Tampere University

Nguyen Quoc Hung

DATA PLATFORM FOR ANALYSIS OF APACHE PROJECTS

Bachelor of Science Thesis
Faculty of Information Technology and Communication Sciences
Davide Taibi
Nyyti Saarimäki
April 2020

ABSTRACT

Nguyen Quoc Hung: Data Platform for Analysis of Apache Projects
Bachelor of Science Thesis
Tampere University
International Degree of Science and Engineering (B.Sc)
April 2020

This Bachelor's Thesis presents the architecture and implementation of a comprehensive data platform to fetch, process, store, analyze and finally visualize data and statistics about open source projects from the Apache Software Foundation. The platform attempts to retrieve data about the projects from the official Apache organization Jenkins server and Sonarcloud online service. With a huge community of contributors, the projects are constantly evolving. They are continuously built, tested and static-analyzed, making the stream of data everlasting. Thus, the platform requires the capability to capture that data in a continuous, autonomous manner.

The end data demonstrate how lively these projects are compared to each other, how they are performing on the build, test servers and what types of issues and corresponding rules have the highest probability in affecting the build stability. The data extracted can be further extended with deeper and more thorough analyses. The analyses provided here are only a small fraction of what we can get out of such valuable information freely available out there.

Keywords: open source software, data platform, data processing

PREFACE

I would like to sincerely thank Professor Davide Taibi and Doctor Nyyti Saarimäki for their guidance, constructive comments and feedback. It would not be possible to finish this thesis without their excellent help and suggestions.

Tampere, 29 April 2020

Nguyen Quoc Hung

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

ASF	Apache Software Foundation
API	Application Programming Interface
CI	Continuous Integration
CSV	Comma-separated Values
DAG	Directed Acyclic Graph
DB	Database
DF	DataFrame
ETL	Extract Load Transform
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
PR	Precision Recall
PRA	Public Repository Analysis
RDBMS	Relational Database Management System
REST	Representational State Transfer
ROC	Receiver Operating Characteristic
SQL	Structured Query Language
UI	User Interface

1. INTRODUCTION

In the world of software, the term “open source” refers to the fact that the software projects are allowed, by their creators, to be modified, contributed and used by any individuals regardless of their intention or, in other words, their source code is open. These software projects can be overseen either by individuals or by large, prestigious foundations. These foundations operate in a charitable, non-profit manner, with an aim to foster the growth of open source software development. Some of the most popular names are the Linux Foundation and Apache Software Foundation.

The Linux Foundation was founded in 2000 to foster the growth of Linux, the most used operating system and a symbol of open source software movement, developed by Linus Torvalds and under open source licensing [21]. With over 1,119,785,328 lines of code committed, 7600 volunteer committers, 350 active projects and thousands of projects in total, the Apache Software Foundation is the world’s largest open source foundation. It manages over $20 billion worth of software products, which are all contributed by the community at no cost and provided to millions of users freely [5].

Software projects under the management of the Apache Software Foundation are mostly hosted on GitHub. Contributors directly contribute through the project's repositories. The huge numbers of repositories and community developers result in a multitude of commits of code to ASF every day. Keeping track of the mileage of one or just a couple of projects is easy by examining the insights into a repository provided by GitHub. Thanks to their essence of being open, not only the progress of the source code, other data like their performance on build/test servers, issues, code reviews or static code analyses are, as well, readily available. This is solely feasible per a relatively small number of projects, what if we want to keep up to date with all of these statistics from hundreds of projects or compare them with each other in the context of some attributes. This could prove to be largely helpful, for instance, to identify what kind of technologies are leading the chart in popularity among volunteer contributors, for researchers to understand and analyze the factors that may lead to poor or outstanding performance of a project on their build/test servers, or, for developers to draw experience about what could produce low-grade static-analyses… Just like any other kind of data, there is no limit on the amount of valuable knowledge that we can extract. However, this task is next to impossible

without a well-architected platform that can harness the gigantic sources from the open source projects.

The goal of this thesis is to demonstrate the architecture and the process of constructing such a platform. The process includes developing individual components separately, coordinating the parts to form a functional system and finally analyze the resultant data to get precious knowledge about the Apache projects.

Chapter 2 provides fundamental knowledge about the data of interest from ASF projects, essential concepts about such an application, as well as basic ideas of the tools employed in the platform and what role they play in the whole picture. Chapter 3, utilizing the tools presented from the previous chapter, demonstrates the actual, concrete implementation of each tool, and how they are assembled and orchestrated. Chapter 4 shows a prototype of what can be expected from the platform, such as some visualization and analyses of ASF projects. Chapter 5 concludes about the performance and usage of the platform and suggests some ideas for improvements to extend the boundaries even further.

2. BACKGROUND

This chapter lays a material and technological background foundation for the whole platform. It starts with the data sources of interest and then moves on to the technology stacks employed to build the platform from scratch.

2.1 Data Source

2.1.1 Jenkins

The first source of data for our platform is the official Jenkins server of Apache. To understand what Jenkins is, what problem it solves in the cycle of a software project, we need to have a grasp of the idea of Continuous Integration. It stems from the desire of the software company to stay competitive in the software market by being able to ship new updates and features of their application to the customers in a fast and timely manner. To achieve this, they encourage members of a development team to continuously integrate their new, developed codes. Their integrations are then verified in an automated manner on a separate build/test server. This server also detects errors and defects in the code as quickly as possible giving the software developer feedback about their work [17]. Detecting the issues earlier helps to reduce the cost in the future. Since this practice leads to frequent code merger, therefore, if the code is tested to be erroneous, or breaks the working version, it is simpler to fix this small integrated chunk rather than a big chunk of code due to long interval integration [16]. Some of common practices for Continuous Integration are automated builds, a widely covered test suite and frequent commits to the mainline branch. There is a multitude of tools supporting Continuous Integration, namely Travis CI, Bamboo, Gitlab CI, Circle CI… and Jenkins.

In that picture of Continuous Integration, Jenkins plays a vital role as the build/test servers. Jenkins is an open source project written in Java and stems from project Hudson from Oracle. "In 2009, Oracle purchased Sun and inherited the code base of Hudson. In early 2011, tensions between Oracle and the open source community reached rupture point and the project forked into two separate entities: Jenkins, run by most of the original Hudson developers, and Hudson, which remained under the control of Oracle" [25]. Its main task is to automate the building of software, run tests and report outcomes and any

detected errors, issues based on pre-set criteria. Jenkins offers certain advantages that help it remain popular. The first one is that it is open source, open to modification and use under zero costs. It is highly scalable through a master-slave topology of servers, and highly extensible due to the variety of plugins and ease to develop plugins in Java [13]. Last but not least, its community of contributors and users is huge, reactive and dynamic [25].

This application's first data source is the official Jenkins server of the Apache Software Foundation. What is really of interest is the build information from the jobs of the projects. Each job upon finishing a build will have a set of attributes like the build result, whether it is a SUCCESS, FAIL, UNSTABLE or ABORTED, duration and estimated duration of that build, the number of tests passed, failed, skipped and the total duration, the revision of the project at the time of building, the latest commit id and its timestamp... These are the attributes that we focus on, although there can be a multitude of other statistics for more profound analyses.

2.1.2 SonarQube

Throughout the process of software development, it is always desirable to improve the quality as well as the security of source code. SonarQube, which is written in Java and also has open source roots, performs continuous code inspection, or static code analysis [27]. Static code analysis is the process in which the source code is analyzed in a non-runtime environment, meaning without executing the code. Static code analysis programs are called checkers. “They read the program and construct some model of it, a kind of abstract representation that they can use for matching the error patterns they recognize. They also perform some kind of data-flow analysis, trying to infer the possible values that variables might have at certain points in the program. Data-flow analysis is especially important for vulnerability checking, an increasingly important area for code checkers” [22].

Designed to be embedded into existing workflows, SonarQube offers Continuous Integration and Continuous Delivery integration and supports 27 programming languages [27]. It is built on the core Seven Axes of Quality: design/architecture,

duplications, comments, unit tests, complexity, potential bug, and coding rules [14]. SonarQube has a variety of metrics which is sub-divided into 9 domains: Complexity, Duplications, Issues, Maintainability, Quality Gates, Reliability, Security, Size and Tests. We will only discuss important concepts of the SonarQube platform.

First of all, rules are what acts on the source code to produce issues. Users can create custom rules or utilize existing ones. If the code breaks a rule, it generates an issue. Issues fall into 3 categories Bug (domain: Reliability), Code Smell (domain: Maintainability) and Vulnerability (domain: Security). There are 5 levels of severity for an issue ranging from INFO, MINOR, MAJOR, CRITICAL to BLOCKER, depending on how likely it will affect the performance of the program. Quality gates are created by setting a threshold of metrics on which the projects are measured. If the project meets the required threshold, it passes the quality gate. This helps enforces a quality policy across projects in the same organization.

Apache Software Foundation carries out static code analysis at Sonarqube's online service called Sonarcloud. A project is a single object to be analyzed by the service. Each time a project is analyzed, Sonarcloud records it as an analysis. A project can have up to a couple up to 30 analyses throughout their lifetime on Sonarcloud. Each analysis comes with a set of issues, that are either removed or introduced, indicated by their status. Additionally, other attributes of an issue like its severity, resolution, type, the rules associated, creation and update date are also of interest. Besides issues, the measures of an analysis are also an essential aspect. The measures including cognitive complexity, coverage rate of unit tests... are an excellent indicator of the quality of projects. There can be over 100 measures for each analysis, although not all of them have a valid value. This platform tries to fetch all of the measures available. Analyses, issues and measures are the three main facets of Sonarcloud.

2.2 Tools and Services

2.2.1 ETL

The core of this data platform is the ETL (Extract, Transform, Load) process. In short, during this process data is taken from a number of sources (extract), applied certain

transformations to fit in with a certain schema, or to make it appropriate for analysis (transform) and loaded to a target system, usually a data warehouse (load). ETL became popular as a result of an increase in both the amount and heterogeneity of input sources of data. The data can be in any kind of format from structured, unstructured, tabular, text, binary data, image, video, audio. ETL aims to manage different types of data and integrate them to gain a consolidated view from the data for important decisions [12][24].

There exist a lot of tools for ETL and the most dominating ones have a Graphical User Interface for developers to visually interact with components of the process. As with any GUI-based tools, they are appealing and have a low learning curve. However, they are prepared tool or a "piecemeal" and only cater to a limited number of scenarios [1]. This gives rise to designing ETL by programming. This approach has a steep learning curve as you have to learn at least one programming language to create an ETL pipeline. However, once you own such a skill, the flexibility is endless. Take Python for instance as a programming language, there are third-party libraries for anything one can think of: drivers for all kinds of databases from structured to no-structured, libraries to work specifically with images and audio files, or to integrate high speed, multi-node processing engine… Coding ETL pipeline ensures there is no corner case. The whole platform itself is an instance of ETL process. Data is first extracted from the sources, then transformed into adequate form and undergone computation heavy processing and finally loaded into the backend database.

2.2.2 Data Processing with Apache Spark

Apache Spark is an open source distributed large-scale data processing engine. The engine aims to be lightning-fast and general purpose. Its speed is achieved by extending the Map Reduce framework for cluster computing of extremely large datasets. In short, the workload is divided among a cluster of nodes for computation and then reassembled for the outcome. The general-purpose nature is expressed by the fact that Spark allows for different types of processing from Batch to Streaming, interactively executing SQL commands directly on the datasets or applying Machine Learning on the datasets. Thanks to this generality, it is easy for users to integrate various processing types in the same platform. Spark is written in Scala but provides high-level API in Java, Scala, Python and R, making it extremely friendly to developers, data scientists and data engineers [20].

Together with its core, the framework ships with 4 other components: Spark SQL, Spark Streaming, MLlib and GraphX. Spark SQL assumes the data is structured and stores them in a DataFrame, users can run interactive SQL queries on the data in Spark [10]. Spark Streaming facilitates the development of streaming applications [11]. MLlib provides a wide range of Machine Learning algorithms and components to build a complete Machine Learning pipeline from feature extracting, model training, hyper-parameter tuning, model evaluation to saving and loading Machine Learning models [9]. Finally, GraphX offers computation of graph data [8].

Spark is designed to run on a distributed system like Hadoop. In such a cluster, there are popular cluster managers like YARN and Apache Mesos, and Spark can integrate with them well. Alternatively, Spark can also run in a standalone mode using something called a standalone scheduler, which is Spark's own cluster manager [20].

Spark plays a central role as a data processing step in this platform. It is deployed in a single node standalone mode since the data at hand is not at the scale of a several-node cluster. However, the distributed computation nature of Spark is still taken advantage of by the utilization of multi-cores from the host machine. It is possible due to the fact that libraries and algorithms that Spark provides are implemented with the distributed-computing paradigm in mind. Not only does Spark process the data to transform it into an adequate form, but it also applies end-to-end Machine Learning processes on the data to produce fully functional models that are highly capable to predict future outcomes.

2.2.3 Workflow Management and Scheduling with Apache Airflow

Once the ETL script is ready, it needs to be executed automatically in a scheduled interval. “Apache Airflow is a platform to programmatically author, schedule and monitor workflows” [2]. Written in Python, this platform allows users to create workflows in the form of DAGs. DAG , Directed Acyclic Graph, is a graph of nodes and edges, the edges have a direction from one node to another and it is ensured that “no nodes connect to any of the other nodes already in their series” [15].

These DAGs are the Python files in which the tasks correspond to the nodes in the DAG, and the directed edges are expressed through the dependencies between the tasks. Other important configurations for a workflow like when and how often to execute the pipeline, what to do in the event of malfunction… are also defined in the Python file. As with designing ETL by programming, scheduling and developing workflows with Airflow allows for great dynamics and flexibility in easily defining new categories of tasks (operators) or new executors of the tasks [2].

Two core parts of Airflow platform are the scheduler and the Web UI. The scheduler, which runs as a persistent service on the host, manages all the DAGs defined in the system and all of their tasks. It will execute the individual task instances once all the dependencies and requirements are met [3]. The Web UI provides an interactive way to oversee the performance and status of all the workflows on the system. Some other management tasks can also be done via the Web UI like creating or modifying connections to other services that the DAGs use, setting variables which Airflow uses to increase flexibility [4].

The whole ETL process relies solely on Airflow to trigger its processing. Airflow makes sure that the data extraction tasks are finished before the processing part is initiated. Any failure in the extraction stage will suspend the processing stage until there is a successful retry. Otherwise, the subsequent steps are canceled, ensuring the integrity of the whole process. Airflow schedules the run of the platform at an exact time every day, keeps a close monitor, and reports on the performance of the platform.

2.2.4 RDBMS using PostgreSQL

PostgreSQL is the most advanced open source Relational Database Management System (RDBMS) [23], a system specifically for relational databases. Relational databases work with only well-structured data or data with a specific schema. It stores data in rows that contain fields corresponding to the columns of a table. The tables within a database share a relation in a sense. This connection allows queries to be executed against multiple tables at a time [19].

PostgreSQL database is the destination of the processed data. It serves as the data persistence step. Spark connects directly to PostgreSQL to ingest new data and load old

data for training and testing of Machine Learning models. Another connection to the database is the visualization tool, Apache Superset, mentioned in the next section. Beside the main data about the open-source projects, PostgreSQL also hosts metadata so that the tools in the application, Apache Airflow, Apache Superset, are functional by providing an isolated database in the system for each of the tools.

2.2.5 Data Visualization with Apache Superset

Data is stored persistently in the form of tabular data. To help viewers to easily understand and make sense of the data, a visualization tool is imperative. Apache Superset is an incubating ASF business intelligence web application that provides a simple and straight forward approach to data visualization and exploration. It allows integration with most Relational Database Management System including PostgreSQL or even data within Spark SQL [6].

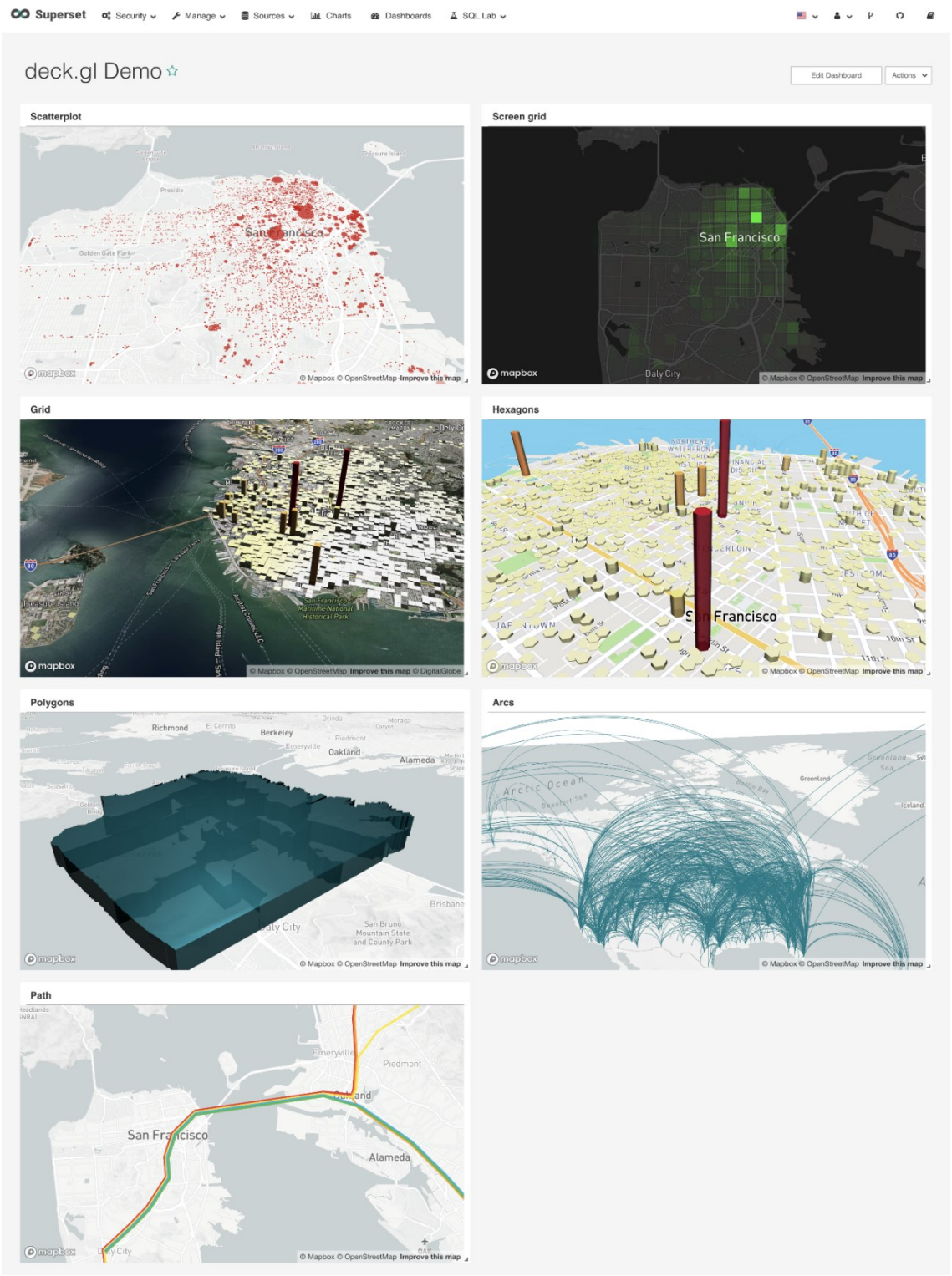

Figure 2.1. Sample report dashboard from Apache Superset

3. IMPLEMENTATION

3.1 Data Extraction

The first step in any data-driven application is to work with the data source. In this platform, we have two main sources of data. The first one is from the public build and test server for projects of Apache Software Foundation, available at *https://builds.apache.org/*. The second data source is the Sonarcloud online service, available at *https://sonarcloud.io/organizations/apache/projects*. It is important to realize that not all of the ASF projects are on these two services.

The data from these sources are exposed through a REST API. However, the returned response is in the form of JSON and is not at all ready for any type of analysis. Therefore, scripts to transform these JSON data into a tabular form are required. Although this phase is called 'Data Extraction', there already involves some processing of raw input into a more adequate shape to be stored on the file system. The scripts in this phase are written in Python version 3.7, any other programming languages will be able to achieve the same goal.

3.1.1 Jenkins Extraction

Both of the data sources expose data through their REST APIs, therefore, simply using the HTTP to request the data from the Jenkins server will do the job. However, there is a third-party library in Python, that acts as a higher-level wrapper over the Jenkins server's REST API and allows easy interaction with the Jenkins server. It is called 'python-jenkins' [18]. It can operate on any Jenkins server including the Apache server, by providing the link to that server. This library provides a wide range of functionality to control and interact with Jenkins such as create, copy, update, delete jobs or nodes, control the builds of jobs. However, our main intention with this library is purely to get data about the jobs and their builds and it plays a central role in the script.

It is often easier to first determine what we want as the output. In general, what we look for from the Jenkins server is, certainly, the builds and tests information about the jobs. There are two obvious functions from the library for the task:

get_build_info(name, number, depth=0)
get_build_test_report(name, number, depth=0)

Both of the functions take name of the job, number of the build and the depth level of data. Through exploratory data analysis, the maximum depth is 2, greater numbers produce the same results. From what the functions return, the structure of the output files can be decided:

```
JENKINS_BUILD_DTYPE = OrderedDict({
    "job" : "object",
    "build_number" : "Int64",
    "result" : "object",
    "duration" : "Int64",
    "estimated_duration" : "Int64",
    "revision_number" : "object",
    "commit_id" : "object",
    "commit_ts" : "object",
    "test_pass_count" : "Int64",
    "test_fail_count" : "Int64",
    "test_skip_count" : "Int64",
    "total_test_duration" : "float64"})

JENKINS_TEST_DTYPE = OrderedDict({
    "job" : "object",
    "build_number" : "Int64",
    "package" : "object",
    "class" : "object",
    "name" : "object",
    "duration" : "float64",
    "status" : "object"})
```

Listing 3.1. *Structure of output CSV files of Jenkins extraction script*

This listing displays the dictionaries containing the fields as keys and their respective data type as values, where “object” simply means “string”. There are a set of arguments, passed as command line arguments, to customize how the script behaves. However, the default arguments will do just fine.

```
usage: fetch_jenkins_data.py [-h] [-o OUTPUT_PATH] [-b] [-p PROJECTS]

Scrip to fetch data from Apache Jenkins Server at https://builds.apache.org/

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT_PATH, --output-path OUTPUT_PATH
                        Path to output file directory, default is './data'
  -b, --build-only      Write only build data.
  -p PROJECTS, --projects PROJECTS
                        Path to a file containing names of all projects to
                        load, if not provided, load data from all jobs
                        available on the server.
```

Figure 3.1. *Command line description of Jenkins extraction script*

The script operates primarily in two modes depending on the starting point. By default, the program initiates itself to fetch data from all the jobs now available on the Jenkins server, therefore, the first step is to try to get all the job's names and their build numbers. Alternatively, -p/--projects, followed by a path to a file containing the project's names, will only fetch data from those jobs belonging to the projects. It is not an apparent task to determine which jobs are from a certain project. However, the library comes in handy with a function that lists all jobs whose names match a regular expression containing the project name. It is critical to realize that a project may contain hundreds of jobs on Jenkins server, thus, not all the builds from a particular project lands in the same file but may end up in different files depending on the jobs. Most of the time, the program is run in the default manner.

```
def process_jobs(name, is_job, server, first_load, output_dir_str ='./data', build_only = False):

    for job_info, latest_build_on_file in get_jobs_info(name, server, is_job, output_dir_str= output_dir_str):

        latest_build_on_file = -1 if latest_build_on_file is None else latest_build_on_file
        fullName = job_info['fullName']
        print(f"\tJob: {fullName}")

        builds = []
        #get builds info:
        for build in job_info['builds']:
            build_number = build['number']
            if build_number <= latest_build_on_file:
                continue
            try:
                build_data = server.get_build_info(fullName, build_number, depth=1)
                builds.append(build_data)
            except JenkinsException as e:
                print(f"JenkinsException: {e}")

        builds_data, tests_data = get_data(builds, fullName, server, build_only)
        print(f"{len(builds_data)} new builds.")

        df_builds = None
        if builds_data != []:
            df_builds = pd.DataFrame(data = builds_data, columns=list(JENKINS_BUILD_DTYPE.keys()))
            # Explicitly cast to Int64 since if there are None in columns of int type, they will be implicitly casted to float64
            df_builds = df_builds.astype({
                "build_number" : "Int64",
                "duration" : "Int64",
                "estimated_duration" : "Int64",
                "test_pass_count" : "Int64",
                "test_fail_count" : "Int64",
                "test_skip_count" : "Int64"})

        df_tests = None
        if tests_data != []:
            df_tests = pd.DataFrame(data = tests_data, columns=list(JENKINS_TEST_DTYPE.keys()))
            df_tests = df_tests.astype({"build_number" : "Int64"})

        write_to_file((fullName,df_builds, df_tests), output_dir_str, build_only)
```

Listing 3.2. *Basic operation of Jenkins extraction*

The basic operation of the script is as follows: with the job name and the build number, get data about the build, extract essential attributes and append into a Pandas

DataFrame called *df_builds* which contains data about all builds from a single job. Similarly, there is a DataFrame called *df_tests,* which has data about the test report of one build and iteratively updated with every build. After finishing extracting of all builds from a job, these DataFrames are written to CSV files, under the name: *[JOB_NAME]_builds_staging.csv* or *[JOB_NAME]_tests_staging.csv* in the respective builds/tests folder in the *output_dir_str* directory.

However, there is one requirement for the script. It is meant to update the existing files with new data every day instead of querying the server all over again or “incremental load”. Experimentally, one pass over the whole server can take up to 4 hours to retrieve only the build information. Whereas the test output file, if there exist test reports from the server, is multiple times larger than the build file for the same job. This means fetching everything from the server every day is not a viable solution. There needs to be a mechanism to load only the new data from the server. To achieve this, we need to know if a build is considered new by retrieving the latest build from the CSV files. Now the script has to assume that the existing CSV files are in the *output_dir_str*, then it reads the respective build file of a job by deciding the file name from the job name. The latest build number is simply the largest one. With this number, we can easily determine whether a build is already recorded in the file.

There is still one challenge. We try to avoid querying the server for everything due to the huge amount of data there. In a similar manner, the data processing program, in the next phase, needs to be able to identify which are the newly extracted data. If we do not draw a clear distinction between already processed and unprocessed data, some of the data will be treated multiple times leading to redundancy. That is the reason why the output CSV files of the extraction script have "_staging" ending, to mark that they are new and unprocessed. Those that already undergo processing do not have this ending, only *[JOB_NAME]_builds.csv* or *[JOB_NAME]_tests.csv*. Therefore, these build files are used to obtain the latest build number of jobs that are processed. At the end of the pipeline, there comes a stage to handle the merger of the files of the same job, which will be discussed in the next section of the chapter.

3.1.2 Sonarcloud Extraction

Unlike Jenkins data source, there is no Python wrapper library to interact with the Sonarcloud online service. Therefore, it is required to perform HTTP request to retrieve

data from the server REST API. Moreover, this is not an Apache own server, we need to request the server for projects under ASF with the key *organization* and value *apache*. The API also provides a variety of functions for controlling, administering, or authorizing according to the user's access rights. We mainly focus on retrieving data that a free user can directly access [28]. The response to the request contains data in the form of JSON.

Although the server exposes a multitude of endpoints, only 5 of them are used in this extraction script. The first step is to retrieve the projects under ASF, this information is available at endpoint *api/components/search* with arguments about the organization and type of components which are projects. As mentioned earlier, the three facets of Sonarcloud that we focus on extraction are the analyses, measures and issues. They are tightly connected as each analysis of a project produces a set of measures and issues. Data about analyses are exposed at *api/project_analyses/search,* which returns all analyses of a project in chronological order. Each entry contains the project name, analysis key, date of analysis, the version of the project and the revision string of the project at the time of analysis. The revision string is vital since it will be used later to join with the Jenkins data on its build's *revision_string*. The below listing shows the structure of the output CSV files of analyses as a Python Ordered dictionary.

```
SONAR_ANALYSES_DTYPE = OrderedDict({
    "project" : "object",
    "analysis_key" : "object",
    "date" : "object",
    "project_version" : "object",
    "revision" : "object"
})
```

Listing 3.3. *Structure of output CSV files of Sonarqube analyses*

Having the data on analyses, we can move on to extract measures and then issues data. At *api/metrics/search*, we can get all the metrics that the online service produces. However, not all of the metrics are used for a project, therefore, most of the measures will be just empty. It is hard to decide what metrics should be taken into account since some of them are used in only certain projects. Thus, it is decided that all of the metrics are recorded. Nevertheless, there is one exception, the metric *sonarjava_feedback,* a long piece of text which easily raises a lot of exceptions during processing and may not have a lot of meaningful data, are, therefore, left out.

Once the projects, together with their keys, and the metrics of interest are available, the data can be retrieved at *api/measures/search_history*. The measures are shown per metrics, which contains the date of analysis and the value for the metrics, thus, the number of values is equal to the number of analyses and the values are listed in chronological order. Afterward, the measures are concatenated since only 15 metrics can be used per call of the *measures* endpoint, which are then all joined with the *analysis_key* to form a DataFrame. This DataFrame is then written to a CSV file containing the project name, analysis key and all of the fetched measures.

The part of the program to extract issues data is more complicated. This is due to the fact that the API endpoint that we use for this task, *api/issues/search,* does not have a chronological order similar to analyses or measures endpoints. Instead, it lists all the issues of a particular project containing certain attributes. These attributes can be divided into two categories. The first one is the analysis-related attributes, the second one is the remaining attributes. *updateDate* and *creationDate* are the analysis-related attributes since they will be used to determine at what the analysis keys the issues are produced or updated. The attributes from the second category are also recorded. The listing below shows the structure of the CSV files produced for the issues.

```
SONAR_ISSUES_DTYPE = OrderedDict({
    "project" : "object",
    "current_analysis_key" : "object",
    "creation_analysis_key" : "object",
    "issue_key" : "object",
    "type" : "object",
    "rule" : "object",
    "severity" : "object",
    "status" : "object",
    "resolution" : "object",
    "effort" : "Int64",
    "debt" : "Int64",
    "tags" : "object",
    "creation_date" : "object",
    "update_date" : "object",
    "close_date" :  "object"
})
```

Listing 3.4. *Structure of output CSV files of Sonarqube analyses*

```
def process_project_analyses(project, output_path):

    project_key = project['key']

    output_path = Path(output_path).joinpath("analyses")
    output_path.mkdir(parents=True, exist_ok=True)
    staging_file_path = output_path.joinpath(f"{project_key.replace(' ','_').replace(':','_')}_staging.csv")
    archive_file_path = output_path.joinpath(f"{project_key.replace(' ','_').replace(':','_')}.csv")

    last_analysis_ts = None
    if archive_file_path.exists():
        try:
            old_df = pd.read_csv(archive_file_path.absolute(), dtype=SONAR_ANALYSES_DTYPE, parse_dates=['date'])
            last_analysis_ts = old_df['date'].max()

        except ValueError as e:
            print(f"\t\tERROR: {e} when parsing {archive_file_path} into DataFrame.")

        except FileNotFoundError as e:
            # print(f"\t\tWARNING: No .{format} file found for project {project_key} in output path for")
            pass

    lines = []
    from_ts = None if last_analysis_ts is None else last_analysis_ts.strftime(format = '%Y-%m-%d')
    analyses = query_server('analyses',1, project_key = project_key, from_ts = from_ts)
    for analysis in analyses:
        analysis_key = None if 'key' not in analysis else analysis['key']

        date = None if 'date' not in analysis else process_datetime(analysis['date'])
        if date is not None and last_analysis_ts is not None:
            if date <= last_analysis_ts:
                continue

        project_version = None if 'projectVersion' not in analysis else analysis['projectVersion']
        revision = None if 'revision' not in analysis else analysis['revision']

        line = (project_key, analysis_key, date, project_version, revision)
        lines.append(line)

    print(f"\t\t {project_key} - {len(lines)} new analyses.")
    if lines != []:
        df = pd.DataFrame(data = lines, columns= SONAR_ANALYSES_DTYPE.keys())
```

Listing 3.5. *Algorithm for incremental load of Sonarqube data using analyses*

Identically with the Jenkins data, the challenge with the script is its ability to load only new data instead of over querying the server. As established earlier, measures and issues both rely on analyses. If there is no new analysis, there is surely no fresh data on measures and issues. The first step is to determine whether there are any new analyses for a project from the server. The solution is somewhat the same as in Jenkins source. There are *_staging* files for new, unprocessed data and others without *_staging* ending are processed ones. The difference lies at the fact that there is no build number for reference now. Instead, the date of analyses is utilized to identify new analyses. Fortunately, the endpoints take an argument *from* to only return those analyses taken from the passed argument and after. However, there is a possibility of duplication upon merging the new and the existing CSV files. This is due to the fact that the response from the server will also include the analyses on the *from* date, which are already recorded in the existing file and will be re-recorded in the new file. This is a task for the merger script at the end of the whole process to eliminate any duplicates due to overlapping.

Given there are new analyses from the server, and the extraction phase for those new instances is finished, measures endpoint provides a similar *from* argument to get measures from a specific timestamp that corresponds to the new analyses. But the situation with the issues is different. As the endpoint does not provide the *from* argument, what we have to do is to iterate through all the issues again once there are any new analyses and ingest those issues, that are updated in these analyses only.

3.2 Merger of Data Files

In response to the challenge of incremental load, after extraction from the data sources, every job, in the case of Jenkins, or project, in the case of Sonarcloud, will have two CSV files under its name, one with *_staging* ending and the other without it. The idea is to make a clear distinction between unprocessed and processed data. The unprocessed data will continue to undergo the subsequent phases after extraction. At the end of the pipeline, they will become processed, thus, require some methods to merge themselves into the processed data. This merge script achieves exactly that purpose.

This script takes two arguments which are the paths to the Jenkins and Sonarcloud data files. It operates on the build files, test files from Jenkins and the files from Sonarcloud. The inner working is quite simple:

- Iterate through the folder for *_staging* files
- With every result, find its corresponding file without that ending
- If there is not, rename the file by removing the *_staging* ending
- If there is, read both into Pandas DataFrames and union them, then drop duplicates.
- Write the result into a file without _staging ending

```
1   def merge(file_directory, DTYPE):
2
3       if not file_directory.exists():
4           return
5
6       for file in file_directory.glob("*_staging.csv"):
7           archive_file = Path(str(file).replace("_staging", ""))
8           if archive_file.exists():
9
10              old_df = pd.read_csv(archive_file.resolve(), dtype=DTYPE, header=0)
11              new_df = pd.read_csv(file.resolve(), dtype = DTYPE, header = 0)
12
13              df = pd.concat([new_df, old_df], ignore_index = True)
14              df.drop_duplicates(inplace=True)
15
16              df.to_csv(path_or_buf= archive_file, index=False, header=True)
17
18              file.unlink()
19          else:
20              file.rename(archive_file)
```

Listing 3.6. *Program to merge processed and unprocessed data files*

This simple merge script resolves the problem of having to over query on a daily basis which burdens the platform a great deal.

3.3 Backend Database and Visualization Containers

3.3.1 Containers

One essential pillar for the whole platform is the backend database. The start and end of the pipeline involve working directly with CSV files stored on the disk. If the goal is to visualize and analyze the data, it needs to reside in a relational database. Furthermore,

the tools we use like Apache Superset, Apache Airflow all require a backend database to store their metadata about the processes and status.

In this application, a container of PostgreSQL is deployed to be the backend database. The original image of PostgreSQL can be found from the docker hub. But we need to configure the image according to our usage and purpose. That is done in a docker file.

```
1  FROM postgres:10
2
3  ENV POSTGRES_USER hung
4  ENV POSTGRES_PASSWORD hung
5  ENV POSTGRES_DB hung
6
7  COPY init-user-db.sh /docker-entrypoint-initdb.d/init-user-db.sh
```

Listing 3.7. *Dockerfile for PostgreSQL*

The Dockerfile adds environment variables about the root user of the database and delivers a copy of the initialization script into the container. It creates the necessary databases, users, adjusts rights and roles for other services like Airflow, and Superset. It also instantiates a database to store the data of the platform called *pra* and login credentials to use later. This script is executed the first time docker-compose brings up the container, that is, the bind volume *./db_home* is not yet created. The compose file performs regular operations like mapping host machine port to the container port, attach a bind volume to the container, identify a network for this container. The network part is extremely important. Since it allows the containers within the same network to freely communicate without restrictions. We need to manually create the *PRA_net* network before bringing up any containers here.

```
$ docker network create PRA_net
```

Listing 3.8. *Bash Command to create PRA_net docker network*

```yaml
version: "3.7"

services:
  db:
    image: postgres
    build:
      context: .
      dockerfile: postgres-dockerfile
    restart: unless-stopped
    ports:
      - "127.0.0.1:5432:5432"
    volumes:
      - ./db_home:/var/lib/postgresql/data
    networks:
      - PRA_net

networks:
  PRA_net:
    external: true
    name: PRA_net
```

Listing 3.9. *Docker-compose file PostgreSQL*

The other container in this application is the container of Apache Superset. The docker-compose file is nearly the same from the official repository [7]. Some tweaks involve removing the default backend PostgreSQL to use our own database as well as adding the container to the same PRA_net network. Thus, it is extremely vital that PRA_net is created and PostgreSQL container is already brought up before the container of Apache Superset is initialized.

3.3.2 Backend Database

The platform employs PostgreSQL as the backend relational database management system. Within the system, there are three databases: *airflow*, *superset* and *pra*. The first two databases are used by the other tools of the platform Apache Airflow and Apache Superset to store metadata, states of operation, authorizing credentials... We should not meddle with these databases and should leave them self-managed.

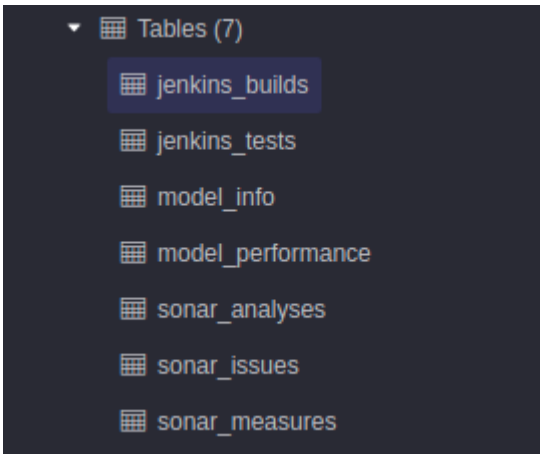

Figure 3.2. *Tables of pra database*

The third database is *pra*, short for Public Repository Analysis, which stores the main data of the platform. There are 7 tables within it. The first two tables, *jenkins_build*s and *jenkins_tests* are from Jenkins data source. Similarly, the three sonar facets analyses, measures and issues contribute three tables *sonar_analyses, sonar_issues,* and *sonar_measures,* which also share the structure with their respective CSV files. *model_info* table stores measures taken on training data, and the top 10 important features as well as the importance values of all the models. This table is only updated once during the first run to train the model. The second model related table is the *model_performance* model. This table records the measures of the models on unseen data. This table is updated daily on the latest extracted data.

3.4 Data Processing

The most central and important phase in the whole platform is the central data processing. The input to this phase is the CSV files from the extraction stage as well as the data from the backend database. There involve two important tasks. The first one is to ingest the new data from the CSV files into the backend database for persistence while making sure that the fields in the tabular data are in an appropriate form. This task is straightforward and would not require such a computation-heavy distributed system like Spark. The second task is exactly where Spark really shines. Spark is used to prepare data, train, test and validate a range of Machine Learning models. There are three main categories of Machine Learning models depending on what attributes are used, however, the final outcome is to try to predict the build result.

3.4.1 General Processing

This data processing is a program written in Python to be submitted to Spark through command *spark-submit* in an environment running Spark. This processing script has three operation modes: *first*, *incremental* and *update_models*. Which mode to execute the script depends on the situation of the data in the backend database and the availability of Machine Learning models. Specifically, Machine Learning models after training are saved to files so that they can be loaded and reused on unseen data in the future. It would be extremely time-consuming to train the whole models every time there is new data.

```
# Check for resources that enable incremental run
if run_mode == "incremental":
    for i in ['1','2','3']:
        for suffix in ["", "_top_10"]:
            for obj in
[f"pipeline_{i}",f"LogisticRegressionModel_{i}{suffix}",f"DecisionTreeModel_{i}{s
uffix}",f"RandomForestModel_{i}{suffix}",
                    f"ChiSquareSelectorModel_{i}", "label_indexer_3"]:

                obj_path = Path(spark_artefacts_dir).joinpath(obj)
                if not obj_path.exists():
                    print(f"{obj} does not exist in spark_artefacts. Rerun with
run_mode = first")
                    run(jenkins_data_directory, sonar_data_directory,
spark_artefacts_dir, "first")

    # Data from db
    try:
        db_jenkins_builds = spark.read.jdbc(CONNECTION_STR, "jenkins_builds",
properties=CONNECTION_PROPERTIES)
        db_sonar_analyses = spark.read.jdbc(CONNECTION_STR, "sonar_analyses",
properties=CONNECTION_PROPERTIES)
        db_sonar_measures = spark.read.jdbc(CONNECTION_STR, "sonar_measures",
properties=CONNECTION_PROPERTIES)
        db_sonar_issues = spark.read.jdbc(CONNECTION_STR, "sonar_issues",
properties=CONNECTION_PROPERTIES)

        for table,name in [(db_jenkins_builds,"jenkins_builds"),
(db_sonar_analyses, "sonar_analyses"), (db_sonar_measures, "sonar_measures"),
(db_sonar_issues, "sonar_issues")]:
            table.persist()
            if table.count() == 0:
                print(f"No data in table [{name}]. Rerun with run_mode = first")
                run(jenkins_data_directory, sonar_data_directory,
spark_artefacts_dir, "first")

    except Exception as e:
        print(f"Exception thrown when reading tables from Postgresql - {str(e)}.
Rerun with run_mode = first")
        run(jenkins_data_directory, sonar_data_directory, spark_artefacts_dir,
"first")
```

Listing 3.10: *Start of execution in incremental mode*

incremental mode is the primary mode of operation which will be executed when the platform goes into permanent operation. This mode starts by checking certain preconditions. The first condition is that there is data in the four tables: *jenkins_builds, sonar_analyses, sonar_measures* and *sonar_issues.* This is checked by loading these tables into separate DataFrames and verifying their count. The second condition is the availability of the Machine Learning models and pipeline models, which transform Spark DataFrames into an appropriate form for Machine Learning, on the file system. These models are fitted and saved in a preceding operation in *first* mode. If any of the conditions

is not met, the script will be re-executed in *first* mode. Next, it loads all *_staging* CSV files, or files that are not processed yet and ingests them into respective tables in the backend database.

```
elif run_mode == "first":
    db_jenkins_builds = None
    db_sonar_analyses = None
    db_sonar_measures = None
    db_sonar_issues = None

new_jenkins_builds = get_data_from_file("jenkins builds",jenkins_data_directory, run_mode)
new_jenkins_builds = new_jenkins_builds.filter("job IS NOT NULL")
new_jenkins_builds.persist()
print("Jenkins builds Count: ", new_jenkins_builds.count())

new_sonar_analyses = get_data_from_file("sonar analyses", sonar_data_directory, run_mode)
new_sonar_analyses = new_sonar_analyses.filter("project IS NOT NULL AND analysis_key IS NOT NULL")
new_sonar_analyses.persist()
print("Sonar analyses Count: ", new_sonar_analyses.count())

new_sonar_measures = get_data_from_file("sonar measures", sonar_data_directory, run_mode)
new_sonar_measures = new_sonar_measures.filter("project IS NOT NULL AND analysis_key IS NOT NULL")
new_sonar_measures = new_sonar_measures.drop(*TO_DROP_SONAR_MEASURES_COLUMNS)
new_sonar_measures.persist()
print("Sonar measures Count: ", new_sonar_measures.count())

new_sonar_issues = get_data_from_file("sonar issues", sonar_data_directory, run_mode)
new_sonar_issues = new_sonar_issues.filter("project IS NOT NULL AND issue_key IS NOT NULL")
new_sonar_issues.persist()
print("Sonar issues Count: ", new_sonar_issues.count())

# UPDATE DB_ DF
db_jenkins_builds = None if db_jenkins_builds is None else db_jenkins_builds.union(new_jenkins_builds)
db_sonar_analyses = None if db_sonar_analyses is None else db_sonar_analyses.union(new_sonar_analyses)
db_sonar_measures = None if db_sonar_measures is None else db_sonar_measures.union(new_sonar_measures)
db_sonar_issues = None if db_sonar_issues is None else db_sonar_issues.union(new_sonar_issues)

if write_data:
    # WRITE TO POSTGRESQL
    write_mode = "overwrite" if run_mode == "first" else "append"
    new_jenkins_builds.write.jdbc(CONNECTION_STR, table="jenkins_builds", mode = write_mode, properties=CONNECTION_PROPERTIES)
    new_sonar_measures.write.jdbc(CONNECTION_STR, table="sonar_measures", mode = write_mode, properties=CONNECTION_PROPERTIES)
    new_sonar_analyses.write.jdbc(CONNECTION_STR, table="sonar_analyses", mode = write_mode, properties=CONNECTION_PROPERTIES)
    new_sonar_issues.write.jdbc(CONNECTION_STR, table="sonar_issues", mode = write_mode, properties=CONNECTION_PROPERTIES)
```

Listing 3.11. *Load, ingest new data to db and also fetch processed data from db*

The mode *first* kicks off processing of Spark by loading all of the CSV files from the extraction stage, including both *_staging* and non *_staging* files, into Spark DataFrames. The DataFrames are directly written to the backend corresponding tables, overwriting any existing data there. The third mode, *update_models*, replicates this procedure, however, it leaves out the stage to write data to the database.

At this stage, the two modes of operation, *incremental* and *first (update_models* mode closely resembles *first* mode), differ mainly in the sources of data they require before preparing data for Machine Learning. In *first* mode, we have *new_jenkins_builds, new_sonar_analyses, new_sonar_issues* and *new_sonar_measures* which store data from all CSV files. While in *incremental* load, these DataFrames represent only *_staging* CSV files or unprocessed files. In addition, there are *db_jenkins_builds, db_sonar_analyses, db_sonar_issues,* and *db_sonar_measures* DataFrames, which are fetched from the backend database and concatenated with the new DataFrames. The need for two separate sources of data will be shed light on in a subsequent section.

```
1  # APPLY MACHINE LEARNING
2  apply_ml1(new_jenkins_builds, db_jenkins_builds, new_sonar_measures,
   db_sonar_measures, new_sonar_analyses, db_sonar_analyses, spark_artefacts_dir,
   run_mode)
3  apply_ml2(new_jenkins_builds, db_jenkins_builds, new_sonar_issues,
   db_sonar_issues,  new_sonar_analyses, db_sonar_analyses, spark_artefacts_dir,
   run_mode)
4  apply_ml3(new_jenkins_builds, db_jenkins_builds, new_sonar_issues,
   db_sonar_issues,  new_sonar_analyses, db_sonar_analyses, spark_artefacts_dir,
   run_mode)
```

Listing 3.12. *Start Machine Learning processes*

With these DataFrames and the corresponding mode of operation, it is ready to apply Machine Learning on the data. The next section introduces common steps in preparing the data for the three Machine Learning model categories. It is followed by an in-depth consideration of each of the three pipelines. Finally, the train, test and predict the data after the pipelines' procedure is described.

3.4.2 Common Data Preparation for Machine Learning

The first stage of all three types of Machine Learning models is to change the field *result* from Jenkins builds DataFrames to binary format, SUCCESS and FAIL. Any entry that is not SUCCESS is considered a FAIL.

```
1  modify_result = udf(lambda x: "SUCCESS" if x == "SUCCESS" else "FAIL",
   StringType())
2  spark.udf.register("modify_result" , modify_result)
3
4  if new_jenkins_builds is not None:
5      new_jenkins_builds = new_jenkins_builds.withColumn("result",
   modify_result("result"))
6
7  if db_jenkins_builds is not None:
8      db_jenkins_builds = db_jenkins_builds.withColumn("result",
   modify_result("result"))
```

Listing 3.13. *Change the build result column to binary*

This is done by defining a UDF, user-defined function, and calling it upon the *result* column of the DataFrame. The procedure is repeated twice on both of Jenkins builds DataFrames.

The first category of Machine Learning models uses measures from Sonarqube as well as build information to predict Jenkins build results. However, the involved data resides in separate DataFrames. Therefore, there has to be a mechanism to determine the Sonarqube measures corresponding to a build from Jenkins. To achieve this, we need the relationship between the measures and instances of analysis from Sonarqube where each analysis produces a set of measures. The measures DataFrame has a field named *analysis_key*, which indicates the analysis in which the measures are generated. We can use this field to join with the same *analysis_key* from the analyses DataFrame. Thus, for each set of measures, we also have information about the corresponding analysis including the *revision* string of the project at the time of analysis. The revision string is a common field with the Jenkins builds DataFrame, where its name is *revision_number*. All in all, at a revision of a project we can have both the build's information and the measures from Sonarqube. The below listing shows the exact procedure of joining the three DataFrames.

```
def prepare_data_ml1(jenkins_builds, sonar_measures, sonar_analyses):

    ml_sonar_df = sonar_measures.join(sonar_analyses, sonar_measures.analysis_key
== sonar_analyses.analysis_key,
    how = 'inner').select(*(['revision'] + SONAR_MEASURES_NUMERICAL_COLUMNS +
SONAR_MEASURES_CATEGORICAL_COLUMNS))
    df = jenkins_builds.join(ml_sonar_df, jenkins_builds.revision_number ==
ml_sonar_df.revision, how = 'inner')

    # Change data type from Int to Float to fit into estimators
    for column_name in ML1_NUMERICAL_COLUMNS:
        if column_name in JENKINS_BUILD_DTYPE:
            if JENKINS_BUILD_DTYPE[column_name] == 'Int64':
                df = df.withColumn(column_name,
df[column_name].astype(DoubleType()))
        elif column_name in SONAR_MEASURES_DTYPE:
            if SONAR_MEASURES_DTYPE[column_name] == 'Int64':
                df = df.withColumn(column_name,
df[column_name].astype(DoubleType()))
    return df
```

Listing 3.14. *Function to prepare data for Machine Learning model type 1*

The arguments to this function are the Jenkins builds, sonar measures and sonar analyses DataFrames. We have two DataFrames for each of these sources, *new_SOURCE* and *db_SOURCE.* For example, for Jenkins builds as a source, we have *new_jenkins_builds* as well as *db_jenkins_builds*. Therefore, we need to decide which one to pass into the function.

```
# PREPARE DATA
if run_mode == "first":
    df = prepare_data_ml1(new_jenkins_builds, new_sonar_measures,
new_sonar_analyses)

elif run_mode == "incremental":
    # New jenkins ~ db sonar
    df1 = prepare_data_ml1(new_jenkins_builds, db_sonar_measures,
db_sonar_analyses)
    # New sonar ~ db jenkins
    df2 = prepare_data_ml1(db_jenkins_builds, new_sonar_measures,
db_sonar_analyses)

    df = df1.union(df2).drop_duplicates()

df.persist()
print(f"DF for ML1 Count: {str(df.count())}")
```

Listing 3.15. *Merging the old and new data for preparation*

In *first* operation mode, all the *db_SOURCE* DataFrames are None, thus, we simply pass in the *new_SOURCE* DataFrames to prepare data. On the other hand, the *incremental* mode tries to seek for the new builds from Jenkins and new measures from Sonarqube. If we simply join the *new_jenkins_builds, new_sonar_analyses* and together with *new_sonar_measures*, there is a chance that measures of a new build is already recorded to the backend database in a preceding execution and vice versa. To resolve this problem, we need to join the new builds with all the sonar measures available and join the new measures with all the builds available. That is the reason behind the *db_SOURCE* DataFrames which stores all entries of a specific table, both old and new entries. One point to note is that the sonar analyses are the middle key to join measures and builds, therefore, the sonar analyses table can always be in its fullest form without the risk of including old measures. Finally, the two prepared DataFrames are concatenated and removed any duplicates.

The situation is similar with the other two categories of Machine Learning models. We need to join the builds from Jenkins with the issues data from Sonarqube and the middle key is also the Sonar analyses. Furthermore, the problem and solution to only fetch new data are the same. There is a small deviation from this procedure in category 3, where what we are actually interested in is the rules rather than the issues themselves. This will be fully explained in the respective subsection.

These prepared data are ready for a pipeline and subsequent Machine Learning algorithms. At this stage, the difference between the two modes of operation, *first* and *incremental*, is straightforward. The *first* mode will fit the prepared data into a pre-built pipeline to generate a pipeline model, save it to the file system for later use. Similarly, the prepared data, after the transformation step of the pipeline model, becomes machine learning ready. The Machine Learning models are generated by fitting on part of the data, train set, and tested against another distinguished part, test set. The models are then saved to file system for later application. While in *incremental* mode, both the pipeline model and the Machine Learning models are loaded from files rather than fitted from scratch. They are simply used to transform the latest data into an adequate form and tested against loaded Machine Learning models.

3.4.3 Machine Learning Model Type 1 Preparation

Each of the Machine Learning model categories will have a distinctive pipeline model. Pipeline is a mechanism deployed by Spark to make sure that a DataFrame will undergo the same series of transformations to transform into another. In our case, the output DataFrames will be ready for training or testing again Machine Learning models. As mentioned earlier, this type 1 has attributes about the builds like duration of the builds, expected duration, number of tests passed, failed, skipped, and total test duration. In addition, the input also has attributes about Sonarqube measures of the build. From the list of metrics, we only take those numerical metrics and one categorical metric called alert_status and all numerical attributes from the builds. Last but not least, we also take the build result, which is the label to predict. The input DataFrames has to satisfy the requirement of containing at least all of these fields so that it can be fed into the pipeline model. For easy manipulation, all of these fields are grouped into a constant named ML1_NUMERICAL_COLUMNS and ML1_CATEGORICAL_COLUMNS. Next, we define a pipeline by establishing a set of stages that the input DataFrames will undergo sequentially.

```
1  def get_ml1_pipeline():
2      stages = []
3
4      imputer = Imputer(inputCols=ML1_NUMERICAL_COLUMNS , outputCols=ML1_NUMERI-
   CAL_COLUMNS )
5      stages.append(imputer)
6
7      ohe_input_cols = []
8      ohe_output_cols = []
9      for categorical_column in ML1_CATEGORICAL_COLUMNS:
10         str_indexer = StringIndexer(inputCol=categorical_column, outputCol=cate-
   gorical_column + "_index", handleInvalid='keep')
11         ohe_input_cols.append(str_indexer.getOutputCol())
12         ohe_output_cols.append(categorical_column + "_class_vec")
13         stages.append(str_indexer)
14
15     encoder = OneHotEncoderEstimator(inputCols=ohe_input_cols, out-
   putCols=ohe_output_cols, handleInvalid="error", dropLast=False)
16     stages.append(encoder)
17
18     numerical_vector_assembler = VectorAssembler(inputCols=ML1_NUMERICAL_COLUMNS
   , outputCol="numerial_cols_vec", handleInvalid="keep")
19     scaler = MinMaxScaler(inputCol="numerial_cols_vec", outputCol= "scaled_numer-
   ical_cols")
20     stages.append(numerical_vector_assembler)
21     stages.append(scaler)
22
23     label_str_indexer = StringIndexer(inputCol="result", outputCol="label",
   handleInvalid="keep")
24     stages.append(label_str_indexer)
25
26     assembler_input = encoder.getOutputCols() + [scaler.getOutputCol()]
27     assembler = VectorAssembler(inputCols= assembler_input, outputCol="features",
   handleInvalid="skip")
28     stages.append(assembler)
29
30     pipeline = Pipeline(stages = stages)
31     return pipeline
```

Listing 3.16. *Function to build the pipeline for Machine Learning model type 1*

The first stage in the pipeline is the Imputer. Which fills in invalid values of a column with the mean of all values present in the column. This is performed directly on all of the numerical columns of the DataFrame. The second stage is StringIndexer, which indexes the string value based on the frequency of appearance in the column. The string with most appearance will have index 0.0, the second is indexed 1.0, ... This stage operates on categorical columns only and produces another column. The number of StringIndexer steps is equal to the number of categorical columns, which is 1. The next stage operates on the newly generated index column to produce a vector of 0.0 and 1.0, where only one value 1.0 is placed at the index value, called the OneHotEncoder. For example, if the

output of the StringIndexer is 3.0 in a column of 5 distinct string values, it will have representation as a vector [0.0, 0.0, 0.0, 1.0, 0.0]. Next, we concatenate all numerical columns into one vector via a VectorAssembler. This vector undergoes a MinMaxScaler step to scale all columns to have a magnitude of 0 to 1 since they are all positive values. This helps force all the columns of the original DataFrames to have the same weight in predicting the label. These scaled vectors are concatenated with the vector from OneHotEncoder to produce vectors called *features*. The *result* column also undergoes StringIndexer to produce a column named *label*. These two columns will be fed to Machine Learning models.

3.4.4 Machine Learning Model Type 2 Preparation

Type 2 Models use the count of sonar issues to predict the build outcomes. As mentioned earlier, we use the Jenkins builds, sonar issues and sonar analyses for this category.

```
1  def prepare_data_ml2(jenkins_builds, sonar_issues, sonar_analyses):
2
3      with open('./sonar_issues_count.sql', 'r') as f:
4          query1 = f.read()
5      with open('./sonar_issues_count_with_current.sql', 'r') as f:
6          query2 = f.read()
7
8      sonar_issues.createOrReplaceTempView('sonar_issues')
9      sonar_issues_count = spark.sql(query1)
10     sonar_issues_count.createOrReplaceTempView('sonar_issues_count')
11     sonar_issues_count_with_current = spark.sql(query2)
12     sonar_df = sonar_issues_count_with_current.join(sonar_analyses,
   sonar_issues_count_with_current.analysis_key == sonar_analyses.analysis_key,
13         how = "inner")
14     df = sonar_df.join(jenkins_builds, sonar_df.revision ==
   jenkins_builds.revision_number, how = "inner").select(*(['result'] +
   ML2_NUMERICAL_COLUMNS))
15
16     # Change data types to fit in estimators
17     for numerical_column in ML2_NUMERICAL_COLUMNS:
18         df = df.withColumn(numerical_column,
   df[numerical_column].astype(DoubleType()))
19
20     return df
```

Listing 3.17. *Function to prepare data for Machine Learning model type 2*

However, there need to be some aggregations over the sonar issues DataFrame to put it into a usable format. This is accomplished by first creating a temporary view inside SparkSQL for interactive query of the dataset. Then we load a SQL query to be run against this temporary view to create a new DataFrame. The first query is the *sonar_issues_count.sql*. It scans over the whole *sonar_issues* view twice. The first iteration counts the number of introduced issues divided into types and severity levels by grouping the view by the *creation_analysis_key* on issues that have status OPEN, REOPENED, CONFIRMED and TO_REVIEW. The second iteration counts the number of removed issues divided into types and severity levels by grouping the view by the *update_analysis_key* on issues that have status RESOLVED, CLOSED and REVIEWED. These two sub-tables are joined on their *analysis_key,* therefore, for each *analysis_key* there will be a count of introduced and removed issues sub-divided by types and severity levels. There is no introduced or removed issues of a type, severity, the count is simply 0. The output DataFrame of this query is also registered as another temporary view in SparkSQL. Another query is loaded to determine the current count of issues.

The second query, *sonar_issues_count_with_current.sql* stored in the main working directory, partitions the *sonar_issues_count* view by project, and compute a running sum in order of date of the *analysis_key*. The number is the difference between the introduced and removed issues plus the sum of issues in the last analysis, thus, it is the current count of issues of the analysis. With this DataFrame, we only need to join with sonar analyses and Jenkins builds DataFrames as above. The output of this is ready for pipeline model 2.

```
1  def get_ml2_pipeline():
2      stages = []
3
4      numerical_vector_assembler = VectorAssembler(inputCols=ML2_NUMERICAL_COLUMNS
   , outputCol="numerial_cols_vec", handleInvalid="keep")
5      scaler = MinMaxScaler(inputCol="numerial_cols_vec", outputCol= "features")
6      stages.append(numerical_vector_assembler)
7      stages.append(scaler)
8
9      label_str_indexer = StringIndexer(inputCol="result", outputCol="label",
   handleInvalid="keep")
10     stages.append(label_str_indexer)
11
12     pipeline = Pipeline(stages = stages)
13     return pipeline
```

Listing 3.18. *Function to build pipeline for Machine Learning model type 2*

This pipeline is simpler than pipeline of model type 1 due to the fact that there is no categorical column and the invalid count is already imputed as 0 in the SQL queries. We only need to assemble the fields in a vector, scale them with MinMaxScaler, and indexing the label string. The pipeline model can now convert input DataFrames into DataFrames that are ready for Machine Learning.

3.4.5 Machine Learning Model Type 3 Preparation

This category deals directly with the rules and their count of removed and introduced ones in an analysis, but still to predict the build results. There are up to 487 different rules in the sonar issues DataFrames, either removed or introduced, thus, totaling 974 columns of features

```
1  pipeline_path = Path(spark_artefacts_dir).joinpath("pipeline_3")
2  label_idx_model_path = Path(spark_artefacts_dir).joinpath("label_indexer_3")
3  #PREPARE DATA
4  if run_mode == "first":
5
6      pipeline_model = get_ml3_pipeline().fit(new_sonar_issues)
7      pipeline_model.write().overwrite().save(str(pipeline_path.absolute()))
8
9      label_idx_model = StringIndexer(inputCol="result",
   outputCol="label").fit(new_jenkins_builds)
10
   label_idx_model.write().overwrite().save(str(label_idx_model_path.absolute()))
11
12     ml_df = prepre_data_ml3(new_jenkins_builds, new_sonar_issues,
   new_sonar_analyses, pipeline_model, label_idx_model)
13
14 elif run_mode == "incremental":
15
16     pipeline_model = PipelineModel.load(str(pipeline_path.absolute()))
17     label_idx_model =
   StringIndexerModel.load(str(label_idx_model_path.absolute()))
18
19     ml_df1 = prepre_data_ml3(new_jenkins_builds, db_sonar_issues,
   db_sonar_analyses, pipeline_model, label_idx_model)
20     ml_df2 = prepre_data_ml3(db_jenkins_builds, new_sonar_issues,
   db_sonar_analyses, pipeline_model, label_idx_model)
21
22     ml_df = ml_df1.union(ml_df2)
```

Listing 3.19. *General data preparation step for Machine Learning model type 3*

The structure of the preparation step in this category is a bit different. First, we examine the pipeline model.

```
1  def get_ml3_pipeline():
2
3      stages = []
4      str_idx = StringIndexer(inputCol="rule", outputCol="rule_idx")
5      ohe = OneHotEncoderEstimator(inputCols=["rule_idx"], outputCols=["rule_vec"],
   dropLast=False)
6      stages = [str_idx, ohe]
7      return Pipeline(stages= stages)
```

Listing 3.20. *Function to build pipeline for Machine Learning model type 3*

This pipeline operates on the *rule* column of the DataFrame like any usual categorical column. First, a StringIndexer indexes the entries based on the appearance frequency then comes a OneHotEncoder to turn that index into a vector of 0 and 1. This pipeline model and the label indexer model are passed into a function to prepare data from Machine Learning model type 3. The reason why we cannot put label indexer into the pipeline is due to the fact that they are used at different stages when preparing data.

```
def prepre_data_ml3(jenkins_builds, sonar_issues,sonar_analyses ,pipeline_model, label_idx_model):

    removed_rules_df = sonar_issues.filter("status IN ('RESOLVED', 'CLOSED', 'REVIEWED')").select("current_analysis_key","rule")
    df1 = pipeline_model.transform(removed_rules_df)
    rdd1 = df1.rdd.map(lambda x : (x[0],x[3])).reduceByKey(lambda v1,v2: sum_sparse_vectors(v1,v2)).map(lambda x: Row(current_analysis_key = x[0], removed_rule_vec = x[1]))
    removed_issues_rule_vec_df = spark.createDataFrame(rdd1)

    introduced_rules_df = sonar_issues.filter("status IN ('OPEN', 'REOPENED', 'CONFIRMED', 'TO_REVIEW')").select("creation_analysis_key","rule")
    df2 = pipeline_model.transform(introduced_rules_df)
    rdd2 = df2.rdd.map(lambda x : (x[0],x[3])).reduceByKey(lambda v1,v2: sum_sparse_vectors(v1,v2)) \
                                                .map(lambda x: Row(creation_analysis_key = x[0], introduced_rule_vec = x[1]))
    introduced_issues_rule_vec_df = spark.createDataFrame(rdd2)

    joined_sonar_rules_df = removed_issues_rule_vec_df.join(introduced_issues_rule_vec_df,
        removed_issues_rule_vec_df.current_analysis_key == introduced_issues_rule_vec_df.creation_analysis_key, how = "outer")

    joined_sonar_rules_df.createOrReplaceTempView("sonar_rules")
    joined_sonar_rules_df = spark.sql("""SELECT
        coalesce(current_analysis_key, creation_analysis_key) AS analysis_key,
        introduced_rule_vec,
        removed_rule_vec
        FROM sonar_rules
        """)

    num_rules = len(pipeline_model.stages[0].labels)
    imputed_sonar_rules_rdd = joined_sonar_rules_df.rdd.map(lambda row: Row(
        analysis_key = row[0],
        introduced_rule_vec = SparseVector(num_rules,{}) if row[1] is None else row[1],
        removed_rule_vec = SparseVector(num_rules,{}) if row[2] is None else row[2]))

    imputed_sonar_rules_df = spark.createDataFrame(imputed_sonar_rules_rdd)

    v_assembler = VectorAssembler(inputCols=["removed_rule_vec", "introduced_rule_vec"], outputCol="features")
    sonar_issues_df = v_assembler.transform(imputed_sonar_rules_df).select("analysis_key","features")

    sonar_df = sonar_issues_df.join(sonar_analyses, sonar_issues_df.analysis_key == sonar_analyses.analysis_key, how = "inner")
    df = sonar_df.join(jenkins_builds, sonar_df.revision == jenkins_builds.revision_number, how = "inner").select("result", "features")
    ml_df = label_idx_model.transform(df).select("label", "features")
    return ml_df
```

Listing 3.21. *Function to prepare data for Machine Learning model type 3*

Similar to the preparation of model type 2, the goal is to get the count of removed and introduced but subdivided by rules. To get the count of removed violations of rules, from the sonar issues DataFrames, we first filter out entries whose status is not either RESOLVED, CLOSED, REVIEWED, taking only the rule and current_analysis_key column. The result is run through the pipeline model to produce the vector from the rule column. This is then aggregated on current_analysis_key where the vectors of rule's violations are summed up. This operation cannot be done when data is in the form of DataFrame. Therefore, it needs to be in RDD form for easier treatment since RDD is the low-level API to operate with data. After the aggregation, the output RDD is transformed back into a DataFrame. The procedure for introduced violations of rules is similar except for that the status of entries is expected to be either OPEN, REOPENED, CONFIRMED, or TO_REVIEW and creation_analysis_key gets chosen instead of current_analysis_key. These two keys are then combined to retrieve the introduced, removed rule violations for each of the analysis_key. If an analysis_key only has a vector of introduced rule violations or removed rule violations. The missing vector is automatically imputed with a vector full of 0. The last step is to assemble these two vectors into a features vector. The DataFrame now contains features and analysis_key, the latter column is then used to join with sonar analyses and subsequently Jenkins builds. Consequently, we have a DataFrame of label from running label indexer against the result column and features column, ready for Machine Learning.

3.4.6 Feature Selection with Chi Square Selector

An important phase in Machine Learning is to do feature selection. There can be around one thousand features for a Machine Learning algorithm. Reducing the number of features can have benefits on the performance of models as well as gives users a sense of which features plays a more vital role in the prediction of labels.

```
def feature_selector_process(ml_df, spark_artefacts_dir, run_mode, i):

    # APPLY CHI-SQUARE SELECTOR
    name = f"ChiSquareSelectorModel_{i}"
    selector_model_path = Path(spark_artefacts_dir).joinpath(name)

    feature_cols = []
    if i == 1:
        feature_cols = ML1_COLUMNS
    elif i == 2:
        feature_cols = ML2_NUMERICAL_COLUMNS
    elif i == 3:
        feature_cols = ML3_COLUMNS

    if run_mode == 'first':

        # ChiSq Test to obtain ChiSquare values (higher -> more dependence between feature and lable -> better)
        r = ChiSquareTest.test(ml_df, "features", "label")
        pValues = r.select("pvalues").collect()[0][0].tolist()
        stats = r.select("statistics").collect()[0][0].tolist()
        dof = r.select("degreesOfFreedom").collect()[0][0]

        # ChiSq Selector
        selector =ChiSqSelector(numTopFeatures= 10, featuresCol="features", outputCol="selected_features", labelCol="label")
        selector_model = selector.fit(ml_df)

selector_model.write().overwrite().save(str(selector_model_path.absolute()))

        top_10_feaures_importance = []
        top_10_features = []
        for j in selector_model.selectedFeatures:
            top_10_feaures_importance.append(feature_cols[j])
            top_10_features.append(feature_cols[j])
            top_10_feaures_importance.append(stats[j])

        model_info = [name, ml_df.count(), None, None, None, None, None, None, None] + top_10_feaures_importance
        model_info_df = spark.createDataFrame(data = [model_info], schema = MODEL_INFO_SCHEMA)
        model_info_df.write.jdbc(CONNECTION_STR, 'model_info', mode='append', properties=CONNECTION_PROPERTIES)

    elif run_mode == 'incremental':
        selector_model = ChiSqSelectorModel.load(str(selector_model_path.absolute()))
        top_10_features = []
        for j in selector_model.selectedFeatures:
            top_10_features.append(feature_cols[j])

    ml_df_10 = selector_model.transform(ml_df)
    ml_df_10 = ml_df_10.drop("features")

    #Solve a problem with ChiSqSelector and Tree-based algorithm
```

Listing 3.22. *Feature selection process with Chi Square Test*

This function aims to reduce all of the models' number of input features to 10, which means it return only the 10 most important features. The importance test is carried out by the ChiSquareSelector of Spark. The Selector transforms the input DataFrames into a DataFrame with a column of chose features and produces a list of indices for the chosen features. Unfortunately, the selector does not provide an exact score of importance for the chosen features. This has to be done separately by calling a Chi Square statistics test on the input DataFrames. The test produces a vector of Chi Square statistics corresponding to the columns of features. From the list of indices, we can get the name and importance value of the chosen features. This is recorded into the *model_info* table in the backend database as *ChiSquareSelector_[n]* where n is the model type number. The function returns the DataFrame of chosen features.

3.4.7 Machine Learning Model Training and Evaluation

For each of the Machine Learning model types, there are two datasets, one with full features and the other with only the top 10 most important features. Each of these datasets will be used against three different Machine Learning algorithms: Logistic Regression, Decision Tree and Random Forest. All in all, there are 6 different models. The naming convention is as follows: [ALGORITHM]Model_[TYPE](_top_10). For instance, the Decision Tree model of type 2 trained against the dataset of top 10 features will have the name DecisionTreeModel_2_top_10, while the Random Forest model of type 3 trained against the full-feature dataset is named RandomForestModel_3. The three different types of models all share the same training and evaluation step.

```
train,test = ml_df.randomSplit([0.7, 0.3])
train.persist()
test.persist()

replication_factor = 10
negative_label_train = train.filter("label = 1.0")
negative_label_train.persist()
negative_label_train.collect()

for i in range(replication_factor):
    train = train.union(negative_label_train)
    train.persist()

train.persist()
train_count = train.count()
print("Training Dataset Count: " + str(train_count))
test_count = test.count()
print("Test Dataset Count: " + str(test_count))
```

Listing 3.23. *Augmenting train set with replicated data*

The input dataset is divided into two parts the train set and the test set with ratio 7:3. Before starting training using the train set, it is augmented with duplicated data. This is due to the fact that the raw data from Jenkins builds is heavily skewed. The number of "negative" labels, build results that are not SUCCESS, is much less than that of "positive" labels. Therefore, the models tend to lean towards predicting 0.0, positive label. To combat this, the train set is augmented with duplicated entries that have label 1.0, up to several factors of all the entries are replicated.

```
lr = LogisticRegression(featuresCol='features', labelCol='label', maxIter=10)
dt = DecisionTreeClassifier(featuresCol='features', labelCol='label', maxDepth=5)
rf = RandomForestClassifier(featuresCol = 'features', labelCol = 'label',
numTrees=100)

model_performance_lines = []
model_info_lines = []
for algo, model_name in [(lr, lr_model_name), (dt, dt_model_name),
(rf,rf_model_name)]:
    print(model_name)
    model = algo.fit(train)
    model_path = Path(spark_artefacts_dir).joinpath(model_name)
    model.write().overwrite().save(str(model_path.absolute()))

    # Tree-based algorithm's Feature Importances:
    if algo in [dt, rf]:

        f_importances = model.featureImportances
        indices = f_importances.indices.tolist()
        values = f_importances.values.tolist()

        value_index_lst = list(zip(values, indices))
        value_index_lst.sort(key = lambda x: x[0], reverse= True)

        importance_sorted_features = []
        for value, index in value_index_lst:
            importance_sorted_features.append(ML_COLUMNS[index])
            importance_sorted_features.append(value)

        length = len(importance_sorted_features)

        if length > 20:
            importance_sorted_features = importance_sorted_features[:20]
        elif length < 20:
            importance_sorted_features = importance_sorted_features + (20 -
length)*[None]

    else:
        importance_sorted_features = 20 * [None]
```

Listing 3.24. *Fitting Machine Learning algorithms and obtain important features*

The same three algorithms and their inner arguments are used across different types of models. Within each category of Machine Learning model, each algorithm will be trained against the train set to produce a model by fitting the train set into the algorithm. This model is saved to be used later in *incremental* load. For the tree-based algorithms like Decision Tree and Random Forest, the model object comes with an attribute called *featureImportances* which contains the indices of the most important feature in predicting the label and the importance values. We then extract the names, values of importance of the features and sort them by those values. The top 10 features of each model will be recorded into the model's information in *model_info* table along with their importance

values. The idea is the same as the Chi Square Selector and test. However, the Chi Square Selector selects the top features to train model with, the top features at this stage are the features the models deem most important after training.

```
predictions = model.transform(test)
predictions.persist()
predictions.show(5)

train_predictions = model.transform(train)
train_predictions.persist()
train_predictions.show(5)

measures = []
train_measures = []

ma_eval = MulticlassClassificationEvaluator()
for metricName in ["f1","weightedPrecision","weightedRecall","accuracy"]:
    measure = ma_eval.evaluate(predictions, {ma_eval.metricName: metricName})
    measures.append(measure)
    print(f"\t{metricName}: {measure}")

    train_measure = ma_eval.evaluate(train_predictions, {ma_eval.metricName: metricName})
    train_measures.append(train_measure)
    print(f"\tTrain-{metricName}: {train_measure}")

bin_eval = BinaryClassificationEvaluator()
for metricName in ["areaUnderROC" , "areaUnderPR"]:
    measure = bin_eval.evaluate(predictions, {bin_eval.metricName: metricName})
    measures.append(measure)
    print(f"\t{metricName}: {measure}")

    train_measure = bin_eval.evaluate(train_predictions, {bin_eval.metricName: metricName})
    train_measures.append(train_measure)
    print(f"\tTrain-{metricName}: {train_measure}")

# Predicted negatives
predicted_negative_rate = predictions.select("label").filter("label = 1.0 AND prediction = 1.0").count() \
    / predictions.select("label").filter("label = 1.0").count()
print(f"\tpredicted_negative_rate: {predicted_negative_rate}")
measures.append(predicted_negative_rate)

train_predicted_negative_rate = train_predictions.select("label").filter("label = 1.0 AND prediction = 1.0").count() \
    / train_predictions.select("label").filter("label = 1.0").count()
print(f"\ttrain_predicted_negative_rate: {train_predicted_negative_rate}")
train_measures.append(train_predicted_negative_rate)
```

Listing 3.25. *Model application and evaluation*

When the Machine Learning model is ready, it can transform a DataFrame of appropriate form into a *predictions* DataFrame. Here, we are interested in the performance of models

on the test set, as well as, its performance on its own train set. The *predictions* DataFrames contain two essential columns, the *prediction* for that row of data and the *label* for the actual label of that row. This DataFrame can be passed directly to different Evaluators to get evaluation metrics of the Machine Learning models. There are two different evaluators here, the MultiClassificationEvaluator and the BinaryClassificationEvaluator. Each evaluator has a set of metrics associated with it. Although the type of classification we are handling in all the models is binary, MultiClassificationEvaluator can be used to evaluate the models under certain metrics: f1, weightedPrecision, weightedRecall and accuracy. While the BinaryClassificationEvaluator offers areaUnderROC and areaUnderPR as metrics. As mentioned earlier, the data is heavily skewed, and due to the fact that we are more interested in the cases of failed builds, the rate at which a model is capable of predicting a FAIL build is essential. This rate is calculated by the predicted true negative entries divided by the total numbers of entries with negative labels (1.0). All of these metrics are utilized on predictions from the train and test set. The measures on train set are recorded to the *model_info* table. While those on the test set are written to *model_performance* table.

```
elif run_mode == "incremental":

    model_performance_lines = []
    for model, name in [(LogisticRegressionModel, lr_model_name),
(DecisionTreeClassificationModel, dt_model_name),
(RandomForestClassificationModel, rf_model_name)]:

        print("\n\n" + name)
        model_path = Path(spark_artefacts_dir).joinpath(name)
        ml_model = model.load(str(model_path.absolute()))

        predictions = ml_model.transform(ml_df)
        predictions.persist()
        predictions.show(5)

        measures = []
        ma_eval = MulticlassClassificationEvaluator()
        for metricName in ["f1","weightedPrecision","weightedRecall","accuracy"]:
            measure = ma_eval.evaluate(predictions, {ma_eval.metricName:
metricName})
            measures.append(measure)
            print(f"\t{metricName}: {measure}")

        bin_eval = BinaryClassificationEvaluator()
        for metricName in ["areaUnderROC" , "areaUnderPR"]:
            measure = bin_eval.evaluate(predictions, {bin_eval.metricName:
metricName})
            measures.append(measure)
            print(f"\t{metricName}: {measure}")

        # Predicted negatives
        predicted_negative_rate = predictions.select("label").filter("label = 1.0
AND prediction = 1.0").count() / predictions.select("label").filter("label =
1.0").count()
        print(f"\tpredicted_negative_rate: {predicted_negative_rate}")
        measures.append(predicted_negative_rate)
```

Listing 3.26. *Model application in incremental mode*

In *incremental* mode, the models are loaded from files to transform the whole set of new data. All evaluation metrics on the data are recorded to *model_performance* table.

3.5 Scheduling Workflow

The continuous and autonomous nature of the platform is handled and ensured by Apache Airflow. Airflow handles a complete process from start to finish as a DAG. A DAG consists of small individual tasks. At this stage, there are the extraction scripts, the Spark program to process the data, and a merger script to merge the newly processed data

files into the old ones. Airflow will assemble these separate pieces into a compact system, trigger its execution every day. To achieve this, have to define a DAG which is a python file with ".py" extension.

```
default_args = {
    'owner': 'hung',
    'depends_on_past': False,
    'start_date': datetime(2020, 4, 8),
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG('platform', default_args = default_args, schedule_interval =
timedelta(days= 1))
```

Listing 3.27. *Setting default arguments for DAG*

We start off by defining default arguments for our DAG. Properties like *owner*, *start_date* (the date at which to start the execution), *retries* (the number of retries attempts when the DAG fails), *retry_delay* (interval to wait until a retry attempt) and *depends_on_past* (whether an instance of execution of the DAG depends on the past instances)are all set as default arguments. Then we define a DAG object specifying the name, default arguments to use and, most importantly, the interval of execution. In our DAG, the interval is 1 day meaning the DAG will be triggered once every day.

```
t1_jenkins = PythonOperator(
    task_id = 'fetch_jenkins_data',
    provide_context=False,
    python_callable= fetch_jenkins,
    op_args=[True, None, f'{repo_dir}/jenkins_data/data', True],
    dag = dag
)

t1_sonar = PythonOperator(
    task_id = 'fetch_sonarqube_data',
    provide_context=False,
    python_callable= fetch_sonar_data,
    op_args=[f'{repo_dir}/sonarcloud_data/data'],
    dag = dag
)

t2 = BashOperator(
    task_id = "spark_processing",
    dag = dag,
    bash_command = f"cd {repo_dir} && spark-submit --driver-class-path
postgresql-42.2.12.jar spark.py"
)

t3 = PythonOperator(
    task_id = "merge_stage_archive",
    provide_context=False,
    python_callable= main,
    op_args=[f"{repo_dir}/jenkins_data/data",
f"{repo_dir}/sonarcloud_data/data"],
    dag = dag
)

t1_jenkins >> t2
t1_sonar >> t2
t2 >> t3
```

Listing 3.28. *Define tasks and their dependencies*

Next, we define our available components as tasks of the DAG. The extract scripts are python scripts, which can be executed in a PythonOperator. This operator takes a function for argument *python_callable*, the arguments to the function can be passed to *op_args* as a list. We have two separate scripts for two separate sources of data, therefore, there are two separate tasks in the DAG file for data extraction. Next, the processing Spark program is not called as a normal Python program although it also has ".py" extension. The program is executed when it is submitted in an environment running Spark. The submission is done via a bash command; therefore, we need a BashOperator from Airflow for this task. The bash command we wish to run is passed to the *bash_command* argument. Here, the working directory is changed to the main directory of the program, where the spark.py file and the driver for Spark to interact with the backend database, postgresql-42.2.12.jar reside and the Spark program is submitted via

spark-submit. The final task is a PythonOperator for the merger script. It is similar to the first two PythonOperators.

All of the tasks defined here are passed with a DAG object we created earlier to the *dag* argument so that Airflow understands what DAG a task belongs to. A name for each task is also given to be displayed on the web UI. The last step is to define the dependency between the tasks. The operator ">>" places the right operand before the left operand in terms of execution. Thus, both tasks, *t1_jenkins* and *t1_sonar*, are executed first. Their successful execution will trigger task *t2,* the Spark program. Finally, the merger program is called as in task *t3*. After defining this DAG file, it needs to be copied to a folder where Airflow picks up the DAGs. By default, it is available at $AIRFLOW_HOME/dags. Now it will be visible on the web API at localhost:8080.

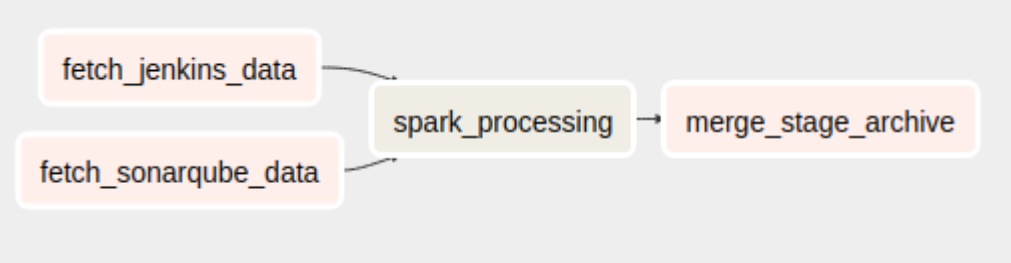

Figure 3.3. *Airflow Graph View of tasks*

This shows the workflow of the tasks defined in our DAG. The arrows show the direction of execution. A task is only executed when all its dependencies are met. If one task fails, the whole DAG fails.

Figure 3.4. *Airflow Tree View of tasks and their statuses*

The picture shows the tree view of the DAG which is built from bottom up. The first tasks are at the bottom, the last task is at the top. The view on the right is the execution result of the DAG, where dark green means success, light green means running and gray means in queue waiting for execution. The vertical orientation follows the tree view with the tasks lining up vertically from bottom up, represented by a square. The horizontal orientation represents of DAG instances of execution throughout time, represented by circles. We can easily track the execution status of the platform.

4. OBSERVATION

At the end of the platform is the visualization component to help make sense of the data. This visualization is powered by Apache Superset. We can access the dashboards through the web browser if Superset is running on the host machine, by default, at localhost:8088. The prepared dashboard is called "PRA - Model" which focuses on the trained and applied Machine Learning models. From the three model types, we can draw certain conclusions about what might affect the build stability and the predictive power of the models.

4.1 Performance of models

In general, the models show a good prediction capability. With certain models shows staggering results on all of the employed evaluation metrics on both the train set and several test sets.

Figure 4.1. *Superset Dashboard about Decision Tree Model 1 Information*

For instance, Decision Tree Model 1 trained with full feature data set gives over 0.8 score in all evaluation metrics. The model also performs well on unseen data, keeping the scores high throughout 3 days of new data.

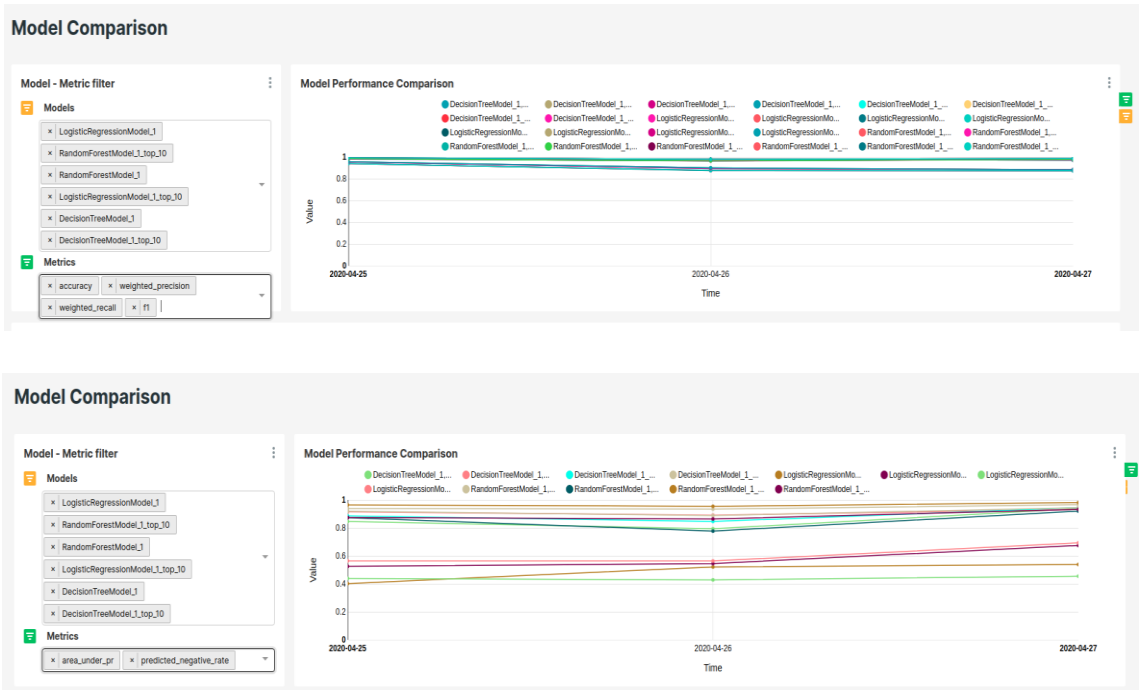

Figure 4.2. *Performance of Models of type 1 in different evaluation metrics*

The type 1 models perform excellently on the f1, accuracy, weighted precision and weighted recall scales, all got over 0.87 on the test sets. However, the situation is different when it comes to area under PR and rate of predicted negatives. In these evaluation metrics, the Logistic Regression models are outperformed by others. All the lines of range 0.4 to 0.6 belong to the Logistic Regression models. It can be concluded that these models do not perform well on skewed data.

The situation with models of type 2 is somewhat similar. The models perform relatively well on unseen data throughout 3 days of testing in terms of f1, weighted recall, weighted precision and accuracy. However, these models are generally not as good as those of types 1 with the scores ranging from 0.65 to 0.99.

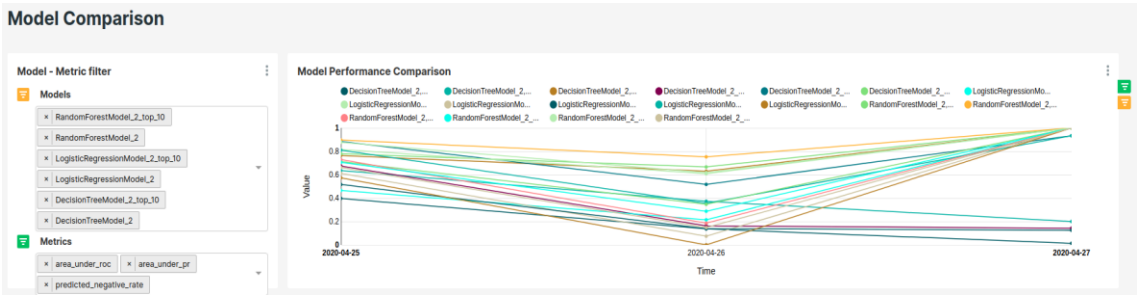

Figure 4.3. *Performance of Models of type 2*

But when it comes to the other three metrics, area under ROC, PR and predicted negative rate, the models vary greatly with extremely low scores as well as absolute 1.0 scores.

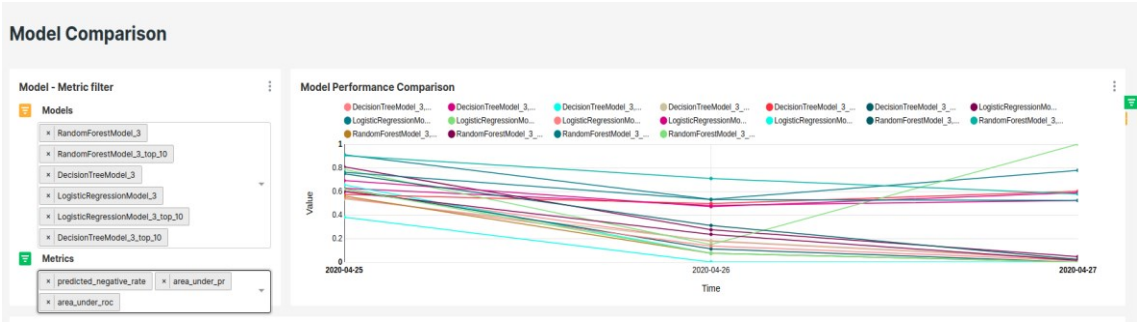

Figure 4.4. *Performance of Models of type 3*

Models of type 3 hold the same trend in the scales of f1, accuracy, weighted precision and recall. But some of their figures in predicting negative entries and area under PR are extremely low. It is obvious that the rules violations are not as good indications of failed builds as the sonar measures.

4.2 Important Features

For each of Chi Square Selector models, there is a subset of features from the input features that is considered important. They as an individual feature has a high dependency on the label. This is verified through the independency test, the higher the Chi Square test statistic, the more dependence there is between the feature and the

label. Similarly, tree-based algorithms also produce a vector of important features and important values.

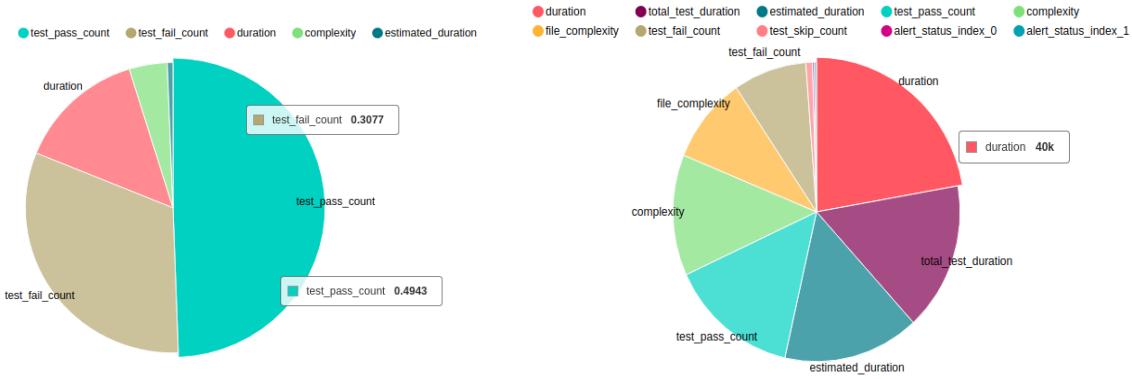

Figure 4.5. *Feature Importance pie charts of Decision Tree model 1 (left) and Chi Square Selector Model 1 (right)*

All the Machine Learning models of type 1 point out the top 4 most important features in their dataset are the test failed and passed count, duration of the build and complexity. This confirms the Chi Square Test values, although the order of importance varies. However, the differences in Chi Square statistics of the top features are not significant.

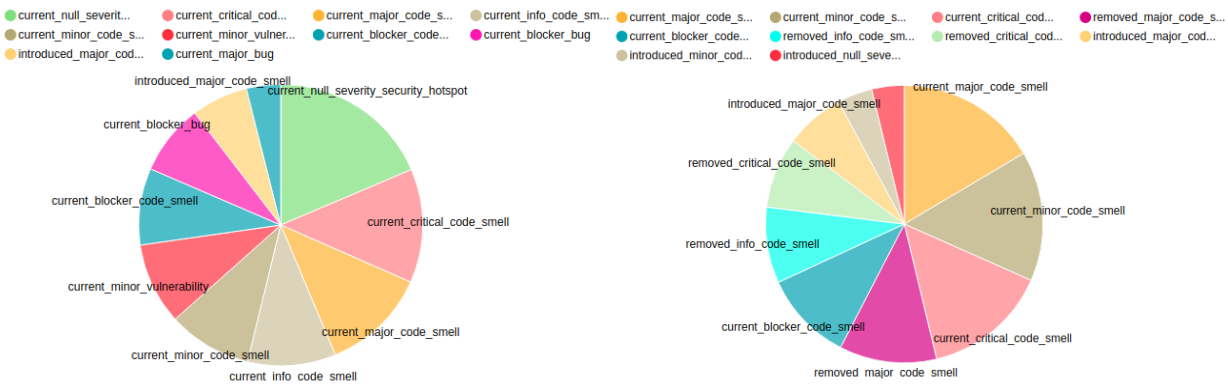

Figure 4.6. *Feature importance pie charts of Chi Square Selector model 2 (left) and Random Forest model 2 (right)*

The feature importance graph from Decision Tree model 2 is consistent with the charts in figure 4.6 above. From the importance values of features that the Machine Learning

models and Chi Square selector, we can at least be sure about certain issues that will definitely affect the build result. These issues lie in the intersection between the features from these graphs such as the current amount of code smells of all severity levels, the introduced major code smells or the current minor vulnerabilities.

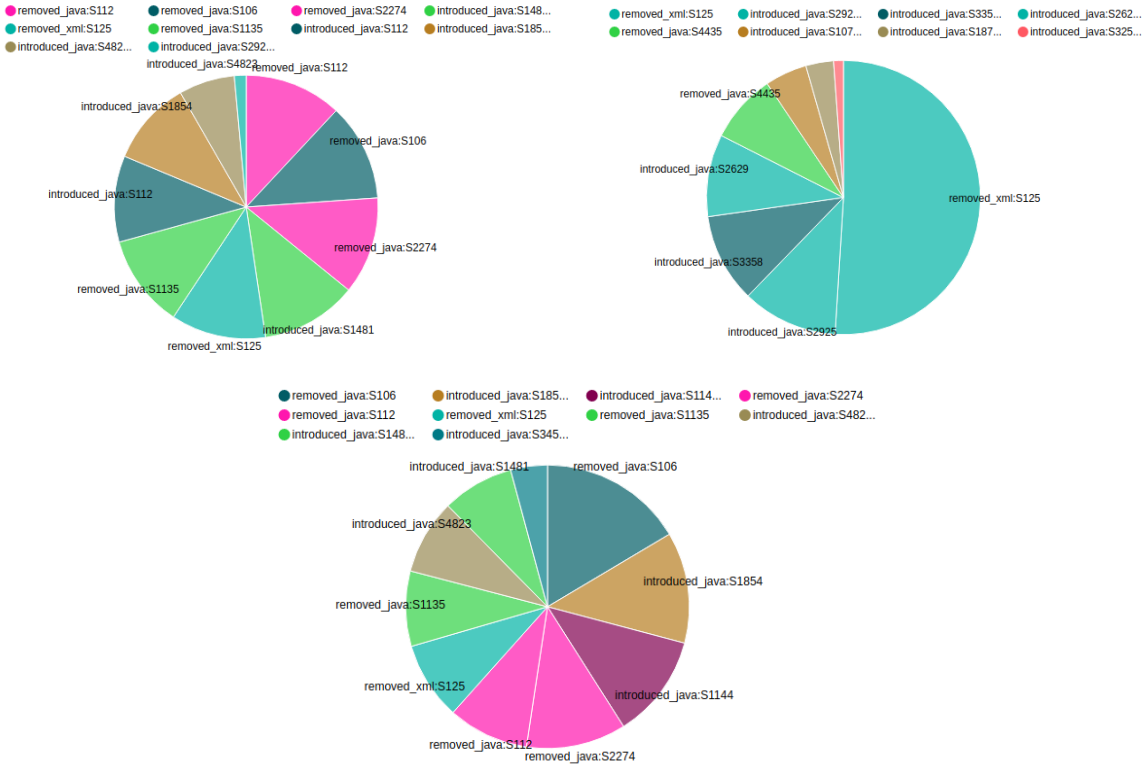

Figure 4.7. *Feature importance pie charts of Chi Square Selector model 3 (upper left), Decision Tree model 3 (upper right) and Random Forest model 3 (lower)*

As can be observed, the Random Forest model is really consistent with the Chi Square Model. However, they differ quite much from the Decision Tree model. But at least there is one rule that all the models agree on is the removed_xml:S125, which can significantly affect the build result for all the models.

5. CONCLUSION

With the growth in both the size and contribution of the open source community to open source projects, such a tool can greatly boost the development of new products. The development of such a platform is an extremely sophisticated and iterative process that requires thorough planning, careful development and continual update for each of the components. The first task is to analyze the data sources to build programs to extract data and transform them into a suitable format for permanent storage on the file system. Then a complex program is submitted to a distributed, computation-intensive system to process the extracted files, store them in persistent storage for easier analysis and visualization and apply end to end pipelines and Machine Learning algorithms. There are a backend relational database management system that stores the major data as well metadata of other tools in the platform, a scheduling system to trigger the whole process in a timely, fault-tolerant, highly resistant manner and finally a visualization service to make sense of all the data. With the separate components at hand, coordinating those components so that they can function appropriately in a continuous, autonomous manner as a compact system is the core mission. The components do not automatically cooperate out of the blue. They require constant change in source code until the point where they can function individually and system-wise.

The end result is a fully functional platform, constantly extracting, ingesting, processing, analyzing and visualizing the incoming sources of data. It demonstrates the most important attributes in the code that might affect the build result, presents the features that are vital indicators of the outcomes of the builds. From the data sources of this platform, there are various untouched raw data that can be extracted and utilized. Furthermore, there are a lot of other sources of data available they can be integrated as extended data sources. All of these can give light to deeper insight and knowledge about the open source projects from the Apache Software Foundation. The findings at the end of this platform are only a tiny fraction of the profound knowledge that is waiting to be excavated under the gigantic amount of free raw data on the net.

REFERENCES

[1] H. Agrawal, G. Chafle, S. Goyal, S. Mittal and S. Mukherjea, An Enhanced Extract-Transform-Load System for Migrating Data in Telecom Billing, 2008 IEEE 24th International Conference on Data Engineering, Cancun, 2008, pp. 1277-1286.

[2] A.S. Foundation, Apache Airflow Documentation, Available: https://airflow.apache.org/docs/stable/ (retrieved on 15/04/2020).

[3] A.S. Foundation, Apache Airflow External Triggers, Apache Airflow Documentation, Available: https://airflow.apache.org/docs/stable/scheduler.html#external-triggers (retrieved on 15/04/2020)

[4] A.S. Foundation, Apache Airflow UI, Apache Airflow Documentation, Available: https://airflow.apache.org/docs/stable/ui.html (retrieved on 15/04/2020)

[5] A.S. Foundation, Apache Software Foundation Homepage, Available: https://www.apache.org/ (retrieved on 14/04/2020).

[6] A.S. Foundation, Apache Superset Homepage, Available: https://superset.apache.org/ (retrieved on 16/04/2020).

[7] A.S. Foundation, Apache Superset Git Repository, Available: https://github.com/apache/incubator-superset (retrieved on 16/04/2020).

[8] A.S. Foundation, Spark GraphX, Apache Spark Documentation, Available: https://spark.apache.org/graphx/ (retrieved on 16/04/2020).

[9] A.S. Foundation, Spark MLLib, Apache Spark Documentation, Available: https://spark.apache.org/mllib/ (retrieved on 15/04/2020).

[10] A.S. Foundation, Spark SQL, Apache Spark Documentation, Available: https://spark.apache.org/sql/ (retrieved on 15/04/2020).

[11] A.S. Foundation, Spark Streaming, Apache Spark Documentation, Available: https://spark.apache.org/streaming/. (retrieved on 15/04/2020).

[12] S.K. Bansal, Towards a Semantic Extract-Transform-Load (ETL) Framework for Big Data Integration, 2014 IEEE International Congress on Big Data, Anchorage, AK, 2014, pp. 522-529.

[13] A.M. Berg, Jenkins Continuous Integration Cookbook, Packt Publishing, Jun 2012.

[14] G.A. Campbell, P.P. Papapetrou, Sonarqube in Action, Manning Publications, Nov 2013.

[15] F. Foundation, An Introduction to DAGs and How They Differ From Blockchains, Jun 2018, Available: https://medium.com/fantomfoundation/an-introduction-to-dags-and-how-they-differ-from-blockchains-a6f703462090 (retrieved on 15/04/2020).

[16] W. Filho, Why Is Continuous Integration So Important, Jun 2016, Available: https://medium.com/the-making-of-whereby/why-continuous-integration-is-so-important-7bb63ba5dc57 (retrieved on 14/04/2020).

[17] M. Fowler, Continuous Integration, May 2016, Available: https://moodle2019-20.ua.es/moodle/pluginfile.php/2228/mod_resource/content/2/martin-fowler-continuous-integration.pdf (retrieved on 14/04/2020).

[18] W. Garage, Python Jenkins Documentation, Available: https://python-jenkins.readthedocs.io/en/latest/index.html (retrieved on 16/04/2020).

[19] J. L. Harrington, Relational Database Design and Implementation, Elsevier Science & Technology, May 2016.

[20] H. Karau, A. Konwinski, P. Wendell, M. Zaharia, Learning Spark Lightning-fast Data Analysis, O'Reilly Media, Jan 2015.

[21] L. Foundation, About The Linux Foundation, Available: https://www.linuxfoundation.org/about/ (retrieved on 14/04/2020).

[22] P. Louridas, Static Code Analysis, IEEE, vol. 23, no. 4, pp. 58-61, 2006.

[23] P.G.D. Group, PostgreSQL Documentation, Available: https://www.postgresql.org/ (retrieved on 16/04/2020).

[24] SAS Company, What Is ETL? , Available: https://www.sas.com/en_us/insights/data-management/what-is-etl.html (retrieved on 15/04/2020).

[25] J.F. Smart, Jenkins: The Definitive Guide, O'Reilly, May 2015.

[26] S. Company, Sonarqube Documentation, Available: https://docs.sonarqube.org/ (retrieved on 15/04/2020).

[27] S. Company, Sonarqube Homepage, Available: https://www.sonarqube.org/ (retrieved on 15/04/2020).

[28] S. Company, Sonarcloud Web APIs, Available: https://sonarcloud.io/web_api/api/ (retrieved on 16/04/2020).