Classification of Software Engineering Artifacts Using Machine Learning

Bernd Bruegge, Joern David, Jonas Helming, Maximilian Koegel

Technical University Munich
Institute of Computer Science / I1
D-85748 Garching, Germany
{bruegge,david,helming,koegel}@in.tum.de

ABSTRACT

In this paper, we present an approach to the automatic classification of software artifacts. The classification result can be used as a foundation for software metrics. Our main contribution is to tailor a content-based machine learning method to the processing of software development artifacts. These artifacts are instances of a unified software engineering model and serve as input for a neural network classifier. Project-relevant characteristics are learned by the classifier from the project history.

We evaluate our technique by two classification tasks, which are important for project management. In the first application, we classify artifacts according to their software development activity. Our second application is the automatic classification of the status of action items; that is, to decide whether they are still under examination or already irrelevant. Five-fold cross-validation of both applications resulted in classification accuracies of 80.51% (six categories) and 83.72% (two categories), respectively.

1. INTRODUCTION

Software projects usually produce a variety of artifacts as outcome of different development activities. These artifacts include, for example, use cases for the activity requirements engineering, components for system design, or action items for project management. This is even true for agile methodologies like Scrum [15] or XP [4], which work with artifacts such as user stories or backlog items. For model-based approaches like the *unified requirements model* [5], these artifacts usually consist of a number of attributes; for example, a functional requirement could consist of a name and a description. Additional information is captured in associations between artifacts, e.g. if one requirement is refining another requirement, or an action item is annotated with the object of the represented task. Artifacts are classified according to selected attributes of interest, e.g. the type or the priority of a certain requirement [28], which we call *classification attributes*. These classification attributes can be used as a foundation for metrics and analyses; Mockus [22], for example, uses the attributes *state* and *resolution* of bug reports. Scrum uses burn-down charts, which are calculated based on the attribute *status* of ToDo items.

These approaches rely on the availability and up-to-dateness of the used information, i.e. the classification attributes. However, up-to-date and available classification attributes are not self-evident. This can be the case if the need for a certain classification attribute was not anticipated and the attribute was therefore not captured at creation time. For example, an analysis of the progress of different software engineering activities (e.g. requirements analysis) might be required during an ongoing project, but the captured tasks are not categorized by activities. In this scenario, automatic activity classification is very useful for an ex-post assignment of the originating activities. Furthermore, some attributes might not be entered completely and correctly by users. We observed a significant number of users who are reluctant to close irrelevant tasks. The effort for manually completing and updating required information may exceed the benefit drawn from this information.

Consequently there are approaches in software engineering, which aim at the automatic classification of artifacts (e.g. [31]). We propose a new approach to automatically classify software engineering artifacts. The approach combines the modular recurrent neural network MRNN [10, 11] with the Rational-based Uniform Software Engineering model [33]. This technique offers three major benefits compared to existing approaches: (1) It is capable of handling fuzzy, incomplete and partially incorrect data, which may result from incomplete and inaccurate user input. 2) It provides traceability between software engineering artifacts, which generates additional input information for the classification mechanism and therefore improves the quality of the classification result [23, 20]. 3) It is able to process different types of artifacts within the same classification task. Even more, our approach is highly extensible with respect to additional artifacts. Thus our approach can be reused for classification problems in different projects regarding various types of software artifacts.

To evaluate the feasibility and performance of our approach, we selected two exemplary classification problems, which are motivated and detailed in section 4. The first one is to classify action items according to the development activity in which they were formulated [7]. The second one is to classify the status of action items; that is, the machine

PROMISE '09 Vancouver, Texas Canada

learning engine decides whether they are still under examination or already completed.

The paper is organized as follows: in section 2, we present the RUSE model in detail and explain how it is used by the classification engine. In section 3, we elaborate on the design of the generic machine learning engine called MRNN and describe its training setup for a classification problem. In section 4, we apply our approach to both types of artifact classifications and present the evaluation results. We also compare our machine learning approach with the capability of humans to classify unseen software artifacts with respect to their corresponding activity in section 5.

2. RUSE

This section shortly describes the existing Rational-based Uniform Software Engineering model (RUSE) [33]. Further, we will motivate, why we based our approach on RUSE. RUSE supports distributed software engineering projects in system modeling, collaboration and organization. The key idea of RUSE is the combination of different software engineering models into one unified model. Thereby it is possible to link artifacts from different aspects of a software engineering project what is called "horizontal traceability" [6]. As an example, a task can be linked to the person the task is assigned to as well as the artifact the task deals with. Like many software engineering models, RUSE also provides vertical traceability, which means one can link artifacts of the same model between different levels of abstraction [6]. For example, a person is linked to his team or a requirement is linked to a detailing use case. Both types of traceability provide additional context information about software engineering artifacts. As all the linked artifacts are available in the same model (and repository), it is easy to follow these links and to use the context information as additional input for our approach. Furthermore, all artifacts in RUSE are based on the same meta-model, which enables our generic machine learning approach to flexibly exploit different types of artifacts.

As shown in figure 1, every element in RUSE is a `ModelElement`.

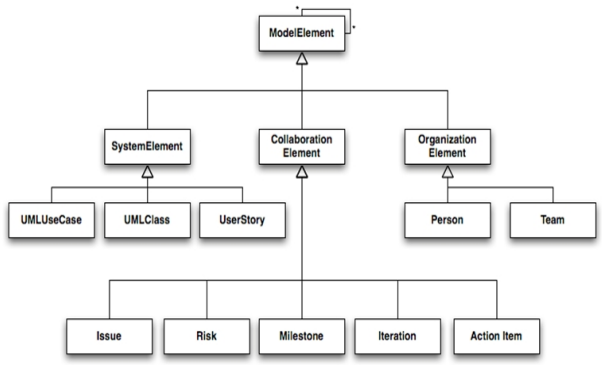

Figure 1: Class diagram showing the Rational-based Uniform Software Engineering model (RUSE).

The RUSE model supports three different categories of `ModelElements`: `SystemElements`, `CollaborationElements`, and `OrganizationElements`. `SystemElements` are used to model the system under construction on different layers of abstraction. This includes requirements, test cases, and the detailed specification of the system. Most of the `SystemElements` are based on the Unified Modeling Language (UML) [1], but RUSE also supports other abstractions such as user stories used in XP [4]. `CollaborationElements` capture the communication and collaboration of users and are mainly based on the QOC model [21]. `CollaborationElements` range from comments, action items and risks to milestones and iterations. `OrganizationElements` describe the organizational structure of a software development project and are used to model organizational units such as teams and participants and their associations.

RUSE is implemented in a tool suite called Sysiphus [15], which provides online and offline collaboration support for distributed development teams. In our case study, Sysiphus was employed as the central project- and CASE-tool for a large student-driven project. Sysiphus also supports a change-based software configuration management (SCM) approach [18]. This approach allows to track every change made on the unified model; and it can further be used to navigate over time in the project, as it was outlined in the PAUSE approach [14]. In other words, this allows to recreate every state of the whole project and to reproduce every applied change. This capability can be used to create input data from past versions of the models' instances for the training of our machine learning approach, as it was conducted in our second case study. Furthermore, Sysiphus supports flexible customizations and extensions of the used model. This means that one can easily add new attributes and associations also called links. In our case study, we added the classification attribute *activity* to the model element `ActionItem`, the value of which is to be entered by the users. This classification attribute served as input (target feature for supervised learning) for the training of our machine learning engine.

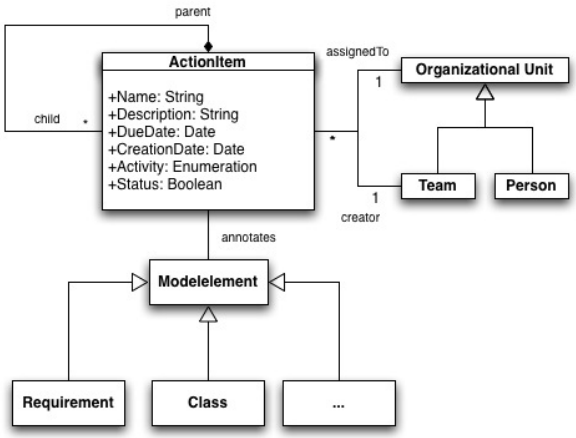

Figure 2: Class diagram showing the detailed `ActionItem` model of the Rational-based Uniform Software Engineering model (RUSE).

One particular type of software artifact is an action item, which represents a task in RUSE. In the following, we will describe which input information of the artifacts of type `ActionItem` was captured in Sysiphus and was therefore used for our evaluation. As shown in figure 2, an `ActionItem` con-

Attribute	Meaning
Name	A short and unique name for the represented task.
Description	A detailed description of the task.
DueDate	The deadline for the completion of the task, if there is one.
CreationDate	The date where the task was created in the system. The attribute is set automatically.
Activity	The *Activity* attribute classifies an `ActionItem` according to the software engineering activity it originated from. In our case the activity could be: -Analysis -System Design -Object Design -Implementation -Testing -Project Management
Status	Determines if the corresponding task is still in progress or already done, that is, *irrelevant*.

Table 1: Description of the non-link attributes of an `ActionItem`.

Attribute	Meaning
parent	Link to a parent task. That is, a task that can be broken down to this child and other children.
child	Link to child tasks.
assignedTo	Link to a person or a team the task is assigned to.
creator	Link to a person or a team, which is the creator of the task.
annotates	Link to the object of the task, e.g. a requirement the task refers to.

Table 2: Description of the links between an `ActionItem` and other `ModelElements`.

sists of six attributes as well as five different links between `ActionItem` and other `ModelElements`. All information except *CreationDate* has to be entered manually by the developer or the project planner. Table 1 shows an overview of the six relevant attributes, table 2 shows the five links of interest. In the following section, we present the classification engine used for all conducted artifact classifications.

3. THE CLASSIFICATION ENGINE

In this paper, we employ a connectionist method for the classification of software artifacts based on their attributes. We use a modular recurrent neural network (MRNN) [11] as enabling technology of our classification approach. Neural networks in general have shown to be capable of handling distorted, incomplete, or erroneous input data. It is not reasonable to use a Naive Bayes classifier for this kind of fuzzy classification task, for example, since one can hardly assume that the attributes are stochastically independent (e.g. *CreationDate* and *DueDate* of an `ActionItem`). Even in the cases, where this is not harmful for the classification accuracy[1]), one does not want to rely on those premises. To substantiate the choice of a connectionist classifier empirically, we present a comparison with alternative classification techniques in section 4.2.

Recurrent neural networks are a subclass of artificial neural networks, which are characterized by recurrent connections between their units. These typically form a directed cycle, while common feed-forward networks do not allow any cycles [8]. The MRNN is able to process different types of artifacts within the same classification task. Furthermore, the MRNN classification engine is robust, since it is capable of handling fuzzy, incomplete and partially incorrect data. Robustness [16] is especially important when processing software engineering knowledge. For example, many instances of the type `ActionItem` created during the software project considered in section 4 lack several attribute values [27]. Only the attribute *Name* is constantly available. Alternative classifiers such as k-nearest-neighbor classifiers (kNN) [9] do not cope well with missing values in the feature vector representation. Finally, the application to software engineering data requires to learn from examples, since the knowledge is not explicitly given in a rule-based or in another declarative form. Thus, logical inference based on an ontology of software engineering concepts (terminological box), for example, is not applicable here. The fundamental data structure that is processed by the MRNN are sequences of arbitrarily dimensional feature vectors [19] that stand for *multi-represented objects* [3]. Multi-representation is a concept to address the manifold contents carried by complex domain objects, that is multi-represented objects capture several aspects of the same domain object. A recent example is the encapsulation of all biometric features of a person like voice pattern, image and finger print in a single multi-represented object. A multi-represented (MR) object is an element of a multi-dimensional feature space: $o = (r_1, \ldots, r_n) \in F_1 \times \ldots \times F_n$, where F_i is a feature that can be weighted by a factor $w_i \in [0, 1]$ additionally. Missing values r_i should principally be allowed in the object representation, since modern machine learning algorithms like the MRNN are able to deal with incomplete data. The allowed feature types are: *Unstructured text*, *Metric* (numerical), *Ordinal* (ordered) and *Categorical* (unordered). Each feature type has to be preprocessed in a different way, with respect to its possible value range and the appropriate data normalization to be applied. For example, each metric feature is individually scaled to the value range $[0, 1]$.

3.1 Recurrent Network Model

The basic design of the recurrent neural network is defined by the following propagation model and is depicted by the schema of figure 3. The vector $\vec{s}_t$ stands for the internal state at the discrete time step t. The therefrom composed state layer is the backbone for learning the input-target sequences and for classifying or predicting symbol sequences. Each neuron record (the block arrows depicted in figure 3) serves

[1]It is harmful, for example, if the different classes only discriminate in the correlations between the different attributes.

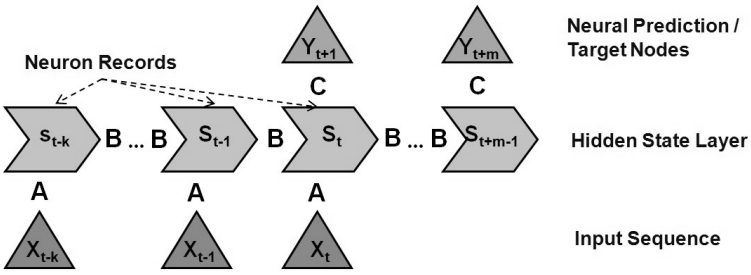

Figure 3: Schematic topology of the proposed modular recurrent network MRNN. The block arrows indicate the internal state transition $\vec{s}_t \rightarrow \vec{s}_{t+1}$. A, B and C are weight matrices. $\vec{x}_t$ is the external input vector at time t, $\vec{y}_{t+m}$ is the correspondingly predicted output vector. For classification as opposed to sequence prediction only one output unit $\vec{y}_{t+1}$ is used. The depicted block arrow direction shows the forward propagation phase.

both as *hidden* and as *context unit*, because $\vec{s}_{t-1}$ provides a context for the recursive computation of the subsequent hidden state $\vec{s}_t$.

- A is a ${}^h\mathbb{R}^{d_1}$, B is a ${}^h\mathbb{R}^h$ and C is a ${}^{d_2}\mathbb{R}^h$ matrix.
- d_1 is the dimensionality of the input- and d_2 is the dimensionality of the output feature space.
- $h = dim(\vec{s}_i)$, $(i = t\text{-}k, \ldots, t{+}m)$ is the dimensionality of the state layer. h is independent from d_1 and d_2 and was set to $h = 15$ (experimentally determined) to provide sufficient network resources.

$$\vec{s}_t = f(B\vec{s}_{t-1} + A\vec{x}_t) \quad (1)$$

$$\vec{o}_{t+1} = f(C\vec{s}_t) \quad (2)$$

$$\vec{o}_{t+i} \overset{training}{\longrightarrow} \vec{y}_{t+i},\ i = 1, \ldots, m \quad (3)$$

The crucial recurrent equation 1 combines an external input $\vec{x}_t$ with the previous state $\vec{s}_{t-1}$ to the subsequent state $\vec{s}_t$, which indirectly depends on all foregoing external inputs $\vec{x}_{t-k}, \ldots, \vec{x}_{t-1}$ *and* internal states $\vec{s}_{t-k}, \ldots, \vec{s}_{t-1}$. In case of supervised network training, the target symbols $\vec{y}_{t+1}, \ldots, \vec{y}_{t+m}$ are known, while in case of actual structure classification the *output sequence* $\vec{o}_{t+1}, \ldots, \vec{o}_{t+m}$ is computed solely based on the respective inputs. Here, the activation function is chosen as sigmoid function $f(x) = \frac{1}{1+\exp(-x)}$. The MRNN is trained with a modified *Backpropagation Through Time (BPTT)* algorithm [8, 30] and is able to process variably dimensional vectors $\vec{x}_{t-k}$ and $\vec{y}_{t+m}$.

The MRNN is able to process variably dimensional vectors as encoding of input and target data, so different object types such as `ActionItems` or Requirements can be associated with the same set of classes during the training phase, for example. In other words, action items and requirements, which are described by different sets of attributes and thus are represented by feature vectors of different size, can be learned according to the same originating activities such as analysis or implementation. In the following section, we describe the mathematical fundamentals of the advanced text representation – namely Latent Semantic Indexing – used as preprocessing of the input information fed into the neural classifier.

3.2 Latent Semantic Indexing of Textual Attributes

Each artifact that contains unstructured text has to be transformed into a numerically processable representation. An example for unstructured text is the "description" attribute of an ActionItem, which can be freely filled in by the user. The content is considered by a text mining approach that provides a rich artifact representation with the semantics of the textual attributes, since a conceptual similarity measure between artifacts is provided thereby.

According to the vector space model [25], a feature vector $\vec{x} \in \mathbb{R}^k$, $k < d_1$, (for d_1 also see section 3.1) is computed for each textual attribute (*bag-of-words* approach). The well-known preprocessing steps *stemming* [24] and *stopword removal* were realized by the *Apache Lucene* indexing and search framework (http://lucene.apache.org.). The vector space model can be refined by applying *Latent Semantic Indexing* [12] to the set of computed feature vectors. Thereby, a matrix $M_{i,j}$ of keyword frequencies per text unit is spanned. Text units are given by descriptions consisting of sentences, paragraphs and sections or they even represent whole documents. The rows of the matrix denote the frequency of occurrence for term i in text unit j. The matrix is decomposed by *Singular Value Decomposition* (SVD), which is a generalization of the *Principal Component Analysis* (PCA) that determines the inherent key concepts that characterize all d-dimensional feature vectors. SVD is able to analyze the correlations among terms as well as the correlations between text units and comprised terms, which are described by a non-quadratic matrix. Thereby, the term-frequency matrix $M = UDW^T$ is decomposed into two orthonormal matrices U and W and one diagonal matrix D. After diagonalizing this matrix M, the singular values $\sigma_j = D_{j,j}$ in the diagonal of the matrix D reveal the insignificant dimensions to be discarded. These k least informative dimensions with singular values $\sigma_{d-k}, \sigma_{d-k+1}, \ldots, \sigma_n$ are ignored by the transformation to a $(d\text{-}k)$-dimensional subspace. The resulting feature vectors $\vec{x}_j \in \mathbb{R}^{d-k}$ represent the content of an artifact $v_j \in V$: $\vec{x}_j^T := (W_{1,j}^T, W_{2,j}^T, \ldots, W_{d-k,j}^T)$, where $W_{i,j}^T$, $i = 1, \ldots, d\text{-}k$, $j = 1, \ldots, |V|$ are the entries of the transposed right-singular matrix. This global vector space model enables the MRNN to learn the latent semantics of domain-specific notions and textual concepts. SVD also covers semantical peculiarities such as synonymy and polysemy. Synonymy is the phenomenon of several distinct words holding the same linguistic meaning. Polysemy is the contrary phenomenon, that is, a single expression has

different meanings in different linguistic contexts.

4. EVALUATION

The developed classification of model-based software development data based on a recurrent neural network is a new approach at the intersection of software engineering and machine learning. To evaluate the feasibility and performance of our approach, we conducted two classification experiments. In both experiments, RUSE model elements of the dedicated type `ActionItem` were used (also see [7]).

The first one is to classify action items according to the activity in which they were formulated. In the second experiment, SymboConn is employed to classify the status of `ActionItems`; that is, the machine learning engine decides whether they are still under examination or already completed.

Our activity- and status-classification approach is independent from the concrete life cycle model. To show its feasibility and performance, we selected a real-world project with a particular life cycle model. This project used the activities *Analysis*, *System Design*, *Object Design*, *Implementation*, *Testing*, and *Project Management*.

The Case Study Project: DOLLI.

As training and evaluation examples for our approach, we used data of a student-driven project which employed Sysiphus as central project repository. The Dolli project (Distributed Online Logistics and Location Infrastructure) [2], a large project (approx. 50 students), was carried out as a cooperation between the Technical University of Munich (TUM) and the Munich Airport (FMG). The objective of Dolli was to improve the airport's existing tracking and locating capabilities and to integrate all available location data into a central database; thereby, luggage tracking and dispatching of service personal should be supported as well as a 3D visualization of the aggregated data was to be implemented. The project duration was one semester. The students worked on the project for more than five months, partly full-time. They were organized in eleven sub-teams and their efforts resulted in a comprehensive project model consisting of about 15.000 model elements. We used this project model for our evaluation.

4.1 Activity Classification of ActionItems

The logistics project DOLLI first followed a variant of a Unified Process for 4 months. The development activities analysis, system design, object design, implementation and testing as well as project management were executed sequentially in the first part of the project. For the remainder of the project, the developers followed the Scrum methodology [15]. All `ActionItems` created during the project were manually classified into the predefined development activities.

XX kill[This information was mainly used for research purposes. Part of the research was to find out, at what time the teams worked on which activities.] Figure 4 illustrates the development of open `ActionItems` over time and per activity. The interesting result was that although a sequential lifecycle model was planned, none of the activities were really finished with the beginning of the subsequent activity. Also it can be observed, that in the Scrum-oriented phase at the end of the project, there was a significant rise of analysis- and implementation-related `ActionItems`. Table 3 shows the distribution of `ActionItems`. Note that the

Activity	Number of `ActionItems`
Analysis	258
System Design	158
Object Design	58
Implementation	202
Testing	7
Project Management	1
Total	**684**

Table 3: Distribution of `ActionItems` according to the activity they were assigned to by the project participants. The activities testing and project management were rarely assigned. We assume that project management tasks were mostly managed in the team wikis, whereas comprehensive testing was not in the scope of the DOLLI project.

`ActionItems` were rarely classified as belonging to *Testing* and *Project Management* activities. In fact, the activities are not normally distributed. We formally compared the given empirical distribution with a theoretical normal distribution using the Chi-Square test. The `ActionItem` significantly differs from a normal distribution $\mathcal{N}(\mu, \sigma^2)$ with parameters $\mu = 2.33$ and $\sigma^2 = 1.65$.

As mentioned, the `ActionItems` were classified manually by the project participants. However, during the project, the requirement for an automatic classification arose for three reasons: (1) The manual input of the activity attribute was intrusive for the project participants as they had no obvious benefit from entering this information. It was necessary to continuously motivate the developers to fill out the activity attribute[2]. (2) `ActionItems` arising in meetings also need to be automatically classified according to their activity. This is especially important for the automatic capture of `ActionItems` using word spotting techniques. (3) In our case study, we started to work in a sequential- and activity-oriented approach and then switched to a more agile and Scrum-oriented process. In future projects, we plan to perform such a process shift also in converse order. Hence, when following an agile process, the `ActionItems` will not be classified by their activity. An automatic classification would help to retrospectively add this information whenever the process is turned into an activity-oriented process.

For an automatic classification, we trained the neural classifier with data from the DOLLI project. The following paragraph describes the technical setup of the training patterns learned by the neural network and reports on the results of the activity classification.

Training Data Representation.

For each `ActionItem`, 13 attributes are captured in the Sysiphus tool (see section 2) at the time of creation within a certain development activity. These are *Name, Team, Activity, State, DueDate, OrganizationalUnit, Description, ParentActionItem, ChildActionItems, Annotatables, URLElements, Attachments* and *CreationDate*. The attributes of the `ActionItems` to be classified by our neural classifier are

[2]This is a common issue of research approaches, which need additional information to be captured.

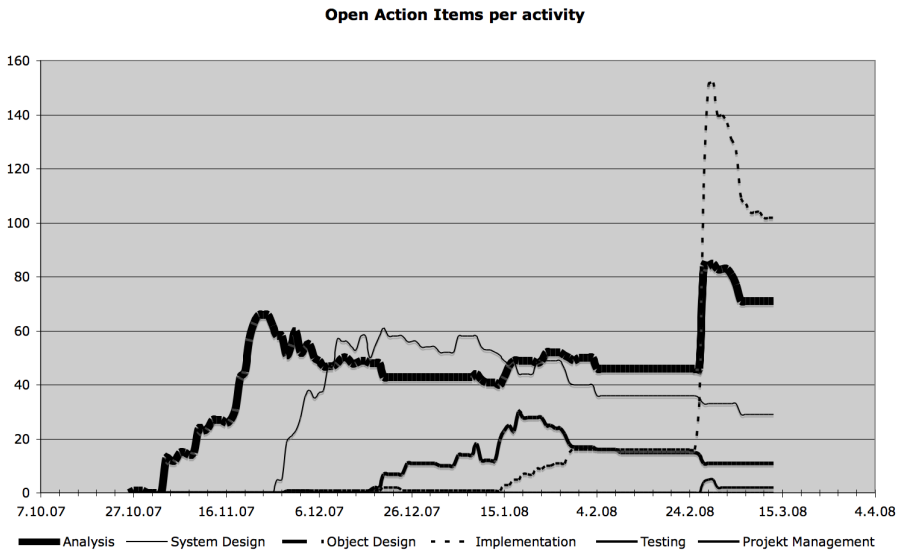

Figure 4: Fraction of open `ActionItems` with respect to the total number of `ActionItems` managed in the DOLLI project in percent [%], broken down by activity. Beginning from February 24th, the Scrum-oriented phase reveals itself by a momentary peak in the relative number of open `ActionItems` – especially in the activities Analysis and Implementation.

described by 26 to 2536 feature dimensions, depending on the respective representation technique. As an example, a minimal training pattern with only two input attributes results in the following straight-forward feature vector representation.

$$\vec{x}_t = (\underbrace{x_1, \ldots, x_{k_1}}_{Team}, \underbrace{x_{k_1+1}, \ldots, x_{k_2}}_{CreationDate}) \mapsto \underbrace{(a_1, \ldots, a_6)}_{\vec{y}_{t+1}}, \quad (4)$$

where $x_i \in \mathbb{R}$, $a_j \in \{0, 1\}$. Categorical attributes such as Team or OrganizationalUnit are encoded in a unary form, that is, each symbol to be encoded is assigned to an orthogonal bit vector with a "1" at the i^{th} component, $(0, 0, \ldots, 0, 1, 0, \ldots, 0)$. Numerical (metric) attributes such as CreationDate (relative point of time with respect to the beginning of the project measured in days) are assigned to a fixed-width intercept of the whole feature vector, for example, a numerical value is scaled to the range $[0, 1]$ and the respective value is replicated 10 times (as often as the width of the other attributes' representations). Even if a single feature dimension would suffice to represent a numerical value, due to balance reasons, the value is replicated in order to achieve the same weight than other types of represented attributes (e.g. categorical).

The training and test patterns both hold the form *input* $\mapsto$ *target* of expression 4, while the test patterns were excluded from MRNN training. Five-fold cross-validation was used to obtain significant accuracy measurements, therefore the 684 objects were divided into 5 disjoint test sets. The common measures precision and recall to assess the quality of a classifier were computed according to the following formulas:

$$recall_i = \frac{|\{o \in C_i | K(o) = C(o)\}|}{|C_i|} = \quad (5)$$

$$precision_i = \frac{|\{o \in K_i | K(o) = C(o)\}|}{|K_i|} \quad (6)$$

C_i, $i = 1, \ldots, r$ is i^{th} class out of the set of classes $C = \{C_1, \ldots, C_r\}$ and K_i is the set of objects that were predicted to belong to class C_i, no matter if this is true or not. $K(o) \in C$ is the classification of object o predicted by the machine learning engine. The classification $K(o)$ of unseen objects is compared with their actual class membership $C(o)$. The precision and recall values are weighted with the size $\alpha_i := |C_i|$ of each class by a weighted sum $precision = \sum_{i=1}^{r} \frac{\alpha_i}{n} \cdot precision_i$, $n = \sum_{i=1}^{r} \alpha_i$, analogously for the *recall*.

Table 4.1 shows the results of the evaluation process, which depend on the choice of attributes included in the training process (column *Input Attributes*). Instead of applying a feature selection algorithm, the input attributes were chosen according to the realities in the application domain, which is software engineering. Therefore a focal point was the ability to exploit textual attributes besides categorical ones such as *Team*.

During the training phase, the machine learning engine is faced with incomplete data, for example, the values of the *DueDate* attribute are missing in 68.44% of the training examples. However, the connectionist machine learning unit is capable of handling fuzzy, incomplete and partially incorrect data. The MRNN coped with the incomplete attribute values, as shown by the evaluation variants 1 and 2. Due to the additional *DueDate* attribute in variant 2, the accuracy could be slightly improved and stabilized (smaller variation) – despite the majority of missing *DueDate* values.

In case of variant 2, the training error cannot be reduced as much as for variant 3 or 4, since the mapping from the input attributes to the classification attribute *activity* is less unique than in the other cases. This is due to the lower discrimination provided by the attributes *Team* (categorical), *DueDate* and *CreationDate* (both numerical), which do not always uniquely determine the activity an artifact originated from. Since the neural network realizes a functional mapping, the training error does not vanish completely.

Variant	Input Attributes	LSI	Measure	Precision	Recall	F-Measure	Δ
1	Team, CreationDate	–	**Mean**	75.77	76.40	76.08	8.70
			Variation	[70.74 - 80.46]	[72.55 - 81.19]	[72.11 - 80.24]	
2	Team, DueDate, CreationDate	–	**Mean**	76.83	77.37	77.10	7.48
			Variation	[75.11 - 78.93]	[74.51 - 79.21]	[74.81 - 79.07]	
3	All except *Activity*, *Team*, *DueDate*, *CreationDate*	No	**Mean**	54.15	54.43	54.28	0.85
			Variation	[45.87 - 61.21]	[47.06 - 62.50]	[46.47 - 61.85]	
4	All except *Activity*, *Team*, *DueDate*, *CreationDate*	Yes[1]	**Mean**	48.71	52.37	50.47	10.27
			Variation	[42.67 - 54.46]	[46.32 - 58.52]	[44.42 - 56.42]	
5	All except *Activity*	No	**Mean**	75.88	76.85	76.35	3.39
			Variation	[69.54 - 80.98]	[71.32 - 81.62]	[70.42 - 81.30]	
6	All except *Activity*	Yes[2]	**Mean**	**80.64**	**80.38**	**80.51**	7.60
			Variation	[78.03 - 84.28]	[76.47 - 83.70]	[77.24 - 83.99]	

[1] LSI using $\sigma^2 = 0.65$.
[2] LSI using $\sigma^2 = 0.35$.

Table 4: Average classification accuracy measured in terms of *Precision* and *Recall* for the 684 `ActionItems` after 5-fold cross-validation. The measure *Variation* indicates the range of the obtained accuracy values over the 5 individual test sets used for cross-validation. For all predictions, the network was trained till a residual error of Δ. The F-Measure is a weighted mean of Precision and Recall: $F\text{-}Measure = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$. If Latent Semantic Indexing (LSI) was used (only applicable in case of textual attributes), 35% and 65% of the variance σ^2 in the training set were kept respectively (via discarding the dimensions corresponding to the 35% or 65% smallest eigenvalues respectively). All values are given in percent [%], best accuracy is in bold.

For activity classification, *overfitting* [17] occurs very early when using textual attributes, such that the training procedure has to be stopped quite early. For example, reducing the training error to an amount of 3.14% leads to a 4.8% lower classification accuracy on unseen objects (generalization) than accepting a training error of 7.93% in case of variant 4.

We see that the attribute *CreationDate* is highly significant for the classification of the activity that produced the respective artifact. The sound classification result when using the attributes *CreationDate* and *Team* (variants 1,2,5, and 6) shows that certain periods of time in the project existed, in which certain teams worked in a specific activity.

The evaluation variants 3 and 4 showed that it is even possible to classify `ActionItems` without having available any time-related information, even though the results are unproportionally less accurate. Surprisingly, in this case, variant 4 using the advanced text representation LSI provides a slightly lower classification accuracy ($\delta = 1.74$) than variant 3 without using LSI and is considered to represent a statistical outlier. One reason for this might be the significantly higher training error Δ of variant 4, which is a sign of the less unique input-target (class) mapping that was more difficult to learn. The effect of hindered training progress in the case of latent semantic indexing is due to the lossy transformation ($\sigma^2 = 0.65$) of the input information, which discards both redundant information and information used to distinguish the artifact class. Normally, the improved representation (less redundant and more compressed) overcompensates the negative effect of loosing information which is usable for the class distinction.

4.2 Classification of the Artifact Status

All `ActionItems` in the DOLLI project were classified manually according to their status, which is either open or closed. Figure 5 shows the distribution of open `ActionItems` over time. After the above-mentioned process shift from a traditional sequentially oriented software lifecycle model to Scrum-oriented methods, we observed a large number of `ActionItems` which were neither touched (read or changed) nor closed until the end of the project. A survey among the project participants revealed that 81% of these `ActionItems` were either irrelevant or were attached to a task which was already closed. This implies that the respective `Action-`

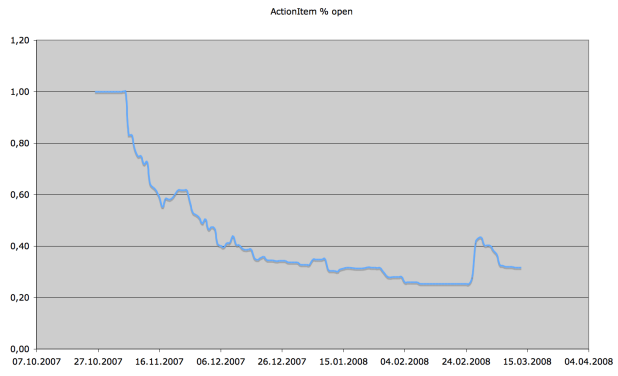

Figure 5: Fraction of open `ActionItems` with respect to the total number of `ActionItems` managed in the DOLLI project in percent [%]. Beginning from February 24th, the scrum oriented phase is revealed by a momentary peak in the relative number of open `ActionItems`.

`Items` should have been closed, too. Again, an automatic classification approach could support the user to mark these objects as irrelevant. In contrast to the classification of the artifact activity, the classification according to the status would have a visible benefit to the project participants: the mechanism determines `ActionItems`, which are most likely irrelevant, and recommends to close or to delete these items.

Variant	Input Attributes	LSI	Measure	Precision	Recall	F-Measure	Δ
1	Team, Activity, CreationDate	–	**Mean**	74.30	71.20	72.65	22.01
			Variation	[65.09 - 83.33]	[58.33 - 83.33]	[61.53 - 83.33]	
2	Team, Activity, CreationDate, DueDate	–	**Mean**	72.06	68.66	70.27	19.41
			Variation	[59.35 - 84.72]	[54.17 - 83.33]	[56.64 - 84.02]	
3	All Attributes	No	**Mean**	78.18	75.36	76.74	12.33
			Variation	[69.63 - 86.67]	[66.67 - 83.33]	[68.12 - 84.97]	
4	All Attributes	Yes[1]	**Mean**	81.13	80.47	80.79	22.87
			Variation	[67.71 - 89.47]	[70.83 - 87.50]	[69.24 - 88.48]	
5	All Attributes except *DueDate*	Yes[2]	**Mean**	**85.49**	**82.14**	**83.72**	16.31
			Variation	[78.19 - 92.71]	[73.91 - 91.67]	[75.99 - 92.18]	

[1] LSI using $\sigma^2 = 0.65$.
[2] LSI using $\sigma^2 = 0.85$.

Table 5: Average accuracy for the classification of the artifact status *irrelevant* after 5-fold cross-validation. All values are given in percent [%], best accuracy is in bold.

To gather trainings cases for our machine learning approach post-mortem, we used the change-based SCM approached [18] to recreate the state of the DOLLI project before the process shift was done. As training set, we chose `ActionItems` which were not yet closed at that time. Subsequently, we determined which `ActionItems` had not been closed until the end of the project. Under the assumption that all of the chosen `ActionItems` which had not been closed at the end of the project were irrelevant, we accordingly set the classification attribute *irrelevant* of each `ActionItem`. As opposed to the activity classification with a target space consisting of six classes, the binary classification of the artifact status explicitly considers the development of an artifact over time, that is, the individual artifact history or life cycle.

For the classification of the artifact status, we again used a network topology with a hidden layer dimension of $h = 30$. Compared to the multi-class activity classification, the classification of the `ActionItem` status even shows a higher tendency to overfitting. To avoid an overadaption with respect to the training set, for each cross-validation we used a small auxiliary test set during training to check the current quality of the classification model. When the classification error on this test set started to rise, the training process was cut off. This is the reason for the higher level of the residual training error denoted by Δ.

Compared to the first scenario, the status classification should only be used to provide the user with a recommendation, as it is not acceptable to wrongly classify an `ActionItem` as irrelevant (alpha error). For the purpose of recommendation, the given precision is sufficient to effectively support project participants. As shown in table 5, LSI had a positive effect on the classification accuracy in this second scenario, which is demonstrated by variant three and four.

The most interesting observation we made during the evaluation was the role of the attribute *DueDate*. One could expect that a defined and prompt *DueDate* would tendentiously led to `ActionItems` which are not irrelevant and the other way round. However, we found out that the attribute *DueDate* had even a negative effect on the precision. To discover the reason behind that anomaly, we trained the classification engine only with the *DueDate* information without any further attributes. Due to the missing values of the *DueDate* attribute in 54.24% of the training cases used for status classification, the training error stagnated on a high level of about $\Delta = 24\%$, because of the lack of information that could be used to distinguish and classify the respective `ActionItems`. This fact alone is not the reason for the repressed classification accuracy, since the neural classifier can deal with incomplete information, if further attributes are available. In fact, the problem was that contradicting time information was imposed by the *DueDate* attribute. This means, for example, that there are 9 `ActionItems` which have the same *DueDate* value of 80 days since the first day of the project, but which are in the one case *irrelevant* (5), while in the other case not (4). Of course, such ambiguous information is misleading and counterproductive for the classification (non-dichotomous or non-disjoint `ActionItem` distribution). It argues for the robustness of the classification engine, that the classification accuracy is not even more distorted by the *DueDate* attribute. One might raise the objection that decision-trees would have immediately explained the ambiguousness of the *DueDate* attribute, but decision-tree based classifiers such as the `RandomForest` used in the following paragraph showed a performance on or even below average, especially when high-dimensional and sparse bag-of-words vectors (textual content) have to be exploited for the classification.

Comparison with Competing Classification Techniques.

In order to compare the performance of our classifier with alternative classification techniques, we applied several established classifiers to the problem of relevancy classification. Amongst others, we trained a support vector machine (SVM)[3] classifier on the same training set. The results for different input attributes are listed in table 6, which correspond to the different input variants shown above in table 5. Five-fold cross-validation resulted in a classification accuracy of 69.45% for the SVM, which means that the SVM approach falls short by more than 14% compared to the performance of the MRNN classifier in case of the variant *All Attributes except DueDate*. The MRNN classifier wins all comparisons except for the input variant *Team, Activity, CreationDate, DueDate*, where the SVM classifier shows an

[3]The SVM classifier (SMO) from the *Weka* data mining package was used [32].

advantage of about 3% in terms of the F-measure.

Furthermore the comparison underpins the difficulty of reliable classification of software development artifacts when these are afflicted with incompleteness and noise, since no classifier was able to break through an accuracy of 84% in terms of the F-measure – no matter which input attributes were used. The benefit of our recurrent neural network be-

Input / Classifier	Accuracy
Team, Activity, CreationDate	
MRNN	**73.90**
SVM	73.70
RandomCommittee	70.25
RandomForest	69.05
Bayes Net	65.05
MultiBoostAB	63.60
Team, Activity, CreationDate, DueDate	
SVM	**73.60**
MRNN	70.27
RandomForest	68.35
RandomCommittee	66.00
Bayes Net	65.05
MultiBoostAB	63.60
All Attributes	
MRNN	**76.74**
SVM	68.55
RandomCommittee	65.95
RandomForest	65.30
Bayes Net	65.05
MultiBoostAB	64.50
All Attributes,	
with Latent Semantic Indexing ($\sigma^2 = 0.85$)	
MRNN	**80.79**
SVM	67.65
Bayes Net	62.05
RandomCommittee	59.30
RandomForest	57.30
MultiBoostAB	56.70
All Attributes except DueDate,	
with Latent Semantic Indexing ($\sigma^2 = 0.85$)	
MRNN	**83.72**
SVM	69.45
Bayes Net	62.05
MultiBoostAB	56.70
RandomForest	56.50
RandomCommittee	54.00

Table 6: Comparison of the classification accuracy (F-measure) of the MRNN classifier with different classification and meta-classification techniques from the Weka data mining package such as Support Vector Machines (SVM) or `MultiBoostAB`. The action items were classified according to their relevancy, which was evaluated by 5-fold cross-validation.

comes apparent when classifying action items that are represented by raw or refined (by LSI) textual content. In the case of the input variant *All Attributes except DueDate with LSI* in table 5, the MRNN has a mean advantage of almost 24% compared to the average of all other classifiers. This is why we have chose the connectionist MRNN classifier, which outperforms the other classification techniques especially in the case of exploiting textual representations.

5. BETTER THAN GUESSING?

In the previous section, we evaluated the performance of the machine learning system, which is considerably high. But of which practical quality is this achievement, or in other words, how difficult is the classification task for humans? To figure this out, we conducted an experiment with three persons with different degrees of expertise in the DOLLI project, who should classify `ActionItems` according to their activity by hand.

The *Informed Outsider* knew the RUSE model and the information about the DOLLI project provided in this paper. The *Knowledgeable Observer* worked part-time in the DOLLI project as a teaching assistant. The *Expert* played a central role as an active project participant in DOLLI. We chose a layered single sample (random selection in groups, the class distribution of the objects in the sample is proportional to that in the basic population) of $n = 70$ `ActionItems` from the basic population of $N = 684$ `ActionItems` to obtain significant results. Since the actual distribution of the action items is not *normal*, we cannot use a standard sample size calculator. XX, that is, the minimal sample size of $n_{min} \geq 30$ was met [29, 13].As shown in table 7, the

Expertise	Precision	Recall	F-Measure
Informed Outsider	38.07	32.86	35.27
Knowledgeable Obs.	50.17	41.43	45.38
Expert	61.35	51.43	55.95

Table 7: Evaluation of the ability of humans to classify `ActionItems`. Three persons with different degrees of expertise and insight into the software project were compared.

quality of the classification significantly increases with the project-related expertise of the interviewee. Nevertheless, even the *Expert* was by far not able to match the classification accuracy of the machine learning system.

6. CONCLUSION AND FUTURE RESEARCH

We developed a combined approach of a unified software engineering model (RUSE) and a neural classifier called MRNN aiming at software artifacts as work products of software development activities. In general, heterogeneous sequences consisting of objects of different types, or in other words, from different classes, can be learned by the recurrent neural network. All in all, this paper provides two major contributions:

1. ***Combination of a Unified Software Engineering Model with a Machine Learning Engine***
   We combined two different technologies from software engineering and connectionist artificial intelligence to obtain an intelligent classification mechanism specialized in processing of heterogeneous knowledge.

2. ***Automatic Classification of Software Artifacts***
   We demonstrated that complex software artifacts can be classified by the neural classifier. The employed neural network is also capable of handling fuzzy, incomplete and partially incorrect data. Our new technique was successfully applied to the classification of software artifacts according to development activities,

whose instances should be classified based on 13 domain-specific attributes.

There are further possibilities of exploiting software development knowledge based on our machine learning technique; we wish to mention two examples. The first is the automatic classification of quality attributes namely quality attributes of requirements. Approaches as proposed by Wilson and his colleagues [31] aim at the classification of requirements according to specific quality attributes in order to calculate quality metrics, which express the quality of captured requirements. We expect that our approach is able to solve such a classification problem very effectively.

Another example is the prediction of *burn-down charts*, which represent a method from the agile project management methodology *Scrum* [26]. Burn-down charts reflect the project progress and the features to be implemented for each planned release. The prediction of the remaining implementation time can answer the question of whether the planned release will be on time and whether it will meet the functionality and quality requirements of the client. The high precision in the status classification including temporal information leads us to the assumption that even burn-down charts can be predicted with our approach.

7. REFERENCES

[1] Omg unified modeling language specification version 2.0. 2004.

[2] Distributed online logistics and location infrastructure (dolli), http://www1.in.tum.de/static/dolli/, 2007.

[3] E. Achtert, H.-P. Kriegel, A. Pryakhin, and M. Schubert. Hierarchical density-based clustering for multi-represented objects. In *Workshop on Mining Complex Data (MCD'05), ICDM, Houston, TX*. Institute for Computer Science, University of Munich, 2005.

[4] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.

[5] B. Berenbach and T. Wolf. A unified requirements model; integrating features, use cases, requirements, requirements analysis and hazard analysis. In *Second IEEE International Conference on Global Software Engineering, ICGSE 2007*, pages 197–203, 2007.

[6] B. Bruegge, O. Creighton, J. Helming, and M. Kögel. Unicase Ű an ecosystem for unified software engineering research tools. In *Third IEEE International Conference on Global Software Engineering, ICGSE 2008, Bangalore, India*, 2008.

[7] B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, ISBN 0-13-0471100, 2004.

[8] R. Callan. *Neuronale Netze im Klartext*. Pearson Studium, 2003.

[9] J.-H. Chen, H.-M. Chen, and S.-Y. Ho. Design of nearest neighbor classifiers using an intelligent multi-objective evolutionary algorithm. In *Pattern Recognition Letters*, volume 23, pages 1495 – 1503. Elsevier Science Inc., New York, NY, USA, 2002.

[10] J. David. Navigation recommendation on knowledge artifacts. In *Workshop "Agile Knowledge Sharing for Distributed Software Teams", Lecture Notes in Informatics*. Springer, 2008.

[11] J. David. Recommending software artifacts from repository transactions. In *The Twenty First International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE 2008), LNAI 5027*, pages 189–198. Springer-Verlag Berlin Heidelberg, 2008.

[12] S. Deerwester, S. T. Dumais, G. W. Furnas, R. Harshman, and T. K. Landauer. Indexing by latent semantic analysis. volume 41, pages 391–407, 1990.

[13] J. Hartung, B. Elpelt, and K.-H. Klösener. *Statistik, 12. Auflage*. Oldenbourg, 1999.

[14] J. Helming, M. Koegel, and H. Naughton. Pause: A project analyzer for a unified software engineering environment. In *In Workshop Proceedings of ICGSE 2008, Bangalore, India*. IEEE CS Press, 2008.

[15] http://www.scrumalliance.org. Scrum alliance. 2007.

[16] A. Katz, M. Gately, and D. Collins. Robust classifiers without robust features. In *Pattern Recognition Letters*, volume 2, pages 472–479. Neural Computation, 1990.

[17] T. M. Khoshgoftaar, E. B. Allen, and J. Deng. Controlling overfitting in software quality models: Experiments with regression trees and classification. In *Seventh International Software Metrics Symposium (METRICS'01)*, 2001.

[18] M. Koegel. Towards software configuration management for unified models. In *ICSE CVSM'08 Workshop Proceedings*, pages 19–24, 2008.

[19] D. D. Lewis. Representation and learning in information retrieval. Technical report, University of Massachusetts, 1992.

[20] H. Liu and H. Motoda. Feature selection for knowledge discovery and data mining. In *The Springer International Series in Engineering and Computer Science*, volume 454, pages 1226–1238, 1998.

[21] A. MacLean, R. M. Young, V. M. Bellotti, and T. P. Moran. Questions, options, and criteria: Elements of design space analysis. In *HCI*, volume 6, pages 201–250, 1991.

[22] A. Mockus. Missing data in software engineering. In *Guide to Advanced Empirical Software Engineering*, pages 185–200, 2008.

[23] H. Peng, F. Long, and C. Ding. Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 27, pages 1226–1238, 2005.

[24] M. Porter. An algorithm for suffix stripping. Technical Report 3, 1980.

[25] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. In *Communications of the ACM*, volume 18, page 613Ű620, 1975.

[26] K. Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004.

[27] M. S. B. Sehgal, I. Gondal, and L. S. Dooley. Collateral missing value imputation: a new robust missing value estimation algorithm for microarray data. volume 21, pages 2417–2423. Bioinformatics, 2005.

[28] J. Singer, S. Sim, and T. Lethbridge. Software

engineering data collection for field studies. In *Guide to Advanced Empirical Software Engineering*, pages 9–34, 2008.

[29] P. M. von der Lippe. Grenzwertsätze, gesetze der großen zahl(en). Technical report, 2008. Induktive Statistik.

[30] P. Werbos. Backpropagation through time: what it does and how to do it. In *Proceedings of the IEEE*, volume 78, pages 1550–1560, 1990.

[31] W. Wilson, L. Rosenberg, and L. Hyatt. Automated analysis of requirement specifications. In *Proceedings of the 19th International Conference on Software Engineering*, pages 161–171, 1997.

[32] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. 2nd Edition, Morgan Kaufmann, San Francisco, 2005.

[33] T. Wolf. Rationale-based unified software engineering model. In *Dissertation, Technische Universität München*, 2007.