to be configured in Nagios.
Välja (2015)	- Both approaches use automatically collected data. - Both approaches identify application components.	- Välja (2015) uses an indirect way to collect data by the use of NeXpose (network scanner) and Wireshark (network traffic analyzer), our approach places agents on devices in order to collect data. - Välja (2015) focuses on the infrastructure layer, whereas we focus on the application layer. - Välja (2015) also focuses on identifying relationships between entities of the infrastructure and application layers. - The main goal of Välja (2015) is to define a process for integrating data from different sources.
Farwick (2016)	-	- The approach by Farwick (2016) supports various types of information sources, e.g., CMDB, ESB, and Server Configurations, whereas our approach's primary information source is the running device itself. - Farwick (2016) manually maps the data import to the organization-specific information model. - The approach by Farwick (2016) supports the whole EA documentation, whereas our approach focuses only on the application components. - Farwick (2016) defines a process to adapt automated collection of data to specific organizational contexts.
Johnson (2016)	- Both approaches make the use of machine learning, however Johnson (2016) investigates the state estimation problem, in contrast our approach tackles a categorization problem.	- The focus of Johnson (2016) lies on the infrastructure layer, whereas we focus on the application components. - Johnson (2016) does not provide an implementation of the proposed approach, wheareas we evaluate the basic technical feasability. - Johnson (2016) uses a Dynamic Bayesian Network to account for insecurities in data collection.

criterion, we identified five[2] approaches [2, 3, 9, 10, 15]. Subsequently, we compared these approaches by contrasting which EA entities can be automatically retrieved from the different respective information sources.

Table 2. Comparison of approaches and respectively used information sources for automatically generating and populating EA models.

	Information Source	Buschle (2012)	Hauder (2012)	Buschle (2011) and Holm (2014)	Välja (2015)
	Excerpt of ArchiMate 3.0.1 entities, automatically extracted from information sources				
Business Layer	Business actor			NeXpose network scanner	
	Business interface	SAP PI (possibly)			
	Business process	SAP PI (possibly)			
	Business function		Iteraplan		
	Business service	SAP PI (possibly)			
	Business object	SAP PI (possibly)	SAP PI (depends on concrete instance)		
	Representation	SAP PI (possibly)			
	Product	SAP PI (possibly)			
Application Layer	Application component	SAP PI	Iteraplan, SAP PI	NeXpose network scanner	NeXpose and Wireshark network scanners
	Application collaboration	SAP PI	SAP PI		
	Application interface	SAP PI	Iteraplan, SAP PI	NeXpose network scanner	
	Application service	SAP PI (possibly)			
	Data object	SAP PI			
Infrastructure Layer	Node	SAP PI	Iteraplan, SAP PI, Nagios (possibly)		
	Device	SAP PI	Nagios (possibly)	NeXpose network scanner	
	System software	SAP PI		NeXpose network scanner	NeXpose and Wireshark network scanners
	Technology interface			NeXpose network scanner	NeXpose and Wireshark network scanners
	Path	SAP PI			
	Communication network			NeXpose network scanner	

To enable comparability between retrieved EA entities, we use the concepts defined in the ArchiMate 3.0.1[3] framework as a basis [14]. The ArchiMate framework defines a meta-model with generic EA entities and EA entity relationships across three different layers: the Business Layer, the Application Layer and the Technology Layer.

The detailed comparison of implemented approaches is depicted in Table 2. We excluded ArchiMate entities which could not be automatically populated in any of the identified approaches. On the Business Layer, the excluded entities are Business Role, Business Collaboration, Business Interaction, Business Event, and Contract. On the Application Layer, the Application Function, Application Interaction, Application Process and Application Event entities are excluded. Finally, on the Technology Layer the entities Technology Collaboration, Technol-

[2] Note that Holm et al.[10] is an extension of Buschle et al.[3].

[3] In most papers, the authors use an earlier version of the ArchiMate framework, e.g. ArchiMate 2.0. To allow comparability, the EA entities of earlier version versions have been carefully mapped to ArchiMate 3.0.

ogy Function, Technology Process, Technology Interaction, Technology Event, Technology Service, and Artifact could not be populated. This shows, that even though some approaches for automatic EA modeling have already been evaluated, they are far from capturing the whole EA model. One more approach that should be mentioned here is [12]. The authors provide a full list of all ArchiMate 2.0 entities and possible information sources for automated modeling. For example, to populate the entity Technology Service, information could be retrieved from network scanners, directory services, software asset inventory Tools and possibly network sniffers. Nevertheless, the use of these information sources for automatic modelling has not been implemented nor evaluated.

Summarizing, only a few approaches to automatic EA modelling have actually been evaluated and these approaches only cover limited parts of the EA model. This emphasizes the relevance of research in automated EAD.

4 Evaluation

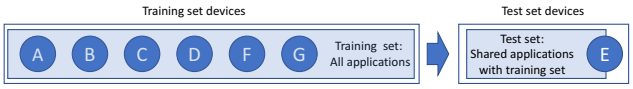

Fig. 1. Leave-one-out training- and test-set splitting: inspired by k-fold cross-validation, we leave the binary executables for applications from one device as test-set out, while applications from all other devices constitute the training set. The test-set contains only applications where a record is present in the training set (shared applications), while the training set contains all records irrespective of the application's presence in the test-set.

We investigate the basic technical feasibility of our envisioned machine learning based approach to detect and identify applications in an IT landscape. Our approach identifies all binary executables on a device and then identifies the respective application through a machine learning classifier. We investigate the basic technical feasibility especially with respect to the major challenge of a many-label classification problem. To the best of our knowledge, there exists no (large-scale) dataset, yet. Therefore, we constructed a small dataset to conduct initial experiments.

A machine learning approach based on artificial neural networks requires a careful investigation of the neural network structure, the parameter selection, and regularization. In the following, we distinguish two viewpoints: from a machine learning point of view, we investigate the generalization capabilities of machine learning methods on such a dataset. Secondly, we also investigate the task from the perspective of practical applicability. In particular, we attempt to answer the following research questions regarding the technical feasibility of our approach by conducting experiments:

1. Do application binary executables hold discriminative information that allows to identify the applications? (RQ1)
2. Can machine learning algorithms tackle this multi-class problem? (RQ2)
3. Are machine learning algorithms capable of generalizing the classification task for different devices? (RQ3)

We created a dataset from seven different MacBook Pro devices from researchers of our research group. We used the python-magic library (a wrapper to libmagic) that allows us to identify executable binary files in the */app/* directory[4]. All experiments have been carried out on a MacBook Air (1.6 GHz Intel Core i5, 8GB RAM) with Keras[5] and Tensorflow[6]. The dataset consists of 3026 total records with the first 8096 bits of an executable binary as features (cf. Figure 4) which is labeled with the application name (filename). The applications are, for example, labeled with *Dropbox* or *MS Word*, but also helper executables, for example, *CacheHelper* are contained. On the one hand, for an initial evaluation this leads to a high-quality, labeled dataset of standard software with low effort. On the other hand, the dataset is limited to applications from one operating system, few different versions of the same applications and predominantly non-server applications.

Table 3. General dataset characteristics: the dataset consists of 3096 binary executables labeled with their filename collected from seven different Macbook Pro devices running Mac OS X from the applications folder. Note that the features differ for equal version applications on different devices. We use the Hamming distance as an indicator of the variation among the features for two applications and accumulate these for specific training- and test-set splittings with shared applications among training- and test-set, cf. Figure 1.

#devices	7
OSX versions	10.12.5, 10.12.6
#total appl.	3026
# unique appl.	1172

	Accum. Hamm. Distance	# Shared Appl.
Max	1251.71	370
Avg	1009,87	212.14
Min	923.13	370

In order to answer the research questions with experiments, we choose a train- and a test-set split of the dataset inspired by k-fold cross-validation. The records from all except one device serve as a training-set. The applications from the omitted device serve as test-set when there are corresponding records in the training set. I.e. the test set contains only applications that are present in the training set, while the training set contains applications that are not present in the test-set, cf. Figure 1.

We use the Hamming distance (number of non-equal bits between a pair of binary executable strings) as a similarity measure between the features of two

[4] Executable binaries identified as *Mach-O 64-bit x86_64 executable* filetype
[5] Keras, v. 2.0.4, https://keras.io/, last accessed in November 2017
[6] Tensorflow v. 1.0.1, https://www.tensorflow.org/

Table 4. Training- and test-set combinations and characteristics: For seven devices, seven different training- and test-set combinations can be formed using the leave-one-out approach. The total number of records in a leave-one-out dataset with shared applications in the test-set only is around 850 to 1550. The accumulated Hamming distance between all pairs of applications in the training- and test-set as well as the Hamming distance normalized using the number of pairs considered. Here, duplicates are included. Regarding these characteristics, the datasets appear very similar.

Test-set device	**Total records**	**Accumulated Hamming distance**	**Normalized accumulated Hamming distance (including duplicates)**	**Normalized accumulated Hamming distance (duplicates removed**
A	1100	1018971	926.3	1281.7
B	846	813668	961.7	1329.5
C	1055	1151599	1091.6	1258.5
D	1061	1021264	962.5	1289.4
E	1049	998913	952.2	1102.5
F	1569	1448398	923.1	1849.8
G	1155	1445735	1251.7	1897.2

records. We use the Hamming distances as an indicator of how well a machine learning algorithm could work to predict the application from its executable binaries, i.e. whether the binary strings contain discriminative information. From a machine learning point of view, the generalization capabilities can be only determined when exact duplicates of applications (Hamming distance equals zero) are removed, i.e. the test dataset does not contain exactly the same samples as in the training sets. We distinguish among training- and test-sets with and without duplicates. To create a dataset with no duplicates all instances of binary strings are removed from the training-set when they are equal to the test-set. Table 4 shows the Hamming distances for all dataset splittings. If no test-set instance remains, the test-set instance is removed. The datasets with duplicates contain all records. We assume that in a real-world setting duplicates occur often.

If not stated otherwise, all following experiments have been carried out with simple feed-forward neural networks (FFNN) with one dense hidden layer with 50 neurons, a batch size of 32, that are trained for 100 epochs. We choose the accuracy measure to evaluate classification performance. The precision/recall and derived F1 measure are not well suited for this evaluation, because we are interested solely if an application was classified correctly or not, i.e. there is no *relevance* criterion for this problem that is present, for example, in information retrieval tasks.

Experiment 1: Network structure & parameters for classifier: In order to obtain credible results using a neural network classifier, one has to empirically determine a suitable network structure and reasonable values for hyperparameters. We ran experiments with more neurons in a hidden layer (25,

Table 5. Prediction accuracy (train and test-set) for a FFNN (1 hidden layer with 50 neurons) after 100 epochs of training for all training-/test-set combinations where exact duplicates of binary executables have been included or removed. For the more real-world like case that exact duplicates of application binaries in the training-set also occur in the test-set, we achieve very good results with 98 percent accuracy. For the more scientifically interesting case that exact duplicates are removed we achieve reasonable results with up to 64 percent accuracy. However, two test-sets achieve very poor results of around eight percent accuracy and we investigate this in the remainder of the paper. The best results are indicated with *, the most relevant results are indicated in bold font.

		With duplicates				Without duplicates			
Test set device	**Normalized accumulated Hamming distance**	**Train samples**	**Test samples**	**Train acc. (%)**	**Test acc. (%)**	**Train samples**	**Test samples**	**Train acc. (%)**	**Test acc. (%)**
A	926.3	1081	159	99.07	94.34	795	159	99.12	**61.64***
B	961.7	846	100	98.83	**98.00***	612	100	99.02	58.00
C	1091.5	1035	159	94.30	55.97	915	159	94.10	44.65
D	962.5	1042	138	94.63	86.96	792	138	94.44	57.25
E	952.2	1033	162	98.84	81.48	906	162	99.23	59.26
F	923.1	1119	146	94.10	96.58	783	146	93.49	**8.22**
G	1251.7	930	146	94.19	97.26	762	146	93.70	**8.22**

50, 100, 300, 500, 1000), but accuracy did not improve. We chose 50 neurons for one hidden layer. We experimented with two layers, but more layers did not improve the results significantly. We can conclude that there are no higher-order correlations among the positions of the bits. We also varied the batch size (25, 50, 64, 75) without a major difference in the results.

Experiment 2: Prediction results with neural networks: The key result for a classification algorithm is the prediction performance on the task at hand. To answer the research questions (especially RQ1 and RQ2), we carried out experiments using the already identified network structure and parameters on all possible splittings of the dataset with duplicates and removed duplicates. The results after 100 epochs of training are displayed in Table 5. For five out of seven dataset splittings, we can report reasonably good performance on the test-set in the scientifically relevant case where exact duplicates have been removed and very good results for the more practically relevant case with duplicates included. However, two dataset splittings give very poor performance results (F and G). For the bad performing device test-set F, most applications are wrongly identified as the *autoupdate* application. Despite the fact, that this particular application occurs very often (but not most often) in the training set, we also investigated the Hamming distances for all applications in the test-set against this particular application. In contrast to the well-performing test device split A, the accumulated Hamming distance for *autoupdate* with the other applications in

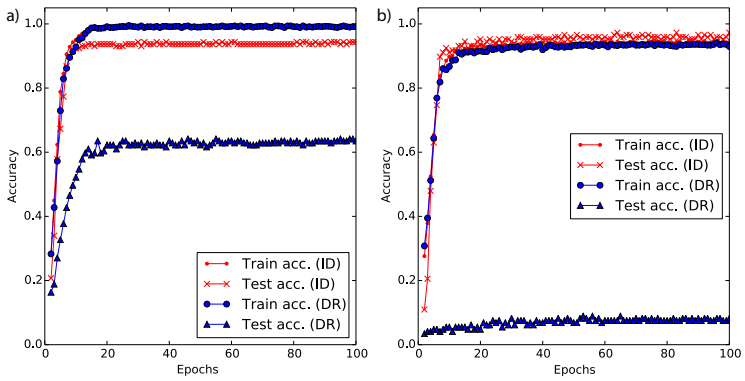

Fig. 2. Training- and test-set accuracy over 100 epochs for the best performing dataset A (a) and the worst performing dataset F (b) with a FFNN (1 hidden layer, 50 neurons). Despite around 3000 samples, our dataset is comparably small and the networks converge already after around 20 epochs which takes circa one minute. (ID=Including duplictes, DR=Duplictes removed)

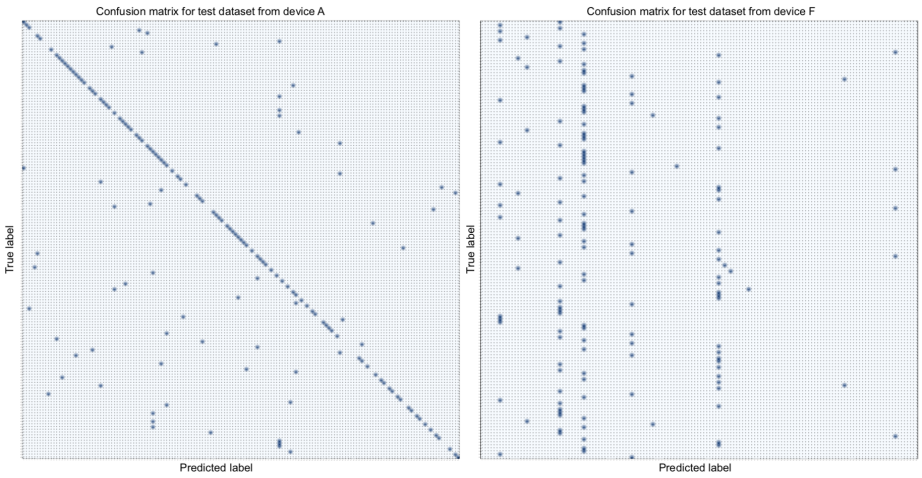

Fig. 3. The confusion matrices for datasets A and F (duplicates removed) reveal that in the well performing case (left), despite the challenge of a very large number of different classes, the majority of applications is classified correctly. On the other hand, for the bad performing dataset F, few applications are dominant and responsible for many wrongly classified records and we were able to identify a problem using Hamming distances, described in the remainder of this paper.

the test-set for test device split G was significantly lower (157249 versus 108933) and also showed up in the top 10 in an ascending list of accumulated Hamming distances. We conclude that this very low accumulated Hamming distance can serve as an indicator for the poor prediction results. However, since five out of seven test device splittings perform well, we conclude that the many-label problem can be feasible for this classification task and our approach. For the best and worst performing dataset splittings we also examined the training over time, cf. Figure 2 that shows that the training is stable and network convergence is reached after almost 20 epochs.

Experiment 3: Different machine learning algorithms: In order to rule out a biased success using neural networks, we also trained a decision-tree classifier[7] with default parameters. The results displayed in Table 6 show that other machine learning algorithms can achieve similar results. As can be expected, our optimized neural networks outperform the decision-tree algorithm with default parameters in certain cases by 10 percent.

Table 6. Comparison of different machine learning algorithms: Tree-based classifier (with standard parameters) versus FFNN classifiers on the best-performing dataset splitting A as well as the worst-performing dataset split F (best-performing regarding the different dataset splittings with FFNN). On the best dataset-split the FFNN performs significantly better than the tree-based classifier. On the worst performing dataset-split the tree-based classifier performs slightly better than the FFNN. The result that different classification algorithms can perform well on the problem at hand helps us to rule out exclusively positive side-effects of FFNNs.

Test set device	Duplicates	1-layer FFNN	Decision Tree
A	included	**99.07**	89.31
A	removed	**61.64**	53.45
F	included	96.58	**98.63**
F	removed	8.22	**9.58**

Experiment 4: Feature engineering: Feature length: We conducted experiments with our default setup and varied the number of bits that enter the classifier to see if this reduces the amount of discriminative information present in the data. The results, depicted in Figure 4 indicate that this is the case (for dataset split A with duplicates removed and included), but we would have expected a much stronger drop in the classification performance for 100 bits. However, we assume for larger datasets an increasing number of features will help classifiers.

Experiment 5: Network regularization: Over-fitting is a problem that occurs in any neural network application and is tied to RQ3. A standard way to tackle this problem is to use a regularization method, for example dropout, to prevent the networks during training from over-fitting. A randomly selected

[7] scikit-learn, v. 0.19.1, http://scikit-learn.org/stable/modules/tree.html, last accessed in November 2017

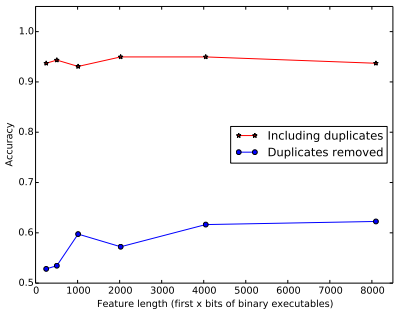

Fig. 4. The feature length, i.e. the amount of the first X bits of the binary executables, have a comparably small effect on the accuracy achieved on the test-set on the best performing dataset A for both cases: duplicates included or removed. However, a too small feature length performs poorly, because no discriminating information is left. We assume that for larger datasets also the number of features needs to be increased.

number of neurons is deactivated during training, e.g., 10 percent of the neurons corresponding to a dropout rate of *0.1*. However, several experiments with our standard network structure and also two layers with 50 neurons in each layer did not significantly improve classification accuracy on the test-set for dataset split A. For the poor performing dataset split F, a low dropout rate (1.0 for the first layer, 0.08 for the second layer) can improve classification accuracy on the test set around one percent. For larger dropout rates, the classification performance gets, as expected, worse. We conclude that a carefully chosen small dropout rate is useful, but it does not significantly improve the classification results.

5 Limitations & Critical Reflection

We conducted a series of first experiments to evaluate the initial technical feasibility of our approach. Despite a dataset with roughly 3000 samples, the major drawbacks of our evaluation are the artificial setup and the small dataset. An evaluation of a much larger and real-world dataset is necessary. On the one hand, many-label classifications are technically difficult from a machine learning perspective and require a larger dataset, in general, but also specifically for the evaluation of this task. On the other hand, we neglected certain aspects of a real-world setup, for example the classification of applications installed on different operating systems, or the classification of different versions of the same application. Moreover, our dataset is restricted to desktop applications and does not fully reflect an IT-landscape with server applications. We also did not include custom-developed applications or applications within application servers. We identified the Hamming distance as a potential tool to identify and investigate poor classification performance results, however, so far we have not identified the reason why abnormal Hamming distances occur among applications.

6 Conclusion & Future Work

We envision a novel, machine learning based approach to discover and identify standard software in large IT-landscapes that can be used for software asset management or enterprise architecture documentation in an automated and continuous manner by framing the application detection and identification problem of applications as a classification problem of executable binaries. We identified related and complementary approaches in the domain of enterprise architecture documentation and evaluated the EA model coverage and the degree of automation in the form of a small literature study. We identified two major challenges for our approach: the many-label nature of the classification problem and the scarce occurrence of poor classification results. Despite the challenge of a many-label problem, we can report promising results for the technical feasibility of the approach evaluated with experiments on a dataset of applications collected from MacBooks from researchers at our research group. We can report that the hamming distance distributions among applications executable binaries are a good indicator to predict the quality of the results.

For the future, an evaluation on a larger, real-world dataset is necessary to examine the applicability of the approach under real-life conditions including applications from different operations systems and different versions of the same application. A deeper understanding of the causality between hamming distance distributions and classification results or other measures to predict the quality of classification results would be beneficial. An investigation of unsupervised methods to identify groups of related applications seems beneficial to us, e.g., to identify applications that have similar functionality, e.g., databases and application servers. Eventually, other machine learning approaches (e.g., Iyer et al. [11]) could be used to additionally identify and classify individually developed applications. We also see a strong benefit in combining entity detection and identification methods with dynamic models of an EA such as proposed by Johnson et al. ([12]).

Acknowledgment

This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

References

1. Brückmann, T., Gruhn, V., Pfeiffer, M.: Towards real-time monitoring and controlling of enterprise architectures using business software control centers. In: Software Architecture - 5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011. Proceedings. pp. 287–294 (2011)

2. Buschle, M., Ekstedt, M., Grunow, S., Hauder, M., Matthes, F., Roth, S.: Automating enterprise architecture documentation using an enterprise service bus. In: AMCIS 2012 Proceedings. AIS Electronic Library (AISeL) (2012)
3. Buschle, M., Holm, H., Sommestad, T., Ekstedt, M., Shahzad, K.: A Tool for Automatic Enterprise Architecture Modeling, pp. 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
4. Das, S., Dey, A., Pal, A., Roy, N.: Applications of artificial intelligence in machine learning: Review and prospect. International Journal of Computer Applications **115**(9) (2015)
5. Dijkman, R.M., Pires, L.F., Rinderle-Ma, S. (eds.): 20th IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2016, Vienna, Austria, September 5-9, 2016. IEEE Computer Society (2016)
6. Farwick, M., Breu, R., Hauder, M., Roth, S., Matthes, F.: Enterprise architecture documentation: Empirical analysis of information sources for automation. In: 2013 46th Hawaii International Conference on System Sciences. pp. 3868–3877 (2013)
7. Farwick, M., Agreiter, B., Breu, R., Häring, M., Voges, K., Hanschke, I.: Towards living landscape models: Automated integration of infrastructure cloud in enterprise architecture management. In: Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on. pp. 35–42. IEEE (2010)
8. Farwick, M., Schweda, C.M., Breu, R., Hanschke, I.: A situational method for semi-automated enterprise architecture documentation. Software & Systems Modeling **15**(2), 397–426 (2016)
9. Hauder, M., Matthes, F., Roth, S.: Challenges for automated enterprise architecture documentation. In: Trends in Enterprise Architecture Research and Practice-Driven Research on Enterprise Transformation, pp. 21–39. Springer (2012)
10. Holm, H., Buschle, M., Lagerström, R., Ekstedt, M.: Automatic data collection for enterprise architecture models. Software & Systems Modeling **13**(2), 825–841 (2014)
11. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers. The Association for Computer Linguistics (2016)
12. Johnson, P., Ekstedt, M., Lagerström, R.: Automatic probabilistic enterprise IT architecture modeling: A dynamic bayesian networks approach. In: 20th IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2016, Vienna, Austria, September 5-9, 2016. pp. 1–8 (2016)
13. Roth, S., Hauder, M., Farwick, M., Breu, R., Matthes, F.: Enterprise architecture documentation: Current practices and future directions. In: Wirtschaftsinformatik. p. 58 (2013)
14. The Open Group: Archimate®3.0.1 specification, an open group standard (2016)
15. Välja, M., Lagerström, R., Ekstedt, M., Korman, M.: A requirements based approach for automating enterprise it architecture modeling using multiple data sources. In: Enterprise Distributed Object Computing Workshop (EDOCW), 2015 IEEE 19th International. pp. 79–87. IEEE (2015)