EPAA - Using commit messages to get insights

Authors: Pau Bernat Rodríguez, Elías Abad Rocamora, Andrea Garcia Valdés, Àlex Martí Guiu

Work tracker
Github repository

Abstract: *Nowadays tools for static code analysis to evaluate the quality of code in projects have risen in terms of popularity as Technical Debt has been an increasing problem in software development. We have used the TDD dataset to analyze how much information can be extracted from git commit messages, since this text is mandatory in a commiting process that most software developers use when pushing or pulling code. Commit texts in itself are data that contains explanations and information inside, that hasn't been thoroughly explored and is often overlooked. In order to prove this point and use this powerful source of information we attempt to get meaningful insights from commit messages of the TDD.*

2.1. Business understanding

2.1.1. Business objectives

- Background

Software development is nowadays done in a very analytical way. Everything from number of erroneous builds or number of commits to more complex metrics as code complexity is measured. This allows developers and companies to extract insights of the project evolution, helps developing projects of better quality and ensures they are successful and maintainable.

There are many software development and analysis tools available in the market, some of them are: SonarQube, Github, Jira or Jenkins. These platforms are a very important source of information for developers. In this project we will focus on analyzing a particular source of information that has not gotten much attention until now: git commit messages.

Github is the world's largest repository hosting app and the most used coding platform in the world with 400.000 repositories and 1 billion files. Github allows developers to perform the git version control of their project on the cloud, manage issues, perform collaborative integration and almost every software development task. Github commit messages contain meaningful information for developers about what changes were introduced in that commit, but this information is not easy to analyze automatically by a computer. Our goal in this project is to analyze these texts with novel data science techniques and provide developers new insights about their projects.

- Business goals and success criteria

The main goal of this project is to extract new insights from git commit messages to help the retrospective analysis of a project one is finished, but also analyze the evolution of the project while it is being developed. In order to get these insights, we want to answer some questions like: Is there information in the commit message about the bugs in the code? Can we segment authors in a project based on how they write their commit messages? Is this segmentation related to the quality of their commits? Can we detect outlying commit messages or authors based on commit messages?

Our success criteria will be to define if meaningful data can be extracted from commit messages. This will be assessed by our teacher, class colleagues and members of our own group. Furthermore, if meaningful data can be extracted, another success criteria will be to develop a proper analysis tool to automatically process commit messages and visually analyze them.

2.1.2. Assessment of the current situation

- Inventory of resources

In this section, we will define the resources we need to carry out this project. First of all, it’s essential to have a work team in charge of making the project a reality. Our team is made up of four junior data scientists: Elias Abad Rocamora, Àlex Martí Guiu, Pau Bernat Rodríguez and Andrea Garcia Valdés.

All of us are fourth-year students of data science and engineering degree at Universitat Politècnica de Catalunya (UPC). Our expertise includes data mining techniques, machine and deep learning, statistics and programming. We also have some experience developing projects using an agile methodology as well as some business knowledge acquired by completing an entrepreneurship course. Each member of the team will assume part of the tasks every week so that we will all participate in the work at the same level. We also have the supervision of Silverio Martínez, a professor at UPC, who will advise us and help us.

In order to develop this project, we need a dataset to extract the input data and exploit them to achieve our goal. We will work with "The Technical Debt Dataset, Version 1.0" built by Valentina Lenarduzzi, Nyyti Saarimäki and Davide Taibi, presented in the ‘’The Technical Debt Dataset. Proceedings for the 15th Conference on Predictive Models and Data Analytics in Software Engineering. Brazil. 2019’’ paper. This dataset contains measurements data from four tools executed on all commits of 32 projects. We downloaded the dataset in .csv format, but it’s also available as a SQLite Database with the .db extension, a specific format for working with SQL.

Furthermore, the hardware that we will use are our own laptops. In principle it should be enough to carry out the tasks but if at any time we need more computing capacity we have the possibility of requesting a server from the faculty.

Finally, in terms of software we will use Python for programming and developing our code and Google Collab or Jupyter Notebook to create the different Python files. To manage all the documentation we have a repository on GitHub, where we will upload any new or updated file. In order to manage the communication between the members of the team and do coworking, we will use Discord, where we will have our meetings. To document we are using this Google Docs uploaded on Google Drive.

- Requirements, assumptions, and constraints

The requirements of the project are:

- The project must be finished before the deadline - October 28th, and the tasks have to be completed according to the project schedule and the weekly milestones.
- The final results must satisfy the success criteria established in both, business and data mining terms.
- The project must follow the CoookieCutter Data Science project structure, use Static analysis to check code quality and work against a shared backlog with the help of Trello.

To develop the project we make the following assumptions:

1. The dataset should be coherent and veridical enough to draw accurate conclusions that can be seen reflected in reality.

Finally, we also consider some constraints:

1. The project only has 4 people working, who will dedicate about 60 hours each, which limits the amount of work dedicated to the development of tasks.
2. The data is limited as well as the resources that we have available, for instance, the storage capacity.
3. Techniques that the team members know and have worked with will be used, it may be that at some point the data mining objectives cannot be achieved due to limited knowledge.

- Risks and contingencies

The biggest risk assumed in this project is the fact that there may not be a real relationship in the data, such as there is no correlation between the commit text and the authors or the commits text are too random. If this is the case, the action we will take will be to change the focus of the objectives and try to solve a more achievable and reasonable task with the commits text. Besides, it would be possible to obtain meaningful insights but with low accuracy. In this case, we should evaluate options to increase the precision following other methods, and if it is not possible, establish how significant the result is.

In addition, we are aware of the pandemic situation which forces us to be willing to do complete telework if necessary, assume the tasks of another member if one is in a high-risk situation caused by the covid

- Terminology:
    - RNN recurrent neural network
    - LSTM long short term memory network
    - GRU gated recurrent unit
    - TDD technical debt dataset
    - NLP natural language processing
    - ML machine learning, DL deep learning
- Costs and benefits:

- Costs:
    - 30€/h junior engineers
    - Servers
    - 7 weeks of work
- benefits:
    - Give insights that are meaningful for project development
    - Resolve conflicts in the development phase.
    - Establish a precedent for future work on commit message analysis.

2.1.3. Data mining goals and success criteria

The main goal of this project is to obtain valuable insights for Github from the unstructured textual data found in the Technical Debt Dataset. Specifically, our main goals of the project are:

- Finding the relationship between the commit text, where developers briefly describe what changes have they made in the code of a project, and if this commit caused the code to crash predicting the error severity.
- Finding the relationship between developers based on their commit texts and also relate this information to how often their commits cause the code to crash.
- Analyzing developers that write commits in a very distinguished manner (outlier authors) and also in detecting which authors tend to write misleading commits.

It is important to mention that the goals are substrate to modification or even fully removal as their feasibility is further tested. This could be caused due to a wide variety of reasons but the main one is that the size or quality of the database may be insufficient to provide significant results.

Based on the previous goals, we defined the following success criteria for each of the specific goals:

- Being able to identify errors added to the code in the last push only by only using the commit text with at least 70% accuracy.
- Being able to differentiate authors between 2 defined and balanced groups of developers.
- Being able to identify some developers that write commits in a very distinguished manner and some developers that tend to write misleading commit texts (and assess that in some kind of manner).

2.1.4. Project plan

In order to conduct the project smoothly we have defined an initial project plan. It consists of around 8 hours of weekly work on a project that will last 7 weeks.

Steps to be performed during the project:

The following steps are initial and may be subject to changes as we will follow the agile methodology of revising timings and goals and change them whenever necessary.

- **Stage 1**: Business understanding:
    - Duration: 1 week, 20-26 sept.
    - Resources required: Project team, stakeholders.
    - Inputs: Project goal & abstract
    - Outputs: Business Objectives, Data Mining Goals and Project Plan
    - Dependencies: None
- **Stage 2**: Data understanding:
    - Duration: 1 week, 27-3 oct.
    - Resources required: Dataset
    - Inputs: Dataset
    - Outputs: Dataset with its thorough understanding
    - Dependencies: Stage 1.
- **Stage 3**: Data preparation:
    - Duration: 2 weeks, 27-10 oct.
    - Resources required: Dataset that has been previously understood
    - Inputs: Dataset
    - Outputs: Meaningful Data without redundancies and with the format needed to model.
    - Dependencies: Stages 1-2
    - Risk: This stage takes longer than expected and modeling gets delayed
        - Solution: Start modeling iterations while data is still being further prepared if needed.
- **Stage 4**: Modeling:
    - Duration: 1 week, 11-17 oct.
    - Resources required: Computing Power, Curated Dataset, ML knowledge.
    - Inputs: Curated Dataset
    - Outputs: Results with insights given data.
    - Dependencies: Stages 1-3.
    - Risk: Models get poor results that lead to non meaningful insights
        - Solution: Refocus the project or choose alternative techniques.
- **Stage 5**: Evaluation:
    - Duration: 2 weeks, 11-24 oct.
    - Resources required: Business success criteria
    - Inputs: Model Results
    - Outputs: Metrics and decisions.
    - Dependencies: Stage 4.
- **Stage 6**: Deployment:
    - Duration: 1 week, 25-7 nov.
    - Resources required: Client interface and workspace
    - Inputs: Trained models and full data processing pipeline
    - Outputs: Integrated pipeline in client's workspace
    - Dependencies: Stages 1-5.

Evaluation strategy used in terms of project plan: Work is on time and evaluation metrics are met when the models are trained and tested.

Initial assessment of tools and techniques:

- Use **agile methodology** to cowork efficiently and with constant communication.
- Using Python and R to develop the code and algorithms.
- GLM & PCA for attribute correlation study and redundancy analysis.
- Most suitable technique we will use in our big model:
    - LSTM for error identification given text from commits.
- Other options we might explore:
    - GRU neural network model.
    - Text to image and CNN models
    - MLP, SVM, kernels, etc

2.2. Data understanding

2.2.1. Initial data collection

As we mentioned above, the data used on this project is collected from Tech Debt Dataset. To obtain the data, we downloaded each table locally in .csv format. The data is also available in .db format, but as we want to use Python to process it, it would be easier for us to have one .csv file per table.It is worth mentioning that once the project started the dataset was updated, and the new data was provided just in .db format, so we implemented a code to convert the .db to .csv to adapt the format to which we were using.

Once we have the data, we decide which table will be useful for our tasks. We remember that our goal is to be able to get insights from the commit texts. The tables that contain the necessary measures to achieve our goals are: GIT COMMITS and SONAR MEASURES. These tables are connected with the SONAR ANALYSIS data, so it might be possible to also need this table.

2.2.2. Data description

The dataset we are working with consists of a Technical Debt analysis using 33 Java projects, where the creators have analyzed 78K commits, detecting 1.8M SonarQube issues, 38K code smells, 28K faults and 57K refactorings. The dataset is made available through CSV files (one for each table) and an SQLite database, which allows for a great on possible analysis of the data.

The main tables we are working with are:

- Commits table: 81072 rows × 13 columns
    - Commit Hash: string
    - Commit Message: string
    - Author & Committer: string
- Sonar Measures: 66711 rows × 240 columns
    - Project Id: string
    - Analysis key: string

    - Complexity: float
        - Mean: 18026.58, Var: 791117703.26
        - Max: 147175.0, Min: 0
        - Number of different values: 15298,
    - Increment in Complexity
    - Increment in Violations
    - Increment in Development Cost
    - Other complexity and class issues (we will probably not use them)
- Sonar Analysis: 67550 rows × 4 columns
    - Project id: string
    - Analysis key: string
    - Date: date
    - Revision: string

To sum up, we see how the data fulfills the requirements we have previously set for our project, and allows for combining the tables to get a single table with the attributes needed to train our models.

2.2.3. Data exploration

In this section we will be exploring the two main tables that we will be using which are GIT_COMMITS and SONAR_MEASURES. We also checked the SONAR_ANALYSIS table but in the context of our project it only makes sense to analyze the first two as this table only serves as a link between the two mentioned tables. Before starting the exploration it is important to understand which are the most important variables for us:

- The most important variable is the COMMIT_MESAGE variable found in the GIT_COMMIT table, which contains the text from each commit. This is the variable that we will use to predict the rest of variables.
- The variables that are candidate to being predicted are found in (or derived from) the SONAR_ANALYSIS table and some of the more important are the increase of bugs, violations, open issues or development costs after a commit.

Starting with the GIT_COMMIT table, we firstly saw that the table had mostly categorical variables which was the main reason why we decided to aggregate some of the variables in order to get a clear view of the table. Firstly, we can see that there are 31 projects uploaded in this table with an average of 2615 commits per project. The average team has 34 members, with the smallest team having 5 members and the biggest team having 132. If we observe the distribution of both these variables (the size of the team, and the numbers of commits per project) we observe that the most frequent size for a team is between 20-40 members, even though there are some smaller teams and teams with 40-80 members. The majority of teams execute less than 2000 commits per project, although there is a big group of teams that execute 2000-4000 commits. Lastly there are 2 groups with more than 12000 commits in their project.

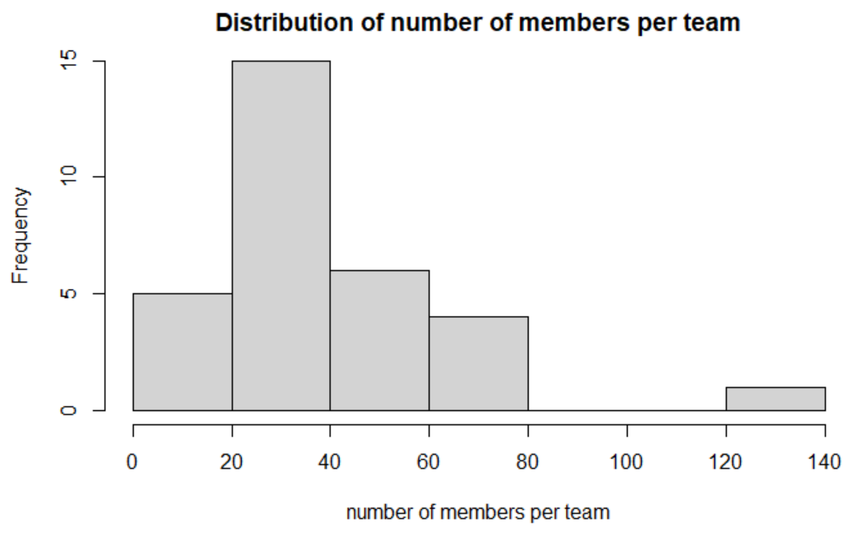


Figure 1: Distribution of number of members per team

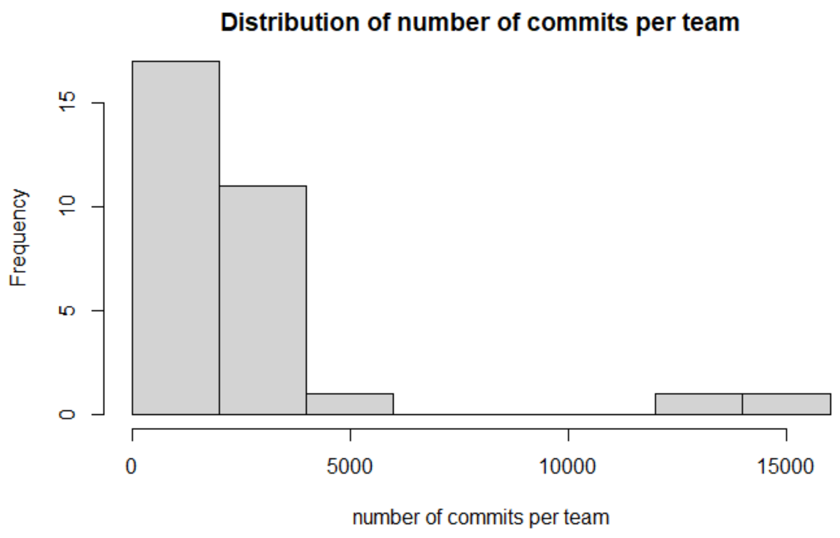


Figure 2: Distribution of number of commits per team

We also checked the statistics for the number of characters for each commit. We saw that the average value of it was 191 characters, but there was a considerable amount of high valued outliers. We can see that the longest commit message in the database has 29011 characters whereas the smallest text has only 5 characters.

The second table that we decided to use was the SONAR_MEASURES, which included 240 columns of metrics of the code quality, complexity, number of issues found in the code, etc. We observed that there was a considerable amount of NA values in this table, so we decided to erase the columns that had more than a 50% of NA, in order to perform the exploration.

This step reduced the number of columns to 61, which is a considerable simplification. Then, we proceed to check each of the variables in the table in order to see interesting aspects to have in mind when operating with the variables. Firstly, we observed that complexity metrics range from 0 to some kind of positive value, which makes complete sense as the first commit will have a code with 0 complexity (no code exists) and the last commit will have a considerably high complexity. We observe the same phenomena with the duplicate metrics. For the coverage metrics, as we're exploring which percentage of the code is run most of the time, we see that there are some NA's (as in the beginning there's no code to run so the program finds itself dividing 0 by 0). There also are some files about the amount of classes, directories, files, functions but we're not really interested in those for the purpose of the project (and there's nothing especially interesting about them). Some of the other relevant columns are the referent to violations, open issues, bugs and development costs which we're going to use when modeling. These lines have no NA's and follow the same pattern as the complexity metrics (increasing during project time). Lastly, the sqale metrics are also mildly interesting, but we're not as interested as in the rest of the metrics.

2.2.4. Data quality

It is important to have good quality data to provide good results and exportable insights after our project. We will analyze the number of errors, missing values and the distribution of them in relation to other values. This analysis will be conducted over the three different tables that we will use.

- **SONAR_ANALYSIS:**

In this table we find 67550 rows. Each one of them should be related uniquely to one of the git commits, so we filter out the duplicates and NAs in the primary key, that is ‘ANALYSIS_KEY’. In total, 0 duplicate rows and 839 rows with NA were found. leaving a total of 66711 rows.

- **SONAR_METRICS:**

In this table we find 66711 rows, which should be related one-to-one with the rows in SONAR_ANALYSIS. No duplicates or NAs were found in the primary key, that is ‘ANALYSIS_KEY’, so the 66711 rows match perfectly one-to-one with the ones in SONAR_ANALYSIS.

- **GIT_COMMITS:**

In this table we find 81072 rows, no duplicates or NA’s were found in the primary key, that is in the commit hash. For each one of these commits, the sonar analysis is performed, so there should be a row in sonar analysis for each commit. The 66711 rows of SONAR_ANALYSIS matched with a row in GIT_COMMITS, so in the end only 14361 commits don’t have their corresponding sonar analysis. These errors are not random, these missing rows are due to code crashes, sonar analysis is only performed when code can be compiled, meaning that these 14361 commits contained compilation errors.

2.3. Data preparation

Once we have studied and thoroughly analyzed and understood the data, we proceed to prepare the data in order to make it suitable for our later models and tests. We will select the data attributes and rows we need, clean the data, construct databases by arregating tables and finally integrate the data.

2.3.1. Data selection

First off, we have selected the tables that contain the relevant data for our task, these are: GIT_COMMITS, SONAR_MEASURES and SONAR_ANALYSIS.
Once we have the tables loaded, we are only interested in those useful variables for our later work, so we just keep the columns that we will work with.
From GIT_COMMITS we will use:

- Commit Hash: primary key
- Project ID: redundant key for context
- Author: for context and target variable in clustering studies
- Commit Text: target variable in NLP studies

From SONAR_ANALYSIS we just keep the ANALYSIS_KEY and REVISION since these variables are the ones that connect the commits with the sonar measures, being REVISION the same as COMMIT_HASH from GIT_COMMITS and analysis_key the primary key of SONAR_MEASURES. Finally, we extract from SONAR_MEASURES:

- Analysis key: primary key
- Complexity: target variable
- Violations: target variable
- Development cost: target variable

2.3.2. Data cleaning

In order to clean our dataset, we removed all the NA values in the aforementioned columns. As mentioned in section 1.2.4, several NA values were found in the SONAR_ANALYSIS table. Then, after computing increments in complexity, violations and development cost (see section 2.3.3 for more details in this derivation), because some commits didn't have a sonar analysis, this made us not able to compute the increments for the following commit. So for every missing SONAR_ANALYSIS row, that row and the following one in date are lost.

2.3.3. Data construction

For the variables related with the sonar measures, we also want to compute their increment over time for each commit. In order to do it, we first merged the sonar analysis table with the sonar measures and then with the git commits so we have each project with their respective commits and measures. After that, we sort the resulting table by project id and committer date, so we will have the table ordered by projects and each commit ordered from old to new.

Finally we create the columns corresponding to the increment of the complexity, violations and development cost called inc_complexity, inc_violations, inc_development_cost by subtracting the values of each entry with the previous entry of the project if available. The values of the columns are filled using the value of the row above and the current value of the commit variable.

Apart from this, we haven't had the need nor ability to generate new records. We keep an open door to derive more attributes once the models are trained if we feel that we need the networks to have more context.

2.3.4. Data integration

Once we have all the relevant data without NA's, we proceed to integrate the data by joining the database tables we have been preparing. Before joining the tables, we have had to change some of the key names in the Sonar Measures tables to match the "Commit Hash" key name of the Git Commit tables. Once this preparatory step has been completed, we have performed the following left joins:

- Sonar = Inner Join between Sonar Analysis and Sonar Metrics using analysis key as primary key
- Final Database = Join between Git Commits and Sonar using Commit hash

2.3.5. Dataset description

After the selecting, cleaning, constructing and integrating the final dataset, we have one unique table to work with that has the following characteristics:

62k rows x 8 columns

- Project ID: string
- COMMIT_HASH: string, primary key
- Author: string
- COMMIT_MESSAGE: string
- COMMITTER DATE: date
- inc_complexity: float
  - Mean: 4.26, Var: 258242.80
  - Max: .15921, Min: -15919
- inc_violations: float
  - Mean: 1.94, Var: 31177.24
  - Max: 11079, Min: -6437
- inc_development_cost: float
  - Mean: 8838.8, Var: 6004734069.0
  - Max: 2978520, Min: -2978520

2.4. Modeling

2.4.1. Modeling assumptions

- **Embeddings**

In order to use the data we have to give meaningful insights, the first step of our modeling was to get a representation of such data that was in a format suitable for our later models. That is why we created sentence embeddings from our main data source, the commit texts, where we get a numerical representation for each sentence. This initial step is a model in itself, that transforms sentences into embeddings with the assumption that the sentences can have variable length and that they have grammatical sense. These assumptions have led us to use a pre-trained Bert sentence embedding model.

- **Complexity increment prediction**

Our first question regarding commit text is whether or not it contains information about the change in software quality metrics of the project. In order to answer this question, we assume the aforementioned embeddings are able to represent all the information contained in the commit text and that they are able to represent the complex vocabulary of a commit text, with paths, links, function names, etc. We also assume that our target is binary: 1 for when complexity increases after that commit and 0 when it does not.

- **Clustering**

One of our purposes is also to try to make clusters of authors based on their commit text. For this we have used unsupervised clustering methods to separate the text commits into groups, and from there to differentiate authors. In order to implement the clusters, we will use the embeddings generated by the previous model, so what we are really doing is a classification of these embeddings. Finally, once we have the embeddings classified, we will assign each author in the cluster where he has the maximum number of commits.

Having a very large number of samples,62917, makes many cluster methods not suitable for our sample, so the option we considered most successful to carry out this classification is **Kmeans**, using **mini-batches**. The KMeans algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the *inertia* or within-cluster sum-of-squares. Inertia can be recognized as a measure of how internally coherent clusters are.

Mini-batches are subsets of the input data, randomly sampled in each training iteration. These mini-batches drastically reduce the amount of computation required to converge to a local solution. In contrast to other algorithms that reduce the convergence time of k-means, mini-batch k-means produces results that are generally only slightly worse than the standard algorithm. The algorithm iterates between two major steps. In the first step, b samples are drawn randomly from the dataset, to form a mini-batch. These are then assigned to the nearest centroid. In the second step, the centroids are updated.

Prior to applying the Mini-batches Kmeans algorithm, with the aim of visualizing our results and improving the algorithm results, we have computed a **Principal Component Analysis**. This analysis allows us to reduce the dimensionality of our embeddings to 3 or 2 dimensions, so we will be able to represent the clusters obtained in 3D and 2D.

2.4.2. Test design

- **Embeddings**

For the embeddings, we decided that the best way to test the pretrained model that generates the sentence embeddings was to check for various examples where we analyzed the cosine similarity between sentence embeddings of similar / dissimilar sentences (commit messages). This way, we get an idea of how well the embeddings capture the difference between sentences. We won't need a testing set for this part of the modeling since we use a pretrained model that has been already tested for its original usage.

- **Complexity increment prediction**

In order to train a model able to predict if there will be an increment in complexity, we will separate our dataset of 62917 samples into two sets: 70% for training and 30% for test. At every epoch of our training, we will use two metrics for evaluating the progress of the training: Loss (more details on section 2.4.3) and Accuracy. These two metrics will be saved alongside with the weights of the model in order to analyze the evolution during and after the training phase.

In order to compare the performance of our model, we take a naive classifier that always assigns observations into the most common class. In our case, the most common class is 0, with 46052 observations, representing 73.2% of the total. Meaning that a classifier of these characteristics would be able to attain 73.2% accuracy. In every accuracy plot, we add an horizontal line at this value so that we can compare it with the performance of our model (see figure X).

- **Clustering**

As the cluster method is an unsupervised method, we do not need to separate our data into test and training sets. We run the algorithm over the entire set of observations.

2.4.3. Model description

- **Embeddings**

The pretrained Bert sentence model (https://arxiv.org/pdf/1908.10084.pdf) uses Siamese BERT-Networks where the main source of learning that the model uses is based on the Bert transformer network, that uses attention to relate words between sentences. The model we used is **all-MiniLM-L6-v2**, the fastest model of the ones available in the library **sentence-transformers.** The model outputs an embedding 1D vector of size 384 for each sentence in reads as input. The schema is as follows:

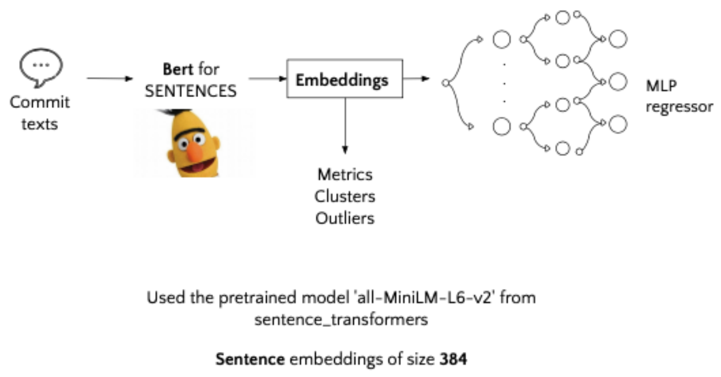

Commit texts
Bert for SENTENCES
Embeddings
Metrics
Clusters
Outliers
MLP regressor
Used the pretrained model 'all-MiniLM-L6-v2' from sentence_transformers
Sentence embeddings of size 384

Figure 3: Schema of our creation and usage of embeddings

- **Complexity increment prediction**

For the complexity increment prediction we use a Multi-Layer Perceptron (MLP) architecture with 384-1024-120-1 neurons per layer, ReLU activations in between and Sigmoid activation after the last layer. As we are modelling the probability of complexity to increase, we use Binary Cross Entropy Loss as our loss function. For optimizing the loss function, we chose ADAM with a learning rate of 0.0001 and a weight decay of 0.0005.

- **Clustering**

The first step to construct our K-Means algorithm was to compute the Principal Component Analysis for the embeddings. The result is a three-dimensional feature for each embedding. Displaying our principal components we obtained:

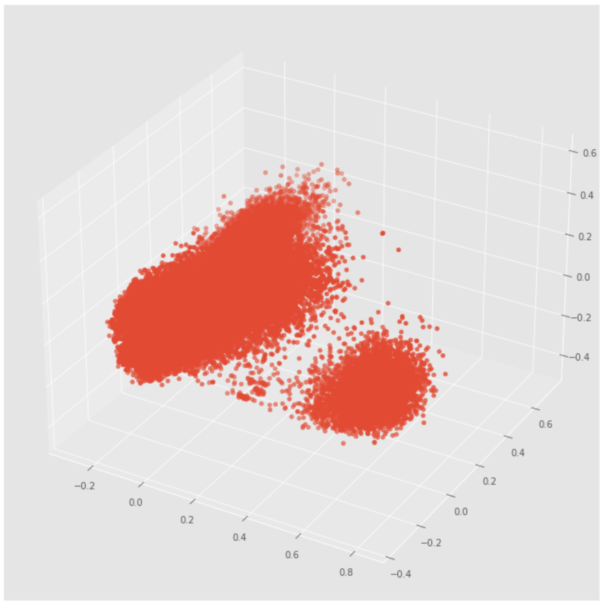


Figure 4: 3D overview of our PCA

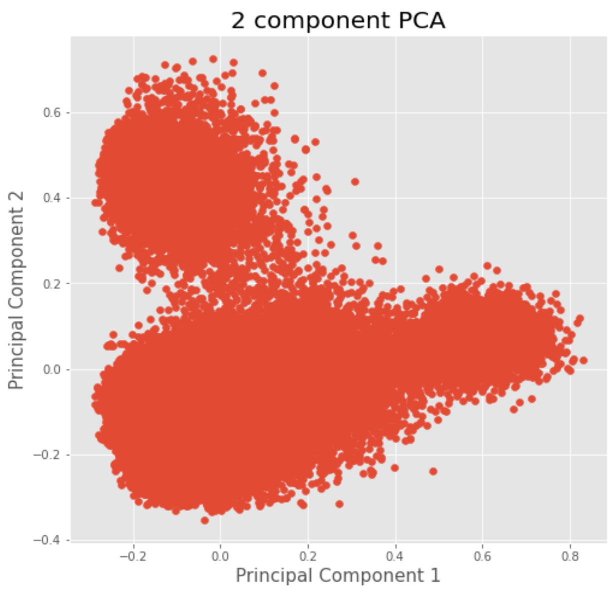


Figure 5: 2D view of our PCA

Once we have our input computed we proceed to design the model. As mentioned above, we use MiniBatchKMeans. This method needs two principal hyperparameters: the number of clusters and the batch size:

- Number of clusters: to find the optimal number of clusters we computed an elbow curve. This plot indicates the inertia for each total number of clusters. We must choose the number of clusters that marks a clear change in the y axis, and therefore draw an elbow. That means that adding another cluster does not give a much better modeling of the data. In our case, we obtained:

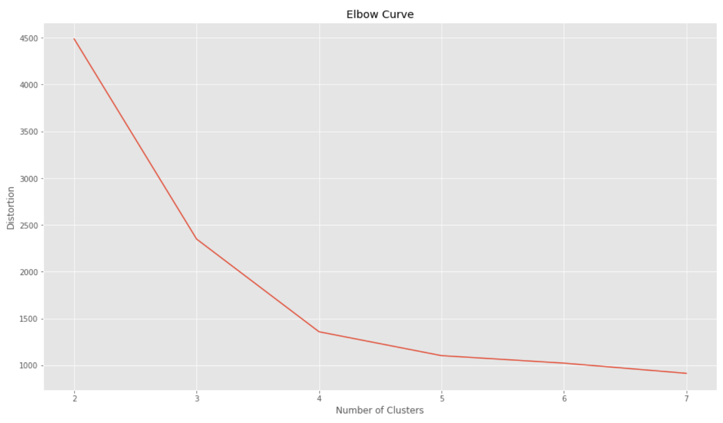


Figure 6: Elbow curve given k (number of clusters)

Looking at the plot (Fig. 6), we decided to compute our k-means algorithm with k = 3 and k = 4, since we see two possible elbows and to ensure taking the best option.

- Batch size: as we have a large number of samples we set the batch size to 1024, which is actually the default parameter in the method that we use.

2.4.4. Model assessment

- **Embeddings**

Following the cosine similarity checking criteria, we went through some of the commit messages and their embedding representations to check if the similarities we observed between sentences made sense. They indeed presented very positive results, which already is a very good step in our project since we can say that we are offering a numerical representation of the clustering messages data that represents such data appropriately and that can be used for any modeling task that uses embeddings. These are a few examples of the tests we took:

```
a = add test PR: MRM-9
b = add some more tests PR: MRM-9
c = ZOOKEEPER-2172: Cluster crashes when reconfig a new node as a participant

Similarity {emb(a),emb(b)} = 0.95
Similarity {emb(a),emb(c)} = 0.09
Similarity {emb(b),emb(c)} = 0.14
________________________________________________________________________________

a = http://issues.apache.org/bugzilla/show_bug.cgi?id=40577
b = http://issues.apache.org/bugzilla/show_bug.cgi?id=39695
c = [MRM-1578] add layout

Similarity {emb(a),emb(b)} = 1.00
Similarity {emb(a),emb(c)} = 0.13
Similarity {emb(b),emb(c)} = 0.13
```

As you can see, in both cases sentences a and b are very similar and their embeddings' cosine similarity reflects so.

- **Complexity increment prediction**

Following the quality metrics mentioned in section 2.4.2., we analyze our model in terms of loss and accuracy. In figure X, one can observe that in the first epochs, loss rapidly decreases for both validation and training sets, but after epoch 5, only training loss decreases and validation loss fluctuates around 0.51. In the accuracy plot, we observe a similar fenomena, after the first 10 epochs, validation accuracy stops increasing and fluctuates around 0.74, while training accuracy never stops increasing.

Both these results lead to assessments that the model finds no correlation between our embeddings and it only remembers training observations. This is not an overfitting problem as weight decay is being used and validation loss doesn't increase.

Furthermore, we observe that accuracy only surpasses our baseline by a very small margin, which leads us to think that the model is not significantly better than our baseline of assigning every observation to zero.

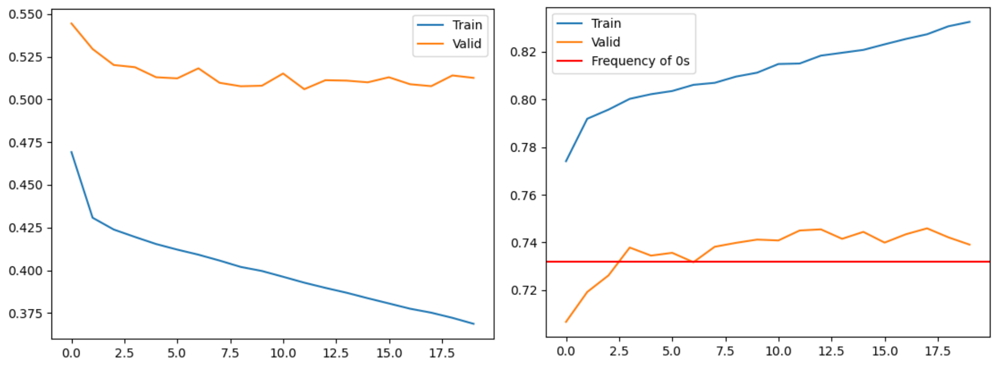


Figure 7: Loss and Accuracy plots (Left and Right respectively)

- **Clustering**

Following the criteria mentioned above, we decided to compute the clusters with k equal to three and four, the principal components of the embeddings as inputs and batches of 1024 samples. With these parameters we proceed to create our clusters, and the results were the follows

→ K = 3

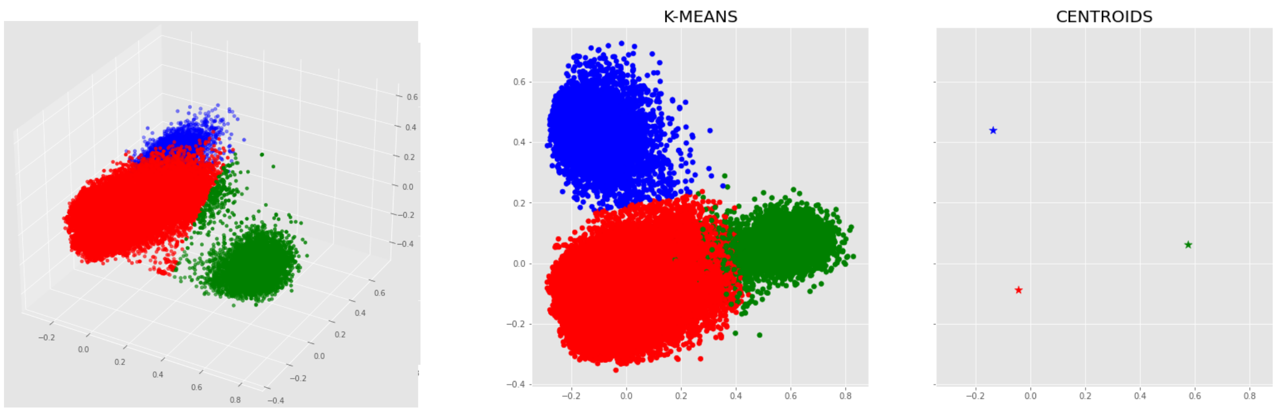


Figure 8: 3D plot of PCA components with 3 clusters

→ K = 4

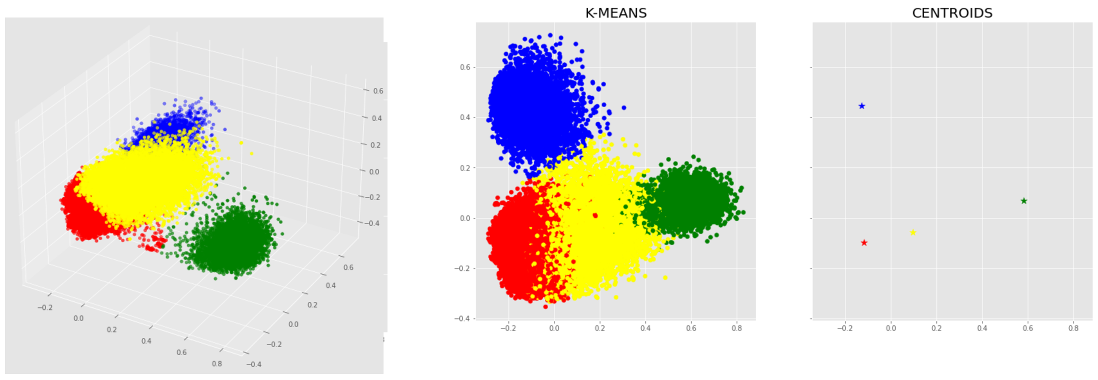


Figure 9: 3D plot of PCA components with 4 clusters

We see that when plotting the results we have two well-defined clusters, while the more centrally located samples seem not to be so differentiated. In the case of 4 clusters, it is clear how these samples are divided into two clusters without a very obvious differentiation.

To see objectively which of the two methods gives us the best differentiation between clusters, we are going to use two evaluation metrics: Calinski-Harabasz Index and Davies-Bouldin Index. Both are used taking into account that we do not have ground truth information, which is our case, and they analyze internal characteristics of the clusters obtained. Calinski-Harabasz Index, as known as variation ratio criterion, computes the ratio of the sum of between-clusters dispersion and of within-cluster dispersion for all clusters, where dispersion is defined as the sum of distances squared. That means that higher index relates to a model with better defined, dense and well-separated clusters. Davies-Bouldin Index represents the average similarity between clusters, where the similarity is a measure that compares the distance between clusters with the size of the clusters themselves. Lower index relates to a model with better separated clusters.

The metrics we obtained were:

	K = 3	K = 4
Calinski-Harabasz Index	**4879.38**	4121.24
Davies-Bouldin Index	**2.2**	3.48

Figure 10: Cluster goodness of fit metrics

We can conclude, that the model which fits better our data, and therefore, the one that we will use to perform deeper analysis is Kmeans with three clusters, since it has a higher Calinski-Harabasz Index, so better defined clusters, and lower Davies-Bouldin Index, which means better separated clusters.

2.5. Evaluation

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

- **Embeddings**

After having created the sentence embedding representations, we evaluated the results with the use of cosine similarities between them and concluded that they indeed fulfill the function that we were looking for in encoding accurately the meaning and content of the commit messages. Having said that, this already is a satisfactory step with regards to the business success criteria, where we had a very clear main objective of providing insights given commit messages. With these embeddings we open the door to being able to do so in a very wide range of possibilities.

- **Complexity increment prediction**

For this section, our goal was to find if there existed a relationship between commit text and the increment in complexity after a commit. After our analysis we can conclude that for our commit text representation and model architecture, there is no relationship between commit text and increment in complexity. Even though our success criteria of better than 70% accuracy is met, this is not meaningful as the baseline classifier that we benchmark with also attains this quality metric.

- **Clustering:**

In order to evaluate the results of the clustering section, we're first going to review if final results achieve the success criteria, and after that, we will take a look at some of the extra insights obtained from the analysis of the clusters.

The first main objective of the clustering section of the project was to obtain a clear separation between 2 well balanced clusters. We slightly changed the objective, as we saw in the elbow curve that with 3 or 4 clusters we would obtain far better results and we decided to go for three clusters.

For the second parts of this objective (to obtain balanced clusters) the number of entries per cluster, although is not a balanced (⅓, ⅓, ⅓) proportion it is not disjointed to the point it neglects further analysis with: entries in the cluster "0", 6067 in the cluster "1", 8815 entries in the cluster "2" with proportions 0.76, 0.10,0.14 respectively. Although we have a clear majority of entries in cluster "0" we considered this analysis as a success in terms of balance because of three reasons: The first reason was that we clearly observed a separation graphically when we plotted the results of the k-means algorithm. The second reason was the fact that the number of entries was big enough per cluster to obtain representative insights of each cluster. The last reason was the fact that even if the proportions were not as balanced as expected, they did not show an imbalance large enough to preclude an analysis.

For the second cluster related objective, which was to being able to identify some developers that write commits in a very distinguished manner or that tend to write misleading commit texts, we found two sources of outlier (or distinguished commit texts) which were the commit texts written by two authors called “Sebb” and “markt ”or more generally some of the commits found in the smaller cluster, where we could find considerably long messages or texts with a lot of specific field notation. Further explanation of this phenomenon will be explained later in this section.

After reviewing the success criteria, we will explain some of the extra insights obtained by performing clustering that may be considerably useful:

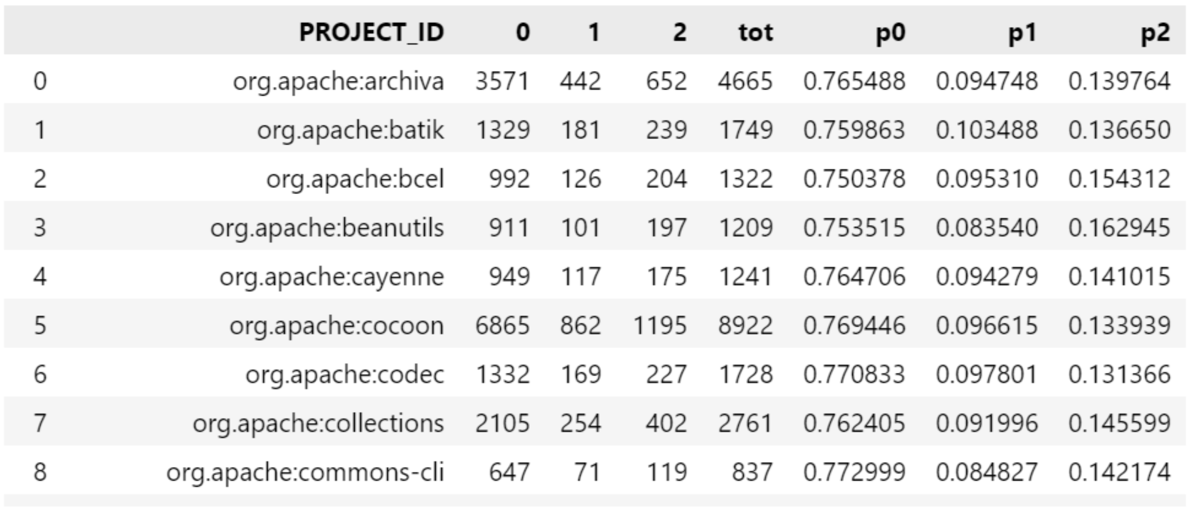
- Author-cluster relation

In this subsection we will explain how we tried to find some relation between authors and the clusters. In order to do that, we assigned each author to the cluster to which belonged the majority of its commits. By doing this, we obtained the following results: From the 358 authors, we found that 341 belonged to the cluster “0”, 11 belonged to the cluster “1” and 6 belonged to the cluster “2”. With these new assignations, we decided to analyze entries of each cluster in order to identify some characteristics. In the commit texts of the authors assigned to the cluster “0” we didn’t find any patterns nor keywords, which makes sense as this cluster contained most of the entries in the database. The cluster “1” cluster contained 6 authors, being the aforementioned authors Sebb and markt part of this group; there only was a total of 10 commit messages, three of which had been written by Sebb and only one by markt. The commits by these two authors didn't contain the usual link that precedes every message and one of these wasn’t even a sentence as*“+= isLegalFile(CharSequence)”* was the full commit message. We can also observe that boths authors' commits mostly caused the complexity and the development time of the project to increase. The other four authors’ commits of this cluster didn’t follow specific patterns. Lastly, we observed that the commit texts of the authors assigned to the cluster “2” were kind of heterogeneous, as we only found some keywords that appeared in some commit texts of the group. The keyword that appeared the most was “fix” or “fixed” but some other words like “update” had some appearances. Despite that, most of the commits of this cluster didn’t follow a clear grammatical or syntactical pattern.

- Project-cluster relation

In this section we decided to explore the distribution of the cluster of each commit in all projects. We computed how many commits of each cluster there were in the projects and then computed the percentage of each cluster commit, in order to see if these percentages were significantly different to the proportions (0.76, 0.10, 0.14) which are the percentage of entries of each cluster in the set all of commits.

We observed that for all the projects, the proportions were followed consistently which meant that the embeddings didn’t differentiate between the projects, which is considerably positive as we want to avoid biases when predicting the increase of the metrics.



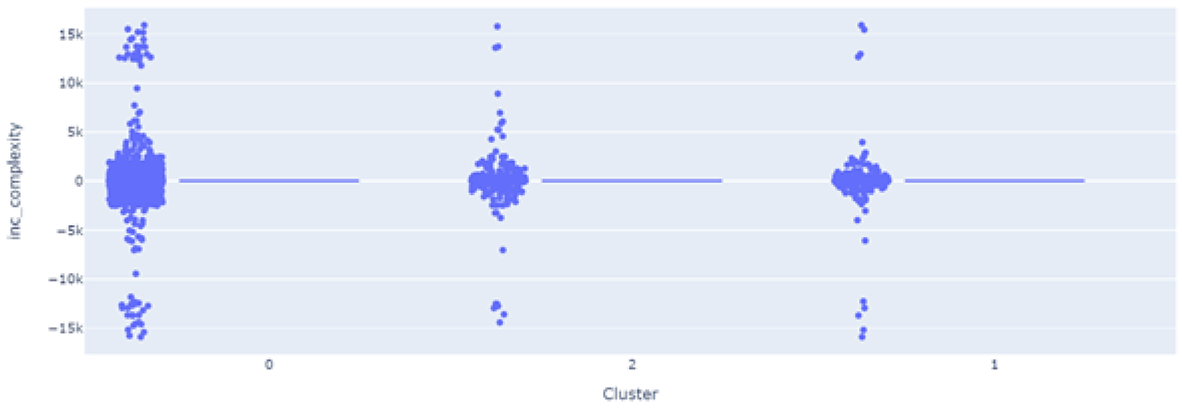
	PROJECT_ID	0	1	2	tot	p0	p1	p2
0	org.apache:archiva	3571	442	652	4665	0.765488	0.094748	0.139764
1	org.apache:batik	1329	181	239	1749	0.759863	0.103488	0.136650
2	org.apache:bcel	992	126	204	1322	0.750378	0.095310	0.154312
3	org.apache:beanutils	911	101	197	1209	0.753515	0.083540	0.162945
4	org.apache:cayenne	949	117	175	1241	0.764706	0.094279	0.141015
5	org.apache:cocoon	6865	862	1195	8922	0.769446	0.096615	0.133939
6	org.apache:codec	1332	169	227	1728	0.770833	0.097801	0.131366
7	org.apache:collections	2105	254	402	2761	0.762405	0.091996	0.145599
8	org.apache:commons-cli	647	71	119	837	0.772999	0.084827	0.142174

Figure 11: Percentage of authors in a project belonging to each one of the three clusters

- Metrics-cluster relation

In this section we decided to analyze if there were differences between the metrics (*increase of complexity, increase of violations and increase of development costs*) varied based on the cluster where they belonged. In order to perform this analysis we first looked at the mean and variance of each of the metrics mentioned above for the three clusters, and we obtained that, even though means appeared to be different, the huge variance in each metric meant that it was impossible to prove significant differences between the metrics. In order to solve that, we used boxplots to find differences between the distributions of the metrics. The results were the following. Note that although we have very big variance, most of the entries are found in a very small interval (which makes that the box transforms into a line due to the closeness of the first and third quartiles).

We observe that although the distributions seem similar, the values out of the box for cluster "0" seem to go a little further than for the other clusters. Despite that, the distributions are considerably similar, as we have most of the values in a pretty small range, really high outliers and some values out of the box.



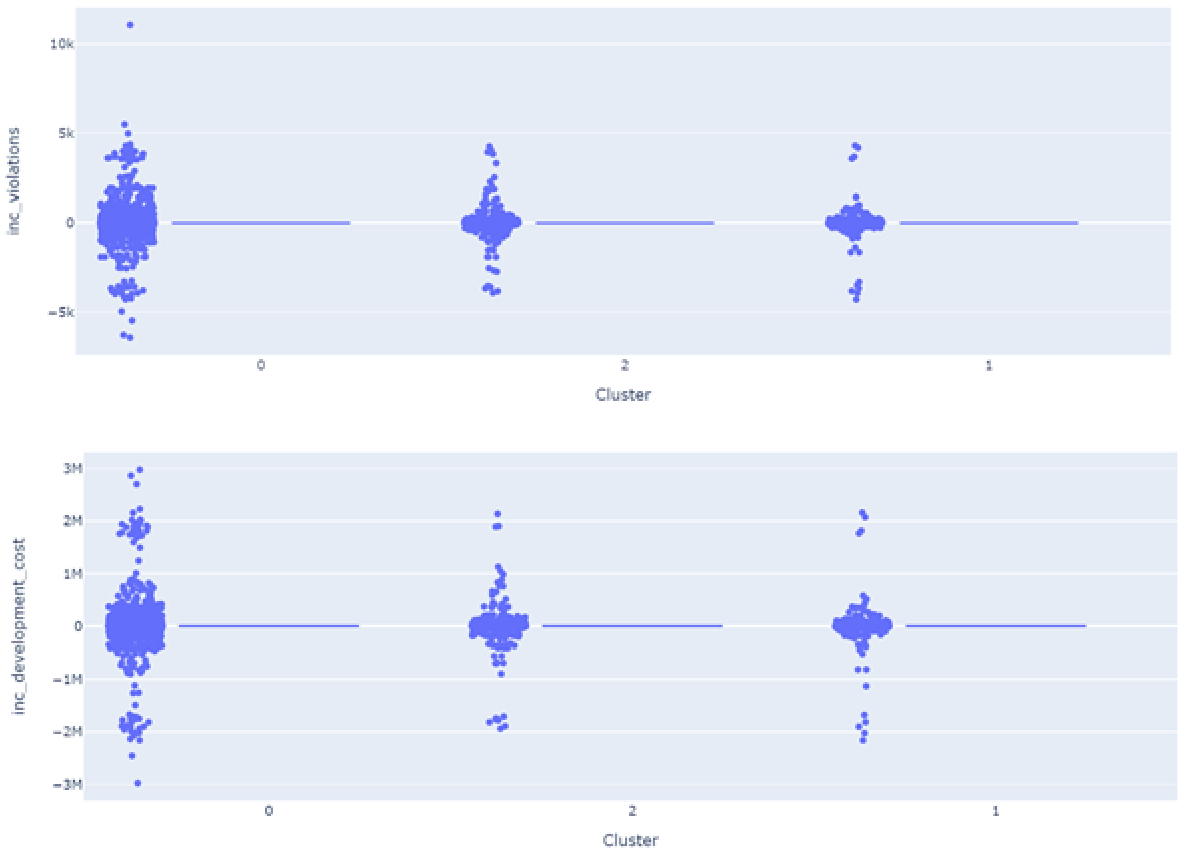


Figure 12: Boxplots of each of the three variables inside each of the three clusters

- Cluster characteristics:

Lastly we decided to search for some of the characteristics of the commit texts in each cluster. Firstly, we found that the entries in cluster “0” don’t follow any specific pattern in terms of the words used or the structure. In the cluster “1” we found that, although there was some variety, some outlier values like really long commits, commits with strange words or even the commits by authors like “Sebb” or markt that didn't have the github link after the text. Lastly,in the cluster “2” we found that there were a lot of commits with the keywords “fix” and “add” and derivatives of these words, although there was a lot of variety too. The main difference between these results and the results found with the clusters by authors are that in the cluster “1” now we can see different kinds of outliers and that now the clusters “1” and “2” have more entries than before (as expected).

2.5.2. Review of process

Having modeled and evaluated the models, following good software developing practices we performed a revision of the different models and processes, to see what went well and what could be improved or should be revisited

- **Embeddings:**

After a first careful review of the process we followed to create the embeddings, we saw that we had missed a very key aspect of the data that we were working with. We realized that all commit messages in our curated dataset had a link at the end of the commit in the github project. After analyzing the effects of all messages having a very large link in the end we saw that the latter was too distracting for the embedding model and that we should not pay attention to the link since it gave us non interesting information from which to learn. Therefore, we decided to remove the links of all messages and re-run the pretrained Bert sentence model to create new embeddings. These corrected embeddings made much more sense after evaluating their correctness analyzing similarities between them.

- **Complexity increment prediction**

In general, the modelling of the increment of complexity was a very straightforward task and no steps were performed wrong. All the training and evaluation is easily repeatable and the code was built so that it could be easily modified and re-run. The only aspect that could have been improved is the testing of newer architectures or other classical machine learning methods. Also, other variables like increment in violations or in development cost were not analyzed because of the large time needed to train the models, with more time or a bigger team this could have been done. Even though these variables weren't analyzed, they are present in the dataset and can be analyzed just by changing 1 line in our code.

- **Clustering:**

In terms of the clustering section, we obtained satisfactory results even though we had to relax some of the success criteria conditions. We performed clustering with KmeansMiniBatch because the other methods were considerably slow and didn't reach convervenge and tried 3 and 4 clusters, to which we decided to pick 3 as the best option after comparing some cluster quality metrics. We considered also using PCA on the embeddings instead of using the raw embeddings, which we decided to use as the results were considerably better.

After that, we decided to deeply analyze the results of the embeddings and to relate these embeddings with some of the other variables of interest in the database like the authors, metrics like the increase of complexity or even the project of the commit in order to obtain valuable insights that weren't bound on the success criteria but were considerably useful. The results of this deeper analysis provided a satisfactory view on the utility and further usage of the embeddings. Having said that, using a trained network to create our embeddings, even though avoiding overfitting problems, may be the cause of not having more balanced clusters. The possible solutions to this problem will be provided in the following section "list of possible actions".

Another factor to consider may be to consider the problems that cause working with links, and the problems that may have caused not working with some kind of tokenization in the sentences. Although we consider that this shouldn't have a big effect in our specifica case, in further analysis it would be interesting to use some kind of tokenization of working with word or subword based networks.

2.5.3. List of possible action

List possible further actions along with the reasons for and against each option.
possibles millore

- **Embeddings:**

The embeddings are the key for all our posterior modeling to finally provide insights. Therefore, it is crucial that they are the best possible. We thought about the fact that the embeddings we are using have been pre trained for a different task than ours, and that a good way to make them even better with better capturing of the information in the commit messages would be to train a specific embedding model using our data. (Keep in mind we only used the model with our data, we didn't train it with it). This way, we would have an end-to-end representation of our data that could yield better results for complexity prediction and clustering. We think that this would be a very good starting point for further improvements in this project.

- **Complexity increment prediction**

This project has allowed us to establish a first stone for further work in commit text analysis. Future analysis would be:

  - Training end-to-end complexity increment prediction:

As mentioned above, this would allow us to both obtain better representation (embeddings) of the commit texts and obtain more adequate features for predicting increment in complexity. An approach to this could be to train an LSTM or Attention based encoder that captures the relevant information from the commit and then a MLP that classifies the commit based on these embeddings.

  - Exploring performance of current approach with different variables

As stated in section 2.5.2., the analysis of the predictive power of commit text could be extended to other variables like increment in violations, increment in development cost or other simpler values like the increment in number of lines.

- **Clustering:**

We considered two main sources of improvement in the clustering section. Firstly, we thought that that the method of creation of the embeddings, could be changed in order to provide different and interesting results to the problem:

- Finetune a language modelling network to create the embeddings: we consider that this is the best alternative to our approach, as we would have some previous language knowledge provided by the network and by fine tuning in a task like predicting metrics (freezing the bottom layers) we would add some specific field knowledge. The problem with this approach is that even though it would still need a considerable amount of data to train well and not overfit and we considered that we didn't have enough rows to do so.

- Training a recurrent neural network or a transformer-like model: this alternative may provide good results as we think that it could work better because of the considerable big amounts of specific language that the commits have. The counterpart of this is that there may not be enough entries yet in order to not overfit while training, which makes us consider this option as the best option when the database grows bigger.

Another problem mentioned above is the fact that no tokenizing is performed before creating the embeddings in the network. We don't think that providing not tokenized sentences to the network was a problem as we consider that it was positive to add the maximum amount of information available to the network. Despite that, if we worked with word or subword based language modelling networks, tokenizing may be considerably useful and even in our case it may work fine as we could delete some information that we considered not useful for the clustering or prediction of the metrics.

Lastly, the fact that we used PCA in order to perform clustering caused a big improvement in the quality of the results which makes us think that other methods of dimensional reduction like TSNE or UMAP may provide interesting results.

2.6. Deployment

2.6.1. Deployment plan

Our deployment plan consists of a detailed reporting of our results, which we are doing in this document. Along with this report we add our Readme file that is in our git repository that includes a detailed explanation of how to repeat our code and interpret our files. This in itself is our deployment to the public, any user that wants to use our embeddings and our studies to further research on how to provide insights given commit messages from software projects.

2.6.2. Monitoring and maintenance plan

Since we have provided our github repository to the public, our maintenance plan consists of simply keeping the git repository up to date and allowing the users to post issues there. If any user posts an issue they will get an early reply by any of our team's developers. We will also monitor the popularity of our repository by analyzing the number of forks and stars it gets once we publish this project.

2.7. Final executive summary

- Summary of business understanding: background, objectives, and success criteria

Nowadays there is a lot of information, provided from different platforms, about the quality of the code and in software development projects. This information is useful to track the evolution of the projects and find insights for projects developers and companies. The extracted metrics are used to be very analytical, such as complexity, development cost or violations. Github is a source code repository that provides access control and several collaborative features. Our motivation to carry out this project is to extract useful data from the commit text of the commits in Github. This is something new since until now, text has not been used as a data source to analyse the quality of the commit.

We want to be able to offer new meaningful insights from commits messages. For example we want to know if there is any relation between the commit message and their quality or if there exists some segmentation of authors in terms of how they write the commits and if this segmentation has any relation with the code quality.

Translated into data mining terms, our objective is to predict the error severity of one commit based on the commit message, find relationships between developers also based on the commit texts and determine if the code quality has some influence in this relation and finally, detect some outlying authors.

Our success criteria, in business terms, is to be able to determine if meaningful data can be extracted from commit texts or not. And for the data mining, we want at most a 30% error when predicting the metrics, be able to segment developers into, at least, two distinguishable groups, and identify some outlying authors.

- Summary of data mining process

In order to develop this approach, we use the Technical Debt Dataset that provides us data from different software development sources and tools, one of them the GitHub commits. This data set contains several tables with different kinds of metrics and sources. For our task we only are interested in SONAR_MEASURES, SONAR_ANALYSIS and GIT_COMMITS tables. In fact, we derived a personalized dataset that contains: project_id, commit_hash, commit_message, author, committer_date, incremental complexity, incremental violations and incremental development cost. The last three variables represent the difference between the actual commit and the previous one from the same project of each metric. The result is a dataset with 62917 rows.

To be able to extract data from text the first thing we have to do is to transform our raw text into embeddings to get a suitable representation of the text and later apply predicting and clustering methods. To create these embeddings we use a pretrained Bert sentence model based on the Bert transformer network that gives us a vector of size 384 for each commit text. Once we have our embeddings computed, we can use them as input for our next models.

For the complexity prediction we used a MLP architecture, where the input was the embeddings of the commits texts, and the output is a binary output, where one indicates that the complexity increases, and zero that it does not.

For the segmentation, we made a clustering analysis where we used the K Means algorithm to get our classification. The input of the method was the Principal Components of the embeddings obtained from Bert. At first, we computed the algorithm for three and four clustering, following the elbow curve criteria and we concluded to work with just three clusters since the clusters obtained were better.

- Summary of data mining results

Once the models are implemented, we are going to observe what results we obtain.

In the prediction, to evaluate the model, we use the loss and accuracy metrics. In terms of loss, we saw that in the validation set the loss remained around 0.51, while in the training set it decreased throughout the epochs. With accuracy it was somewhat similar. In the validation set it remained around 0.74 while in the test it did not stop improving.

In the segmentation tasks, we obtained the following clusters: 48347 entries in the cluster “0”, 5746 in the cluster “1”, 8824 entries in the cluster “2” with proportions 0.76, 0.10,0.14 respectively. When associating the authors to clusters, from the 358 authors, we found that

341 belonged to the cluster “0”, 11 belonged to the cluster “0” and 6 belonged to the cluster “2”.

- Summary of results evaluation

Finally, we are going to analyze the results to determine how good they are.

First of all we want to know if the embeddings are really representative or not. For this we have used the cosine similarity so that, for similar commits texts we have similarities close to one, and for very disparate commit texts we have similarities close to zero. Testing some examples we have seen that this is indeed true.

For the prediction, seeing the results obtained by the model, we determined that we have not been able to capture any relationship between the commit texts and the increase of complexity.

For clustering, we have done a deeper analysis to find some characterization of the clusters.The most remarkable insight is that cluster 1, where we have 6 authors, we have been able to detect two authors outliers that are "SEBB" and "markt".

When we analyzed the cluster of commits we saw that the entries in cluster “0” don’t follow any specific pattern, in cluster “1” there was some variety, some outlier values like really long commits, commits with strange words or even the commits by authors like “Sebb” or markt and in cluster “2” we found that there were a lot of commits with the keywords “fix” and “add” and derivatives of these words.

- Conclusions for the business

Despite the fact that we were able to create meaningful embeddings for our text, our models do not seem to provide the expected results. Our initial idea was to be able to obtain information from these messages, and to characterize the committee based on this.

Nevertheless, when working with the different models we have realized that the relationship between the text and the quality metrics is not so trivial. In any case, we believe that embeddings computed and the segmentation can be of great help for later projects. Especially, the embeddings can serve as a first step for possible later work in which one wants to work also with the text of the committees. In the case of segmentation, the fact of being able to identify authors with differentiable commit text can also become an advantage for subsequent analyzes, even from the company itself.

- Conclusions for future data mining

After performing the data mining process, we believe that, still not obtaining models capable of capturing the expected relationships, we offer a good baseline to be able to do later analysis for commit messages. In addition, we have created an efficient and repeatable process to merge and clean the tables to create the final curated database and reproduce results.

Regarding the fulfillment of the success criteria, in the prediction of complexity we have not achieved a model adjusted to our task, and therefore we have not found a relationship between text and complexity.

For the segmentation, on the other hand, we have been able to segment the commits into three separable and distinguible groups. Associating these clusters to the developers we can also segment the authors. This same analysis has allowed us to find outliers authors.

Finally, we believe that future improvements could have been made in later work, such as standardizing data, creating end-to-end embeddings, and exploring LSTM models with word embeddings instead of sentence embeddings.

Annexes. Good engineering practices

Annex A. File structure and replication package

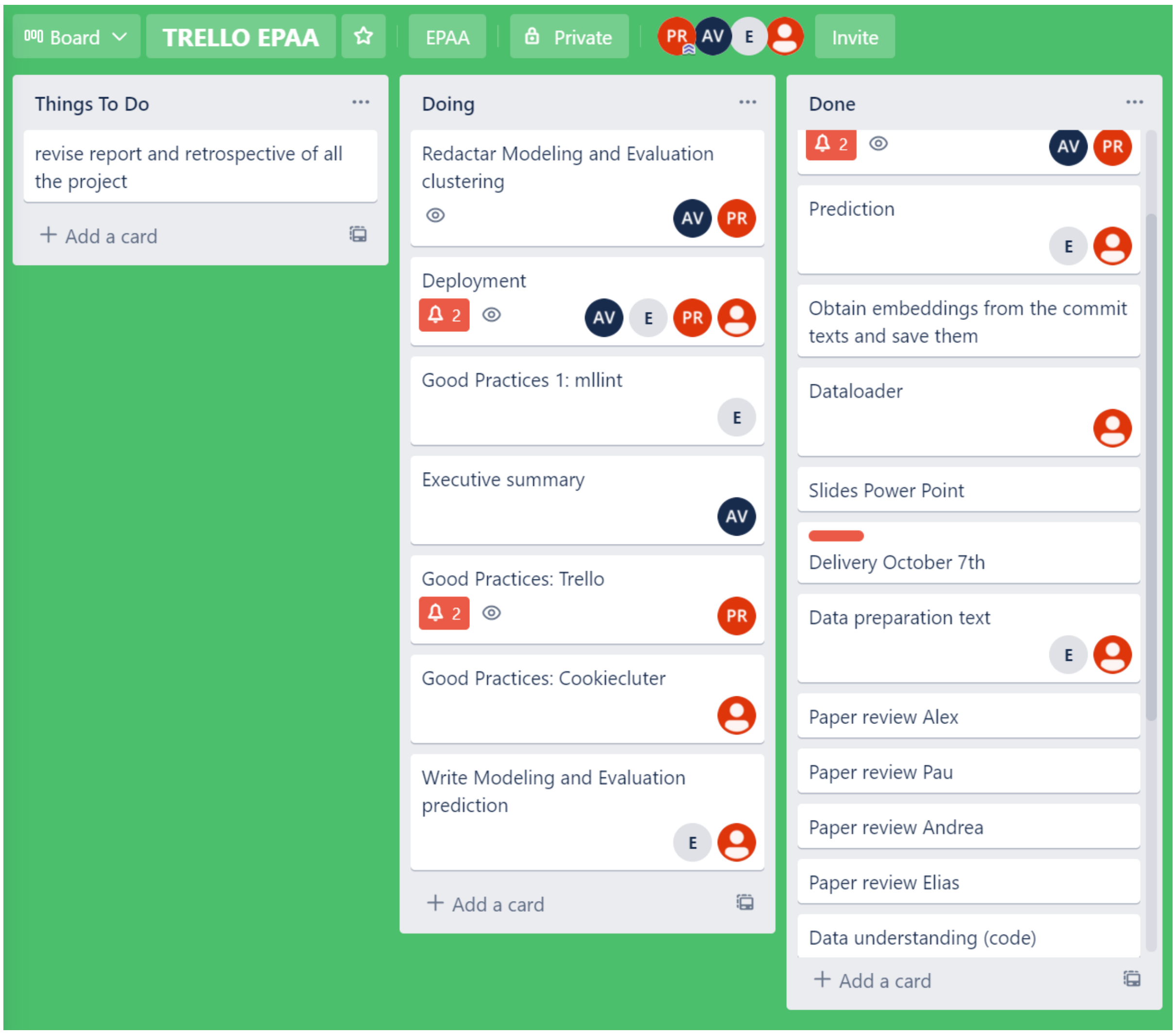
In order to keep track of the versions and changes we want to make during the software development process, we need an organized github repository that can then be used to allow future users to replicate and find accessible the code they need. Therefore, we have used a file structure using a replication package from CookieCutter, which offers *"A logical, reasonably standardized, but flexible project structure for doing and sharing data science work".* The structure is as follows:

- Crucial interpretability and repeatability files such as **Readme** and **requirements.txt**. Our readme explains the steps and which files to use in order to repeat our steps and get our results.
- A **data** folder that we subdivided into the raw data (where we have the raw TDD csv files) and the processed data (where we have saved the processedDB and the embeddings extracted from such database).
- A **docs** folder with configuration files such as the Makefile and conf.py
- A **models** folder where we will store the big models that need to be saved outside a python file. In our case we haven't had to save any model since both the embeddings and complexity prediction models were small and could be loaded and created inside the python scripts.
- A **notebooks** folder where we stored all the python notebooks (.ipynb) that we used for testing, development and pipelining. In the folder we have the clustering studies, as well as the preprocess and data understanding processes. Finally we also have Bert and MLP tests.
- A **references** folder to save any useful information of our sources. In our case it is empty since our sources are already explained with detail in this report.
- A **src** folder to save the python scripts that we used to create embeddings, predict complexity and cluster authors.

Annex B. Trello

In order to keep track of the workload of each of the project members and to distribute this workload in a structured and balanced way, we used Trello. Trello is a web-based, Kanban-style, list-making application self described as *"the ultimate project management tool"*.

In order to keep things simple we created a single board where each of the tasks would be added and assigned to a member or members of the group. The board followed the classic structure of having three sections with “TO DO”, “DOING”, “DONE”. The majority of the cards were subsections of the that only had one person working on it, even though the sections that required collaboration between two members were assigned to both members. The usage of this tool has been considerably useful in order to help to keep an order in the workload, but in order to have an easy pathway of communication between the members of the team we also used Discord. The usage of this tool was considerably useful when we wanted to decide the assignment of the cards, to well define the limits of each of the members of the project and to provide us help with the incremental code development.



Things To Do
- revise report and retrospective of all the project
- Add a card

Doing
- Redactar Modeling and Evaluation clustering — AV, PR
- Deployment — 2 — AV, E, PR
- Good Practices 1: mllint — E
- Executive summary — AV
- Good Practices: Trello — 2 — PR
- Good Practices: Cookiecluter
- Write Modeling and Evaluation prediction — E
- Add a card

Done
- 2 — AV, PR
- Prediction — E
- Obtain embeddings from the commit texts and save them
- Dataloader
- Slides Power Point
- Delivery October 7th
- Data preparation text — E
- Paper review Alex
- Paper review Pau
- Paper review Andrea
- Paper review Elias
- Data understanding (code)
- Add a card

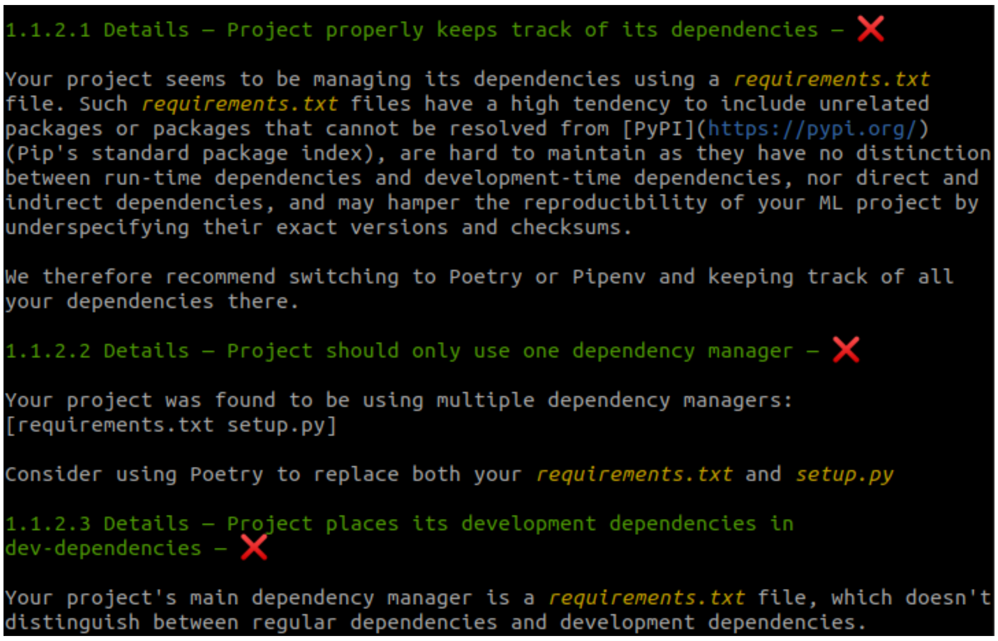
Figure 13: Screenshot of our Trello board

Annex C. mllint

This project was required to include a good software engineering practice related to code development, so we decided to use mllint. Mllint is a static code analysis tool designed for Machine Learning projects. This tool allows creating and maintaining ML projects and helps assessing the quality of the projects.

In this project, we decided to use mllint to check the quality of our repository and improve some aspects so that it is easier to reproduce our results and use our work as a base for future projects.

After running mllint on our project we got some interesting insights:

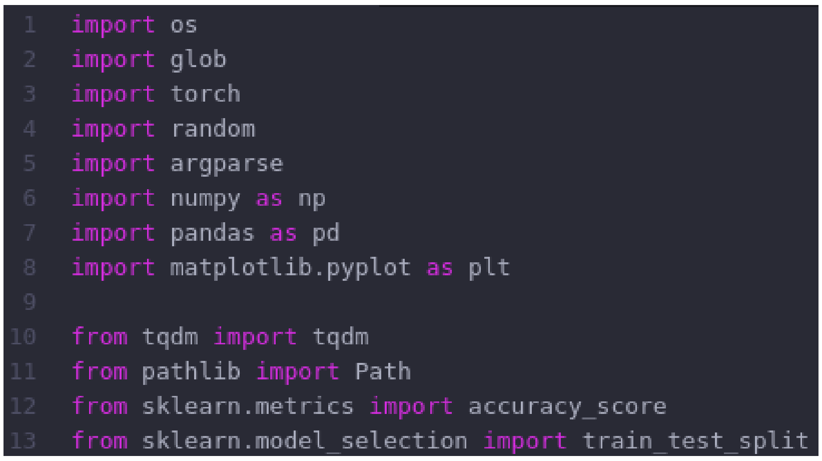


```
1.1.2.1 Details — Project properly keeps track of its dependencies — ❌

Your project seems to be managing its dependencies using a requirements.txt
file. Such requirements.txt files have a high tendency to include unrelated
packages or packages that cannot be resolved from [PyPI](https://pypi.org/)
(Pip's standard package index), are hard to maintain as they have no distinction
between run-time dependencies and development-time dependencies, nor direct and
indirect dependencies, and may hamper the reproducibility of your ML project by
underspecifying their exact versions and checksums.

We therefore recommend switching to Poetry or Pipenv and keeping track of all
your dependencies there.

1.1.2.2 Details — Project should only use one dependency manager — ❌

Your project was found to be using multiple dependency managers:
[requirements.txt setup.py]

Consider using Poetry to replace both your requirements.txt and setup.py

1.1.2.3 Details — Project places its development dependencies in
dev-dependencies — ❌

Your project's main dependency manager is a requirements.txt file, which doesn't
distinguish between regular dependencies and development dependencies.
```

Figure 14: mllint results related to dependency management.

In figure X, we can observe that the use of requirements.txt seems to not be the best tool for dependency management. But as our project has very few dependencies and they are very popular ones like numpy, pandas or pytorch, we will stick to this simpler dependency manager. Also for section 1.1.2.2, we realized that we had requirements.txt and setup.py at the same time, which could confuse future developers, so we decided to remove the latter.

Mllint also reported that no linter was installed, so we installed pylint and re-run mllint to see which were the code quality warnings from this linter. Most of the warnings were related to the naming convention of the variables or the order of the imports. For the naming of variables and functions we used lowercase and "_" for separating words, while pylint suggested using different conventions for different types of variables. For import order, we imported first when import starts with "import" and then by increasing the length of the import line (See figure X), while pylint suggested importing first standard libraries. In conclusion, we believe this warnings are not relevant as we follow some coding conventions and pylint warnings would be avoided by configuring our conventions before the analysis.



```python
import os
import glob
import torch
import random
import argparse
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from tqdm import tqdm
from pathlib import Path
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
```

Figure 15: Example of importing convention used in our project