A Novel Method for Improving Productivity in Software Administration and Maintenance

Andrei Panu
Faculty of Computer Science, Alexandru Ioan Cuza University of Iasi, 11 Carol I Blvd., Iasi, Romania

Keywords: Information Extraction, Named Entity Recognition, Knowledge Engineering, Machine Learning, Web Mining.

Abstract: After the initial development and deployment, keeping software applications and their execution environments up to date comes with some challenges. System administrators have no insight into the internals of the applications running on their infrastructure, thus if an update is available for the interpreter or for a library packaged separately on which an application depends, they do not know if the new release will bring some changes that will break parts of the application. It is up to the development team to assess the changes and to support the new version. Such tasks take time to accomplish. In this paper we propose an approach consisting of automatic analysis of the applications and automatic verification if the changes in a new version of a software dependency affect them. With this solution, system administrators will have an insight and will not depend on the developers of the applications in such situations, and the latter will be able to find out faster which is the impact of the new release on their applications.

1 INTRODUCTION

The increasing adoption of cloud computing services changed the way how software is developed, how it is deployed and executed, and how we use and interact with it (as developers or as simple users). We now have applications that do not need to be locally installed, are accessible from everywhere using the Internet, and are provided as a service (SaaS – Software as a Service). These applications are usually composed of multiple services that run ”in the cloud”, in various environments. They can also be monolithic applications of different sizes. Even mobile applications that are installed locally use backend services hosted on servers. All these are executed in some pre-configured environments. After the initial development and deployment, some challenges appear regarding the maintenance of the execution environment and of the application. For example, system administrators face a dilemma when an update is available for the interpreter of a certain language (PHP, Python, etc.), especially if it is a major one. Typically, they are not the developers of the hosted applications or services that rely on the interpreter, thus they do not know if the update will bring changes that will break some parts of the software. It is also not in their responsibility to know any details about the internals of the applications. If the developers are faced with the task to support a new version of the interpreter, they must make an assessment of the changes brought by the update and the changes to be made in the application. The same problem appears when updating a library on which the software depends. These tasks require some effort and time.

In this paper we propose a novel approach that gives the administrators an insight for the mentioned problem, and the developers information about the changes to be made, all that in an automatic manner, improving their efficiency for such kind of tasks. Our solution uses machine learning and natural language processing techniques. It is independent on the language in which the software was developed.

The closest approaches in similarity that verify the differences between various versions of code are those that treat the backward compatbility problem. A lot of research is conducted on automating the assessment of backward compatibility of software components, but the solutions do not offer the same functionality as our proposal, they provide a complementary one (Ponomarenko and Rubanov, 2012; Welsch and Poetzsch-Heffter, 2012; Ahmed et al., 2016; Tsantilis, 2009). Their source of information are software repositories, which they monitor and analyze. The approaches are based on different techniques that evaluate the differences between the old source code and the new one (entire code or only the interfaces).

Panu, A.
A Novel Method for Improving Productivity in Software Administration and Maintenance.
DOI: 10.5220/0006407002200229
In *Proceedings of the 12th International Conference on Software Technologies (ICSOFT 2017)*, pages 220-229
ISBN: 978-989-758-262-2

The focus is on assessing if a new version of a software component is compatible with its old version, from a functional point of view. They do not mention anything about the capability to provide information about the change of support status of functionalities in different versions of the libraries/interpreters.

There are some approaches that address this compatibility problem with the same purpose as our proposal, but they are not that general and require a lot of manual work for creating their source of knowledge. For example, for PHP, there is a tool, PHP CodeSniffer (Sherwood, 2017), that tokenizes the source code and detects and fixes violations of code formatting standards. This is used by another tool, PHPCompatibility (Godden, 2017), that contains a set of sniffs for the former and is able to check for PHP version compatibility. Various similar custom tools were developed for different scopes and languages and libraries. The major disadvantages with this kind of tools is that they require a tremendous manual effort to create their knowledge base and that they are developed specifically for a certain language. To the best of our knowledge, a system similar with our approach does not exist.

Our proposal has also a contribution regarding the possibility to identify and extract entities in the programming domain. Known solutions that are capable of identifying various terms, like AlchemyAPI, OpenCalais, TextRazor, MeaningCloud, recognize entities from categories like people, places, dates, not entities in the programming context. To the best of our knowledge, a similar system does not exist.

In the next section we present our approach and give details about how it solves the presented problem. In section 3 we describe a prototype platform that implements our ideas. Section 4 presents some experimental results. Our approach may also be applicable in other fields, use cases which are briefly described in section 5. Finally, section 6 contains future development and research ideas for the evolution of the technology.

2 OUR PROPOSAL

In this vast landscape of software applications of different sizes that run in execution environments configured directly on barebone servers, on virtual machines, or in containers, the management of the applications and the environments after the initial development and deployment is a challenge. Our proposal addresses a specific management problem, regarding keeping the software up to date.

System administrators maintain the infrastructure for running different applications. One of their major tasks is to maintain the systems up to date and this generates some difficulties. When they are faced with the situation of updating the execution environment for the deployed applications, for example updating an interpreter (for PHP, Python, Perl, etc.) to a new version, they have to answer questions like: will the existing applications run on the new version of the interpreter? Are there any parts of the applications that will not run because of the changes?

These are not questions that can be answered easily, because the administrator is usually not the developer of the application(s). The same thing applies in case of libraries which are packaged and installed independently and on which the applications depend. Given the fact that the sysadmin does not have any knowledge about the application (except the necessary versions of the interpreter/libraries when the application was developed and deployed), he/she can only base its decision on assumptions to make the update. One intuitive assumption is that if there is a minor version update for the interpreter or library, everything should be fine, as no major changes in functionality occurred. In the majority of cases, this holds true, but it is still an assumption, not a certainty. The problem arises when there is a major update and the development team does not plan to update the application to support the new release. The problem is very serious in the case of interpreted languages, because errors appear at run-time, only when certain blocks of code get executed, not initially when everything gets compiled, as is the case with compiled languages. Thus, some parts of the application may work, while other parts may not. The system administrator simply will not know if there will be parts of the application that will not execute, he/she is not the developer, he/she is not the tester. This problem scales, because a single administrator can have multiple applications running on his/her infrastructure. Our proposal offers a solution for this situation, by automatically analyzing and providing information about whether the new version will bring some changes that will break the application or not.

Nowadays there is also a shift regarding the management of the execution environment, from the dedicated system administrators to the teams developing the applications, by using containers like *Docker*. This does not solve the problem, only shifts it to the developers, although it does not mean that administrators do not care about the situation of the environments of the containers running on their infrastructure. Thus, when the development team wants to update the interpreter or some libraries used, it faces the same problem described above. Even though the team knows the application, it does not know exactly which blocks of code will execute and which will not. The solution is to analyze the changelog/migration guide and to assess the changes that need to be done. This manual procedure is time consuming. Our solution helps reduce this time.

The approach that we propose towards solving this problem is a technology that is able to automatically scan the code and verify if the used functionalities are still supported in a targeted version of the interpreter/library. This technology is independent of the used programming language. As we have stated earlier, the focus is on interpreted languages. This technology is comprised of three parts:

1. A tool that automatically analyzes the code, extracts the used functionalities, and queries a knowledge base that helps to answer the following questions: is the functionality *X* supported in the new version *N*? If not, what are the changes that were made?
2. A knowledge base (Noy et al., 2001) created automatically that contains information about the functionalities supported in every version of the interpreter/library;
3. A platform that extracts specific entities from online or offline manuals, independently of the programming language, and populates the knowledge base.

The tool is the part that must have access to the code and which uses the (remote) knowledge base to verify if there are functionalities used that are not supported anymore (or are marked to be removed in the future) in the targeted version. It generates a report based on the findings. The key enabler of our technology is the knowledge base. The most important aspect is the contained information and the way it is obtained. For the current version of the platform that populates the knowledge base, which is presented in section 3, the functionalities taken into consideration are the supported functions in all versions of the interpreter/library (the provided APIs). Thus, the analysis tool is a pretty basic component, all that it needs to do is to extract all the functions in the code, eliminate those declared locally, and query the knowledge base to check for support in the targeted version. The only functionality that is more complex is the filtering of the functions provided by a certain used library or by the interpreter.

In this paper, the focus of our work is on the platform that creates the knowledge base. As we have mentioned, the data contained consists of details regarding all the supported functions in the interpreter/library. For each function, we have different attributes, like its signature's components (the function's name, the number of arguments, the types of arguments, the order of the arguments), the return type, its short description, its availability (supported, deprecated, obsolete), and the version number of the interpreter/library in which it is supported. All the information is extracted from online manuals available on the Web or offline ones. The platform described in section 3 is capable to automatically extract the desired data, independently of the content (certain manuals for certain languages/libraries) or the structure of the web pages. The extraction technology does not have implemented any adapters for specific manuals. It supports any manual for any language. The only restriction is regarding the syntax used for writing the functions in the manual. This capability is achieved using machine learning algorithms and different natural language processing (NLP) techniques.

By having such kind of knowledge and being able to automatically analyze the code and obtain a status report regarding the unsupported changes offers some major advantages. The administrators can now have an insight into what will happen with the execution of the application if the update is made. All this can be done without him/her knowing any details about the implementation of the application. For the development team, although it knows the internals of the application, it does not know exactly which blocks of code will execute and which will not, in case of existing changes of some functions. The solution for this is to read the changelog/migration guide, to extract all the functions that were modified and to search for all occurrences of the functions that need to be modified in the application. Another approach is to do a thorough test of the application and check if everything works or not, and if there are problems, the changelog must be consulted. This manual procedure requires a lot of time. By having the tool that is capable of scanning the code, interrogating our knowledge base and reporting what functions are not supported anymore or suffered some changes, pointing the developers exactly to the blocks of code that need to be modified, a lot of time is earned, by eliminating the manual steps described above. The report also provides details about the differences between the old and the new function(s), easing the job of the developers to make the changes, by not requiring them to manually search for that information. All these aspects will improve the delivery time for the updates. Such a report that can be created automatically, containing all the changes that need to be done, is also very useful in case of project estimation. It can be used by all the persons involved in estimating a project (e.g. project managers, business analysts, etc.) to find out from the start what are the implications of making a certain update (in the context described above), without requiring some developers to make an assessment of the changes that need to be done. It eliminates all that manual labor.

By employing our solution which needs access to the source code, the developing teams and the owners of the applications can have some real privacy concerns. Privacy is a very important issue nowadays, and there are many legal and technological aspects regarding this (Pearson, 2009; Alboaie et al., 2015). Our approach and design is in line with principles like Privacy by Design and Privacy by Default (Ferretti, 2015). The functionality of the analysis tool that runs on the users' machines and scans the code is transparent, and is easily verifiable by the user. This tool extracts only the functions used in the code, which represent public information, they are provided by the interpreter/library, ignoring everything else in the source code. Further, only those functions are transmitted into the requests made to the knowledge base. Thus, effectively no proprietary code is extracted and transmitted.

3 SYSTEM ARCHITECTURE AND DESIGN

In this section we present the architecture of a prototype platform that is capable of automatically accessing the manuals of interpreters/libraries available on the Web or offline, identifying and extracting the specific entities, and populating the knowledge base. Its components are designed to be context independent and decoupled. Each component offers its functionality as an independent service. The designed architecture is based on SOA (Service Oriented Architecture) principles. The platform is fully implemented in Python. Figure 1 depicts the system's architecture. The platform has four main components, *CorpusTrain SigDetection*, *CorpusTrain VerDetection*, *WebMiner*, *OntoManager*, and its functionality is split into two phases.

First one, the training phase, uses a machine learning algorithm to generate two models used for the detection of function signatures and version numbers of the interpreter/library in which the function is supported. The training data contains manually annotated information downloaded from PHP and Python online manuals. The components involved in this step are *CorpusDownloader*, *CorpusReader*, *CorpusReader TrainingData*, *CorpusTrain SigDetection* and *CorpusTrain VerDetection*. *Classifier SigDetection* and *Classifier VerDetection* are the classifiers that result from this step.

The second step, the extraction phase, consists of accessing the manuals, analyzing them, and extracting specific knowledge. The main orchestrator of all these operations is *WebMiner* component. The other components used are *CorpusDownloader*, *CorpusReader*, *SigContext*, *NER Version Number*, *NER SigComponents*, *NER SigDescription*. The identified data is then saved in the knowledge base, operation managed by *OntoManger*. Further we present implementation details about each component.

3.1 CorpusDownloader

This module is a specialized Web crawler that accesses online manuals. It accomplishes this through the use of *lxml* library (lxml, 2017). For navigating through the links of the manual we use *XPath* expressions, which are custom built for each site, guiding the crawler to the desired web pages. The content of each page is then downloaded and saved to a local storage using *Lynx* (Dickey, 2017). This is a text web browser, but we use it as a *headless browser*, without requiring any user input, to save the rendered content, exactly like an user sees it. This is an important optimization that improves the extraction process. Libraries like *lxml* or *BeatifulSoup* are able to download web pages, but they download them with the HTML code included, and when we want to clean the tags and extract only the content, there are situations when a sentence (e.g. a function signature) is split into multiple rows, thus is way harder to detect it. This depends on how the developers wrote the HTML code. By using a headless browser, the advantages are that our component is independent of the structure of the page, of its stylization, thus it is not affected from future layout changes. It gives us the content exactly like a human being is seeing it. The only dependency is towards reaching the desired pages.

3.2 CorpusReader

This component loads downloaded content, splits it into a list of sentences, then each sentence is split into a list of words. The order of sentences and words is maintained. The sentence segmentation is done using Punkt sentence segmenter from NLTK platform (NLTK, 2017). It contains an already trained model for English language. For word segmentation, Penn Treebank Tokenizer is used, which uses regular expressions to tokenize text. After that, all punctuation signs are removed, because, in the current version of implementation, they do not bring any value to our extraction process, thus saving some memory.

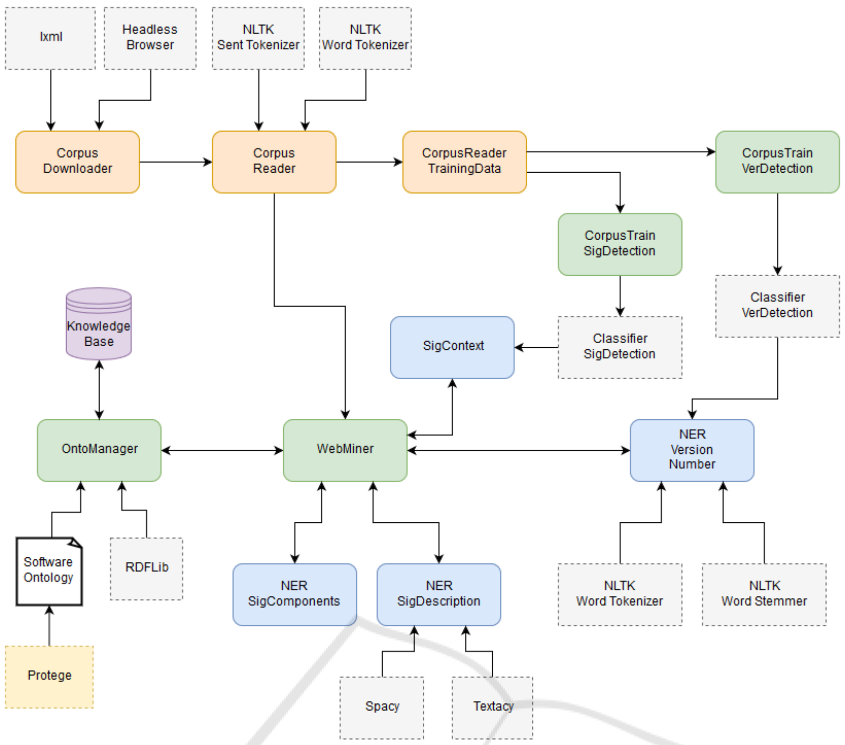

Figure 1: System Architecture.

3.3 CorpusReader TrainingData

This module loads the training data that is used for generating classifiers that can identify signatures and version numbers. It uses all the functionalities provided by *CorpusReader* and, additionally, categorizes each instance based on its label, which can be one of the following two: *pos* or *neg*.

3.4 CorpusTrain SigDetection

The purpose of this component is to train a binary classifier to detect signatures. The assigned labels are *pos* and *neg*. *CorpusReader TrainingData* provides the labeled data, from which the feature sets are obtained. For a signature, we use the following features: if the character before last is parenthesis, if it contains an equal sign before first occurrence of '(' char, if it contains special keywords (like *if*, *for*, *foreach*, *while*, etc.) before first '(' char, the number of colons before first '(' char, if there are more than three words before first '(' char, if there are letters before first '(' char, if it contains unusual words (like ones that do not contain letters or contain only one character) before first '(' char, if the last char is in a predefined list of chars (like !, @, &, *, etc.), if after last ')' char there are more than one characters, if there is a single pair of top level brackets, if there are more than one top level bracket pairs.

The training data contains examples from PHP and Python manuals. The generated feature sets are used to train a Naive Bayes classifier. Although it is one of the simplest models, we have chosen Naive Bayes because it is known to give good results for various use cases and because it does not need a large amount of training instances. The generated classifier is used further by *Classifier SigDetection* component, which offers a single service: it receives a sentence and establishes if it represents a signature or not.

3.5 CorpusTrain VerDetection

This module trains a Naive Bayes binary classifier for detecting version numbers. The assigned labels are also *pos* and *neg*. The feature sets are obtained from the data provided by *CorpusReader TrainingData*. We analyze each word from each sentence, taking also into consideration the word's context. We use for learning the following features: if the first character is 'v', the number of dots, if most frequent chars are digits, the word before, if the word before is in a gazetteer list (which contains names of programming languages).

The training data contains examples from PHP, Python and jQuery manuals. The generated classifier is used by *Classifier VerDetection* component, which offers a single service: it receives a sentence and a word in that sentence, and establishes if the word represents a version number.

3.6 SigContext

This component receives all the sentences of the entire document, identifies the signature(s) using *Classifier SigDetection* and established the descriptive context of each signature. The algorithm for identifying the context is intuitive and models the way humans analyze this: usually all the descriptive details are after the line containing the signature or there are situations when there is a line containing only the function's name (as is the case with PHP manual). This marks the beginning. The context ends when a line containing another signature is identified or the end of document is reached. *SigContext* builds a model of the document containing indexes that represents the positions of the signatures, the start, and the end of their contexts.

3.7 NER Version Number

This component receives the signatures and their descriptive contexts and searches for the version numbers of the interpreter or library in which the function is supported/unsupported. First it checks if there is a version number outside the context, usually meaning that the version applies to all the signatures. After that, it extracts the number(s) detected inside the context (which have higher priority over the one outside the context). It uses Penn Treebank Tokenizer from NLTK for tokenizing each sentence into words, and our trained classifier provided by *Classifier VerDetection* for detecting version numbers. Beside the version, it verifies its context for identifying the function's availability state: supported, deprecated, removed. For that, we have built a dictionary containing stems of different keywords (like changed (in), added (in), removed (from), etc.) that denote the state. Then each word in the context of the version is stemmed and compared with the entries in the dictionary in order to detect the state. Stemming is the technique that removes affixes from a word. We use it as an optimization, allowing us not to store all the forms of words in the dictionary. For this procedure, the Porter stemming algorithm from NLTK is used.

3.8 NER SigComponents

This module receives a sentence representing a signature and identifies its components (e.g. function's name, return type, parameters, etc.).We defined several regular expression rules for this extraction.

3.9 NER SigDescription

This component is capable to identify the short description of a signature, which contains details about its functionality. It searches for it in the signature's descriptive context. We are using a dependency parser provided by *spaCy* library (ExplosionAI, 2017) to analyze the grammatical structure of each sentence. The *textacy* library (DeWilde, 2017) is used for manipulating SVO triples (subject-verb-object) identified by spaCy. The detected lexical items and their grammatical functions represent the source of information for our identification algorithm. This algorithm implements some observed patterns commonly used to express the description and selects the first sentence that meets its rules. This represents a good choice because the description is usually near the signature. We first look at the subject linked to the root verb. If it contains the function's name, then the sentence is very probable to be the description. If the root verb does not have a subject, we look at its tense. If it is labeled as VB (base form) or VBZ (3rd person singular present), then it is very likely to be the description, this being a very common way used to express it. Also large sentences are split and each part analyzed individually, because of the errors of the dependency parser in such situations, in identifying the root verb, its subject, etc.

3.10 WebMiner

This is the orchestrator of the entire information extraction process. Through *CorpusReader* it obtains the content of the page(s) that contain the signatures. Then it provides this list of sentences to *SigContext*, which returns a model containing indexes that represent which sentence is a signature and which one is the start or the end of the sentence's description context. After that, it sends the signature to *NER SigComponents* to obtain its components. *NER SigDescription* receives the context and returns the signature's description. *NER Version Number* receives the signature's context and also the general context, as we named it, and returns the version numbers. All data is put together and is provided to *OntoManager*.

3.11 OntoManager

This component receives the extracted information and generates instances for the knowledge base. The data is expressed using RDF (Resource Description

Framework) model. We use *RDFLib* (RDFLib, 2017) to work with the RDF triples. These triples contain information structured according to concepts illustrated by our software ontology. This ontology was created using Protege. The existing ontologies that are specific to this domain, *SEON Code Ontology* (Würsch et al., 2012), *Core Ontology of Programs and Software (COPS)* (Lando et al., 2007) and *Core Software Ontology (CSO)*, with its extension, *Core Ontology of Software Components (COSC)* (Oberle et al., 2009), do not contain all the conceptual descriptions that are needed in our case. Thus, we have extended them with some new concepts, attributes and relations that model accurately the extracted information.

4 EXPERIMENTAL RESULTS

The performance of the platform for the current targeted use cases is given by the capability of the classifiers used to detect the signatures and the version numbers. For the classifier that detects signatures, the training set contains 280 positive examples and 290 negative ones, and was created from PHP and Python manuals. The first 70% of the labeled instances for each label are used for training, and the rest for testing. The trained model has an accuracy of around 98%. For the classifier that detects version numbers, the training instances are created from examples from PHP, Python, and jQuery manuals. The positive set contains 48 rows with 201 words in total, within which there are 62 words representing a version number. The negative set contains 43 rows with 341 words in total. The feature sets are split, the first 70% of the labeled instances for each label is used for training, and the rest for testing. The model has an accuracy of around 95%. We say that the models have an accuracy value around a certain percent because at train time we randomize the instances, thus the value is slightly different on different executions.

As for validation test, we have pointed the platform to extract data from various pages selected randomly from Node.JS (version 7.7.0) (Node.JS, 2017), Ruby (Ruby, 2017), PHP (PHP, 2017), Python (version 3.6.0) (Python, 2017), and Laravel (Laravel, 2017) online manuals.

Table 1 summarizes the performance of the platform in detecting the signatures for each case. The second column contains the total number of functions that exist in each page, and the last column the percentage of the detected signatures. For Node.JS, PHP, and Python, it extracted all the functions. In case of Laravel, it missed one function who's name contains only one character. In case of Ruby, it detected only 38 functions from a total of 59. This result is not very good because in the page there are many functions which are not written using parenthesis, this being a very important feature when searching for the pattern. In case of PHP, the platform identified 5 more functions (false positives), because there are a lot of comments with code examples in the page, being unsuccessful in filtering all of the functions mentioned there. In the rest of the cases we did not have any false positives or negatives.

Table 1: Signature detection performance.

Man page	No. of functions	Detection rate
Node.JS	26	100%
Ruby	59	64.4%
PHP	1	100%
Python	30	100%
Laravel	80	98.75%

Regarding the identified versions, we obtained the following results:

- for Node.JS: the page contains 26 functions, 5 of them having specified a single version representing when it was added, the rest containing two versions (when it was added and since when it is deprecated). The platform correctly identified all of them, with their status. We do not have any false positives or negatives;
- for Ruby: the page does not contain any version numbers, thus the system correctly did not identify any;
- for PHP: the page contains a single function with 3 version numbers mentioned. The platform successfully identified all of them;
- for Python: the page contains 30 functions, 4 of them having specified two versions, 1of them having specified 3 versions, and the rest only 1 version. The system correctly identified all of them, without any false positives or negatives;
- for Laravel: the page does not mention any details about versions inside the context of the functions, thus the system correctly did not identify any.

Regarding the additional information extracted, for each of the identified functions, the component *NER SigComponents* successfully extracted all of its parameters and its return type. For the detection of the functions' descriptions, the system obtained the results presented in Table 2.

The second column represents the total number of signatures that were detected (each having a single description), and the last one the percentage of the detected descriptions. In case of Node.JS, it failed to

Table 2: Description detection performance.

Man page	No. of det. functions	Detection rate
Node.JS	26	76.9%
Ruby	38	68.4%
PHP	6	100%
Python	30	93.3%
Laravel	79	96.2%

identify 6 descriptions. For Ruby, it missed 12 descriptions. Regarding PHP, it successfully identified the description of the single function. For the other 5 false positives, it did not detect anything because they are examples of code, thus they do not have descriptions. In case of Python, it did not identify the descriptions of 2 functions. Finally, for Laravel, it failed to extract 3 descriptions.

5 ADDITIONAL APPLICABILITY

Our solution directly targets the persons involved in assuring software administration and software maintenance, for the use cases described previously. The features offered by the platform can also be used in the context of the Semantic Web. The purpose of Semantic Web is to make the content available on the Web understandable not only by humans, but also by computers, without using artificial intelligence. This is mainly done by enriching the Web documents with semantic markup in order to add meaning to the content. Thus, a machine will be able to process knowledge about text. There are many semantic annotation formats that can be used in HTML documents, like Microformats, RDFa, and Microdata. In order to facilitate the annotation process, many systems were developed (Reeve and Han, 2005; Uren et al., 2006; Laclavik et al., 2012; Sánchez et al., 2011; Charton et al., 2011). Considering the annotation process, there are tools that allow users to manually create annotations, and tools that create them semi-automatically or automatically. In latter case, considering the methods used, they can be classified in two categories, pattern-based and machine learning-based (using supervised or unsupervised learning techniques). The tools that are based on supervised learning require training in order to identify our specific types of entities. Also the majority of them require the existence of an ontology that defines the semantics of the domain. To the best of our knowledge, an ontology for the kind of information we deal with does not exist. Also, the existing approaches typically cover real world entities. Our platform provides such an ontology and also the information extraction techniques needed to identify and extract the entities that must be annotated (e.g. the function, return type, parameters, parameters type, etc.). Thus, the capabilities of the platform can enable automatic generation of semantic markup for specific Web documents. A special tool must be developed that is able to match each entity extracted by our platform with the information on a page, adding the appropriate annotations.

Another use case is more advanced and represents a vision. It refers to assisting developers in the process of building new applications, especially for the Internet of Things (IoT). In the context of the IoT, there is the need for a single software application to have support for different devices from different manufacturers. This capability is obtained usually through the implementation of adapters, which are dedicated software modules for each device. Each module uses the specific API (Application Programming Interface) of the vendor to interact with its device. In the case of supporting devices from different manufacturers that have the same functionality (e.g. smart plugs, air quality monitors, dimmer switches, thermostats, smoke detectors, etc.), each API is learned and similar adapters are built, because the functions that interact with the devices are mostly the same, like setting a certain value for a threshold or getting a certain value. The differences are in their name and possibly parameters, but their functionality is the same. Our idea implies the development of a single adapter and the use of a special built tool that is able to analyze the functions used in that adapter and propose the equivalent functions in the APIs of other similar devices for which adapters must be developed. Our knowledge base, that can contain all the functions in each API, is the source of information. Based on that information, we envision the development of a new technology that is able to suggest similar functions in the targeted APIs, by analyzing a function's signature, and most importantly, the meaning of its description and, eventually, each parameter's description. This capability can be achieved with our platform by using NLP resources, like WordNet, a semantically oriented dictionary of English, and techniques for analyzing the structure of a sentence (by using context free grammars, dependency grammars, feature based grammars, etc.) and its meaning (by using first order logic, λ calculus, etc.) (Bird et al., 2017). To accomplish semantic analysis is very hard. It represents the most complex phase of natural language processing. There is a lot of ongoing research towards this goal and on the task of computing sentence similarity (Atoum et al., 2016; Dao and Simpson, 2005; Crockett et al., 2006; Miura and Takagi, 2015; He et al., 2015; Liu and Wang, 2013; Sanborn and Skryzalin, 2015; Erk, 2012). There are also many tools available capable of comparing the meaning of two sentences, with different success rates (SEMILAR, 2017; DKPro, 2017; RxNLP, 2017; Pilehvar et al., 2013; Linguatools, 2017; Cortical.io, 2017). At the moment, we think that existing solutions can provide acceptable results, but still much further research must be done to accomplish our vision. To the best of our knowledge, there is no available system that is using this kind of techniques and is capable of generating code in this context.

6 CONCLUSIONS AND FURTHER WORK

The proposed solution is based on many components, each with varying degrees of complexity. Refinements can be made to any of them for further improving the platform (e.g., using an improved headless browser that is aligned with the latest Web standards, improving the classifiers, implementation of new patters for the detection of signatures' descriptions, etc.). Regarding the system's architecture, we plan on adding a REST API to each of the components, thus making the platform easily deployable and accessible in cloud environments and highly scalable. It could also be easily integrated in environments that make use of Enterprise Service Buses (Chappell, 2004), or newer approaches like Swarm Communication (Alboaie et al., 2014; Alboaie et al., 2013). Besides the improvements that will be made to the platform, much work will also be directed towards the development of the tools that enable the use cases. First, there is the tool that is able to scan code written in different languages, extract all the used functions, eliminate those that were locally defined, interrogate our knowledge base, and build the report regarding support status for the targeted version of the interpreter/library. Then, we have the semantic annotator tool, that is able to mark up HTML code referencing the specific entities.

In this paper we have addressed the problem of keeping software applications and their execution environments up to date. As we have seen, this task comes with some difficulties. System administrators have no insight about the internals of the applications that run on their infrastructure, so if they are faced with updating the execution environment, they do not know if the applications will be fully functional after the update. It is up to the development team to do the necessary verifications and eventual changes. The same applies in case of software libraries on which the applications depend. The tests require a lot of manual work and take time to accomplish. We have proposed a novel approach that improves productivity in accomplishing such tasks, by automating the assessment of the changes that were made in a new version and their impact on the functionality of the application. Our solution is based on one major component, the platform that is able to automatically create a knowledge base containing details about supported functionalities in each version of the targeted interpreter/library, independent of the programming language used. It has applicability in many fields, like software administration, software maintenance, and even project management. In this paper we presented the prototype of the platform. To the best of our knowledge, similar solutions do not exist. The existing tools that are used require manual specification of the changes that were made and are also language dependent.

REFERENCES

Ahmed, H., Elgamal, A., Elshishiny, H., and Ibrahim, M. (2016). Verification of backward compatibility of software components. US Patent 9,424,025.

Alboaie, L., Alboaie, S., and Barbu, T. (2014). Extending Swarm Communication to Unify Choreography and Long-lived Processes. In *Proceedings of the 23rd International Conference on Information Systems Development (ISD 2014)*.

Alboaie, L., Alboaie, S., and Panu, A. (2013). Swarm Communication – A Messaging Pattern Proposal for Dynamic Scalability in Cloud. In *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013*, pages 1930–1937.

Alboaie, S., Alboaie, L., and Panu, A. (2015). Levels of Privacy for eHealth Systems in the Cloud Era. In *Proceedings of the 24th International Conference on Information Systems Development*, pages 243–252.

Atoum, I., Otoom, A., and Kulathuramaiyer, N. (2016). A Comprehensive Comparative Study of Word and Sentence Similarity Measures. *International Journal of Computer Applications*, 135(1):10–17.

Bird, S., Klein, E., and Loper, E. (2017). Natural Language Processing with Python – Analyzing Text with the Natural Language Toolkit. http://www.nltk.org/book/.

Chappell, D. (2004). *Enterprise Service Bus*. O'Reilly Media, Inc.

Charton, E., Gagnon, M., and Ozell, B. (2011). *Automatic Semantic Web Annotation of Named Entities*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Cortical.io (2017). Similarity Explorer. http://www.cortical.io/similarity-explorer.html.

Crockett, K., McLean, D., O'Shea, J. D., Bandar, Z. A., and Li, Y. (2006). Sentence Similarity Based on Semantic Nets and Corpus Statistics. *IEEE Transactions on Knowledge and Data Engineering*, 18(undefined):1138–1150.

Dao, T. N. and Simpson, T. (2005). Measuring similarity between sentences. *WordNet.Net, Tech. Rep.*

DeWilde, B. (2017). textacy NLP library. http://textacy.readthedocs.io/en/latest/.

Dickey, T. E. (2017). Lynx Text Web Browser. http://lynx.browser.org/.

DKPro (2017). DKPro Similarity Framework. https://dkpro.github.io/dkpro-similarity/.

Erk, K. (2012). Vector space models of word meaning and phrase meaning: A survey. *Language and Linguistics Compass*, 6(10):635–653.

ExplosionAI (2017). spacy NLP library. https://spacy.io/.

Ferretti, E. (2015). Privacy by design and privacy by default. http://europrivacy.info/2015/06/09/privacy-design-privacy-default/.

Godden, W. (2017). PHPCompatibility. https://github.com/wimg/PHPCompatibility.

He, H., Gimpel, K., and Lin, J. (2015). Multi-perspective sentence similarity modeling with convolutional neural networks. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1576–1586.

Laclavik, M., Šeleng, M., Ciglan, M., and Hluchý, L. (2012). Ontea: Platform for pattern based automated semantic annotation. *Computing and Informatics*, 28(4):555–579.

Lando, P., Lapujade, A., Kassel, G., and Fürst, F. (2007). Towards a general ontology of computer programs. In *Proceedings of the International Conference on Software and Data Technologies*, pages 163–170.

Laravel (2017). Manual page. https://laravel.com/docs/5.4/helpers.

Linguatools (2017). DISCO. http://www.linguatools.de/disco/disco_en.html.

Liu, H. and Wang, P. (2013). Assessing Sentence Similarity Using WordNet based Word Similarity. *JSW*, 8(6):1451–1458.

lxml (2017). lxml XML toolkit. http://lxml.de/.

Miura, N. and Takagi, T. (2015). WSL: Sentence Similarity Using Semantic Distance Between Words. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 128–131, Denver, Colorado. Association for Computational Linguistics.

NLTK (2017). Natural Language Toolkit. http://www.nltk.org/.

Node.JS (2017). Manual page. https://nodejs.org/api/util.html.

Noy, N. F., McGuinness, D. L., et al. (2001). Ontology development 101: A guide to creating your first ontology.

Oberle, D., Grimm, S., and Staab, S. (2009). An ontology for software. In *Handbook on ontologies*, pages 383–402. Springer.

Pearson, S. (2009). Taking account of privacy when designing cloud computing services. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 44–52. IEEE Computer Society.

PHP (2017). Manual page. http://php.net/manual/en/ function.chmod.php.

Pilehvar, M. T., Jurgens, D., and Navigli, R. (2013). Align, Disambiguate and Walk: A Unified Approach for Measuring Semantic Similarity. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 1341–1351. ACL.

Ponomarenko, A. and Rubanov, V. (2012). Backward compatibility of software interfaces: Steps towards automatic verification. *Programming and Computer Software*, 38(5):257–267.

Python (2017). Manual page. https://docs.python.org/ 3/library/os.path.html.

RDFLib (2017). RDFLib RDF library. https://rdflib.readthedocs.io/en/stable/.

Reeve, L. and Han, H. (2005). Survey of Semantic Annotation Platforms. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 1634–1638, New York, NY, USA. ACM.

Ruby (2017). Manual page. https://ruby-doc.org/docs/ruby-doc-bundle/Manual/man-1.4/function.html.

RxNLP (2017). Text Similarity API. http://www.rxnlp.com/api-reference/text-similarity-api-reference/.

Sanborn, A. and Skryzalin, J. (2015). Deep learning for semantic similarity.

Sánchez, D., Isern, D., and Millan, M. (2011). Content annotation for the semantic web: an automatic web-based approach. *Knowledge and Information Systems*, 27(3):393–418.

SEMILAR (2017). SEMILAR: A Semantic Similarity Toolkit. http://deeptutor2.memphis.edu/Semilar-Web/public/contact.html.

Sherwood, G. (2017). PHP CodeSniffer. http://pear.php.net/package/PHP_CodeSniffer/.

Tsantilis, E. (2009). Method and system to monitor software interface updates and assess backward compatibility. US Patent 7,600,219.

Uren, V., Cimiano, P., Iria, J., Handschuh, S., Vargas-Vera, M., Motta, E., and Ciravegna, F. (2006). Semantic Annotation for Knowledge Management: Requirements and a Survey of the State of the Art. *Web Semant.*, 4(1):14–28.

Welsch, Y. and Poetzsch-Heffter, A. (2012). Verifying Backwards Compatibility of Object-oriented Libraries Using Boogie. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 35–41. ACM.

Würsch, M., Ghezzi, G., Hert, M., Reif, G., and Gall, H. C. (2012). SEON: a pyramid of ontologies for software evolution and its applications. *Computing*, 94(11):857–885.