An extraction and analysis of software development profiles from project quality data

Authors: Cristina Aguilera, Jesús Antoñanzas, Laia Albors, Sonia Rabanaque

Abstract:
In this project we use The Technical Debt Dataset (a set of project measurement data from 33 Java projects) to find distinct groups of developer profiles based on distinct characteristics, by clustering the developers using the K-means algorithm. Then, an analysis of the different types of bugs that each developer group tends to introduce is performed.

The results of our project are expected to be used by department leaders and general managers to help their developers be more efficient by reducing the number of bugs they produce providing them with training options, for example.

Table of contents

1. Business understanding

1.0. Project Goals

Our goal is straightforward: **to relate or understand how different developer profiles tend to introduce technical debt / bugs**. This goal also comes with the issue of defining or identifying dev. profiles.

1.1. Business objectives

- **Background**

  Our problem lies within the area of business development. We want to be able to tailor resources to the needs of our developers in a semi-supervised fashion, which will improve the quality of our software. That is, provide them with the resources needed to round up their competences. This, then, will be achieved by **determining the bugs or complexities each profile of developer is prone to introduce**.

  The results of our project are expected to be used by department leaders and general managers to help their developers be more efficient by reducing the number of bugs they produce by providing them with training options, for example; or directly by developers if they want to improve their self-awareness. Bearing in mind that developers want to improve their abilities and managers are (or should be) interested in dealing with their projects more efficiently, we see the potential results as very favourable for the company.

- **Business goals and success criteria**

  The problem that we intend to solve is to identify the most common types of bugs that each developer profile makes so that actions can be taken to minimize the number of bugs they produce. In order to do this, we consider the main questions to be asked:

  - What are the types of developer profiles in our business?
  - Which are the main bugs that each developer's profile tends to produce?

  Of course, we consider it as an additional business requirement to keep developers integrated, content and efficient. The benefits that good results from this project could bring are significant: it would be possible to tailor the skills and capacities of our developers to the actual problems they face all keeping in mind their weaknesses and strengths. In turn, this will allow them to be more efficient in the development of projects, thus creating higher quality products.

We consider the following as success criteria, both from an analysis perspective and a business one:

1. To discover, at least, three distinct developers profiles and their associated common issues.
2. To reduce by ten percent the number of bugs created by each developer's profile by giving useful insight about the bugs they are more likely to do.

Developers, analysts and managers will assess the success criteria. The first ones more personally (considering if the analysis is true to what they feel and if it really has helped in how they write code or develop applications), analysts in considering if there has been some meaningful discovery, and managers in assessing the success criteria in a general way (the impact of this study on the quality of life trend of their projects - quantity of bugs, issues and meeting of deadlines, among others).

1.2. Assessment of current situation

In this section we discuss where we are right now, what resources we have available, how we are going to tackle this project and other aspects related to the development.

- **Inventory of resources.** Organized in different categories, we present the resources available at the moment that make this project possible.
    - **Hardware resources**: computing equipment, desktop and office material, available to our employees at any time.
    - **Data sources**: the Technical Debt Dataset (a csv dataset with information about technical debts, code smells, issues and general information from several Java projects).
    - **Knowledge sources**: we have papers given in class (written documentation), lectures (experts) and other sources of information (mainly online sources).
    - **Personnel sources**: the 4-person project team, with huge knowledge on data science and related fields; the domain expert is Dr. Davide Taibi, the creator of the Technical Debt Dataset, that will be available in an ask-me-anything session (oct. 5-9); Silverio Martínez Fernández is available until late October for technical or business support to the project.
- **Requirements, assumptions, and constraints**
    - **Target group profile**: department leaders and general managers to help their developers be more efficient.

- The project must fit some important **requirements**. The developed code must be comprehensive for other developers as well as repeatable for anyone inside and outside the work team. It must have accurate results and be easily maintainable by the company.
- **Time constraints**: given that this is an academic project belonging to half a course, it is clear that the results achievable will not be the same as if we had more available hours.
- **Scope constraint**: related to the number of available hours, the scope of the analyses that could be performed with our data is much bigger than what we have now. So, we could in fact raise a lot of questions once the initial ones are answered, or even iterate in order to refine the latter ones, but that would be out of the scope of the project.
- An important **assumption** is taken: in order to perform the analyses, we take for granted that there are significant patterns (relevant to our questions) in the Technical Debt Dataset, but that may very well be false. We will find out either way and may be able to answer other questions along the way, though.

- **Risks and contingencies**. In order to be the most prepared to deal with any kind of adverse situation, here we establish the risks (**R**) that could hinder the development of the project and briefly go over the contingency methods (**C**). This is just an overview, but in a more thorough risk management plan there would be, among others:

  I. More, thorough risks.
  II. A more detailed analysis of each risk.
  III. A Risk Priority Number (RPN) for each risk.

  - **Business risks**: There really is not any business risk: the worst that can happen is that we do not gain any insights.
  - **Organizational risks**: The goal of the project may become irrelevant, or upper management may lose interest in it (R1).
  - **Financial risks**: Possibly fueled by R1, all depending on preliminary data mining results and their reception, insufficient funding from the get-go is a risk (R2).
  - **Technical risks**: To not have the necessary tools to develop the project ready from the start could hinder production and push deadlines back (R3). We do not consider lack of computing power as a risk.
  - **Data/data sources risks**: although we are assuming the data is rich and useful, poor quality (as in exhaustive pre-process needed) and coverage or lack of enough data for the results to be significant are risks that we consider (R4).
  - **SARS COVID-19:** due to the current situation, it is a remarkable risk that any team member could fall ill (R5).

Contingency Plan (overview)

Here are laid out the main actions that are to be undertaken to mitigate, or even avoid at all, the risks. These actions are for all 5 risks, but in a more serious scenario these would be laid out according to each risk RPN. On that note follow the contingencies.

Risk nº	Contingency action/s
R1	The scope of the project, initial documentation and business questions all must be clearly laid out and firmly understood and approved by upper management before the initial date.
R2	A detailed, broken down funding plan needs to be developed and approved before starting. A prediction of resource usage needs to be conducted.
R3	Again boiling down to planning, an initial overview of the methods and techniques that are going to be used will give us an idea of the tools needed. Further consulting with related departments could also be interesting.
R4	As it is our only source of data, no real actions can be taken to be sure that our data is good to go. But, in a real scenario, other sources would probably have to be collected as backup, and conducting preliminary exploratory analyses would give a nice idea of what we are working with.
R5	- Above all, follow the instructions of the local health authorities. - As presential work is restricted, everyone will stay at home and work remotely most of the time, following the instructions of the project manager. - Weekly and daily meetings will be performed virtually via Google Meet. - Face masks will be provided to each team member. - All used material is to be thoroughly disinfected daily.

- **Terminology**. A glossary of terminology relevant to this project can be found in this section.
    - **Business terminology**: Glossary of the general business understanding for the project.

      → Technical debt, developer profile, minimize bugs, complexities, tailor resources, competences, improve abilities, efficiency, improve abilities, success criteria.
    - **Data mining terminology**: glossary of data mining terminology relevant to the business problem.

→ Technical debt dataset, SQL database, data mining, data exploration, data quality, statistics, distributions, correlation, modelling techniques, segmentation, classification, semi-supervised algorithms, training, testing, evaluation of results, deployment, monitoring, maintenance, Python, clustering methods, neural networks, visualization techniques, PCA.

- **Costs and benefits**
    - **Costs for data collection**: All the data we need is available in The Technical Debt Dataset. Therefore, there are no costs associated with collecting the data. In a realistic scenario, we would have to collect the necessary data from all the developers of the company for each task of each of the projects in which they have worked. It should also be taken into account that these costs can multiply if the data has to be re-extracted for different reasons.
    - **Costs of the solution**: In the case of the development and implementation of the solution, the costs are mainly related to the work of the developers in a project of the company itself. Therefore, we mainly consider those that have to do with developers if there is no change in the workflow.
    - **Benefits**: As we have explained previously, it will allow future projects to be carried out more efficiently and quickly, and with better quality. Therefore, we will increase customer satisfaction and it allows us to do more projects than before.
    - **Operating costs**: These costs are related to the day-to-day of the project. In this category would be the salaries of the workers and the possible rent of cloud (if necessary).

1.3. Data mining goals and success criteria

- **Data mining goals**
    - *What are the types of developer profiles in our business?* → Developers must be segmented to create the different profiles based on some attributes selected from the data. So it is a segmentation problem.
    - *Which are the main bugs that each developer's profile tends to produce?* → For each of the profiles found above, we look for the most frequent type of bugs. Therefore, it consists of a concept description problem.

- **Data mining success criteria**
    - **Model assessment**: In our case, it is not possible to have an accuracy of whether we have segmented a developer well or not as it is a non-supervised problem.
    - **Benchmarks for evaluation criteria**: we can use the variance between and within profiles to verify if the commits of a group are sufficiently homogeneous with each other or not and, at the same time, if they are different with respect to those of another profile.
    - **Subjective assessment criteria**: a valid criterion would be to take into account if the separation of the groups is clear, that is, if different clearly visible profiles are formed.

1.4. Project plan

- **Project Plan.** In this section we present the different stages that will be executed in the project in order to achieve the data mining goals and the business goals:

Stage 1: Collect the data	**Duration:** 1 day
Resources: None	**Dependencies:** None
Inputs: None	**Outputs:** SQL database or csv files
Description: Collect all the data necessary to carry out the project and gather the information in a dataset.	

Stage 2: Description, exploration and quality verification of the data	**Duration:** 3 days
Resources: Python	**Dependencies:** Stage 1
Inputs: Dataset	**Outputs:** New insights from the data
Description: Describe information of the attributes (like its type and range), compute basic statistics of each one of them, analyze their distribution and look for correlations. Moreover, find possible errors and missing values.	

Stage 3: Data preparation	**Duration:** 7 days
Resources: Python	**Dependencies:** Stage 1 & 2
Inputs: Dataset	**Outputs:** Dataset in optimal conditions

Description: Select the necessary data, clean it (so that there are no errors, missing values or noise), construct it, integrate it and format it to be able to work with the resulting dataset.

Stage 4: Modeling Techniques and Test Design	**Duration:** 5 days
Resources: None	**Dependencies:** None
Inputs: Dataset	**Outputs:** Documentation
Description: Select the techniques that will be used to segment the data taking into account the assumptions of each approach. In addition, decide how to evaluate the models and prepare the data for testing.	

Stage 5: Modeling	**Duration:** 5 days
Resources: Python	**Dependencies:** Stage 3 & 4
Inputs: Dataset	**Outputs:** Model
Description: Initialize the model and run it on the dataset, and evaluate the results obtained with the criteria defined above.	

Stage 6: Evaluation	**Duration:** 2 days
Resources: Python	**Dependencies:** Stage 5
Inputs: Model	**Outputs:** Approved models
Description: Check the results obtained regarding the business success criteria and the objectives.	

Stage 7: Review and next steps	**Duration:** 2 days
Resources: None	**Dependencies:** Stage 6
Inputs: Evaluated models	**Outputs:** Next steps
Description: Summarize the process review, overview of the data mining process and review the results, and refine the process plan.	

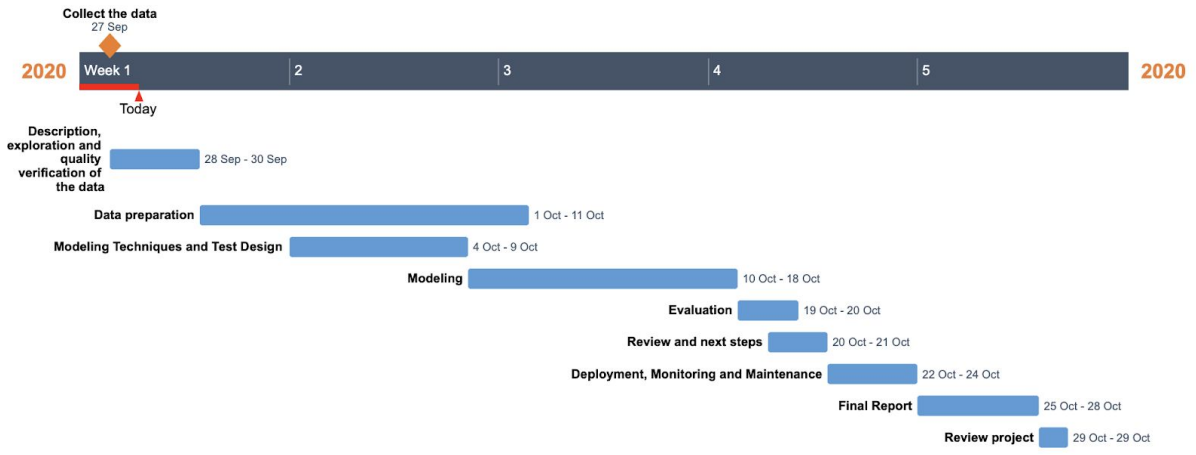
Stage 8: Deployment, Monitoring	**Duration:** 2 days

and Maintenance	
Resources: None	**Dependencies:** Stage 7
Inputs: Models	**Outputs:** Documentation
Description: Define how the product will be used by users and possible future changes, so that the software developed would be upgradeable.	

Stage 9: Final Report	**Duration:** 3 days
Resources: None	**Dependencies:** Stage 8
Inputs: None	**Outputs:** Final Report
Description: Management summary, findings, models' explanation, data mining goals achieved and possible recommendations for future work.	

Stage 10: Review project	**Duration:** 1 day
Resources: None	**Dependencies:** Stage 9
Inputs: Project	**Outputs:** Review
Description: Obtain feedback, analyze the process (things to improve, things that have gone well, etc.) and document the specific data mining process.	

Next, we show the Gantt chart that we obtained with this project plan.



- **Initial assessment of tools and techniques**
    - **Criteria for tools and techniques**: The tools and techniques that we want to use must be complete and general, that is, they must be used

for any data set that contains, for example, different types of attributes (int, bool, vector, etc.).

- **Potential tools and techniques**: The tool that can help us the most in this project is Python. To do the segmentation, we can use different clustering methods, neural networks and visualization techniques to differentiate the developer profiles.
- **Appropriateness of techniques**: Through clustering we assign profiles taking into account a representative of each of the groups and the distance between them. Regarding neural networks, they can discover relationships between attributes that we cannot see, that are invisible to us. Finally, the visualization techniques can give us an idea, but they are limited to a 3D space, that is, 3 variables, so we would have to reduce dimensions using, for example, PCA.

2. Data understanding

2.1. Initial data collection

In this project we will be using The Technical Debt Dataset, a set of project measurement data from 33 Java projects. This data is obtained using different tools to analyze each commit: *PyDriller* for mining Git repositories and extracting information (like the commit message, modifications or the source code of the commit); *Refactoring Miner* to collect information on the refactoring applied in each commit; *SonarQube* to collect Technical Debt information; *Ptidej* to detect code smells and antipatterns; and last but not least, the *Jira issue tracker* to obtain fault information about detected issues. After that, an implementation of the SZZ algorithm is executed for identifying the fault-inducing commits. This dataset enables data scientists to work on a common set of data and thus compare their results.

Therefore, all the information needed for this project can be obtained from the previously mentioned dataset (The Technical Debt Dataset) which is available to download at this link. This dataset is composed of 9 csv files with information about Git Commits, Jira Issues, SonarQube Issues, SonarQube Rules and Refactoring Mining (a list of refactoring activities applied in the repositories) among others.

Especially, for our first goal (determinate different developer's profiles) we will be interested in attributes that can somehow characterize a developer, while for our second goal (find the most frequent bugs made by each profile) we will focus more on issues information. Based on this criteria, in the next subsection we will make a preselection of the attributes.

Moreover, based on these goals, we can define general criteria to select the attributes that would make possible for us to achieve them. For the first one, that is, to determine the profile of the developers, we think that we should focus on their experience and code quality. For example, we think that it is important to gain information about how long a certain developer works in a project, its number of commits, the number of projects it is involved in, if it fixes issues usually, etc. We are also interested in the characteristics of its codes so that we could try to extrapolate them and, then, define the developer's profiles with more precision. For the second goal, the one in which we try to find the most frequent bugs of each profile, we need all the attributes related to the type of issues and their importance. Moreover, we can be interested in how much it costs to fix each type of problem to know more about the gravity of the issues made by each profile.

To understand each of The Technical Debt Dataset's attributes and determine if they are useful for our project, we will be consulting the paper *Lenarduzzi, V., Saarimäki, N., & Taibi, D. (2019, September). The technical debt dataset.* (which explains some of the attributes) but also specific web pages of SonarQube, Jira and GitHub; like the

SonarQube web page (which has the definitions of their metrics, some of which are included in the SONAR tables of The Technical Debt Dataset).

2.2. Data description

Based on the information gathered from the above-said sources, we next explain which attributes are considered more important on each table from The Technical Debt Dataset and, so, the attributes that will be studied in more detail. The analyses of the tables can be found in the annex section and in different Colab Notebooks.

- **PROJECTS**: This table contains the links to the GitHub repository and the Jira issue tracker of each project. The attribute of this table that we need is basically the key (`projectID`) in order to relate all the other tables. This table has 33 records, the number of projects analysed in this dataset. This is the link to the notebook.

- **JIRA_ISSUES**: This table contains the Jira issues for the 33 analysed projects. As in the previous one, we select the key attribute to relate it with the other tables (`projectID`), the creation and resolution date of the issue (to estimate how long it tooks to solve the issue), its `type`, `priority` and the number of `votes` (to know the importance of the detected issue); and, finally, the `assignee` and the `reporter` of the issue, that is, the account that is working on the JIRA issue and the person who created it, respectively. It has 67.428 records. In the notebook we can find the analysis of the selected attributes.

- **SONAR_MEASURES**: It contains the different measures SonarQube analyses from the commits. We need the primary keys of the table (`projectID` and `commitHash`), the number of classes, files and functions; the density of lines that are commented (as we think that it is related to possible explanations of the code, so a more commented code is better explained) and duplicated (we hypothesise that a code with a lot of duplicated lines is worse), and the number of effective lines of code. Moreover, we select all the attributes associated with the code `complexity`, the `violations` (the number of times that the coding rules are not met), the `bugs` and the code smells; information of ratings related to the maintainability, reliability and security issues, and the effort to fix all code smells, bugs and vulnerability issues. This table has 55.629 records. In this notebook we can find the analysis of the mentioned attributes.

- **SONAR_ISSUES**: This table has 1.941.508 records and it lists all of the SonarQube issues as well as the anti-patterns and code smells detected. As in the previous tables, we select the key attributes (`projectID`, `creationCommitHash` and `closeCommitHash`, these last two to join with

SONAR_MEASURES and know the characteristics of the commit that solved the issue), the creation and closing dates of the issue (to estimate the duration of the SonarQube issue), its `type` (bug, vulnerability or code smell), the severity of the issue and the time to resolve the issue (`debt` attribute). Also, we take the `author` to identify each developer. In this notebook, we have made a brief analysis of the selected attributes.

- **SONAR_RULES**: It lists the rules monitored by SonarQube, so it provides a short description for each rule. Therefore, it explains the rules that SonarQube uses to find issues. In our case, we only care about the type of issues generated, not what was used to detect them. For that reason we do not use any attribute of this table.

- **GIT_COMMITS**: This table reports the commit information retrieved from the git log. As in the previous tables, we select the key attributes (`projectID` and `commitHash`), the `committer` and the `commiterDate` (to find the frequency of commits that a developer does) and the attribute `inMainBranch` (to see if the developer introduced a bug in the main branch). We are also considering the attribute `author`, which could be the leader of the project. It has 140.687 records. In the notebook we can find the analysis of the selected attributes.

- **SZZ_FAULT_INDUCING_COMMITS**: This table has 27.340 records and reports the results from the execution of the SZZ algorithm, which labels the fault-inducing and -fixing commits. As in the previous tables, we select the attribute `projectID` in order to be able to join the tables in the future. We also consider the attributes `faultFixingCommitHash` and `faultInducingCommitHash` that will let us relate this table with the others that have the attribute `commitHash`. The last attribute that we will take from this table is `faultFixingfileChanged` as it informs us about the file that has been changed in the commit. In this notebook we can find its analysis.

- **GIT_COMMITS_CHANGES**: This table contains the changes performed in each commit. As in the previous tables, we select the attributes to join this table with the others (`projectID` and `commitHash`), the type of the change that has been done in the commit (added, deleted, modifies or renamed) and some metrics about the changes (like the number of lines added and removed, the complexity or the number of tokens). This table has 891.711 records. In this notebook, we have made a brief analysis of the selected attributes.

- **REFACTORING_MINER**: This table reports the list of refactoring activities applied in the repositories. As in the previous tables, we select the attributes to join this table with the others (`projectID` and `commitHash`) and also the

type of refactoring that has been applied to the code. It has 57.530 records. In the notebook we can find the analysis of the mentioned attributes.

2.3. Data exploration report

In this section, we will tackle the data mining questions that can be addressed using simple operations as querying, visualization and reporting techniques. You can find in this notebook the analysis done. Our first data mining goal is:

1. ***What are the types of developer profiles in our business?***

To get a first approximation to this objective, we need to start gathering knowledge on which developer profiles can exist and how the different attributes on data can define them according to their characteristics, properties, patterns…

First, we need to make some hypotheses about which attributes can discriminate between profiles and how they must be (which value they must take) in order to represent their particular features. Below are explained the most important obtained results.

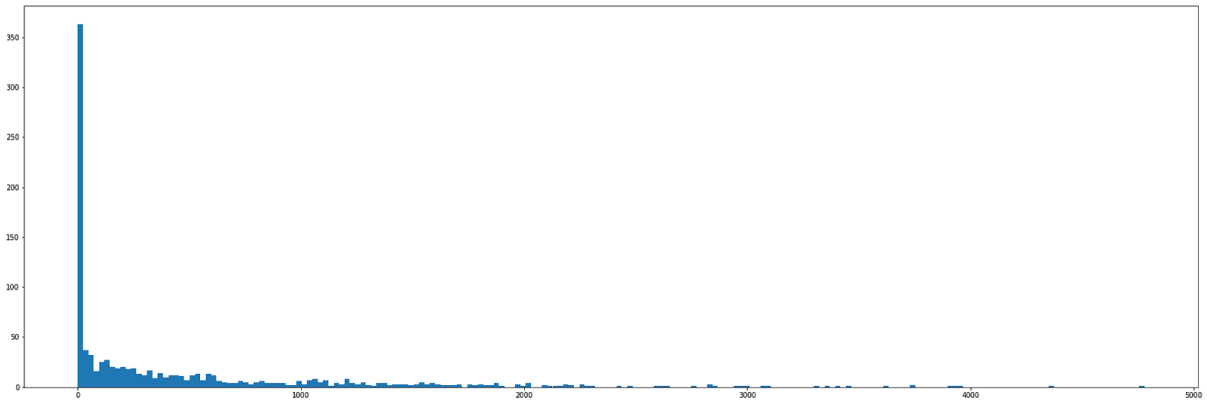
- *Hypothesis 1: We can divide each developer profile according to how long on average they were in the different projects in which they participated.*

We have used data stored in the table GIT_COMMIT, as it contains information about commits performed by each developer. However, we don't have any developer field so we are going to assume that the field ***committer*** is its equivalent. For each *committer*, we have computed the time in days they have spent on each project in which he at least made **one commit**. This period of time is approximately computed as the **difference in days** between the first and last commit he made in the project. Finally, to have one single value for each developer, we have performed the average of days between the different projects. This is equivalent to knowing on average how many days each developer spends working in one project.

The resulting data from the previously described operations can be interpreted as a distribution of probability. Hereby, we can plot them as an histogram of 200 bins. The *x*-axis represents the average quantity of days of developers working in a project and the *y*-axis the quantity of developers that spend the same quantity of days. In it we can see three well-defined behaviors.

- Developers who have been very few days working in the project, even 0 days (273 people). They are represented by the first long and concentrated bin that takes between 0 and 24 average work time per developer. They can be considered as contributors to the project but not developers.
- Developers who work on projects for a reasonable period of time in order to develop a good enough final product and not overwhelm it. They are represented by medium high bins that take a moderate range of days.

- Developers who work for lots of days. They are represented by a very short and wide range of days distributed.



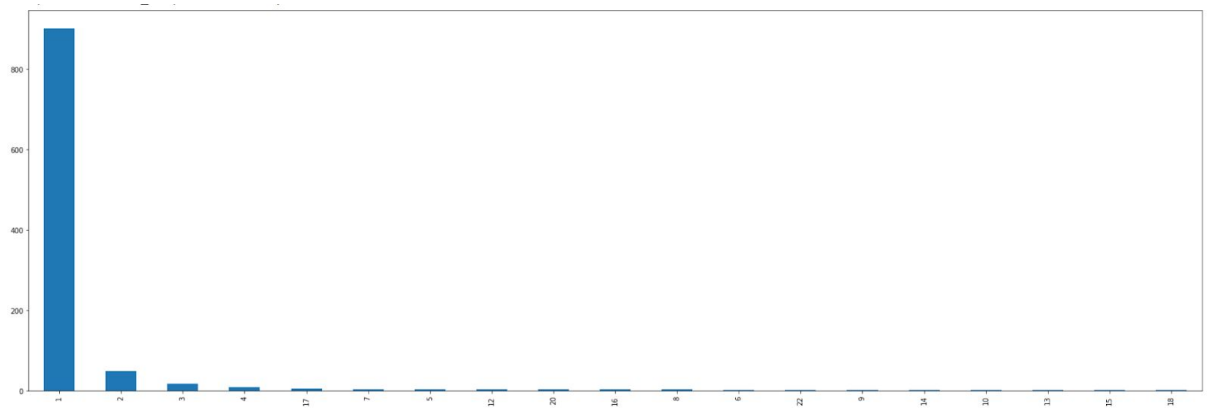
We can also extract similar groups from the histogram using percentiles as, for example 10%, 70% and 90% percentile, respectively.

As a conclusion, this attribute of years working in projects can be very useful to discriminate between long term workers or prompt workers. However, we can not use it alone. We need extra information to solve possible deficiencies such as developers that just make two commits far apart in time (we will consider that all this time he has been working on the project.

- *Hypothesis 2: We can divide each developer profile according to the quantity of projects he has worked on.*

Again we have used data stored in the table GIT_COMMIT. For each *committer* (equivalent to developer), we have computed the number of different projects he/she has worked on. This is done by counting the number of different ***projectID*** on which the developer has made at least one commit.

These results can not be expressed as a distribution as they consist of categorical data. For that reason, we have plotted a bar chart. In this representation, we can see that there is a bar that excels from the rest. It is the bar corresponding to one single project. In this particular case, 901 developers have just worked on one project. Both the second and the third bar have a relatively remarkable density. However, all the rest of users are equally distributed until two developers that worked on 22 different projects.



That way, developers can again be assign to three different profiles:

- Developers that just worked on one single project: 901 developers belong to this group.
- Developers who worked in a moderate number of projects (2-3): 66 developers to the intermediate one.
- Developers that participate in lots of projects (> 3): 49 to the third one.

These groups are very unbalanced. However, these results make a lot of sense since the boss developer might work in lots of different projects but with small contributions while the vast majority of developers work in a small number of projects and tend to only have visibility on what he/she or his/her close colleagues are developing. We could get more valuable information if we combine it with other attributes, like the one before.

In addition we have tried to classify the groups obtained with the previous hypothesis in this new scenario.

- **Developers who have been very few days working in the project**: we obtain a mean value of 1.56, that is, on average all developers in this group tend to work in one or two projects. This is contrary to what we might expect as developers that make small contributions to projects are expected to be working on several projects at the same time.
- **Developers who work on projects for a reasonable period**: their average of projects is 2.1. This means that this group tends to work in a bit more projects than the previous one.
- **Developers who work for lots of days.** As we may expect, this type of developers mainly focus on a small number of projects and spend a lot of time working on them. Their mean value is 1.58, more or less similar to the first group.

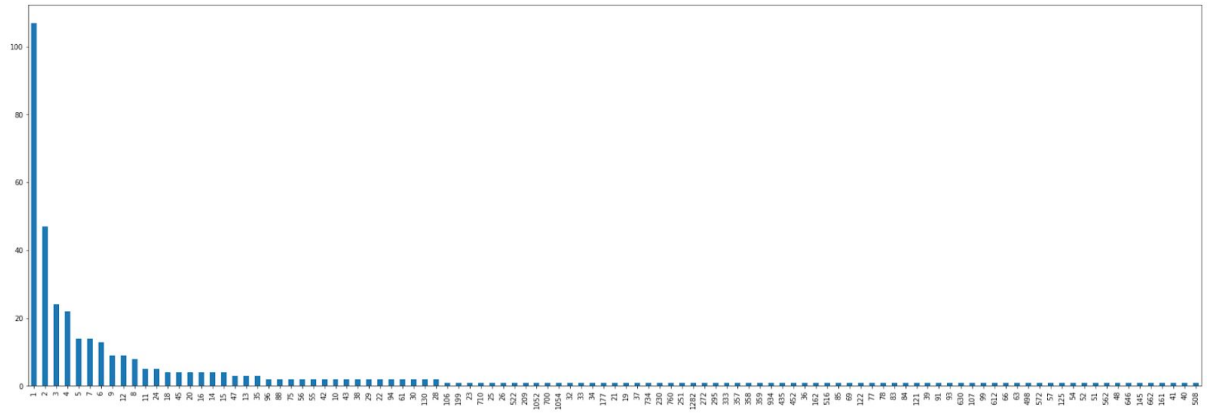
With this final analysis, we can conclude that the different groups that we detect on both hypotheses are not the same. The first and the third one from the previous section shared similar characteristics if we look at them in the perspective of projects working on while the second has a slight difference.

Our analysis is not very accurate as almost 90% of developers just work on one single project. This way, the mean of each group has a strong bias towards one. In order to create a more accurate one we might look case by case in order to get an idea of what changes are happening.

- *Hypothesis 3*: *We can divide each developer profile according to the quantity of bugs they tend to fix.*

This time, we have used data stored in the table JIRA_ISSUES, as it contains information about reported jira issues and the user who solved the bug. This person is called ***assignee*** and we are going to consider that he/she is also a developer. For each *assignee*, we have computed the different issues (different creationDate) for which he/she has fixed the bug of himself or another developer, called the *reporter*. We must take into account that the resultant developers are going to be the one that made at least one bug fixing.

Next, we plot a bar chart with the result data. We can clearly appreciate that the majority of developers are concentrated in the range of bugs fixed between 1 and 9. A small part of them are spread through values higher than 10. This way, we have identified two different developer behaviours. However, we must take into account another important group: the ones that do not fix any bug. This quantity is quite higher than the quantity of developers that fixed just 1 bug.



Again we are trying to relate the different groups of developers obtained with the previous hypotheses.

- **Developers who have been very few days working in the project**: we obtain a mean value of 65.21, that is, on average all developers in this group tend to fix 65 bugs.
- **Developers who work on projects for a reasonable period**: their average fixing bugs value is 185.4, that is more than double the value of the previous group.
- **Developers who work for lots of days**: their mean value is 323.5 of fixed bugs.

We must comment that this result makes a lot of sense, since the more days you work on a project, the more likely you are to make bugs and have to solve them. For that reason, we can say that hypothesis 1 and hypothesis 3 can be complementary and be used for the same purpose.

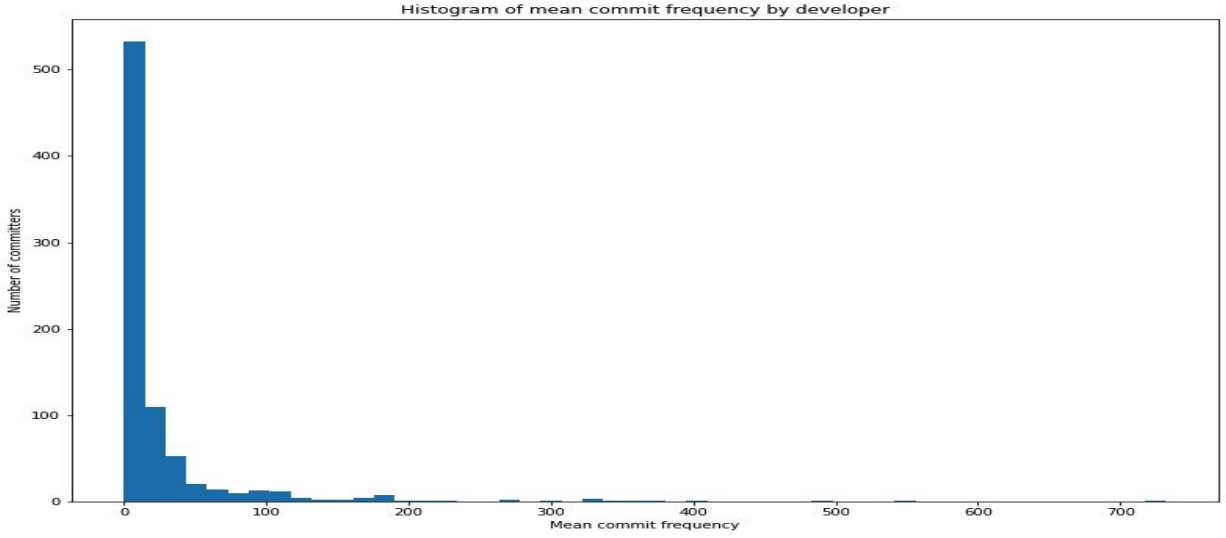
- **Developers that just worked on one single project**: in average they are used to fix 134.3 bugs.
- **Developers who worked in a moderate number of projects (2-3):** we are not able to find any case where these developers solve any bugs.
- **Developers that participate in lots of projects (> 3)**: their mean value is 191.5, which is quite superior to the first case but more or less similar.

For this third hypothesis we are not able to see any pattern on the results.

- *Hypothesis 4*: *We can divide each developer profile according to the frequency of commits.*

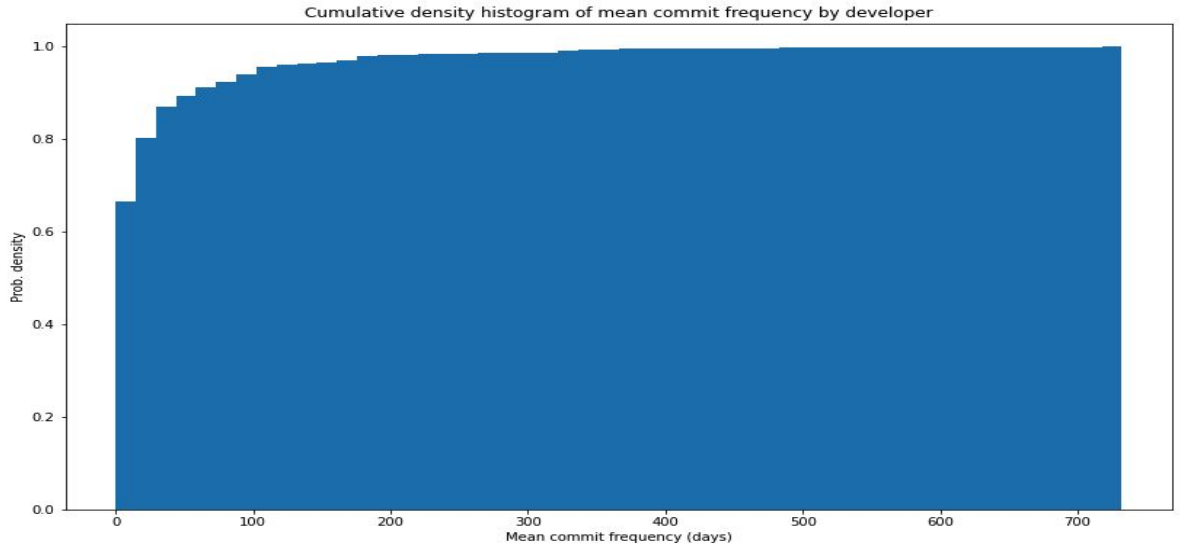
We have used data stored in the table GIT_COMMIT, as it contains information about commits performed by each developer. Again we are going to use ***committer*** as a synonym for developer. For each *committer*, we have to compute the total number of commits he/she has done. Then, for the developers with more than one commit, we compute the difference of time between the timestamps of sorted commits in seconds. In order to have just one value for each committer, the mean value of time between commits is performed. This way, we obtain the average time each developer takes in making a new commit. Finally, we just change from seconds to days in order to have a more meaningful value.

As the resulting data consist of continuous values, we can again plot its histogram so we can observe what type of distribution it follows.



We clearly see that the majority of developers commit at least once every two weeks (about 70% of density). Another 20% of density approximately belongs to developers that commit between once every 2 weeks and once every 3 months (approximately).

The 10% left of density belongs to developers that very rarely commit (> 3 months). Thus, we think that this measure can be a good feature to observe developer characteristics by. Following is the plot showing cumulative density of the previous histogram and from where one can appreciate the mentioned approximate populations.



As we point out at the beginning of the section, we have just mentioned the most important results. Other significant hypotheses we have made and that can be used in the project are:

- Take a look at how many and which type of refactoring is performed by each developer.
- If when doing refactoring the developer is prone to introduce bugs.
- Take a look at which type of bugs has each developer made more.
- Take into account the complexity of the code for each developer.

Our second data mining goal is:

> 2. ***Which are the main bugs that each developer's profile tends to produce?***

We have already seen that if we take a look at individual features it is very difficult to find relevant patterns that allow us to divide developers into clear profiles. For this reason it is very difficult to classify users according to what we have just observed, as they do not show extremely clear characteristics.

This way, we can not deal with this data in simple ways in this section and we must rely on clustering algorithms so they can extract sufficient relevant patterns in the combination of different attributes and not just one of them as we have been analyzing.

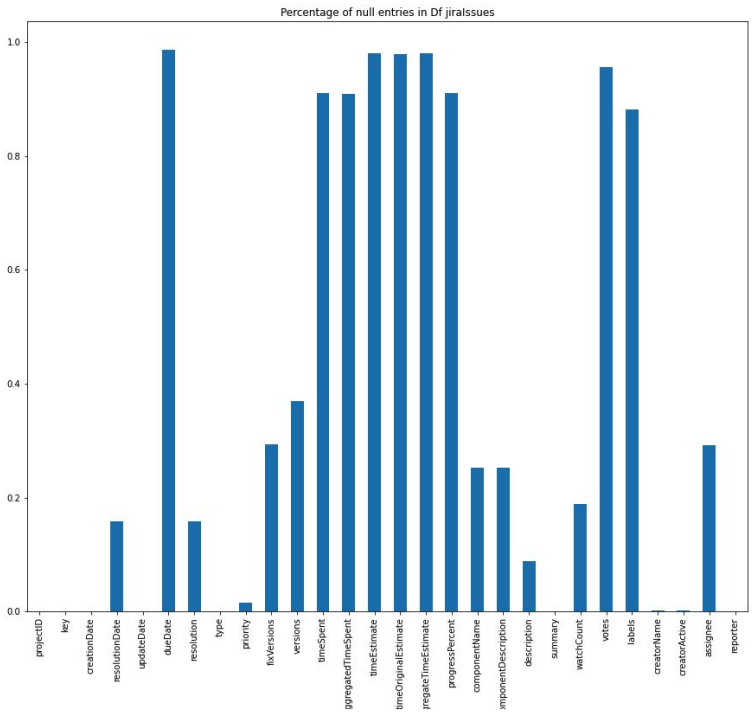
2.4. Data quality

In this section, we analyze the state of the data in terms of quality. Overall, the task here is to check if 'everything is as it's supposed to be'. But, this task is too broad, so we focus the scope on some smaller, clearer tasks. Before beginning, we would like to clarify that this data comes from only 1 source, and has already been processed, a fact that is very helpful in the preparation and understanding phases. You can find the results in this notebook. So, here are the different quality checks performed over the data.

- **Keys**
  Keys are consistent and there aren't any clashes or inconsistencies (manual check).

- **Fittage of attribute meaning and values**
  All data frames contain consistent attributes in terms of their fittage to the significance of the values they contain (manual check).

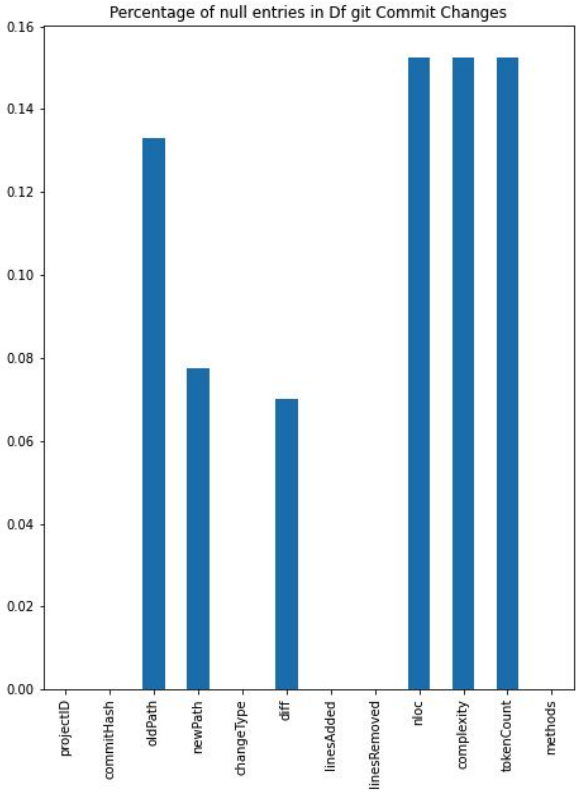
- **Missing attributes**
  The task here has consisted in looking at the percentage of null entries of each particular feature in a dataframe. Then, a manual check has been performed in all cases that were considered serious enough (number of cases or/and no logical explanation).

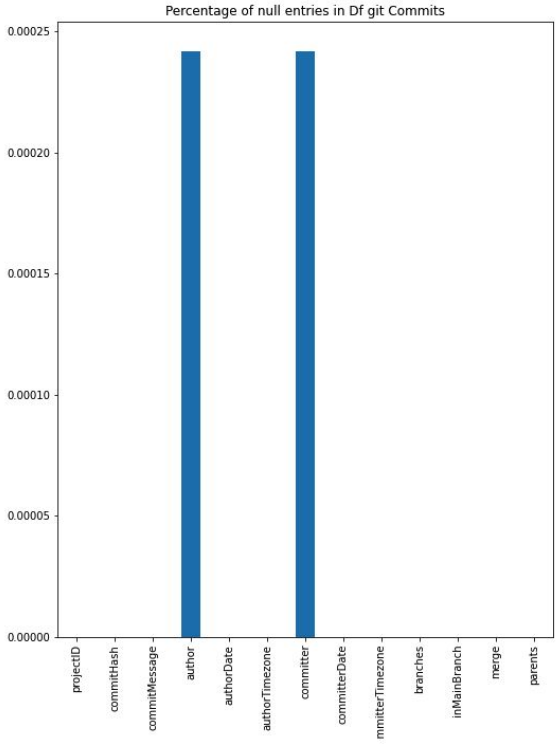


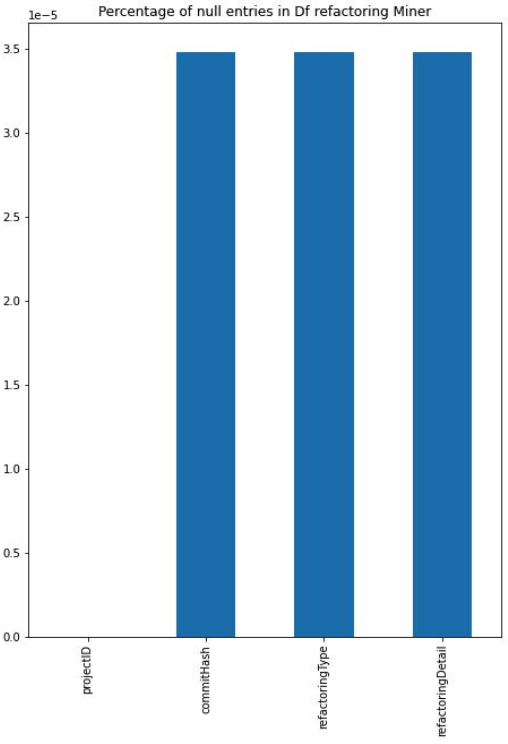
In the Jira Issues dataframe there are quite a lot of missing values. But, taking a close look at the features with more missing values, it is only natural that they are that way. Many times, when a developer reports a Jira Issue, things such as the time estimates, the due date are not explicitly defined. That way, the majority of the issues still have information, but lack features that are self-reported. On the other hand, the 'assignee' feature has missing values (about 30%), which is a bad

sign, because it is important for our profile identification (issues of an assignee and their characteristics). We can either simply ignore those issues or extract their information without using the assignee.

Looking at the changes that commits introduced, there are some NA's. `nloc`, `complexity` and `tokenCount` are about 15% NA's, which is not good, especially `complexity` as it somehow tells us the probability of a commit introducing faults. Still, we have lots of data, and 15%, although big, is not a game-changer. The other variables containing NULLS are not relevant to our application.



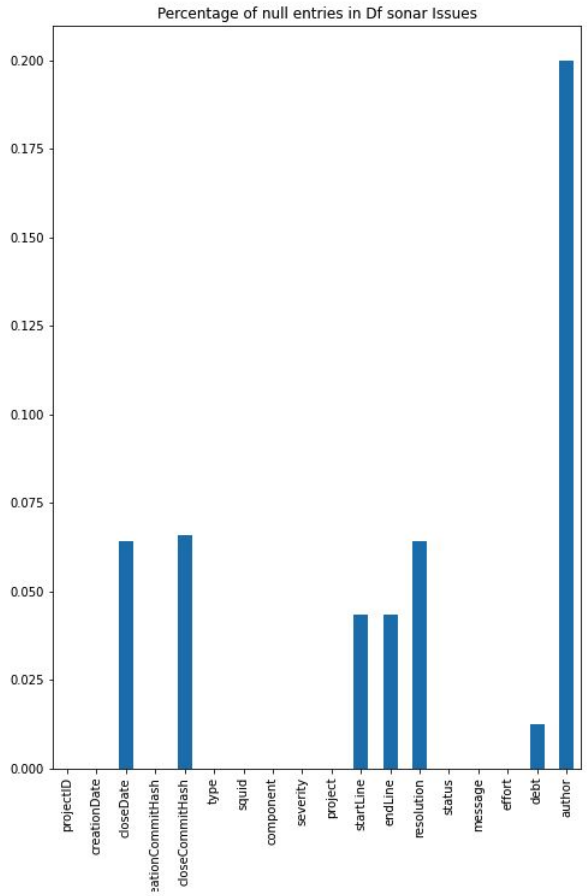


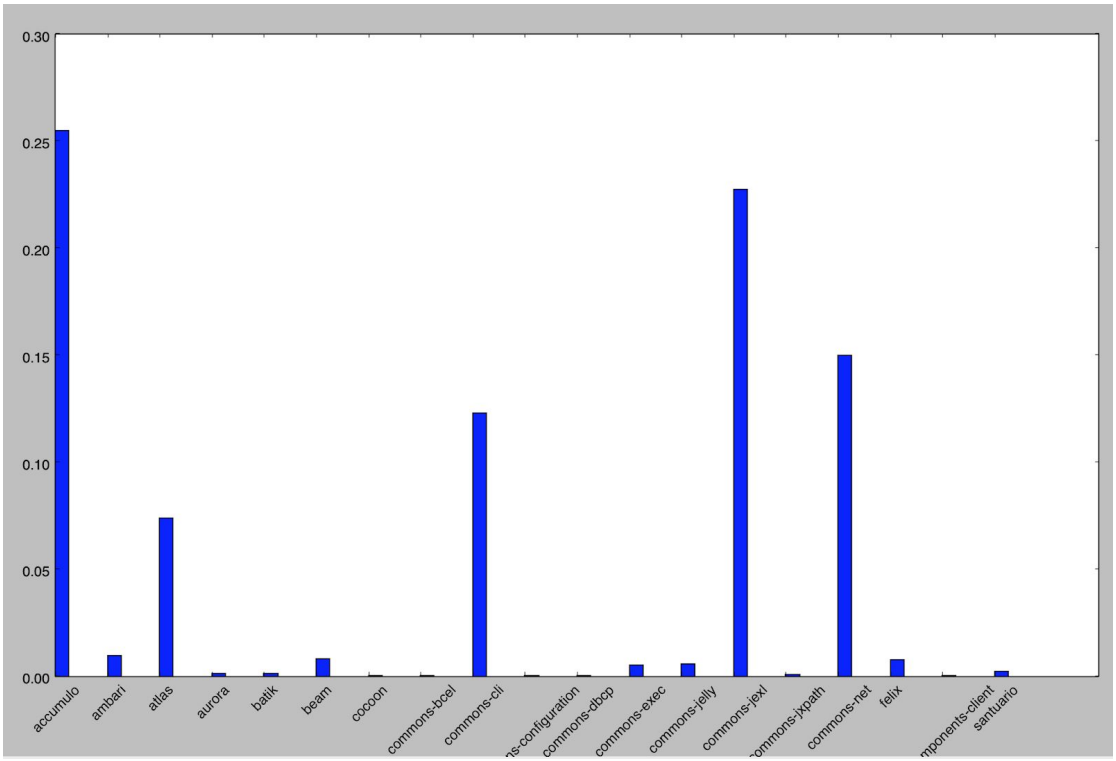


In the data frame containing git commits, while it is odd that some commits lack author and commiter, again, their significance is simply not high enough for it to be considered a problem. No comments on the cause, as it can be due to many things (but probably not human-related).

For the refactoring miner data frame, again the density of NA's is simply not enough for causing any issues (vertical axis is e-5).

Moving on to Sonar Issues, it seems that neither the closing date nor the hash of the closing commit has been found in 7.5% of cases. The starting line and the ending line haven't been identified either in a smaller 5% of cases. Resolution is missing as many times as the has of the closing commit, which tells us that the issue could be related between the two. The debt of the issue hasn't been able to be identified in about 1% of entries, and more significantly, almost 20% of authors are missing. Again, this is quite important and will have an impact on our analysis somehow. But, having in mind that we have quite a lot of data, we can either omit those rows or use them for other general purpose analyses.





Looking at the percentage of null authors in the data frame Sonar Issues by project, we see that that 4 projects contain the majority of the missing entries. So, we will not be able to analyse some values of these four projects. Still, we will explore without having the author identified, as it can help us reach conclusions either way.

Dataframe ‘Sonar Rules’ has the same null distribution as ‘Sonar Issues’. ‘Sonar Measures’ and ‘SZZ Fault Inducing Commits’ have very small amounts of NULLS to even begin to consider them problematic.

Note that we have not observed any patterns in the appearance of null values over the data frames to state or consider that there has been some underlying issue with the collection process.

- **Similarity of attributes with different meanings**
  We have not noticed any set of attributes that hold very similar meanings although they are supposed to encode different information. Of course, there are similar values, but they still hold distinct meanings, even if only slightly (manual check).

- **Spellings & format**
  The only issue we have found regarding the spelling or format of our data are:
  1) Messages such as the ones in the commits have encoding errors.
  2) Features in the table ‘SONAR_RULES’ follow a different naming convention than the other tables.

  Fixing the encoding errors should be performed if any kind of meaning extraction is to be performed on the messages, or any kind of parsing looking

for commits or issues that contain certain keywords. In order to do it, using task-specific software is required (manual check).

- **Deviations**
  Deviations have been described in the data description section.

- **Value plausibility**
  There are not any value plausibility issues (manual check).

- **Format consistency**
  There are not any format consistency issues, all features contain the correct (desired) types (manual check).

- **File type check**
  As we are using the CSV format for this initial data exploration and understanding phase, we need to check their consistency. In all files, the value separator is ‘,’, without exception. The format of the feature names in all data frames is camelCase, except in the table ‘SONAR_RULES’, that uses snake_case (as discussed in **Spellings & Format**). A simple rename when performing analysis over this table is enough to make things consistent, then (manual check).

The conclusion of this section is that the data is quite clean. This is a fact that we expected given where it comes from and the fact that it has been processed. It really is an exception to find such properties from data in the wild. It is a matter of the attributes one wants to use to perform their analyses which will dictate how one deals with the issues discussed in this section.

3. Data preparation

3.1. Select data

Taking into account the results obtained from the Data Understanding section (mainly the Data Description and the Data Quality subsections) we have made the final selection of the attributes of each table.

JIRA_ISSUES

In the previous section we have seen that some of the preselected attributes have missing values, but we can consider that there are not enough NAN's or that we can resolve these issues as we will see in the next section. Therefore, we select all these attributes, that is, we will use the following columns: `projectID`, `key`, `creationDate`, `resolutionDate`, `type`, `priority`, `assignee` and `reporter`.
In this link we can find the notebook that makes this final selection of the attributes of the JIRA_ISSUES table.

SONAR_MEASURES

In this case, as we have already mentioned, the number of missing values is very small and, therefore, irrelevant. Therefore, there is no problem with the parsed columns. So the attributes finally chosen from this table are the following ones: `commitHash`, `projectID`, `functions`, `commentLinesDensity`, `complexity`, `functionComplexity`, `duplicatedLinesDensity`, `violations`, `blockerViolations`, `criticalViolations`, `infoViolations`, `majorViolations`, `minorViolations`, `codeSmells`, `bugs`, `vulnerabilities`, `cognitiveComplexity`, `ncloc`, `sqaleIndex`, `sqaleDebtRatio`, `reliabilityRemediationEffort` and `securityRemediationEffort`.
In this link we can find the notebook that makes this final selection of the attributes of the SONAR_MEASURES table.

SONAR_ISSUES

From the previous section we have seen that any of the initially selected attributes of this table have enough missing values in order to not consider them and the categorical attributes have their categories sufficiently represented, so we will keep all the initially selected attributes: `projectID`, `creationDate`, `closeDate`, `creationCommitHash`, `closeCommitHash`, `type`, `severity`, `debt` and `author`.
In this link we can find the notebook that makes this final selection of the attributes of the SONAR_ISSUES table.

GIT_COMMITS

Regarding the GIT_COMMITS table, in the Data Description analysis we saw that the attribute `inMainBranch` had the value "True" in all the records, so the attribute does not give us any information and can be removed. Therefore, the finally selected attributes of this table are the following: `projectID`, `commitHash`, `author`, `committer` and `committerDate`.
In this link we can find the notebook that makes this final selection of the attributes of the GIT_COMMITS table.

SZZ_FAULT_INDUCING_COMMITS

In the Data Description subsection we saw that, although any of the attributes have missing values, the `faultFixingfileChanged` did not seem to give us any important information considering our mining goals so we have decided to not consider it. Therefore, the finally selected attributes of this table are the following ones: `projectID`, `key`, `faultFixingCommitHash` and `faultInducingCommitHash`.
In this link we can find the notebook that makes this final selection of the attributes of the SZZ_FAULT_INDUCING_COMMITS table.

GIT_COMMITS_CHANGES

In the Data Understanding section we saw that some of the initially selected attributes had a significant number of missing values. In addition, some of these attributes did not seem to give us any important information considering our mining goals or the information could be taken from other attributes of other tables, so we have decided to not consider them. Therefore, the finally selected attributes of this table are the following: `projectID`, `commitHash`, `changeType`, `linesAdded` and `linesRemoved`.
In this link we can find the notebook that makes this final selection of the attributes of the GIT_COMMITS_CHANGES table.

REFACTORING_MINER

From the previous section we have seen that any of the initially selected attributes of this table have enough missing values in order to not consider them and the categorical attributes have their categories sufficiently represented, so we will keep all the initially selected attributes: `projectID`, `commitHash` and `refactoringType`.
In this link we can find the notebook that makes this final selection of the attributes of the REFACTORING_MINER table.

3.2. Clean data

In this subsection of the Data Preparation we mainly deal with the missing values, either erasing the records that contain one or more (in each table), either relabeling them with another value.

JIRA_ISSUES

The only attributes from this table that have missing values are `resolutionDate`, `priority` and `assignee`. In the case of `priority`, we can not obtain this information from anywhere else and, since only 1.6% of the records have this missing value, we can erase these records without causing any problem. Regarding the `resolutionDate`, a missing in it means that the issue has not been resolved yet so the null values are giving us information. Therefore, we can relabel them as "not-resolved", but it is inconvenient in next steps as we cannot aggregate this value with integer ones. It is for that reason that we modify these missing values by the timestamp of the last commit of the project. In the case of the `assignee` attribute, the missing values mean that the issue has not been assigned to any developer, so we can keep these records by relabeling the 'NaN's as "not-assigned".
The cleaning of this table can be found in this notebook.

SONAR_MEASURES

The finally selected attributes of this table have no missing values so they do not need to be cleaned.
The link to the notebook is this one.

SONAR_ISSUES

In this case, we have four attributes with some missing values: `closeDate`, `closeCommitHash`, `debt` and `author`. In the first case, these values represent the 6.4% of the rows, but it means that the issue has not been resolved yet, so we change the value to the timestamp of the last commit of the project. In the second one, the NaN's represent 6.6% of the total Sonar issues. We can think that they do not have values because the issue still exists; so we change these values to 'not-resolved'. Moreover, we see that all the rows that have the attribute `closeDate` with missing value have also this type of value in `closeCommitHash`. Instead, we eliminate those rows that do have `closeDate` but not `closeCommitHash`, since they represent 0.17%. Moreover, we see that the rows in which `debt` has no value represent 1.24% of the total. Therefore, as we cannot extract this information from other tables and there are few rows, we remove these issues from the table. Finally, the rows with a missing value in the `author` represent around the 20%. First of all, we try to relate this table with GIT_COMMITS to fill some values as we can know the author of the commit in this last table. The problem is that in SONAR_ISSUES, `author` is the e-mail address whereas in GIT_COMMITS the `committer` attribute is its name. So we cannot relate them immediately. After that, we have filled some rows but most of them still have a missing value in this variable, so we remove them.
The cleaning of this table can be found in this notebook.

GIT_COMMITS

In this case, we have 34 rows with missing values in both attributes `committer` and `author`. As it does not represent a big amount of rows, we can remove these rows.

This procedure can be found in this notebook.

SZZ_FAULT_INDUCING_COMMITS

The finally selected attributes of this table have no missing values so they do not need to be cleaned.
The link to the notebook is this one.

GIT_COMMITS_CHANGES

As in the previous case, all the selected attributes of this table are correct and do not have any quality problem, therefore, we do not have to change it.
The link to the notebook is this one.

REFACTORING_MINER

In this table we have two rows with a NaN in the columns corresponding to `commitHash` and `refactoringType`. As we cannot extract this information from other tables, we remove these rows to obtain the clean table.
The cleaning of this table can be found in this notebook.

3.3. Construct data

In this subsection we are going to create new attributes from the existing ones. The new attributes that we think that will be useful for our mining goals are the following ones:

a) An attribute that we think that will be useful for our data mining goals is, for each developer, the time that they have spent in each project. Therefore, to create this new attribute we need the table GIT_COMMITS. In this table, for each `committer` and project, we subtract the first `committerDate` from the last `committerDate`.
   The creation of this table can be found in this notebook.
b) Another attribute that can be interesting for our goals is the number of commits of each developer, as it can give us information about the work of each developer. That is, we can see if a certain person is more likely to program or if, however, it does it sometimes.
   This procedure can be found in this notebook.
c) Moreover, we believe that the number of fixed issues can be interesting to know if a certain developer focuses on this type of work or if, on the contrary, it does not concentrate on fixing issues. So we create a new dataframe in which we store, for each developer, the number of Sonar and Jira issues, and the SZZ faults that each developer has made in its commits.
   The link to the notebook is this one.
d) Related to the previous one, we also consider that the number of commits that induce an issue can be interesting to know which developers are more likely to produce some issues and, therefore, what kind of developers produce each one of them. In this case, we can only obtain information from the

SONAR_ISSUES and SZZ_FAULT_INDUCING_COMMITS, relating them with the GIT_COMMITS table as previously.
The link to the notebook is this one.

e) From the REFACTORING_MINER table, we think that it can be interesting to know which commits produce an issue when this type of task has been done. So we add a new attribute, that is, a new column, to this table in order to know if the commit has induced an issue or not. To obtain this type of information, we relate this table with SZZ_FAULT_INDUCING_COMMITS and, if a hash appears in both, we consider that it has induced a fault.
The modification of this table can be found in this notebook.

f) Another attribute that we think that is interesting is the amount of time that has been needed to resolve a Jira issue (in hours). Therefore, we need the JIRA_ISSUES table. To construct this new attribute (called, `resolutionTime`) we just have to subtract `creationDate` from `resolutionDate`.
This procedure can be found in this notebook.

g) This type of attribute can also be constructed from the SONAR_ISSUES table subtracting `creationDate` from `closeDate`, and creating an attribute called `closeTime`. This way, we get the time needed to close a SonarQube issue (in hours).
This procedure can be found in this notebook.

h) Finally, in the table SONAR_MEASUES we see that the values of the attributes are about all the code of the project. Since we are interested in the contributions that are made by each developer, we are going to transform all the attributes in this table to the difference between a commit and the previous commit. That is, given a commit, we do not want to know the total number of functions (for example) but the number of functions added by this developer in this commit. So, we have to: for each project, sort the commits by time and then compute the difference of the values with respect to the previous commit. Since this table does not have a time attribute, we have to join it with the GIT_COMMITS table (which has the attribute `committerDate`) using the commitHash attribute.
This procedure can be found in this notebook.

3.4. Integrate data

In this section, we combine the information from multiple tables, the ones that we have cleaned and created in the previous parts. To do it, we will use principally the primary and foreign keys that relate the different tables. In our case, after the selection attributes, we will use `commitHash` and all of the related ones as `closeCommitHash`, `faultFixingCommitHash`, etc. The integration of the tables can be found in this notebook.

Moreover, we generate the final table that will be used in the following steps, by aggregating values. In particular, we create a table that has, for each developer, all the relevant information that can be used to define its profile. In some cases we average the values of the attributes if we have, for a certain developer, several rows with different values. For example, the number of bugs in the SONAR_MEASURES table depends on each commit, so we can average the number of bugs in all its commits. In other cases, like some of the tables that we have created in the previous section, we only have to join them as they contain, for each developer, the number of fixed issues they have repaired.

Another aspect to highlight in this step is that many of the developers appear in some tables but not in all of them. Therefore, when merging the tables, some attributes for some developers were 'NaN', so we have replaced them with values that make sense. For example, when we join the tables COMMITS_FREQUENCY and FIXED_ISSUES (both of them have been created at the "*3.3. Construct data*" section), some committers have no values in the `time_between_commits` attribute. We can say that this is because they are developers who have fixed issues but have not made any commit. Therefore, to reflect this fact, we substitute these missing values for infinity, since this value indicates that the time between commits is infinite, so we can consider that it never commits. On the other hand, for those developers who have made commits but have not fixed any issues, we replace the 'NaN' values with 0.

To finish this section, after joining all the tables with the attributes that we are interested in and aggregating the values by the committer (developer), we obtain a table with 2,460 developers (rows) and 83 attributes (columns).

3.5. Format data

Finally, once we have created the final table with all the relevant information, we have to format it. This procedure can be found in this notebook.

One task is to modify the data in order to prepare it for the modeling tool that will be used to achieve the project goals. For example, we have to reorder the attributes so that the first column refers to the primary key whereas the last one contains the outcome of the model. In our case, we have as primary key the identifiers of the developers. However, we have an unsupervised problem as we do not have any label that indicates the real developer profile. But this is not a problem because, as we will see in the next section, the model that we will use does not need a target value.

Moreover, we modify the attributes names in order to be consistent with the information from the different tables. For example, we separate the bugs detected by Jira and Sonar by calling the first ones `jiraBug` whereas the other ones are identified by `sonarBug`.

Finally, we check that each of the attributes is of the desired type, that is, we check if they are integers, floats, strings, etc. In addition, we modify the infinite values to the highest value that Python supports in order to not have problems with the modeling tool.

3.6. Dataset description

After all these steps (selection, cleaning, constructing, integrating and formatting) we end up with our final dataset, with which we will carry out the modeling.

This final dataset has 2460 records, each one representing a developer, and 83 columns, each one representing an attribute about the developer. These attributes are the following ones:

Name attribute	Type	Description
`numberCommits`	int	Total number of commits made by the developer
`fixedSZZIssues`	int	Number of issues detected with the SZZ algorithm fixed by the developer.
`fixedSonarIssues`	int	Number of SonarQube issues fixed.
`fixedJiraIssues`	int	Number of Jira issues fixed.
`inducedSZZIssues`	int	Number of issues detected with the SZZ algorithm induced by the developer.
`inducedSonarIssues`	int	Number of SonarQube issues induced.
`timeInProject`	int	Mean time that the developer spends on projects, in seconds.
`resolutionTime`	float	The average time that each developer spends working on Jira issues.
`jiraBug`	int	Total number of bugs in which the developer has worked on.
`jiraDependencyUpgrade`	int	Number of Jira issues of type “Dependency Upgrade” in which the developer has worked on.
`jiraDocumentation`	int	Number of Jira issues of type “Documentation” in which the developer has worked on.
`jiraEpic`	int	Number of Jira issues of type “Epic” in which the developer has worked on.

jiraImprovement	int	Number of Jira issues of type “Improvement” in which the developer has worked on.
jiraNewFeature	int	Number of Jira issues of type “New Feature” in which the developer has worked on.
jiraQuestion	int	Number of Jira issues of type “Question” in which the developer has worked on.
jiraStory	int	Number of Jira issues of type “Story” in which the developer has worked on.
jiraSub-task	int	Number of Jira issues of type “Sub-task” in which the developer has worked on.
jiraTask	int	Number of Jira issues of type “Task” in which the developer has worked on.
jiraTechnicalTask	int	Number of Jira issues of type “Technical Task” in which the developer has worked on.
jiraTest	int	Number of Jira issues of type “Test” in which the developer has worked on.
jiraWish	int	Number of Jira issues of type “Wish” in which the developer has worked on.
jiraBlocker	int	Number of Jira issues of priority “Blocker” in which the developer has worked on.
jiraCritical	int	Number of Jira issues of priority “Critical” in which the developer has worked on.
jiraMajor	int	Number of Jira issues of priority “Major” in which the developer has worked on.
jiraMinor	int	Number of Jira issues of priority “Minor” in which the developer has worked on.
jiraTrivial	int	Number of Jira issues of priority “Trivial” in which the developer has worked on.
commitChangeAdd	float	Mean number of commits of change type “Add”.
commitChangeDelete	float	Mean number of commits of type “Delete”.
commitChangeModify	float	Mean number of commits of type “Modify”.
commitChangeRename	float	Mean number of commits of type “Rename”.
commitChangeUnknown	float	Mean number of commits of type “Unknown”.

linesAdded	float	Mean number of lines that the developer adds in a commit.
linesRemoved	float	Mean number of lines that the developer removes in a commit.
refactoringInducedBug	float	Mean number of bugs induced after a refactoring.
refactoringChangePackage	float	Mean number of refactorings of type “Change Package” made by the developer.
refactoringExtractAndMoveMethod	float	Mean number of refactorings of type “Extract And Move Method” made by the developer.
refactoringExtractClass	float	Mean number of refactorings of type “Extract Class” made by the developer.
refactoringExtractInterface	float	Mean number of refactorings of type “Extract Interface” made by the developer.
refactoringExtractMethod	float	Mean number of refactorings of type “Extract Method” made by the developer.
refactoringExtractSubclass	float	Mean number of refactorings of type “Extract Subclass” made by the developer.
refactoringExtractSuperclass	float	Mean number of refactorings of type “Extract Superclass” made by the developer.
refactoringExtractVariable	float	Mean number of refactorings of type “Extract Variable” made by the developer.
refactoringInlineMethod	float	Mean number of refactorings of type “Inline Method” made by the developer.
refactoringInlineVariable	float	Mean number of refactorings of type “Inline Variable” made by the developer.
refactoringMoveAndRenameAttribute	float	Mean number of refactorings of type “Move And Rename Attribute” made by the developer.
refactoringMoveAndRenameClass	float	Mean number of refactorings of type “Move And Rename Class” made by the developer.
refactoringMoveAttribute	float	Mean number of refactorings of type “Move Attribute” made by the developer.
refactoringMoveClass	float	Mean number of refactorings of type “Move Class” made by the developer.

refactoringMoveMethod	float	Mean number of refactorings of type “Move Method” made by the developer.
refactoringMoveSourceFolder	float	Mean number of refactorings of type “Move Source Folder” made by the developer.
refactoringParameterizeVariable	float	Mean number of refactorings of type “Parameterize Variable” made by the developer.
refactoringPullUpAttribute	float	Mean number of refactorings of type “Pull Up Attribute” made by the developer.
refactoringPullUpMethod	float	Mean number of refactorings of type “Pull Up Method” made by the developer.
refactoringPushDownAttribute	float	Mean number of refactorings of type “Pull Down Attribute” made by the developer.
refactoringPushDownMethod	float	Mean number of refactorings of type “Pull Down Method” made by the developer.
refactoringRenameAttribute	float	Mean number of refactorings of type “Rename Attribute” made by the developer.
refactoringRenameClass	float	Mean number of refactorings of type “Rename Class” made by the developer.
refactoringRenameMethod	float	Mean number of refactorings of type “Rename Method” made by the developer.
refactoringRenamePackage	float	Mean number of refactorings of type “Rename Package” made by the developer.
refactoringRenameParameter	float	Mean number of refactorings of type “Rename Parameter” made by the developer.
refactoringRenameVariable	float	Mean number of refactorings of type “Rename Variable” made by the developer.
refactoringReplaceAttribute	float	Mean number of refactorings of type “Replace Attribute” made by the developer.
refactoringReplaceVariableWithAttribute	float	Mean number of refactorings of type “Replace Variable With Attribute” made by the developer.
codeFunctions	float	Mean number of functions created by the developer.
codeCommentLinesDensity	float	Density of commented lines of the developer.

codeComplexity	float	Mean complexity of the code of the developer in a commit.
codeFunctionComplexity	float	Mean complexity of the functions.
codeDuplicatedLinesDensity	float	Density of duplicated lines of the developer.
codeViolations	float	Mean number of violations that the developer introduces in a commit.
codeBlockerViolations	float	Mean number of violations of severity “Blocker” that the developer introduces in a commit.
codeCriticalViolations	float	Mean number of violations of severity “Critical” that the developer introduces in a commit.
codeInfoViolations	float	Mean number of violations of severity “Info” that the developer introduces in a commit.
codeMajorViolations	float	Mean number of violations of severity “Major” that the developer introduces in a commit.
codeMinorViolations	float	Mean number of violations of severity “Minor” that the developer introduces in a commit.
codeCodeSmells	float	Mean number of violations of type “Code Smells” that the developer introduces in a commit.
codeBugs	float	Mean number of violations of type “Bugs” that the developer introduces in a commit.
codeVulnerabilities	float	Mean number of violations of type “Vulnerabilities” that the developer introduces in a commit.
codeCognitiveComplexity	float	Mean complexity related to how hard it is to understand the code’s control flow.
codeNcloc	float	Mean number of effective lines of code per commit.
codeSqaleIndex	float	Mean effort (in minutes) induced by the developer in a commit to fix all code smells.
codeSqaleDebtRatio	float	Mean ratio between the cost to develop the software of a commit and the cost to fix it.
codeReliabilityRemediationEffort	float	Mean effort (in minutes) induced by the developer in a commit to fix all bug issues.

`codeSecurityRemediationEffort`	float	Mean effort (in minutes) induced by the developer in a commit to fix all vulnerability issues.

4. Modeling

4.1. Modeling assumptions

The nature of our problem is to create homogeneous groups of developers that share some characteristics and differ significantly from the other groups. For that reason, we need to perform clustering analysis. This is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups.

Cluster analysis is not an automatic task so it consists in an iterative process of knowledge discovery that involves trial and failure. It is often necessary to modify data preprocessing and model parameters until the result achieves the desired properties.

It can be achieved by various algorithms that differ significantly in their understanding of what constitutes a cluster and how to efficiently find them. The appropriate clustering algorithm and parameter setting (including parameters such as the distance function to use, a density threshold or the number of expected clusters) depends on the individual data set and intended use of the results.

We are going to use *scikit-learn*, in particular its module named *clustering*, where there are several methods implemented. To start to get an idea of the type of cluster that we were going to find, we have tried several methods roughly. The results for all of them were worse or similar to the popular K-means algorithm. For that reason, we have decided to go deeper in just one method, K-means in this case, and try to get the best possible results.

The used algorithm make some important assumptions with respect to data:

- The variance of the distribution of each attribute is spherical.
- All variables have the same variance.
- The prior probability for all the clusters are the same (i.e. each cluster has roughly the same number of observations).

In addition, we have to identify the more frequent issues of each developer profile. To do that, we assume that the clusters are well defined, that is, that they separate the developers clearly according to their characteristics. So we have used the centroid of each cluster, that is, the representative of the developers of the same profile, to obtain this information. In other words, once we have obtained the centroids of each profile, we have searched for the most frequent issues of this representative.

4.2. Test design

To test our models we will use two techniques: see if different clearly visible profiles are formed and see if the developers of a certain group are sufficiently homogeneous with each other or not and, at the same time, if they are different with respect to those of another profile.

Therefore, we have one visual test, in which we will plot the observations after the dimensionality reduction to see if the clusters are sufficiently distinguishable; and another test related to the homogeneity of the profiles, that is, we can use the variance between and within the developers of the different clusters.

- *Within cluster variance:* it is an estimation of the dispersion of the observations within a cluster and it can be computed as the average of the distance between individual observations and the center of their cluster (also called centroid). So we get one value for each cluster k:

  $$\sigma^2_{w,k} = \frac{||X_{k,n} - c_k||}{N_k},$$

  where $||\cdot||$ is the Euclidean distance, N_k is the number of observations of this cluster, $X_{k,n}$ are the observations contained in the cluster and c_k is the centroid of the cluster.
  Summing over all the clusters we get the total within-cluster variance:

  $$\sigma^2_w = \sum_{k=1}^{K} \sigma^2_{w,k}$$, where K is the total number of clusters.

- *Between cluster variance:* it is an estimation of the variation between all clusters. A large value can indicate clusters that are spread out, while a small value can indicate clusters that are close to each other. This metric can be computed as:

  $$\sigma^2_{b,k} = N_k \cdot \left\| c_k - \overline{X} \right\|,$$

  where $||\cdot||$ is the Euclidean distance, N_k is the number of observations of the k-th cluster, c_k is the centroid of this cluster and $\overline{X}$ is the variables mean over all the observations (not just the ones from this cluster).
  Summing over all the clusters we get the total between-cluster variance:

  $$\sigma^2_b = \sum_{k=1}^{K} \sigma^2_{b,k}$$, where K is the total number of clusters.

This can also be seen in the *Calinski-Harabasz index*, where the within- and the between-cluster variances are used to get an index that indicates us which clustering technique has the most well-defined clusters, with a large σ^2_b and a small σ^2_w. Therefore, the clustering algorithm that produces a collection of clusters with the biggest Calinski-Harabasz index is considered the best algorithm based on this criterion. This index is defined as:

$$CH = \frac{\sigma^2_b}{\sigma^2_w} \cdot \frac{N-K}{K-1},$$

where N is the total number of observations and K is the total number of clusters.

Also, regarding this intuition that items in the same cluster should be more similar than items in different clusters, we can use the *Davies–Bouldin index* to assess the quality of clustering algorithms used. This index can be calculated by the following formula:

$$DB = \frac{1}{K} \sum_{k=1}^{K} \max_{k \neq k'} \left(\frac{\sigma_{w,k}^2 + \sigma_{w,k'}^2}{||c_k - c_{k'}||} \right) ,$$

where K is the total number of clusters, $||\cdot||$ is the Euclidean distance, c_x is the centroid of the cluster x and $\sigma_{w,x}^2$ is the within-cluster variance of the cluster x.

In this case, the clustering algorithm that produces a collection of clusters with the smallest Davies–Bouldin index is considered the best algorithm based on this criterion.

Finally, to evaluate our models, since we cannot compute its accuracy because we do not have a target, we will study the different profiles obtained with each model to see if they make sense to us, considering our knowledge of the topic and what we expect to obtain.

Then, to evaluate the list of most frequent issues made by each profile that we have obtained, since we do not have a target, again, we will use our knowledge to determine if they make sense.

4.3. Model description

In order to construct our final model, we follow two steps:

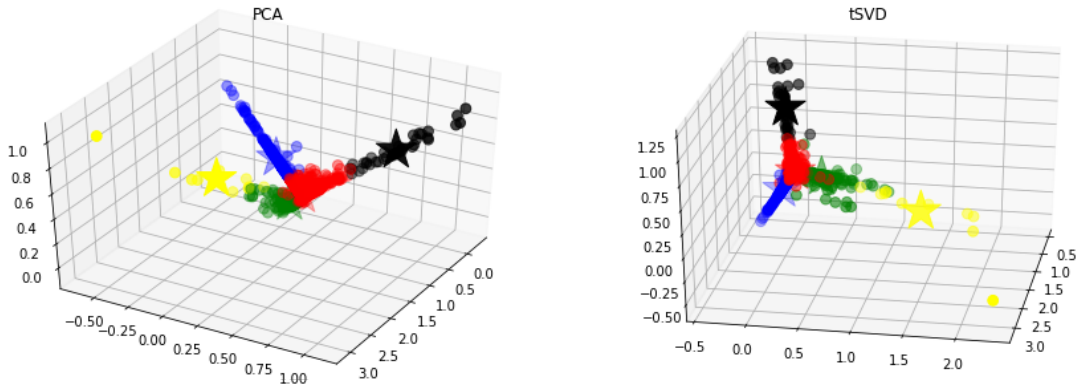
1. **Dimensionality reduction:** The prepared dataset from the previous section contained more than 80 distinct features. The feature space, then, was too big (so **too sparse**) to perform clustering directly over it. Moreover, each of the features in the original space did not explain much variance (not even 5%), so a feature selection was not possible. We wanted to extract the maximum information possible from these 83 variables and compress it into as few dimensions as possible. In order to do this, a simple **PCA** (and later on another technique, called **Transposed SVD**) was performed. We then picked the first n 'components' that compose 75% **of the variability**. We argue that 75% of variance explained is enough as the other 25% is, quite possible, noise. So, we project the original data into this subspace of n dimensions. Note that the Principal Component space is made up of linear combinations of the original features, thus allowing for an interpretability component of the model. That is, we can tell the importance of each of the original features to each principal component.

2. **Clustering** Over the components extracted from the previous step we performed clustering with **KMeans**. We performed a number of experiments to find the optimal number of clusters over the data. Finally, with the found clusters of the data projected onto n principal components, we extracted what each of them meant. This could be done due to the fact that each principal component was mostly correlated to a/some original feature/s. Then, we found the characteristic in terms of the principal components of each cluster, equivalently telling us which original features characterized the cluster. Moreover, given that we know **the importance of each original feature to each principal component** and that we can extract the importance of each principal component for each cluster centroid, we can quantify the relation between each centroid and each original feature. This allows for a very precise characterization of each cluster given the original features.

4.4. Model assessment

In this section we describe the results of testing the models obtained in the previous section according to the test design. As we have mentioned in the *4.2. Test design* part, as we cannot obtain an accuracy of the results, we think that it can be interesting to see if the developers of a certain group are sufficiently homogeneous with each other or not and, at the same time, if they are different with respect to those of another profile. Moreover, we can take into account if different clearly visible profiles are formed; and if the found profiles make sense to us (considering our knowledge of the topic).

On one hand, with respect to the visualization test, we can see in the figures that there are different directions to represent the clusters. In the first case, in which we have used the Principal Component Analysis (PCA) to reduce the dimension of the data, we can see that the profiles are separated, first, in three main directions and, then, in two of them they are divided too. In the other image, relating to the tSVD method, we can see three main directions too. Moreover, we can see that one cluster refers to the origin, three of them correspond to the three main directions and, the last one, are developers that do not match with any of these directions. So it seems that this is not the best option to verify the success of our model because we are limited by the 3D space.



Then, we have defined another test related to the homogeneity of the profiles, that is, we can use the variance between and within the developers of the different clusters. In the case of the PCA, the sum of all the within cluster variances is almost 118 million, whereas the total variance between clusters is about 61.000 millions. However, in the tSVD method, the values obtained are very similar, but the total within cluster variance is lower and the between cluster variance is bigger. Therefore, it seems that this last method is better as developers of the same clusters are more homogeneous and, at the same time, they are different with respect to those of another cluster.

Regarding the *Calinski-Harabasz index*, with the PCA technique we get a value of $CH_{PCA} = 319228.82$, while the tSVD technique has a value of $CH_{tSVD} = 320721.96$. Hence, since $CH_{tSVD} > CH_{PCA}$, regarding this criterion, the tSVD techniques gives a better collection of clusters.

Also, we have used the *Davies–Bouldin index* to assess the quality of clustering algorithms used. In our case, using the PCA technique we get a value of $DB_{PCA} = 7.353$, and with the tSVD techniques a value of $DB_{tSVD} = 7.407$. Therefore, since $DB_{PCA} < DB_{tSVD}$, with this criterion the best technique is PCA.

It is important to remark that it does not show if our results are good or not, but it gives us an idea of the best model. But since using different criterions to evaluate the collections of clusters obtained using two different techniques end up with different conclusions (with the Calinski-Harabasz index the tSVD is slightly better but with the Davies–Bouldin index is the PCA), we cannot deduce which technique is preferable.

Moreover, in the evaluation of the models, we can see from the attributes that are more important in each cluster if they distinguish the developers profiles or not. As it can be seen in the notebook used in the modelling part, we are available to extract the values more characteristics of each of the features more important of each cluster. In the case of the Principal Component Analysis used to reduce the data dimensionality, we can see that the most important attributes that characterize the profiles are the following ones:

1. **Profile 1:** The developers of this profile make a considerable amount of commits. They spend a lot of time in a project but not on fixing Jira issues. They sometimes apply some type of refactoring. They also fix some issues but also induce them.

2. **Profile 2:** This cluster is made of people that never do commits, work in the same project for a short period of time, spend a lot of time working on Jira issues but rarely fix them and are almost never assigned to work on this type of issues of priority Blocker, Critical, Major or Minor, or of type Bug, Task, Improvement or Sub-task.

3. **Profile 3:** These developers make some commits, are the ones that fix more Jira issues and the ones that are assigned to work on the greatest number of Jira issues of different priority and type.

4. **Profile 4:** These developers are the ones that make more commits, they fixed the greatest number of Sonar and SZZ issues but they induced these types of issues too. Moreover, these are the workers that stay more in the same project, use more functions in the code, program the most complex functions among all the profiles and apply many refactorings.

5. **Profile 5:** These programmers work a reasonable amount of time in the same project and in working on Jira issues which sometimes fix. Also, they are sometimes assigned to some Jira issues of different types and priorities.

So, we can see from the characterisation of each cluster that we have extracted information that differentiate the developers profiles on most of the cases. In the second cluster, there are people that work for a short period of time in the same project and do not tend to fix bugs. However, in the third profile,they are clearly the ones that fix the most number of issues. The 4-th profile consists of programmers that fix many issues but also induce them, their functions are the most complex ones, sometimes apply some refactoring and focus on a single project. The last profile is made up of developers that spend some time working on Jira issues on which they are assigned and which sometimes fix. Finally, in the first profile we have developers that, even though they do many commits and spend much time in a project, they fix just some issues (but also induce them).

From the clusters obtained after dimensionality reduction with tSVD, we study the most important attributes and their values in each cluster:

1. **Profile 1:** Regarding the developers of the first cluster, we can say that their code tends to be less expensive to fix than to make, they create some violations of type Blocker, Info and Major, as well as bugs but not many. They do not usually document their code and the complexity of their functions is low; but the general complexity of the code that they make is a little high but

not significantly. Finally, these developers create some functions but not many.

2. **Profile 2:** The code that they produce tends to be less expensive to fix than to make, they document a little bit their code, these developers are the ones that create the least amount of violations of type Blocker, Info and Major, and the least amount of bugs. They are the ones that spend more time working in the resolution of Jira issues but the ones that spend less time in a project. The complexity of their functions is very low as well as the general code that they make.

3. **Profile 3:** The code of this profile of developers tends to be less expensive to fix than to make. Also, these programmers are the ones that document the most of their code. On the other hand, they create a considerable amount of violations of type Blocker, Info and Major, and also bugs. Their functions are just a little bit complex but the general code that they make has a higher complexity. Finally, they spend a reasonable amount of time in a project but they rarely work in fixing Jira issues.

4. **Profile 4:** These developers dedicate a considerable amount of time to fix Jira issues, the code that they make tends to be less expensive to fix than to make, they document a little bit their code, rarely produce violations of type Blocker or bugs, they are the ones that fixed the most number of Jira issues, the complexity of their functions is very low and, also, they are the ones that have worked in fixing the most number of Jira issues of type Minor, Blocker and Major.

5. **Profile 5:** On the 5-th profile we have developers that tend to make code which is 5 times more expensive to fix than to make, that spend a lot of time in a project, do not document much their code, they do not spend time fixing Jira issues, are the ones that create the most number of violations of time Blocker and of type Info, and the most number of bugs, their functions are the most complex ones, they are the ones that make the most number of refactorings of the type “Extract Method” and the most number of commits.

Again, we can see from the characterisation of each cluster that we have extracted information that differentiate the developers profiles. In this case, in the fifth profile the developers are clearly the ones that make the most amount of issues, do not document their code, do not fix Jira issues, their functions are very complex, make many refactorings and their code is more expensive to fix than to make. On the other hand, the forth profile corresponds to the ones that fix most of the jira issues, rarely produce issues and their functions have a low complexity, even though they do not document much of their code. The other three profiles are in between these two.

Taking into account all these assessment processes and results, we can say that both models have successfully passed all tests. And, from the evaluations section, it

seems like the tSVD technique is slightly preferable since, with it, we obtained more consistent profiles with características that make more sense.

From these final profiles, we have obtained a list of frequent issues according to their characteristics:

1. **Profile 1:** Developers from the first profile create some issues, mainly of type Violation or Code Smell, with priority Major or Minor.
2. **Profile 2:** This group of developers almost never induce issues of any kind, but if they create one, it is usually of type Violation or Code Smell and of priority Major or Minor.
3. **Profile 3:** This is the second profile which makes the most number of issues; and, again, they usually create Violations and Code Smells with priority Major or Minor.
4. **Profile 4:** In general, this profile does not induce issues, but regarding the few that they create, the most part are Violations and Code Smells and the priority is usually Major or Minor.
5. **Profile 5:** These developers are the ones that induce the most number of issues in general, of all types and priorities. But the ones that they create the most are, again, Violations and Code Smells and the priority is usually Major or Minor.

As we can see, all the profiles tend to do the same type of issues, and with the same priority. The difference between the profiles is then in the amount of issues, not in their type.

5. Evaluation

5.1. Assessment of data mining results with respect to business success criteria

In this first section of the evaluation, the data mining results obtained previously are compared with the business objectives and the business success criteria. Therefore, we start by remembering the business goals and success criteria of this project that we have explained in the *1.1. Business objectives* section. Regarding the business goals, we wanted to distinguish between different developers profiles and, then, we wanted to know the main bugs that each one of them tends to produce. To achieve our first goal, we have implemented a segmentation model that creates different profiles of developers based on the selected attributes. After that, we have made a description problem to establish the most frequent bugs of each developer and, hence, achieve our second goal. Finally, we recall the business success criteria. The first one was to discover at least three very differentiated developers profiles and their associated common issues and, the second one, to reduce by ten percent the number of bugs created by each developer's profile by giving useful insight about the bugs they are more likely to do.

As we have seen, from the clusters obtained after dimensionality reduction with tSVD, we can study the most important attributes that characterize the profiles and their values in each cluster. We have seen that one of the profiles is made up of developers that clearly are the ones that make the most amount of issues, they do not document their code, they do not fix Jira issues, their functions are very complex, they make many refactorings and their code is more expensive to fix than to make. On the other hand, another profile corresponds to the ones that fix most of the Jira issues, rarely produce issues and their functions have a low complexity, even though they do not document much of their code. The other three profiles are in between these two, so we think that we can join them in order to create the third profile in which we have developers with some experience that can not be considered of the other two "extreme" groups. Therefore, we have obtained three clearly separated profiles of programmers, that is, we have fulfilled our main business success criteria.

- Don't document the code
- Complex functions
- Many refactorings
- Don't work on fixing issues

- Document the code
- Functions a little bit complex
- Work on fixing issues

- Document a bit the code
- Functions with low complexity
- A lot of time fixing issues

To see if our idea of joining three of the original profiles into one and finally getting 3 profiles makes sense, we have trained again our model but now indicating that we want three clusters (instead of five). We have seen that one cluster (first one) is made of developers that do not fix Jira issues, that work a lot of time in a project, that induce a lot of bugs and Blocker, Info and Major violations, that program complex code and functions and that do lots of refactorings. Then, we have seen another profile (third one) that fixes lots of Jira issues, that induces some bugs, but does not induce violations, and that makes functions of very low complexity. Finally, the third profile (second one) consists of developers that fix some Jira issues, produce some Blocker, Info and Major violations, and that program a low complexity code and functions. Therefore, we can say that our idea of having just three profiles by joining three of the five profiles into one makes sense, because the clusters obtained training the model with 3 clusters have more or less the same characteristics that ours and can also be classified the same way.

Regarding the second success criteria, it is not possible for us in this first iteration of the project to check if we have reduced by ten percent the number of bugs created by each developer's profile, since we would need some months to apply our results to the studied developers and see if they have improved their coding or not, but we do not have this amount of time. Moreover, we have seen in 4.4*. Model assessment* section that all of the profiles tend to induce the same type of issues, that is, major and minor violations and code smells, so we can say that the success criteria of discovering the most common issues for each profile has not been completely achieved because all the profiles have the same type of issues.

5.2. Review of process

In this section we focus on the effectiveness of the project and, after that, we identify factors that have not been taken into account but that can be considered important too.

First of all, as we have seen previously, we consider that the business objectives have been satisfied as we have been able to distinguish the developers according to different profiles. These groups have been obtained from different attributes related to commits, code characteristics and issues. So a company can use this information to improve their projects with respect to the quality, efficiency and speed. That is, it can be useful to create a better team formed by specific profiles depending on the objectives.

As we have mentioned before, we have used, basically, information about their commits, the code characteristics and changes, and the Sonar, Jira and SZZ induced and fixed issues. With this information we have been able to use 83 features to cluster the developers according to 5 different profiles. Moreover, we have been using a dataset that contains information about 33 projects. So in a more realistic scenario, the dataset would contain information about the workers of the company and, therefore, we would be able to extract more precise information.

Finally, we think that there are some factors that should be taken into consideration as improvement of this project. We think that it can be interesting to get information about the project domain. With this in mind, we can obtain more relevant features in order to distinguish better between developers using new insights. For example, we would be able to see if a certain developer always works according to the same profile or if, on the contrary, depending on the project it works like different profiles. As a result, we would be capable of checking if they are more versatiles or not and, moreover, we would be able to define more developers profiles with this new information.

5.3. List of possible action

Based on the previous sections, now we have to think about the next steps in the project. As we have mentioned, one of the main objectives of this project was to know what types of issues are more likely to be induced by each profile. So, once we have separated the developers by their characteristics, we can study each group independently in order to obtain this type of information more accurately, since now we just have a vague idea of the kinds of issues more frequently made by each profile. This can be useful to know the aspects that each programmer has to pay more attention to in future projects if they want to improve its efficiency and, consequently, the efficiency of the total project in terms of quality and speed. That is, we can use these new insights to reduce the number of issues of each programmer. But we can also take into account information about its comments, complexity of code, etc. in order to improve other skills. That is, we would have a model to personalize these recommendations depending on the developer. This type of information can be useful for both the programmer and the company.

Another important aspect is to reduce the number of induced issues, as we have previously said. That is, one of the business success criteria is to reduce by 10% the

number of induced bugs. In this case, we would need time to apply our results to the studied developers and see if they have improved their coding or not. So this is a task that has to be made in the next steps of the project. Moreover, we could check if there is an improvement in the other aspects we have mentioned previously, like the number of functions of the code, the commented lines, etc.

6. Deployment

6.1. Deployment plan

DEPLOYABLE RESULTS

The main results of our data mining project are the clustering model and the software developed to create this model (code in Python).

On one hand, the clustering model is basically the centroid of each cluster which, given a set of new values of the corresponding attributes for a new developer, computing the distance between the centroid of each cluster with these new set of values, we obtained the cluster (and so, the profile) to which the developer belongs (as the cluster whose centroide is closer to these new values). This model can be used by the managers to create more practical and efficient teams of developers to work on a specific project or task in a project. For example, if there is a task that has to be done in a very short time (for some reason), the manager can assign it to those developers that induce less bugs and make better code (the last group in the figure), but if there is a task for which we have more time, maybe it can be assigned to some other group, so they can learn from them without putting in risk the deadline of the project (in case they create bugs that have to be fixed later).

On the other hand, the code developed during this project can be used to prepare new data (obtained after the realization of new projects) and use it to update the previously mentioned model. Therefore, the clustering model will be more precise each time, and also, more personalized to the group of developers to which it will be applied.

DEPLOYMENT PLAN

The results previously mentioned have to be used by managers in an easy way. Therefore, a simple program has to be developed in order to facilitate the interaction with the model. That is, a program has to be created that asks for a set of values for some attributes of a developer and gives the profile to which he/she belongs and his/her characteristics. Also, managers should be able to update the model with new data in order to make it more reliable and personalized to a set of developers. Hence, the program has to offer the option to do so by accepting new data obtained after the realization of a project. The program, then, will use our software to prepare this raw data so that it can be used to update the model (also using our code).

After creating this program, we will have to explain to the managers how to use it and which purpose it has.

Also, using the new data used to update the model, we can know if the number of induced bugs is being reduced in each project. That is, if our model is helping managers create better teams or better assign tasks so that less issues are

produced. This way, we can monitor and measure the benefits of our project or, otherwise, detect if it has to be improved.

To sum up, our deployment plan has two steps:

1. Create a program to facilitate the use and update of our model by the managers (the users).
2. Inform the managers on how to use this program and why it can be useful to them.

6.2. Monitoring and maintenance plan

RESULTS DEPLOYMENT AND UPDATING

As we have mentioned previously, the main deployable results of our project are the clustering model (that is, the centroids) and the software (the code developed in Python).

With respect to the first one, the centroids of the models have to be updated while new data is introduced to the model in order to refine and improve the results. Moreover, with these incorporations, the model can be changed, that is, there can be more clusters or they may be different, meaning the characteristics that split them change in terms of relevance. In our case, we think it is proper to update the centroids of the model, that is, to introduce new data, at the end of each project. In this way, we would be able to incorporate all the aggregated data of the project and, then, to do it by developer. So if we introduce a new developer in the model, we would use this new data of this developer directly, but if the developer had already been used in the model, we combine the data with the previous one to update its features. In order to aggregate the data in an appropriate way, we would keep the selected features of each developer and project. Therefore, we would have several rows for a single developer, depending on the number of projects he/she has been involved in. So we will introduce the new data in this database that will be used to aggregate the attributes by the developer and, therefore, to create a single input for each programmer. With these ones, we would execute the model and, then, obtain the updated profile of each one of them.

Regarding the software, we think that it can be updated if we consider new features that can be interesting in our goal, that is, if we introduce new relevant attributes. Moreover, as we introduce new values, it can be changed too to fit the data better. In both cases, it is due to the fact that we try the number of clusters according to the elbow curve, which can change with these new additions. So we think that the number of clusters is important in order to obtain the best profiles of developers, but we also believe that it is not critical to improve the performance of the projects. For this reason, we think that the model can be updated annually with the data available until this moment.

RESULTS UPDATING PROCESS

In this subsection we summarize the results updating process of both deployable results. First of all, we have to bring up to date the model by updating the centroids of the clusters that define the developers profiles. To do that, as we have explained before, we store in a database the desired information of each developer and project where he/she has been working. So we will update this database with new information whenever a project finishes. In order to introduce the characteristics of each programmer, we aggregate the data by developer to obtain a unique input for each one of them. Finally, we fit the model to obtain the new centroids of the clusters and, then, to assign each developer according to these profiles.

In addition, it is recommended to update the model often, for example, annually, in order to select the optimum number of clusters based on the elbow curve. This one can change when new features and data are introduced. So we only have to modify the number of clusters according to the more appropriate number of clusters, in principle.

7. Final report

Here we present an overview of our project. From the motivation, all the way to the procedure and conclusions, the reader will grasp the meaning of the core concepts used in this analysis, as well as what the applicable results are. It is not meant as a thorough guide, but rather as a reminder document.

7.1. Summary of business understanding: background, objectives, and success criteria.

The main goal of our analysis is to find distinct groups of developer profiles based on distinct characteristics. After this goal is completed, an analysis of the different types of bugs that each developer group tends to introduce will be performed.

The results of our project are expected to be used by department leaders and general managers to help their developers be more efficient by reducing the number of bugs they produce providing them with training options, for example; or directly by developers if they want to improve their self-awareness. Bearing in mind that developers want to improve their abilities and managers are (or should be) interested in dealing with their projects more efficiently, we see the potential results as very favourable for the company.

Related to our goal, the main questions to be asked are:

- What are the types of developer profiles in our business?
- Which are the main bugs that each developer's profile tends to produce?

Which can be related to the following data mining goals, respectively:

- Developers must be segmented to create the different profiles based on some attributes selected from the data. So, it is a segmentation problem.
- For each of the profiles found above, look for the most frequent type of bugs. Therefore it consists of a concept description problem.

Moreover, we consider the following as success criteria, both from an analysis perspective and a business one:

1. To discover, at least, three distinct developers profiles and their associated common issues; using the variance between and within profiles to verify if the groups are sufficiently homogeneous and, at the same time, if they are different with respect to other profiles.
2. To reduce by ten percent the number of bugs created by each developer's profile by giving useful insight about the bugs they are more likely to introduce.

7.2. Summary of data mining process

In this project we have used The Technical Debt Dataset, a set of project measurement data from 33 Java projects. This data is obtained using different tools to analyze each commit. From this dataset, we had to select some attributes over which we trained our model. For our first goal (determinate different developer's profiles) we were interested in attributes that can somehow characterize a developer, while for our second goal (find the most frequent bugs made by each profile) we focused more on issues information.

Taking into account this selection criteria and after doing an exploratory and quality analysis of the data, we made the final selection of the attributes of each table. Then, we cleaned the data; that is, we had to deal with the missing values, either erasing the records that contain one or more (in each table), either filling them with another value. After that, we created new attributes that we thought would be useful for our mining goals from the existing ones. Next step was to combine the information from multiple tables (the ones that we had cleaned and created) by aggregating values, to generate the final dataset. During this integration step, we saw that many of the developers appeared in some tables but not in all of them. Therefore, when merging the tables, some attributes for some developers were 'NaN', so we replaced them with values that made sense. Then we just had to modify the attributes names in order to be consistent with the information from the different tables and check that each of the attributes was of the desired type.

After all this process, we obtained our final dataset, which had 2460 records, each one representing a developer, and 83 columns, each one representing an attribute about the developer. Some of these attributes are: total number of commits made by the developer; number of issues fixed and induced by each developer, detected with the SZZ algorithm, SonarQube or Jira; mean time that the developer spends in projects; information about issues like the number of issues made by each developer of an specific type or priority; information about the changes that a developer makes in the commits, like the mean number of commits of an specific type; mean number of bugs induced after a refactoring or mean number of refactorings of each type made by the developer; general information about the code of the developer, like the mean number of functions that he/she creates, the complexity of the code or the density of commented lines; etc.

Then, we used this final dataset to create our model. The nature of our problem was to create homogeneous groups of developers that share some characteristics and differ significantly from developers of the other groups. For that reason, we needed to perform clustering analysis. This is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups. We used *scikit-learn*, in particular its module named "clustering", where there are several methods implemented. We tried several methods but the results for all of them were worse or similar to the

popular K-means algorithm, in terms of both performance and interpretability. For that reason, we decided to go deeper with just one method, K-means in this case, and try to get the best possible results.

The used algorithm makes some important assumptions with respect to data:

- The variance of the distribution of each attribute is spherical.
- All variables have the same variance.
- The prior probability for all the clusters are the same (i.e. each cluster has roughly the same number of observations).

In addition, we had to identify the more frequent issues of each developer profile. To do that, we assumed that the clusters are well defined, that is, that they separate the developers clearly according to their characteristics. So we used the centroid of each cluster to obtain this information. In other words, once we obtained the centroids of each profile, we searched for the most frequent issues of this representative.

Moreover, in order to construct our final model we followed two steps:

1. **Dimensionality reduction** The prepared dataset from the previous section contained more than 80 distinct features. The feature space, then, was too big (so too sparse) to perform clustering directly over it. Moreover, each of the features in the original space did not explain much variance (not even 5%), so a feature selection was not possible. We wanted to extract the maximum information possible from these 83 variables and compress it into as few dimensions as possible. In order to do this, a simple PCA (and later on another technique, called Transposed SVD) was performed. We then picked the first n 'components' that compose 75% of the variability. We argue that 75% of variance explained is enough as the other 25% is, quite possible, noise. So, we project the original data into this subspace of n dimensions. Note that the Principal Component space is made up of linear combinations of the original features, thus allowing for an interpretability component of the model. That is, we can tell the importance of each of the original features to each principal component.

2. **Clustering** Over the components extracted from the previous step we performed clustering with KMeans. We performed a number of experiments to find the optimal number of clusters over the data. Finally, with the found clusters of the data projected onto n principal components, we extracted what each of them meant. This could be done due to the fact that each principal component was mostly correlated to a/some original feature/s. Then, we found the characteristic in terms of the principal components of each cluster, equivalently telling us which original features characterized the cluster. Moreover, given that we know the importance of each original feature to each principal component and that we can extract the importance of each principal component for each cluster centroid, we can quantify the relation between

each centroid and each original feature. This allows for a very precise characterization of each cluster given the original features.

See this notebook for the actual modelling procedure and a more thorough explanation.

Finally, to test our models we used two techniques: a visual test, in which we plotted the observations after the dimensionality reduction to see if the clusters were sufficiently distinguishable; and another test related to the homogeneity of the profiles, that is, we used the variance between and within the developers of the different clusters. Then, we used these variances to compute the *Calinski-Harabasz index*, which indicates which clustering technique has the most well-defined clusters (the larger, the better) and the *Davies–Bouldin index* (small is better).

Finally, to evaluate our models and the list of most frequent issues made by each developer profile, since we could not compute their accuracy because we did not have a target, we studied the different profiles obtained with each model to see if they made sense to us, considering our knowledge of the topic and what we expected to obtain.

7.3. Summary of data mining results

After the implementation of the model with the data selected, we obtained different developers profiles. In particular, we separated them in 5 clusters as we saw that it is the best number of clusters according to the elbow curve method. So we ended up with each one of the developers assigned to one of the clusters. Therefore we evaluated the results according to the developers profiles that the model created. We recall that we used two techniques to reduce the data dimensionality, PCA and tSVD; so we had to select the best technique from the results.

First of all, we tried to test the results using visualization techniques. We saw that the clusters were separated in three main directions, but we could not see them clearly forming groups. Therefore, we verified that this is not the best option to verify the success of the models as we are limited by the 3D space.

Then, we used techniques related to the homogeneity of the profiles using the variance between and within the developers of each cluster. Comparing both variances of each technique, we saw that the values obtained are very similar, but it seemed that the tSVD technique was slightly better. It was due to the fact that the within cluster variance was lower than the one obtained with PCA, and because the between cluster variance was bigger. Moreover, we used some indexes that relate these values and that indicated which technique had the most well-defined clusters. With the first one, the Calinski-Harabasz index, it seemed that the results were better when we used the tSVD to reduce the data dimensionality. However, using the Davies-Bouldin index, we obtained that the best results were obtained from the PCA. Finally, we remark that these results did not check if the results were good or not, but

they gave us insights to select the more appropriate model. Moreover, we saw that the conclusions obtained from the tests were not conclusive, as we could not deduce which technique was the best one in our case.

So, we decided to use the attributes that distinguish the clusters to select the best technique according to a subjective measurement. Therefore, we looked for the most important attributes that characterize each profile defined by each one of the techniques. In both cases we found profiles that were similar and that passed the tests, as we considered that the profiles were well defined and separated the developers in five profiles with different characteristics. Finally, we selected the tSVD as the best technique since the profiles seemed more consistent and their characteristics made more sense. We found two profiles with distinct characteristics, and another three that were in between these two groups and that differed on the characteristic values of the main attributes. Next, we describe the attributes and the values that were more important to each cluster:

1. **Profile 1**: The code of these developers tends to be less expensive to fix than to make it, they create some blockers, info and major violations, and some bugs. They don't usually comment code and the functions they program have low complexity, while the code is a little bit more complex.

2. **Profile 2**: As in the previous case, the code tends to be less expensive to fix than to make it, and it is sometimes commented on. They induce the least amount of violations of type blocker, Info and Major, and the least amount of bugs. They spend a lot of time fixing Jira issues, but they do not spend much time on a project. Moreover, they program functions and code of low complexity.

3. **Profile 3**: The code of these developers tends to be less expensive to fix than to make it, too. They are the ones that document the most of their code, that create lots of blocker, info and major violations, and also bugs. They program code and functions that are sometimes difficult, do not fix Jira issues usually, and they spend a reasonable amount of time on a project.

4. **Profile 4**: As before, it consists of developers that make code that is less expensive to fix than to make. Moreover, they spend enough time fixing Jira issues, they document a little of their code, and rarely produce blocker violations or bugs. These programmers are the ones that work the most in fixing Jira issues, and that program functions of low complexity.

5. **Profile 5**: Unlike the previous groups, these developers tend to make code that is more expensive to fix than to make. Moreover, they spend a lot of time on a project, they do not document their code, and their functions are the most complex. Regarding the issues, they do not fix Jira issues, but they create the majority of bugs and blocker and info violations. Finally, they do most of the refactorings and commits.

Next, from these obtained profiles we got a list of their frequent issues to get the most frequent errors of each cluster. But, in all cases, we saw that they tend to introduce the same type of issues and with the same priority. That is, all of them create, to a greater extent, Violations and Code Smells of Major or Minor priority. So, in this case, the differences between profiles were in the amount of issues that they create, not in their type.

7.4. Summary of results evaluation

The **business goals** for this project were to distinguish between different developers profiles and, then, to know the main bugs that each of them tends to produce. To achieve the first goal, we have implemented a segmentation model that creates different profiles of developers based on the selected attributes. After that, we have made a description problem to establish the most frequent bugs of each developer and, hence, achieve our second goal.

In addition, we had two main **business success criteria** for the project. The first one was to discover at least three very differentiated developers profiles and their associated common issues and, the second one, to reduce by ten percent the number of bugs created by each developer's profile by giving useful insight about the bugs they are more likely to introduce. From the clusters obtained after dimensionality reduction with tSVD, we can study the most important attributes that characterize the profiles and their values in each cluster. This way, we have seen that we can clearly distinguish between three groups of developer profiles based on the following aspects: whether they document the code, the complexity of their functions, number of refactorings, work done fixing issues and quantity of specific issues introduced. Therefore, we have fulfilled our main business success criteria. Regarding the second success criteria, it is not possible for us in this first iteration of the project to check if we have reduced by ten percent the number of bugs created by each developer's profile, since we would need some months to apply our results to the studied developers and see if they have improved their coding or not. This, and the fact that our results are not only applicable for detecting the types of bugs each profile introduces is discussed more thoroughly in **section 7.6**.

This project can be useful to teach the aspects that each programmer has to pay more attention to in future projects if they want to improve its efficiency and, consequently, the efficiency of the total project in terms of quality and speed. That is, we can use these new insights to reduce the number of issues of each programmer. Moreover, the company can use this information to improve their projects with respect to the quality, efficiency and speed. That is, it can be useful to create a better team formed by specific profiles depending on the objectives. Again, a specific action plan derived from our results is in **section 7.6.**

7.5. Summary of deployment and maintenance plans

There are two main deployable results of our data mining project:

- **The clustering model**. It consists of a set of centroids for each resulting cluster and the dimensionality reduction model (tSVD). It is needed in order to classify new developers to some group according to its corresponding new data.
- The software developed to create this model (code in Python). It can be used to update the clustering model with more training data in order to achieve better results in terms of precision and accuracy of the model.

The deployment plan for this project consists in two steps:

1. Create a program to facilitate the use and update of our model by the users. This program has to have two main functionalities. On the one hand, it asks for an input consisting in a set of values for some attributes of a developer and gives the profile to which he/she belongs and his/her characteristics. On the other hand, it has to update the model with new data in order to make it more reliable and personalized to a set of developers.

2. Inform the managers on how to use this program and why it can be useful to them. That is, presenting actionable key points that the direction should take into account.

Regarding the maintenance plan, it is proper to update the centroids of the model, that is, to introduce new data at the end of each software project. This way, we would be able to incorporate all the aggregated data of the project and, then, to do it by developer. So if we introduce a new developer in the model, we would use this new data of this developer directly, but if the developer had already been used in the model, we combine the data with the previous one to update its features. Then, the centroids of the models have to be updated and some changes may occur regarding for instance the number of proper clusters or the main characteristics of each one. Moreover, the software can be updated just if we consider new features that can be interesting to our goal, that is, if we introduce new relevant attributes.

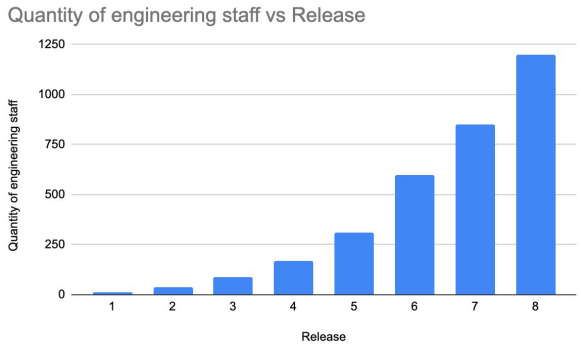
7.6. Cost/benefit analysis

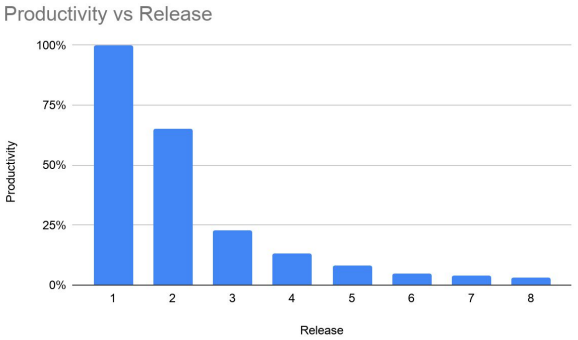
Let us delve deep into a business use case of our modelling in order to understand the potential benefits of adjusting things according to our results. We have extracted the data used from a real technology company (Martin 5-12[1]).

Consider this: until now, a company has released 8 versions of their software. With each release, the number of engineering staff increased exponentially, from around

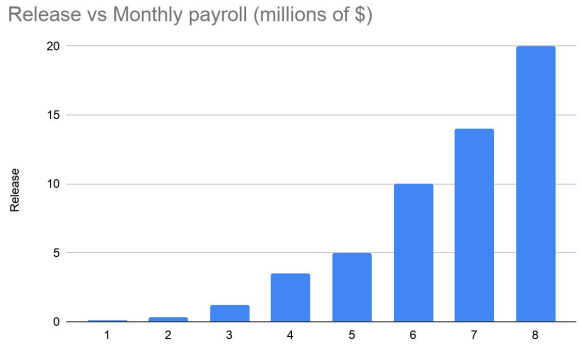
[1] Martin, Robert C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

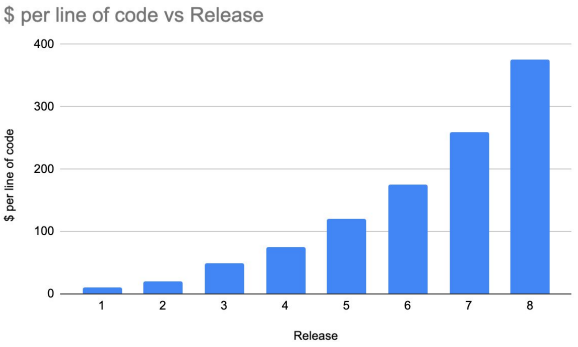
10 all the way to 1200. The productivity, measured in lines of code, decreased exponentially.





Which means that the cost of each line of code increased, also exponentially. Then, if we see the monthly payroll costs, it truly looks like a bad situation: the company is using up more and more resources for each new release while productivity is decreasing drastically.





The company got here by not establishing maintainability protocols from the beginning. Messy code was published in order to enter the market quickly but was never maintained. So, technical debt increased very fast, and now making a small modification to the codebase can generate many issues at once, even undetectable ones. The software is not being taken good care of.

Given this situation, how can our results help? The underlying issue of the above scenario is that developers are not accustomed to make maintainable code. Their skills are too focused on writing fast. Now, our analysis has not been centered around maintainability, but we have seen that 2 of our clusters contain developers that do not properly prepare their code for maintainability (i.e. null documentation). If they are not even documenting properly, it is almost certain that no unit testing or git commit discipline is being taken into account. This is where our results come into place. We have certainly identified 2 groups of developers that have bad behaviours: they write complex code, don't document, perform lots of commits. Also, we have observed that the problems introduced by the bad behaviour of these first two groups cause the third group to be fixing issues very regularly. If we can adjust the skills of

the first two group for a more disciplined approach to software development, three things will happen:

1. The first two groups will truly add value to the software being developed.
2. The third group will not be fixing issues so regularly, being able to dedicate their time to more enriching tasks.
3. Due to the latter points, productivity will start rising, technical debt will start decreasing and future releases will be cheaper and more consistent. The business could redirect 'saved' resources into other agenda and engineering staff could dedicate themselves to more fulfilling tasks.

We believe a good effort on these 2 bullet points will make an impact. Going back to the real business use case[2], we believe that, at least, a very conservative 1% increase in productivity (which we think could actually be higher) could be achieved for the next software release (not much because training takes time). Dividing the monthly payroll cost of release 8 (20.000.000$) by the price of each line of code on release 8 (375$) we get an estimate of the number of lines produced in release 8: 53333. If we interpreted that an increase in productivity produces an equal drop in $ per line of code, that means that the same lines of code with the same 1200 developers would cost the business a monthly total of about 19.800.000$ (367,5$ per line of code): 200.000 dollars saved per month. In a year, the business would have saved roughly 2.5 million dollars. This use case example is applicable to any other business in a similar situation.
Thus, the cost/benefit ratio of this project for the next software release, in this concrete case, would be of 10000$ / 2.500.000$ = 0,004, which is excellent. We consider the cost of the project to be the sum of an estimated monthly payroll for 4 junior engineers on a common salary and the resources used for modelling and data recollection. Of course, this ratio depends on the scale of the company the results are being applied to (small amounts of data would not be significant, so this analysis only makes sense in the context of a big company). Still, as the cost was not too high and the impact it can have is quite important, it will be small and keep decreasing through the years. We have not taken into account the cost of the future action plan.

7.7. Conclusions for the business

The previous use case solution defines an accordion effect: more productivity means more resources to fix past issues, which means maintainable code, which means more productivity, etc. Thus we believe that the effects of the application of our results will be of great value to the business. Let us state the main contingencies we believe are important (in any order):

1. **Reorganize teams**. For a short term contingency, reorganize developers from groups 1, 2 and 3 according to their described characteristics into teams such

[2] Keep in mind that we consider the data used in this project to be from projects of the same business. A fictional environment that allows us to derive meaningful conclusions.

that the odds of producing complex and non-maintainable code are minimized. That is, balance the types and number of developers in each team according to their current production of faulty code.

2. **Plan and re-adjust training**: Readjust the business' employee training plan or create one. It is clear that groups 1 and 2 need specialized training. They need to produce (1) **maintainable**, (2) **robust**, and (3) **simpler** code. Respectively, the training needs to be focused on:

   (1) **documentation standards** and **maintainable software architectures**.
   (2) producing and using **unit tests**.
   (3) **advanced** use of their respective **programming languages** (so the code is more refined).

3. Robust and tested code will not create issues, so it is very important. Other, more focused trainings are advised to be put in place:

   a. One that focuses on the types of issues they introduce in the code (we have seen that all three groups introduce the same issues but in different proportions).
   b. **Commit etiquette**. Group 1 (and 2, but less) need to learn commit etiquette. Again, a small training would have to be devised. Messy, very frequent, undocumented commits are a recipe for hard-to-maintain code.

   This training would be used on new employees and current ones according to what cluster they belong to.

We argue that the increment in productivity derived in the previous section would be bigger for the next release (8+) and so on, for the reasons stated in the use case introduction. Moreover, we believe that the stated training would very much accomplish a 10% drop in code issues introduced for the next release, which is what our 2nd business success criteria stated.

Thus, we conclude that, given the potential results for the business if everything here is taken into account, this project has been successful in respect to what the goals were and its costs. We believe that the business will greatly benefit from the conclusions drawn here.

7.8. Conclusions for future data mining

The obtained results after modelling and evaluating have led us to think of new actions that can be taken in future iterations of the project. Below, we detail some of them.

The resulting dataset from the data preparation step consists of 83 features with aggregated information for each developer about different coding aspects such as

refactoring, commits, bugs, etc. We have checked that feeding a Principal Component Analysis (PCA) with this data just results in nine relevant principal components that account for 75% of data dispersion. This means that although we have quite a lot of different features, most of them are not relevant enough overall and with just nine Principal Components (combination of the others) we can explain almost the same variability.

For feature iterations of the project, we would like to have a preprocessed dataset of higher quality. This change could have two different consequences: we could not need to use dimensionality reduction or, if we perform it, we would obtain more data dimensions to use when clustering (less correlation between the original features). For this purpose, different techniques can be applied:

1. Taking into account which features are the ones that explain the least variability on the dataset and removing them: the result will barely be affected by this change.
2. Creating new derived features as the result of combining some of the original ones. However, this can be a very tedious task as there are a lot of attributes that can be created as a combination of others. For this reason, we should focus only on the ones that are meaningful for the project purpose and the context where we are working.
3. Gathering and adding data from other internal sources. It would be helpful to have data on more projects apart from the 33 on the original data, so we can obtain better results closer to the reality of the company. Moreover, we think that the clustering model could get a lot out of it if we have information of each project context and the tasks each developer takes part in.

Note that we have performed step 2 in this project, but not to the extent that we think could be optimal.

In the first iteration of this data mining project we have used basic linear dimensionality reduction and euclidean clustering analysis in order to obtain and characterize different developer profiles in the available data. For future iterations, other types of algorithms, in particular the non-linear ones, can help in the research of better results. These kinds of methods are not as interpretable so we could hardly induce which features are the ones that are more important for the clustering and for each group of developers. However, the type of pattern they are able to capture in the data is more complex and thus more relations between attributes can be identified. In particular,

- **Dimensionality reduction**: We have already used two different methods, PCA and tSVD. They are linear, so both of them construct new relevant attributes as linear combinations of the original features, minimizing different functions with some constraints. In the future, we could substitute them by, for example, kernel PCA, that allows the model to construct the new attributes or

principal components as non-linear combinations of the original attributes using the kernel matrix through the feature space.

- **Clustering analysis**: In this iteration we have mainly focused on the popular K-means algorithm and we have tried to obtain the best possible results with them. However, there are several other algorithms that do use other kinds of distances (not Euclidean) to create the clusters or non-linear algorithms, such as the agglomerative one with several linkages. It would be useful to study in depth which ones (according to their data assumptions) could be applied to our data.

Clustering analysis is a non-supervised problem, that is, no ground truth exists. In case that we would like to change the algorithms in the future, we would need to use some common metric, as the ones we used in this iteration or other. This way, we would be able to identify whether the new algorithm is performing better than the current (linear) one or not.

This idea of assigning each developer to a profile gives an idea of its mean characteristics in all his time as a developer in the company. However, we know that the more time a developer spends working and sharing experiences with other mates, the more their coding techniques change. This way, it can also be interesting to identify how the profile of each developer evolves with time or with projects. For this feature business goal, we just need to adapt how data is prepared. That is, we would need to aggregate data by developer and project. Then try to identify to which cluster belongs each person while working on a particular project. It would be very interesting to identify who tends to adapt better or worse practices the more they work on different projects and, that way, try to correct bad conducts.

7.9. Replication package

All of the steps taken in the data description, pre-processing, modelling, and evaluation steps can be reproduced exactly. Most, if not all, of the codebase is based on **jupyter notebooks**, which allow a nice interactiveness and are very descriptive. So, following the structure indicated by the readme file one will be able to replicate our results. Moreover, the required packages and their corresponding versions are indicated in the requirements.txt file. Note that the results that relied on random components, such as initialization, were seeded. Here is the explicit order one would follow as per the names of the folders:

1. **Data Description** (optional, does not affect results)
2. **Data Preparation**. In here there are five things that need to be done:
    a. Select data: select the relevant attributes.
    b. Clean data: deal with the missing values.
    c. Construct data: create new attributes.
    d. Integrate data: join the different tables.

    e. Format data: check the type of all the attributes.
3. **Modelling**: a full step by step modelling guide.
4. **Evaluation**: where we evaluate the different results obtained in the previous step.

An alternative for **2. Data Preparation**, **3. Modelling** and **4. Evaluation** is to run the *.py* scripts included in the package (refer to the readme file).

We have also included in the replication package the final selected model in a *joblib* object: using tSVD for dimensionality reduction and KMeans with 5 clusters. This model could in fact be put in production as an Amazon Lambda function (a serverless auto-scalable solution), but we do not believe it would be of great use: the conclusions stated and the recommendations to the business (and the actions taken by it) are the important part of this study. In the future, one could re-perform our analysis to see if the clusters have changed in a good way (i.e. more clean code, less issues induced).

Annex

Annex 1: Data Description

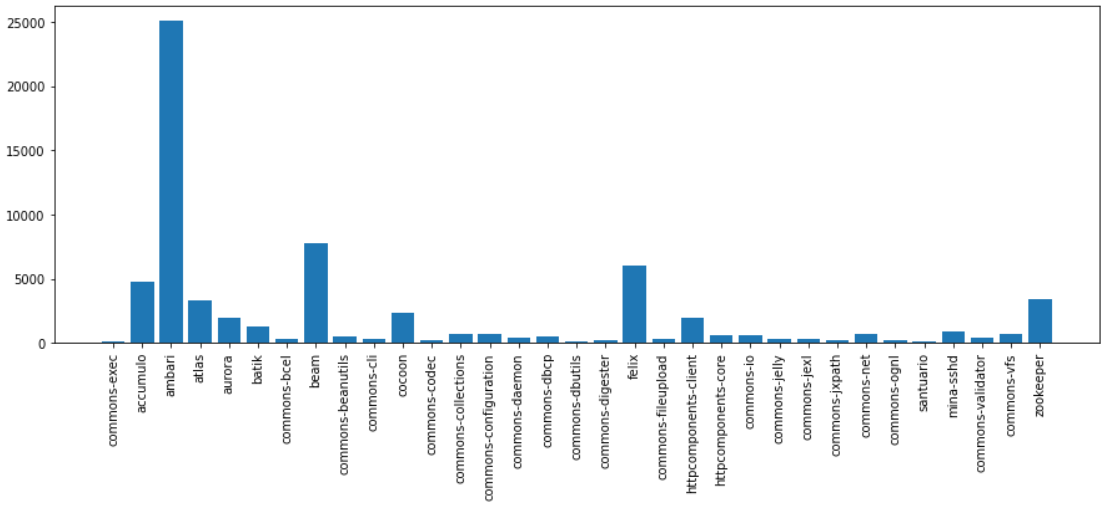
PROJECTS

This table has 33 records, the number of projects analysed in this dataset. The only attribute considered in this table is `projectID` which is a string. And, since this attribute is the primary key of the table, there are no duplicates.

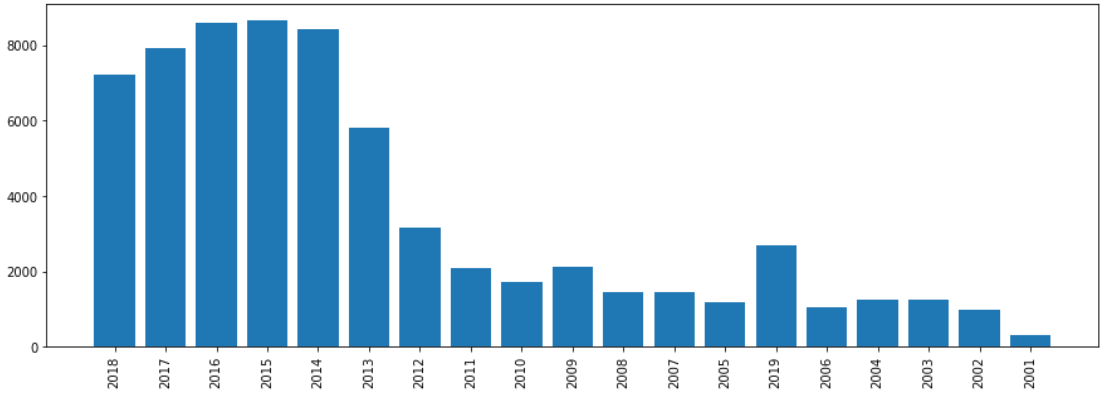
JIRA_ISSUES

This table has 67.428 records. Next, we analyse each of the selected attributes.

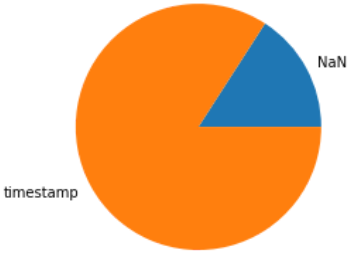
- `projectID` → is a string with 33 different values with the following distribution:



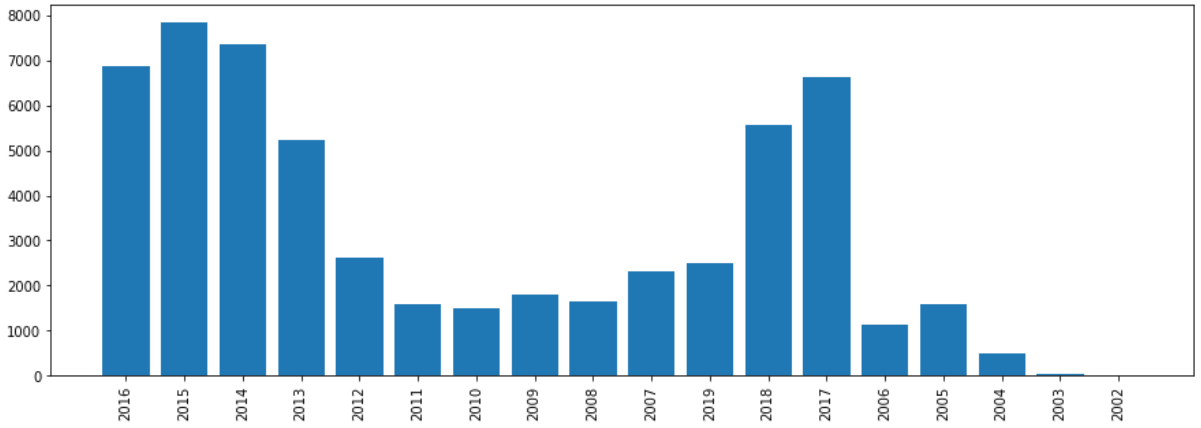
- `creationDate` → is a timestamp that goes from "2001-01-22T14:46:21.000+0000" to "2019-07-16T10:38:29.000+0000" and has the following distribution per year:



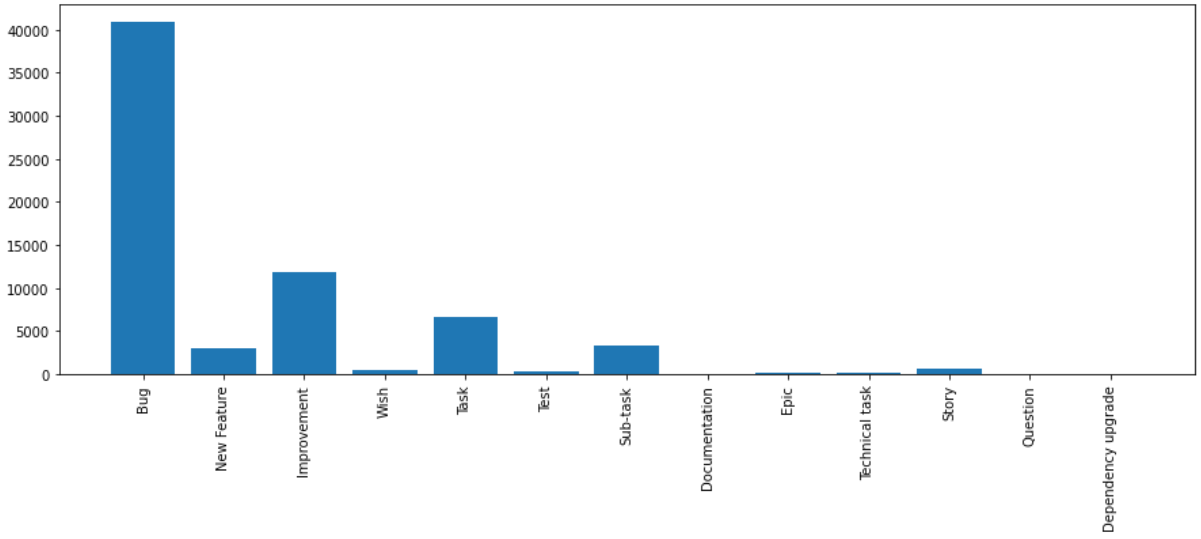
- `resolutionDate` → is also a timestamp but, in this case, the value can be "NaN" if the issue has not been resolved yet. The number of rows with a "NaN" value for this attribute is 10.705. We can see the percentage that this number represents in comparison with the total number of rows in the pie chart.



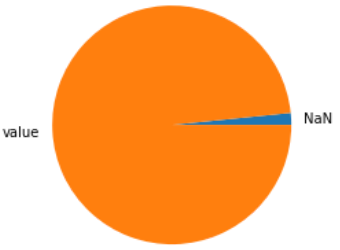
Considering only the rows with value, the minimum timestamp is "2002-08-22T12:26:40.000+0000" and the maximum is "2019-07-16T10:30:04.000+0000". The distribution per year of these timestamps is:

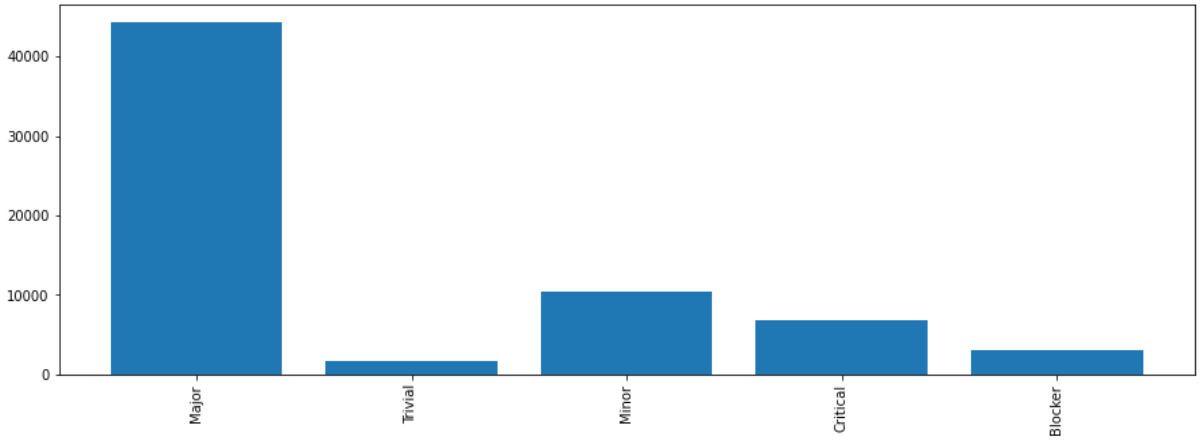


- `type` → is a string which takes 13 different values with the following distribution:

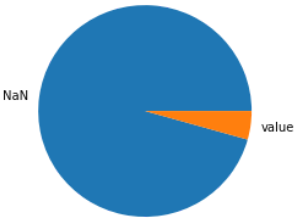


- `priority` → this attribute has also missing values ("NaN"); in particular 1.082 rows are missing this attribute as we can see in the pie chart.
  The rest of the rows that do have a value, it is a string that can take 5 different values following the next distribution:

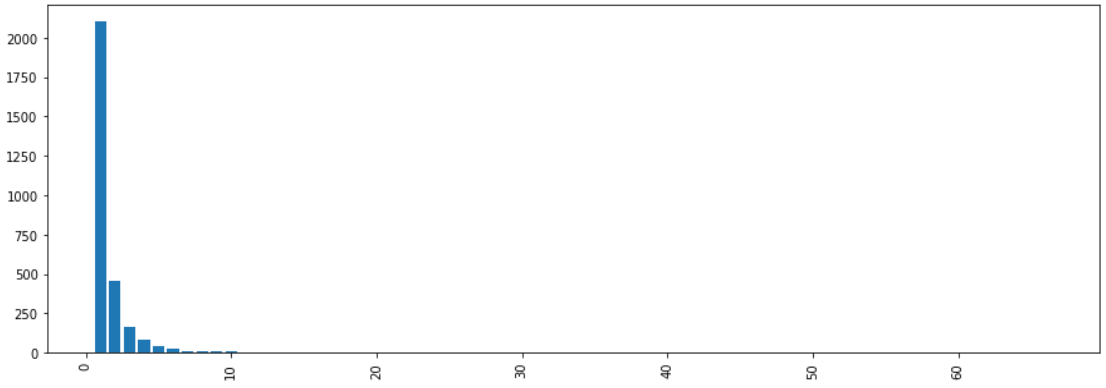




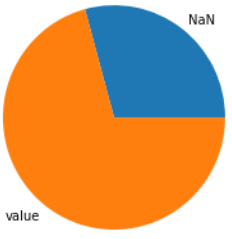
- `votes` → is an integer but most of the values are missing, as we can see in the pie chart. So maybe this attribute will not give us enough information, therefore we will not consider it anymore for our project.



  The values that we do have go from 1 to 66 with the following distribution:



- `assignee` → is a string that takes 1.511 different values but some of the rows have missing values, 19.698 in particular, as we can see in the pie chart.

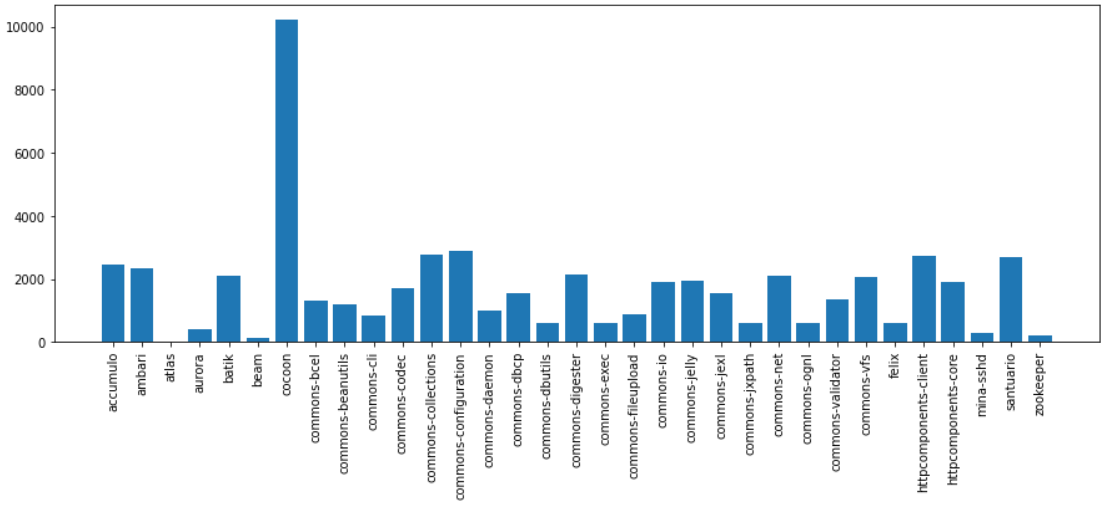


- `reporter` → is a string that takes 10.866 different values and has no missing ones.
  If we compare these last two attributes (because they both make reference to developers), we see that 190 people in `assignee` are not in the `reporter` attribute, so there are 190 people who are working on an issue that have never created a Jira issue.

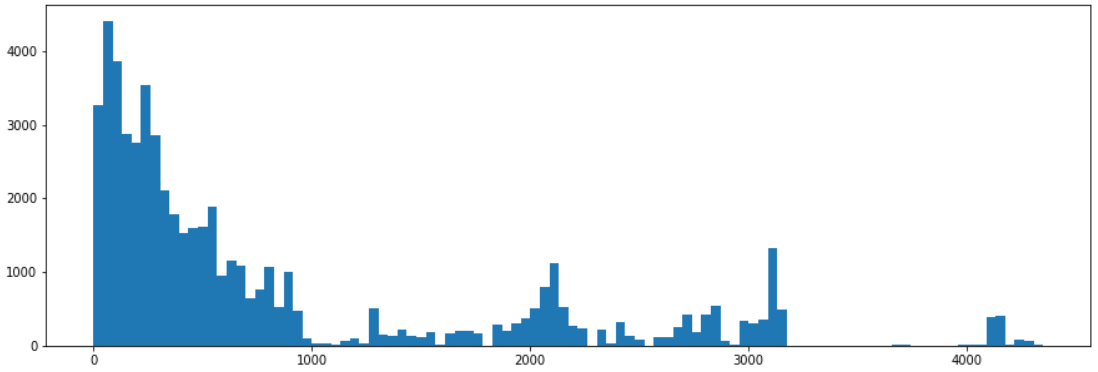
SONAR_MEASURES

This table has 55.629 records. Next, we analyse each of the selected attributes.

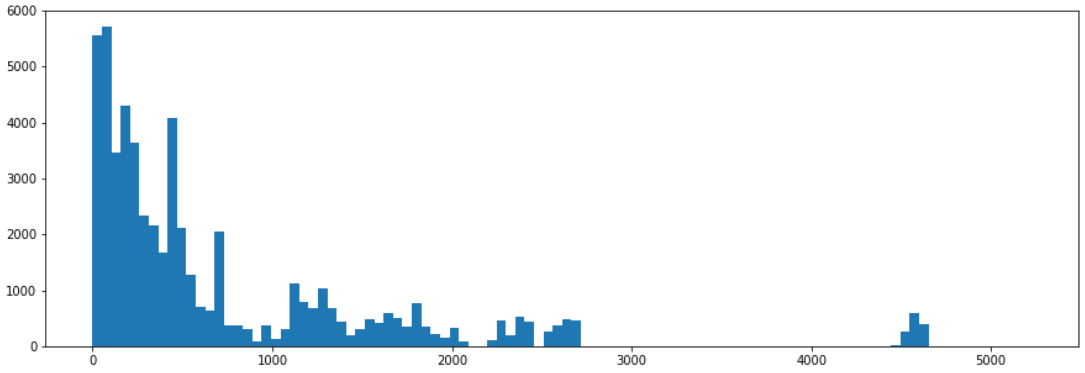
- `commitHash` → is a string with 55.629 different values, so all of its values are unique.
- `projectID` → is a string with 33 different values with the following distribution:



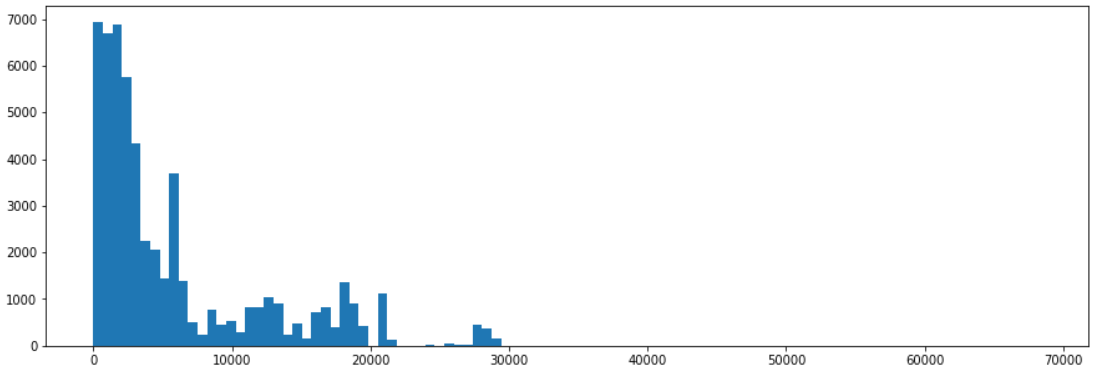
- `classes` → is an integer with 1.981 unique values, none of them a missing value. The values go from 0 to 4.351 classes.



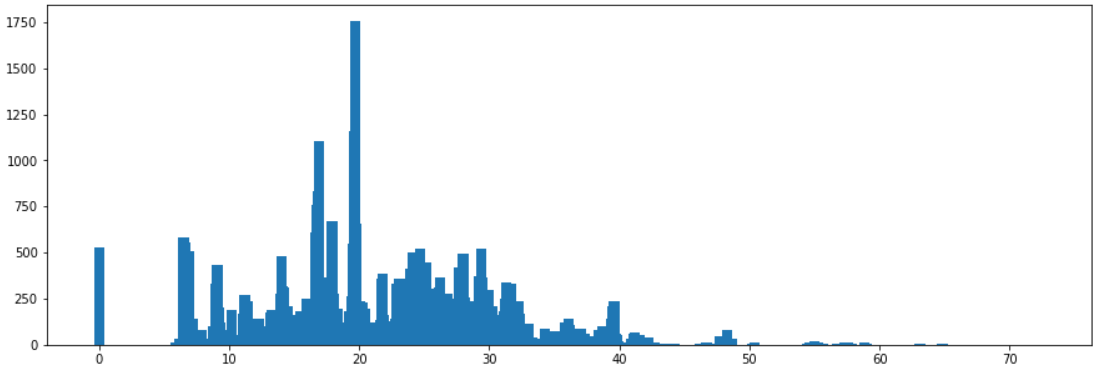
- `files` → is an integer with 1.826 different values, and with any “NaN” as in the previous case. The minimum value is 0 while the maximum one is 5.227.



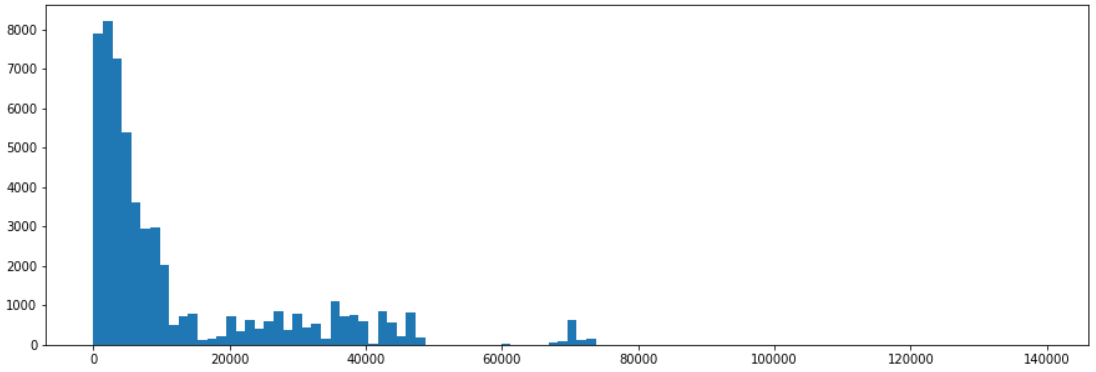
- `functions` → is an integer with 7.526 unique values from 0 to 68.425, none of them a missing value.



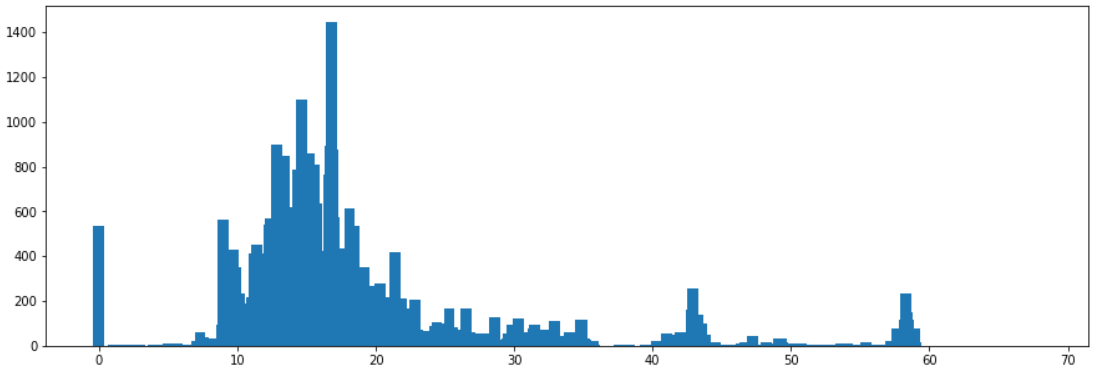
- `commentLinesDensity` → is a float with 466 different values, and with any "NaN". It represents a percentage, so its minimum value is 0 while the maximum one is below 100, in particular, 72,2.



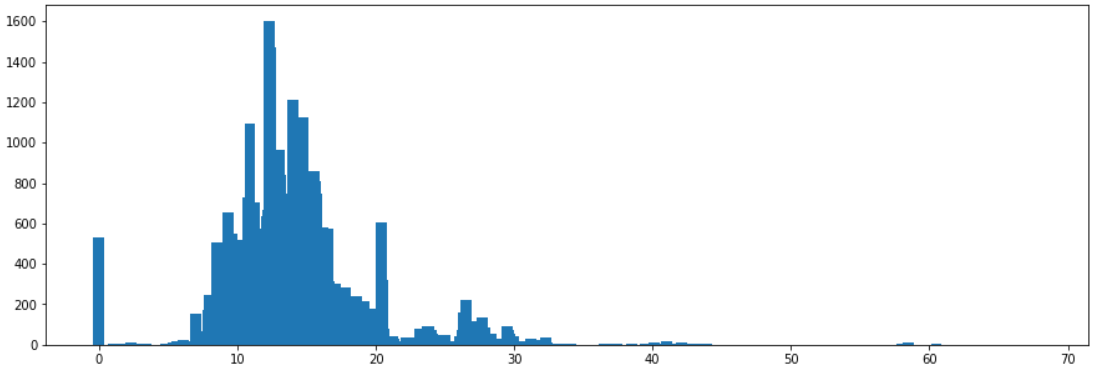
- `Complexity` → is an integer with 11.131 unique values, from 0 to 139.214, and any missing value. The distribution it follows is shown in the following bar plot:



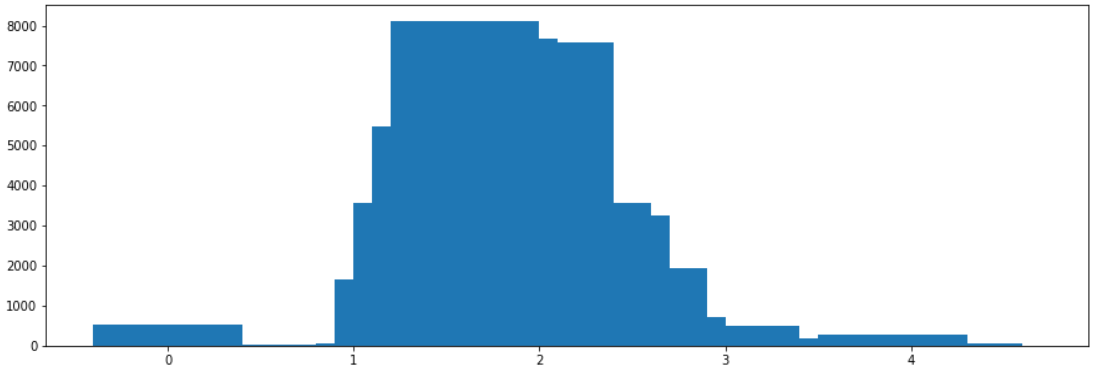
- `fileComplexity` → is a float with 477 different values. The minimum value is 0 whereas the maximum one is 67,7.



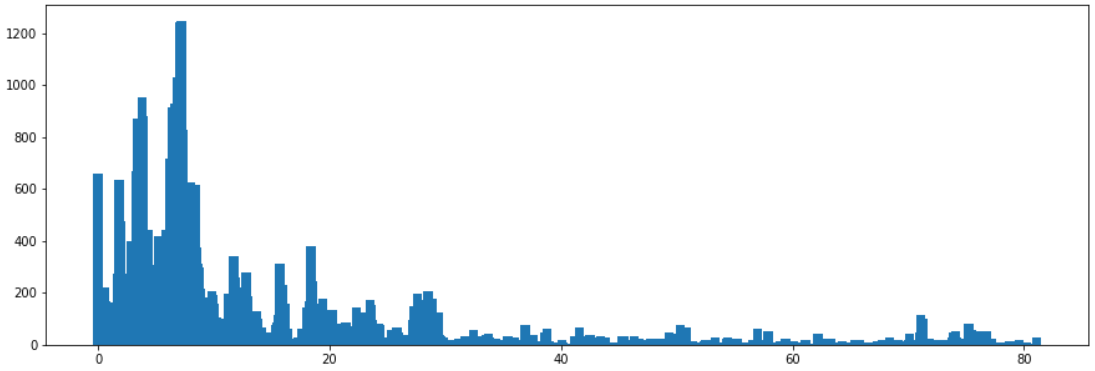
- `classComplexity` → is a float with 351 unique values, from 0 to 67,7, and any missing value.



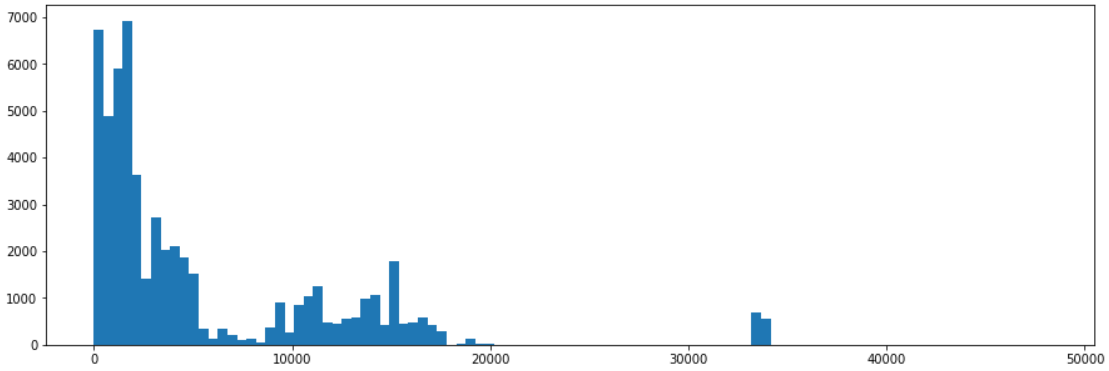
- `functionComplexity` → is a float with 41 different values. The minimum value is 0 whereas the maximum one is 4,3.



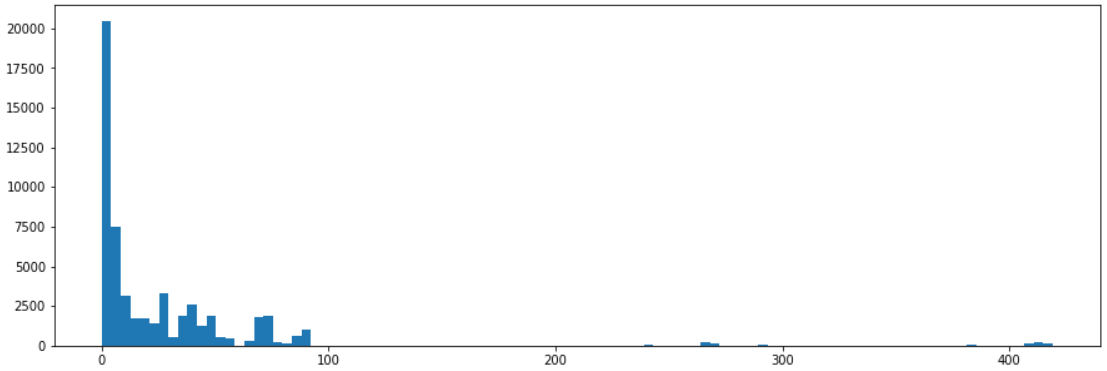
- `duplicatedLinesDensity` → is a float with 762 unique values, none of them a missing one. The values go from 0 to 81,1.



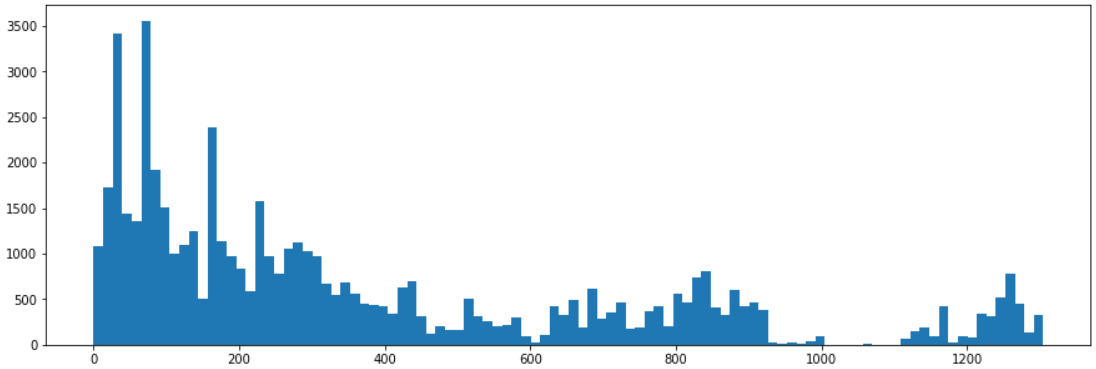
- `violations` → is an integer with 7.909 unique values, from 0 to 48.093, and any missing value. The distribution it follows is shown in the following bar plot:



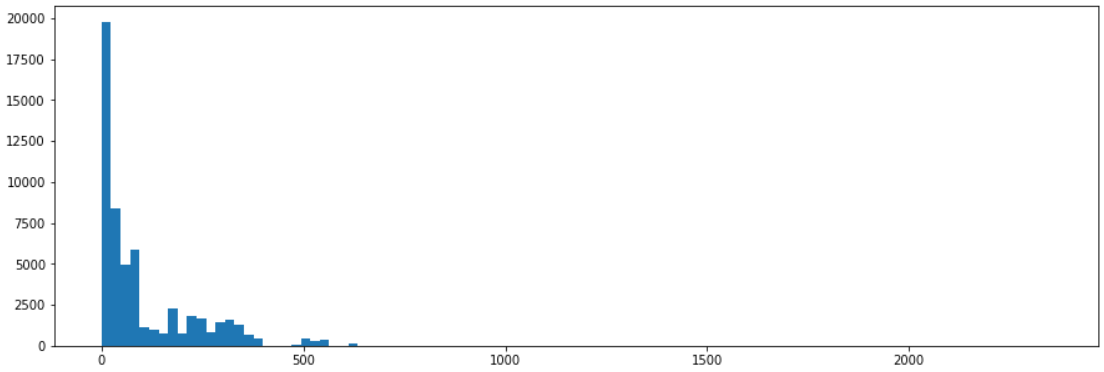
- `blockerViolations` → is an integer with 134 different values, and with any “NaN” as in the previous cases. The minimum value is 0 while the maximum one is 419.



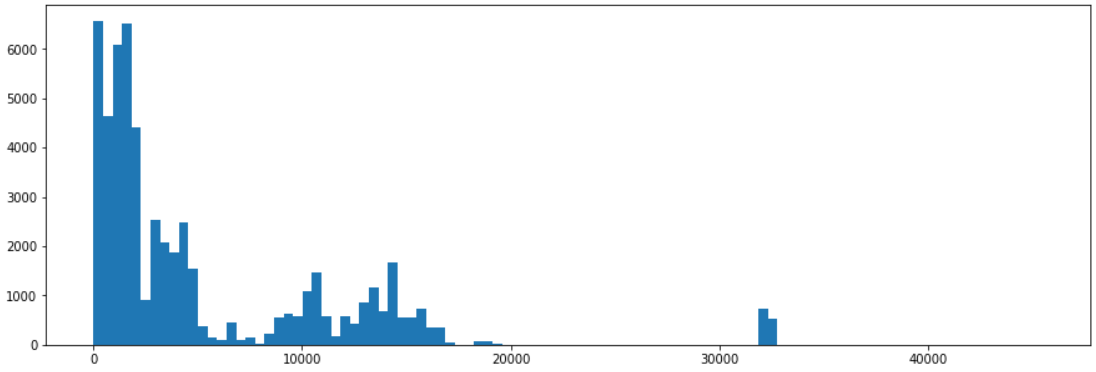
- `criticalViolations` → is an integer with 1.045 unique values, none of them a missing one. The values go from 0 to 1.305.



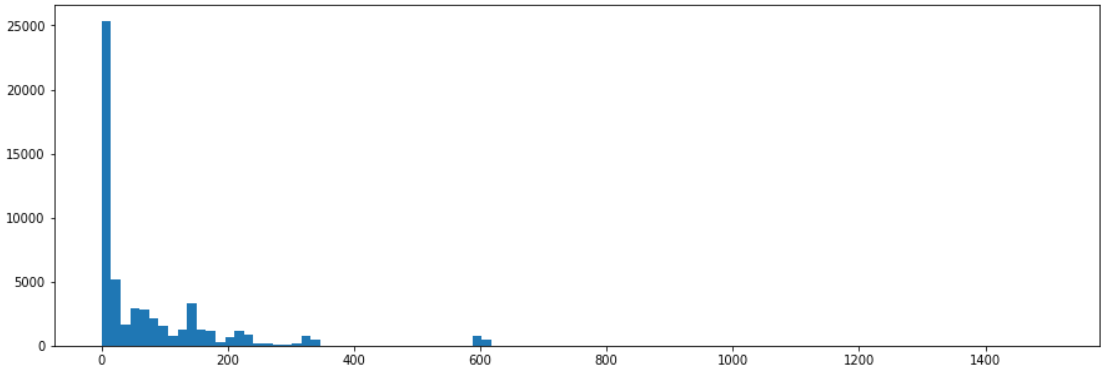
- `infoViolations` → is an integer with 429 unique values, from 0 to 2.351, and any missing value. The distribution it follows is shown in the following histogram:



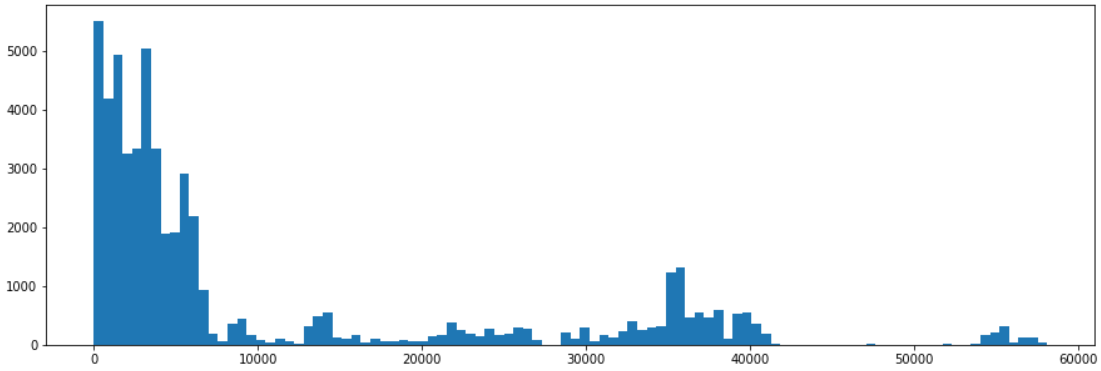
- `codeSmells` → is an integer with 7.739 unique values, from 0 to 45.538, and any missing value.



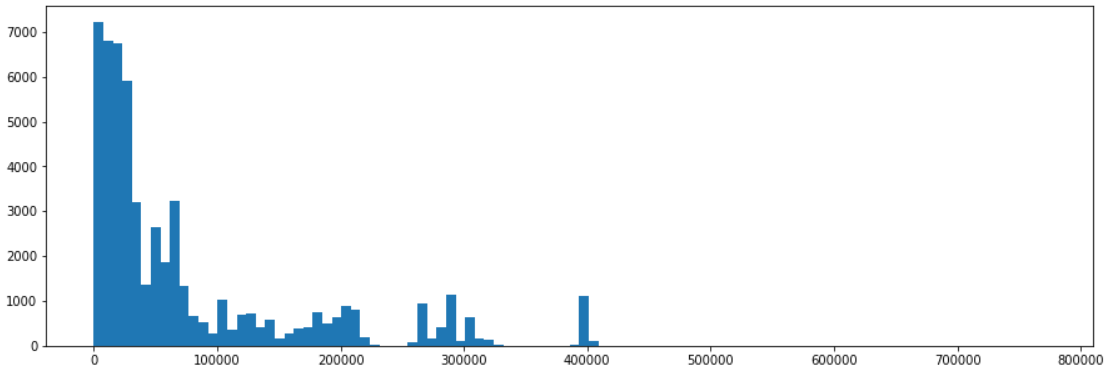
- `bugs` → is an integer with 329 different values, from 0 to 1.505, and any missing value. The distribution it follows is shown in the following bar plot:



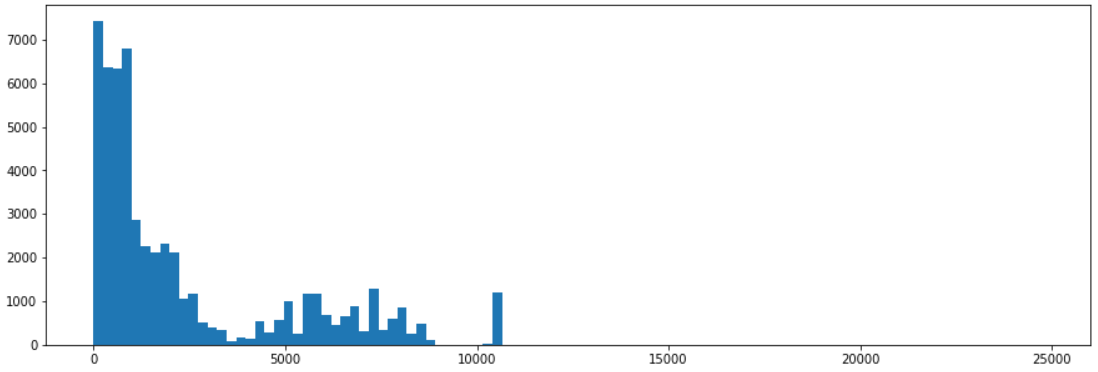
- `cognitiveComplexity` → is an integer with 8.940 different values, and with any “NaN”. The minimum value is 0 while the maximum one is 58.086.



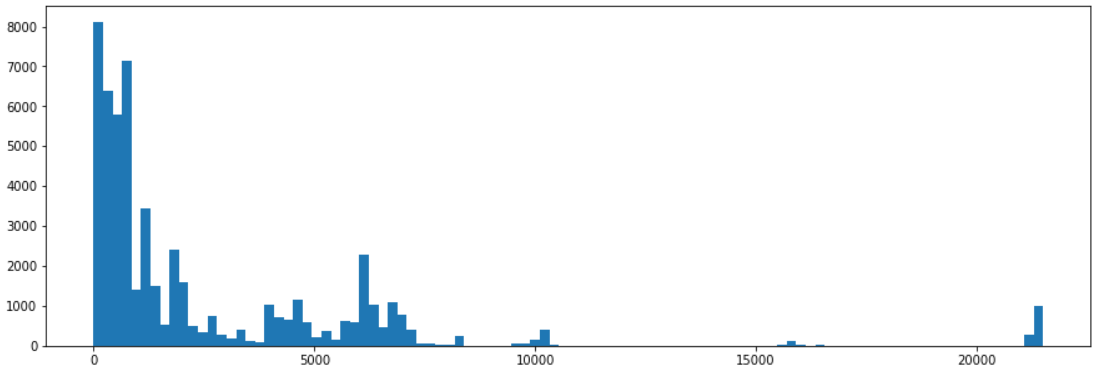
- `ncloc` → is an integer with 21.647 unique values, from 0 to 771.688, and any missing value.



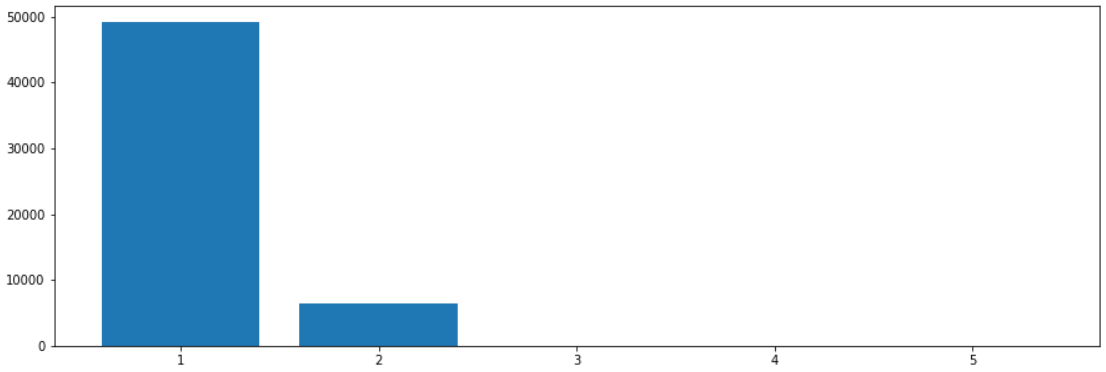
- `majorViolations` → is an integer with 4.777 different values, and with any "NaN" as in the previous cases. The minimum value is 0 whereas the maximum one is 24.772.



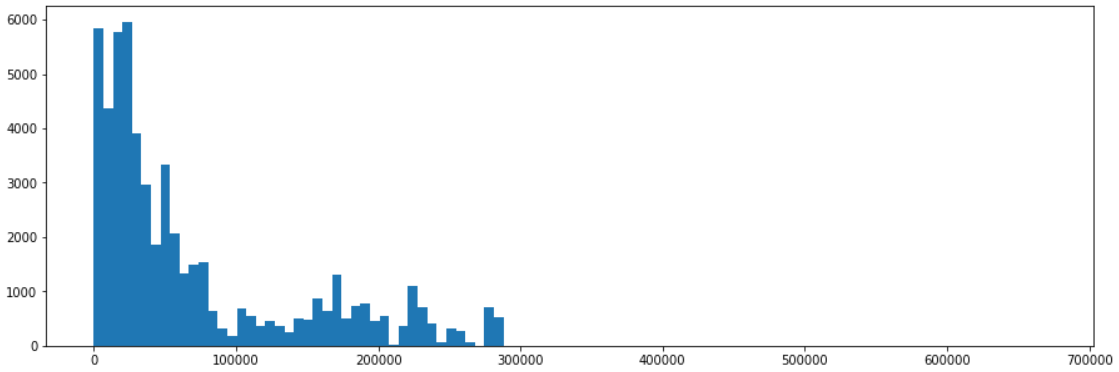
- `minorViolations` → is an integer with 4.402 unique values, from 0 to 21.509, and any missing value. Its distribution is the following one:



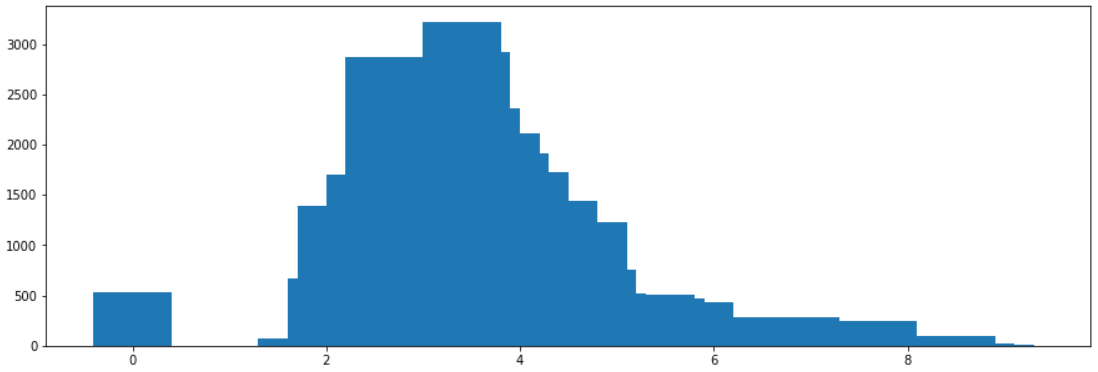
- `sqaleRating` → is an integer with 2 unique values from the 5 possible ones because of the definition of the attribute, and any missing value.



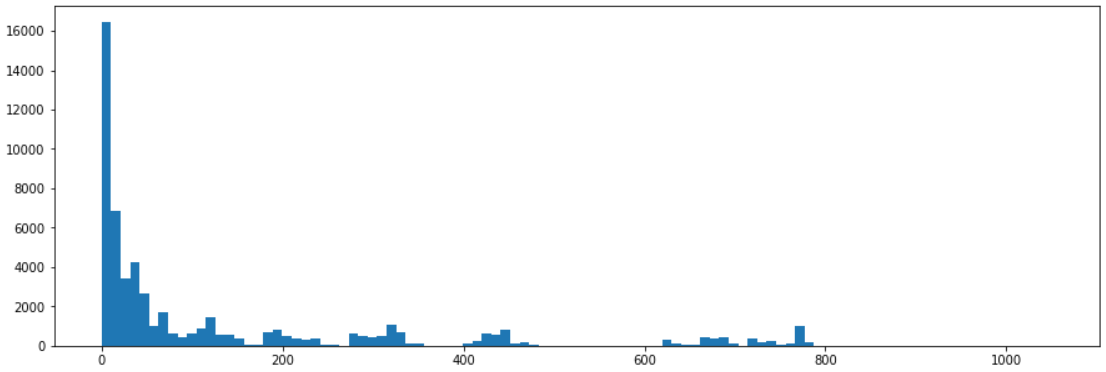
- `sqaleIndex` → is an integer with 17.146 different values, from 0 to 669.813, and any missing value. The distribution it follows is shown in the following bar plot:



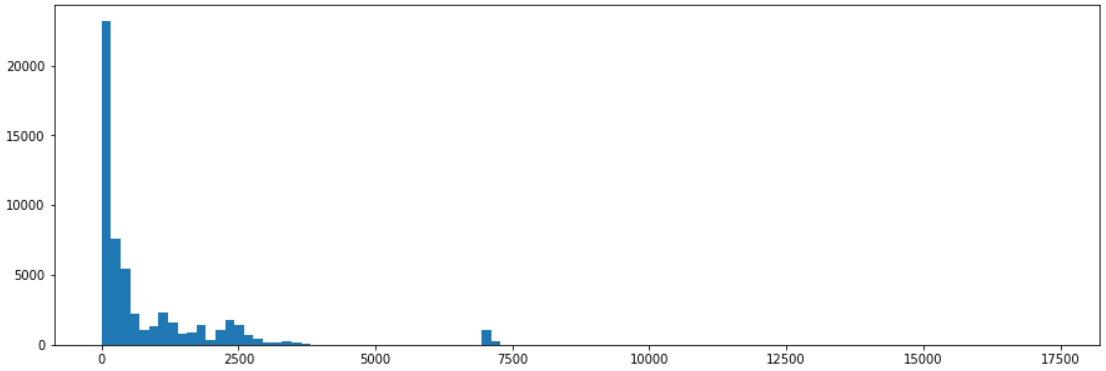
- `sqaleDebtRatio` → is a float with 77 unique values, from 0 to 9, and any missing value.



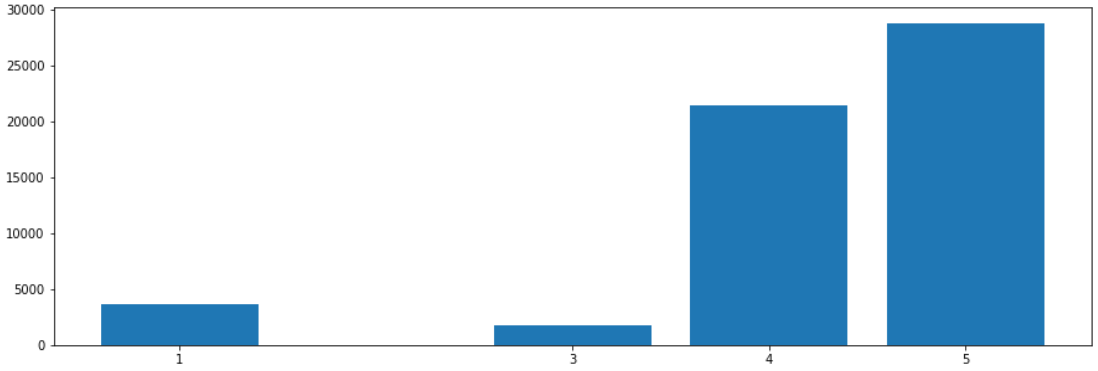
- `vulnerabilities` → is an integer with 419 unique values, from 0 to 1.050, and any missing value. Its distribution is the following one:



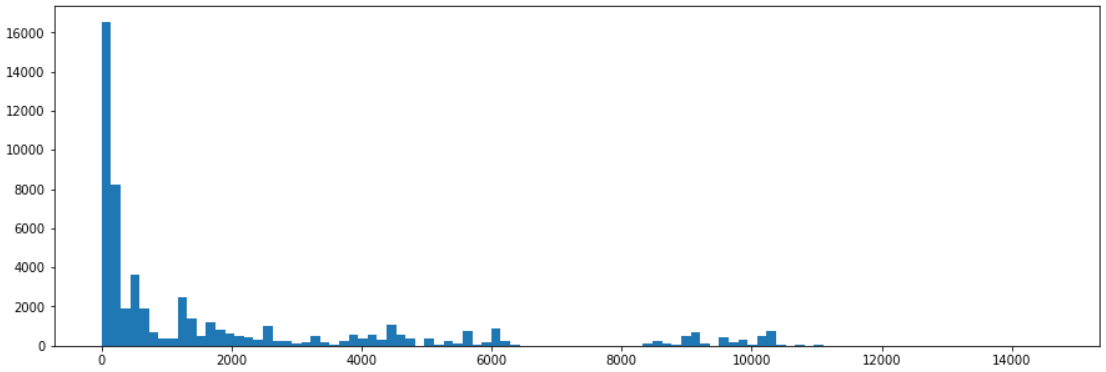
- `reliabilityRemediationEffort` → is an integer with 756 different values, and with any “NaN”. The minimum value is 0 while the maximum one is 17.343.



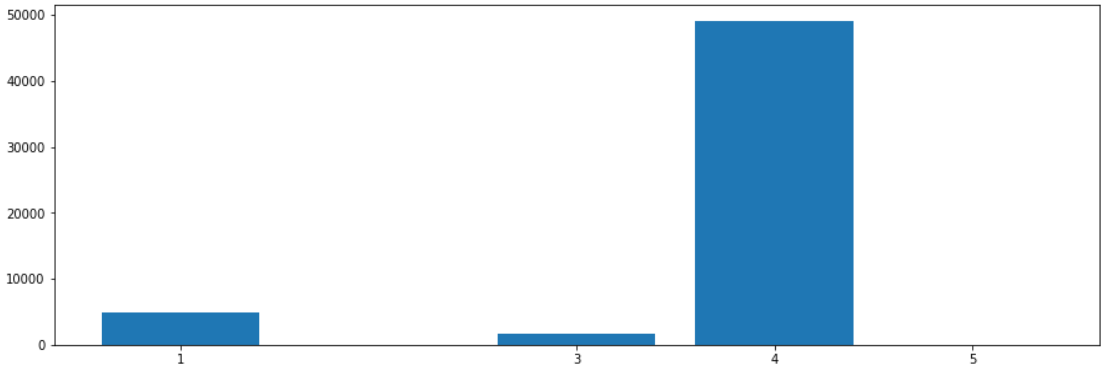
- `reliabilityRating` → is an integer with 4 unique values, from 0 to 5, and any missing value.



- `securityRemediationEffort` → is an integer with 690 different values, and with any “NaN” too. The minimum value is 0 whereas the maximum one is 14.615.



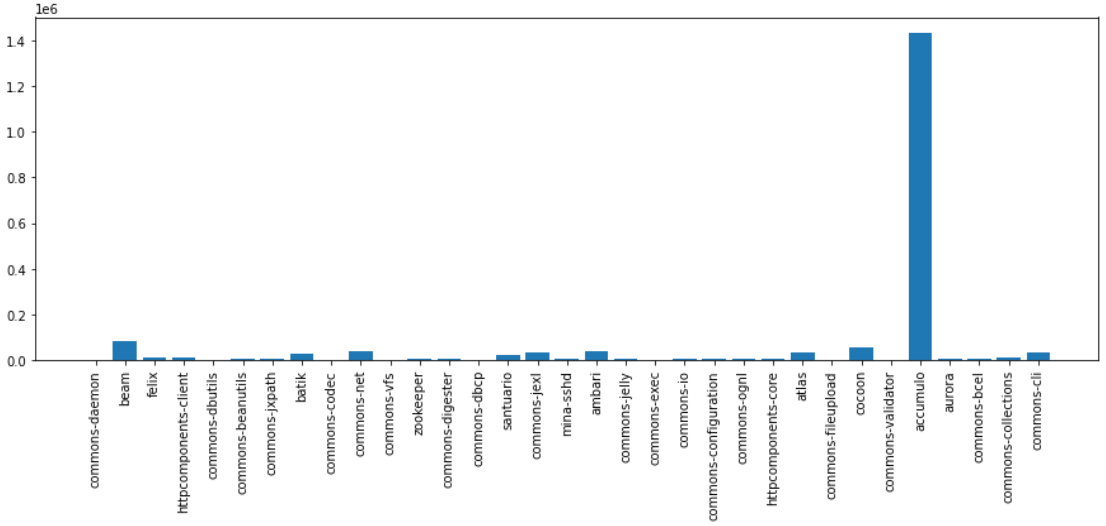
- `securityRating` → is an integer with 3 unique values from the 5 possible ones because of the definition of the attribute, and any missing value.



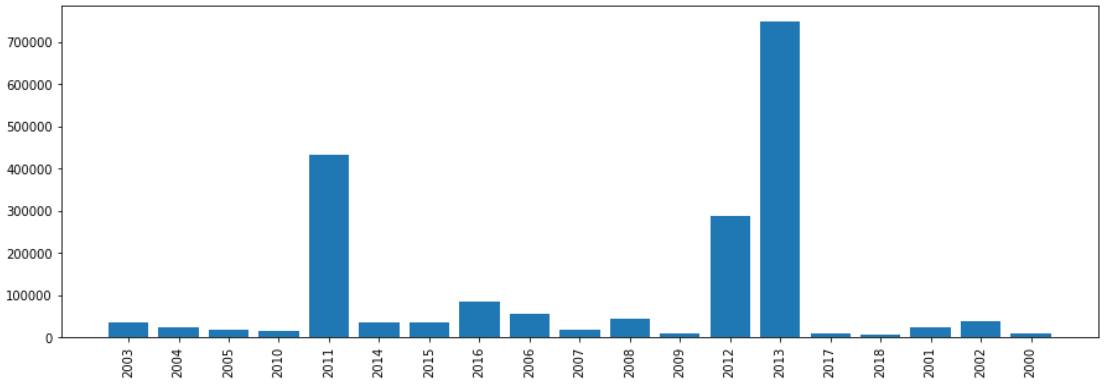
SONAR_ISSUES

This table has 1.941.508 records. Next, we analyse each of the selected attributes.

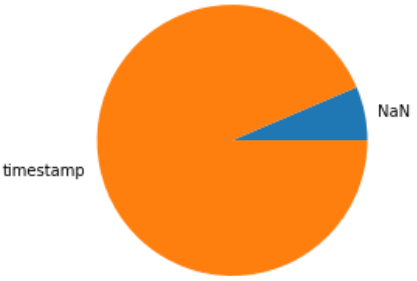
- `projectID` → is a string with 33 different values with the following distribution:



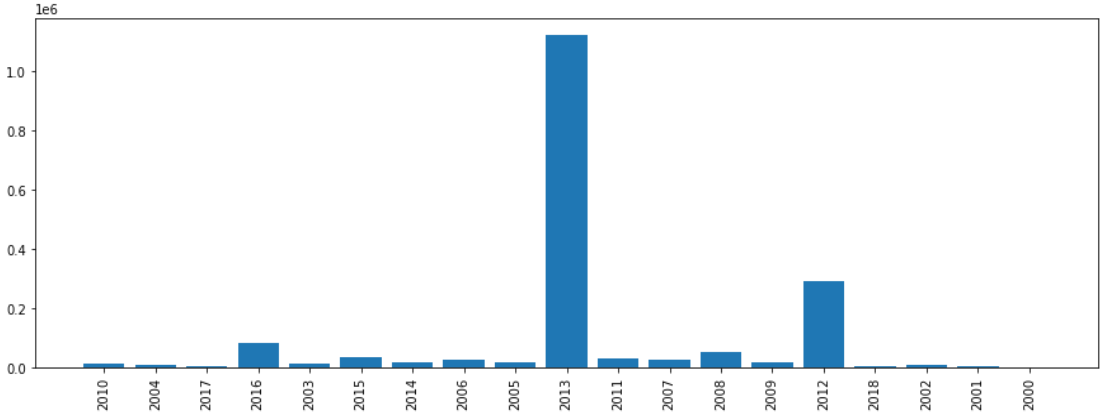
- `creationDate` → is a timestamp that goes from “2000-10-10T18:36:52Z” to “2018-09-19T15:09:16Z” and has the following distribution per year:



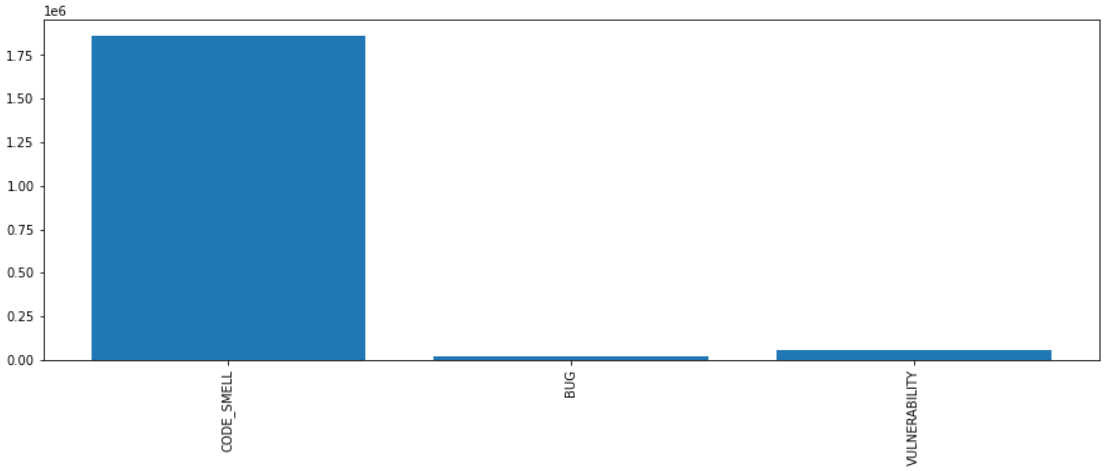
- `closeDate` → is also a timestamp but, in this case, the value can be “NaN” if the issue has not been resolved yet. The number of rows with a “NaN” value for this attribute is 124.457. We can see the percentage that this number represents in comparison with the total number of rows in the pie chart.



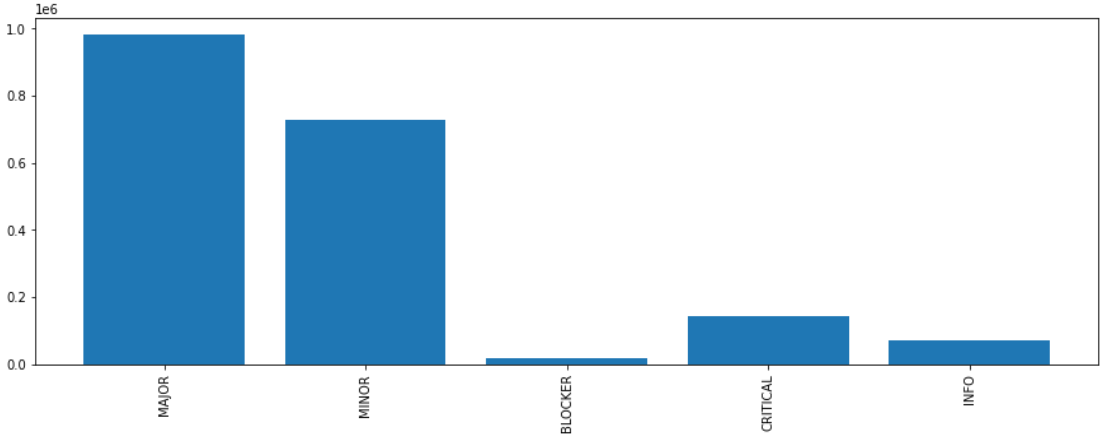
Considering only the rows with value, the minimum timestamp is “2000-10-11T18:54:20Z” and the maximum is “2018-09-19T15:09:16Z”. The distribution per year of these timestamps is:



- `creationCommitHash` → is a string with 19.749 different values.
- `closeCommitHash` → is a string with 127.832 different values.
- `type` → is a string that can take 3 different values with the following distribution:

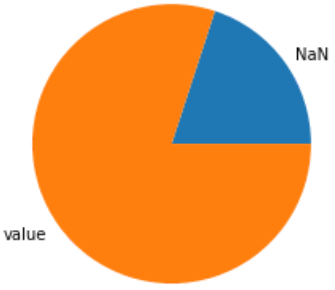


- `severity` → is a string which can take 5 different values and has the following distribution:



- `debt` → is a string that represents a value of time in days, hours and minutes; and it has 375 different values. It has 24.088 missing values, which represents the percentage that can be seen in the pie chart.

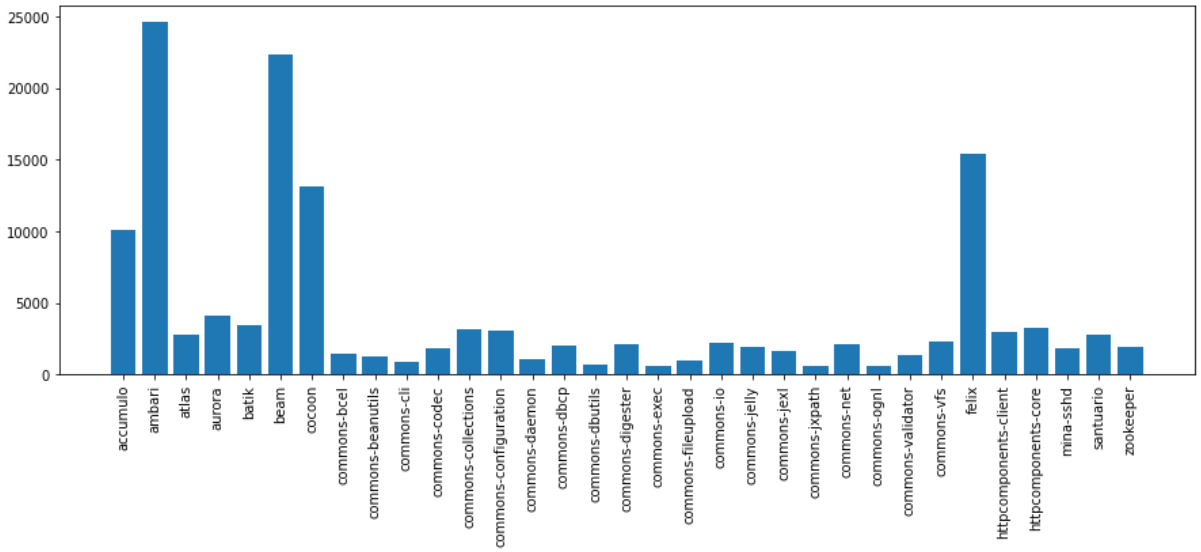
- `author` → is a string which takes 566 different values and has 387.753 missing values, as we can see in the pie chart.



GIT_COMMIT

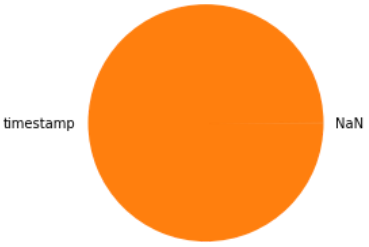
This table has 140.687 records. Next, we analyse each of the selected attributes.

- `projectID` → is a string with 33 different values with the following distribution:

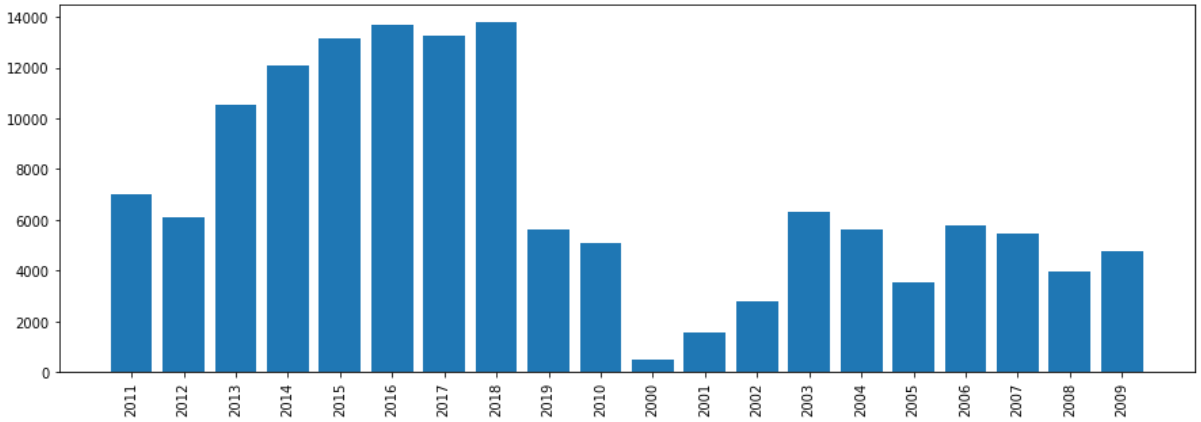


- `commitHash` → is a string with 140.687 different values, so each line has a different value (it is the primary key of this table).

- `author` → is a string that can take 1.884 different values and has 34 missing values. As we can see in the pie chart, 34 is a very small number compared to the total number of records of this table.



- `committer` → is a string that takes 1.017 different values and also has 34 missing values. So, as in the previous case, this number is irrelevant in comparison with the total number of records.
- `committerDate` → is a timestamp with no missing values, that goes from “2000-10-01T07:37:01Z” to “2019-07-19T16:02:58Z” and has the following distribution per year:

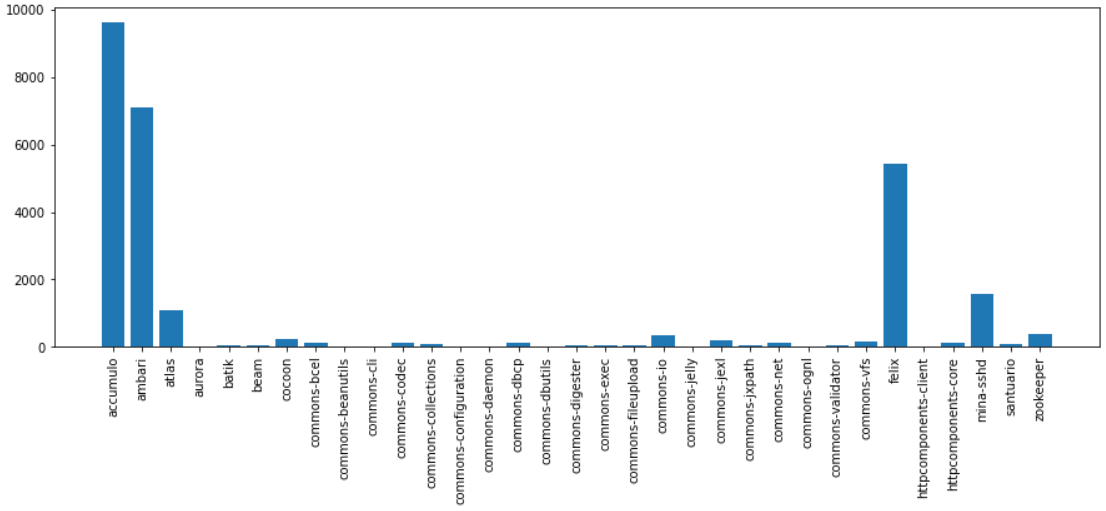


- `inMainBranch` → is a bool variable but in the whole table, it only takes the value `True`, and it does not have any missing value; so it does not give us much information. We may not consider this attribute for our project.

SZZ_FAULT_INDUCING_COMMITS

This table has 27.340 records. Next, we analyse each of the selected attributes.

- `projectID` → is a string with 33 different values with the following distribution:

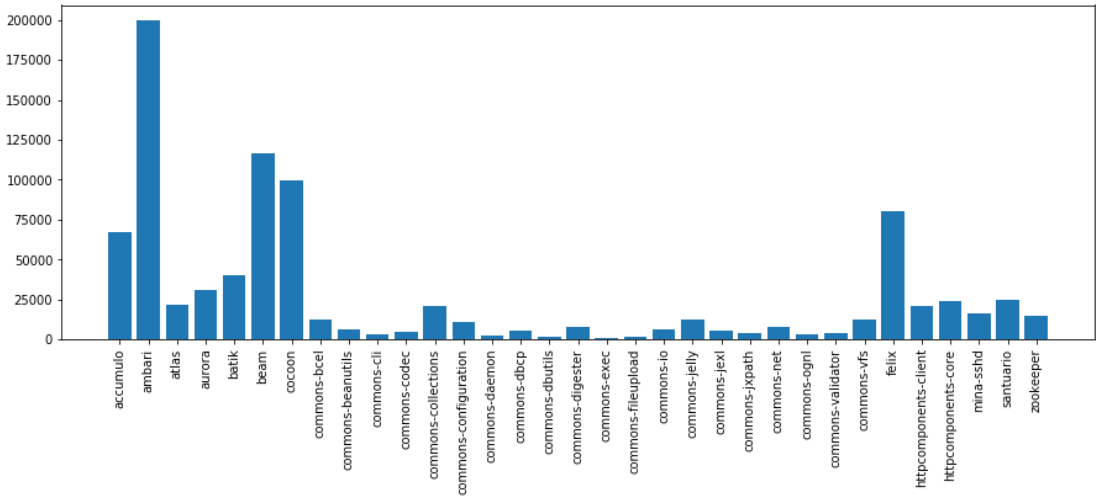


- `faultFixingCommitHash` → is a string with 8.538 different values and any missing value.
- `faultFixingfileChanged` → is a string with 10.132 unique values and this column has no missing values.
- `faultInducingCommitHash` → is a string with 669 different values and, as the previous cases, there are no 'NaN' values.

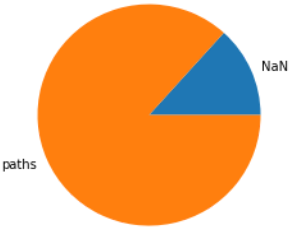
GIT_COMMITS_CHANGES

This table has 891.711 records. Next, we analyse each of the selected attributes. Before starting, we have seen that the distributions are not very realistic because the maximum value is very high compared to the rest of the values, so we can think that there are some outliers. For that reason, in some integer or float attributes, we do not show the distribution plot, because it does not make sense at the moment. This fact will be addressed in future sections, specifically, in the ***2.4. Data Quality***.

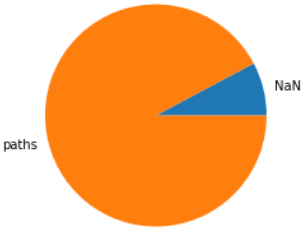
- `projectID` → is a string with 33 different values with the following distribution:



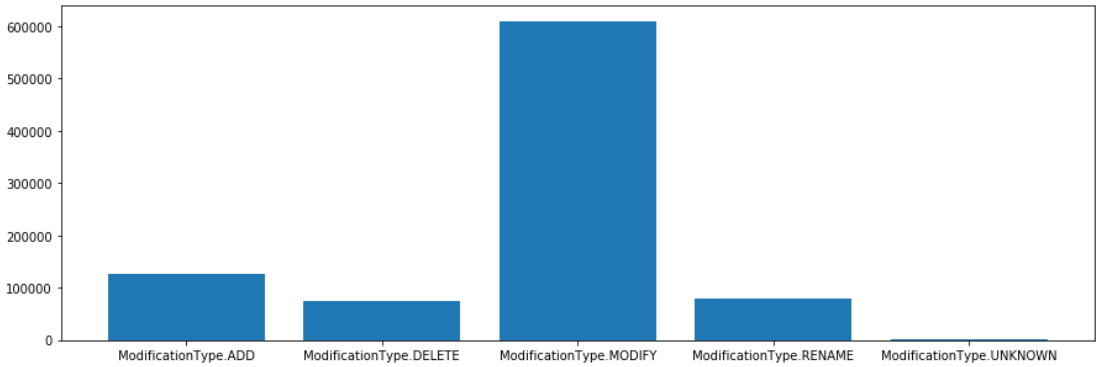
- `commitHash` → is a string with 128.279 unique values and there are no 'Na' values.
- `oldPath` → is a string with 179.239 unique values and it has 118.672 missing values. As we can see in the pie chart, the 'NaN' values do not represent a big percentage of the total ones.



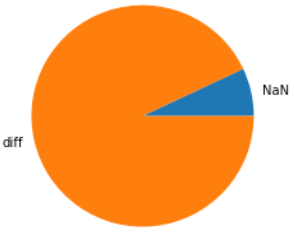
- `newPath` → is a string with 197.247 different values and it has 69.131 'NaN' values, as we can see in the following chart.



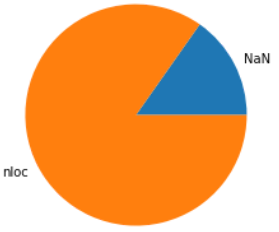
- `changeType` → is a string with 5 unique values and any missing value.



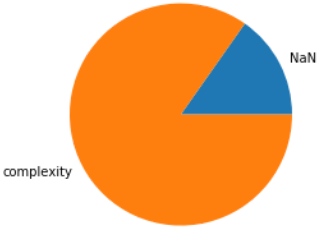
- `diff` → is an object value which can be considered as a string, it has 743.259 unique values and 62.389 rows have a missing value at this attribute.



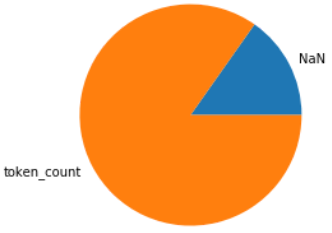
- `linesAdded` → is an integer with 2.565 unique values and 0 missing values. Its minimum value is 0 while the maximum one is 478.967 lines removed.
- `linesRemoved` → is an integer with 2.407 different values and there are no 'NaN' values.
- `nloc` → is a float with 5.373 unique values and there are 135.938 missing values as we shown is the pie chart. The maximum value is 478.110.



- `complexity` → is a float with 1.191 different values and there are 135.938 rows with a 'NaN' as the value of this column.



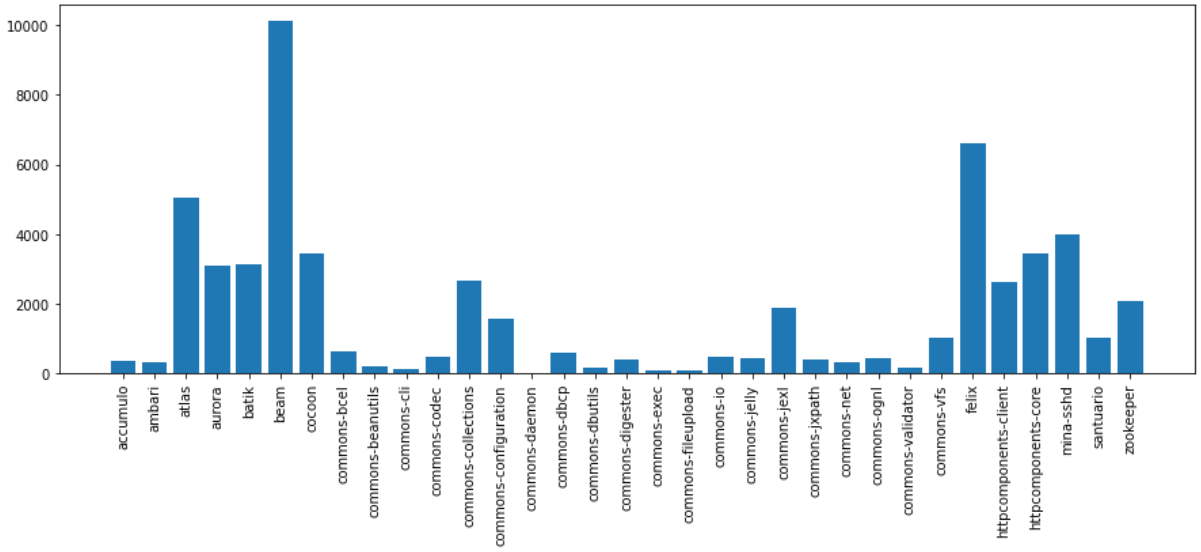
- `tokenCount` → is a float with 19.893 different values and 135.938 missing values. Its maximum value is 4.534.351.
- `methods` → is a string with 130.355 unique values and with no missing ones.



REFACTORING_MINER

This table has 57.530 records. Next, we analyse each of the selected attributes.

- `projectID` → is a string with 33 different values with the following distribution:



- `commitHash` → is a string with 11.698 different values with 2 missing values.
- `refactoringType` → is a string that takes 29 different values and has 2 NaNs. The distribution of these 29 values is the following one: