Speeding Up the Data Extraction of Machine Learning Approaches: A Distributed Framework

Martin Steinhauer
SeSa Lab - University of Salerno
Fisciano (Salerno), Italy
m.steinhauer@studenti.unisa.it

Fabio Palomba
SeSa Lab - University of Salerno
Fisciano (Salerno), Italy
fpalomba@unisa.it

ABSTRACT

In the last decade, mining software repositories (MSR) has become one of the most important sources to feed machine learning models. Especially open-source projects on platforms like GitHub are providing a tremendous amount of data and make them easily accessible. Nevertheless, there is still is a lack of standardized pipelines to extract data in an automated and fast way. Even though several frameworks and tools exist which can fulfill specific tasks or parts of the data extraction process, none of them allow neither building an automated mining pipeline nor the possibility for full parallelization. As a consequence, researchers interested in using mining software repositories to feed machine learning models are often forced to re-implement commonly used tasks leading to additional development time and libraries may not be integrated optimally.

This preliminary study aims to demonstrate current limitations of existing tools and Git itself which are threatening the prospects of standardization and parallelization. We also introduce the multi-dimensionality aspects of a Git repository and how they affect the computation time. Finally, as a proof of concept, we define an exemplary pipeline for predicting refactoring operations, assessing its performance. Finally, we discuss the limitations of the pipeline and further optimizations to be done.

KEYWORDS

Machine Learning Pipelines; Distributed Mining; Mining Software Repositories.

ACM Reference Format:
Martin Steinhauer and Fabio Palomba. 2020. Speeding Up the Data Extraction of Machine Learning Approaches: A Distributed Framework. In *MaLTeSQuE 2020: International Workshop on Machine Learning Techniques for Software Quality Evaluation, November 13, 2020 - Sacramento, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The success of machine learning heavily relies on the amount and quality of data the algorithm is trained on. This is true not only for analyzing source code repositories but all learning strategies in general. At the the same time, the generation and extraction of suitable data is often based on a manual process which is costly and time-consuming. While getting the data for other machine learning settings is often difficult, the process of mining software repositories benefits from GitHub as one of the most important data sources and therefore has a high ability to be automated. The availability of GitHub data through big and public available open-source projects is one of the key aspects in MSR. Automation not only decreases the time spent on building datasets, it also improves the reproducibility of research projects and reusability of mining fragments throughout different projects when designed with generalization in mind.

However, in this preliminary work we show that the level of automation can be extended and improved by building a pipeline for automated repository mining and also improve the computation performance by applying a distributed parallelization approach already renowned in the area of big data processing. An exemplary pipeline is built that is able to detect and extract refactoring operations directly from GitHub repositories and calculates software quality metrics before and after the refactoring operation has been conducted. Eventually, granular performance data is collected and evaluated to affirm the potential of further improvement and work.

2 BACKGROUND AND LIMITATIONS OF THE STATE OF THE ART

In this section, we provide background information on the frameworks available that ease the data extraction process of machine learning processes as well as their limitations.

2.1 Related Work

Previous research shows that there is a lack of reproducibility in most git-based research projects [?]. Since then, some new frameworks and libraries evolved targeting the traversal of Git projects.

Common used frameworks to analyze the commit history of software repositories are RepoDriller [?] and PyDriller [? ?]. Whereas both tools simplify the traversing of Git commits, PyDriller is able to calculate some basic metrics like complexity and method count. The metric calculation is done within the library and could not easily be replaced with custom metric logic. Additionally, PyDriller states to be multithreaded and therefore improve the performance and lower the calculation time. A closer look into the library shows that this statement is only partially true: PyDriller is based on the library GitPython which is restricted by visiting only one commit at a time. This is caused by the underlying file-based Git archive that enables thread-safety by locking the files during e.g. check out operations. Hence it is only possible to parallelize the processing in the file dimension but not in the time dimension (the commit history). In other words, only the iteration over files within one commit and then checkout the next commit which can be parallelized. The same theory applied to RepoDriller which relies on the Java-based alternative JGit [?]. And besides, both tools allow no multi-repository mining and require additional orchestration and process management.

Other projects try to avoid direct file system access and transferring repositories in an additional database before the first analysis. One well-known project is GHTorrent [? ?] which allows various file-independent queries after importing the repository into a MySQL or MongoDB database. Unfortunately, the file content is not considered within this approach and only tooling for generating and importing is covered by them.

The project 'Public git archive' was introduced by researchers at SOURCED and enables the analysis of repositories from GitHub at a very large scale. They provide a lot of tools written in Go to query and also introduce a new storage format called Siva to reduce the stored size of forked repositories. Through a publicly available index file no additional processing is needed for developers. After the company was sold, the public available index file was taken down due to high costs and apparently was lost completely [?].

2.2 Limitations of the state of the art

Despite the notable effort spent so far, in our investigation into the matter we noticed a number of limitations of the existing frameworks that can represent challenges to be addressed.

2.2.1 Reproducibility challenge. Related work shows that reproducibility of the mining pipeline is a key feature to enable other researchers an easy reproduction and verification of the used dataset. Therefore, hosted datasets can become a nightmare since they require additional (financial) resources of the project holder and also are dependent on the life cycle of the initial project. If the project gets deprecated, all data could be gone like in the case of sourced. Mining data directly from GitHub may not resolve the problem of deleted or switched to private repositories and also introduce some overhead in terms of cloning and preprocessing but decreases the dependence to a hosting company or the creator of the project. Consequently, we will focus on the raw Git archive as a source of data.

2.2.2 Multi-dimensionality of Git data. At first glance, a GitHub repository looks like a tree-like timeline which contains different revisions and file versions along that axes. But from an algorithmic perspective, that can rapidly become an issue in terms of performance as every axis results in an iteration loop to reach at least each file once. This could also be described as multi-dimensionality of the Git data, as each additional dimension added to the mining process, will result more computation time and without considering the actual computation cost for detecting refactorings or calculating software quality metrics. Depending on the mining problem, we identified the following dimensions:

- **Time dimension:** Time dimension describes the total size of the analyzed commits. Git normally consists of one or more branches containing one main branch, denoted as *master*. When traversing over the master branch, one will get all branches that have been merged into the master. In our case, we assume that for refactoring detection it would be sufficient to detect all valuable refactorings that have been considered as useful by merging them back into the master branch. Therefore, the time dimension of the refactoring detection could be seen as linear. When an analysis problem requires visiting every commit in the repository, the time dimension is considered as the total count of commits within the whole repository.
- **File dimension:** Defined by the file count one commit has in total. This amount of files can vary significantly and often depend on the project size, age of the project or the used programming language. Additionally, it is important to note that the file dimension is different for each commit as a file can be added or deleted within a commit. If the latest commits has only a few files and all others have many, the file dimension will affect the performance more as if the last commit have many files and the others have only a few.
- **Structural node dimension:** Similar to the file dimension, the total number of interesting nodes, for example, classes, methods or code lines, are counted in the content of each file. Considering object-oriented languages, a file normally contains one class, but also could consist of more than one class declaration. If the mining problem is dependent on method or class member declarations, those would also be considered as a number of interesting nodes per file.
- **Metric dimension:** The last dimension is very specific particularly regarding the mining problem. In our example, be would like to extract refactoring operations and calculate software quality metrics. Therefore, the metric dimension is highly coupled with the structural node dimension and defined by the method the metric is collected. RefactoringMiner [?] uses an abstract UML algorithm to detect refactorings, while CK [?] is using an abstract syntax tree (AST). Those algorithms also add on top of the file or structural dimension, depending on how they are implemented and designed.

2.2.3 Parallelization challenge. One of the biggest challenges is building a mining tool, that allows scaling of the mining process along the presented dimensionality axes. Both presented tools, PyDriller and RepoDriller, do not or only partially support the parallelization. While RepoDriller does not do any kind of parallelization, PyDriller allows scaling along the file dimension axis. Files within a single commit can be analyzed by multiple threads and increase the performance and decrease the computation time as the use of those libraries shows [?]. As the multi-dimensionality of GitHub repositories outlines, we can not presume that repositories contain many files. When a repository contains fewer files but has a large revision history, the runtime of the mining process will get worse.

As already explained, the structural node dimension and the metric dimension are highly dependent on the type of analysis that should be executed. To make this approach generalizable, we focus on parallelization along the time and file dimension axes. As seen in PyDriller, the file dimension can easily be parallelized. In opposite, the commit axis is rather difficult: Since Git writes lock files when accessing the repository to avoid file inconsistencies by modifying the repository, only one commit can be visited concurrently. This limitation reflects in JGit and GitPython, which has all file-based actions marked as non-threadsafe. This limitation results in libraries not implementing a time dimension parallelization approach.

2.2.4 Generalization challenge. Even though we are conducting this distributed approach in the context of detecting refactoring operations, it is designed to be generalizable and can be reused for other types of mining problems. Since mining problems can vary from detecting developer behavior [], social aspects [] up to high technical questions like code evolution [] and smell detection[], it is not easy to build a pipeline that matches all type of problems. Therefore, this pipeline concentrates on technical, content-based repository mining. Additionally, libraries are not standardized in their input and output design. While RefactoringMiner takes the repository and commits as an input, CK is designed not designed Git-specific and takes just the source directory as an input. This requires the pipeline to adopt the corresponding libraries in terms of in- and outputs of each pipeline step and even more important to keep track of the memory usage, since the libraries itself write outputs to disk, while within a pipeline, results partially are stored in memory. This necessitate careful pipeline design and memory management to avoid out of memory exceptions.

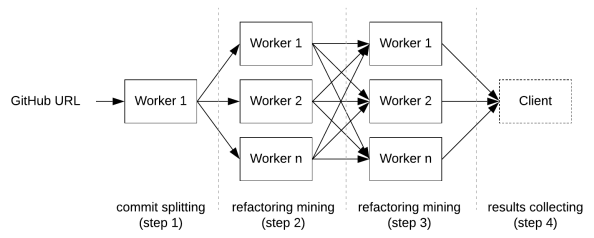
3 IRIS: A DISTRIBUTED FRAMEWORK TO SUPPORT MACHINE LEARNING RESEARCH

This section presents a distributed approach to support the data extraction process of machine learning approaches. For the sake of understandability, in the remaining of the paper we use the example of learning software metrics to predict the effect of refactoring operations on code quality: specifically, the example problem consists of computing software quality metrics before and after each refactoring applied on a certain system in order to assess the extent to which the refactoring has impacted on the values of those metrics. It is important to know, however, that the distributed approach proposed in this paper is general in scope, meaning that it can be exploited for any kind of data extraction activities related to the mining of software repositories.

We are using a MapReduce-inspired pattern for data distribution and Apache Spark for pipeline implementation. MapReduce was originally invented by Google and targets big data analysis and calculations by splitting the computation process into a map and a reduce phase [?] on a cluster of computation nodes. Those concepts are reimplemented and available as open source software in Apache Hadoop [?]. Spark improves this pattern by introducing resilient distributed datasets (RDDs) which kept mainly in memory, which results in a faster processing speed than the traditional, hard-drive storage based Hadoop system. Also, Spark allows a better fault tolerance due to directed acyclic graphs (DAGs) for computation tasks. In case one task in the graph fails, Spark can easily restart that specific task without canceling the whole computation.

The pipeline is basically split into three major computation steps: The first step extracts the refactoring operations from a list of given repositories. Spark is cloning one repository instance and scans for commits along the master branch. The list of interesting commits is split into batches of a definable size of k-commit and sent by shuffling to each of the other worker nodes.

Figure 1: Overview of the refactoring mining pipeline.


GitHub URL
Worker 1
Worker 1
Worker 2
Worker n
Worker 1
Worker 2
Worker n
Client
commit splitting (step 1)
refactoring mining (step 2)
refactoring mining (step 3)
results collecting (step 4)

In the second step, each worker starts cloning an separate instance of the repository if does not already exist and runs an instance of RefactoringMiner on the partial list of commits each worker has assigned. The previously described number of k-commits depends on the size of memory each worker node has available, as the results are kept in the heap until one batch hash finished the computation. Therefore it is important to adjust k accordingly: Is k set to high, Spark will fail caused by out of memory exception. Is k to small, the performance will decrease as instantiating an instance of RefactoringMiner is an expensive operation. This step outputs a list where each row consists of the repository name, the commit id of the refactoring operation and the type of the refactoring operation (e.g. move method, extract method...). The intermediate results are written as structured data using the Apache Parquet file format [?] to reduce storage size and improve performance.

The third step takes the shuffled list of refactoring commits identified in the second step and starts again with checking if the repository exists. If not, it gets cloned to that worker node. Now for every batch of refactoring commits an instance of CK is created and the parent commit of the given commit is identified. After that, both commits are checked out and CK calculates the metrics before and after the commit. Additionally, to reduce the computation time and avoid useless data processing, those files that are marked as changed are identified by the Git diff command and the metric calculation is only executed for that list of modified files. As a result, another parquet file is generated containing information about the repository, the involved commits, the refactoring operations (flattened by comma separation), the *side* of the operation (left = commit before refactoring, right = commit after refactoring), the file name, and the metrics.

3.1 Implementation details

Both libraries have not been very suitable to fit into our defined in and output format. Therefore they have been forked and slightly modified. RefactoringMiner only provides a method to detect a range of refactoring operations between two commit ids, though it found under certain circumstances more commits than actually existed in this range. Another method that identifies refactoring operations only in one commit was also not suitable in terms of performance and object instantiation. So we added a method that could accept a list of commits without instantiating each time a whole Git repository object.

As CK accepts only a source directory, we also forked and modified this library to take a list of interesting files within that directory. That allows the calculation of changed files only within the repository.

Besides, the calculation of metrics is designed as abstract metric processors, which improves the simplicity and extensibility of adding custom metric calculation libraries.

4 PRELIMINARY ASSESSMENT OF IRIS

4.1 Experimental Settings

The *goal* of the study is to provide a preliminary assessment of the performance of the proposed framework, with the *purpose* of understanding its potential usefulness for the data extraction stage of machine learning approaches. More specifically, we seek to understand computational time, which heavily depends on the structure and dimension of the analyzed repositories. Hence, we pose the following research question:

RQ$_1$. *To what extent can Iris improve the computational time with respect to a single-threaded baseline?*

Context selection. The *context* of the study is composed of the version history of the systems reported in Table 1. The selection process of those repositories is driven by their amount of commits available on the master branch (or more specifically, the branch which HEAD is pointing to). We assume that repositories with the most commits also contain a lot of refactoring operations. Additionally, projects are selected by their popularity and active contribution of developers. 1 shows auxiliary metrics to get a better understanding about the shape of the repository in terms of time and file dimension. Since the history is changing, we extracted information about the minimum and maximum amount of files as well as the mean number of files and the deviation. Time dimension is represented by the amount of commits available on the master branch.

Table 1: Repository Dimensionality Metrics

	Files			
Repository	**Min**	**Max**	**Mean**	**Commit**
google/guava	4	1862	1589.4	5295
apache/mahout	5	1245	4145.1	4417
reactivex/rxjava	65	3173	4878.9	5755
apache/parquet-mr	2	802	2013.4	2188
apache/commons-io	21	317	2107.9	2384

Evaluation methodology. For the runtime evaluation, a Spark cluster with a maximum amount of eight worker nodes is used. To observe the impact of distribution, each project is run with a different number of worker nodes w. Since no tool currently is capable of running a whole pipeline, the setting with only one worker node ($w = 1$) is used as an reference implementation for no parallelization. Each worker only makes use of one core, therefore the parallelization focuses on the distribution within the cluster. The jobs are executed with $w = 1$, $w = 2$, $w = 4$ and $w = 8$. To provide a better insight in the measured runtimes, we also collected additional metrics about the each step (refactoring mining and metric calculation) and the amount of output data generated by each repository. Additionally, we cloned and prepared the repositories before each run to avoid the bias of the network connection.

Evaluation hardware setting. The evaluation is done on a single server running multiple virtual machines, each representing a Spark node. The machine has four Intel Xeon E-7 4850 deca-core CPUs and 64GB of memory. Based on this machine, each job is executed with a limit of 7GB of memory to keep space for the master node, the virtual machine and the Java virtual machine overheads.

4.2 Analysis of the results

Figure 2 shows the relative runtime per each repository and by increasing the number of worker nodes in the cluster as described before. All projects are benefitting from the parallelization approach, especially when between one and four worker nodes. When using eight worker nodes, the impact is less. The Apache Mahout project is the only one that gets even worse.

Figure 2: Relative runtime per project and worker w

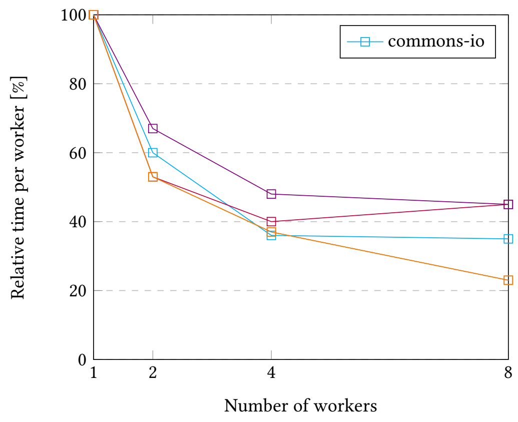


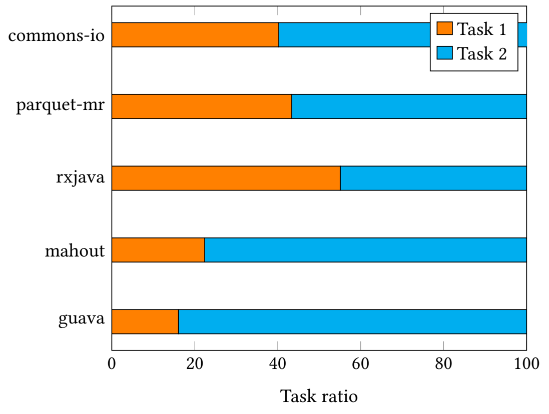
Figure 3 visualizes the consumption of computation time between task 1 and task 2 over all projects. We decide to use the mean value over all w-values since the ratio is more or less the same for all w. Task 2 is in the most cases more dominant in computation time consumption than task 1.

Table 2 contains information about the amount of found refactoring operations and metrics.

Table 2: Detected refactorings and metrics per project

Repository	Refactoring operations	Metrics
google/guava	24732	18148
apache/mahout	16471	20861
reactivex/rxjava	41391	5654
apache/parquet-mr	5459	5500
apache/commons-io	3482	1517

Figure 3: Mean ratio between task 1 and task 2 for each project



We also discovered that in parallelized mode some worker nodes have a longer calculation time than others. Therefore, the variance is calculated for *w = 8* and task 1 and 2 to get an impression about how much both tasks vary in their single runtime (table 3).

Table 3: σ^2-comparison between Task 1 and Task 2 for w = 8

Repository	Task 1	Task 2
google/guava	24.3	8.6
apache/mahout	5.3	2.0
reactivex/rxjava	73.0	66.9
apache/parquet-mr	0.6	0.3
apache/commons-io	0.4	0.5

Ultimately, we consider the throughput which is basically the number of outputs over the computation time needed to produce those output values. The throughput should result in an understandable visualization of the overhead introduced through parallelization. We slightly modified the throughput calculation showed below where *wid* is the id of the worker available, *r* is the amount of refactorings and m the amount of metrics.

$$throughput = \frac{\sum_{wid=1} runtime(wid)}{r+m}$$

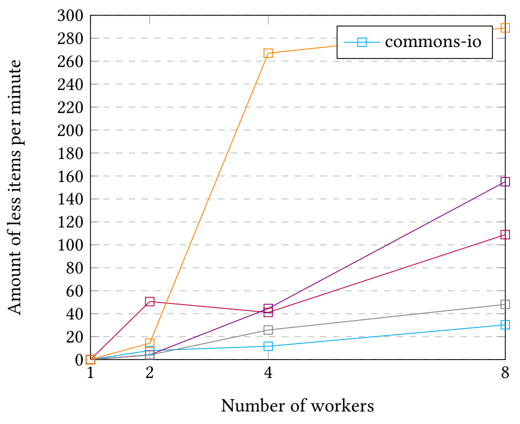
Throughput is calculated by the accumulation of all single worker tasks. On a single node cluster this is just the maximum runtime. In parallelized mode, the time of each worker is accumulated, even when those workers are running concurrently and the effective time is shorter. How many items less can be calculated when using parallelization is presented in table 4.

5 DISCUSSION AND LIMITATIONS

5.1 Discussion

The results in 2 are showing that parallelization of the time dimension is has a positive impact on the overall runtime. The biggest impact is introduced when using two and four workers, eight workers minimize the runtime only by a very small amount and in case

Figure 4: Throughput



of Mahout even add additional time. This effect can be explained by the nature of Spark: While jobs can run concurrently, Spark waits until one computation step, in this case task 1 and 2, are completed. Because of reducing the heap usage, the Git history is split in batches of size *k* which are typically ordered chronological. Since the amount of files is mostly increasing over time, the first batches have less computational effort than the ones with new commits. Therefore some workers just do nothing until all jobs are finished. Strengthened by table 3, the variance indicates that all job runtimes of task 1 are much more varying than task 2. The second task gets more homogeneous batches since for each refactoring commit one metric can be calculated.

The task ratio table 3 represents the way of the repository usage. Assuming that detecting refactorings is more effortful than only calculating some software quality metrics, task 1 is mostly less computational intense than task 2. The difference is how both libraries are accessing files: RefactoringMiner reads file contents directly from the object database, while CK needs a file system as input. Therefore, the repository has to be checked out before each metric calculation. This obviously takes more time than just reading the object database.

Finally, the throughput in figure 4 shows that even when the effective runtime is decreased by parallelization, there is still some overhead introduced through Spark and data redistribution. This overhead is expressed by the number of less items that can be processed. By using the accumulated time, the waiting time described earlier is not considered.

5.2 Limitations of the framework

Git locking. The biggest limitation of the IRIS implementation is the Git repository itself. The locking mechanism forces to clone the repository multiple times when parallel mining is intended. Therefore each worker node is using only one core (and therefore is single threaded) while in most Spark cluster scenarios each worker is capable of multi-threading and has typically more than one core. A local repository replication would allow parallelization on the worker itself but may also impact disk read negatively when accessing the repositories simultaneously.

Batching. Batching is needed to avoid out of memory exceptions in the heap. It is also intended to reduce the instantiating frequency of each mining library. Nevertheless, especially task 1 shows that the divide and conquer approach using batches can impact the runtime negatively when one worker finishes the mining process much earlier than the others and parts of the cluster are running while doing nothing.

Generalization. Currently the prototype is limited to only two usecases: Mining refactoring operations and calculating the change of software quality metrics. This limits the usefulness and adaptability for other usecases. Hence, the steps are only slightly coupled through the list of commit hashes but there are no definitions about what input and output formats are available. Additionally, the prototype shows the different repository access used by CK and RefactoringMiner. A non-standardized repository access impacts the distribution behavior and overall performance as seen in the fine-grained runtime analysis.

Spark UDF. Using a Git repository directly on each worker is in general an anti-pattern when writing Spark jobs. While the mining operations are running within one node of the DAG, it can be seen as a user defined function (UDF). Those functions are more or less black boxes for Spark as it does not know the internal calculations and can not optimize the DAG with techniques like query optimization.

File dimension parallelization. Unlike PyDriller, the prototype does not yet make use of parallelization of the file dimension.

6 CONCLUSION AND FUTURE WORK

The presented prototype of the IRIS framework clearly shows that parallelization of the time dimension can increase the performance in MSR. We also discussed the major issues that affecting the runtime negatively and analyzed the current problems of the implementation. Therefore, there is a lot of potential improving the prototype in terms of parallelization as well as generalization. The the following we are presenting further steps to optimize the framework and make it more flexible for other mining tasks.

Replacing the file-based repository by database. This step is needed to avoid problems introduced by file-based Git repositories and the described locking mechanism. This approach is inspired by the GHTorrent project [**? ?**] and the Sourced project [**?**]. Unlike GHTorrent, we'd like to additionally store file contents together with meta-information about the project in a large database. To avoid hosting problems like in Sourced and improve reproducibility, our pipeline should support the automatic generation of this database by a given list of interesting repositories. It should be evaluated, which storage format is most suitable in interoperability with Spark (e.g. Apache Cassandra [**?**], Neo4J [**?**] or just simple parquet files [**?**]).

Using a database as input not only avoids the file lock in git repositories, it also is likely to improve the overall performance since Spark can use query optimization and higher data distribution since it is not limited to file-based data sources. Also, basic operations like counting commits along a branch can be done much faster in a structured and pre-processed environment. The pre-processing overhead has to be evaluated.

Generalization through standardized in- and outputs Building a pipeline upon Apache Spark improves a standardized design of each analysis step. Building well-defined in- and output formats for each processing step raise the level of re-usability of certain processing steps and allow further research much easier adoption of other research questions.

Graphs by default Spark enhances big data analysis by integrated support for graph data and adjusted, for distributed environments optimized graph algorithms [**?**]. If the dataset provides access to graph-like structures makes answering research-questions in the area of developer-interaction much easier.

Additionally, it could be considered to store source code information not as plain text but as graph data using abstract syntax trees (AST). Smaller projects have already constructed such use cases in combination with Neo4j [**? ?**].

Reproducibility by default Through pre-processing, the data becomes much easier to deal with and besides, there is no dependency on existing datasets because it can be generated by everyone just by accessing the public Git API. Furthermore, Spark allows us to run on common cloud platforms like Amazon AWS, Azure, or Google Cloud as well as on local clusters or even on a developer's laptop. This makes executing the pipeline fairly easy.