University of Groningen

Department of Artificial Intelligence

Detecting & Classifying Architectural Design Decisions in Issues from Issue Trackers using Issue Properties

Authors:	***Student Numbers:***
Arjan Dekker	s3726169
Matthias Drijfhout	s3786196
Jesse Maarleveld	s3816567
Thomas ten Napel	s3349985

June 6, 2023

Abstract

Nowadays, a software architecture consists of both the design *and* the design decisions. These design decisions are often not documented. We can recover these decisions by extracting them from issue tracking systems. Issue tracking systems are often used by developers for the development of open source software projects as a mean of communication.

Before design decisions can be extracted, we first have to find out which issues contain design decisions. Dekker and Maarleveld tried to detect and classify design decisions by using issue properties [6]. Issue properties are all the data except for the issue summary, description and the comments. This includes for example issuetype (e.g. 'bug' or 'feature'). However, they did not do much preprocessing on the data. Therefore, we developed and applied a more sophisticated preprocessing in an attempt to improve the performance. We were not able to improve the detection performance, but we improved the classification performance by 9.2% to 0.432 F_1-score. Furthermore, we used white-box machine learning methods to gain insight in which issue properties are important for detecting and classifying design decisions. These insights might help with developing search tools for finding issues containing design decisions.

Contents

1 Introduction

The architecture of a software system is a high-level decomposition of a system, which describes the major components and how these components interact with one-another. The design of the software architecture dictates the global high-level design for the entire system. The effects of the decisions made while designing the architecture may influence the further development of the system at every subsequent step [17].

We usually refer to the high-level description of the system as the architectural design, or *design* for short. However, the design does not capture the entirety of the *Architectural Knowledge* (AK). Architectural knowledge is the entirety of knowledge required to come up with the design for a system, and understand why it is designed that way. The design is one part of the architectural knowledge [9]. However, another important part of the architectural knowledge, is the design decisions. Design decisions are decisions made by software architects while coming up with the design of the system or components of the system (or making changes to an existing system). They explain why the system is the way it is [9, 17]. Hence, we often say that the entirety of the architectural knowledge is given by the combination of the design *and* the design decisions [9].

However, a large problem is that design decisions are often not documented by software architectures. This can be problematic in multiple ways. If design decisions are not explicitly documented, future maintainers of the system do not have access to all the architectural knowledge about the system. This may hurt the maintainability of the system or the ability to further evolve the system [9]. Additionally, previous design decisions made by software architects can be reused for other projects [11]. However, this would require explicit documentation of design decisions.

To work around this, research has shifted towards the recovery of design decisions from other artefacts. Developers often use issue-trackers (e.g. JIRA) in order to streamline discussions in the from of so-called issues. Some of the issues in these issue-trackers discuss architectural design decisions. Previous research has focused on identifying discussions potentially containing architectural design decisions using 1) source code analysis ([15, 13]), 2) dependency-file analysis ([15]), 3) keyword searches ([15]), and 4) machine learning and deep learning techniques applied to the text (summary (title) and description) of issues ([15, 2, 6]).

However, issues often also have so-called properties, such as the status of an issue (e.g. "Open" or "Closed"), or the type of issue (e.g. "Bug"). Issue properties generally do not get the same thorough treatment the text content of issues does. Bhat et al. ([2]) did not consider issue properties at all. Soliman et al. only considered issue properties briefly without any intricate preprocessing [6]. Our first goal in this report is to obtain better classification performance than Soliman et al. did using issue properties. We want to achieve this by looking more thoroughly at issue properties; we will perform actual preprocessing on the data by 1) removing data which does not make sense to be used as a feature, and 2) use domain knowledge to create encodings for certain issue properties. The second goal is to gain insight into how the classifiers make decisions. We can achieve that by using white-box ML methods (naive Bayes, decision tree and random forest) and by using a method called feature shuffling, which should point out which issue properties are the most important for classification.

In section 2, we will cover more about the background of issue properties and architectural design decisions. We will describe the dataset we used in section 3. Next, we will explain our entire methodology in section 4. We will then explain how we used the dataset to come up with feature vectors in section 5. In section 6, we explain why we did not use any dimensionality reduction techniques. In section 7, we describe the classifiers we used. In section 8, we explain feature shuffeling in more detail. The results are presented in section 9 and discussed in section 10.

2 Background

In this section, we will cover some more details about issue properties and architectural design decisions.

2.1 Issues-Trackers & Issues

Issue-trackers, such as JIRA, are websites where developers discuss issues related to software projects. The most basic elements of an issue are the summary (title), description, and the comments. These combined form the discussion between the developers. However, issues also have other properties which we can use as features or which we can use to compute features. An example of an issue is given in figure 1. In JIRA, the following issue properties are present:

- *Status*: the current status of the issue. The most basic options are "Open" for an active issue, and "Closed" for an issue which is either done or determined to be otherwise irrelevant.
- *Resolution*: the final resolution on an issue. This can be seen as the reason why the issue status was set to "Closed". Most often this is "Fixed", but can also be something like "Not a problem", indicating that a problem described in an issue is not actually considered a problem (but intended behaviour).
- *Votes*: the number of votes on an issue. People can vote on issues to mark them as important.
- *Watches*: the number of people watching an issue. People can "watch" issues to automatically get notified when there is activity on the issue.
- *Parent*: a link to the parent issue. Some issues may be part of a larger overarching issue or programming effort. This is often indicated by including a link to the parent issue. Not all issues will have a parent. Many issues are in general standalone.
- *Priority*: the importance of the issue. Some issues are minor problems (e.g. documentation), while others are critical (e.g. security vulnerabilities).
- *Issue-links*: links to other issues. Linked issues are related, but are not child issues (see sub-tasks). An issue for a bug may for instance contain a link to the issue which introduced the change which caused the bug.
- *Attachments*: attachments added to the issue. An example attachment would be a diagram depicting the design of a part of the system.
- *Components*: a list of software components of the system which are affected by the issue.
- *Labels*: labels added to the issue. These labels serve to give additional information or classify the issue into a certain category. An example label could be "beginner issue", indicating that an issue is suitable for a developer who is new to the project.

  To avoid confusion with the target labels which we want to predict, we will call these labels attached to issues by developers the *issue labels*. The target labels will be referred to as the *target labels*.

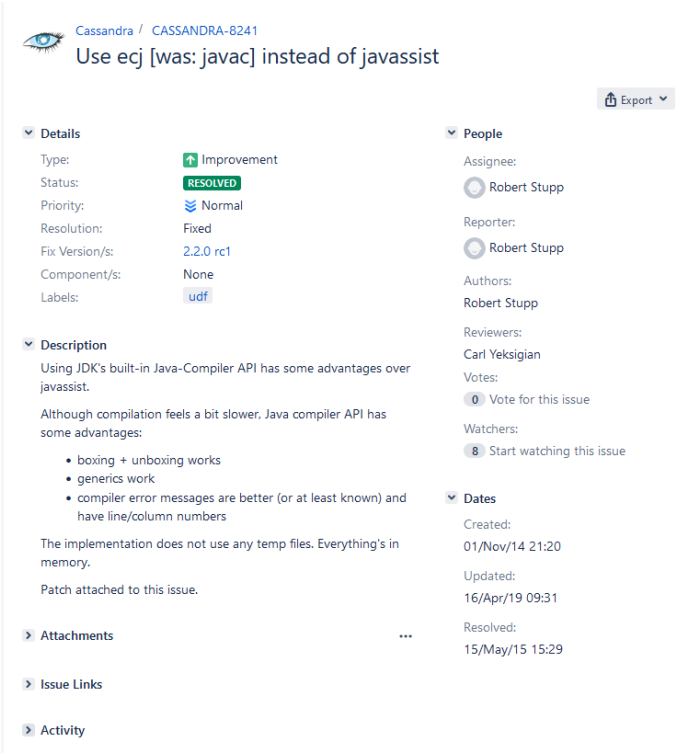

Fig. 1. Example of an issue containing an architectural design decision (executive design decision). Source: `https://issues.apache.org/jira/browse/CASSANDRA-8241`

- *Issue Type*: the type of the issue. An example issue type is "Bug". Other important types are "Task" and "Sub-Task". A task is an issue requiring work outside of the software system itself; e.g. work on some external documentation or monitoring tool. A sub-task is generally an issue which is part of some larger coding effort, and will often have a parent issue.
- *Sub-tasks*: A list of all the child-issues of an issue. Hence, all sub-tasks of an issue will have that issue as their parent.
- *Dates*: each issues has a creation date. Additionally, an issue may have a last-updated data, and resolved issue will have a resolution date.
- *Affected Versions*: versions of the software affected by the issue.
- *Fix Versions*: version of the software in which the particular issue should be fixed or resolved.
- *Environment*: environment (e.g. operating system) in which the issue was encountered. This field is rarely used and not available for most issues.
- *Assignee*: person assigned to work on the issue.
- *Reporter*: person who reported the issue.

2.2 Types of Architectural Design Decisions

Architectural design decisions can be further categorised into more fine-grained types. It is useful to distinguish between these types of design decisions because they may be relevant in very different situations. We will focus on the three main types of architectural design decisions defined by Kruchten in [10]:

- **Existence** design decisions are decisions resulting in the creation or alteration of components or interactions between components in the system. For instance, an existence decision could be the choice of a micro-service oriented architecture versus a monolithic architecture. Existence decisions are sometimes further subdivided into structural and behavioral design decisions. Here, the former relate to decisions related to components, while the latter are related to the interactions between components. Additionally, so-called ban-decisions also fall into the category of existence decisions. Ban-decision express negative existence: they specify that some component or behaviour will not show up in the system.
- **Property** design decisions specify enduring and overarching traits of the system. These traits often relate to quality attributes such as performance or security. An example of a property design decisions is to only make use of open source software which does not prohibit closed-source distribution of projects using it.
- **Executive** design decisions are decisions driven by the external environment of the software. This may include decisions made for financial reasons, or other business-driven decisions. Executive decisions also include decisions about the use of external software or dependencies [15].

2.3 Measuring ML performance

One of the most basic and most direct ways to show the performance of a classifier, is using a confusion matrix. A confusion matrix can be used to show the quality of the predictions on a set of testing data. Specifically, the rows of the confusion matrix correspond to the true labels, and the columns correspond to the predicted label. Each entry C_{ij} in the confusion matrix gives the amount of samples in the testing set from class i which were predicted as class j. The values on the diagonal should be as large as possible, because those correspond to correct classifications. The off-diagonal elements correspond to incorrect predictions. We can use the confusion matrix to observe how good the performance of the classifier is, and what mistakes are made [3].

From the confusion matrix, we can compute a number of other metrics which can be used to express the performance of a classifier. The four most basic ones are

- true positives (tp): number of samples from class i correctly classified as being from class i.
- false positives (fp): number of samples from a class $j \neq i$ classified as being from class i.
- true negatives (tn): number of samples not from class i classified as not being from class i.
- false negatives (fn): number of samples from class i not classifies as being from class i.

Note that we can compute true positives/false positives/true negatives/false positives for every class i separately. That means that if we have n classes $\{C_1, C_2, \ldots, C_n\}$, we can compute each metric n times for all the different classes. Only in binary classification problems ($n = 2$) do we define one class as the positive one and the other as the negative one, meaning that compute every metric only once [3].

A common metric used for measuring the performance of a classifier is the F_1-score. To understand the F1-score, we first explain the metrics recall and precision.

Recall is a metric for determining how many samples of a certain class were predicted to be of that class. This essentially penalises a classifier that 'misses' samples. It is therefore defined as follows:

$$\text{recall} = \frac{\text{tp}}{\text{tp} + \text{fn}} \qquad [3] \tag{1}$$

Just like with the number of true positives etc. explained above, we compute recall per class, except when we have a binary classification problem.

If the classifier 'misses' samples, the number of false negatives increases. This would in turn reduce the recall value. However, if a classifier were to predict all data points to be of a certain class, the recall for that class would be maximal.

Therefore, we introduce the precision metric. The precision metric measures how many samples that were predicted to be from class i were really from class i. Precision can therefore be calculated as follows:

$$\text{precision} = \frac{\text{tp}}{\text{tp} + \text{fp}} \qquad [3] \tag{2}$$

The same considerations for computation per class as outlined above apply for precision.

Both recall and precision determine how useful a classifier is. One essentially does not want to miss samples of a certain class, but also wants the predictions to be accurate. Therefore, we introduce F_1-score. This is the harmonic average between the recall and precision and is defined as follows:

$$F_1 = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \qquad [3] \tag{3}$$

The problem is that all metrics introduced to this point are class specific; they have to be computed for every class in the dataset. However, we would like to express the performance of a classifier as a single number. In this report, we will do this through the use of the macro F_1 score, defined as

$$F_{1,\text{macro}} = \frac{1}{n}\sum_{i=1}^{n} F_{1,i}, \qquad [3] \tag{4}$$

which is the unweighted average of the F_1 scores for all the classes. The macro F_1 score expresses the believe that the performance on every class should be good, which makes it an appropriate metric for use with imbalanced classes [3].

2.4 Related Work

As alluded to in the introduction, machine learning has been employed by others previously in an attempt to automatically extract architectural design decisions from issue tracking system [2, 15, 6].

Bhat et al. used various traditional machine learning techniques (i.e. support vector machine, decision tree, logistic regression, naive Bayes, and one-vs-rest) in order to detect and classify architectural design decisions from issue trackers. They used a dataset of 1469 randomly sampled issues from the projects Apache Hadoop and Apache Spark, and used the summary and description of these issues as a basis for creating feature vectors. The researchers proposed a two-step process: first, a classifier is used for detection (i.e. determining whether an issue contains a design decision). Next, if the issue is architectural, it is classified. Bhat et al. classified issues into the three sub-types of Existence decisions: structural decisions, behavioral decisions, and ban decisions. In the end, they were able to detect architectural design decisions with an $F1$-score of 0.91, and classify them with an F1-score of 0.83. Both these results were obtained using a linear support vector machine [2].

However, Dekker and Maarleveld were unable to reproduce the results obtained by Bhat et al. [6]. Additionally, they believe that the dataset of Bhat et al. is incorrectly labeled, because the label "Ban Decision" is used in a way that is inconsistent with the ontology defined by Kruchten [6]. Furthermore, Soliman et al. argue that the two-step approach proposed by Bhat et al. should be replaced with a single-step approach, where issues are classified into the three main categories defined by Kruchten (existence, executive, property), *or* non-architectural, if possible. This is because a two-step approach leaves two opportunities to introduce errors. Finally, Soliman et al. also argue that classifying into the three major design decision types defined by Kruchten is better than classifying into the sub-types of existence. This is because restricting the classification to existence decisions may cause search tools built upon these classifiers to miss important architectural information. Additionally, classification into the different types of existence decisions is often not very valuable, because most existence decisions are always both structural and behavioral [15].

Soliman et al. experimented with a number of neural network architectures in order to detect and classify architectural design decisions in issues. For classification, they used their definition described in the previous paragraph. They still experimented with detection, because their results for classification were mediocre. Specifically, they were able to achieve an F_1-score of 0.83 for detection, and 0.57 for classification. The features they used were mostly based on the summary and descriptions of issues. However, issue properties were also briefly considered, but without the application of preprocessing or domain knowledge for handcrafted features. Using these issue properties, an F_1 score of 0.81 was achieved for detection, and 0.34 for classification.

3 Dataset Description

For this assignment, we will be using the dataset created in [15][1]. The dataset contains 2179 issues classified into the categories Existence, Executive, and Property. Some issues may have multiple target labels, and some issues may fall in none of the categories (i.e. they are said to be non-architectural).

For our purposes, we will be simplifying the target labels. Specifically, for issues falling into multiple categories, we will only be using the "most important" category. Here, we define importance (from most to least) as Executive - Property - Existence. This is because executive decisions drive (result in) property decisions, and property decisions drive existence decisions [15]. Taking this approach allows us to compare our results with the current state-of-the-art techniques ([6, 15]).

The dataset contains issues from seven different Apache projects: Hadoop, HDFS, MapReduce, Tajo, Yarn, HBASE, and Cassandra. Table 1 gives a full breakdown of the dataset in terms of projects and target labels.

The dataset itself is a list of issue keys (unique identifiers) and the labels for those issues. The Github repository[2] accompanying [6] provides code for downloading the issue properties for the issues in the dataset.

4 Methodology

In this section we give an overview of the approach we used for this research. In section 3, we already gave a description of the dataset we will be using.

As stated in the introduction, the main goal we want to achieve in this assignment, is to investigate whether we can improve the

[1] Dataset downloadable from: `https://github.com/mining-design-decisions/mining-design-decisions/blob/main/datasets/labels/EBSE_labels.json`

[2] `https://github.com/mining-design-decisions/mining-design-decisions`

Project	Architectural			Non-Architectural	Total Per Project
	Existence	Executive	Property		
Yarn	202	7	25	91	325
Tajo	69	38	13	98	218
MapReduce	44	10	17	53	124
HDFS	164	27	46	95	332
Hadoop	171	64	51	134	420
HBASE	0	0	0	1	1
Cassandra	249	149	85	276	759
Total	899	295	237	748	2179
		1431			

Table 1. The amount of issues per label and per project in the dataset created by Soliman et al. in [15].

performance of of issue-property based detection and classification of architectural design decisions.

The first step in doing this, is the creation of manually crafted features from the raw data we can obtain from issues. This process is described in section 5.

Because high dimensional can make machine learning tasks more difficult ([7]), we performed PCA analysis to determine whether these methods are useful for reducing the dimensionality of the data. Section 6 contains the details of this analysis. We found that PCA does not provide much benefit for reducing the dimensionality and hence we are not using it further.

Since we are not using PCA, we use the preprocessed data (as described in section 5) as the input to the ML classifiers. The second goal of our research was to gain insight into what features were important for the classifiers. Because of this, we decided to use classifiers with some explanatory capabilities.

We wanted to use the classifiers for two different learning tasks: detection and classification. Here, detection is a binary classification problem, where the classifier has to identify all issues containing an architectural design decisions. The target labels for the dataset are obtained by taking the target labels for the dataset as described in section 3, and assigning target label 0 to all non-architectural issues, and 1 to all issues containing *some* type of architectural design decisions.

The other learning task is classification. For classification, the classifiers have to learn and predict the exact labels as outlined section 3.

This means that besides being white-box classifiers, all classifiers must also be usable for multi-class classification problems. For us, this rules out logistic regression, since logistic regression can only be used for binary classification problems [12]. In the end, we chose Naive Bayes, decision trees, and random forest. These are described in more detail in section 7.

In order to determine which parameters are the best for each classifier, we use GridSearchCV from sklearn[3]. GridSearchCV allows us to specify a set of possible values for each hyper-parameter, and then performs an exhaustive search of all possible combinations. For each set of specified parameters, the performance is estimated using cross-validation. We perform this grid search with the default 5-fold cross-validation.

This 5-fold cross-validation splits the data 5 folds of similar size. Then, the classifier is trained five times. Each time, one of the folds is used for validation, and the other 4 four testing. This is done in such a manner that each fold is used for validation exactly once. We used macro F_1 score in order to quantify the performance of the classifiers on the validation sets. Finally, the performance of the classifier is determined by averaging the scores on the validation sets.

Note that we do not want to use all the data for the grid search. That is because with the grid search, we are doing hyperparameter optimization of the models. This optimization might overfit on that data [3]. Hence, we only use 80% of the available data for the grid search and the other 20% is kept apart as a so-called test set. The split of the data is a stratified split, meaning that we force the distribution of the labels is the same for each of the splits. Note that when we test a classifier on the test set, we train the classifier using the entire 80% of the data. This is to make sure the classifier has most data available for training. A complete overview of the data splitting is given in figure 2.

Another remark we have to make is that we use class limiting, as this was found to work the best in [6]. Class limiting means that we randomly remove samples from the majority classes until we have roughly the same amount of samples of each class (i.e. we perform undersampling). We only applied class limiting for the classification task – just as in [6]. The amounts of items from every class was limited to 205 – the amount of items in the smallest class, after removing unresolved issues. This choice of limit is not ideal because it does not take into account the true class distributions. However, the true class distributions in "real life" are not known. The dataset we use, consists of issues collected specifically with techniques designed to find issues containing architectural design decisions [15]. Hence, we cannot expect the class distributions in the dataset to be representative of the real world. Since we do not have any information about the true real word distribution, we assumed uniform priors. This choice may be sub-optimal, but it is the option generally used by state of the art research [15, 6].

Finally, we want to determine which features (i.e. issue properties) are the most important for obtaining good performance. Not only did we use classifiers with explanatory power, but we also used feature shuffling for this. More information about feature shuffling can be found in section 8. We only performed feature shuffling with the best performing classifier and settings that were obtained in the previous step.

Note that we perform the ML classifier and feature shuffling steps for both the detection task and classification task separately. This is because the best ML settings for detection might not be the best settings for classification.

As our research is a continuation of a larger research that also provides open source code, we uploaded our code to a GitHub repository[4] as well.

5 Manually Crafted Features

In this section, we will be discussing the feature vectors we will be constructing. Some properties extracted from issues are usable (almost) directly as features, while others require some more

[3] `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html`

[4] https://github.com/mining-design-decisions/exploring-issue-properties

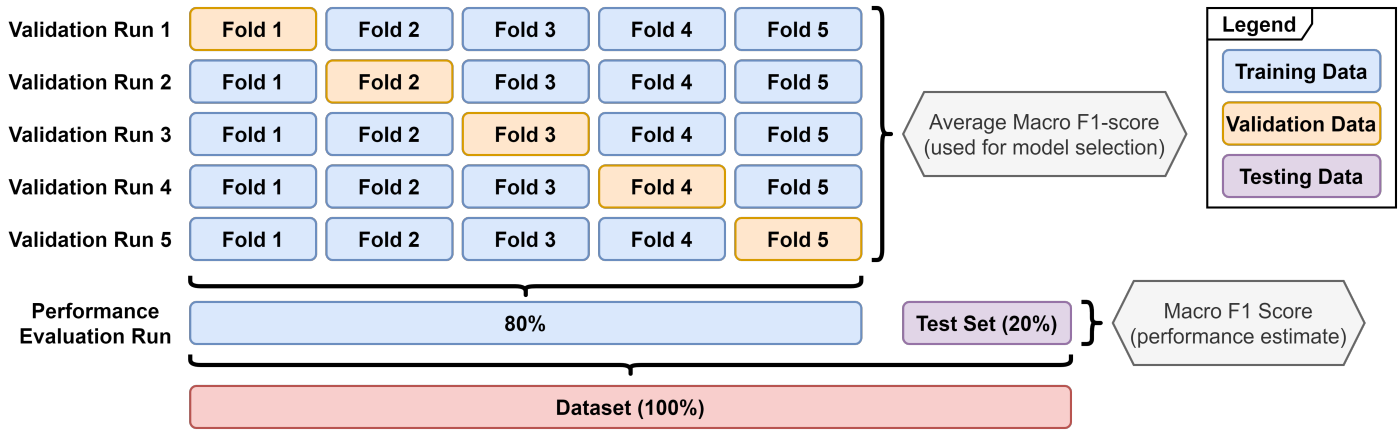

Fig. 2. Graphical depiction of how we split up the dataset. 80% of the data is used to select hyperparameters using 5-fold cross validation. 20% Is set aside as a test set. After model selection, the best model is trained on the 80% initially used for cross validation. The final performance estimate is obtained by testing on the test set

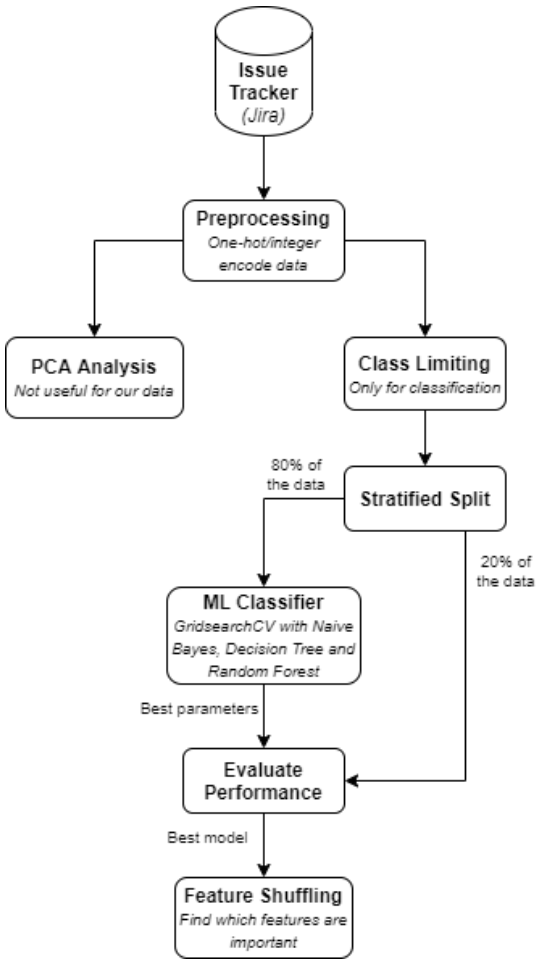

Fig. 3. Overview of the methodology.

elaborate preprocessing. In this section, we will describe the preprocessing for every issue property.

5.1 Unused properties

We will not be using the reporter and assignee properties. Even if they were to contain some valuable information for the classifiers, they are highly project-specific and unlikely to generalise well. Because of this, we decided not to include them as features.

We also did not do anything with the environment property, because it is absent for most issues (only 93 out of the 2179 issues in the dataset have this property set to a non-null value).

5.2 Issue Priority

Issues are often assigned a priority. These are represented using words such as "trivial" or "urgent". In all the projects from which the issue dataset was constructed, two different priority schemes are used. Yarn, Tajo, MapReduce, HDFS, Hadoop, and HBASE all use the same scheme: trivial - minor - major - critical - blocker. Cassandra uses the scheme "Low - Normal - High - Urgent". As a first step, we want to find a way to unify these two schemes. The amount of issues with a given priority label per project is also displayed in Figure 4.

Fig. 4. Amount of issues with a given priority label per project.

What we observe in Figure 4, is that very few issues in the dataset have the "trivial" priority. Hence, we chose to consider these to be "the same" as the minor priority. This means that we reduced the 5-class scheme used by the Hadoop-related project to a 4-class scheme. We now one-to-one map this to the Cassandra scheme.

Because these priorities are a form of ordinal data (categorical data with ordering), we simply encode the classes as integer. Formally, we use the function f_p defined as

$$f_p(x) = \begin{cases} 0 & \text{if } x \in \{\text{trivial}, \text{minor}, \text{low}\} \\ 1 & \text{if } x \in \{\text{major}, \text{normal}\} \\ 2 & \text{if } x \in \{\text{critical}, \text{high}\} \\ 3 & \text{if } x \in \{\text{blocker}, \text{urgent}\} \end{cases} \quad (5)$$

in order to encode the priorities as integers.

5.3 Issue Type

The next property we are considering, is the issue type. Table 2 gives as overview of all issue types in the dataset, alongside the frequency of each type in every project. What we observe is that five issue types occur rather frequently ("New Feature", "Improvement", "Task", "Sub-task", "Bug"), and two labels are fairly rare ("Test" and "Wish").

	Issue Type						
Project	**New Feature**	**Improvement**	**Task**	**Sub-task**	**Bug**	**Test**	**Wish**
Yarn	44	38	9	196	38	0	0
Tajo	27	81	22	48	39	1	0
MapReduce	23	23	4	26	47	2	0
HDFS	78	93	3	114	43	1	0
Hadoop	129	143	3	70	71	2	2
HBASE	0	0	0	0	1	0	0
Cassandra	98	357	39	62	203	0	0
Total	399	735	80	516	442	6	2

Table 2. Breakdown of the amounts of issues with a given type per project in the dataset.

There does not seem to be any major overlap between the categories. Additionally, even though the "Wish" and "Test" types are rarely used, we keep them as-is. This is because there is no clear category these two can be merged with. We use a one-hot encoding in order to represent them. This means that the issue type is encoded according to the function f_t defined as

$$f_t(x) = \begin{cases} e_1^7 & \text{if } x = \text{New Feature} \\ e_2^7 & \text{if } x = \text{Improvement} \\ e_3^7 & \text{if } x = \text{Task} \\ e_4^7 & \text{if } x = \text{Sub-task} \\ e_5^7 & \text{if } x = \text{Bug} \\ e_6^7 & \text{if } x = \text{Test} \\ e_7^7 & \text{if } x = \text{Wish} \end{cases} \quad (6)$$

Here (and from now on) e_i^n denotes a real n-dimensional unit vector with all elements equal to 0 except for the i-th element, which is equal to 1 instead.

5.4 Status

We can also consider the issue status as a feature. The issues statuses in the dataset are listed in Table 3. Note that the issue statuses are dynamic and may change during the lifetime of an issue. Hence, we are somewhat unsure whether issue status can be an effective indicator in determining whether an issue is architectural or not, or in classifying an architectural issues.

However, we still think there is value in including this feature. For instance, there could be a differences between the two statuses "Resolved" and "Closed". There could be a subtle difference in the exact status used by developers. For instance, some issues are resolved, but never closed (e.g. [5]).

In general, it can be expected that the other remaining issue statuses eventually are changed to either "Resolved" or "Closed". Because of this, we consider the other statuses to be less meaningful. In particular, we do not think it is relevant for this application to consider the difference between the different types of statuses of issues which are still open or in progress. Because of this, we merged these into one single category.

	Issue Status							
Project	**Resolved**	**Closed**	**Open**	**Patch Available**	**In Progress**	**Reopened**	**Triage Needed**	**Awaiting Feedback**
Yarn	192	93	35	4	1	0	0	0
Tajo	211	1	3	1	1	1	0	0
MapReduce	31	88	2	3	0	0	0	0
HDFS	161	114	36	19	1	1	0	0
Hadoop	115	269	24	9	0	3	0	0
HBASE	0	1	0	0	0	0	0	0
Cassandra	720	0	32	1	1	0	2	3
Total	1430	566	132	37	4	5	2	3

Table 3. Breakdown of the amounts of issues with a given status per project in the dataset.

In the end, we use the function f_s defined as

$$f_s(x) = \begin{cases} e_1^3 & \text{if } x = \text{Resolved} \\ e_2^3 & \text{if } x = \text{Closed} \\ e_3^3 & \text{otherwise} \end{cases}$$

in order to map the issue status to a numerical representation.

5.5 Resolution

In Table 4, we list all the possible resolutions for issues in the dataset. There is quite a significant number of different resolutions, although most issues simply have the resolution "Fixed". A few other resolutions also express that an issue has been completed or implemented: "Done", "Implemented", and "Resolved". We count these resolutions as a single category. Next, we have resolutions for invalid issues. These are issues which will not be acted upon because the issue is unclear, or the issue turns out to be about something which was actually intended. The resolutions belonging to this category are "Cannot reproduce", "Duplicate", "Incomplete", "Invalid", and "Not A Problem". Next, we have resolutions for issues which are essentially valid, but which will not be acted upon. These are "Abandoned", "Won't Do", and "Won't Fix". We consider "Later" to be its own category, because the issue is still relevant, but simply not being worked on at the moment; possibly because other issues have to be resolved first. None of the other categories express this, so we kept "Later" as a separate category.

There is also a number of issues without resolution (N/A). These are issues which had not been resolved at the time of collecting the data for the features. Hence, we can consider the resolution information to be missing for these issues. We will discuss this further in section 5.18.

[5] `https://issues.apache.org/jira/browse/HADOOP-15091`

	Resolution													
Project	**N/A**	**Abandoned**	**Cannot Reproduce**	**Done**	**Duplicate**	**Fixed**	**Implemented**	**Incomplete**	**Invalid**	**Later**	**Not A Problem**	**Resolved**	**Won't Do**	**Won't Fix**
Yarn	40	1	1	0	7	266	0	0	1	1	0	0	1	7
Tajo	6	0	1	0	0	211	0	0	0	0	0	0	0	0
MapReduce	5	0	0	0	6	106	0	1	2	1	0	0	1	2
HDFS	57	0	1	2	6	253	0	0	1	3	0	3	0	6
Hadoop	36	1	0	1	5	362	1	1	2	0	1	0	0	10
HBASE	0	0	0	0	0	1	0	0	0	0	0	0	0	0
Cassandra	39	0	2	0	16	666	0	0	2	7	5	0	0	22
Total	183	2	5	3	40	1865	1	2	8	12	6	3	2	47

Table 4. Breakdown of the amounts of issues with a given resolution per project in the dataset. Issues with resolution "N/A" had not been resolved at the time of data retrieval.

In the end, we use the function f_r defined as

$$f_r(x) = \begin{cases} e_1^4 & \text{if } x \in \{\text{Done, Implemented, Resolved, Fixed}\} \\ e_2^4 & \text{if } x \in \{\text{Cannot Reproduce, Duplicate, Incomplete, Invalid, Not A Problem}\} \\ e_3^4 & \text{if } x \in \{\text{Abandoned, Won't Do, Won't Fix}\} \\ e_4^4 & \text{if } x = \text{Later} \end{cases}$$

in order to map the resolution to a one-hot encoded vector.

5.6 Components

Issues often have a list of components they affect or apply to. The list of components for the issues in the dataset is given in Table 5. In this table, we can see that the components are considerably different between the different project. This is not surprising, because different projects logically have different components.

Because all the components are different per project, we do not think it is beneficial to include the components themselves (in some encoded form) in the input feature vectors directly. In stead, we include a field for the number of components referenced in an issue.

5.7 Issue Labels

Many different types of issue labels can be assigned to issues. In Table 6, we present a list of the different issue labels used for issues across the different projects. We see that there is a large variety in the issue labels used – and also in the *types* of issue labels used: some issue labels reference quality attributes, some reference components, some reference the issue status, and there are many more different types of issue labels.

In order to deal with this large variety in issue labels, we use an approach proposed by Soliman, Galster, and Riebisch in [14], and further extended by Dekker and Maarleveld in [6]. In these two works, words were grouped into so-called ontology classes – classes of words containing related concepts. Words in text were then replaced with their respective ontology classes. We did the same here: we take the ontology classes from [6][6], and replaced issue labels with their corresponding ontology classes. We also extended the ontology classes. The additions to the ontology classes are presented in appendix A. The assignment from issue labels to ontology classes is presented in appendix B.

One thing to note is that, since an issue may have multiple issue labels, an issue may have multiple issue labels with the same ontology class. However, we observed that this is only the case for 23 issues in the entire dataset. As such, we think that taking the amount of issue labels from any given ontology class into account will probably not add any useful information in the best case, and may lead to over-fitting in the worst case.

Instead, we encode the issue labels based on the idea of one-hot encoding. Suppose that we have an issues with a set of issue labels I. Then, for every issue label $i \in I$, we first look up its corresponding ontology class according to table 14. We will denote this using $\text{I2O}(i)$. Next, we encode the result using a one-hot encoded vector. Finally, we take the binary OR operation over all one-hot encoded vectors for a given issue. The result is a vector of length 18 with a 1 in every entry corresponding to an ontology class from which the issue has an issue label. For clarity, we will also denote this mathematically. We use $\text{I2O}(i)$ as defined above, and then define the function f_ℓ as

[6]Available from `https://github.com/mining-design-decisions/mining-design-decisions`

Project	Components
Hadoop	auth (1), bin (1), build (35), common (1), conf (11), documentation (8), filecache (1), fs (39), fs/adl (2), fs/azure (8), fs/cos (1), fs/oss (1), fs/s3 (36), fs/swift (1), ha (7), io (21), ipc (20), kms (5), metrics (10), native (4), net (2), performance (4), record (4), scripts (3), security (44), site (1), test (11), tools (4), tools/distcp (2), tracing (1), util (12), viewfs (1)
HBASE	
HDFS	auto-failover (3), balancer & mover (5), block placement (1), build (10), caching (2), contrib/hdfsproxy (1), datanode (52), distcp (1), encryption (2), erasure-coding (7), federation (2), fs (6), fuse-dfs (1), ha (20), hdfs (11), hdfs-client (36), ipc (1), kms (1), libhdfs (3), namenode (105), nfs (5), nn (1), ozone (16), performance (5), qjm (2), rolling upgrades (1), scm (1), security (9), test (5), tools (4), webhdfs (8)
MapReduce	applicationmaster (6), benchmarks (1), build (5), client (4), contrib/fair-share (2), contrib/raid (1), contrib/streaming (1), distcp (1), documentation (1), examples (1), jobhistoryserver (1), jobtracker (6), mr-am (8), mrv1 (1), mrv2 (41), nodemanager (6), performance (1), resourcemanager (2), scheduler (1), security (8), task (6), task-controller (2), tasktracker (5), test (3)
Tajo	Build (20), Catalog (22), Data Shuffle (3), Data Type (3), Documentation (5), Expression (1), Function/UDF (7), Index (1), JDBC Driver (2), Java Client (12), Metrics (1), Offheap (3), Orc (1), Physical Operator (7), Planner/Optimizer (18), Pull Server (1), QueryMaster (10), RPC (6), Resource Manager (6), S3 (3), SQL Parser (6), SQL Shell (5), Scheduler (1), Site (1), Sort algorithm (1), Storage (37), TajoMaster (8), Tools (1), Unit Test (7), Web UI (1), Worker (7), conf and scripts (1), distributed query plan (3)
Yarn	ATSv2 (1), RM (2), api (13), applications (4), applications/unmanaged-AM-launcher (1), capacity scheduler (9), capacityscheduler (14), client (6), distributed-scheduling (2), fairscheduler (3), graceful (5), log-aggregation (2), nodemanager (58), resourcemanager (64), rolling upgrade (2), router (1), scheduler (18), security (5), timelineclient (1), timelinereader (5), timelineserver (37), webapp (2), yarn (22), yarn-native-services (3), yarn-ui-v2 (1)
Cassandra	Build (14), CI (2), CQL/Interpreter (1), Cluster/Gossip (2), Cluster/Membership (1), Consistency/Coordination (3), Consistency/Hints (1), Consistency/Repair (11), Consistency/Streaming (1), Dependencies (24), Documentation/Javadoc (2), Documentation/Website (1), Feature/2i Index (8), Feature/Authorization (1), Feature/Compression (1), Feature/Counters (1), Feature/Lightweight Transactions (7), Feature/Materialized Views (8), Feature/SASI (3), Legacy/CQL (64), Legacy/Coordination (17), Legacy/Core (24), Legacy/Distributed Metadata (6), Legacy/Documentation and Website (3), Legacy/Local Write-Read Paths (37), Legacy/Observability (4), Legacy/Streaming and Messaging (17), Legacy/Testing (12), Legacy/Tools (42), Local/Caching (1), Local/Commit Log (1), Local/Compaction (25), Local/Config (17), Local/Memtable (2), Local/Other (4), Local/SSTable (6), Local/Startup and Shutdown (3), Messaging/Client (9), Messaging/Internode (7), Messaging/Thrift (1), Observability/Metrics (3), Packaging (16), Test/benchmark (1), Test/dtest/java (9), Test/dtest/python (2), Test/unit (2), Tool/auditlogging (1), Tool/nodetool (4)

Table 5. Components used in issues per project in the dataset.

$$f_\ell(x) = \begin{cases} e_1^{18} & \text{if } x = \text{Coding Effort} \\ e_2^{18} & \text{if } x = \text{Component Element Names} \\ e_3^{18} & \text{if } x = \text{Component Names} \\ e_4^{18} & \text{if } x = \text{Connector Data Names} \\ e_5^{18} & \text{if } x = \text{Connector Data} \\ e_6^{18} & \text{if } x = \text{Features} \\ e_7^{18} & \text{if } x = \text{InternalProtocol} \\ e_8^{18} & \text{if } x = \text{Misc} \\ e_9^{18} & \text{if } x = \text{Pattern Names} \\ e_{10}^{18} & \text{if } x = \text{Quality Attribute Names} \\ e_{11}^{18} & \text{if } x = \text{Release Names} \\ e_{12}^{18} & \text{if } x = \text{Tax-Better} \\ e_{13}^{18} & \text{if } x = \text{Tax-Depend} \\ e_{14}^{18} & \text{if } x = \text{Tax-Easy} \\ e_{15}^{18} & \text{if } x = \text{Tax-Hard} \\ e_{16}^{18} & \text{if } x = \text{Tax-Problem} \\ e_{17}^{18} & \text{if } x = \text{Tax-Programming Activities} \\ e_{18}^{18} & \text{if } x = \text{Technology Names} \end{cases} \tag{7}$$

The final encoding for the issue labels is then defined as the function $f_{\mathcal{L}}$ defined as

$$f_{\mathcal{L}}(I) = \bigvee_{i \in I} f_\ell(\text{I2O}(i)). \tag{8}$$

This can be seen as a some sort of binary pattern encoding, where we assign a number from 0 up to and including 2^18 to every combination of labels an issue may have. However, when using a binary pattern encoding, arbitrary fake/artificial relationships are formed. Take for example the case where $\{\text{a}, \text{b}, \text{c}, \text{d}\}$ is encoded as $\{[0,0],[0,1],[1,0],[1,1]\}$. This seems to suggest there is some link between b and d (and c and d), which may not be the case at all. Because this creation of artificial relationships, binary pattern encoding is often a poor choice [7]. The encoding we use is not completely arbitrary; the placement of the ones is meaningful. Hence, this is a meaningful encoding in this case.

We also included a field in the feature vector that contains the total number of labels attached to an issue.

5.8 Votes

The votes property is simply the number of votes. We directly include this as a feature.

5.9 Watches

The watches property is simply the number of people watching the issue. We directly include this number as a feature.

5.10 Parent

An issue may or may not have a parent. We include this as a categorical variable, which is 1 if the issue has a parent issue, and 0 if the issue does not have a parent.

5.11 Issue Links

This is a list of links to other issues. We cannot extract any useful information from the links themselves. What we do instead, is count the number of links and include this number as a feature.

5.12 Attachments

We will not be considering the content of the documents attached to issues. In stead, we will consider the number of attachments as a feature.

5.13 Sub-tasks

Similar to the issue links, this is simply a list of links to other issues. Hence, we simply count the amount of sub-tasks and include this number as a feature.

5.14 Dates

The exact dates on which an issue was created, updated, or resolved is not particularly interesting. Issues which have yet to come may also contain architectural design decisions. Hence, looking at dates themselves should not give any information.

In stead, we consider the time it takes to resolve an issue. Hence, we consider the time between the creation date and the resolution date, in seconds. We felt we could do this safely since most issues in the dataset have been resolved. In section 5.18, we will describe what we did for those issues which were not resolved.

5.15 Affected Versions & Fix Versions

We included the number of affected versions and the number of fix versions as features. Including or encoding the versions themselves does not make much sense, since there may also be future versions of the software for which architectural changes will have to be discussed. However, the amount of versions involved could potentially be a useful feature.

5.16 Text-based properties

While we did not consider the content of the text for this assignment, we did consider a number of features derived from the text. Specifically, we considered the length of the summary, the length of the description, the average length of a comment, and the combined total length of all comments as features. These lengths are the lengths in characters, also counting white-space. Additionally, we also included the number of comments as a feature.

We could have performed a more intricate analysis of the length of the summary, description, and comments. However, we decided not to do this. This is mostly because of time constraints and for sake of simplicity. Analysing the length of the text in different ways would lead to a lot of different things to try. Some example considerations: should we count words in stead of characters? Should we remove stop-words from the text? Should special formatting be removed? Should error messages pasted into issues be removed? We decided to leave this for future research, and only included the rather naive "length in terms of characters without additional processing" for now.

5.17 Final Feature Vectors

In the end, we came up with 21 hand-crated features. Due to the use of one hot and binary pattern encoding, this resulted in 49-dimensional feature vectors. It should be noted that this is a substantial reduction from the features used by Soliman et al. in [15]: they used 431 dimensional feature vectors. To summarize, we presented an overview of all features in table 7.

5.18 Missing Data

As mentioned in sections 5.5 and 5.14, not all issues have been resolved at the time of extracting data. We decided to remove these issues from the dataset. This is not only because these features are missing, but also because this means that the issue status is not stable; it would change in the future. This would result in 2 missing values and one feature subject to change. In Table 8, we give an overview of the amount of issues removed from the dataset. Less than 10% of all data is removed (183 issues). Additionally, most issues are removed from the two majority classes (non-architectural issues and existence issues). Few issues are removed from the executive class. It would have been

Project	Components
Hadoop	2.0.4 (1), 2.6.1-candidate (1), BB2015-05-TBR (2), Rhino (2), avro (1), build (1), classloading (1), classpath (1), compression (1), dependencies (1), documentation (1), features (1), hadoop (2), hdfs (1), hdfs-ec-3.0-must-do (1), jmx (1), maven (2), patch (1), perfomance (1), pull-request-available (3), qos (1), releasenotes (1), rhino (1), rpc (1), scripts (1), security (2), shell (1), solaris (1), supportability (1), webapp (1), windows (1)
HBASE	
HDFS	2.6.1-candidate (2), 2.7.2-candidate (2), BB2015-05-TBR (8), RBF (3), bigtop (1), critical-0.22.0 (1), decommission (1), features (2), flaky-test (2), gsoc (1), gsoc2015 (1), hdfs-ec-3.0-must-do (1), hdfs-ec-3.0-nice-to-have (4), locks (1), mentor (1), metrics (2), needs-test (1), ozoneMerge (4), performance (1), polling (1), pull-request-available (4), qos (1), rpc (1), snapshots (1), supportability (1), upgrade (1)
MapReduce	BB2015-05-TBR (3), bigtop (2), encryption (1), features (1), merge (1), perfomance (1), performance (1), plugin (2), rdma (1), shuffle (1), sort (1)
Tajo	JDK1.8 (1), Parquet (1), backup (1), failure-handling (1), fault-tolerance (1), hadoop (1), hbase_storage (1), high-availability (1), kafka_storage (3), metrics (1), newbie (3), parquet (1), performance (1), protobuf (1), restore (1), yarn (1)
Yarn	2.6.1-candidate (1), BB2015-05-TBR (3), CSI (1), Docker (20), YARN-5355 (3), auxiliary-service (1), capacity-scheduler (2), client (1), features (2), federation (1), fs2cs (2), incompatible (1), oct16-easy (1), oct16-hard (6), oct16-medium (6), pull-request-available (1), reservations (1), resourcemanager (1), scheduler (2), security (2), windows (2), yarn (1), yarn-2928-1st-milestone (7), yarn-5355-merge-blocker (5)
Cassandra	2i (1), 4.0 (1), 4.0-feature-freeze-review-requested (4), CI (1), CommunityFeedbackRequested (1), Core (1), GC-Seconds, (1), Java11 (1), LWT (6), Performance (4), Tools (1), UDF (1), Windows (3), abstract_types (1), alpha (1), audit (1), authentication (1), authorization (2), avro (1), beginner (1), benedict-to-commit (2), binary (1), bootcamp (1), build (1), bytearrayinputstream (1), bytearrayoutputstream (1), cache (2), cassandra (2), commit_log (1), compaction (12), compression (4), configuration (1), contrib (1), core (1), correctness (2), counters (3), cql (18), cql3 (5), cql3.3 (2), cqlsh (3), daemon (1), deletes, (1), dense-storage (1), distributed_deletes, (1), doc-impacting (12), docs (2), dtcs (4), dtest (2), easyfix (3), easytest (1), encryption (3), fallout (1), fqltool (1), hadoop (3), hints (1), incompatible (1), incremental_repair (1), index (1), inputformat (1), jbod-aware-compaction (2), jmx (3), keys (1), kubernetes (1), lcs (6), lhf (6), license (1), low-hanging-fruit (1), mapreduce (1), materialized views (1), memory (1), merkle_trees (1), messaging-service-bump-required (1), metrics (1), netty (1), newbie (1), nodetool (1), partitioners (1), patch (2), performance (41), pig (2), ponies (4), protocol (2), protocolv4 (1), protocolv5 (6), protocolv6 (1), pull-request-available (25), qa-resolved (11), remove-reopen (4), repair (9), repair, (1), sasi (3), secondary_index (2), security (9), service (1), storage (1), streaming (6), stress (3), synchronized (1), technical_debt (1), test (2), thrift (3), timeseries (1), timestamps (1), twcs (1), udf (2), virtual-tables (1), vnodes (1), windows (1)

Table 6. Issue labels used in issues in the dataset, listed per project.

Feature Name	Encoding	# Entries
Priority	Integer	1
Resolution	One-hot	4
Status	One-hot	3
Issue Type	One-hot	7
Issue Labels	Binary Pattern	18
Resolution Time	Integer	1
# Components	Integer	1
# Issue Labels	Integer	1
# Comments	Integer	1
# Attachments	Integer	1
# Votes	Integer	1
# Watches	Integer	1
# Issue links	Integer	1
# Subtasks	Integer	1
Parent (has a parent)	Boolean	1
Summary Length	Integer	1
Description Length	Integer	1
Cumulative Comment Length	Integer	1
Average Comment Length	Integer	1
# Affected Versions	Integer	1
# Fix Versions	Integer	1

Table 7. Final list of all the features we used, together with the amount of entries the occupy in the final feature vectors.

desirable to keep more issues from the property class, but we believed that removing the issues was still the best decision.

We also could have used imputation to fill in the missing values. However, simple imputation through means, medians or means could introduce bias in the dataset [3]. Additionally, for nearest neighbour imputation, we would need to define a distance measure between issues or features [3]. We believe it is difficult to do this in am unambiguous way, given the large amount of categorical data we are dealing with. This makes it difficult to define a good distance measure between data points.

The final key observation which led us to remove unresolved issues from the dataset, is that most (if not all) issues will eventually be resolved. Hence, when mining issues from repositories in the working phase of the classifiers, unresolved issues could be ignored temporarily and examined again later. We believe (or at least hope) that by excluding unresolved issues from both the training and working phase, no harmful bias is introduced in the dataset.

Target Label	# Removed
Existence	94
Executive	13
Property	32
Non-Architectural	44

Table 8. Amount of issues removed from the dataset by removing all issues that have not been resolved yet. Amounts are given per target label

6 Dimensionaliry Reduction

When working with large datasets and many features it can be useful to analyse what components in the data are important. Reducing the dimensionality, and increasing the interpretability without losing information is a big part of preprocessing the data before running classification or detection [8]. Reducing the dimensionality at the cost of a small amount of information can be useful, because high dimensional data generally makes learning difficult [7]. In this section we look at principal component analysis (PCA) for dimensionality reduction.

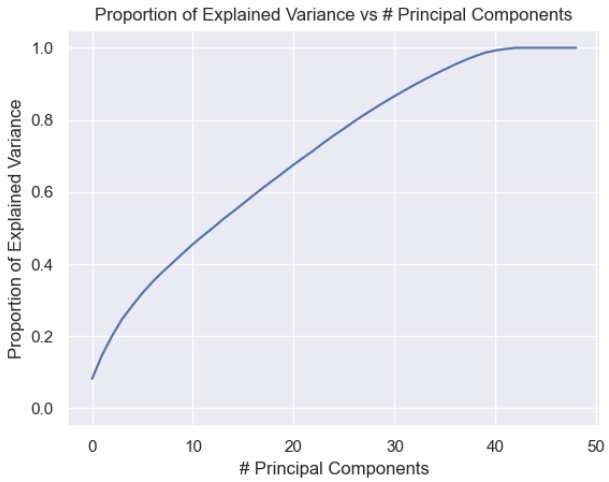

Fig. 5. The results of applying PCA on our dataset. The plot shows the fraction of the variance explained by the first x principal components for $1 \leq x \leq 49$.

6.1 Principal Component Analysis

Principal Component Analysis (PCA) is a standard exploratory data analysis (EDA)[8].

Given a centered collection of N data points (centered potentially after a transformation) in n dimensional space, commonly represented using a matrix $X \in \mathbb{R}^{N \times n}$, the goal of PCA is to find n (assuming $n < N$) orthonormal vectors $u_1, u_2, \ldots, u_n \in \mathbb{R}^n$ such that u_1 represents the direction in which the data displays the most variance. Next, u_2 is the direction orthogonal to that in which the data displays the most variance. This continues up to and including u_n [3].

The vectors $u_1, \ldots, u_n$ are called the principal components. They are given by the eigenvectors of the covariance matrix $\Sigma = \frac{1}{N} X^T X \in \mathbb{R}^{n \times n}$ [7]. Σ is positive semi-definite, because for all $\mathbf{x} \neq 0$, $x^T \Sigma x = x^T X^T X x = (Xx)^T Xx = |Xx|^2 \geq 0$. This means that the eigenvalues of Σ are real and nonnegative. Hence, without loss of generality, we can assume that the eigenvalues of Σ are ordered; $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \geq 0$. The percentage of the variance explained by an eigenvector u_i is dependent on its corresponding eigenvalue λ_i, and is given by

$$\frac{\lambda_i}{\sum_{k=1}^{n} \lambda_k} = \frac{\lambda_i}{\text{trace}(\Sigma)} \quad [7]. \tag{9}$$

Now, we define the matrix $U_r = \begin{bmatrix} u_1 & u_2 & \ldots & u_r \end{bmatrix}$. The product $X' = XU_r$ is a matrix of size $N \times r$, with $r \leq n$. Additionally, the way we built U_r guarantees that X' preserves as much of the variance in the data as possible. Hence, if we choose $r < n$, we can reduce the dimension of our data, while keeping as much of the variance as possible; i.e. we reduce the dimension, while (hopefully) retaining as much explanatory power as possible [3, 7].

We investigated whether PCA could be useful to reduce the dimensionality of our data. To this end, we plotted the proportion of explained variance as a function of the number of principal components. The results are given in figure 5. We would have hoped to see some sort of elbow plot, with a clear cutoff point for the amount of principal components. In stead, the variance keeps gradually increasing, except maybe for the last 5 components. We felt that choosing any particular cutoff point would feel somewhat arbitrary, and instead opted to not use PCA.

7 Classifiers

In an abundance of ML techniques, we decided to compare the performance of three classifiers, which are discussed below.

7.1 Naive Bayes

Naive Bayes (NB) is an example of a probabilistic classifier. It is based on Bayes' theorem with strong assumptions about the independence between features. However, even though these assumptions are often not satisfied in practice, Naive Bayes classifiers often perform rather well [12, 18, 1].

First, a small note on notation. Values for features are generally assumed to come from random variables $X_1, X_2, \ldots$. In general, we will use the shorthand notation $p(x_i)$ in order to denote $p(X_i = x_i)$.

Naive Bayes works based on conditional probabilities. Given a data point $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \ldots & x_N \end{bmatrix}^T$ to be classified into classes $\{C_1, C_2, \ldots, C_n\}$, the classifier uses $p(C_i \mid \mathbf{x})$ in order to come up with a classification. The main problem that it is difficult to estimate this probability from the training data directly, for instance because there are many possible $\mathbf{x}$ while the dataset contains relatively few examples [18, 1].

Instead, Bayes' Theorem is used to rewrite the conditional probability into

$$p(C_i \mid \mathbf{x}) = \frac{p(C_i)p(\mathbf{x} \mid C_i)}{p(\mathbf{x})} = \frac{p(C_i, \mathbf{x})}{p(\mathbf{x})} \quad [18]. \tag{10}$$

We now introduce the notation $\mathbf{x}_{\geq j}$ to denote the vector $\mathbf{x}_{\geq j} = \begin{bmatrix} x_j & x_{j+1} & \ldots & x_N \end{bmatrix}^T$, in order to shorten the following computations. We can now use the chain rule for probabilities in order to rewrite $p(C_i, \mathbf{x})$ as

$$\begin{aligned} &p(C_i, \mathbf{x}) \\ &\quad = p(C_i, x_1, x_2, \ldots, x_N) \\ &\quad = p(x_1 \mid \mathbf{x}_{\geq 2}, C_i)p(\mathbf{x}_{\geq 2}, C_i) \\ &\quad = p(x_1 \mid \mathbf{x}_{\geq 2}, C_i)p(x_2 \mid \mathbf{x}_{\geq 3}, C_i)p(\mathbf{x}_{\geq 3}, C_i) \\ &\quad = \ldots \\ &\quad = p(C_i)p(x_N \mid C_i) \prod_{k=1}^{N-1} p(x_k \mid \mathbf{x}_{\geq k+1}) \quad [1]. \end{aligned} \tag{11}$$

In Naive Bayes classifiers, the "naive" assumption is made that $x_1, x_2, \ldots, x_N$ are all independent, implying that $p(x_k \mid x_{\geq k+1}, C_i) = p(x_k \mid C_i)$. This means that we can further simplify the equation above in order to obtain

$$\begin{aligned} &p(C_i, \mathbf{x}) \\ &\quad = p(C_i)p(x_N \mid C_i) \prod_{k=1}^{N-1} p(x_k \mid \mathbf{x}_{\geq k+1}) \\ &\quad = p(C_i) \prod_{k=1}^{N} p(x_k \mid C_i) \quad [18]. \end{aligned} \tag{12}$$

Substitution of this result into equation 10 now yields that

$$p(C_i \mid \mathbf{x}) = \frac{1}{p(\mathbf{x})} p(C_i) \prod_{k=1}^{N} p(x_k \mid C_i). \tag{13}$$

This still leaves the determination of the probabilities $p(x_k \mid C_i)$. These have to be obtained from the training data, and the computation is dependent on the type of data used. For instance, in sklearn, there are different implementation for continuous data, count data, categorical data, and Boolean data (see below).

Equation 13 is combined with a decision rule to construct a classifier. The classifier should pick the hypothesis that is most probable, which is known as the *maximum a posteriori* (MAP) decision rule [12, 1]. Equation 14 is used to select the number ℓ of the class C_ℓ with the highest probability given the evidence $\mathbf{x}$. Note that the factor $1/p(\mathbf{x})$ is dropped; it is a constant factor, and thus not necessary for the actual computation.

$$\ell = \text{argmax}_{1 \leq i \leq n} \left\{ p(C_i) \prod_{k=1}^{N} p(x_k \mid C_i) \right\} \tag{14}$$

We used sklearn to implement our classifiers. The main problem we encountered is that sklearn offers multiple Naive Bayes implementations for different types of data. Specifically, there is ComplementNB[7] for discrete count data, CaterogicalNB[8] for categorical data, GaussianNB[9] for continuous data, and BernoulliNB[10] for Boolean data. There is no support for datasets consisting of mixed data types.

This meant that we had to make a custom classifier which combined the results from individually trained Naive Bayes classifiers. This is straightforward to do because of the independence assumptions, and because sklearn allows users to obtain the log probabilities for all the class, i.e. we can extract the quantity $\log(p(C_k \mid \mathbf{x}))$ for every class from every classifier. Working with log probabilities and likelihoods yields to more numerically stable computations [7].

Now, suppose that we have Naive Bayes classifiers $P_1, \ldots, P_L$, trained on corresponding disjoint subsets $S_1, \ldots, S_L \subset \mathbb{N}^+$. such that $S_1 \cup S_2 \cup \ldots \cup S_L = \{1, 2, \ldots, N\}$. If $j \in S_i$, then classifier i uses feature $\mathbf{x}_j$. We will let $S_i[\mathbf{x}]$ denote all entries from $\mathbf{x}$ which correspond to the features in S_i. Then, every classifier P_i can compute the quantity $\log(p(C_k \mid S_i[\mathbf{x}]))$ for every class C_k.

Now, note that we have

$$\begin{aligned} &\sum_{i=1}^{L} \log(p(C_k \mid S_i[\mathbf{x}])) \\ &= \sum_{i=1}^{L} \left(\log(p(C_k)) + \sum_{j \in S_i} \log(p(X_j = x_j \mid C_k)) \right) \\ &= L \log(p(C_k)) \sum_{j=1}^{N} \log(p(x_j \mid C_k)) \\ &= \log(p(C_k \mid \mathbf{X})) + (L-1) \log(p(C_k)), \end{aligned} \tag{15}$$

which meant that we can sum the log of the probabilities from all models, and subtract the priors $L - 1$ of times, in order to obtain the log probabilities over all features.

In the end, we used three different Naive Bayes classifiers for three different types of data:

- *MultinomialNB (count data)*: number of components, number of issue labels, number of votes, number of watches, number of issue links, number of attachments, number of sub-tasks, resolution time, number of fix versions, number

[7] https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.ComplementNB.html

[8] https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.CategoricalNB.html#sklearn.naive_bayes.CategoricalNB

[9] https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB

[10] https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html#sklearn.naive_bayes.BernoulliNB

of affected versions, summary length, description length, number of comments, cumulative length of all comments, average length per comments.

- *CategoricalNB (categorical data)*: issue status, issue type, resolution, priority.

  Issue status, issue type, and resolution had to be converted back into a single value. This is because sklearn expects categorical variables to be encoded using a single values. We applied the transformation $e_i^n \to i$ in order to transform the issue status, issue type, and resolution into single values.

- *BernoulliNB (Boolean data)*: parent (has a parent), issue labels.

  We thought it would be best to interpret the binary pattern encoding used for issue labels as a series of Booleans. Each Boolean indicates whether the issue contains a label from the corresponding ontology class.

For the classifiers, we assumed a uniform prior. Hence, $p(C_i) = 1/n$, where n is the total number of class. The same reasoning applies here as for the choice of uniform class limiting, explained in section 4.

There is one hyperparameter we have to optimize. This is the smoothing parameter α. The smoothing parameter is used to avoid probabilities of 0 in case of unseen data. This parameter is used to apply Laplace smoothing, which modifies probability estimates for measurements $m_1, m_2, \ldots, m_d$ with $m_1 + m_2 + \ldots + m_d = M$ coming from a binomial ($d = 2$) or multinomial distribution from m_i/M to $(m_i + \alpha)/(M + d\alpha)$. The problem being solved here is that if one of the m_i was not observed during training but is encountered during prediction, then the probability estimate would be 0, which might not be accurate. [1]. We experimented with $\alpha \in \{0, 0.25, 0.5, 0.75, 1.0, 1.5, 2.0, 3.0\}$.

7.2 Decision Tree

Decision trees are one of the most intuitive classification algorithms. They predict the output class of an input feature by making several 'decisions' on the attributes of the input feature [4].

As an example, say we have input features consisting of the attributes $A1$ 'makes all homework' and $A2$ 'hours attended'. The possible classes for prediction are $C1$ 'low grade' and $C2$ 'high grade'. Now the decision tree can make decisions on these attributes $A1$ and $A2$ to come up with a predicted class. Example decisions can be: 1) if $A1$ is false then $C1$, else $C2$ and 2) if $A2 < 10$ then $C1$ else $C2$. By putting such decisions in the form of a tree, we obtain a decision tree. Then during the classification process, the algorithm traverses through the tree based on the values of the attributes of the input feature. In Figure 6 we show an example of a decision tree based on the mentioned attributes in this paragraph.

While the decision process is simple, constructing a decision tree is quite hard. In fact, we do not have an efficient algorithm that can construct an optimal decision tree. Hence, we reach out to heuristic algorithms that build a reasonably good tree in reasonable time [4].

Most heuristic algorithms are greedy and construct a tree in a top-down fashion using training samples. This is an iterative approach, where in each iteration we split the current leaf nodes. When a leaf node is pure or we met another stopping criterion, we do not split the leaf node. During the construction of the tree, we want to optimize each split. We want to select and split based on the attribute that yields the most pure branches. A pure branch means that all training samples in the branch are of the same class. The most impure branch is one where each class has a similar amount of input samples. One of the most used

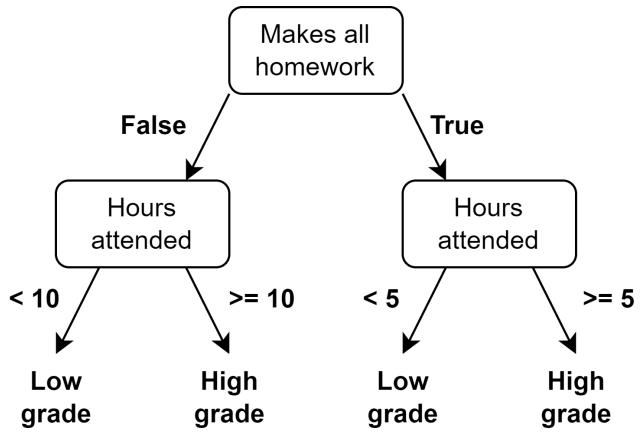

Fig. 6. Example of a decision tree

impurity measures is entropy. The entropy of a split is defined as follows:

$$S_l = -\sum_{i=1,\ldots,q} \frac{n_i^l}{n^l} \log_2 \left(\frac{n_i^l}{n^l} \right) \tag{16}$$

where q denotes the number of classes in the tree, n^l denotes the total number of training samples for this split, and n_i^l denotes the number of data points of the class q [4].

The minus sign in the front ensures that the entropy of a split is always non-negative. The entropy is 0 if all samples are of a single class after split and it is maximal if each class has about the same amount of samples.

One problem of decision trees is that they are susceptible of over-fitting. During the construction of a tree, the decisions might be over-specific for the training samples. When presented with unseen data, the tree might have a difficult time predicting those samples. Fortunately, there exist methods such as pruning that should prevent over-fitting, at least to an extend. With pruning, we remove branches of the tree and replace them by leaf nodes. The final classification is then the majority class of the leaf node [4].

Because of the greedy behaviour of the algorithm, we tend to find important features (i.e. features that best split the data into the categories) at the top of the tree. Since we can also see how the decisions are made in a decision tree, we can find out which issue properties are important for the classification.

For this classification task, we used the DT algorithm provided by `sklearn`, and tuned the following parameters for the best result:

- `class_weight`: Predetermined weights of each class. This might be useful for imbalanced data. The algorithm can e.g. give a higher penalty for incorrectly classifying a minority class.

- `criterion` $\in \{gini, entropy, log\_loss\}$: The function that measures the purity of the splits

- `max_depth` $\in \{5, 10, 15, 20, 25\}$: The maximum depth is the length of the longest path from a root to a leaf, thus essentially the maximal amount of decisions for a single sample. We want to tune this to find a value that prevents overfitting, but gives good performance.

- `max_features` $\in \{None, sqrt, log2\}$: The algorithm randomly selects a subset of `max_features` attributes, calcu-

lates the impurity for each of the attribute splits and selects the most pure split.

- `splitter` $\in \{best, random\}$: When splitting on an attribute, the algorithm can either try all possible splits and select the best (i.e. the *best* splitter) or it tries a random split for each attribute (i.e. the *random* splitter). The *best* splitter is more prone to overfitting, while also being computationally more expensive.

7.3 Random Forest

Random Forests (RF) is an ensemble learning method for classification. Such a method uses more than one learning algorithm in order try to improve the overall performance. RF creates multiple decision trees and combines the output of the decision trees using majority voting [5].

Breiman provided a theorem (Equation 17) where the generalization error (PE^*), the probability that a random forest makes wrong decisions on unseen data, is given by a statistical correlation measure ($\bar{\varrho}$) and the strength of the tree learning algorithm (s)[5]. It shows an upper bound for the generalization error for an infinitely large forest size.

$$PE^* \leq \bar{\varrho}(1 - s^2)/s^2 \tag{17}$$

The $\bar{\varrho}$ is a measure of how often different trees provide the same classification results on the test data. It is maximal if all trees obtain the same classification, while it is minimal if the results are not correlated.

The strength of the algorithm (s) ranges in [0,1]. When $s = 1$ all the trees make correct decisions on the test data consistently, and for $s \to 0$ the trees makes more incorrect decisions than right decisions [5].

What this theorem essentially describes is that we want to have good performing individual trees, while also making sure that the individual trees differ a lot from each other in terms of how they make decisions. For example, it is important that different trees predict the output based on different attributes. To achieve the latter, we need an algorithm that introduces randomness into the construction of a decision tree.

For this, we use bagging and random feature selection [5]. Bagging is the process of randomly copying samples from the original dataset to a new dataset, until this new dataset has the same size as the original dataset. Random feature selection is related to the construction of a tree. With random feature selection, only a random subset of features is available that can be used for the split. From this random selection of features, the feature that yields the best split (i.e. the most information gain) is chosen. These two methods together introduce randomness into the tree creation.

For this classification task, we used the RF algorithm provided by `sklearn`. The hyperparameters for this algorithm have a lot of overlap with those of the DT. We tuned the same parameters with one exception; instead of `splitter` the following parameter was tuned:

- `n_estimators` $\in \{50, 100, 200, 500\}$: A real value indicating the number of decision trees in the forest.

8 Feature Shuffling

We experimented with feature shuffling to get an estimation of the importance of each of the issue properties. This works as follows. First, we train a classifier using all the non-shuffled issue properties and measure the performance. Then we perform a loop, where for each iteration we select one of the features. Now it is useful to think about the dataset as a matrix, where the columns represent the issue properties and the rows represent the data points. During each iteration we shuffle the data in a single column. After shuffling this column, we train a new classifier and measure the performance again.

This can give us insight into the usefulness of each issue property as follows. If the performance of the classifier drops a lot after shuffling a certain issue property, it is very likely to be an important feature. If the performance does not change, the issue property is probably not important. In case the performance increases, which is highly unlikely, the issue property is probably noise that harms the classifier.

One important remark to make is that we use column and issue property interchangeably. This is due to the following. If we one-hot-encode a certain issue property, it leads to multiple columns. However, for feature shuffling we consider all the columns belonging to a single issue property as a single column. This is the only way we can measure the importance of the issue properties with feature shuffling. Otherwise, shuffling these one-hot-encoded features does not yield easily explainable information.

9 Results

In this section we present the results we obtained with our experiments.

First of all, we show the best parameters that were found for each of the classifiers using sklearn GridSearchCV. Table 9 contains the best parameters for the detection task and Table 11 contains the best parameters for the classification task. We used these settings to obtain the results for the detection task in Table 10 and the classification task in Table 12.

For the detection task, the Naive Bayes classifier has a F_1-score of 0.461. The decision tree performs a lot better with a F_1-score of 0.660. The root of the detection tree is shown in Figure 11.

When we use an ensemble method with many trees, i.e. random forest, we improve this performance by 3.1% to 0.691.

For the classification task, Naive Bayes performs worse than random guessing performance (i.e. 0.25 [6]) with a F_1-score of 0.181. Decision tree improves a lot upon this with a F_1-score of 0.432. The root for the classification tree is shown in 12.

Although the random forest classifier is better than the decision tree classifier according to the results of the grid search, the performance on the test set is worse: 0.396.

Figures 7 and 8 show the confusion matrices for the detection task and classification task respectively. These were obtained with the best classifier for each of the tasks.

The results of feature shuffling are shown in Figures 9 and 10. Both results were obtained using the best classifier for each task, thus Random Forest for detection and Decision Tree for classification. For these classifiers, we used the settings in Tables 9 and 11 respectively, as these settings were performing the best.

Classifier	**Parameter**	**Value**
Naive Bayes	alpha	0
Decision Tree	class_weight	Enabled
	criterion	log_loss
	max_depth	5
	max_features	None
	splitter	random
Random Forest	class_weight	Enabled
	criterion	entropy
	max_depth	10
	max_features	log2
	n_estimators	50

Table 9. Settings used for the detection task. Enabling the class_weight means that we divided the total number of samples by the amount of samples of each class. This means the minority class has the highest weight and the majority class the smallest weight.

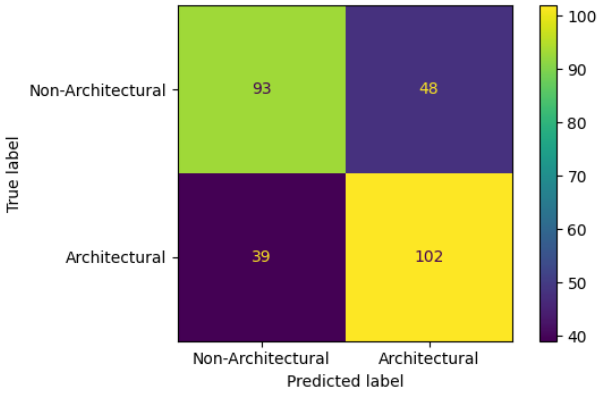

Fig. 7. Confusion matrix for the detection task. This was obtained using the Random Forest classifier.

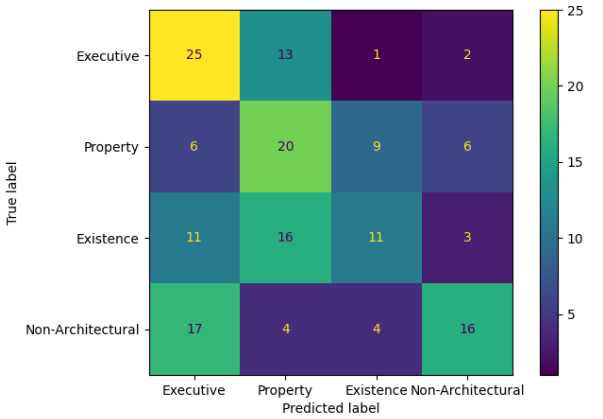

Fig. 8. Confusion matrix for the classification task. This was obtained using the Random Forest classifier.

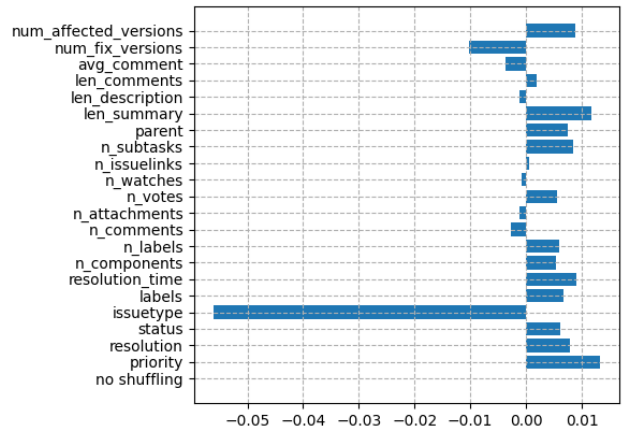

Fig. 9. Results of feature shuffling for the detection task. These results were obtained using the Random Forest classifier.

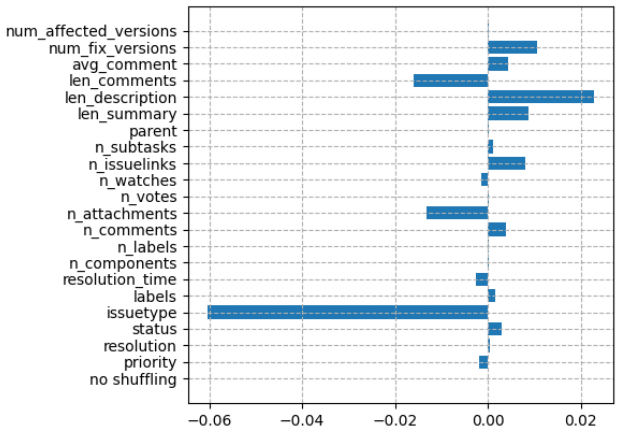

Fig. 10. Results of feature shuffling for the classification task. These results were obtained using the Decision Tree classifier.

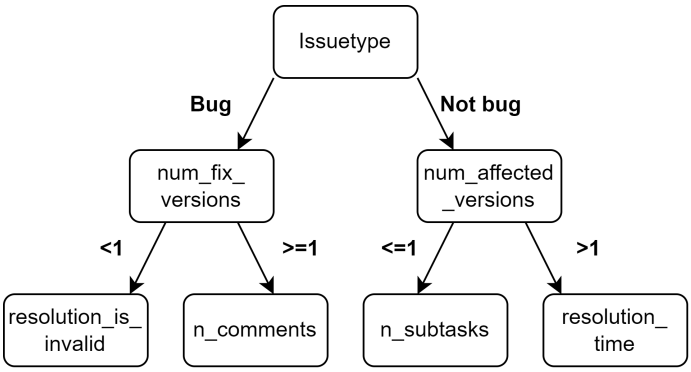

Fig. 11. The root and a few children of the decision tree for the detection task.

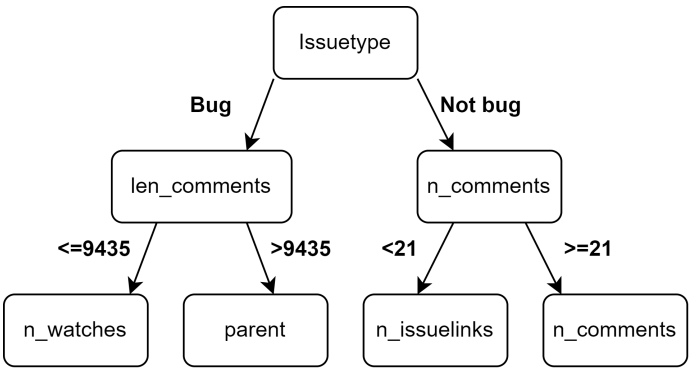

Fig. 12. The root and a few children of the decision tree for the classification task.

Classifier	F1-score training	F1-score test
Naive Bayes	0.510	0.461
Decision Tree	0.678	0.660
Random Forest	0.718	0.691

Table 10. Detection performance of the classifiers.

Classifier	Parameter	Value
Naive Bayes	alpha	0.0
Decision Tree	class_weight	Enabled
	criterion	log_loss
	max_depth	5
	max_features	None
	splitter	best
Random Forest	class_weight	Enabled
	criterion	entropy
	max_depth	10
	max_features	log2
	n_estimators	200

Table 11. Settings used for the classification task. Enabling the class_weight means that we divided the total number of samples by the amount of samples of each class. This means the minority class has the highest weight and the majority class the smallest weight.

Classifier	F1-score training	F1-score test
Naive Bayes	0.203	0.181
Decision Tree	0.406	0.432
Random Forest	0.458	0.396

Table 12. Classification performance of the classifiers.

10 Discussion

For both tasks we found that the Naive Bayes classifier is rather useless. It performs similar or worse than random guessing, making it unusable for classifying issues. For detection, the random guessing performance is around a 0.500 F_1-score [6], while Naive Bayes obtains 0.461 F_1-score. Even if we were to "invert" all the predictions, we would still obtain an F_1 score of only 0.539, which still is barely better than randomly guessing. For the classification task, random guessing performance is around 0.250 F_1-score [6]. Naive Bayes obtains 0.181 F_1-score, which is even below random guessing performance. Because of the poor performance, we also did not examine any of the explanatory aspects of the trained Naive Bayes classifiers.

The random forest classifier outperforms the decision tree by 3.1% (0.691 and 0.660 F_1-score) on the detection task, while the decision tree outperforms random forest on the classification task by 3.6% (0.432 and 0.396 F_1-score). This shows that using a more complicated and computationally expensive model (i.e. random forest) does not always yield superior performance.

For the best performing model for each task, we created confusion matrices. For the detection task this matrix seems to be be fine in the sense that the wrong predictions are quite balanced and the highest values can be found on the main diagonal, although the classifier tends to predict non-architectural issues as architectural more often than the other way around. The confusion matrix for the classification task is quite a bit worse in general, in the sense that the off-diagonal entries are often also quite high – often as high or higher than some diagonal entries. This seems to suggest that there is not a lot of difference between the different issue types in terms of issue properties; it is difficult for the classifier to make the correct predictions.

Now we compare the results with what was obtained in [6], where neural networks were used. The main results with neural networks are 0.81 F_1-score for the detection task and 0.34 F_1-score on the classification task. Note that for this research we used different, and in general more, preprocessing. The random forest classifier performs worse than the neural network for the detection task (0.691 vs 0.81 F_1 score). This could be because 1) neural networks are more powerful for this task or 2) the new preprocessing we introduced in this research does not work for the detection task as effectively. Fortunately, we did make a rather big improvement on the classification task. Here we improved the performance from 0.34 to 0.432 using the decision tree classifier.

In [6], issues were also classified using the issue text (summary and description). For the detection task a F_1-score of 0.83 was obtained and for the classification task 0.57 F_1-score. This is a lot better than what we obtained in this research. However, text preprocessing and classification is much more expensive than issue properties. Hence, it is worth it to explore whether we can improve the performance of the classifiers using issue properties instead.

Figures 9 and 10 show the results of our feature shuffling. For the detection task we see that shuffling the issuetype property leads to the largest performance drop. We therefore suspect that issuetype is a feature that is important in determining whether an issue is architectural. For the other features we do not see much difference and we cannot guarantee whether the performance difference is due to randomness or whether the features have a real impact on the performance.

For the classification we have similar results. There we also see that shuffling the issuetype reduces the performance of the classifier a lot. The properties len_comments and n_attachments also seem to contain somewhat useful information. The property len_description is by far the least useful property according to Figure 10.

Figures 11 and 12 show the root nodes and a few of the child nodes of the decision trees for the detection and classification task respectively. An interesting result is the fact that issuetype is the root node for both trees. More specifically, we see that it makes a distinction based on whether an issue is a bug or not. Note that with our feature shuffling analysis we also found that the issuetype property seems to contain quite a bit of information regarding the classification.

Furthermore, we see that the number of comments is also used for both detection and classification. In fact, three of the top seven splits for the classification task involve some metadata of the comments. For the rest it is hard to draw any further conclusions about these results. These require a separate research on its own.

10.1 Future work

Future work could focus on finding out what causes the performance differences between [6] and our results. We can do this by running the neural networks proposed in [6] with the newly pre-processed data we introduced in this report, and by running the random forest classifier with the data used in [6].

It would also be interesting to find out whether an ensemble model that uses both text and issue properties could improve the overall performance. This was done in [6], but this was not successful back then; no performance gains were observed. However, with our new preprocessing this could be a path to explore.

We can also search for new architectural issues of the minority classes using the classifier obtained in this research, in order to increase the size of the dataset. This could be useful because a lack of data is one of the major challenges for the neural network based approaches [6]. For this we would need to classify new issues pulled from JIRA and manually label these. By comparing the predicted labels and manual labels, we can determine

whether this is an effective approach for finding new issues of the minority classes.

Using feature shuffling, we found that the issue property 'issue type' seems to contains quite a bit of information about the type of issue. It would be interesting to explore exactly which issue types (as in the issue property) correspond to which issue type (as in existence, property, etc.). This might allow us to find new issues of a specific type to increase the amount of samples for the minority classes. Finding important issue properties can also be useful because it opens up opportunities to filter issues based on those properties when retrieving them from JIRA. For instance, if we can pinpoint the relation between issue type and predicted label more accurately, we might be able to retrieve only issues with a specific issue type from JIRA when looking for new issues to expand the dataset.

In the future, we could also look at using other classifiers with explanatory power. One possibility would be to revisit the idea of logistic regression, but used in a one-vs-rest classifier setting [16].

11 Conclusion

For detecting architectural issues, we found that the random forest classifier works best with a F_1-score of 0.691. For classifying the subtypes, we found that the decision tree classifier was the best with a F_1-score of 0.432. Compared to [6], the performance is 10.9% worse for detection. However, for the classification task we improved by 9.2%. Detecting and classifying issues using issue properties still seems to be less effective than using issue text (summary and description). However, using issue properties is much cheaper computationally and is a path worth exploring further. The obtained insights in feature importance may also help in the development of more focused search approaches for issues containing architectural design decisions.

References

[1] Daniel Berrar. "Bayes' theorem and naive Bayes classifier". In: *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics* 403 (2018), p. 412.

[2] Manoj Bhat et al. "Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach". In: *Software Architecture*. Ed. by Antónia Lopes and Rogério de Lemos. Cham: Springer International Publishing, 2017, pp. 138–154. ISBN: 978-3-319-65831-5.

[3] Michael Biehl. *The Shallow and the Deep - A biased introduction to neural networks and old school machine learning*. University of Groningen. 2022.

[4] Leo Breiman. *Classification and regression trees*. 1st ed. Wadsworth International Group, 1984.

[5] Leo Breiman. "Random Forests". In: 45 (2001), pp. 5–32.

[6] Arjan Dekker and Jesse Maarleveld. "Mining for Architectural Design Decisions in Issue Tracking Systems using Deep Learning Approaches". Master's Research Internship Report. University of Groningen, 2022.

[7] Herbert Jaeger. *Machine Learning (Lecture Notes)*. University of Groningen. 2023.

[8] Ian T. Joliffe and Jorge Cadima. "Principal component analysis: a review and recent developments". In: 374 (2016). DOI: http://dx.doi.org/10.1098/rsta.2015.0202.

[9] Philippe Kruchten, Patricia Lago, and Hans van Vliet. "Building Up and Reasoning About Architectural Knowledge". In: *Quality of Software Architectures*. Ed. by Christine Hofmeister, Ivica Crnkovic, and Ralf Reussner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 43–58. ISBN: 978-3-540-48820-0.

[10] Philippe B Kruchten. "An Ontology of Architectural Design Decisions in Software-Intensive Systems". In: *2nd Groningen workshop on software variability* (2004), pp. 54–61.

[11] Christian Manteuffel, Paris Avgeriou, and Roelof Hamberg. "An exploratory case study on reusing architecture decisions in software-intensive system projects". In: *Journal of Systems and Software* 144 (2018), pp. 60–83. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.05.064. URL: https://www.sciencedirect.com/science/article/pii/S0164121218301110.

[12] Stuart Russel and Peter Norvig. *Artificial Intelligence: a modern apporach*. 3rd ed. Pearson Education Limited, 2016. ISBN: 9781292153964.

[13] Arman Shahbazian et al. "Recovering Architectural Design Decisions". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, pp. 95–9509. DOI: 10.1109/ICSA.2018.00019.

[14] Mohamed Soliman, Matthias Galster, and Matthias Riebisch. "Developing an Ontology for Architecture Knowledge from Developer Communities". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. 2017, pp. 89–92. DOI: 10.1109/ICSA.2017.31.

[15] Mohamed Soliman et al. "Evaluating Approaches to Find Architectural Design Decisions in Issue Tracking Systems". Not yet published/no pre-print available. 2023.

[16] David M.J. Tax and Robert P.W. Duin. "Using two-class classifiers for multiclass classification". In: *2002 International Conference on Pattern Recognition*. Vol. 2. 2002, 124–127 vol.2. DOI: 10.1109/ICPR.2002.1048253.

[17] Hans van Vliet. *Software Engineering, Principles and Practice*. 3rd ed. John Wiley & Sons Ltd., 2008.

[18] Geoffrey I. Webb. "Naïve Bayes". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 713–714. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_576. URL: https://doi.org/10.1007/978-0-387-30164-8_576.

A Ontology Class Additions

In table 13, the additions to the ontology classes used in [6] are given. We also added a number of new classes:

- Coding Effort: Organized coding efforts (e.g. "sprints"), usually lasting for a very short duration. This is sometimes organised to get a lot of work done in a short amount of time to meet some particular goal.
- Features: singleton category for the word "Features", because we felt like this label does not fit any particular ontology class, while we also felt like the label is special enough to not add it to the Miscellaneous class.
- Internal Protocol: A category for labels which do not quite fit the definition of a component or connector, nor are they a specific technology. This class is reserved for protocols or concepts which (generally) only make sense within the context of the system itself, but do not fall into any of the other ontology classes. An example is "lcs", which is a compaction strategy in the Cassandra project.
- Miscellaneous: a class for labels which do not fall into any particular category, and do not share any common overarching properties.
- Release Names: label referencing a specific release of version of the software. Such labels are often used to denote that an issue must be fixed before some version of the software is released.

Category	Additions
Coding Effort	bb2015-05-tbr, fs2cs, gsoc, gsoc2015, oct16-easy, oct16-hard, oct16-medium
Component Element Names	bigtop, capacity-scheduler, classpath, counters, locks, scripts
Component Names	auxiliary-service, core, dtest, gcseconds,, hbase_storage, kafka_storage, materializedviews, merkle_trees, partitioners, resourcemanager, udf, virtual-tables, vnodes, webapp
Connector Data Names	2i, binary, index, inputformat, secondary_index, timeseries, timestamps
Connector Names	bytearrayinputstream, bytearrayoutputstream
Features	features
Internal Protocol	compaction, compression, dense-storage, distributed_deletes,, dtcs, federation, incremental_repair, jbod-aware-compaction, lcs, protocolv4, protocolv5, protocolv6, rbf, twcs
Miscellaneous	4, alpha, audit, benedict-to-commit, classloading, communityfeedbackrequested, doc-impacting, hints, lhf, license, mentor, messaging-service-bump-required, metrics, ponies, qa-resolved, remove-reopen, repair, reservations, shuffle
Pattern Names	abstract_types, backup, daemon, failure-handling, plugin, restore, synchronized
Quality Attribute Names	fault-tolerance, high-availability, perfomance, qos
Release Namess	2.0.4, 2.6.1-candidate, 2.7.2-candidate, 4.0, 4.0-feature-freeze-review-requested, critical-0.22.0, hdfs-ec-3.0-must-do, hdfs-ec-3.0-nice-to-have, yarn-2928-1st-milestone, yarn-5355, yarn-5355-merge-blocker
Tax-Depend	dependencies
Tax-Easy	beginner, easyfix, easytest, low-hanging-fruit, newbie
Tax-Problem	fallout, flaky-test, incompatible, technical_debt
Tax-Programming Activities	bootcamp, ci, commit_log, contrib, decommission, deletes,, docs, documentation, merge, needs-test, ozone-merge, pull-request-available, releasenotes
Technology Names	cql3, cql3.3, cqlsh, csi, fqltool, java11, jdk1.8, lwt, maven, nodetool, parquet, rdma, rhino, sasi, shell, solaris, tools

Table 13. Additions to the ontology classes and lexical triggers as defined in [14] and [6]. The "Coding Effort", "Release Names", "Internal Protocol", "Miscellaneous", and "Features" class were newly added.

B Labels Per Ontology Class

Class	Labels
Coding Effort	bb2015-05-tbr, fs2cs, gsoc, gsoc2015, oct16-easy, oct16-hard, oct16-medium
Component Element Names	bigtop, capacity-scheduler, classpath, configuration, counters, keys, locks, scripts
Component Names	auxiliary-service, client, core, docker, dtest, gcseconds,, hadoop, hbase_storage, kafka_storage, mapreduce, materializedviews, memory, merkle_trees, partitioners, resourcemanager, scheduler, service, snapshots, storage, udf, virtual-tables, vnodes, webapp
Connector Data Names	2i, binary, index, inputformat, secondary_index, timeseries, timestamps
Connector Names	bytearrayinputstream, bytearrayoutputstream, streaming
Features	features
InternalProtocol	compaction, compression, dense-storage, distributed_deletes,, dtcs, federation, incremental_repair, jbod-aware-compaction, lcs, protocolv4, protocolv5, protocolv6, rbf, shuffle, twcs
Misc	alpha, audit, benedict-to-commit, classloading, communityfeedbackrequested, doc-impacting, hints, lhf, license, mentor, messaging-service-bump-required, metrics, ponies, qa-resolved, remove-reopen, repair,, reservations
Pattern Names	abstract_types, authentication, authorization, backup, cache, daemon, encryption, failure-handling, plugin, polling, restore, rpc, synchronized
Quality Attribute Names	correctness, fault-tolerance, high-availability, perfomance, performance, qos, security, supportability
Release Names	2.0.4, 2.6.1-candidate, 2.7.2-candidate, 4.0, 4.0-feature-freeze-review-requested, critical-0.22.0, hdfs-ec-3.0-must-do, hdfs-ec-3.0-nice-to-have, yarn-2928-1st-milestone, yarn-5355, yarn-5355-merge-blocker
Tax-Better	upgrade
Tax-Depend	build, dependencies
Tax-Easy	beginner, easyfix, easytest, low-hanging-fruit, newbie
Tax-Hard	stress
Tax-Problem	fallout, flaky-test, incompatible, technical_debt
Tax-Programming Activities	bootcamp, ci, commit_log, contrib, decommission, deletes,, docs, documentation, merge, needs-test, ozone-merge, patch, pull-request-available, releasenotes, repair, sort, test
Technology Names	avro, cassandra, cql, cql3, cql3.3, cqlsh, csi, fqltool, hdfs, java11, jdk1.8, jmx, kubernetes, lwt, maven, netty, nodetool, parquet, pig, protobuf, protocol, rdma, rhino, sasi, shell, solaris, thrift, tools, windows, yarn

Table 14. Overview depicting to which ontology classes the labels of the issues in the dataset have been assigned.