Classifying Unstructured Models into Metamodels using Multi Layer Perceptrons

Walmir Oliveira Couto[1,2], Emerson Cordeiro Morais[2] and Marcos Didonet Del Fabro[1]
[1]*C3SL Labs, Federal University of Paraná, Curitiba PR, Brazil*
[2]*LADES Icibe, Federal Rural University of Amazon, Belém PA, Brazil*
{*walmir.couto, emerson.morais*}*@ufra.edu.br, marcos.ddf@inf.ufpr.br*

Keywords: Classifying Unstructured Models, Model Recognition, Artificial Neural Network, MLP.

Abstract: Models and metamodels created using model-based approaches have restrict conformance relations. However, there has been an increase of semi-structured or schema-free data formats, such as document-oriented representations, which are often persisted as JSON documents. Despite not having an explicit schema/metamodel, these documents could be categorized to discover their domain and to partially conform to a metamodel. Recent approaches are emerging to extract information or to couple modeling with cognification. However, there is a lack of approaches exploring semi-structured formats classification. In this paper, we present a methodology to analyze and classify JSON documents according to existing metamodels. First, we describe how to extract metamodels elements into a Multi-Layer Perceptron (MLP) network to be trained. Then, we translate the JSON documents into the input format of the encoded MLP. We present the step-by-step tasks to classify JSON documents according to existing metamodels extracted from a repository. We have conducted a series of experiments, showing that the approach is effective to classify the documents.

1 INTRODUCTION

Models and metamodels created using model-based approaches have restrict conformance relations, meaning that each element of a given model must conform to a metamodel element. For instance, an element ”Student” in a model could conform to the element ”Class” in a Java metamodel. Similar relationships, with different terminology, are present in other data models, such as a database tuple and its table/column definitions, or an XML document and its corresponding schema.

These relationships are restrictive and cannot be applied to any kind of data model, in particular when relying on semi-structured or schema-free representations, where there is no explicit conformance relation. Such kind of data is very good for fast application development, coming with the cost of being loosely typed.

The most common semi-structured format are document stores, which are often persisted using JSON documents. JSON documents are used for interoperability, storage of application data where flexibility is important and also are becoming a de-facto standard in RESTful APIs implementations. Despite not having an explicit metamodel/schema, there are initiatives, such as JSON schema, as solution to provide typed JSON documents.

When JSON schemas are not defined, i.e., for untyped documents, it is useful to classify JSON documents to discover whether they could be categorized into a given domain and to partially-conform to a metamodel or JSON schema. Recent approaches are emerging to extract information or to couple metamodeling with cognification, i.e., to extract knowledge from models and metamodels and to apply machine learning techniques to discover useful information (Cabot et al., 2017; Perini et al., 2013). The paper from Burgueño (Burgueño, 2019) shows an approach which uses Long Short-Term Memory Neural Networks (LSTM) to automatically infer model transformations from sets of input-output model pairs. Another one from (Nguyen et al., 2019) employed Machine Learning techniques for metamodel automated classification implementing a feed-forward neural network. However, there is a lack of approaches about unstructured models' classification. There are studies comprising classification of complex structures, such as solutions on graph classification (Zhang and Chen, 2018), but they are not focused on unstructured models, meaning there is an open field to be studied.

In this paper, we present a methodology to analyze

and classify JSON documents according to existing metamodels. We extract existing metamodels using a One-hot encoding solution into a Multi-Layer Perceptron (MLP) network, translating the metamodel elements into the input neurons. The neural network is trained and then used to classify input JSON documents, which are as well translated into the input data to be classified. We present the step-by-step tasks to achieve that. We have conducted a series of experiments, using neural networks with different intermediate layers, showing that the approach is effective to classify the documents.

This paper is organized as follows. Section 2 presents our approach to classify document stores into metamodels. Section 3 describes the experimental validation and discussions. Section 4 is the related work, and section 5 presents the conclusions.

2 CLASSIFYING DOCUMENT STORES INTO METAMODELS

In this section we show how to classify document stores into metamodels. We start by a brief introduction to MLP, then we present the definitions and environmental assumptions. Finally we describe the formalization of the solution.

2.1 Bried Introduction to MLP

In this section we present a brief background on MLP (Multi-Layer Perceptron). The *perceptron* is a algorithm that performs binary classification, i.e., it predicts whether a given input belongs to a certain category of interest or not (Kumari et al., 2018). A perceptron is a linear classifier; that is, it is an algorithm that classifies input using a linear prediction function, which needs to be defined. The input is a feature, named *vector x*, where each element is multiplied by a set of weights w and added to a bias b: $y = w * x + b$. A multilayer perceptron (MLP) is a deep, artificial neural network. It is composed of more than one perceptron. They are composed of an input layer to receive the signal, an output layer that makes a decision or prediction about the input, and in between those two, an arbitrary number of hidden layers that are the true computational engine of the MLP.

Formally, a MLP is a function $f : R^D \rightarrow R^L$, where D is the size of input vector x and L is the size of the output vector given by $f(x)$, such that, in matrix notation: $f(x) = G(b^{(2)} + W^{(2)}(s(b^{(1)} + W^{(1)}x)))$, with bias vectors $b^{(1)}$, $b^{(2)}$; weight matrices $W^{(1)}$, $W^{(2)}$ and activation functions G and s. The vector $h[x] \leftarrow \Phi(x) = s(b^{(1)} + W^{(1)}x)$ constitutes the hidden layer. $W^{(1)} \in R^{D \times D_h}$ is the weight matrix connecting the input vector to the hidden layer. Each column $W^{(1)}_{\cdot i}$ represents the weights from the input units to the $i-th$ hidden unit. We use this definition in the remaining of our work.

Multilayer perceptrons are often applied to supervised learning problems: they train on a set of input-output pairs and learn to model the correlation (or dependencies) between those inputs and outputs. Training involves adjusting the parameters, or the weights and biases of the model, in order to minimize errors. Backpropagation is used to make those weight and bias adjustments relative to the error, and the error itself can be measured in a variety of ways, including by Root Mean Squared Error (RMSE).

2.2 Extracting Metamodels into MLP features

First, we consider $\mathcal{M}$ the input metamodel, which defines the structured information used as input to train the network. A metamodel is composed of classes, attributes, and references, which are translated into a collection of name/value pairs in a JSON format. $\mathcal{E}$ denotes the set of classes, attributes, and references in $\mathcal{M}$, where $\mathcal{E} \subset \mathcal{M}$. In order to illustrate our approach, the execution schema is depicted in Figure 1.

The *Driver Program* implements the control flow and it launches the operations, managing the step by step execution schema[1]. It starts reading the input metamodel $\mathcal{M}$, and it assigns it to a top level single dataset D_s as $D_s \leftarrow \mathcal{M}$. A dataset is a collection of data which can be split into others datasets (Armbrust et al., 2015b). D_s is processed using an *extraction* function $f_e(x)$ which selects classes (c), attributes (a) and references (r), where $\{c,a,r\} \subset \mathcal{E}$, assigning each one of them to specific datasets $d_{s(0)}...d_{s(n)}$, where n is the total number of elements. Once this conversion is done, there is no distinction between the types of the elements to encode the MLP features. Then, each one of these datasets $d_{s(0)}...d_{s(n)}$ is converted into a binary number and bundled to create the *MLP Vector X: set of input layer neurons* $\{x_i | x_1, x_2, ..., x_m\}$.

The set of elements in the input data sets are extracted and encoded into a *MLP Vector X* applying a One-hot Encoding (OHE) technique, which is a widely used technique for transforming categorical features to numerical features. Then, the network is trained using a *MLP Classifier*, using a set of training

[1]It was developed on *Apache Spark*, an analytics engine for data processing (Armbrust et al., 2015a).

samples. Once the training step is finished, it is possible to perform the unstructred models classification.

To perform the metamodel extraction we define Algorithm 1. It starts by reading the input metamodel $\mathcal{M}$, applies an *extraction* function $f_e(x)$ and assigns to dataset D_s. *ItemsAmount* receives distinct classes, attributes and references amount which we use to calculate the binary digits amount used to depict these classes, attributes and references in a binary vector which it is used as input features on MLP. For building this binary vector, we use OHE technique because categorical data must be converted to numbers when we are working with a sequence classification type problem and plan on using neural networks.

At line 4, we create *MLPVectorX* to store all distinct classes, attributes, and references of $\mathcal{M}$ as a binary vector. From line 5 to 8, for each $distinct(c,a,r \subset \mathcal{E}) \in D_s$ we apply an *extraction* function $f_e(x)$ splitting D_s in classes (c), attributes (a) or references (r), and it assigns each one to datasets $d_{s(0)}...d_{s(n)}$. It is important to note that for the MLP neural network there is no difference between classes, attributes, and references, each one is a binary number in *MLPVectorX*. At line 10, *binaryDigitsAmount* takes n integer value as the *exponential* function result which takes *ItemsAmount* as a parameter. Then, from line 11 to 15, each $d_{s(0)}...d_{s(n)}$ is converted into a binary number with *binaryDigitsAmount* digits applying a *BinaryGenerator* function which takes *binaryDigitsAmount* as a parameter, and it assigns it to *MLPVectorX*, which it used as the MLP input features (set of input layer neurons). The reference between $d_{s(n)}.element.name$ and its corresponding binary number at $MLPVectorX.[n]$ is assigned to a special dataset *sD* at line 13, and we write *sD* in JSON file $ReferenceElementBinary$ which it will be used to help represent models in JSON documents as the MLP input. This enables to maintain a mapping between the metamodel elements and their corresponding elements in the network.

Consider a simplified Java metamodel[2] represented by a UML class diagram. The algorithm converts each class, attribute, and references in a name/value pair in a JSON file, thereby creating the input metamodel $\mathcal{M}$.

[2]The metamodel used is the one from the following public link: https://www.eclipse.org/atl/atlTransformations/UML2Java/, ExampleUML2Java[v00.01].pdf

Algorithm 1: Extracting Metamodels into a MLP.

Input: Input Metamodel $\mathcal{M}$.
Output: MLP Vector X, ReferenceElementBinary.

1: $D_s \leftarrow f_e(\mathcal{M})$
2: $ItemsAmount \leftarrow count(distinct(c,a,r \subset \mathcal{E}) \in D_s)$
3: $binaryDigitsAmount \leftarrow 0$
4: $MLPVectorX \leftarrow empty$
5: **for** $(n = 0$ **to** $ItemsAmount - 1)$ **do**
6: $\quad d_{s(n)} \leftarrow f_e(D_s.[n].element.name)$
7: $\quad n \leftarrow n+1$
8: **end for**
9: $n \leftarrow 0$
10: $binaryDigitsAmount \leftarrow toInt(exp(2^n = ItemsAmount))$
11: **for all** $d_{s(0)}...d_{s(n)}$ **do**
12: $\quad MLPVectorX.[n] \leftarrow BinaryGenerator(d_{s(n)}, binaryDigitsAmount)$
13: $\quad sD \leftarrow f(d_{s(n)}.element.name, MLPVectorX.[n])$
14: $\quad n \leftarrow n+1$
15: **end for**
16: $SaveFile(sD, ReferenceElementBinary)$
17: **return** $MLPVectorX, ReferenceElementBinary$

In this case, the *JavaElement* class is assigned to $d_{s(0)}$ dataset, *name* attribute is assigned to $d_{s(1)}$ dataset, and so on, and then each dataset from $d_{s(0)}...d_{s(n)}$ is converted into a binary number through a *BinaryGenerator* function, and assigned it to *MLPVectorX*. For instance, after executing the algorithm, *MLPVectorX* will have 20 positions, and its structure can be seen in Table 1.

Table 1: Classes, attributes and references in *MLPVectorX*.

***MLPVectorX* structure for Java metamodel**			
Element	**Type**	**Position**	**Value**
JavaElement	class	0	0000000
name	attribute	1	0000001
Type	class	2	0000010
Modifier	class	3	0000011
isPublic	attribute	4	0000100
isStatic	attribute	5	0000101
isFinal	attribute	6	0000110
PrimitiveType	class	7	0000111
Method	class	8	0001000
isAbstract	attribute	9	0001001
Field	class	10	0001010
type	reference	11	0001011
parameters	reference	12	0001100
field	reference	13	0001101
owner	reference	14	0001110
methods	reference	15	0001111
JavaClass	class	16	0010000
classes	reference	17	0010001
package	reference	18	0010010
Package	class	19	0010011

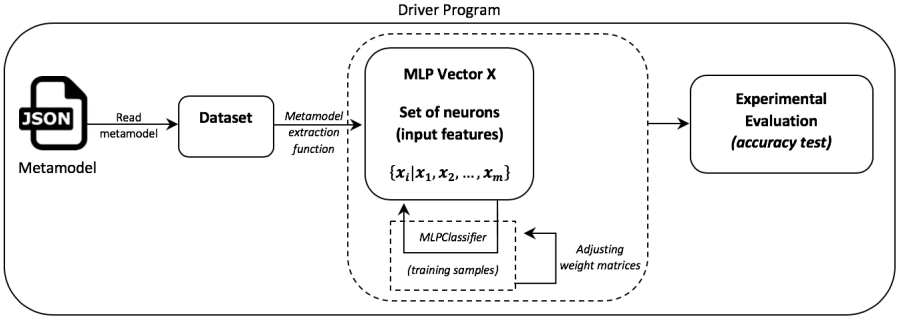

Figure 1: Execution flow for classification of unstructured models.

2.3 Training the MLP Neural Network

Once the input features are extracted, it is necessary to train the MLP neural network. To to this, it is necessary to choose a set of metamodels to extract its features using the explained algorithm. While any set of metamodels could be chosen, we illustrate the training step using a set of 4 metamodels: *MySQL*, *KM3*, *UML* and *Java*. Metamodels are 3rd party metamodels available in the ATL transformations web site[3]. These metamodels will be also used in our detailed experiments. For this subset of metamodels, the *MLPVectorX* has seventy two positions, i.e., we extracted seventy two distinct classes, attributes, and references. Thus, for each position in *MLPVectorX*, it is assigned a binary number with seven digits where $ItemsAmount = 72$ and $2^n = ItemsAmount$ then $n = 7$. All 72 binary conversions and extractions can be found on the github[4]. It is also necessary to choose the number of hidden layers, together with the number of input neuron for each layer. As there are no exact rules for determining the hidden layers number and the neuron number in each hidden layer, we choose three hidden layers, each one with three neurons, and one output layer neurons, which it will be assigned a binary number with two digits, i.e., for four output metamodels we have $2^n = 4$ then $n = 2$. We choose for an artificial network multi-layered with backpropagation training, and conventional random initialization, where each neuron in one layer, e.g. x_1, connects with a certain $w_{[a][b][c]}$ weight to every neuron in the following layer, e.g. j_1, j_2, j_3, where $_{[a]}$ is the origin layer number, $_{[b]}$ is the neuron number in the origin layer, and $_{[c]}$ is the neuron number in the following layer.

In addition, each neuron in the hidden layers is added to a bias $b_{[d][e]}$ weight, where $_{[d]}$ is the hidden layer number, and $_{[e]}$ is the neuron number in the hidden layer. We generate random initial weights, e.g. from the range [–1, 1], for each $w_{[a][b][c]}$ weight, it was generated 237 $w_{[a][b][c]}$ weights and 10 bias $b_{[d][e]}$ weights in total. It is important to note that, one set of updates of all the weights for all the training patterns is called one *epoch* of training. In this first MLP training, we set up 4000 *epoch* of training. We implement a second MLP training with the same amount of input neurons, hidden layers, and *epoch* of training, but with five neurons in each hidden layer, for this second MLP training it was generated 415 $w_{[a][b][c]}$ weights and 16 bias $b_{[d][e]}$ weights in total. All $w_{[a][b][c]}$ and $b_{[d][e]}$ weights can be found on the github[5].

In our MLP training, we choose s the logistic *sigmoid* function for the activation functions, with $sigmoid(a) = 1/(1+e^{-a})$. The output vector is then obtained as: $o(x) = G(b^{(2)} + W^{(2)}h(x))$. To train a MLP, we need to learn all parameters of the model. The set of parameters to learn is the set $\theta = \{W^{(2)}, b^{(2)}, W^{(1)}, b^{(1)}\}$. A neural network is stopped training when the error, i.e., the difference between the desired output and the expected output is below some threshold value or the number of iterations or epochs is above some threshold value; in our approach, MLP training is stopped when the Mean Squared Error (MSE) is less than 0.01 (1%).

2.4 Representing Models in JSON Documents as the MLP Input

Before starting the classification step, it is necessary to translate the input unstructured documents (in JSON) into a compatible format with the MLP input. We use a similar process to the one used to extract metamodels, as described in Algorithm 2 shown above. It starts by reading the input model m, which is formed by classes (c), attributes (a) and references (r), it applies an *extraction* function $f_e(x)$ and assign it to dataset D_sm. At line 2, we open a JSON file $ReferenceElementBinary$, created during the meta-

[3]https://www.eclipse.org/atl/atlTransformations/

[4]https://github.com/walmircouto/MLPTraining

[5]https://github.com/walmircouto/MLPTraining

Algorithm 2: Representing Models in JSON Documents as the MLP Input.

Input: m model, $ReferenceElementBinary$ JSON file.
Output: MLP text file.

```
1: Dsm ← fe(m)
2: ItemsAmount ← count(distinct(c,a,r ⊂ Dsm))
3: sd ← OpenFile(ReferenceElementBinary)
4: n ← 0
5: MLPTextFile ← empty
6: for (n = 0 to ItemsAmount − 1) do
7:    d_s(n) ← sd.select(name,binary) where name =
      Dsm.[n].element.name)
8:    if d_s(n) ≠ null then
9:       MLPTextFile ← MLPTextFile + d_s(n).binary
10:   end if
11:   n ← n + 1
12: end for
13: return MLPTextFile
```

model extraction process, and assign it to dataset *sd* which it will be used to found the reference between *element.name* and its corresponding binary number, assisting in the assembly of the $MLPTextFile$ text file. From line 5 to 11, for each $distinct(c,a,r \subset D_sm)$ we try to found the *element.name* from D_sm in dataset *sd*, applying a select function at line 6, to obtain the corresponding binary number, and the result of this is assigned to $d_s(n)$. If $d_s(n) \neq null$ then we start the assembly of the $MLPTextFile$ text file including the binary number from $d_s(n)$. The $MLPTextFile$ text file will be use as the MLP input to classify the input model m.

3 EXPERIMENTAL EVALUATION

We conduct a number of experiments to validate our approach. The main goal is to evaluate the precision of the constructed and trained network to classify unstructured JSON documents into metamodels, i.e., which is the percentage of documents correctly classified. All experiments were performed in a machine with 8 GB DDR3 RAM, processor 2,53 GHz Intel Core i5, MacOS High Sierra 10.13.6, Spark 2.3.3, Scala 2.12.8, and Java 1.8.0_191. The experiments had the following setting.

Network Configuration: as stated in the previous section, we define 2 MLP networks: both with 3 hidden layers, the first one with 3 neurons on each layer and the second one with 5 neurons on each layer, thus we executed two training, with 4000 *epochs* each. We know that one of the problems that occur during neural network training is called overfitting. The error on the training set is driven to a very small value, but when new data is presented to the network the error is large. The network has memorized the training examples, but it has not learned to generalize to new situations. This is expected, since in this experimental evaluation we do not address the overfitting problem for all kinds of metamodels, since we target domain specific scenario. We intend to carry out new training sessions with boarder testing data sets varying the number of hidden layers toward finding lower overfitting rate.

Our 2 MLP networks have the same amount of input neurons, 72. The neurons are extracted from 4 metamodels: MySQL, KM3, UML, and Java. The extraction process is automatically executed by a script written in Scala language. The input metamodels used are 3rd party metamodels, available in the ATL transformations web site[6]. The metamodels are first translated into JSON, then translated into the network compatible format, as shown in sections 2.2 and 2.4. The training set is composed by automatically generated JSON documents, in this case with elements names extracted from a unique given metamodel, but with a random distribution of the generated elements, i.e., the documents may have different number of classes, attributes or references. We generated 20 different documents for each one of the 4 input metamodels.

Input Documents: we develop a script in Scala to generate the input documents to be classified. The generated documents are different from the training set. They are generated according to two criteria: first, the number of elements: we produce documents with 50 and with 100 elements.

Second, we vary the degree of conformance of the produced documents to evaluate the MLP precision. This means we first automatically generate documents with 50 and 100 instances where all the elements' names are equals to the ones existing in MySQL, KM3, UML or Java metamodels. This means it is not a strict conformance relation, but just to generate JSON elements with a given name. In this first case, we want to check the classification of documents which are 100 percent in conformance with existing metamodels. Then, we generate elements extracted from classes, attributes and references mixed between different metamodels, using the following ratios: 80%-20%, 60%-40%, and 50%-50%. This means we generate documents with 80% of conformance with a given metamodel and 20% to a second one. Then, 60% conforming to a given metamodel and 40% to a second one. Finally, we used a 50-50 ratio. The goal of these distributions is to verify the classification precision when varying the number of elements conforming to a given metamodel. The result of MLP classification is shown in Tables 2 and 3.

[6] https://www.eclipse.org/atl/atlTransformations/

Table 2: MLP classifier with 3 hidden layers.

Evaluating MLP with 3 Hidden Layers				
Models with 50 elements				
%	**MySQL**	**KM3**	**UML**	**Java**
100%	100%	100%	100%	100%
Models mixed				
%	**MySQL + KM3**		**UML + Java**	
80%-20%	96,3%		94,3%	
60%-40%	84,5%		83,2%	
50%-50%	47,2%		45,6%	
Models with 100 elements				
%	**MySQL + KM3**		**UML + Java**	
80%-20%	96,1%		93,9%	
60%-40%	82,7%		86,4%	
50%-50%	46,7%		45,2%	

Table 3: MLP classifier with 5 hidden layers.

Evaluating MLP with 5 Hidden Layers				
Models with 50 elements				
%	**MySQL**	**KM3**	**UML**	**Java**
100%	100%	100%	100%	100%
Models mixed				
%	**MySQL + KM3**		**UML + Java**	
80%-20%	97,2%		96,6%	
60%-40%	87,3%		85,6%	
50%-50%	48,6%		47,3%	
Models with 100 elements				
%	**MySQL + KM3**		**UML + Java**	
80%-20%	97,6%		95,1%	
60%-40%	83,8%		87,6%	
50%-50%	47,7%		46,8%	

3.1 Discussions

In this section we discuss the results our our solution, with respect to the precision of the classifier, the feature encoding technique and the variety of the input models.

Precision of the Classifier. Our objective is to test the applicability and accuracy of the MLP classifier to perform metamodel classification. From the results shown in tables 2 and 3, it is possible to see that when documents are produced with elements 100% according to their respective metamodel, the MLP classifier correctly classifies all the documents. This happens because the MLP is trained with all the elements from the metamodels, and in this case, when a document is perfectly in accordance with a metamodel, the MLP classifier is accurate. Thus, for both three hidden layers and five hidden layers, the precision is 100%.

When the elements are mixed, for instance, we mixed 80% of elements from MySQL with 20% of elements from KM3, the MLP precision decreases, but the precision rates remain high, i.e., classifying the documents according to the predominant number of elements conforming to a given metamodel. The MLP with three hidden layers showed 96,3% of precision, and the MLP with five hidden layers showed 97,2%, improving 0,9%. This means adding two layers had a small impact on the final result when the documents are very alike.

When we increase the number of elements, from documents with 50 elements to models with 100 elements, the precision is slightly lower, from 96,3% to 96,1% into the MLP with three hidden layers; however the MLP with five hidden layers improved the result, from 97,2% to 97,6%, showing a better fit for this case.

Now, when we mix 80% of elements from UML with 20% of elements from Java in a document with 50 elements, the MLP with three hidden layers showed 94,3% of accuracy rate, and the MLP with five hidden layers showed 96,6%, improving by 2,3%. The slightly lower result compared to MySQL and KM3 may be explained because UML and Java have some similar elements that could overlap. But the improvement from 3 to 5 layers is better, meaning that it is a valid setting if models have a more variable structure.

When we mix 60% of elements from MySQL with 40% of elements from KM3, the MLP precision decreases, showing 84,5% into the MLP with three hidden layers, but improving 2,8% to 87,3% into the MLP with five hidden layers, which is a relevant improvement. This means that even with only 60% of elements from a MySQL, the MLP classifier performs well, demonstrating that it can be used in document or model recognition solutions.

Finally, when we mix 50% of elements from MySQL with 50% of elements from KM3, the MLP classifier precision is close to 50%. This result is expected because it could simulate a random test, as we have 50% of elements from a *x* document and 50% of elements from a *y* document, thus the MLP classifier could classify as being the document *x*, sometimes as being the document *y*. We intend to expand these experiments varying the number of hidden layers and the neuron number in each hidden layer aiming to improve the MLP classifier accuracy.

Feature Encoding. The decision of implementing a direct feature extraction technique enables to develop a simple extractor, without encoding relationships be-

tween the metamodel elements. Usually, numeric format data gives better performance for classification, regression and clustering algorithms. In addition, we choose to not make a distinction if the metamodel element is a class, reference or attribute, i.e., they are just metamodel elements with a name. When having metamodels as input, the number of input features remains manageable and the solution can be adapted or reused in other scenarios. It is important to note that the one-hot approach has some challenges as well: the sparseness of the transformed data and that the distinct values of an attribute are not always known in advance. However, in our OHE solution we mitigate theses problems by implementing a direct feature extraction as shown in Algorithm 1.

Other approaches for classifying the document could be used, for instance, adapting schema matching or clustering-based approaches for a metamodel classification. However, our simple encoding scheme showed good results. If the number of features becomes too high, other extraction schemes need to be studied and compared, such as strictly structured-based methods. This aspect could be critical if using, in addition to the metamodels, documents or model elements to encode the input features.

Models Variety. We have evaluated the classifier with models from similar domains and mixing the model elements between them, but other random model elements could be used for testing as well. We chose to automatically generate input metamodels with explicit variation on conformance rates of the input characteristics, to be able to analyze the solution under distinct limits. With this validation done, a future work will be to apply it it in real world scenarios, for instance, to classify metamodels in existing Git repositories. However, we cannot affirm if such repositories would be enough to validate explicit conformance rates. In addition, in such cases, we could do additional pre-processing of the elements, or to use it in conjunction with string similarity algorithms.

To summarize, these experimental results show that we can use neural networks to help us for document classification, with a simple encoding scheme. We intend to extend this approach about model classifying to other neural networks algorithms, such as the Long Short-Term Memory Neural Networks (LSTM), and make precision comparisons.

4 RELATED WORK

There has been extensive works on how to classify different kinds of data, such as text, images or structured models. More recently, the vision paper from paper (Cabot et al., 2017) suggests the application of *Cognification* into Model-Driven Software Engineering (MDSE), which is the application of knowledge to boost the performance and impact of a process. In this context, the paper from Xie (Xie, 2018) discusses recent research and future directions in the field of intelligent software engineering, exploiting the synergy between AI and software engineering, and showing that the field of intelligent software engineering is a research field spanning at least the research communities of software engineering and AI. Several initiatives aim to cognify specific tasks within the MDSE ecosystem, for instance, using machine learning (ML) for requirements prioritization (Perini et al., 2013). A very recent work from (Burgueño, 2019) deals with model transformation problems and relies on a ML-based framework using a particular type of Artificial Neural Networks (ANNs), Long Short-Term Memory (LSTM) ANNs to derive transformations from sets of input/output models given as input data for the training phase. Another one from (Nguyen et al., 2019) employed Machine Learning techniques for metamodel automated classification implementing a feed-forward neural network where an experimental evaluation over a datasetof 555 metamodels demonstrates that the technique permits to learn from manually classified data and effectively categorize incoming unlabeled data with a considerably high prediction rate. Beside that, there are some works from programming research community which mixing ML and code transformation, for instance, the papers from (Chen et al., 2017) use ANNs to translate code from one programming language to another. Our work is inspired by these ideas to take stock from existing AI solutions and adapt them for unstructured documents classification, such as supervised learning procedure which has two main phases: training and predicting. The subject of model classifying using cognification-based tooling has also been relatively unexplored and this is where we make a contribution.

Our approach could be used to support, for instance, metamodel repositories classifying unstructured models into metamodels. The work from (Basciani et al., 2016) proposes the application of clustering techniques to automatically organize stored metamodels and to provide users with overviews of the application domains covered by the available metamodels. The work from (Chang et al., 2015) explores training of structured prediction model which involves performing several loss-augmented inference steps. It proposes an approximate learning algorithm which accelerates the training processes, using a structured SVM neural network. This scenario inspired us to train our MLP model classifier, howe-

ver, in our approach, we use a MLP neural network, which was trained based on a metamodels elements set, where all elements are well known and the encoding schem is simple. The work in (Zhang and Chen, 2018) deal with the link prediction problem in network-structured data, it presents link prediction based on graph neural network, where it proposes a new method to learn heuristics from local subgraphs using a graph neural network (GNN). A document or a model could be encoded as a graph, but there is no specific treatment for the metamodel elements. An integration of these approaches with our solution could improve the capabilities of the classifier.

5 CONCLUSIONS

We presented an approach for classifying JSON documents into existing metamodels. The solution enables discovering the domain of the JSON documents and to serve as an initial typing scheme. We present the automated steps of the approach, consisting on metamodel extraction into an MLP using a one-hot encoding (OHE) of the elements, network training, translation and classification of the input JSON documents. The extraction algorithm relies on the presence (or not) of the elements in a given input document, since it translated the elements into a binary classification problem. The results have showed that the approach is effective from classifying JSON documents, with precision varying from 46 to 97 percent, depending on the kinds of the elements. We achieved our main goal to show that a domain-specific and simple extraction algorithm can be useful for classifying documents, instead of trying to adapt more complex structured based classification approaches. The results are publicly available for download, as well as the algorithms implemented.

There are several open issues subject for future work, such as testing the extraction algorithm output with other classification algorithms. We also plan to extend the algorithm to cover more complex relationships between model elements and to test if the results can be improved.

REFERENCES

Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., Xin, R., and Zaharia, M. (2015a). Scaling spark in the real world: Performance and usability. *Proc. VLDB Endow.*, 8(12):1840–1843.

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. (2015b). Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA. ACM.

Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., and Pierantonio, A. (2016). Automated clustering of metamodel repositories. In Nurcan, S., Soffer, P., Bajec, M., and Eder, J., editors, *Advanced Information Systems Engineering*, pages 342–358, Cham. Springer International Publishing.

Burgueño, L. (2019). An lstm-based neural network architecture for model transformations. In *IEEE/ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*.

Cabot, J., Clarisó, R., Brambilla, M., and Gérard, S. (2017). Cognifying model-driven software engineering. In Seidl, M. and Zschaler, S., editors, *STAF Workshops*, volume 10748 of *Lecture Notes in Computer Science*, pages 154–160. Springer.

Chang, K.-W., Upadhyay, S., Kundu, G., and Roth, D. (2015). Structural learning with amortized inference. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 2525–2531. AAAI Press.

Chen, X., Liu, C., and Song, D. (2017). Learning neural programs to parse programs.

Kumari, G. V., Rao, G. S., and Rao, B. P. (2018). Lm, rp and gd based ann architecture models for biomedical image compression. *i-manager's Journal on Image Processing*, 5(3).

Nguyen, P., Di Rocco, J., Di Ruscio, D., Pierantonio, A., and Iovino, L. (2019). Automated classification of metamodel repositories: A machine learning approach. In *IEEE/ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*.

Perini, A., Susi, A., and Avesani, P. (2013). A machine learning approach to software requirements prioritization. *IEEE Trans. Softw. Eng.*, 39(4):445–461.

Xie, T. (2018). Intelligent software engineering: Synergy between ai and software engineering. In Feng, X., Müller-Olm, M., and Yang, Z., editors, *Dependable Software Engineering. Theories, Tools, and Applications*, pages 3–7, Cham. Springer International Publishing.

Zhang, M. and Chen, Y. (2018). Link prediction based on graph neural networks. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 5165–5175. Curran Associates, Inc.