How Effective Is Automated Trace Link Recovery in Model-Driven Development?

Randell Rasiman, Fabiano Dalpiaz(✉), and Sergio España

Utrecht University, Utrecht, The Netherlands
{f.dalpiaz,s.espana}@uu.nl

Abstract. [Context and Motivation] Requirements Traceability (RT) aims to follow and describe the lifecycle of a requirement. RT is employed either because it is mandated, or because the product team perceives benefits. **[Problem]** RT practices such as the establishment and maintenance of trace links are generally carried out manually, thereby being prone to mistakes, vulnerable to changes, time-consuming, and difficult to maintain. Automated tracing tools have been proposed; yet, their adoption is low, often because of the limited evidence of their effectiveness. We focus on vertical traceability that links artifacts having different levels of abstraction. **[Results]** We design an automated tool for recovering traces between JIRA issues (user stories and bugs) and revisions in a model-driven development (MDD) context. Based on existing literature that uses process and text-based data, we created 123 features to train a machine learning classifier. This classifier was validated via three MDD industry datasets. For a trace recommendation scenario, we obtained an average F_2-score of 69% with the best tested configuration. For an automated trace maintenance scenario, we obtained an $F_{0.5}$-score of 76%. **[Contribution]** Our findings provide insights on the effectiveness of state-of-the-art trace link recovery techniques in an MDD context by using real-world data from a large company in the field of low-code development.

Keywords: Requirement traceability · Trace link recovery · Model-driven development · Low-code development · Machine learning

1 Introduction

Requirements Trace Link Recovery (RTR) is the process of establishing trace links between a requirement and another trace artefact [13]. Many techniques for (requirements) trace link recovery propose semi-automatic processes that rely on information retrieval (IR) [2]. The premise of IR-based approaches is that when two artefacts have high a degree of textual similarity, they should most likely be traced [18]. Commonly used IR algorithms include Vector Space Models, Latent Semantic Indexing, Jenson-Shannon Models, and Latent Dirichlet Allocation [2,5].

More recently, developments from Machine Learning (ML) have been employed in automatic Trace Link Recovery (TLR) [2]. ML approaches treat TLR as a classification problem: the Cartesian product of the two trace artefact sets defines the space of candidate trace links [11,16], a subset of which are valid links (manually defined by

V. Gervasi and A. Vogelsang (Eds.): REFSQ 2022, LNCS 13216, pp. 35–51, 2022.
https://doi.org/10.1007/978-3-030-98464-9_4

the domain experts). A ML classifier is tasked to build a model for predicting whether unseen trace links are valid or invalid. This is achieved by representing the trace links as a vector, derived from features. Most ML TLR approaches use similarity scores of IR-based methods as features [11,16,23] and outperform IR-based TLR approaches [16].

However, in most of the studies, the classifiers are trained either using open-source datasets from universities, or proprietary data regarding safety-critical systems, and this entails an external validity concern [5]. Although using the same datasets is useful for benchmarking and for comparing methods, it poses the risk that the new traceability tools are being over-optimised for these specific datasets. To advance the current state-of-the-art traceability tools, the research community has called for gaining feedback from additional industrial datasets in a broad range of application domains [2,5].

In this paper, we aim to acquire new insights on automated RTR in a model-driven development (MDD) context, a domain which has shown potential for RT integration [27]. Following the Design Science research methodology [26], we conduct a case study at Mendix, a large-scale MDD-platform producer, and we develop a software tool for automated RTR that focuses on vertical traceability [21], which allows for the automated recovery of trace links between artifacts at different abstraction levels. The main contributions of this research are:

1. We provide new insights on the application of RTR in MDD, narrowing the gap between academic research and industrial demands, and moving steps toward the vision of ubiquitous requirements traceability [14].
2. To the best of our knowledge, this is the first study that experiments with the use of Gradient Boosted Trees for RTR.
3. We evaluate the relative importance of four families of features for establishing trace links between requirements (represented as JIRA issues) and model changes (commit files generated by the Mendix Studio low-code development platform).

We follow the recommendations of context-driven research [6]: specifying working assumptions based on a real-world context in order to attain practicality and scalability. We do so by collaborating with Mendix, which allowed us to use their data and to obtain rich insights on their development processes and the possible role of traceability.

This paper is structured as follows: Sect. 2 presents the background on requirements traceability. Section 3 describes how MDD and requirements are supported with the Mendix Studio platform within the Mendix company. Section 4 presents the construction of our automated RTR classifier. Section 5 shows the results, while Sect. 6 discusses the threats to validity. Finally, Sect. 7 concludes and outlines future work.

2 Related Work on Automated RTR

RT practices are mandated by well-known standards such as CMM, ISO 9000, and IEEE 830-1998 [4,9]. Thus, organisations who aim to comply with such standards embrace RT practices. These are expected to deliver benefits for project management and visibility, project maintenance, and verification & validation. Despite the clear benefits, the practice itself is not evident. RT activities are found to be “time-consuming, tedious and fallible" [25]. Even when conducted, manual tracing is favoured, leading to traces which are error-prone, vulnerable to changes, and hard to maintain.

Information Retrieval. For this reason, a considerable amount of RT research focuses on automating the task. Many of the proposed IR-based methods employ Vector Space Models (VSM), which use the cosine distance to measure the semantic similarity between documents. An alternative is the Jenson-Shannon Models (JSM), which consider documents as a probabilistic distribution [1,8], and the Jenson-Shannon Divergence as a measure of the semantic difference. There are two fundamental problems in IR-methods. *Synonymy* refers to using different terms for the same concept (e.g., 'drawing' and 'illustration'), and this decreases the recall. *Polysemy* refers to using terms that have multiple meanings (e.g. 'fall'), and this decreases precision [10]. Latent Semantic Indexing (LSI) aims to solve this problem by replacing the *latent semantics* (what terms actually mean) to an implicit higher-order structure, called latent semantics. This latent structure can then be used as feature set, which better reflects major associative data patterns and ignores less important influences. An example of this approach is the work by Port *et al.* [19]. Although other approaches have further improved performance, the performance gain has flattened, and more recent works make use of machine learning.

Machine Learning. Most state-of-the-art techniques for RTR employ ML nowadays, taking the field to new levels. ML approaches treat the TLR process as a classification problem: the Cartesian product of the two trace artefact sets is calculated, and the resulting elements represent candidate trace links [11,16]. A ML classifier learns from sample data, which is manually traced, and the classifier is then used to predict whether unseen couples of artefacts should be traced to one another. Most ML TLR approaches use the similarity scores from IR-based methods as features [11,16,23], although other features have been proposed. Besides feature representation, researchers have also analysed which ML classification algorithms would perform best. Falessi *et al.* [12] have compared multiple algorithms: decision trees, random forest, naïve Bayes, logistic regression, and bagging, with random forests yielding the best results.

Deep Learning. Recent advances in neural networks can also be employed in automated TLR [15]. Although this an interesting direction with the potential of achieving excellent results, neural networks are only suitable when large datasets are available. This is not the case in many industrial situations, like the one described in this paper.

3 Case Study at Mendix

We conducted a case study at Mendix, the producer of the Mendix Studio Low-Code Platform (MLCP). The MLCP employs MDD principles and allows creating software by defining graphical models for the domain, business logic, and user interface [24]. We study MLCP developers employed by Mendix, who are building applications with the MLCP for Mendix itself. These developers follow the SCRUM development process. Product Owners are responsible for managing and refining requirements, which are documented as JIRA issues and are added to the product backlog. The issues for the Sprint Backlog are chosen by the MLCP development team. Each selected item is assigned to one MCLP developer during a sprint, who is responsible for implementation.

The implementation is broken down into several activities. First, the MCLP developer examines the JIRA issue to become familiar with it. Second, the MCLP developer opens the latest MLCP model, navigates to the relevant modules, and makes the required changes. These changes are stored in a revision and are committed to the repository once they to fulfil the JIRA issue's acceptance criteria. Each revision is supplemented with a log message, in which the MCLP developer outlines the changes he or she made, as well as the JIRA issue ID for traceability purposes.

3.1 Studied Artefacts

We focus on tracing *JIRA* issues to committed revisions, because manual trace information was available from some development teams who followed traceability practices. Figure 1 shows the relationships among the trace artefacts.

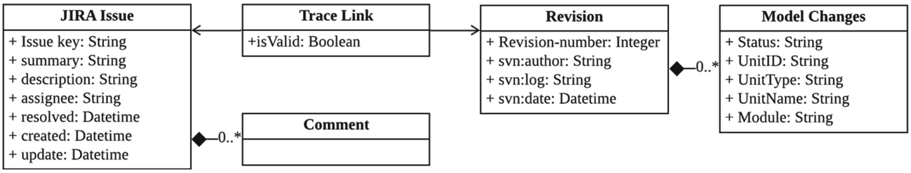


Fig. 1. Model showing the relationships between JIRA issues and revisions

JIRA Issues. Several teams at Mendix use the widespread project management tool Atlassian JIRA. In JIRA, project members define work items called issues, which Mendix uses to document requirements. The following attributes are shared by all JIRA issues: I1) a unique *issue key* serving as identifier, I2) a *summary*, used by Mendix to document a user story written in the Connextra template, I3) a *description*, which further explains the requirements alongside the acceptance criteria, I4) an *assignee*: the person who is responsible for implementing the issue. Finally, each issue has three date/time indicating when the issue was I5) created, I6) last updated, and I7) resolved.

Revisions. The MLCP, like any modern development environment, employs version control. An adapted version of Apache Subversion is integrated into the MLCP, which the developer can access through a GUI. Each revision contains: R1) *revision-number*, a unique integer, R2) *author*, the email of the person who committed the *revision*, R3) *log*, an optional field to write text, and R4) *date*, the date/time when the revision was committed. Finally, each revision contains the changes made to the units, which are stored as an element of an array that contains R5) *unitID*, R6) the *status* (either added, deleted, or modified), R7) *unitName*: the name of that unit, R8) *unitType*: the category of the unit (e.g., microflow or form), R9) *module*, the module where the unit is located.

3.2 Studied Datasets

We acquired data from three internal MLCP projects, produced by two development teams. We refer to them as i) *Service*, ii) *Data*, and iii) *Store*. For each project, we used a data export of one JIRA project and one MLCP repository. We analysed the availability of manual traces (see Table 1). We distinguished between revisions that trace to a single issue, to two or more issues, and to no issues. A large percentage of revisions is untraced. This could be because the revision is too generic (e.g., creation of a branch), or because the developer forgot about tracing. Also, the revisions were not always traced to issue keys of the JIRA projects we acquired. This happens because multiple teams, each with their own JIRA project, may operate on the same repository.

Table 1. Summary of the acquired project data

Dataset	Service	Data	Store
Total JIRA issues	173	58	634
Total revisions	2,930	818	713
Revisions traced to 1 issue	1,462 (49.90%)	556 (67.97%)	202 (28.33%)
Revisions traced to 2+ issues	33 (1.13%)	26 (3.18%)	3 (0.42%)
Revisions traced to no issues	1,435 (48.98%)	236 (28.85%)	508 (71.25%)

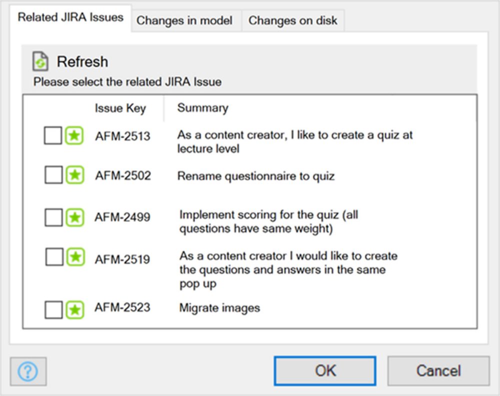
3.3 Objective and Evaluation Scenarios

Our objective is to automate the MLCP developers' tracing process, which is currently manual. We adapt the two scenarios put forward by Rath *et al.* [23]: *Trace Recommendation* and *Trace Maintenance*. Our automated artefact is evaluated for both scenarios using a common traceability metric, the F-measure, which quantifies the harmonic mean between precision and recall. However, in line with Berry's recommendations [3], we employ adjusted versions of the F-measure, as described below.

Trace Recommendation. MLCP developers use a GUI to commit changes to the remote repository. When doing this, the developer outlines the changes made and writes an issue key out of those in JIRA. Integrating a trace recommendation system can improve this scenario (see Fig. 2): the issues that the developer may choose among can be filtered based on the likelihood for that issue to be linked to the current revision. Only those issues above a certain threshold are shown.

The only manual task left for the developer is to vet the trace links. It is cognitively affordable and relatively fast since developers generally know which specific JIRA issue they have implemented. This scenario requires a high level of recall, for valid traces must be present in the list for a developer to vet it. Precision is less important because developers can ignore invalid traces. Therefore, in this scenario, we evaluate the system using the F_2-measure, an F-measure variant favouring recall above precision.

Trace Maintenance. Not all the revisions are traced to a JIRA issue. As visible in the last row of Table 1, between 28% and 71% of the revisions were not traced to issues. Thus, maintenance is needed to recover traces for the untraced revisions, which leads to the goal of the second scenario: an automated trace maintenance system. Such a system would periodically recover traces that were forgotten by the developer, ultimately leading to a higher level of RT. No human intervention is foreseen to correct invalid traces, so precision needs to be favoured above recall. Thus, we evaluate the system using the $F_{0.5}$-measure.



Please select the related JIRA Issue

	Issue Key	Summary
□	AFM-2513	As a content creator, I like to create a quiz at lecture level
□	AFM-2502	Rename questionnaire to quiz
□	AFM-2499	Implement scoring for the quiz (all questions have same weight)
□	AFM-2519	As a content creator I would like to create the questions and answers in the same pop up
□	AFM-2523	Migrate images

Fig. 2. Mock-up of a trace recommendation system

4 Requirement Trace Link Classifier

To accommodate both scenarios, we present an ML classifier to classify the validity of traces, based on the TRAIL framework [16]. Our classifier, which we call *LCDTrace*, is publicly available as open source[1], and a permanent copy of the version used in this paper is part of our online appendix [22].

After describing the dataset the data available at Mendix for training, and how we pre-processed it, we describe the feature engineering process, data rebalancing, and the employed classification algorithms.

4.1 Data Description and Trace Construction

To train the ML classifier, we used the data from the Service, Data and Store datasets.

Revisions. The data was provided in text-format. We used Regular Expressions to transform the data and to extract the issue key(s) from the log message and store it in a distinct issue key column. After extraction, the issue key was removed from the log message, and the log message was pre-processed using common pre-processing steps: 1) all words were lowercased, 2) all the interpunction was removed, 3) all numeric characters were removed, 4) all sentences were tokenised with NLTK, 5) the corpus from NLTK was used to eliminate all stop words, and 6) all remaining terms were stemmed using the Porter Stemming Algorithm [20]. These activities resulted in a pre-processed dataset that consists of (labels were defined in Sect. 3.1): R1 (Revision Number), R2 (Author), R3 (Log), R4 (Date), R7 (Unit Names), R8 (merge of log and unit names), and associated JIRA key (a reference to I1).

JIRA Issues. The JIRA datasets were provided as delimited text files. Pre-processing was carried out in the same manner as for the revisions. This led to a dataset that consists of I1 (Issue key), I2 (Summary), I3 (Description), I4 (Assignee), I5 (Created date),

[1] https://github.com/RELabUU/LCDTrace.

I6 (Last updated date), I7 (Resolved date), plus one additional feature: I9 (JIRA All-Natural Text): the union of I2 and I3.

Trace Link Construction. Because a classifier can only be trained using labelled data, we discarded data that were not traced to issues. For the remaining data, we calculated the Cartesian product between the JIRA project dataset and the repository dataset. Each element is a candidate trace link whose validity was determined by comparing the issue key to the revision's related issue key. If the issue key was present, the trace link was classified as valid; else, the trace link was classified as invalid. Also, we applied causality filtering to the trace links [23]: when a trace link had revision antecedent to the creation of an issue, it was deemed invalid due to causality. The result is in Table 2.

Table 2. Valid and invalid traces before and after applying causal filtering to the project data

Dataset	Causality filtering	Total traces	Invalid traces	Valid traces
Service	Before	258,635	258,215 (99.84%)	420 (0.16%)
	After	89,233	88,813 (99.53%)	420 (0.47%)
Data	Before	33,756	33,305 (98.66%)	451 (1.34%)
	After	27,815	27,364 (98.38%)	451 (1.62%)
Store	Before	129,970	129,884 (99.93%)	86 (0.07%)
	After	33,627	33,541 (99.74%)	86 (0.26%)

4.2 Feature Engineering

The previously produced candidate trace links were then used for training the classifier. For this, we had to represent the candidate trace links as a set of features. Based on literature in the field, we engineered a total of 123 features grouped into four categories: process-related, document statistics, information retrieval and query quality.

Process-Related. These four features build on Rath *et al.* [23]. F1, the first feature, captures stakeholder information by indicating if the assignee of a JIRA issue is the same person as the author of a revision. The remaining three features capture temporal information. F2 is the difference between the date of revision (R4) and the date the issue was created (I5), F3 is the difference between R4 and the date the issue was last updated (I6), and F4 is the difference between R4 and the date the JIRA issue was resolved (I7).

Document Statistics. These features rely on the work of Mills *et al.* [16]: they gauge document relevance and the information contained within the documents. Within this category, seven metrics (hence, 7 features) are included:

- *Total number of terms*, calculated for the JIRA issue (F5) and the revision (F6).
- *Total number of unique terms* for the JIRA issue (F7) and the revision (F8).
- *Overlap of terms between the JIRA issue and the revision.* To calculate this metric, the overlap of terms is divided by the set of terms that are compared to. This is done in three ways, each leading to a feature: F9 divides the overlap of terms by the terms in the JIRA issue, F10 divides is by the terms in the revision, and F11 divides it by the union of the terms in the JIRA issue and in the revision.

Information Retrieval. This feature set captures the semantic similarity between two trace artefacts. We first apply VSM with TF-IDF weighting to transform the trace artefacts to a vector representation. Because we use TF-IDF weighting, the chosen corpus used for weighting impacts the resulting vector. For instance, the term 'want' occurs commonly in the JIRA summary, for Mendix developers put their user story in there. However, it might be a rare term when considering all the terms in a JIRA issue. Since we could not determine which corpus best represents the trace artefact, we opted to explore multiple representations: we have constructed three issues vector representation (I2: Summary, I3: Description, I9: Summary & Description) and three representations for the revisions (R3: log message, R7: unit names, and R8: log & unit names). This results in 9 distinct pairs for each trace link candidate, as shown in Table 3. The cosine similarity of each pair was computed and utilised as a feature. Mills and Haiduc [17] showed that the chosen trace direction (i.e., which artefact in the trace link is used as a query) affect traceability performance. Thus, we calculated the cosine distance in either direction, resulting in a total of 18 IR-features (F12–F29) in Table 3. We used Scikit-learn for TF-IDF weighting and SciPy for calculating the cosine distance.

Table 3. TF-IDF combinations used for VSM

ID	Revision artefact	Issue artefact	Features
1	Log message	Summary	F12–F13
2	Log message	Description	F14–F15
3	Log message	JIRA all-natural text	F16–F17
4	Unit names	Summary	F18–F19
5	Unit names	Description	F20–F21
6	Unit names	JIRA all-natural text	F22–F23
7	Revision all-natural text	Summary	F24–F25
8	Revision all-natural text	Description	F26–F27
9	Revision all-natural text	JIRA all-natural text	F28–F29

Query Quality. The quality of a query determines how well a query is expected to retrieve relevant documents from a document collection. A high-quality query returns the relevant document(s) towards the top of the results lists, whereas a low-quality query returns them near the bottom of the list or not at all. It is important to differentiate between high- and low-quality queries, when using IR-techniques for TLR. Do two artefacts have a low cosine similarity because they are actually invalid, or is it because the similarity was computed using a low-quality query?

Mills and Haiduc [17] devised metrics for measuring query quality (QQ). These QQ metrics are organised into pre-retrieval and post-retrieval metrics. Pre-retrieval metrics merely consider the properties of the query, whereas post-retrieval metrics also consider the information captured by the list returned by the executed query. We focused on implementing pre-retrieval QQ metrics in this study, evaluating three different aspects:

- *Specificity* refers the query's ability to express the relevant documents and to distinguish them from irrelevant documents. Highly-specific queries contain terms which

are rare in the document collection, while lowly-specific queries contain common terms. Highly specific queries are desired, for documents can be differentiated based on the terms.
- *Similarity* refers to the degree to which the query is similar to the document collection. Queries that are comparable to the collection suggest the existence of many relevant documents, increasing the possibility that a relevant document is returned.
- *Term relatedness* refers to how often terms in the query co-occur in the document collection. If query terms co-occur in the document collection as well, the query is considered of high quality.

The computation of these metrics was executed for the six corpora mentioned in the information retrieval paragraph (log message, unit names, revision all-natural text, summary, description, and JIRA all-natural text), because the outcome of the metrics depends on the corpus of which the query is a part. This resulted in a total of 102 QQ features: F30–F131, listed in Table 4.

Table 4. Query quality features from the work by Mills and Haiduc [17]

Family	Measure	Metric	Features	
			Query: Revision	Query: JIRA
Specificity	TF-IDF	{Avg, Max, Std-Dev}	F30–F38	F39–F47
	TF-ICTF	{Avg, Max, Std-Dev}	F48–F56	F57–F65
	Entropy	{Avg, Med, Max, Std-Dev}	F66–F77	F78–F89
	Query Scope		F90–F92	F93–F95
	Kullback-Leiber divergence		F96–F98	F99–F101
Similarity	SCQ	{Avg, Max, Sum}	F102–F110	F111–F119
Relatedness	PMI	{Avg, Max}	F120–F125	F126–F131

4.3 Data Rebalancing

In traceability settings, the training data is generally highly imbalanced because only a few valid links exist [15,23], making classifier training problematic [23]. Table 2 shows this occurs in our datasets too, with a percentage of valid links between 0.26% and 1.62%. The positive samples that the classifier would view are quite low, compared to the negative ones. Thus, we applied four rebalancing strategies [16] to the training data:

1. *None.* There is no rebalancing method applied to the data.
2. *Oversampling.* The minority class is oversampled until it reaches the size of the majority class, by applying SMOTE.
3. *Undersampling.* The majority class is randomly undersampled until it has the same size as the minority class, by applying the random undersampling technique.
4. *5050.* Oversampling via SMOTE is applied to the minority class with a sampling strategy of 0.5. Then undersampling is applied to the majority class until the sizes of both classes are equal.

4.4 Classification Algorithms

We considered two state-of-the-art supervised ML algorithms for classifying trace links as valid or invalid: Random Forests and Gradient Boosted Decision Trees. While the former are shown to be the best RTR classifier in earlier research [16,23], Gradient Boosted Decision Trees outperformed Random Forests in other domains [7,29]. To implement the Random Forest algorithm, we used the framework of Scikit-learn. To implement the Gradient Boosted Decision Trees we used two different frameworks: XGBoost, and LightGBM. These frameworks differ in two major respects. The first distinction is in the method of splitting. XGBoost splits the tree level-wise rather than leaf-wise, whereas LightGBM splits the tree leaf-wise. The second difference is how best split value is determined. XGBoost uses a histogram-based algorithm, which splits a feature and its data points into discrete bins, which are used to find the best split value. LightGBM uses a subset of the training data rather than the entire training dataset. Its sampling technique uses gradients, resulting in significantly faster training times.

5 Results

We performed an evaluation on the different combinations of the rebalancing strategies of Sect. 4.3 and of the classification algorithms of Sect. 4.4. This evaluation was conducted for each dataset independently by dividing each dataset into a training (80%) and testing (20%) sets using stratified sampling, so that the two sets have a comparable proportion of positives and negatives. Due to insufficient memory, we use only 4 out of the 12 relatedness-based QQ features listed in Table 4, leading to a total of 123 features.

To mitigate randomisation effects, we repeated the evaluation (training-testing set splitting, classifier training on the 80%, testing on the 20%) for 25 times, then we averaged the outputs, leading to the results we show in Sect. 5.1. In addition to the quantitative results, we discuss the relative importance of the features in Sect. 5.2.

5.1 Quantitative Results

Table 5 shows the precision, the recall, and the $F_{0.5}$- and F_2-measure for the results, which were obtained using non-normalised data. The table compares the three algorithms (Random Forests, XGBoost, LightGBM) that are visualised as macro-columns; the results for each project are presented in a different set of rows. Per project, the results are shown by showing, one per line, the four rebalancing strategies (none, oversampling, undersampling, 5050). The results for the normalised data were found to be slightly worse, and are therefore only included in the online appendix.

For the trace recommendation scenario, XGBoost ($\overline{x} = 56.25$) has the highest mean F_2 across all rebalancing strategies. LightGBM follows ($\overline{x}$ = 55.16), and Random Forests are the least effective ($\overline{x} = 42.24$). This is interesting, for Random Forests have consistently been found to be the best performing algorithm in prior RTR research [16,23]. This finding indicates that, similar to other tasks [7,29], Gradient Boosted Decision Trees can outperform Random Forests in RTR-tasks too. A similar result holds for the trace maintenance scenario ($F_{0.5}$), where XGBoost ($\overline{x} = 55.45$) performs best, and LightGBM achieves results that are as low as those of random forests.

Table 5. Mean precision, recall, and $F_{0.5}$- (trace maintenance scenario) and F_2-measure (trace recommendation) across all 3 datasets. The green-coloured cells indicate the best results per each dataset. For accuracy and readability, the table shows F-scores in percentage.

Proj.	Rebal.	Random Forests				XGBoost				LightGBM			
Service	None	94.96	19.71	53.13	23.37	81.77	48.86	**71.89**	53.07	64.56	48.62	60.45	51.07
	Over	5.90	95.52	7.26	23.61	6.98	96.33	8.56	27.01	6.59	97.62	8.10	25.92
	Under	69.12	44.67	62.17	48.01	70.23	60.24	67.89	61.94	60.02	65.71	61.02	64.42
	5050	59.59	54.33	58.41	55.27	59.62	69.86	61.37	**67.47**	53.49	72.10	56.34	67.31
Data	None	90.34	29.78	63.91	34.35	84.87	62.65	**79.21**	66.09	82.50	61.75	77.24	64.98
	Over	16.42	92.04	19.65	47.84	20.28	94.44	24.05	54.50	20.01	94.11	23.74	53.99
	Under	75.52	48.33	67.78	52.03	77.08	69.27	75.34	70.68	70.67	69.96	70.47	70.05
	5050	62.33	54.51	60.52	55.86	65.96	74.98	67.54	72.94	63.22	76.26	65.42	**73.19**
Store	None	93.13	42.12	73.66	46.99	86.56	59.06	**78.77**	62.85	46.78	47.53	45.51	45.27
	Over	4.31	90.35	5.32	17.96	2.51	90.35	3.12	11.23	2.98	92.47	3.70	13.17
	Under	72.61	44.47	63.21	47.70	70.51	62.59	68.02	63.42	69.43	65.18	68.18	65.67
	5050	65.31	52.00	61.58	53.84	58.84	65.88	59.63	63.73	55.34	71.06	57.68	**66.89**
Macro-Avg	None	92.81	30.54	63.57	34.90	84.40	56.86	**76.62**	60.67	64.61	52.63	61.07	53.77
	Over	8.88	92.64	10.74	29.80	9.92	93.71	11.91	30.91	9.86	94.73	11.85	31.03
	Under	72.42	45.82	64.39	49.25	72.61	64.03	70.42	65.35	66.71	66.95	66.56	66.71
	5050	62.41	53.61	60.17	54.99	61.47	70.24	62.85	68.05	57.35	73.14	59.81	**69.13**
	Mean	***59.13***	***55.65***	***49.72***	***42.24***	***57.10***	***71.21***	***55.45***	***56.25***	***49.63***	***71.86***	***49.82***	***55.16***

Also, our findings show that the rebalancing strategy has a greater effect than the classification algorithm. With no rebalancing, we achieve the highest precision in 11/12 combinations (algorithm × dataset), with the only exception of LightGBM on the Store dataset. So, for the trace maintenance scenario, no oversampling is the best option.

SMOTE oversampling reduces precision and increases recall: in extreme cases where recall is considerably more important than precision (missing a valid trace is critical and the cost of vetting many invalid candidates is low), it may be a viable option. However, for our two scenarios with $F_{0.5}$ and F_2, SMOTE is the worst alternative.

When we use undersampling for rebalancing, we get a better trade-off than when we use oversampling: the recall increases with respect to no re-balancing, at the expense of precision. However, the decrease in precision is less substantial than for oversampling.

The 5050 rebalancing strategy improves this balance by trading recall for precision. As a result, the classifiers using this rebalancing strategy preserve high recall while offering a more practical precision. The F_2-measure quantifies this: 5050 rebalancing is the best alternative for the trace recommendation scenario.

When taking both the rebalancing and classification algorithm into account, we achieve highest F2-score by combining LightGBM with 5050 rebalancing ($\overline{x} = 69.13$), making it the best configuration for trace recommendation. The XGBoost/5050 combination is, however, very close, and slightly outperforms LightGBM/5050 for the Service dataset. For the Trace Recommendation scenario, we get the best by combining XGBoost with no data rebalancing, which achieves a mean F0.5-measure of 76.62.

5.2 Features Importance

We report on the feature importance to contribute to the model's explainability. We consider the average gain of each feature category, as defined in Sect. 4.2, with QQ broken down into its subcategories due to the many features. The cumulative (total), max, and average gain is shown in Table 6, while Fig. 3 presents them visually.

Table 6. The total, max, and average gain (in percentage over the total gain given by all features) per feature category for the trace recommendation and trace maintenance scenarios.

		Trace recommendation			Trace maintenance		
		Total	Max	Avg	Total	Max	Avg
Process-related	Service	30.79	26.14	7.70	11.43	4.66	2.86
	Data	52.61	32.14	13.15	10.93	3.86	2.73
	Store	7.61	4.48	1.19	5.14	1.705	1.29
Information retrieval	Service	52.82	49.33	2.94	17.83	3.04	0.99
	Data	20.29	15.45	1.12	19.99	2.97	1.11
	Store	46.81	42.71	2.60	14.20	2.46	0.79
Document statistics	Service	3.20	1.76	0.46	7.60	2.17	1.09
	Data	4.08	1.34	0.58	5.06	1.66	0.72
	Store	3.67	1.75	0.52	15.66	8.04	2.23
Query quality (Specificity)	Service	10.59	2.20	0.15	51.01	1.71	0.71
	Data	18.89	4.89	0.26	51.51	5.08	0.72
	Store	39.17	19.85	0.54	51.97	2.96	0.72
Query quality (Similarity)	Service	2.35	0.45	0.13	9.93	1.59	0.55
	Data	3.03	0.59	0.17	10.14	2.35	0.56
	Store	2.54	0.59	0.14	11.65	1.94	0.65
Query quality (Term Relatedness)	Service	0.25	0.14	0.06	2.20	0.74	0.55
	Data	1.09	0.75	0.27	2.37	1.01	0.59
	Store	0.20	0.16	0.05	1.38	0.70	0.34

In the Trace Recommendation scenario, we see that process-related feature categories are important in the Service and Data projects, with gains of 30.79 and 52.61, respectively. Further investigation reveals that the top two process-related features for Service and Data are F4: the difference between the date the issue was resolved and the revision date (18.99 for Data, 26.14 for Service) and F1: whether the issue assignee is the same person who committed the revision (32.14 for Data, 3.8 for Service).

Process-related features contribute much less for the Store dataset, in both scenarios. One explanation is that Service and Data are produced by a different development team than Store. Both teams may have a different level of discipline when it comes to managing JIRA-problems (i.e., promptly updating the status of JIRA issues), resulting in a different level of importance for this feature category.

The Information Retrieval feature category is shown to be important for the Trace recommendation scenario, with total Gains of 52.82, 20.29, and 46.81. Similar to the Process-related feature category, the majority of this increase comes from a single feature, which is the cosine similarity between all-text from a revision and a JIRA-issue summary, utilising summary as a query (F25) for all three datasets. This means that a TF-IDF representation of merely the JIRA issues via the summary is better for the model than a combination of summary and description.

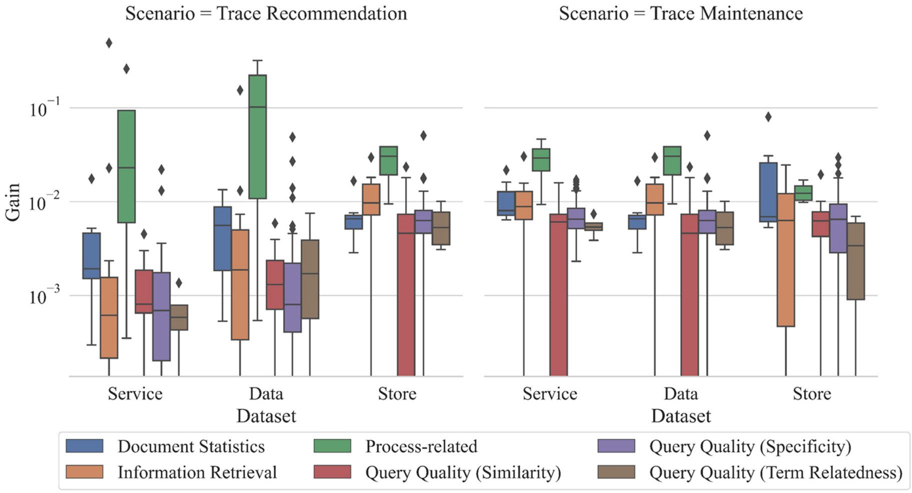


Fig. 3. Average gain per feature family for the trace recommendation scenario (left) and for the trace maintenance scenario (right). The y-axis uses an exponential scale to improve readability.

Furthermore, we find that this feature category is less important in the trace maintenance scenario, with each unique feature contributing more evenly.

Table 6 also reveals that the Document Statistics feature category have a low total gain. Figure 3, however, shows that the average gain per feature in this category is rather significant. Because of this finding, the cost-benefit ratio of implementing this feature category is favourable due to its relative simplicity of implementation.

Finally, for the QQ feature family, only the Specificity sub-category is frequently present in the model, with a total gain of 10.59, 18.89, and 19.89 in the Trace Recommendation scenario and 51.01, 51.51, and 51.97 in the Trace Maintenance scenario for Service, Data, and Store, respectively. It should be emphasised, however, that this sub-category accounts for 58% (72 out of 123) of the total number of features. In the Trace Recommendation scenario, we can observe that the maximum value of QQ (Specificity) for Store is 19.85. Further analysis reveals that this feature is the medium entropy of the JIRA descriptions as query, which was likewise the top performing for Data and the second best for Service in its category. The original intent of the QQ metrics may explain why Specificity has a greater information gain than the Similarity and Term Relatedness QQ metrics. In IR, queries are deemed high-quality when the relevant information is obtained, independent of the document in which it is stored. Both the Similarity and Term relatedness metrics assume that a document collection with many relevant documents is valuable because it raises the likelihood of retrieving a relevant document. However, for TLR, where there is only one genuine artifact to be identified, this is irrelevant. Because of this disparity, the Similarity and Term relatedness metrics are less suited for the TLR task. Specificity can still help since it seeks to differentiate the relevant document from the irrelevant documents, which is also visible in Table 6.

6 Threats to Validity

We present the threats to validity according to Wohlin's taxonomy [28].

Conclusion Validity refers to the ability to draw the correct conclusion about relations between treatment and outcome. In our case, our results have low statistical power since we analysed only three datasets. To cope with these threats, we carefully word our conclusions in such a way that the results are not oversold.

Internal Validity regards influences that may affect the independent variable with respect to causality, without the researchers' knowledge. The datasets are created by teams who follow the development method outlined in Sect. 3. While we compared the common attributes, we excluded those that were used only by certain datasets, e.g., JIRA comments. Furthermore, it is possible that certain trace links were incorrect and some links were missing. However, we picked the original datasets without performing any attempts to repair the datasets, which could have increased the bias.

Construct Validity concerns generalising the result of the experiment to the underlying concept or theory. The main threat concerns the research design: we approximate performance in the two scenarios via the $F_{0.5}$ and F_2 metrics. Although our method is aligned with the state-of-the-art in traceability research, in-vivo studies should be conducted for a more truthful assessment of the effectiveness, e.g., by deploying a system based on our employed algorithms and measuring the performance in use.

External Validity regards the extent to which we can generalise the results of our experiment to industrial practice. Our claims are limited to the low-code development domain, and, in particular, to the case of our industrial collaborator: Mendix. Although we collected projects from two separate teams, using more data would be beneficial. Finally, to minimise overfitting and enhance generalisability, we followed the standard practice of having a distinct training and test set.

Despite our best efforts to mitigate the threats, not everything can be accounted for. All the results were obtained from a single organisation, which could lead to a potential bias. Consequently, we had to be cautious in how we expressed our conclusions. Our results show promising avenues, but we are not yet in a position to generalise.

7 Conclusion and Future Work

In this study, we have provided initial evidence regarding requirements trace classification within an MDD-context. Upon analysing the MDD development process of our research collaborator (Mendix), we identified two scenarios which could benefit from a requirement trace link classifier: trace recommendation and trace maintenance. These scenarios require different performance metrics: F_2 for the former, $F_{0.5}$ for the latter.

After examining the three datasets under four rebalancing strategies, we obtained an average F_2-score (for trace recommendation) across the datasets of 69% with the LightGBM classifier with a mix of under- and oversampling (5050 strategy). For trace maintenance, we obtained an average $F_{0.5}$-score of 76% when employing XGBoost as the ML classifier and with no rebalancing of the training data.

The results are positive when considering that the percentage of traces in our datasets is low, ranging between 0.26% and 1.62% (see Table 1). This imbalance poses serious challenges when training a classifier and it represents a key obstacle to its performance.

We have also analysed which feature families from the literature, which we embedded in our tool, lead to the highest information gain. We found that process-related features seem to lead to the highest information gain, and that most query-quality features have a very marginal information gain and can therefore be discarded.

More research is needed about the specific features to include in production environments. Indeed, a high number of features may lead to overfitting. Also, we need to compare our ML-based approach to its deep learning counterparts. Studying additional dataset is one of our priorities, especially through the collaboration with Mendix. Moreover, analysing the performance of the tool in use is a priority: while we have based our analysis and discussion in F-measures, only a user study can reveal the actual quality of the recommended and recovered traces, that is, whether the developers who have to vet and use the traces find them useful, and whether they actually approve of integrating our approach into their development environment. Finally, studying horizontal traceability, i.e., the existence of links between artifacts at the same abstraction level (e.g., between requirements) is an interesting future direction.

This paper, which takes existing features for ML-based traceability and applies them to the low-code or model-driven domain, calls for additional studies on the effectiveness of the existing techniques in novel, emerging domains. We expect that such research will incrementally contribute to the maturity of the field of requirements traceability.

Acknowledgment. The authors would like to thank Mendix, and especially to Toine Hurkmans, for the provision of the datasets used in this paper and for giving us access to their development practices through numerous interviews and meetings.

References

1. Abadi, A., Nisenson, M., Simionovici, Y.: A traceability technique for specifications. In: Proceedings of ICPC, pp. 103–112 (2008)
2. Aung, T.W.W., Huo, H., Sui, Y.: A literature review of automatic traceability links recovery for software change impact analysis. In: Proceedings of ICPC, pp. 14–24 (2020)
3. Berry, D.M.: Empirical evaluation of tools for hairy requirements engineering tasks. Empir. Softw. Eng. **26**(6), 1–77 (2021). https://doi.org/10.1007/s10664-021-09986-0
4. Blaauboer, F., Sikkel, K., Aydin, M.N.: Deciding to adopt requirements traceability in practice. In: Krogstie, J., Opdahl, A., Sindre, G. (eds.) CAiSE 2007. LNCS, vol. 4495, pp. 294–308. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72988-4_21

5. Borg, M., Runeson, P., Ardö, A.: Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. Empir. Softw. Eng. **19**(6), 1565–1616 (2013). https://doi.org/10.1007/s10664-013-9255-y
6. Briand, L., Bianculli, D., Nejati, S., Pastore, F., Sabetzadeh, M.: The case for context-driven software engineering research: generalizability is overrated. IEEE Softw. **34**(5), 72–75 (2017)
7. Callens, A., Morichon, D., Abadie, S., Delpey, M., Liquet, B.: Using Random forest and Gradient boosting trees to improve wave forecast at a specific location. Appl. Ocean Res. **104**, 102339 (2020)
8. Capobianco, G., De Lucia, A., Oliveto, R., Panichella, A., Panichella, S.: On the role of the nouns in IR-based traceability recovery. In: Proceedings of the ICPC, pp. 148–157, May 2009
9. Cleland-Huang, J., Berenbach, B., Clark, S., Settimi, R., Romanova, E.: Best practices for automated traceability. Computer **40**(6), 27–35 (2007)
10. Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. J. Am. Soc. Inf. Sci. **41**(6), 391–407 (1990)
11. Falessi, D., Di Penta, M., Canfora, G., Cantone, G.: Estimating the number of remaining links in traceability recovery. Empir. Softw. Eng. **22**(3), 996–1027 (2016). https://doi.org/10.1007/s10664-016-9460-6
12. Falessi, D., Roll, J., Guo, J.L.C., Cleland-Huang, J.: Leveraging historical associations between requirements and source code to identify impacted classes. IEEE Trans. Software Eng. **46**(4), 420–441 (2018)
13. Ghannem, A., Hamdi, M.S., Kessentini, M., Ammar, H.H.: Search-based requirements traceability recovery: a multi-objective approach. In: Proceedings of the CEC, pp. 1183–1190 (2017)
14. Gotel, O., et al.: The grand challenge of traceability (v1.0). In: Cleland-Huang, J., Gotel, O., Zisman, A. (eds.) Software and Systems Traceability, pp. 343–409. Springer, London (2012). https://doi.org/10.1007/978-1-4471-2239-5_16
15. Guo, J., Cheng, J., Cleland-Huang, J.: Semantically enhanced software traceability using deep learning techniques. In: Proceedings of ICSE, pp. 3–14. IEEE (2017)
16. Mills, C., Escobar-Avila, J., Haiduc, S.: Automatic traceability maintenance via machine learning classification. In: Proceedings of ICSME, pp. 369–380, July 2018
17. Mills, C., Haiduc, S.: The impact of retrieval direction on IR-based traceability link recovery. In: Proceedings of ICSE NIER, pp. 51–54 (2017)
18. Oliveto, R., Gethers, M., Poshyvanyk, D., De Lucia, A.: On the equivalence of information retrieval methods for automated traceability link recovery. In: Proceedings of ICPC, pp. 68–71 (2010)
19. Port, D., Nikora, A., Hayes, J.H., Huang, L.: Text mining support for software requirements: traceability assurance. In: Proceedings of HICSS, pp. 1–11. E (2011)
20. Porter, M.F.: An algorithm for suffix stripping. Program (1980)
21. Ramesh, B., Edwards, M.: Issues in the development of a requirements traceability model. In: Proceedings of ISRE, pp. 256–259 (1993)
22. Rasiman, R., Dalpiaz, F., España, S.: Online appendix: how effective is automated trace link recovery in model-driven development? January 2022. https://doi.org/10.23644/uu.19087685.v1
23. Rath, M., Rendall, J., Guo, J.L.C., Cleland-Huang, J., Maeder, P.: Traceability in the wild: automatically augmenting incomplete trace links. In: Proceedings of ICSE, vol. 834–845 (2018)

24. Umuhoza, E., Brambilla, M.: Model driven development approaches for mobile applications: a survey. In: Younas, M., Awan, I., Kryvinska, N., Strauss, C., Thanh, D. (eds.) MobiWIS 2016. LNCS, vol. 9847, pp. 93–107. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44215-0_8
25. Wang, B., Peng, R., Li, Y., Lai, H., Wang, Z.: Requirements traceability technologies and technology transfer decision support: a systematic review. J. Syst. Softw. **146**, 59–79 (2018)
26. Wieringa, R.J.: Design Science Methodology for Information Systems and Software Engineering. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43839-8
27. Winkler, S., von Pilgrim, J.: A survey of traceability in requirements engineering and model-driven development. Softw. Syst. Model. **9**(4), 529–565 (2010)
28. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering, vol. 9783642290 (2012)
29. Yoon, J.: Forecasting of real GDP growth using machine learning models: gradient boosting and random forest approach. Comput. Econ. **57**(1), 247–265 (2020). https://doi.org/10.1007/s10614-020-10054-w