Ingeniería y
Competitividad
Vol 25 (3) 2023; doi: 10.25100/iyc.v25i3.13171

Vol 25 (3)

ISSN
0123-3033
e- 2027-8284

Artículo de investigación

Architectural technical debt: an identification strategy

Deuda técnica en arquitectura: una estrategia de identi icación

Boris R. Pérez[1]

[1]Departamento de Sistemas e Informática, Universidad Francisco de Paula Santander, Cúcuta, Colombia

Abstract

Architectural Technical Debt is a metaphor for actions made by architects to achieve short-term goals while potentially harming the system's long-term health. Architectural Technical Debt is difficult to detect since it is associated with a system's long-term maintenance and evolution. In this research, we describe an architectural evolution-based method for debt identification that is backed by a supervised machine learning model and is based on information obtained from artifacts produced during architecture design. We discovered that even with a small amount of data, the machine learning model produces good results in terms of Recall and even Accuracy. The trial provides insights that allow us to conclude that this idea works well and might be utilized as a starting point to assist architects in identifying and managing Architectural Technical Debt.

Resumen

La deuda técnica en arquitectura es una metáfora de las decisiones tomadas por los arquitectos para alcanzar objetivos a corto plazo, pero que pueden dañar la salud del sistema a largo plazo. Esta deuda técnica es difícil de detectar, ya que está asociada a la mantenibilidad y la evolución de un sistema. En esta investigación, describimos un método basado en la evolución de la arquitectura para la identificación de la deuda que está respaldado por un modelo de aprendizaje automático supervisado y se basa en la información obtenida de los artefactos producidos durante el diseño de la arquitectura. Descubrimos que, incluso con una pequeña cantidad de datos, el modelo de aprendizaje automático produce buenos resultados en términos de Recall e incluso de Accuracy. Las experimentaciones realizadas proporcionaron información que nos permite concluir que esta idea funciona bien y podría utilizarse como punto de partida para ayudar a los arquitectos a identificar y gestionar la deuda técnica en arquitectura.

Keywords:
Architectural technical debt, Identification strategy, Machine learning, Software architecture

Palabras clave:
Deuda técnica en arquitectura, Estrategia de identificación, Aprendizaje de máquina, Arquitectura de software

Cómo citar:

Pérez, B. Architectural technical-debt: an identification strategy. *Ingeniería y Compe i ividad*,2023, 25(3); e-21413071. doi: https://doi.org/10.25100/iyc.v25i3.13071

Recibido: 07-17-23
Aceptado 09-20-23

Correspondencia:
borisperezg@ufps.edu.co

Conflicto de intereses:
Ninguno declarado

¿Por qué se realizó?

Todas las etapas del desarrollo de software son importantes, siendo la codificación la etapa más material, y la arquitectura de software la etapa más creativa. Es por esto que gran parte del esfuerzo en identificar, pagar y prevenir la deuda técnica se ha enfocado en el código fuente. A final de cuentas, el código siempre está actualizado y se puede evaluar cuantitativamente. Por otro lado, la etapa más importante en el desarrollo de software es el diseño de la arquitectura, y las decisiones que se toman en esta etapa son las que traen las consecuencias más críticas para el futuro del software. A pesar de esto, pocos trabajos se han enfocado en la deuda técnica a nivel de arquitectura, y tiene sentido, considerando lo poco material que es esta etapa. Arquitecturas diseñadas en cualquier herramienta, documentos no estructurados describiendo las decisiones, información no escrita y que se evapora fácilmente, diseños realizados sin la formalidad requerida. Algunas propuestas se han enfocado en realizar entrevistas a los arquitectos, o intervenir en las reuniones de arquitectura, sin embargo, eso reduce la aplicabilidad de la propuesta por cuanto genera una distracción al proceso. A raíz de esto, se decidió proponer una estrategia apoyada en Machine Learning que utilizara los artefactos más comunes usados por los arquitectos, pero enmarcando la solución a determinadas her mientas y a su correcto diligenciamiento.

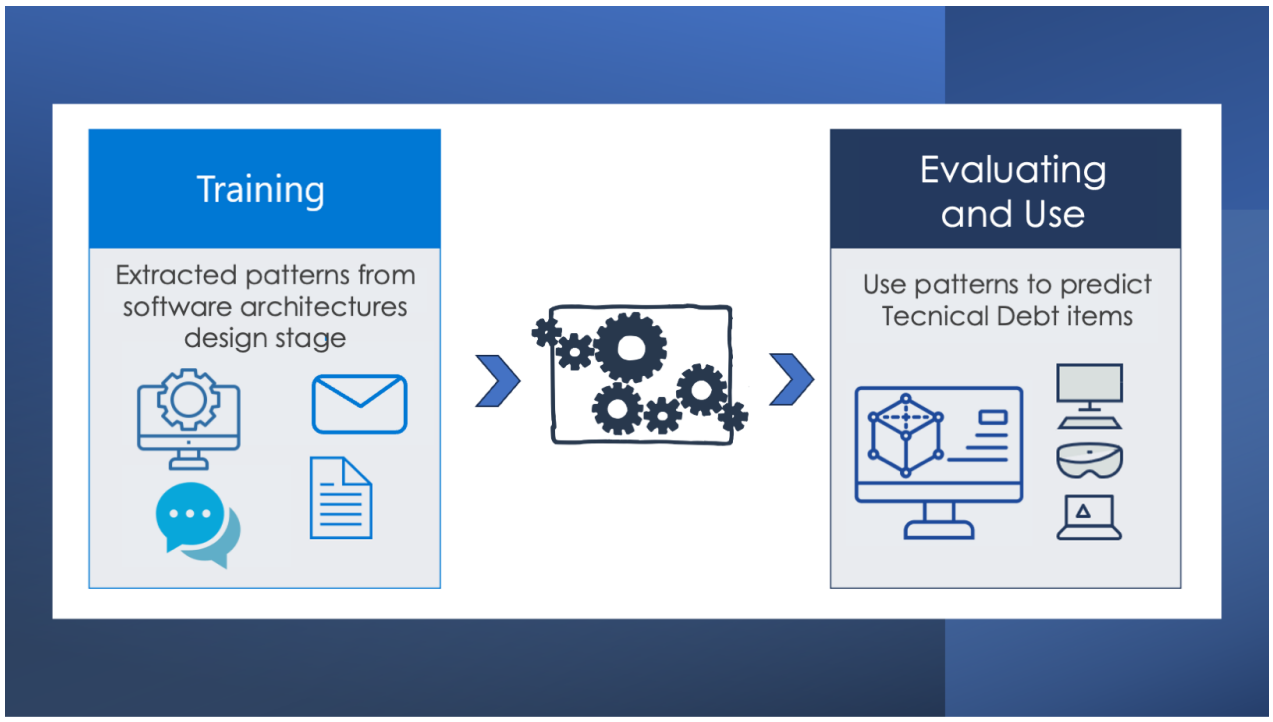
¿Cuáles fueron los resultados más relevantes?

Lo primero fue descubrir que es posible realizar la identificación de deuda técnica a nivel de arquitectura utilizando artefactos propios de la etapa de diseño de la arquitectura, tales como los múltiples diseños de la arquitectura, las decisiones tomadas, los registros de correo y de chat, así como el historial de commit. Sin embargo, también es importante reconocer que los artefactos deben registrar de manera específica. Lo segundo resultado correspondió a la comprensión del uso de algoritmos supervisados para realizar la identificación. La definición de las variables que permitieron el entrenamiento del modelo para lograrlo. Y en este, la participación del arquitecto es clave al realizar una primera identificación base para que el modelo pueda aprender desde ahí. El tercer resultado es la posibilidad de seguir entrenando el modelo en la medida en que nuevos proyectos se sumen a esta actividad. El modelo base puede entrenarse con un solo proyecto, luego este modelo puede reentrenarse con los resultados de nuevas identificaciones y así sucesivamente. Esto podría hacerse de manera privada al interior de una empresa, o hacer el modelo público para que otros proyectos puedan beneficiarse.

¿Qué aportan estos resultados?

Los resultados de este proyecto evidencian que es posible automatizar la identificación de la deuda técnica en arquitectura sin interferir en las reuniones ni restándole tiempo a los arquitectos con entrevistas, utilizando los artefactos propios de la etapa de diseño de la arquitectura. Al mismo tiempo, permiten marcar un camino de investigación que permita la inclusión de nuevos artefactos, o incluso de permitir ambiguedad en ciertos artefactos, como por ejemplo, permitir que los diagramas se realicen en lenguajes diferentes a ArchiMate. También permiten, al ser una herramienta completa, que pueda ser usada en la industria, desde la cual, la retroalimentación permitirá que se pueda refinar y de esta manera, ampliar su aplicabilidad.

Graphical Abstract



doi: 10.25100/iyc.v25i3.13071

Introduction

Software companies are under increasing pressure to provide a high-quality solution that may be utilized continuously while consuming less time or resources (1). As a result, software teams make decisions to prioritize some quality attributes to achieve short-term goals while potentially jeopardizing the system's maintainability. This type of design choice is referred to as Technical Debt - TD (1).

Architectural decisions, according to Ernst et al. (2), are the most significant source of TD. Architectural technical debt (ATD) is TD at the architecture level mostly caused by architectural decisions that jeopardize system-wide quality criteria, notably maintainability and evolvability. Non-uniformity of patterns and policies (3), architecture smells and anti-patterns (3,4), contradictory quality attribute synergy (3), breaches of best architectural practices (4,5), and complicated architectural behavioral relationships (3,4,5) are examples of typical ATD.

Despite the importance of architectural decisions on the system, identifying ATD without evaluating source code remains challenging. Some of the reasons for this include a lack of time, of initiative to handle it, of appropriate instruments to assist this activity, of understanding of how to execute this task, or a lack of methods to determine what type of information must be gathered (6). It must be determined what, where, and when ATD items were injected (7).

Some methods to ATD management focus on particular tasks within an overall ATD management process (1,8) or include all ATD management activities (9). However, these methodologies are centered on the source code, which may need extensive modification to reimburse the ATD (4), or they rely significantly on interviews with the architectural team (9).

In this paper, we describe a method for assisting software architects in recognizing ATD that has been injected into their systems. To achieve this purpose, this technique relies on the growth of the software architecture via architectural models, a collection of artifacts created during the solution architecture design stage, and a supervised machine learning model. This data is utilized to determine the type of ATD, its placement within the architecture, and the time when it was injected. ATD is regarded a significant sort of risk for a software project in this study effort, and it is thus vital to make it apparent so that software firms understand its negative influence on software lifecycle projects and products.

The approach makes three contributions: i) it defines a set of meta-models to abstractly represent the architectural models and information extracted from artifacts used in the architecture design stage, ii) it defines strategies for identifying specific types of candidate ATD based on the evolution of the architecture; and iii) it provides a first attempt of a supervised ML model to support ATD identification. Thus, these contributions improve the capabilities of our approach at architecture level without considering source code.

This approach is evaluated through an analysis of ATD identification cases from a real software project in a software company in Cúcuta (Colombia). The results so far show that the approach can identify certain types of ATD, also that this approach works well as a starting point for ATD identification and can be used within other software projects.

Methodology

This section presents Rebel, an architecture evolution-based approach to assist software architects in identifying ATD injected into their software architectures. Rebel focuses on ATD at the architecture level only without considering source code.

This approach relies on service-oriented architectures and its relationship with the business strategies. It is based on the idea that over time, architects will produce architecture models and exchange knowledge through multiple artifacts, such as architecture models, architectural decisions, chats, emails and commit logs. Therefore, the evolution of the model is an important factor for this approach.

The strategy followed by Rebel consists in using information extracted from structured (architectural models modeled on ArchiMate language) and unstructured (emails, chat logs, commit logs and architectural decisions records - ADR) artifacts to identify ATD in the architecture. This information is represented following a model-driven strategy, where each artifact is represented in a model which is conform to a meta-model. From this point, Rebel can review the changes over the architecture and enrich them with the information extracted and modeled from the rest of the artifacts. Changes are actions over an architectural element such as a change in the value of a property or a new relationship; and are identified among versions of architectural models.

The methodology of Rebel consists of three (3) stages, and six (6) steps, as presented in Figure 1. These stages are iterative and could be performed multiple times during the life cycle of the project. An iteration will depend on the architects' team.

Stage 1 (Extraction) is responsible to extract and store the features of the artifacts and it is presented in Section 2.1. Stage 2 (Synthesis) is responsible to identify a set of changes linked to some specific architectural elements, and to enrich these changes with the information extracted from the artifacts. This stage is also responsible to prepare the setting for model training in the next stage and it is presented in Section 2.2. Stage 3 (Learning) is responsible to prepare the data that will be used to train the ML model. This ML model is used to support the identification of candidate ATD items, and it is presented in Section 2.3.

Stage 1: Extraction

This is the top section of the figure 1 and is focused on having the information in an abstract way, independent of the file type, and through a common language. To achieve this, first the artifacts need to be uploaded. Artifacts can be an ArchiMate model, an architectural decision (ADR), an email, a chat log, or a commit log. Each artifact is processed by reading all the content and extracting the relevant data (Step 1). Then, the corresponding meta-model is instantiated, and the data extracted is modelled (Step 2).

According to Figure 1, there are two groups of models stored after Step 2: GM1 and GM2. They are kept separate because the information modeled is used in different moments.

doi: 10.25100/iyc.v25i3.13071

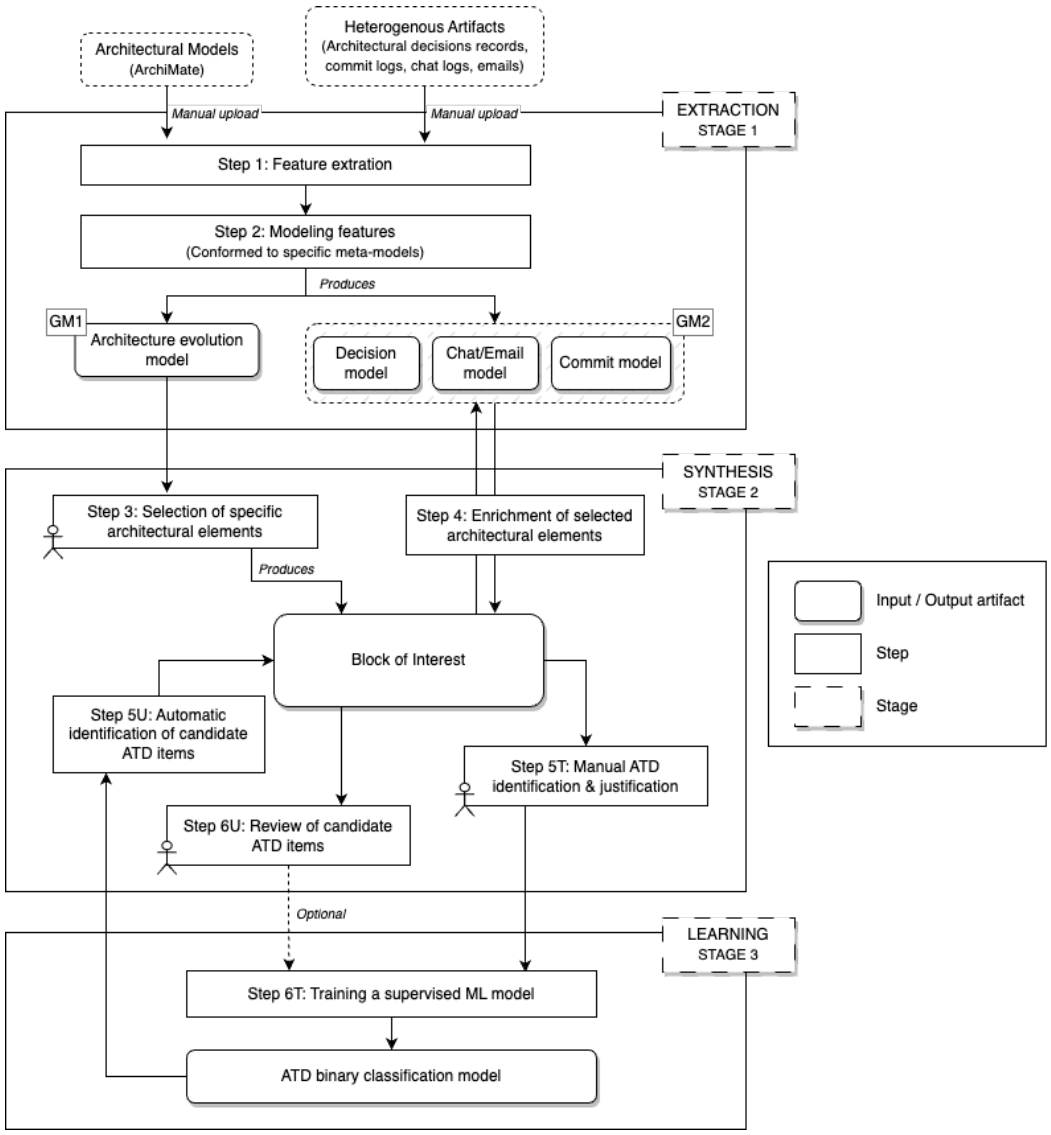


Figure 1. Working flow of the proposed approach for ATD identification.

Stage 2: Synthesis

This is the middle section of Figure 1 and its goal is to prepare the settings to build and use the supervised machine learning model to support the identification of candidate ATD items.

Step 3: Selection of specific architectural elements

Identification of candidate ATD items requires the selection of the most relevant architectural elements. Step 3 requires the architect to select the architectural elements that he/she wants to keep an eye on. This selection could be related to some specific interest of the architect or to some business interest.

Each of these architectural elements may change one or more times during the evolution of the architecture. This step creates a Block of Interest (BoI), which is composed of Facts, and each Fact represents a change in an architectural element (i.e., component), along with its relationships and properties. In other words, a Fact could be a change in the properties of a component, or a change in the properties of a link between two

components, or the creation of a link between two components. These changes are identified by comparing the differences between the ArchiMate architectural models uploaded to Rebel. figure 2 presents the web app screen for the BoI creation.

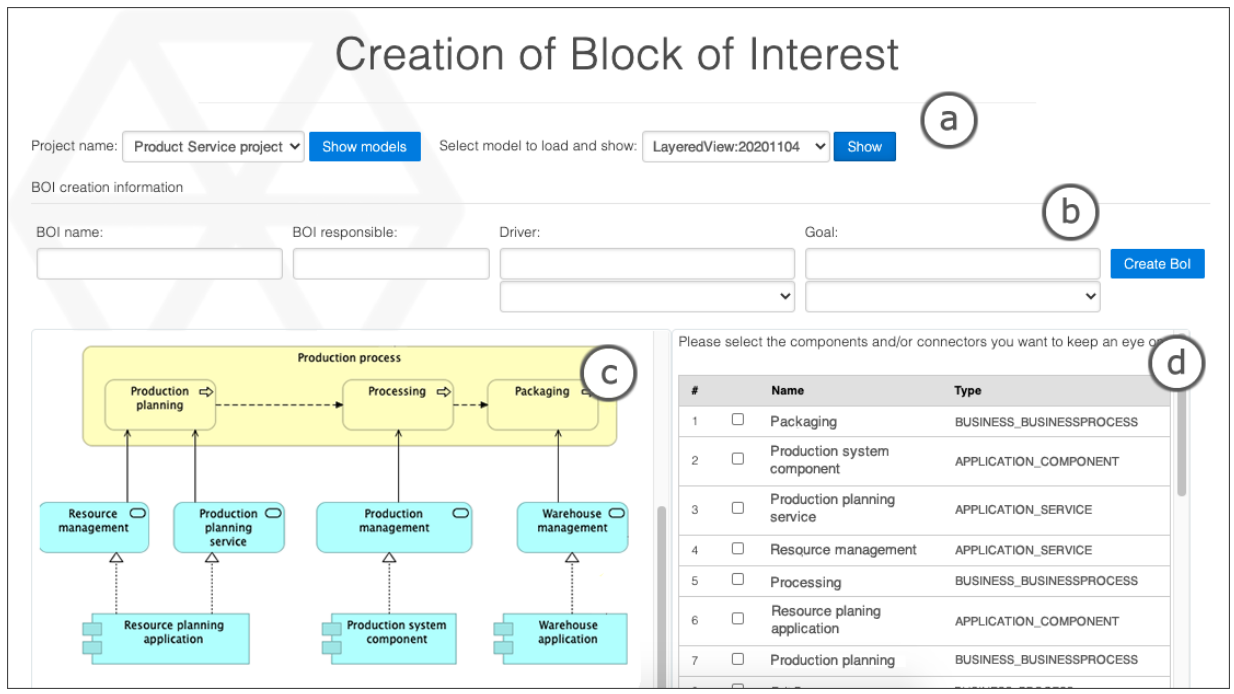


Figure 2. BoI creation screen provided by Rebel

There are four parts in figure 2: Part (a) is used to select the project that will own the BoI, and then, to select the base architectural model used to visualize and select the elements of interest. Part (b) is used to enter all the information related to the BoI, such as name, responsible, driver, and goal. These last two fields can also be selected from a drop-down if the ArchiMate model already includes them. Part (c) is used to visualize the base model selected in Part (a). From this view, it is possible to identify elements from the Strategy, Business, and Application layers of the architectural model. This visualization is relevant to support the selection of architectural elements in Part (d). This last part is used to select all architectural elements the architect wants to keep an eye on.

BoI creation begins by retrieving all ArchiMate models related to the project. Then, only the models where there is at least one of the selected BoI elements are filtered out. It is important to remark that a Fact is a change of an element (or its relationships) between two models, therefore, changes of an element of the BoI are obtained through comparison in models M1 and M2. Likewise, the properties of that element are obtained in both models. Here is where Fact creation occurs. If the element exists in both models, then, each of the properties is compared. If any change is found, then a Fact is created. If the element does not exist in model M1 but it does exist in model M2 then a Fact of creation of the element is created and a Fact for each of its properties is created. If an element exists in model M1 but does not exist in model M2, then a Fact of deletion is created. This algorithm includes changes in elements, their properties, and the links with other elements.

doi: 10.25100/iyc.v25i3.13071

Step 4: Enrichment of selected architectural elements

After BoI is created, Step 4 is executed automatically. This Step focuses on provide more information about the architectural elements included in the BoI. This is accomplishing through two operations: (i) to find semantic similarities of the name of an element among the content of the artifacts, and (2) to associate a Fact with an ADR. Looking for information is used to enrich the BoI and therefore, to strengthen the ATD identification process.

Semantic similarity analysis is implemented in Python and is done for commit logs, chat logs, and email messages. These artifacts could have a lot of information and it could be difficult for the architect to identify where it is saying something about an architectural element.

Related to the second operation (to associate a Fact with an ADR), every change in the architecture should be supported by an architectural decision. Giving the influence of architectural decisions in the architecture, it was established that architects link Facts (changes) to their corresponding architectural decision (ADR). This action is relevant because it allows that some types of ATD can be identified, such as *Violations of best architectural practices* and *Conflicting quality attribute synergies*.

Step 5T: Manual ATD identification & justification

The goal of this step is to mark all Facts that architects identified as ATD items. Each Fact needs to be reviewed. Each Fact includes the date, the action performed (i.e., Create element type business_process), the name of the element (i.e., Next Control Estimation), and the assigned architectural decision. A Fact is related to two concepts: Element and Relation. The former when the change is related to an element, and the latter when the change is about a relationship between two elements.

Additionally, the architect needs to select the compromised quality attribute and the type of the ATD. The rationale of the ATD and the expected benefits of the ATD are described by the architect. The artifacts with matches for the Fact element are also included in the description, giving the architect a comprehensive understanding of the architectural element linked to an ATD item.

This step is marked with a T letter because it is part of the workflow of the training of the machine learning model. Steps 5U and 6U will be explained later in the following section and they are related to the workflow of the use of the machine learning model.

Stage 3: Learning

This is the bottom section in Figure 1 and its main idea behind is to transform the BoI into a valid data format to be used as input to train a supervised ML model. The output of the Step 6T is a multiclass classification model, where the target variables are the type of ATD and the affected quality attribute.

Multiclass classification is a classification task in which each sample is assigned to one and only one label, but with one or more labels to assign. For example, an image of fruit could be labeled as orange, apple, or banana, but fruit can be either an apple or a banana but not at the same time.

After the ML model is created, it can be used to identify candidate ATD items. Software architects will be responsible to accept or reject the candidates. Facts, with all the information related, are used as predictors for this model. The more facts used as predictors, the better the accuracy of the model.

Training the supervised ML model triggers the gathering of all information about the BoI, its Facts, and its relationships with ATD items. This information is used to build the required dataset. Additionally, some pre-processing operations are required before performing the training of the supervised model.

The training dataset has 25 properties that describe relationships, behavior, interaction with other layers, properties and textual information of the heterogeneous artifacts. This dataset ends with 2 properties which are the target variables or class variables. These variables can take one label among several ones. This allows the model to provide more information in classification that only predicts if the sample is an ATD candidate or not. These two target variables are: atdcause (used to represent the type of ATD), and affectedqa (used to represent quality attribute affected by the ATD).

Before a dataset could be used to train an ML model, it is required to perform some transformations. Figure 3 presents the pre-processing done to textual data from the dataset. First, text data is cleaned by applying tokenization, then removing stop words and applying lemmatization. After this, the most relevant pair of words (bigrams) are extracted to identify topics in these text data.

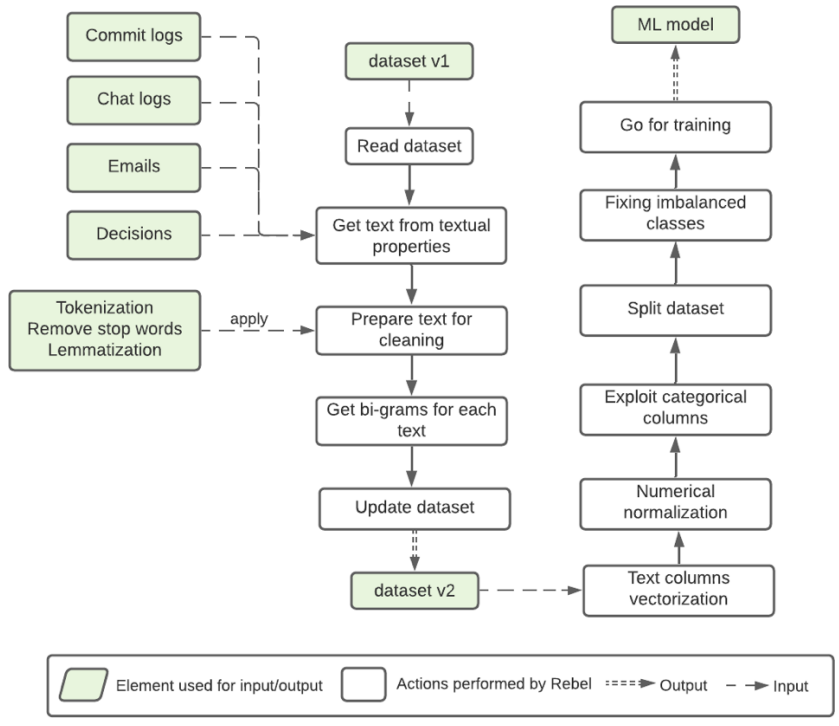


Figure 3. Data processing tasks

This pre-processing produces a second dataset (dataset v2) including the bigrams for commit messages, chat messages, emails, and decisions. Then, based on this second dataset, several operations are done over the dataset: (i) Textual data is processed by applying text features vectorization and vectorize it using TFIDF. (ii) Numerical data is processed by applying normalization to scale the numerical data. And (iii) Categorical data is processed by applying One Hot Encoding (OHE) to vectorize it.

To deal with imbalanced classes two strategies were used: Oversampling and generating synthetic samples. Oversampling is used to increase the number of samples of the minority classes, and then SMOTE is used to balance the number of samples of the

doi: 10.25100/iyc.v25i3.13071

dominant class. After all this processing, the dataset is ready to train the model. Dataset is split into train dataset and test dataset. The test dataset is 20% of the original dataset and the training dataset is 80% of the original dataset. StratifiedKFold was used to split the data and test the model. At the end of Step 6T, the model is built, stored, and ready to use.

Using the Supervised ML Model

The following step (Step 5U) is to use the supervised ML model created in Step 6T to identify ATD candidates in a new BoI from the same or from a different project. In order to achieve this, it is required to create a new BoI, and this BoI could require uploading a new set of artifacts. When the BoI is created and the analysis of similarity is done, the architect will execute the procedure to identify ATD candidate items. After this, each Fact will be reviewed by the ML model, and it will predict the type of ATD and the affected quality attribute. This information is then integrated into the BoI.

Step 6U is the last step in this proposal and it corresponds to the revision of the ATD candidate items predicted by the supervised ML model. In this step the architect can accept or reject the candidates and can mark new Facts as ATD items. The architect could also use this BoI to train and improve the ML model with all this new information. This way, more and more data will help to improve the model. ATD candidate items accepted by the architect should be updated by adding the rationale of the ATD and some possible benefits. Also, he/she can change the type of ATD and the affected quality attribute.

Results and discussion

This Section presents an industrial case study to demonstrate how Rebel can support architects to identify candidate ATD items in their software architectures.

GN (anonymized for privacy reasons) is a company supporting Higher Education institutions in generating their digital transformation. This company focused on digital transformation that advises the public and private sectors in Colombia in the effective introduction of Information and Communication Technologies. This company is in Cúcuta, Colombia.

GN has almost 5,000,000 lines of code, therefore, changing any technology or even updating the version of any technology represents certain difficulties. The modules with high demand are Messaging, Teachers, and Student, which require high availability.

The architecture related to the business process *Grades Grouper* was used to review the approach in a real industrial context. This project uses the supervised ML model trained and improved from two previous projects. The driver of this project is to increase market share by improving the user experience. One of the goals pursued by this driver is to improve the generation of the report of grades at the end of each period. The BoI generated 59 Facts including operations of creation, update and delete. The Facts were created according to the date or version of the architectural model. From the model LayeredView_20210315, 13 Facts were created. From the model LayeredView_20210322, 31 Facts were created, and from the model LayeredView_20210325, 15 Facts were created.

The supervised ML model identified 14 candidate ATD items related to Facts, but not all these candidates were correctly identified. Table 1 presents the confusion matrix to describe the performance of a classification model on a set of test data.

True positive (TP) and true negatives (TN) are the observations that are correctly predicted. The goal of any model is to minimize false positives (FP) and false negatives (FN). False positives and false negatives occur when the actual class contradicts the predicted class. These four parameters (TP, NF, FP, and TN) are used to evaluate the performance of Rebel's supervised ML model through four evaluation metrics: Accuracy, Precision, Recall, and F1 Score.

Table 1. Confusion matrix of the supervised ML model

		Predicted class	
		Positive	Negative
Actual class	Positive	TP = 8	FN = 1
	Negative	FP = 6	TN = 44

Accuracy is the ratio of correctly predicted observations over the total number of observations. This is a simple and effective measurement if the number of observations keeps the same. *Precision* is the ratio of correctly predicted positive observations over the amount of correct and incorrect predictions. High precision relates to the low false positive rate. Precision is a good measure to use when the cost of False Positives is high. *Recall* is the ratio of correctly predicted positive observations over the number of positive observations in actual class. The recall is a good measure to use when the cost of False Negatives is high. Finally, *F1 Score* is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. F1 Score is useful when there is an uneven class distribution. This set of metrics were calculated for Rebel's supervised ML model and there are presented in table 2.

Table 2. Evaluation metrics for Rebel's supervised ML model

Metric	Measure
Accuracy	0.881
Precision	0.571
Recall	0.889
F1 score	0.696

As can be seen in table 2, all four metrics were calculated, however, as presented below, not all of them are required for Rebel's supervised ML model evaluation. First, it is important to mention that, for this approach, False Negatives are the most important parameter. Therefore, Recall could be more useful than Precision. A false negative is when a Fact is injecting debt, but the supervised ML model does not recognize or identify it as debt. In this case, the cost associated with False Negative will be high because it will be related to future maintainability problems. Therefore, the most relevant metric to review is the Recall. For our model, the measure for this metric is 0.889 which is good for this model as it's above 0.5.

Second, the class distribution of the dataset used for training is balanced. Therefore, Accuracy could be more useful than F1 Score. However, F1 Score is used when the False Negatives and False Positives are crucial. But again, for the multi-class classification model (classes are mutually exclusive) accuracy is most favored. For our model, we have got 0.881 which means our model is approx. 88% accurate. To calculate F1 Score it is required to first calculate the Precision. In our model, the measure for Precision is 0.571,

doi: 10.25100/iyc.v25i3.13071

which is somehow low, but in this proposal, the cost associated with False Positives is not necessarily high related to ATD. A Fact incorrectly predicted as ATD could imply a waste of time when analyzing it, but it would not have maintainability problems.

The measure for the F1 Score is 0.696, which means Rebel's model has low false positives and low false negatives, so the model is correctly identifying real threats and not be disturbed by false alarms.

Discussion

GN allowed us to test the approach in a real industrial context. All steps of this approach were presented to the architect, and relevant feedback was provided by the architect. ArchiMate models were acknowledged to be an important source of knowledge about architecture. ArchiMate allows the software team to focus on a specific part of the architecture with a special interest in the business processes. However, it is not used by the software team in this industrial project.

Related to the artifacts, it was clear that architectural decisions are an important source of knowledge for ATD identification. On the other hand, commit logs and chat logs could reduce the accuracy of the supervised ML model.

In this experimentation, four evaluation metrics were used to measure the quality of our model. The recall is the most important one considering the cost of False Negatives in the set of data to be predicted. Our model, with the limited amount of data, provides good results related to Recall and even Accuracy. This experimentation provides insights allowing us to state that this proposal works well and could be used as starting point for support architects in ATD identification and further ATD management.

It is important to note that the supervised ML model used in this section could be not enough to provide reliable results, and the data could not be related to real software challenges and architectures. Providing an adequate classification model would involve the analysis of thousands of data, and this alone would be a research study. But beyond this, using a real case from industry allows us to understand the value for software architects.

Related work

In the literature we found studies related to the practices used in this approach such as machine learning, analysis of evolution and strategies to deal with ATD management.

ML techniques are effective to carry out specific tasks without relying on explicit instructions or rules. For example, supervised ML techniques have been used to build models that can predict the number of architectural smells in future releases of software systems (10), to build a contextualized vocabulary model based on code comments (11), to model and predict TD evolution (12), and to identify which words are more related to TD in issue trackers (13). As stated by Tsintzira et al. (14), the dominant learning style is supervised learning algorithms (89%), followed by unsupervised (6%) and semi-supervised learning (5%). Tsintzira et al. (14) reported that maintainability and its sub-characteristics (namely: testability, reusability, modifiability, and analyzability) are a common target for ML technologies, followed by business quality attributes.

The evolution (changes) of architectural elements is a central part of this proposal. ATD cases are linked to each of these changes. This practice is also common in the literature, usually in the form of taking historical data such as architectural documentation and version history and used as inputs (5). In other study, Kazman et al. (15) proposed an

approach to identify “architectural roots” of systems by applying a technique based on locating co-changing files. Li et al. (16) proposed modularity metrics related to the number of modified components per commit. In the systematic literature review presented by Verdecchia et al. (5), they reported sixteen studies relying on analyses of the evolution of software systems through time to support ATD identification.

Related to how ATD can be represented, in the literature it is possible to find studies proposing models to represent ATD (9,17). Li et al. (9) proposed an ATD conceptual model for capturing and using ATD in the architecting process. In this conceptual model, the core concept is the ATD item, which acts as the basic unit to record ATD. In this model, an architectural decision incurs on ATD. Our study shares some concepts with this model, such as ATD rationale, compromised quality attribute, and benefits. Verdecchia et al. (17) proposed the core categories of the ATD theory and their relations. In this theoretical model, the core concept is, again, the ATD item and it is generated by a cause, leading to a consequence. An ATD item, in this model, affects artifacts. Our study shares some concepts with this theoretical model, such as: cause (Type in our model), consequences (ATDEffect in our model), and the artifact. In our model, this last concept (artifact) is linked to a Fact, and a Fact is linked to an ATD item.

Several approaches have been proposed to support ATD identification and impact measurement. Verdecchia et al. (18) proposed a technique focused mainly on complementing source code analysis with a review of issue trackers, questions and answer sites (e.g., Stack Overflow), documentation and other software artifacts. However, it is not clear if unstructured sources like emails, chat logs or video/audio transcriptions are also included. In addition, the strategy for analyzing these sources is not described, nor how the impact of the identified ATD could be quantified.

Musil et al. (19) employed sources that can be either organization-internal (e.g., enterprise wikis, AK repositories, AKM tools) or external (e.g., websites, blogs, archival publications) to be automatically indexed by the system. However, they did not explain what techniques are used to extract this information neither to processed it. Also, they were not looking for ATD.

Regarding to ATD management, Li et al. (9) proposed a decision-based approach supporting the five activities of the ATD life cycle: identification, measurement, prioritization, repayment and monitoring. However, identification and measurement require a strong presence of software architects. Besker et al. (3), reported the lack of an overall process on successfully managing ATD in practice and stated the need for further and stronger empirical evidence on the full spectrum of ATD activities.

Conclusions

Architectural technical debt is an essential factor that must be considered during the architectural process, although it is seldom handled at the moment. This work provides a mechanism for detecting ATD based on changes in architectural features and information from heterogeneous artifacts (architectural models, chat logs, emails, and so on).

This technique is centered on the architectural design stage, and it has the capacity to allow automatic identification of ATD types that other ATD identification approaches, mostly based on source code analysis, do not. This method is based on architectural objects and architectural knowledge, both of which must be recorded. This documentation may necessitate some time and effort. The quality of the collected data, as well as the explanation of the architecture and logic, is highly reliant on the accessible artifacts.

doi: 10.25100/iyc.v25i3.13071

It is critical to maintain ATD explicit and visible so that it may be considered as part of the architecture decision-making process, both in terms of its immediate impact and its impact on other architectural decisions. Unidentified ATD objects, on the other hand, will continue to attract attention, resulting in prohibitively high system maintenance and evolution costs.

The analysis of ATD leads us to some important conclusions: i) more artifacts need to be considered in order to increase the understanding of ATD injection, such as functional requirements, quality attribute scenarios, or even a catalog of architecture smells; ii) ATD identification has a close relationship with architectural knowledge. To the extent that knowledge can be standardized, identification can also be standardized; and iii) a fully automated approach for ATD identification is still elusive. It is required to include more data and be fully working with general (domain-independent) and domain-specific architectures.

Additionally, this approach still required some improvements in order to be fully used by the software industry. Future work on this proposal could bring, in the near time, a tool closer to the architects and to the development process. This approach serves as a starting point for ATD identification. In the future, we plan to improve our current work by expanding it in several directions: i) include more empirical studies for validating our ATD identification approach. More industrial projects with different sizes and from various domains are required; ii) include a mapping between architectural elements and code elements. This improvement would be useful to agile projects where the architecture could change in every sprint, and some code is delivered; and iii) inclusion of software architectural models represented in UML. This improvement will provide a better relationship between business goals and the specific software architectures used to accomplish them. This inclusion could begin with the Component & Connector and Deployment diagrams. This improvement will support software architects not familiar with the ArchiMate modeling language.

References

(1) Martini A, Sikander E, Madlani N. A semi-automated framework for the identification and estimation of Architectural Technical Debt: A comparative case-study on the modularization of a software component. Information and Software Technology. 2018 Jan; 93: 264–79. Available from: https://dl.acm.org/doi/abs/10.1016/j.infsof.2017.08.005

(2) Ernst NA, Bellomo S, Ozkaya I, Nord RL, Gorton I. Measure it? Manage it? Ignore it? software practitioners and technical debt. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. 2015 Aug 30 [cited 2023 Apr 19]; 50–60. Available from: https://dl.acm.org/doi/10.1145/2786805.2786848

(3) Besker T, Martini A, Bosch J. Managing architectural technical debt: A unified model and systematic literature review. Journal of Systems and Software. 2018 Jan [cited 2019 Nov 4];135:1–16. Available from: https://www.sciencedirect.com/science/article/abs/pii/S0164121217302121

(4) Li Z, Avgeriou P, Liang P. A systematic mapping study on technical debt and its management. Journal of Systems and Software. 2015 Mar;101:193–220. Available from: https://www.sciencedirect.com/science/article/abs/pii/S0164121214002854

(5) Verdecchia R, Malavolta I, Lago P. Architectural Technical Debt Identification: The Research Landscape. In: 2018 IEEE/ACM International Conference on Technical Debt (TechDebt). 2018 [cited 2023 Jul 14]. p. 11–20. Available from: https://ieeexplore.ieee.org/document/8595095

(6) Martini A, Besker T, Bosch J. The Introduction of Technical Debt Tracking in Large Companies. In: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC). 2016 [cited 2023 Jul 14]. p. 161–168. Available from: https://ieeexplore.ieee.org/document/7890584

(7) Nord RL, Ozkaya I, Kruchten P, Gonzalez-Rojas M. In Search of a Metric for Managing Architectural Technical Debt. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture. IEEE Xplore; 2012. p. 91–100. Available from: https://ieeexplore.ieee.org/abstract/document/6337765

(8) Li Z, Liang P, Avgeriou P. Architectural Technical Debt Identification Based on Architecture Decisions and Change Scenarios. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture. 2015. p. 65–74. Available from: https://ieeexplore.ieee.org/document/7158505

(9) Li Z, Liang P, Avgeriou P. Chapter 9 - Architectural Debt Management in Value-Oriented Architecting. In: Mistrik I, Bahsoon R, Kazman R, Zhang Y, editors. Economics-Driven Software Architecture. Boston: Morgan Kaufmann; 2014 [cited 2023 Jul 14]. p. 183–204. Available from: https://www.sciencedirect.com/science/article/pii/B978012410464800009X

(10) Diaz-Pace J. Andres, Tommasel A, Godoy D. [Research Paper] Towards Anticipation of Architectural Smells Using Link Prediction Techniques. In: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE Xplore; 2018 [cited 2023 Jul 14]. p. 62–71. Available from: https://ieeexplore.ieee.org/document/8530719

(11) de A, Gomes M, Luiz A, Spínola RO. A Contextualized Vocabulary Model for identifying technical debt on code comments. IEEE 7th International Workshop on Managing Technical Debt (MTD). 2015 [cited 2023 Jun 15]. p. 25 – 32. Available from: https://ieeexplore.ieee.org/document/7332621

(12) Tsoukalas D, Kehagias D, Siavvas M, Chatzigeorgiou A. Technical debt forecasting: An empirical study on open-source repositories. Journal of Systems and Software. 2020 Dec [cited 2021 Mar 10]; 170:110777. Available from: https://www.sciencedirect.com/science/article/abs/pii/S0164121220301904

(13) Dai K. Identifying technical debt through issue trackers. Vancouver: University of British Columbia Library. [Vancouver: University of British Columbia Library]: University of British Columbia; 2009 [cited 2019 Oct 11]. Available from: https://open.library.ubc.ca/cIRcle/collections/ubctheses/24/items/1.0374920

(14) Tsintzira A, Arvanitou E, Ampatzoglou A, Chatzigeorgiou A. Applying Machine Learning in Technical Debt Management: Future Opportunities and Challenges. In: International Conference on the Quality of Information and Communications Technology. 2020 [cited 2023 Jul 14]. p. 53–67. Available from: https://link.springer.com/chapter/10.1007/978-3-030-58793-2_5

(15) Kazman R, Cai Y, Mo R, Feng Q, Xiao L, Serge Haziyev, et al. A Case Study in Locating the Architectural Roots of Technical Debt. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. 2015 [cited 2023 Jul 14]. p. 179–88. Available from: https://ieeexplore.ieee.org/document/7202962

doi: 10.25100/iyc.v25i3.13071

(16) Zeng Z, Liang P, Avgeriou P, Guelfi N, Apostolos Ampatzoglou. An empirical investigation of modularity metrics for indicating architectural technical debt. In: 10th international ACM Sigsoft conference on Quality of software architectures. 2014. p. 119–28.

(17) Verdecchia R, Kruchten P, Lago P, Malavolta I. Building and evaluating a theory of architectural technical debt in software-intensive systems. Journal of Systems and Software. 2021 Jun;176:110925.

(18) Verdecchia R. Architectural Technical Debt Identification: Moving Forward. In: IEEE International Conference on Software Architecture Companion (ICSA-C). 2018 [cited 2023 Jul 14]. p. 43–4. Available from: https://ieeexplore.ieee.org/document/8432173

(19) Musil J, Ekaputra FJ, Sabou M, Ionescu TC, Schall D, Musil A, et al. Continuous Architectural Knowledge Integration: Making Heterogeneous Architectural Knowledge Available in Large-Scale Organizations. In: IEEE International Conference on Software Architecture (ICSA). 2017 [cited 2023 Jul 14]. p. 189–92. Available from: https://ieeexplore.ieee.org/document/7930216