Polytechnic University of Turin

master's degree in computer engineering

Prediction of faults in software programs using machine learning techniques

Student: Ovidiu Birgu

Registration number: 231567

Academic advisor: Prof. Morisio Maurizio

Company advisor: Davide Piagneri

March 2020

Contents

List of Figures

List of Tables

Listings

1 Summary

Software development is a complex process, which can generate various kind of problems that are hard to identify during development.

This master thesis is about the analysis of data generated by the software production process. The available data is about commits, releases, defects. The goals of this thesis are to identify the solved problems from past history, to create models for the issue type, the severity of the problem, cross project information and bug location.

To achieve those goals, the thesis project was split as follows:
The first stage of the work involved the development of a software module (named PyGitHub) that collects raw data from GitHub using the GitHub API and saves it to local relational database. Then, the module was used to scrape (download) various open source projects.

The second stage consisted in creating a software module (named BugFinder) that reads the created databases, and iterates through all the commits in order to create the evolution history of the project from the files, extract the bugs, solutions to bugs, severity, and other meaningful information.
This information was inserted in Neo4J (a graphical database), then using the cypher query available in Neo4J several CSV files containing datasets were generated.

The third and final stage consisted in using machine learning techinique (Random Forest) and creating the models for issue type, severity and cross project issues.

Finally, the datasets were given as input to the models, and the produced results were analyzed.

2 About company

This master thesis was done in collaboration with EiSWORLD, under the supervision of Davide Piagneri, the head of analytics at EiSWORLD.

2.1 EIS SRL – general information

EiSWORLD was founded in 2011, as an ICT start up providing Business Intelligence and Business Process consulting. Four years later, it featured among the 7 fastest- growing Italian companies in terms of turnover and technical expertise, according to Deloitte.

EiSWORLD Group employs over 180 people and includes 5 Business Units that complement each other and collaborate in perfect synergy: EiS Tech, EiS Consulting, EiS Strategy, EiS Compliance and SOBRIO.

More information about the company can be found on their website at: `https://eisworld.eu/`

2.2 The office

The office where I developed the thesis is located in Via Pietro Giannone 8, Turin 10121. The workspace has a modern, open-space design, so you can communicate easily with the colleagues.

2.3 Why this thesis

One of the reasons I chose to do this thesis is for getting real life experience in application development, problem solving.

Another reason is that I had the opportunity to learn new programming languages, use new tools and complex libraries that will help me in the future.

Last but not least I got to work with experienced people who helped me every time I needed help with a problem or doubt.

3 Literature review

Defect prediction and correction in software development is a time consuming step, which increases the development cost of the software.

An IBM study[7] reported that the cost of a bug fix increases drastically in the later stages of the software development life cycle.

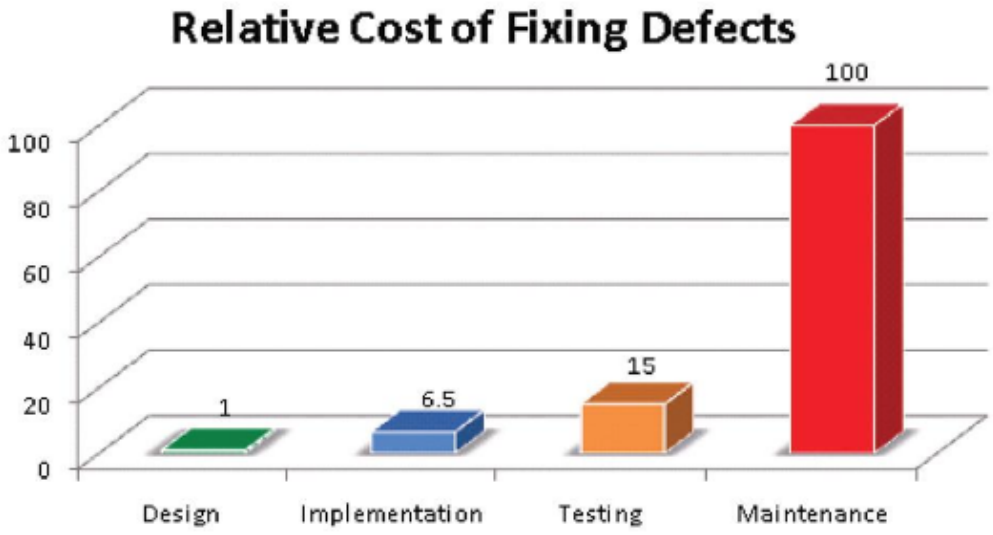

Figure 1: IBM System Science Institute Relative Cost of Fixing Defects Source: [7]

To analyze the software in a universal non biased way, several metrics that show information/properties about the code were proposed.

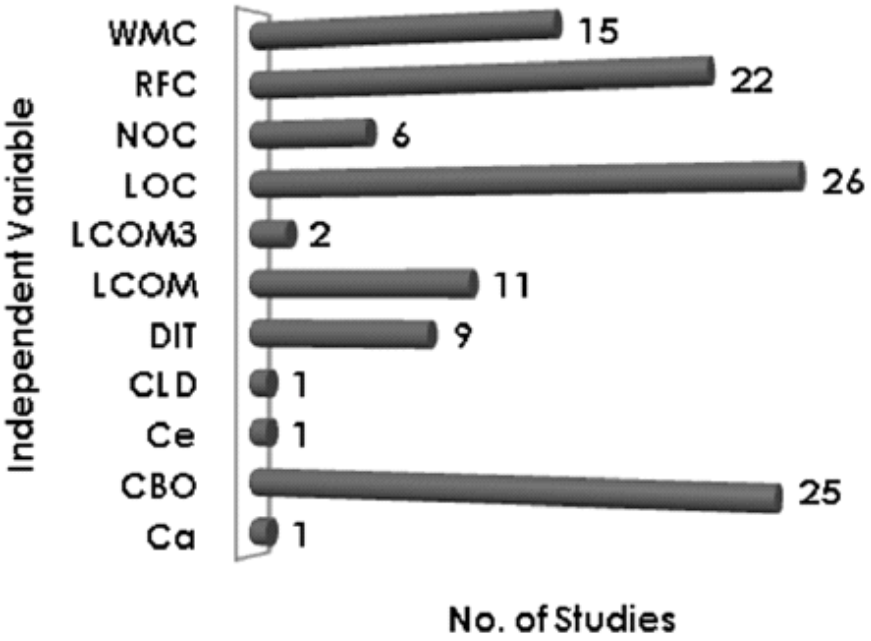

Figure 2: Metrics Fault Prediction
Source: [4]

- **WMC** - Weighted Methods for Class is used to measure the complexity of a class from the class methods
- **RFC** - Response For a Class measures the interactions of the class methods with other methods
- **NOC** - Number of Children is the number of immediate subclasses of a class
- **LOC** - Lines of code is the number of lines in the source code
- **LCOM** - Lack of Cohesion in Methods measures the cohesion on the information extracted from source code
- **DIT** - depth of inheritance tree is the number of classes that a class inherits from
- **CLD** - Class to leaf depth measures the number of lower levels in the hierarchy of the class
- **CBO** - coupling between objects is the number of classes that are coupled to another class

In order to create the models for defect prediction inside the software, various algorithms were used. The following figure illustrates the most common algorithms used in literature from 1995 to 2018.

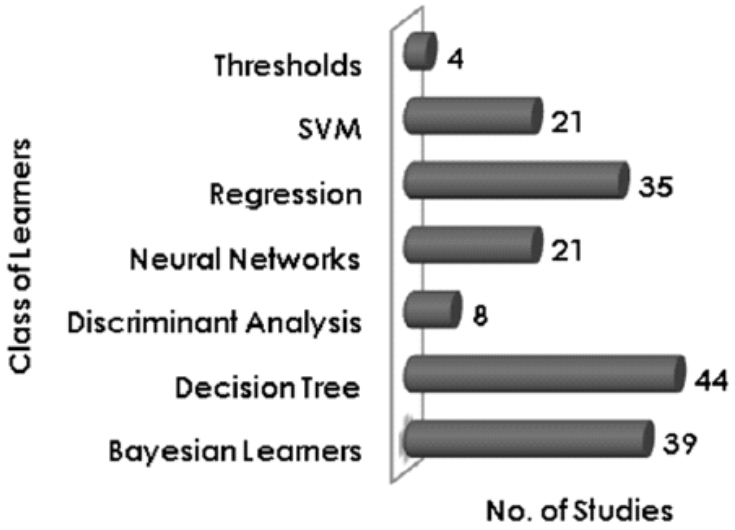

Figure 3: Algorithms and models
Source: [4]

- **Thresholds** - a statistical model, which consists in one or more threshold values, which are used to distinguish ranges of values and behavior variation.
- **SVM** - a classification machine learning algorithm, the main idea behind it is finding a hyperplane in a multi-dimentional space that separates the data points.
- **Regression** - a statistical model which consists in estimating the relationship between a dependent variable and one or more independent variables.
- **Neural Networks** - a machine learning algorithm that has at its core a system of connections, made of small nodes, similar to the actual brain.
- **Discriminant Analysis** - a statistical model similar to regression, where the dependent variable is categorical and the independent variable is interval.

- **Decision tree** - a classification machine learning algorithm, where the internal nodes represent a test on an attribute, the branches represent the outcome of a test and the leafs contain the class label.
- **Bayesian learners** - a classification machine learning algorithm that has at its core the Bayes theorem and is used to separate objects based on certain features.

The most common datasets in defect prediction are open source projects, because of the availability, size, number of contributors. The following figure illustrates the most common datasets used in literature from 1995 to 2018.

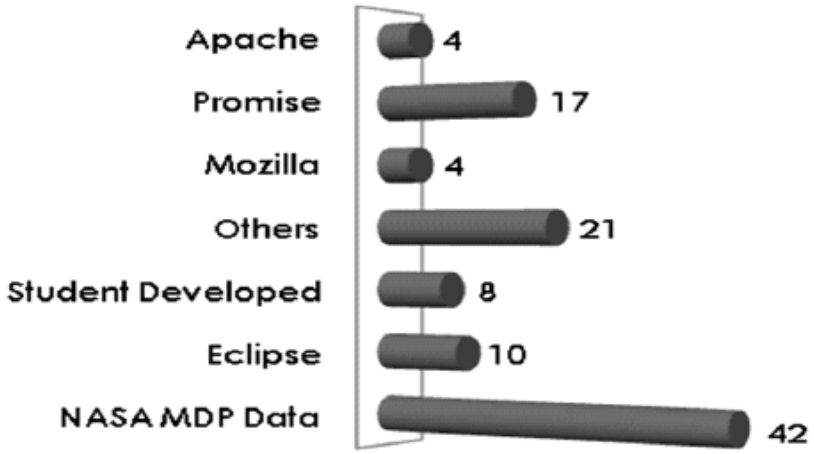

Figure 4: Most common datasets
Source: [4]

The performance of the previously defined algorithms can be measured using various metrics. The most common metrics defined in literature are shown in the following figure:

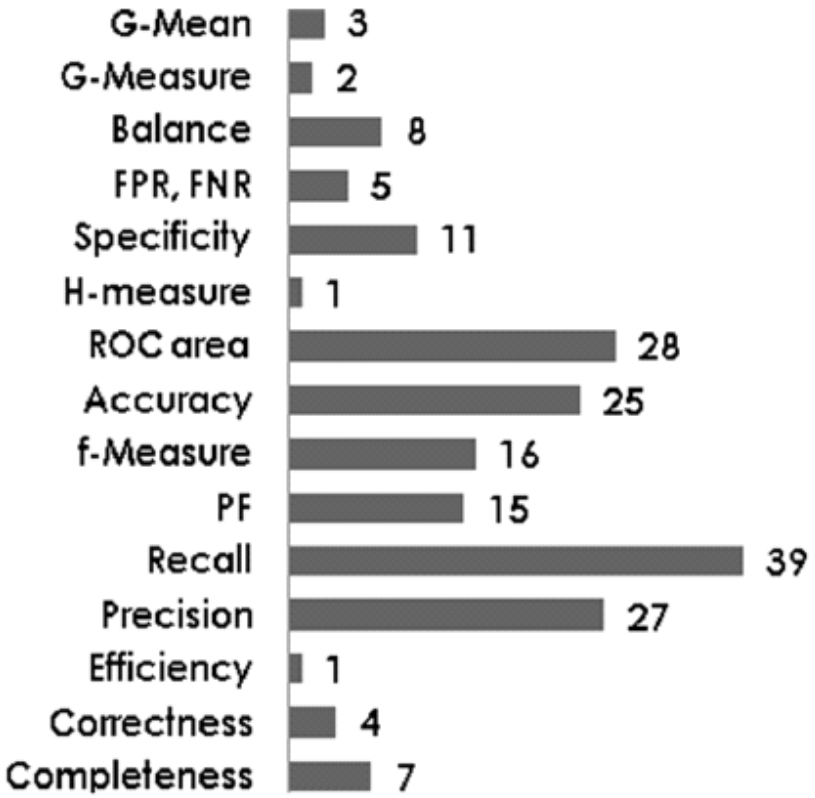

Figure 5: Most common models for performance measures
Source: [4]

The metrics in the previous figure are based on the confusion matrix, which contains two-class problems that measure the positive (hit) and negative (error) made by the classifier.

Predicted condition		Confusion matrix	
		Condition positive	Condition negative
Predicted condition	Predicted positive	TP (true positives) - the number of positive elements predicted as positive	FP (false positives) - the number of negative elements predicted as positive
	Predicted negative	FN (false negatives) - the number of positive elements predicted as negative	TN (true negatives) - the number of negative elements predicted as negative

Table 1: Confusion Matrix

*** Sensitivity, Recall, Completeness, PD (probability of detection), true positive rate (TPR)** - measures how much a classifier can detect positive values

$$\frac{TP}{TP + FN} \tag{1}$$

*** PF, FPR (false positive rate)** - probability of false alarms

$$\frac{FP}{FP + TN} \tag{2}$$

*** Balance** - is the Euclidean distance between (0, 1) and (PF, PD) points

*** Precision, correctness** - measures the ratio of positive values

$$\frac{TP}{TP + FP} \tag{3}$$

*** G-mean, G-measure** - the geometric mean of precision and recall

$$\sqrt{Precision \cdot Recall} \tag{4}$$

*** ROC Curve** - the ROC curve (receiver operating characteristic curve) is a plot that illustrates the true positive rate (TPR) against the false positive rate (FPR) at various threshold values

* **FNR** - False negative rate

$$\frac{FN}{FN + TP} \tag{5}$$

* **Specifity** - measures how much a classifier can detect negative values

$$\frac{TN}{TN + FP} \tag{6}$$

* **H-measure** - an alternative classification method to the area under the ROC curve

* **Accuracy** - determines the correct decisions made by the classifier

$$\frac{TP + TN}{TP + TN + FP + FN} \tag{7}$$

* **f-Measure** - the harmonic mean of sensitivity and precision

$$\frac{(\beta^2 + 1) \cdot Sensitivity \cdot Precision}{Sensitivity + \beta \cdot Precision} \quad \text{where } \beta \geq 0 \tag{8}$$

4 Project planning

The first step involved finding an application that interacts with GitHub and returns all the useful information. Unfortunately, there was no application that met all the requirements needed.

In order to retrieve information from GitHub, various methods can be used (WEB/API), the following table illustrates the differences:

	Advantages	Disadvantages
Web	There are no limitations for number of requests (some sites might have captcha, but they can be bypassed by using captcha solvers, changing ip)	Can break when design of page changes; Slower than API call
API	Data is retrieved in a predefined format (e.g. Json, XML) - Fast request response	Limitations for number of requests

Table 2: Web vs. API scraping

After analyzing the GitHub documentation and API, the UML containing the required information for the thesis project was produced.

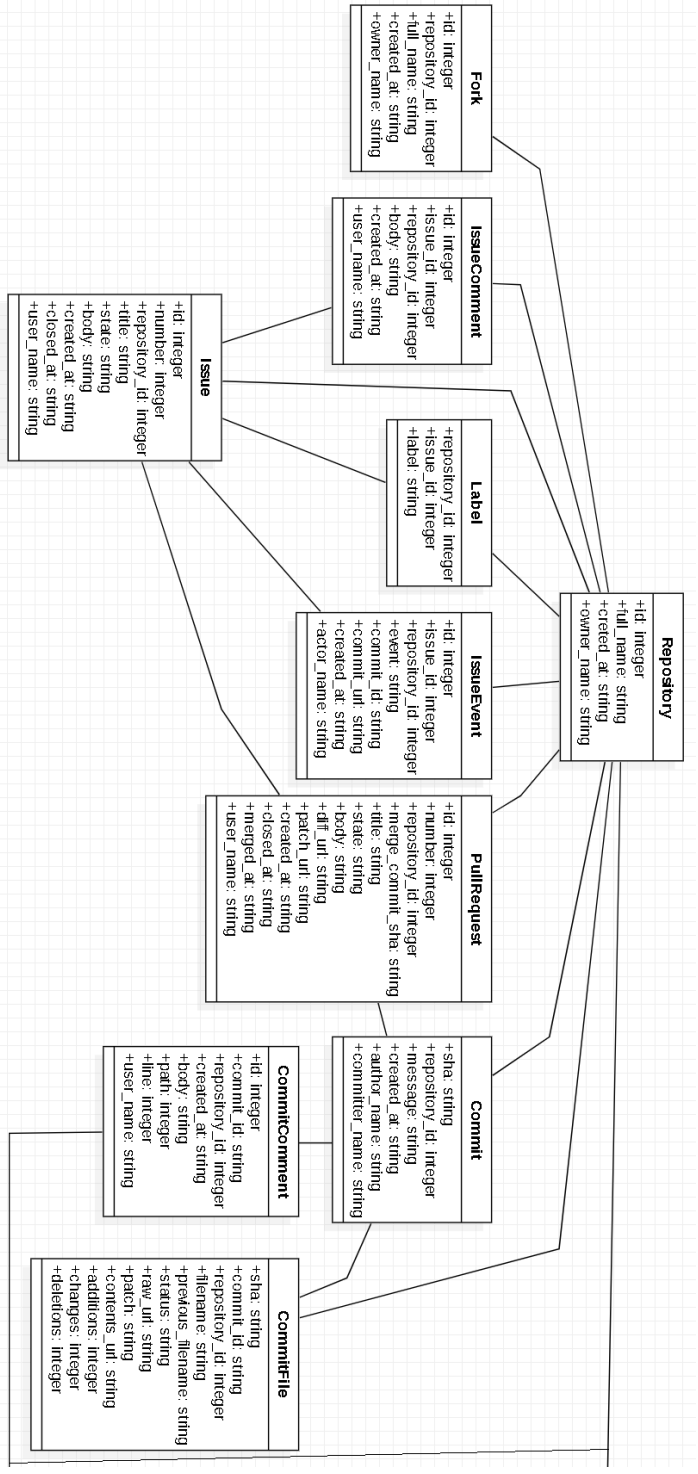

Figure 6: UML Information GitHub

The next step involved the planning of several modules for the tasks required by the thesis: data extraction, models for the issue type, severity of the problem, cross project information and bug location.

The first module has the role of saving the data coming from GitHub API to a database. It uses the following technologies: JSON format for data coming from GitHub API, Python code for the module language, SQLite database for data extraction.

The data is saved inside an SQLite database because it is portable (everything contained in one single file), faster (less disk I/O operations compared to saving to multiple CSV files).

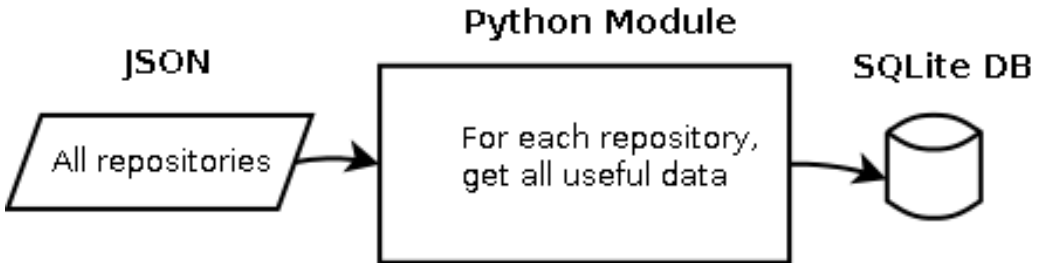

Figure 7: PyGitHub scraper module

The second module reads the previously saved database and generates a graph like database, which can be used to visually see information about commits, issues, bug history, etc.

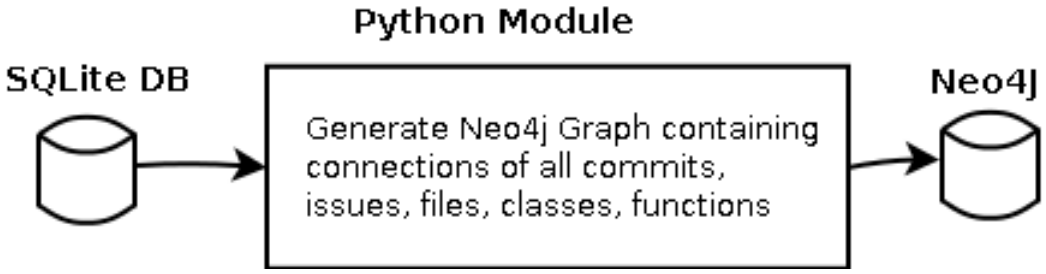

Figure 8: BugFinder module

Inside of this module there is a submodule that reads dataset CSV files (generated from the Neo4J database) and applies them to various models like issue type, severity, cross project.

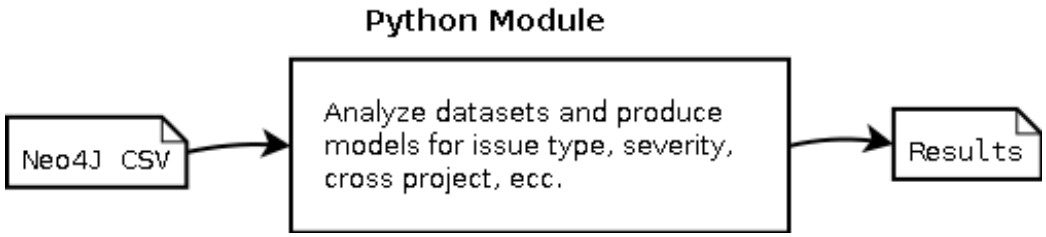

Figure 9: Classifier module

5 Project implementation

5.1 Development Environment

The thesis project was developed on a computer with Microsoft Windows 10 operating system, Python version 3.7 and Community Edition of Visual Studio, a popular IDE that supports many languages, and has a lot of plugins.

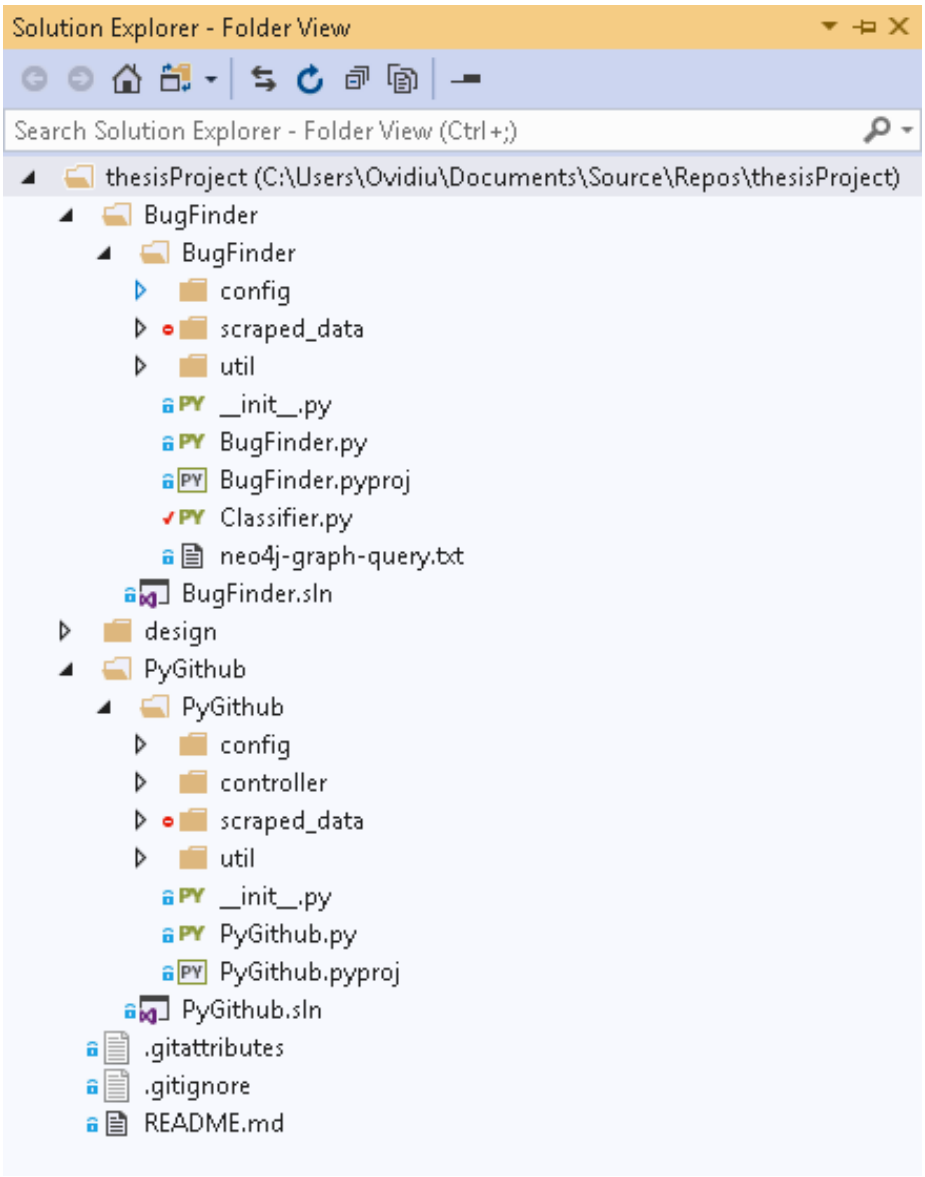

Figure 10: Project structure

On the Visual Studio IDE, the GitLab extension was installed, in order to facilitate the development process.

In order to keep the thesis project modular and easy to maintain, it was split in two subprojects ("BugFinder", "PyGithub").

Besides Visual Studio IDE, other software had to be installed on the computer: Neo4J server, SQLite browser.

Neo4j server was installed as a windows console application, and executed from command prompt after setting the various ports for the Neo4J Bolt driver connector, http/https connectors.

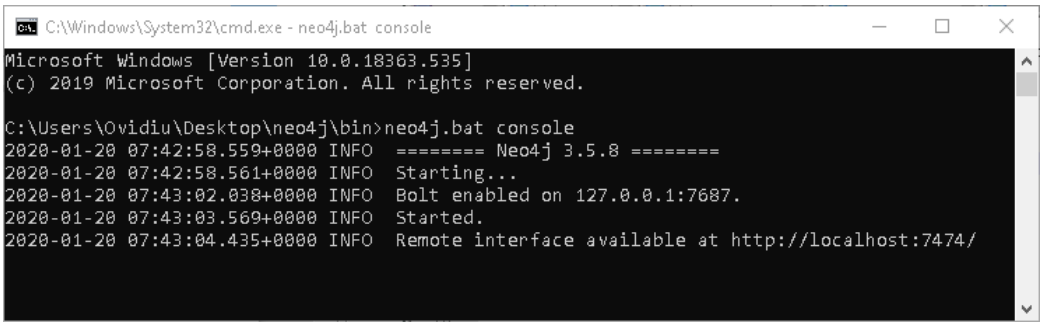

Figure 11: Neo4j server console

5.2 PyGitHub module

This module serves the purpose of downloading the data from GitHub API and saving it to a local database.

It uses the external Python library "PyGithub", which was installed by using the following command:

```
1 pip install PyGitHub
```

Listing 1: PyGitHub external libraries

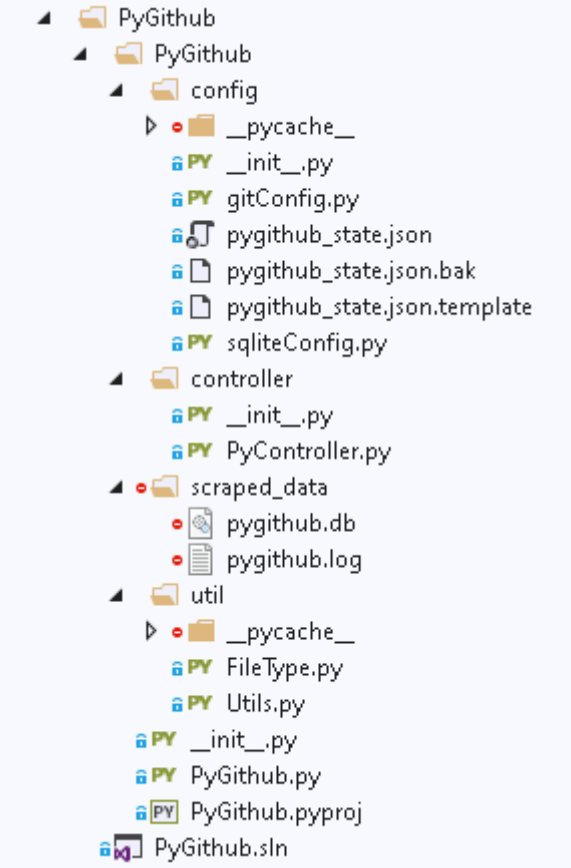

Figure 12: PyGitHub module

The composing blocks of the module are:

1. main execution script PyGitHub.py
2. config
3. util
4. scraped_data
5. controller

1. This script starts the controller, initializes the directory path for scraped data, log file. In case the controller encounters errors and terminates, this script restarts the controller. The script will terminate when the controller calls the system call exit.

2. This folder contains the following configuration files:

(a) gitConfig.py

(b) sqliteConfig.py

(c) pygithub_state.json

(a) The role of this class is the creation and initialization of a GitHub instance object using username and password.

```
1 class GitConfig(object):
2     def __init__(self, user, password):
3         # timeout 10 seconds, elements per page = 100, times
              number retry = 10
4         self.g = Github(user, password, timeout=10, per_page
              =100, retry=10)
5     def getGitHubInstance(self):
6         return self.g
```

Listing 2: PyGitHub module - GitConfig class

(b) This class contains the functions that interact with the database. The code is reusable thanks to wrapper functions, dynamic sql query (not hard-coded, but build query from variables).

```
1  class SQLiteConfig(object):
2      def __init__(self):
3          #...
4      def createConnectionDB(self, urlDB):
5          #...
6      def closeConnectionDB(self):
7          #...
8      def createTables(self):
9          #...
10     def insertItemDB(self, table, columns, values):
11         #...
12     def getItemDB(self, table, field_name, field_value):
13         #...
14     def getItemMultipleCondsDB(self, table, field_names,
           field_values):
15         #...
16     def updateItemDB(self, table, field_name, field_value,
           cond_field, cond_value):
17         #...
18     def updateItemMultipleCondsDB(self, table, field_name,
           field_value, cond_fields, cond_values):
19         #...
20     def getItemsDB(self, table, field_names, field_values):
21         #...
22     # remove any character that is not alphanumeric, underscore
           or dash from string
23     # to avoid sql injections
24     def sanitizeString(self, unsafe_string):
25         #...
```

Listing 3: PyGitHub module - SQLiteConfig class

(c) This configuration file contains the state save for the module. The saving of the state is requred because the scraping process can be long and it depends on the internet connection to the GitHub servers.

Also, the GitHub API imposes a limit for the number of requests that can be made in a time interval. To solve this problem, the configuration file uses multiple GitHub accounts, and rotates them when the number of requests of one account is consumed.

Inside of the configuration file there are also the repositories that need to be scraped, the flags which tell if the scraping of the various elements (e.g. *Commits*, *Issues*, etc.) of the repository finished, and the *session_token* which is useful when a scraped repository needs to be updated.

```
{
  "py_auth": [
    {
      "user": "username1",
      "password": "password1",
      "reset_time": 1558890000
    },
    {
      "user": "username2",
      "password": "password2",
      "reset_time": 1558890000
    }
  ],
  "current_py_auth": 0,
  "repositories": [
    "owner1/reponame1",
    "owner2/reponame2"
  ],
  "current_repository": 0,
  "crawlRepositoryFinished": false,
  "crawlForksFinished": false,
  "crawlIssuesFinished": false,
  "crawlIssueEventsFinished": false,
  "crawlIssueCommentsFinished": false,
  "crawlPullRequestsFinished": false,
  "crawlCommitsFinished": false,
  "crawlCommitCommentsFinished": false,
  "crawlCommitFilesFinished": false,
  "downloadCommitFilesPrep": false,
  "downloadCommitFilesFinished": false,
  "session_token": "qwerty12",
  "session_last_page": 0
}
```

Listing 4: PyGitHub module - state save

3. This folder contains utility functions for type of file detection, time conversion, session token generation.

```
class FileType(Enum):
    @staticmethod
    def from_str(str):
        #...

class Utils(object):
    @staticmethod
    def generate_token(size=8, chars=string.ascii_lowercase +
        string.digits):
```

```
9           #...
10      @staticmethod
11      def getUTCTime():
12          #...
```

Listing 5: PyGitHub module - Utility classes

4. In this folder, the output log file and SQLite database are stored.

The log file contains information about events, errors.

```
1  INFO:root:Started PyController
2  INFO:root:Changed user for rate requests: current UTC timestamp
       1561785988
3  INFO:root:Scraping of repository pandas-dev/pandas started
4  ...
5  INFO:root:Started scraping issue events: current UTC timestamp
       1561811857
6  INFO:root:Rate status 4647/5000
7  INFO:root:Changed user for rate requests: current UTC timestamp
       1561814757
8  WARNING:urllib3.connectionpool:Retrying (Retry(total=9, connect=
       None, read=None, redirect=None, status=None)) after
       connection broken by 'ReadTimeoutError("HTTPSConnectionPool(
       host='api.github.com', port=443): Read timed out. (read
       timeout=10)")': /repos/pandas-dev/pandas/issues/850/events?
       per_page=100
9  WARNING:urllib3.connectionpool:Retrying (Retry(total=9, connect=
       None, read=None, redirect=None, status=None)) after
       connection broken by 'ReadTimeoutError("HTTPSConnectionPool(
       host='api.github.com', port=443): Read timed out. (read
       timeout=10)")': /repos/pandas-dev/pandas/issues/531
10 INFO:root:Changed user for rate requests: current UTC timestamp
       1561816998
11 INFO:root:Finished scraping issue events: current UTC timestamp
       1561817473
12 INFO:root:Rate status 3957/5000
13 ...
14 INFO:root:Scraping of repository pandas-dev/pandas finished
15 INFO:root:Scraping done!
```

Listing 6: PyGitHub module - log

The SQLite database is stored as a regular file on the disk, and its schema includes all information defined in chapter 4 figure(link uml).

Name table	Schema
commit_comments	CREATE TABLE commit_comments (id INTEGER, commit_id TEXT, repository_id INTEGER, created_at TEXT, body TEXT, path TEXT, line INTEGER, user_name TEXT, session_token TEXT)
commit_files	CREATE TABLE commit_files (sha TEXT, commit_id TEXT, repository_id INTEGER, filename TEXT, previous_filename TEXT, status TEXT, raw_url TEXT, patch TEXT, contents_url TEXT, additions INTEGER, changes INTEGER, deletions INTEGER)
commits	CREATE TABLE commits (sha TEXT, repository_id INTEGER, message TEXT, files_scraped INTEGER, created_at TEXT, author_name TEXT, committer_name TEXT, session_token TEXT)
download_commit_files	CREATE TABLE download_commit_files (commit_id TEXT, repository_id INTEGER, filename TEXT, raw_url TEXT, downloaded INTEGER, content TEXT)
forks	CREATE TABLE forks (id INTEGER, repository_id INTEGER, full_name TEXT, created_at TEXT, owner_name TEXT, session_token TEXT)
issue_comments	CREATE TABLE issue_comments (id INTEGER, issue_id INTEGER, repository_id INTEGER, body TEXT, created_at TEXT, user_name TEXT, session_token TEXT)
issue_events	CREATE TABLE issue_events (id INTEGER, issue_id INTEGER, repository_id INTEGER, event TEXT, commit_id TEXT, commit_url TEXT, created_at TEXT, actor_name TEXT, session_token TEXT)
issues	CREATE TABLE issues (id INTEGER, number INTEGER, repository_id INTEGER, title TEXT, state TEXT, body TEXT, events_scraped INTEGER, comments_scraped INTEGER, created_at TEXT, closed_at TEXT, user_name TEXT, session_token TEXT)

Name table	Schema
labels	CREATE TABLE labels (repository_id INTEGER, issue_id INTEGER, label TEXT)
pull_requests	CREATE TABLE pull_requests (id INTEGER, number INTEGER, repository_id INTEGER, merge_commit_sha TEXT, title TEXT, state TEXT, body TEXT, diff_url TEXT, patch_url TEXT, created_at TEXT, closed_at TEXT, merged_at TEXT, user_name TEXT, session_token TEXT)
repositories	CREATE TABLE repositories (id INTEGER, full_name TEXT, created_at TEXT, owner_name TEXT, session_token TEXT)

Table 3: PyGitHub database schema

5. Inside this folder there is the controller, which contains the functions responsible for scraping, database interaction, state initialization/save.

```
1  class PyController(object):
2      def __init__(self, config_state_file, repo_name):
3          #...
4      def getGitHubInstance(self):
5          #...
6      def exceededRateLimit(self, rate_limit, rate_remaining):
7          #...
8      def getConfigStateItem(self, field):
9          #...
10     def updateConfigStateField(self, field, value):
11         #...
12     def getRepo(self):
13         #...
14     def scrapeRepository(self):
15         #...
16     def scrapeForks(self):
17         #...
18     def scrapeIssues(self):
19         #...
20     def scrapeIssueEvents(self):
21         #...
22     def scrapeIssueComments(self):
23         #...
24     def scrapePullRequests(self):
25         #...
```

```
26      def scrapeCommits(self):
27          #....
28      def scrapeCommitComments(self):
29          #...
30      def scrapeCommitFiles(self):
31          #...
32      def downloadCommitFilesPrep(self):
33          #...
34      def downloadCommitFiles(self):
35          #...
36      def downloadFile(self, target_url):
37          #...
38      def getLastParamURL(self, url):
39          #...
40      def saveItemDB(self, table, item, session_token):
41          #...
42      def getItemDB(self, table, itemType, item):
43          #...
44      def getItemMultipleCondsDB(self, table, field_names,
            field_values):
45          #...
46      def getItemsDB(self, table, field_names, field_values):
47          #...
48      def updateItemDB(self, table, field_name, field_value,
            itemType, item):
49          #...
50      def updateItemMultipleCondsDB(self, table, field_name,
            field_value, conds_n, conds_v):
51          #...
52      def closeDB(self):
53          #...
54      def totalRequiredRequests(self):
55          #...
```

Listing 7: PyGitHub module - controller class

The first step is initialization through *__init__* function, which reads the configuration files and the previous state.

In case it is needed to estimate the number of requests required for the scraping of the repository, the *totalRequiredRequests* function can be used, which sums up all the number of items for each element of the repository. This function consumes very few requests from the user requests pool, because it uses the *totalCount* GitHub API method.

```
1       def totalRequiredRequests(self):
2           # how many items can be requested in 1 page
3           # max value set by github api is 100, min is 1
4           per_page = 100
5           # coeff computed after analyzing various repositories
6           coeff = 1.5
7
8           tot_forks = self.getRepo().get_forks().totalCount
9           tot_pr = self.getRepo().get_pulls(state="all").
                totalCount
10          tot_commit_comments = self.getRepo().get_comments().
                totalCount
11          tot_issues = self.getRepo().get_issues(state="all").
                totalCount
12          tot_commits = self.getRepo().get_commits().totalCount
13          number_requests = 1 # initial repository info
14          number_requests += (tot_forks + tot_pr +
                tot_commit_comments)/per_page + tot_issues*(1/
                per_page+2*coeff) + tot_commits*(1/per_page+1)
15
16          return math.ceil(number_requests)
```

Listing 8: PygitHub module - calculation of number of requests

Test case environment:

- SQLite database on RAMDISK storage device
- cpu intel i7 8700k at 5Ghz
- optic fiber 1gbit/s internet connection

Results obtained:

- total number of requests: 50,000
- number of processed requests: 10,000/hour on average

So it would take 5 hours to process and at least 3 GitHub accounts are needed.

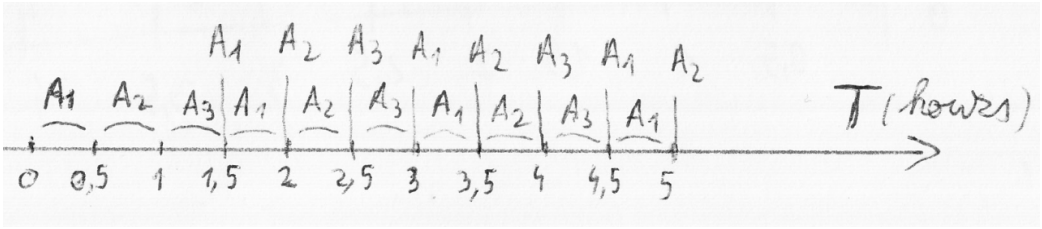

Figure 13: Scraping with 3 accounts and 30 min time window

Next step is the scraping process, which is divided in many functions, each dealing with a different element in the repository (e.g commit, issue, event, etc.). Each time an element is scraped, it is saved to the SQLite database.

```
1     def scrapeCommits(self):
2         logging.info("Started scraping commits: current UTC
              timestamp %s", Utils.getUTCTime())
3         logging.info("Rate status %d/%d", self.g.rate_limiting
              [0], self.g.rate_limiting[1])
4
5         # flag that checks if I reached the position of previous
              scraping session
6         previous_list_reached = False
7         page_number = 0
8         commits_page = self.getRepo().get_commits().get_page(
              page_number)
9         while len(commits_page) != 0:
10            i = 0
11            # process page
12            # save commit data
13            for commit in commits_page:
14                db_commit = self.getItemDB("commits", "sha",
                      commit.sha)
15                if db_commit == None:
16                    self.saveItemDB('commits', commit, self.
                          session_token)
17                else:
18                    db_commit_session_token = db_commit[len(
                          db_commit)-1]
19                    if db_commit_session_token != self.
                          session_token:
20                        previous_list_reached = True
21                        break
22                    else:
23                        # same token from previous scraping
24                        # jump to scraping page of previous run
```

```
25                          # to avoid wasting rate requests
26                          if i == len(commits_page) - 1:
27                              page_number += self.
                                    session_last_page
28                  i += 1
29              # page completed
30              # update session_last_page
31              # note: this condition preserves the
                    session_last_page from next runs of scraper
32              if page_number >= self.session_last_page:
33                  self.updateConfigStateField('session_last_page',
                        page_number)
34
35              if not previous_list_reached:
36                  # advance to next page
37                  page_number += 1
38                  commits_page = self.getRepo().get_commits().
                        get_page(page_number)
39              else:
40                  break
41
42          # update json state
43          self.updateConfigStateField('crawlCommitsFinished', True
                )
44          self.updateConfigStateField('session_last_page', 0)
45          logging.info("Finished scraping commits: current UTC
                timestamp %s", Utils.getUTCTime())
46          logging.info("Rate status %d/%d", self.g.rate_limiting
                [0], self.g.rate_limiting[1])
```

Listing 9: PyGitHub module - commits scraping function

There is also a *downloadCommitFiles* function that interacts with GitHub through HTTP requests and not API requests, because it is faster. This function uses the *downloadFile* function, which implements the Python *requests* library.

```
1       def downloadFile(self, target_url):
2           # timeout for the download request is set to 10 seconds
3           response = requests.get(target_url, timeout=10)
4           if response.status_code != requests.codes.ok:
5               response.raise_for_status()
6
7           return response.text
```

Listing 10: PyGitHub module - file download function

The scraping process speed can be increased greatly by using RAMDISK (RAM memory as a filesystem partition).

Instructions for Linux operating system:

- Creation RAMDISK:
a) create directory for partition: *"sudo mkdir -p /media/ramdisk"*
b) create the partition and mount it to the directory created previously: *"sudo mount -t tmpfs -o size=<partition size in MBytes>M tmpfs /media/ramdisk"*
e.g.: *"sudo mount -t tmpfs -o size=2048M tmpfs /media/ramdisk"* for a 2Gbytes partition
c) check if partition was created and has correct size: *"df -h"*

- Removing RAMDISK:
a) unmount the partition: *"sudo umount /media/ramdisk"*
b) if an error like *"umount: /media/ramdisk: target is busy"* is encountered, then it required to check which process is using the partition: *"sudo lsof -n /media/ramdisk"* get the PID (process id) and terminate the processes with: *"kill -9 <PID>"*
c) remove the directory that you created for the partition: *"sudo rmdir /media/ramdisk"*

Another method to increase scraping speed is to run multiple instances of the scraper, one for each repository. The following figure illustrates the execution of 6 instances of the PyGitHub module, each instance uses 1 execution thread of the cpu.

Figure 14: PyGitHub module multiple instances execution

5.3 BugFinder module

This module reads the SQLite database generated by the previous PyGitHub module and creates the Neo4J history graph containing information about commits, issues, classes, functions, etc. It uses the external libraries neo4j, neo4j-driver, which were installed by using the following commands:

```
1 pip install neo4j
2 pip install neo4j-driver
```

Listing 11: BugFinder external libraries

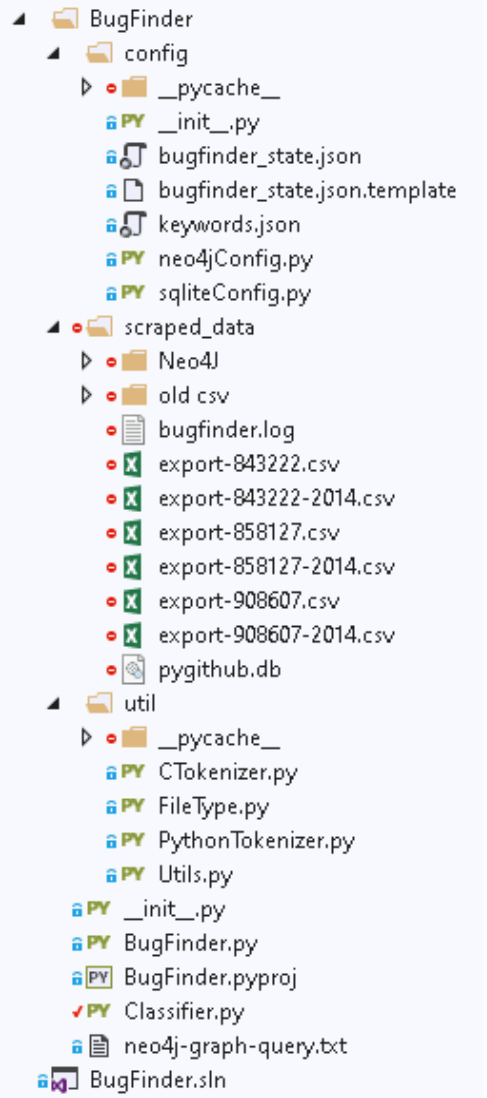

Figure 15: PyGitHub module

The composing blocks of the module are:

1. config
2. util
3. scraped_data
4. main execution script BugFinder.py

1. This folder contains the following configuration files:

(a) sqliteConfig.py

(b) neo4jConfig.py

(c) keywords.json

(d) pygithub_state.json

(a) This class contains the functions that interact with the database. The code is reusable thanks to wrapper functions, dynamic sql query (not hard-coded, but build query from variables).

```
1  class SQLiteConfig(object):
2      def __init__(self):
3          #...
4      def createConnectionDB(self, urlDB):
5          #...
6      def closeConnectionDB(self):
7          #...
8      # returns all items of table from database
9      def getItemsDB(self, table, order_col=None):
10         #...
11     # returns filtered items of table from requested database
12     def getFilteredItemsDB(self, table, filter_op, field_name,
           field_value, order_col=None):
13         #...
14     # returns the requested item from table of database
15     def getItemDB(self, table, field_name, field_value):
16         #...
17     # returns the requested item that verifies the condition
           from table of database
18     def getItemMultipleCondsDB(self, table, field_names,
           field_values):
19         #...
```

```
20      def getItemDBFromPartialField(self, table, field_name,
            field_value):
21          #...
22      # remove any character that is not alphanumeric, underscore
            or dash from string
23      def sanitizeString(self, unsafe_string):
24          #...
```

Listing 12: BugFinder module - SQLiteConfig class

(b) This class contains the functions that interact with the Neo4J graph database. The code is reusable thanks to wrapper functions, cypher query building (not hardcoded, but build query from variables).

```
1  class Neo4jConfig(object):
2      def __init__(self, uri, user, password):
3          #...
4      def close(self):
5          #...
6      def initNeo4j(self):
7          #...
8      # remove any character that is not alphanumeric, underscore
           or dash from string
9      def sanitizeString(self, unsafe_string):
10         #...
11     def doOperation(self, operation, data):
12         #...
13     def getNode(self, node):
14         #...
15     def createNode(self, node):
16         #...
17     # creates a neo4j relationship between two nodes
18     def createConnection(self, src, dest, data):
19         #...
20     # creates a neo4j relationship(that has a property) between
           two nodes
21     def createConnectionWithParam(self, src, dest, data):
22         #...
23     # updates properties of a neo4j node
24     # the update of node is based on identifier from Update
           property name
25     def updateNode(self, node):
26         #...
27     # updates property for a neo4j connection
28     def updateConnectionWithParam(self, src, dest, data):
29         #...
30     def existsConnection(self, src, dest, data):
```

```
31          #...
32      def existsNode(self, node):
33          #...
34      def getNodeOP(self, tx, node):
35          #...
36      def createNodeOP(self, tx, node):
37          #...
38      def createConnectionOP(self, tx, conn):
39          #...
40      def createConnectionWithParamOP(self, tx, conn):
41          #...
42      def updateNodeOP(self, tx, node):
43          #...
44      def updateConnectionWithParamOP(self, tx, conn):
45          #...
46      def existsConnectionOP(self, tx, conn):
47          #...
48      def existsNodeOP(self, tx, node):
49          #...
```

Listing 13: BugFinder module - Neo4jConfig class

(c) This file contains keywords and their variations related to bug identification, solution detection and security.

```
1  {
2    "problems": [
3      "problem",
4      "problems",
5      "issue",
6      "issues",
7      "bug",
8      "bugs",
9      "error",
10     "errors",
11     "fix",
12     "fixes"
13   ],
14   "solutions": [
15     "fix",
16     "fixes",
17     "fixing",
18     "solves",
19     "close",
20     "closes"
21   ],
22   "security": [
```

```
23      "security",
24      "vulnerable",
25      "vulnerability",
26      "ssl",
27      "ssh",
28      "authenticate",
29      "authentication",
30      "forbidden",
31      "authorized",
32      "injection"
33    ]
34  }
```

Listing 14: BugFinder module - Keywords file

(d) This configuration file contains the authentication for the Neo4J server and state save for the module. The saving of the state is required because the Neo4J graph generation process can be long and it depends on the connection to the Neo4J server.

```
1  {
2    "neo4j_auth": {
3      "user": "neo4j",
4      "password": "1234",
5      "url": "bolt://localhost:7687"
6    },
7    "last_processed_commit": "
         ae4e51092d27520d3ed1ac1dcae658be33ff096a"
8  }
```

Listing 15: BugFinder module - state save

The *last_processed_commit* variable contains the SHA (secure hash algorithm) of the last commit (ordered by timestamp) that has to be processed (initially this variable is empty).

2. Inside this folder there are tokenizers for different programming languages and various utility functions useful for time manipulation, token generation, file type detection.

```
1  class FileType(Enum):
2      @staticmethod
3      def from_str(str):
4          #...
```

```
5
6  class Utils(object):
7      @staticmethod
8      def generate_token(size=8, chars=string.ascii_lowercase +
           string.digits):
9          #...
10     @staticmethod
11     def getUTCTime():
12         #...
13     @staticmethod
14     def getNumberdaysInterval(start_date, end_date):
15         #...
16     @staticmethod
17     def stringToDatetime(str):
18         #...
```

Listing 16: BugFinder module - Utility classes

The Python tokenizer uses the Python *tokenize* library and has the purpose of identifying which are the classes and functions composing the Python code. This is done by splitting the Python code in tokens (five tuple elements containing type, content, position in code), which are analyzed and saved in case they are related to classes or functions.

```
1  class PythonTokenizer(object):
2      @staticmethod
3      def tokenizeCode(content):
4          #...
5          try:
6              g = tokenize.tokenize(BytesIO(content.encode('utf-8'
                   )).readline)
7              for fiveTuple in g:
8                  '''''''
9                  available elements:
10                 fiveTuple.type - compare with type from tokenize
                       .<type>
11                 fiveTuple.string - contains the token string
12                 fiveTuple.start - 2-tuple (srow, scol) where the
                       token begins in the source
13                 fiveTuple.end - 2-tuple (erow, ecol) where the
                       token ends in the source
14                 fiveTuple.line - the line on which the token was
                       found
15                 '''''''
16                 if fiveTuple.type == tokenize.NAME:
17                     # check if end of classes/functions was
                           reached
```

```
18                      while True:
19                          # get last element from stack
20                          if not stackData.empty():
21                              stack_elem = stackData.get()
22                              # check if this token is on a new
                                    line and has less indentation
23                              if fiveTuple.start[0] > stack_elem
                                    [2] and fiveTuple.start[1] <=
                                    stack_elem[3]:
24                                  if stack_elem[0] == "class":
25                                      classes.append([stack_elem
                                            [1], stack_elem[2],
                                            fiveTuple.start[0]-1])
26                                  elif stack_elem[0] == "function"
                                        :
27                                      functions.append([stack_elem
                                            [1], stack_elem[2],
                                            fiveTuple.start[0]-1])
28                              else:
29                                  # put element back on stack
30                                  stackData.put(stack_elem)
31                                  break
32                          else:
33                              break
34
35                      # check if beginning of class
36                      if fiveTuple.string == "class":
37                          prevWasClass = True
38                          indentationClass = fiveTuple.start[1]
39                          lineStartClass = fiveTuple.start[0]
40                      # check if beginning of function
41                      elif fiveTuple.string == "def":
42                          prevWasDef = True
43                          indentationDef = fiveTuple.start[1]
44                          lineStartDef = fiveTuple.start[0]
45                      # not class and not function
46                      else:
47                          if prevWasClass:
48                              stackData.put(["class", fiveTuple.
                                    string, lineStartClass,
                                    indentationClass])
49                              prevWasClass = False
50                          elif prevWasDef:
51                              stackData.put(["function", fiveTuple
                                    .string, lineStartDef,
                                    indentationDef])
52                              prevWasDef = False
53                  elif fiveTuple.type == tokenize.ENDMARKER:
54                      # save last elements from stack
```

```
55                      while not stackData.empty():
56                          stack_elem = stackData.get()
57                          if stack_elem[0] == "class":
58                              classes.append([stack_elem[1],
                                    stack_elem[2], fiveTuple.start
                                    [0]-1])
59                          elif stack_elem[0] == "function":
60                              functions.append([stack_elem[1],
                                    stack_elem[2], fiveTuple.start
                                    [0]-1])
61
62          except Exception as e:
63              # in case of parsing problems, just skip the
                    remaining part of the code block
64              logging.info("Problem tokenizer python: "+str(e))
65
66          result.append(classes)
67          result.append(functions)
68          return result
```

Listing 17: BugFinder module - PythonTokenizer class

3. This folder contains the input SQLite database generated by the PyGitHub module in chapter 5.3, the output Neo4J graph database and log file.

The log file contains information about events, errors. Most of the errors are generated from the tokenizer classes if the analyzed input code is not syntactically correct.

```
1 INFO:root:Started bug finder
2 INFO:root:Started creating neo4j graph: current UTC timestamp
    1565348618
3 INFO:root:Problem tokenizer python: ('EOF in multi-line
    statement', (40, 0))
4 INFO:root:Problem tokenizer python: ('EOF in multi-line
    statement', (196, 0))
5 INFO:root:Problem tokenizer python: unindent does not match any
    outer indentation level (<tokenize>, line 63)
6 ...
7 INFO:root:Finished creating neo4j graph: current UTC timestamp
    1565445604
8 INFO:root:Work done!
```

Listing 18: BugFinder module - log

The Neo4J graph database has the following schema and constraints:

Node Label	Property keys
Commit	class_id, author, committer, repository_id, sha, timestamp
Issue	author, created_at, issue_id, issue_security, issue_type, labels, number_comments, number_events, number_people, timestamp
File	commit_file_id, lines_added, lines_removed, name, number_classes, number_functions, number_lines_code, timestamp
Class	class_id, name, timestamp
Function	func_id, name, timestamp

Table 4: Nodes Neo4J graph

Relationship	Source Node	Destination Node
next	Commit	Commit
closes	Commit	Issue
modified	Commit	File
changed	File	Class
		Function
uses	File	File

Table 5: Relationships Neo4J graph

Node Label	Constraint
Commit	CREATE CONSTRAINT ON (c:Commit) ASSERT c.sha IS UNIQUE
Issue	CREATE CONSTRAINT ON (f:File) ASSERT f.commit_file_id IS UNIQUE
File	CREATE CONSTRAINT ON (i:Issue) ASSERT i.issue_id IS UNIQUE
Class	CREATE CONSTRAINT ON (cl:Class) ASSERT cl.class_id IS UNIQUE
Function	CREATE CONSTRAINT ON (fu:Function) ASSERT fu.func_id IS UNIQUE

Table 6: Constraints Neo4J graph

The following figure illustrates an example Neo4J graph:
- dark blue nodes represent commits
- orange nodes represent issues
- light blue nodes represent files
- green nodes represent classes
- red nodes represent functions
- in the first commit, one of the files had 3 functions, in the next commit, that file changed to 1 function + 1 class
- the second commit closes an open issue

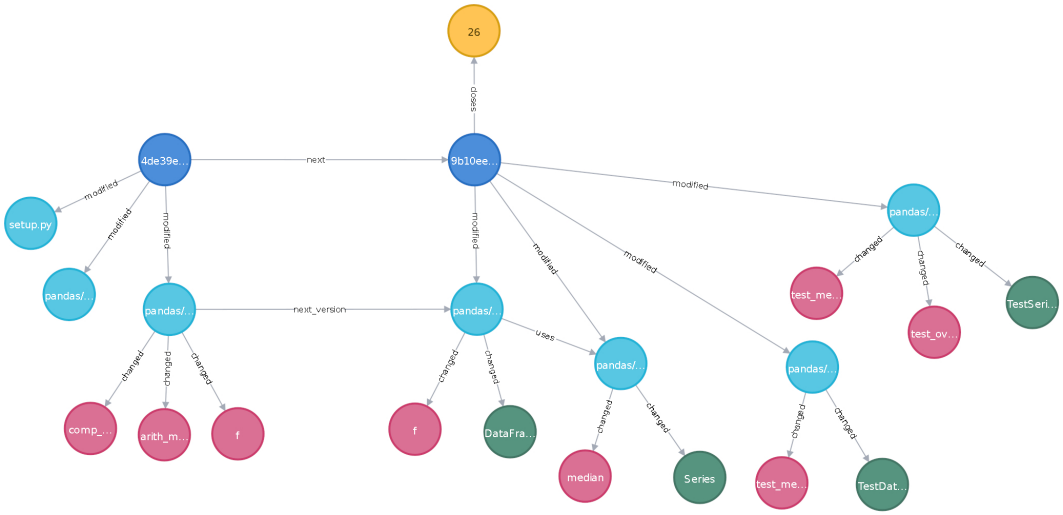

Figure 16: Example Neo4J graph

4. The main execution script is responsible for the Neo4J graph generation, SQLite database interaction, state initialization/save.

```
1  class BugFinder(object):
2      def __init__(self, config_state_file, keywords_file):
3          #...
4      # process input db and create neo4j graph
5      def do(self, until):
6          #...
7      # save in neo4j the file nodes and connections commit-file,
           file-file(new_version)
8      # and connections file-file(uses)
9      def saveFilesCommitNeo4j(self, commit):
```

```
10          #...
11      # save in neo4j the issue nodes and connections commit-issue
12      def saveBugsNeo4j(self, commit):
13          #...
14      # returns a list of [src_sha, dst_sha] items
15      def getConnectionsFile(self, f, contents):
16          #...
17      # returns the list of issue number/commit sha from the
            string str
18      def filterIssuesAndCommits(self, str):
19          #...
20      # returns true if the string str contains keyword related to
             bug
21      def isIssueBug(self, str):
22          #...
23      # returns true if the issue/ issue comments contains
            keywords related to bug
24      def isBugSecurityRelated(self, str):
25          #...
26      def saveCommitBugNeo4j(self, bug_item):
27          #...
28      def isNumber(self, str):
29          #...
30      def convertJavaImportToPath(self, str):
31          #...
32      def convertCIncludeToPath(self, str_include, str_c_file):
33          #...
34      def convertPYImportToPath(self, str_import, str_from,
            str_py_file):
35          #...
36      def convertCSImportToPath(self, str):
37          #...
38      # save for each file node the classes/functions that it
            modified
39      def saveFileInfoNeo4j(self, f_id, f_type, status, content,
            patch, timestamp):
40          #...
41      # returns the requested field from json file
42      def getJSONField(self, field, file):
43          #...
44      # modifies requested field in the json config file
45      def updateConfigStateField(self, field, value):
46          #...
47      # returns true if a string contains the word
48      def containsWord(self, str, word):
49          #...
50      # returns filtered items of table from the database
51      def getFilteredItemsDB(self, table, filter_op, field_name,
            field_value, order_col=None):
```

```
52          #...
53      # returns all items of table
54      def getItemsDB(self, table, order_col=None):
55          #...
56      # returns the requested item from table
57      def getItemDB(self, table, itemType, item):
58          #...
59      # returns the requested item that verifies the condition
60      def getItemMultipleCondsDB(self, table, field_names,
            field_values):
61          #...
62      def getNodeNeo4j(self, node):
63          #...
64      def saveNodeNeo4j(self, node):
65          #...
66      def saveConnectionNeo4j(self, src, dest, data):
67          #...
68      def saveConnectionWithParamNeo4j(self, src, dest, data):
69          #...
70      def updateNodeNeo4j(self, node):
71          #...
72      def updateConnectionNeo4j(self, src, dest, data):
73          #...
74      def existsConnectionNeo4j(self, src, dest, data):
75          #...
76      def existsNodeNeo4j(self, node):
77          #...
78      # closes connection to sqlite/neo4j db
79      def closeDBs(self):
80          #...
```

Listing 19: BugFinder module - main execution script

The first step is initialization through *__init__* function, which reads the configuration files for Neo4j/SQLite, the previous state, and the keywords file defined previously in chapter 5.3 point 1(c).

Next step is the Neo4J graph generation (evolution history of the repository), which is done by iterating through all the commits, creating the various nodes (*Commit, Issue, File, Class, Function*) and the connections between them.

In order to detect Issue items, certain patterns are searched, with the help of the keywords file defined in chapter 5.3 point 1(c). Most repositories solve bugs by using the issue number (e.g *fixes #123*), but there are repositories

(linux) that solve bugs by committing and writing in the body message *fixes <sha commit>*.

```
1 [keyword] [issue] [pr] [number] [:] [/] [#] [number] [,] [number
    ]
2 [keyword] [commit] [sha] [:] [/] [#] [partial_sha]
```

Listing 20: BugFinder module - search patterns

The Class and Function elements are specific to programming languages (e.g. C only has functions, Java has both classes and functions, etc.). These elements are created by using the specific tokenizer (*PythonTokenizer* for Python files, *CTokenizer* for C files, etc.).

Finally, the connection between the various nodes, as defined in table 5 is done as follows:

- *next* (connection between two *Commit* nodes): by looking at the timestamp of the commits
- *closes* (connection between a *Commit* and an *Issue*): is done by looking if the Issue is closed and the Commit date is after the Issue creation date (this is done to avoid false positives)
- *modified* (connection between a *Commit* and a *File*): by listing all the files of the Commit in the SQLite database
- *changed* (connection between a *File* and a *Class/Function*): after File content is passed through the language specific tokenizer
- *uses* (connection between two *File* nodes, e.g includes for .c File, imports for .java File, etc.): after the content of the source File is passed trough the language specific regular expression

Example detection of module imports for Python language:
In Python language, a module is a file containing definitions and statements. The file name is the module name with the suffix .py appended.

```
1  ''''''
2  # template imports Python
3    import <package name>.[<subpackage name>].<module name>[,<
         package name>.  [<subpackage name>].<module name> ...]
4  OR
5    from .[.]<package name>.<subpackage name>.<module_name> import
         <name>[, <name> ...]
6  OR
7    from .[.]<package name>.<subpackage name>.<module_name> import
         ( <name>[, <name> ...]
8  OR
9    from .[.]<package name>.<subpackage name> import <module name
         >[, <module_name> ...]
10
11 # module name extraction
12   group 0: import a,b,c -> a,b,c
13   group 1: from x import m,n,p -> x
14   group 2: from x import m,n,p -> m,n,p
15 ''''''
16 # regex implementation
17 regex_includes = "import[\s\t]+((?:(?:(?:[\w-]+\.)*[\w-]+),[\s\t
       ]*)*(?:(?:[\w-]+\.)*[\w-]+))|from[\s\t]+((?:[\.]*(?:[\w-]+\.)
       *[\w-]*))[\s\t]+import[\s\t\n]+(?:[\*]|[(]|((?:(?:(?:[\w
       -]+\.)*[\w-]+),[\s\t\n]*)*(?:(?:[\w-]+\.)*[\w-]+)))"
```

Listing 21: BugFinder module - Python imports detection

5.4 Classifier module

This module deals with the bug prediction model, severity model, algorithms for bug location and security. It is located in the same folder as the previous module defined in chapter 5.3, and uses the external libraries *pandas, numpy, sklearn, matplotlib*, which were installed by using the following commands:

```
pip install pandas
pip install numpy
pip install sklearn
pip install matplotlib
```

Listing 22: BugFinder external libraries

The composing blocks of the module are:

1. file Neo4J query
2. main execution script Classifier.py

1. This text file contains various cypher query that can be run on the Neo4J database generated in chapter 5.3.

The first cypher query is related to bug location, and works by identifying all the possible candidates that introduced the bug.

```
Step 1: get all commits that solved bug issues:
for each commit
  get the files
  for each file get the classes, functions
    update list [file, class, function]
save list as csv

neo4j query:
''''''
// get info from commits that solve bug issues
MATCH (c:Commit)-[:closes]->(i:Issue)
WHERE i.issue_type="bug"
WITH i, c
// get the files of commits
MATCH (c)-[r]-(f:File)
WITH i, c, f
// get the classes/functions of files
OPTIONAL MATCH (f)-[:changed]-(OtherNodes)
RETURN
```

```
20 c.sha AS commitSHA, c.timestamp AS TimeStamp, f.name AS filename
       , labels(OtherNodes) AS nodeType, OtherNodes.name AS nodeName
21 ''''''
22
23 Step 2: search all commits (with date prior to issue creation
       date) that modified one of the files from step 1, then save
       list [commit, timestamp, file, class, function] as csv
24
25 neo4j query:
26 ''''''
27 //search all commits that might have introduced the bug
28 // get commits that solve bug issues
29 MATCH (c:Commit)-[:closes]->(i:Issue)
30 WHERE i.issue_type="bug"
31 WITH i, c
32 // get the files of commits
33 MATCH (c)-[r]-(f:File)
34 WITH i, c, f
35 // get the commits(and files) which have at least one of files
       and date < issue creation date
36 MATCH (c2:Commit)-[r]-(f2:File {name: f.name})
37 WHERE c2.timestamp < i.created_at and c2.sha <> c.sha
38 // get the classes/functions of files
39 WITH i, c, f, c2, f2
40 OPTIONAL MATCH (f2)-[:changed]-(OtherNodes)
41 RETURN DISTINCT
42 c2.sha AS commitSHA, c2.timestamp AS TimeStamp, f2.name AS
       filename, labels(OtherNodes) AS nodeType, OtherNodes.name AS
       nodeName
43 ''''''
```

Listing 23: Classifier module - bug location algorithm

The second cypher query is related to issue type (bug or enhancement) prediction, issue severity. This query gathers information about the issue, number of comments/people/events, labels, etc. which are then saved as CSV file.

```
1 // get all commits that solve issues
2 MATCH (c:Commit)-[:closes]->(i:Issue)
3 WITH c, i
4
5 // get the files of commits
6 MATCH (c)-[r]-(f:File)
7 WITH c, i, f, r
8
9 // get the number of "uses" connections for files
```

```
10 OPTIONAL MATCH (f)-[r2:uses]->(f2:File)
11 WITH c, i, f, r, count(r2) as TmpNumberUses
12
13 // get the classes/functions of files
14 OPTIONAL MATCH (f)-[:changed]->(OtherNodes)
15 WITH c, i, f, r, count(OtherNodes.class_id) AS NbChangedClasses,
       count(OtherNodes.func_id) AS NbChangedFunctions,
     TmpNumberUses
16 RETURN
17 c.sha AS CommitSHA, c.timestamp AS TimeStampClose, i.created_at
     AS TimeStampCreate, i.issue_id AS IssueNumber, i.issue_type
     AS IssueType, i.number_comments AS NumberComments, i.
     number_people AS NumberPeople, i.number_events AS
     NumberEvents, i.labels AS Labels, size(i.labels) AS
     NumberLabels, count(f) AS NumberFiles, sum(f.lines_added) AS
     LinesAdded, sum(f.lines_removed) AS LinesRemoved, sum(f.
     number_lines_code) AS NumberLinesCode, sum(f.number_classes)
     AS NumberClasses, sum(f.number_functions) AS NumberFunctions,
      sum(NbChangedClasses) AS NumberChangedClasses, sum(
     NbChangedFunctions) AS NumberChangedFunctions, sum(
     TmpNumberUses) AS NbCohesion
```

Listing 24: Classifier module - Neo4J query issue type, severity

The last cypher query is related to security impact of the issues. This is done by retrieving all the Issue nodes that were flagged as security related in the graph. This flag operation was done by using pattern matching for certain security related words in chapter 5.3.

```
1 MATCH (c:Commit)-[:closes]->(i:Issue {issue_type: 'bug',
     issue_security: true})
2 RETURN
3 c.sha AS CommitSHA, c.timestamp AS TimeStamp, i.issue_id AS
     IssueNumber, i.issue_type AS IssueType, i.number_comments AS
      NumberComments, i.number_people AS NumberPeople, i.
     number_events AS NumberEvents, i.labels AS Labels
```

Listing 25: Classifier module - Neo4J query security

The last cypher query is related to security impact of the issues. This is done by retrieving all the Issue nodes that were flagged as security related in the graph. This flag operation was done by using pattern matching for certain security related words in chapter 5.3.

```
MATCH (c:Commit)-[:closes]->(i:Issue {issue_type: 'bug',
    issue_security: true})
RETURN
c.sha AS CommitSHA, c.timestamp AS TimeStamp, i.issue_id AS
    IssueNumber, i.issue_type AS IssueType,  i.number_comments AS
     NumberComments, i.number_people AS NumberPeople, i.
    number_events AS NumberEvents, i.labels AS Labels
```

Listing 26: Classifier module - bug location algorithm

2. This script uses as input CSV files containing datasets and generates metrics about the issue, and various graphs.

```
# Classifier.py
def doIssueTypeModel(dataframe):
    #...
def doSeverityModel(dataframe):
    #...
def getNumberDaysInterval(df_col1, df_col2):
    #...
def createDataframe(csv_name, repo_number):
    #...
def concatenateDataframes(dataframe1, dataframe2):
    #...
```

Listing 27: Classifier module - execution script

The first step is the creation of the dataframe, which is done by using the pandas library function that reads CSV files. In case the dataset CSV file contains too few records, it can be merged with another dataset.

```
def createDataframe(csv_name, repo_number):
    path_scrape = "scraped_data"
    csv_all = path_scrape+"/"+csv_name
    # read the csv file and create the dataframe
    pd_all = pd.read_csv(csv_all)
    # add the column for repository number
    pd_all["repo_number"] = repo_number
    return pd_all
def concatenateDataframes(dataframe1, dataframe2):
    return pd.concat([dataframe1, dataframe2])
```

Listing 28: Classifier module - dataframe creation

The next step is the implementation of the function for Issue type model, which receives as input a dataframe.

Issue type Model: The model consists in defining two classes (bug: 0, enhancement: 1), and several features: NumberFiles, NumberLinesCode, NumberComments, etc.

Then, the dataframe is 'cleaned' by deleting all columns containing useless information, the numerical columns are normalized. The normalization is required because of outliers (some issues have no comments, other have 100; some files have few lines changed, others have 1000 lines changed).

Next, the dataframe is split (20% for testing and 80% for training) and passed to the machine learning algorithm (random forest classifier). After, some parameter tuning (number of estimators, class weight), the forest of trees is built, and the model is applied to the test dataset.

Finally, the metrics (dimension dataset, accuracy score) are extracted, and other graphs (histogram, scatter plot matrix, ROC curve) are generated.

```
1  def doIssueTypeModel(dataframe):
2      #...
3      cols_to_remove = ["TimeStampClose", "TimeStampCreate", "
           CommitSHA", "Labels"]
4      # delete unnecessary columns
5      # in order to clean dataframe
6      del pd_all_grouped[cols_to_remove[0]]
7      del pd_all_grouped[cols_to_remove[1]]
8      del pd_all_grouped[cols_to_remove[2]]
9      del pd_all_grouped[cols_to_remove[3]]
10
11     #...
12     # normalization of the columns
13     for col_f in columns_features:
14         normalized_col = preprocessing.normalize([np.array(
               pd_all_grouped[col_f])])
15         pd_all_grouped[col_f] = normalized_col[0]
16
17     # replace bug with 0 and enhancement with 1 in dataframes
18     mapping_issue_type = {'bug': 0, 'enhancement': 1}
19
20     #...
21     # create the test an training datasets
```

```
22      # 0.2 is 20% of dataset for testing, the remaining 80% is
            for training
23      # shuffle so that the model is not biased towards bugs or
            enhancements
24      X_train, X_test, Y_train, Y_test = train_test_split(
            tmp_pd_all_grouped, targets_type, test_size=0.2, shuffle=
            True, stratify=targets_type)
25
26      #...
27      # 30 is the standard number in statistics used for
            meaningful sample size
28      number_estimators = 30
29      # apply random forest algorithm
30      # random_state is a seed for the number generator
31      clf = RandomForestClassifier(number_estimators, random_state
            =0)
32      # build the forest of trees
33      clf.fit(X_train, Y_train)
34      # Note: Y_test will be used to confront final results
35      # applies the model to the test dataset
36      Y_predicted = clf.predict(X_test)
37
38      print("Dimension dataset: "+str(X_test.shape[0]+X_train.
            shape[0]))
39      print("Accuracy score: "+str(metrics.accuracy_score(Y_test,
            Y_predicted)))
40
41      # see distribution of features
42      printHist = True
43      if printHist:
44          pd_all_grouped.hist()
45          plt.show()
46
47      # see distributed of one feature based on another
48      printScatterMatrix = False
49      if printScatterMatrix:
50          # plot Scatterplot Matrix
51          # horizontal = variable considered, vertical =
                comparison with other variables
52          scatter_matrix(pd_all_grouped)
53          plt.show()
54
55      # ROC: curve
56      # shows true positive rate and false positive rate
57      # goal: get as close as possible to green curve
58      printROC = False
59      if printROC:
60          Y_proba=clf.predict_proba(X_test)
```

```
61          fpr_RF, tpr_RF, thresholds_RF = roc_curve(Y_test,
                Y_proba[:,0], pos_label=0)
62
63          # actual classification roc curve
64          plt.plot(fpr_RF, tpr_RF, 'r-',label = 'RF')
65          # worst case classification roc curve
66          plt.plot([0,1],[0,1], 'k-',label='random')
67          # perfect classification roc curve
68          plt.plot([0,0,1,1],[0,1,1,1], 'g-',label='perfect')
69          plt.xlabel('False Positive Rate')
70          plt.ylabel('True Positive Rate')
71          plt.show()
```

Listing 29: BugFinder module - issue type model function

The accuracy score from the sklearn Python library has the following internal implementation:

$$\texttt{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y}_i = y_i) \tag{9}$$

The last step is the implementation of the function for severity Issue model, which receives as input a dataframe.

Severity Model: The model consists in defining two classes (0: low severity, 1: high severity), and several features: NumberFiles, NumberEvents, NumberPeople, NumberComments, etc.

These two values for low severity and high severity are computed as follows:

- each issue has a label with value: "easy", "intermediate", "advanced", "expert", "low", "medium", "high"

- the following mapping is assigned: "easy":0, "intermediate":1, "advanced":2, "expert":2, "low":0, "medium":1, "high":2

- if label of issue has value <= 1 then severity=0 else severity=1

Next, the dataframe is 'cleaned' by deleting all columns containing useless information, the numerical columns are normalized. The normalization is required because of outliers (some issues have no comments, other have 100; some files have few lines changed, others have 1000 lines changed).

Also, other meaningful features are added to the dataframe, such as time interval for issue create/close.

Next, the dataframe is split (20% for testing and 80% for training) and passed to the machine learning algorithm (random forest classifier). After, some parameter tuning (number of estimators, class weight), the forest of trees is built, and the model is applied to the test dataset.

Finally, the metrics (dimension dataset, accuracy score) are extracted, and other graphs (histogram, scatter plot matrix, ROC curve) are generated.

```
1  def doSeverityModel(dataframe):
2      #...
3      # keeps track of how much time passed from issue create and
           close
4      timeintervals = []
5      for issue in pd_all["RepoNbIssueNb"]:
6     #...
7              tf = tmp_df["TimeStampClose"].max()
8              tc = np.array(tmp_df["TimeStampCreate"])[0]
9              timeintervals.append((tf-tc).days)
10     pd_all["TimeInterval"] = timeintervals
11
12     #...
13     cols_to_remove = ["TimeStampClose", "TimeStampCreate", "
           CommitSHA"]
14     # delete unnecessary columns
15     # in order to clean dataframe
16     del pd_all_grouped[cols_to_remove[0]]
17     del pd_all_grouped[cols_to_remove[1]]
18     del pd_all_grouped[cols_to_remove[2]]
19
20     #...
21     # normalize the columns of dataset
22     columns_features = list(pd_all_grouped.columns)
23     for col_f in columns_features:
24         normalized_col = preprocessing.normalize([np.array(
               pd_all_grouped[col_f])])
25         pd_all_grouped[col_f] = normalized_col[0]
26
27     #...
28     # create the test an training datasets
29     # 0.2 is 20% of dataset for testing, the remaining 80% is
           for training
```

```
30      X_train, X_test, Y_train, Y_test = train_test_split(
            tmp_pd_all_grouped, targets_type, test_size=0.2, shuffle=
            True, stratify=targets_type)
31
32      #...
33      # 30 is the standard number in statistics used for
            meaningful sample size
34      number_estimators = 30
35      # apply random forest algorithm
36      # random_state is a seed for the number generator
37      clf = RandomForestClassifier(number_estimators, random_state
            =0, class_weight='balanced_subsample')
38      # build the forest of trees
39      clf.fit(X_train, Y_train)
40      # Note: Y_test will be used to confront final results
41      # applies the model to the test dataset
42      Y_predicted = clf.predict(X_test)
43
44      print("Dimension dataset: "+str(X_test.shape[0]+X_train.
            shape[0]))
45      print("Accuracy score: "+str(metrics.accuracy_score(Y_test,
            Y_predicted)))
46
47      # see distribution of features
48      printHist = True
49      if printHist:
50          pd_all_grouped.hist()
51          plt.show()
52
53      # see distributed of one feature based on another
54      printScatterMatrix = False
55      if printScatterMatrix:
56          # plot Scatterplot Matrix
57          # horizontal = variable considered, vertical =
                comparison with other variables
58          scatter_matrix(pd_all_grouped)
59          plt.show()
60
61      # ROC: curve
62      # shows true positive rate and false positive rate
63      # goal: get as close as possible to green curve
64      printROC = False
65      if printROC:
66          Y_proba=clf.predict_proba(X_test)
67          fpr_RF, tpr_RF, thresholds_RF = roc_curve(Y_test,
                Y_proba[:,0], pos_label=0)
68
69          # actual classification roc curve
70          plt.plot(fpr_RF, tpr_RF, 'r-',label = 'RF')
```

```
71          # worst case classification roc curve
72          plt.plot([0,1],[0,1],'k--',label='random')
73          # perfect classification roc curve
74          plt.plot([0,0,1,1],[0,1,1,1],'g-',label='perfect')
75          plt.xlabel('False Positive Rate')
76          plt.ylabel('True Positive Rate')
77          plt.show()
```

Listing 30: BugFinder module - severity issue model function

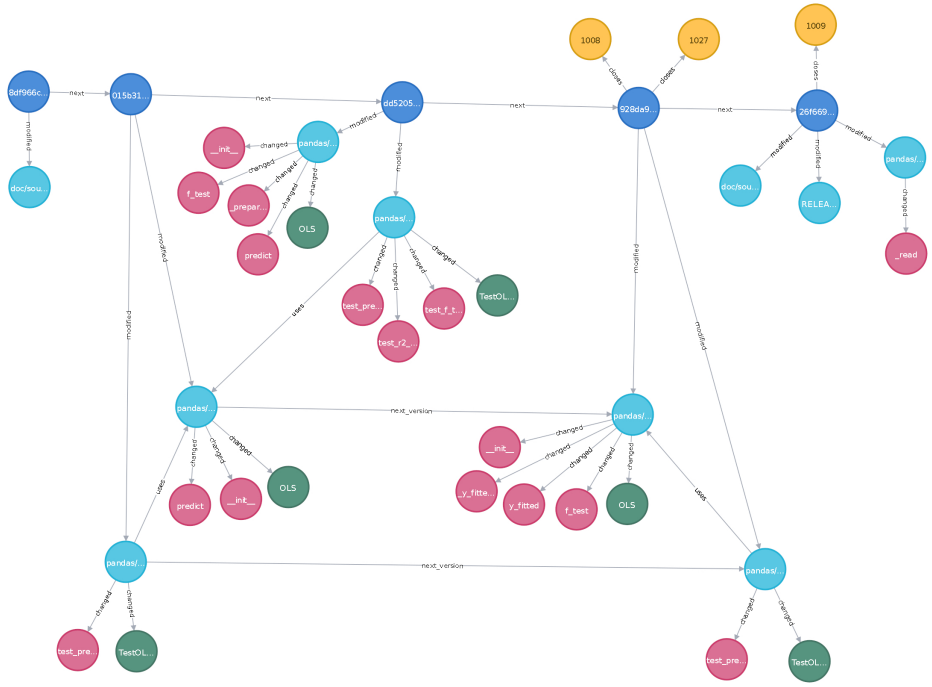

Figure 17: Another example Neo4J graph

6 Results

6.1 Research Questions:

RQ1: Which repositories were used to generated the datasets?

For the datasets the following repositories were chosen: *pandas, sklearn, numpy*. The reason for choosing those repositories is because they share similar features (all are written in python/c code, have same standard for tag labeling of issues, similar project structure)

RQ2: How does the issue type model apply to other repositories?

The issue type model works with all repositories, as long as they use the issue opening/closing feature present in GitHub.

RQ3: How does the issue severity model apply to other repositories?

The issue severity model currently only works with repositories that use certain values for the issue labels feature present in GitHub.

RQ4: Which metrics are used for issue type model?

The metrics used are dataset dimension and accuracy.

RQ5: Which metrics are used for issue severity model?

The metrics used are dataset dimension and accuracy.

Note: For cleaner graph representation, the following abbreviations will be used for dataset features:
IssueType:IT, NumberComments:Ncom, NumberPeople:NP, NumberEvents:NE, NumberLabels:NL, NumberFiles:NFi, LinesAdded:LA, LinesRemoved:LR, NumberLinesCode:NLC, NumberClasses:NC, NumberFunctions:NF, NumberChangedClasses:NCC, NumberChangedFunctions:NCF, NbCohesion:NCoh, RepoNbIssueNb:RNIN, TimeInterval:TI, NumberCommitsPerIssue:NCPI, Severity:S

6.1.1 Pandas repository

Pandas is a very popular repository in the machine learning world, that allows fast and easy data manipulation and analysis.

The following tables summarize information about the SQLite and Neo4J databases which will be used for dataset generation:

PyGitHub DB	Pandas
Database size	8.05 GB
#issues	27105
#labels	102
#issue events	295147
#issue comments	150452
#pull requests	11789
#commits	19633
#commit files	80335
#commit comments	572

Table 7: PyGitHub DB SQLite information pandas repository

Neo4J DB	Pandas
#nodes	362981
#relations	868784
#changes classes	60429
#changes functions	193972

Table 8: BugFinder DB Neo4J information pandas repository

RQ6: How does the issue type model behave?

Metrics issue	Run #1	Run #2	Run #3
Dimension dataset	8613		
Accuracy score	0.843	0.837	0.836

Table 9: Metrics issue type - pandas repository

RQ7: How are the features distributed in the dataset for issue type model?

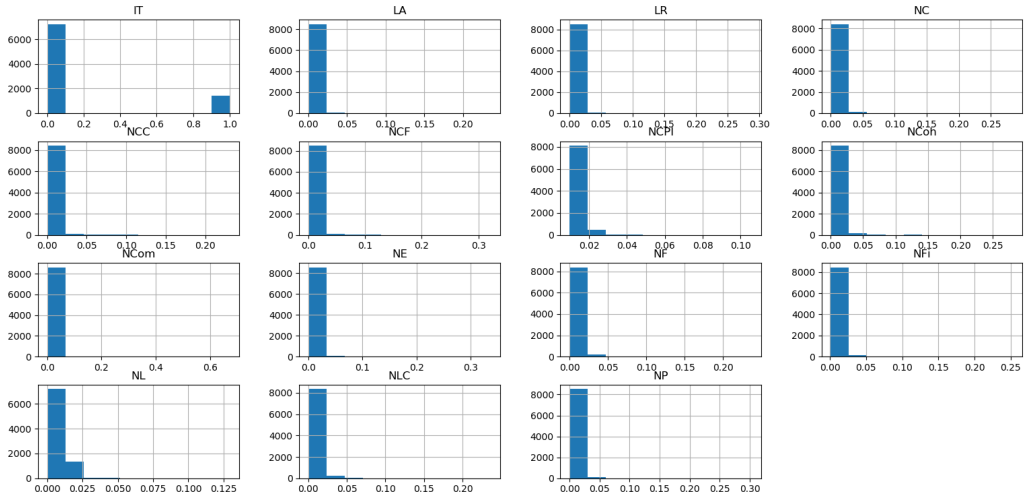

Figure 18: Issue type histogram - pandas repository

From the histogram it can be seen that there is an inequality in the distribution of the features, with small exception for NL, NCPI.

RQ8: How are the features correlated in the dataset for issue type model?

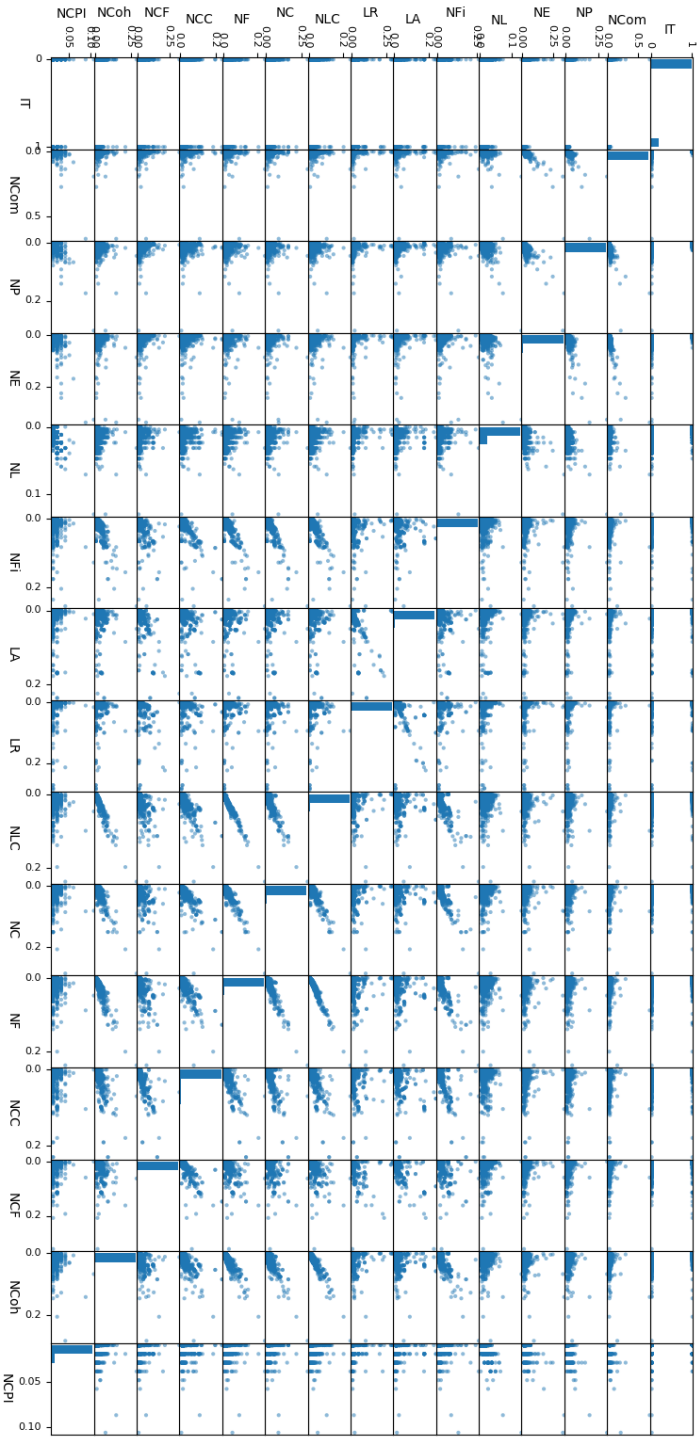

Figure 19: Issue type scatter plot matrix - pandas repository

The scatter plot matrix shows that there is a strong positive correlation

between NLC-NCoh, NLC-NF, NLC-NC, NC-NF, NF-NCoh.

RQ9: What is the behaviour of the ROC Curve for issue type model?

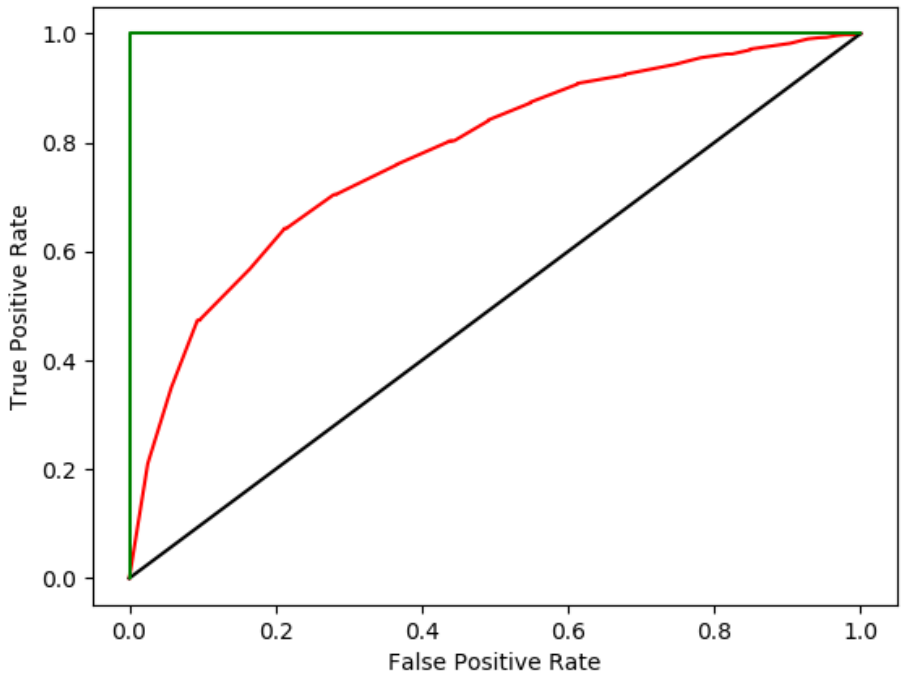

Figure 20: Issue type ROC Curve - pandas repository

RQ10: How does the issue severity model behave?

Metrics severity	Run #1	Run #2	Run #3
Dimension dataset	1330		
Accuracy score	0.875	0.902	0.879

Table 10: Metrics severity - pandas repository

RQ11: How are the features distributed in the dataset for issue severity model?

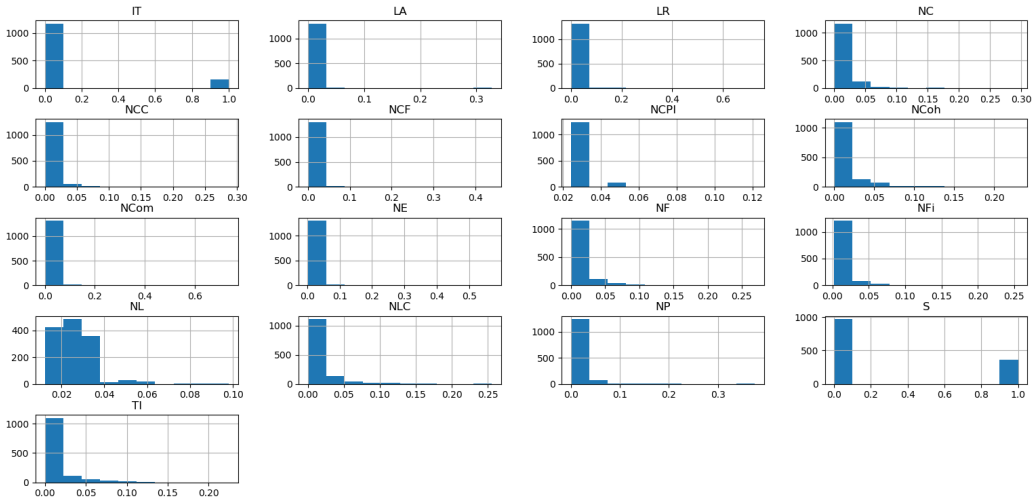

Figure 21: Severity histogram - pandas repository

From the histogram it can be seen that there is an strong distribution of the NL feature, and small distribution for TI, NLC, NF, NCoh.

RQ12: How are the features correlated in the dataset for issue severity model?

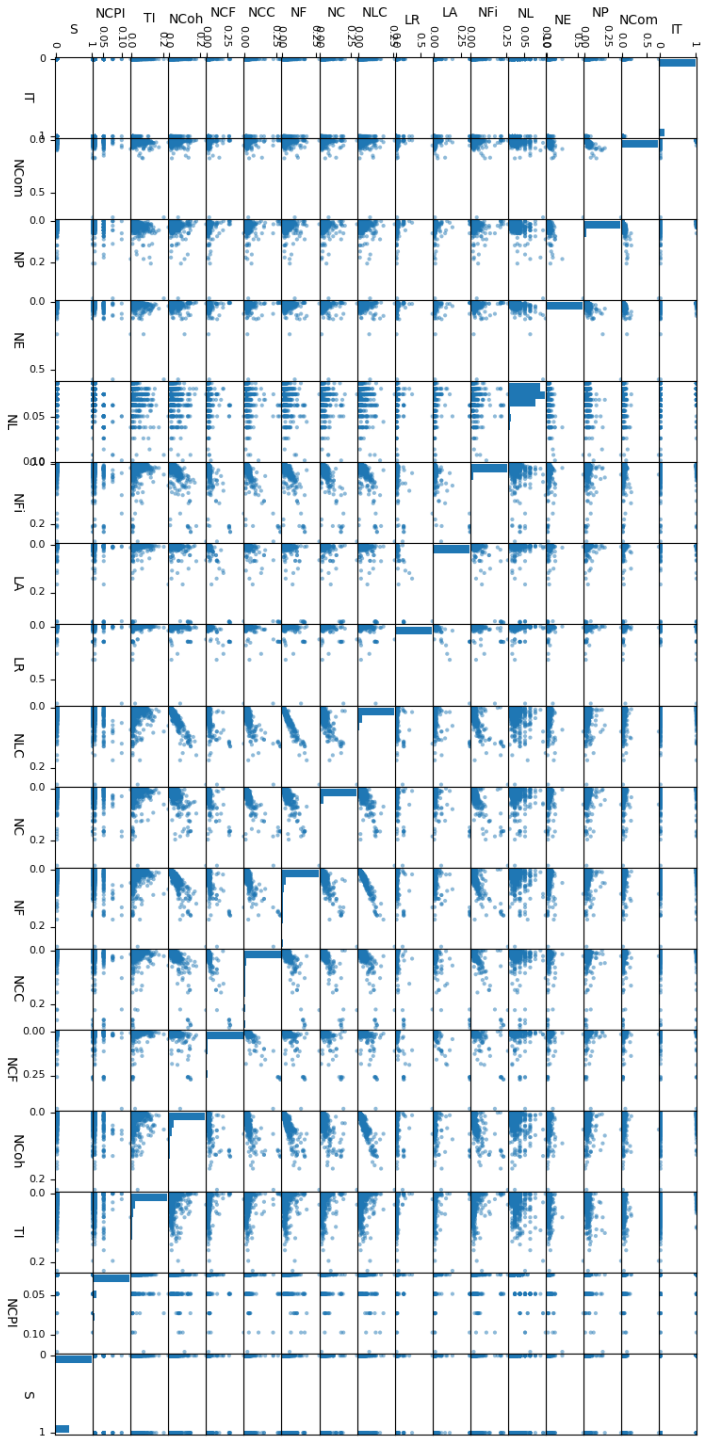

Figure 22: Severity scatter plot matrix - pandas repository

The scatter plot matrix shows that there is a strong positive correlation

between NLC-NCoh, NLC-NF, NF-NCoh.

RQ13: What is the behaviour of the ROC Curve for issue severity model?

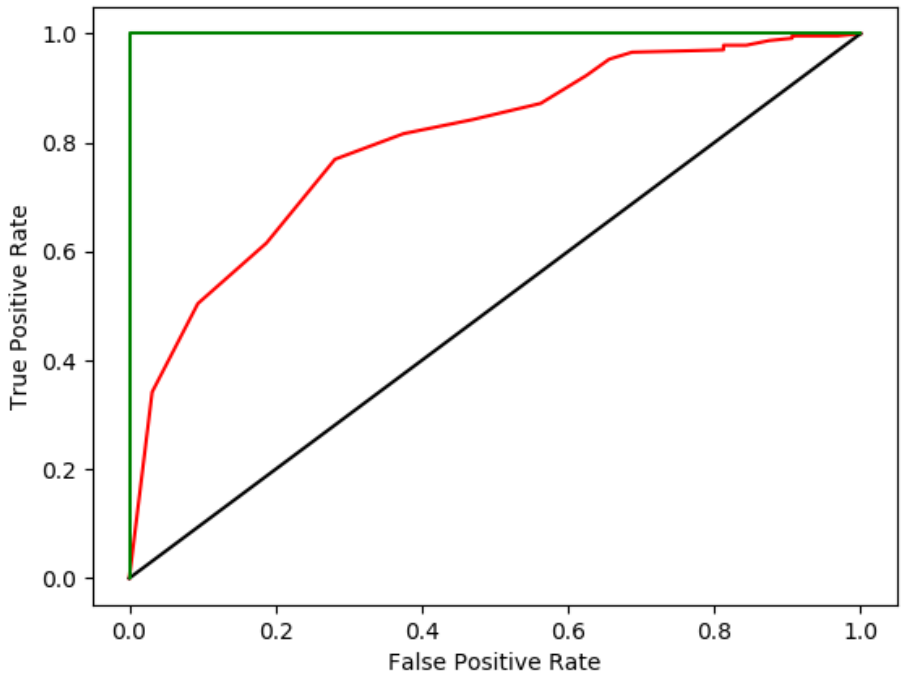

Figure 23: Severity ROC Curve - pandas repository

6.1.2 Sklearn repository

Sklearn is an open source machine learning library that is used for data mining and data analysis.

The following tables summarize information about the SQLite and Neo4J databases which will be used for dataset generation:

PyGitHub DB	Sklearn
Database size	5.63 GB
#issues	14594
#labels	26
#issue events	156692
#issue comments	121509
#pull requests	8120
#commits	24348
#commit files	97512
#commit comments	2491

Table 11: PyGitHub DB SQLite information sklearn repository

Neo4J DB	Sklearn
#nodes	296006
#relations	561335
#changes classes	41622
#changes functions	127989

Table 12: BugFinder DB Neo4J information sklearn repository

RQ6: How does the issue type model behave?

Metrics issue	Run #1	Run #2	Run #3
Dimension dataset	4531		
Accuracy score	0.919	0.918	0.919

Table 13: Metrics issue type - sklearn repository

RQ7: How are the features distributed in the dataset for issue type model?

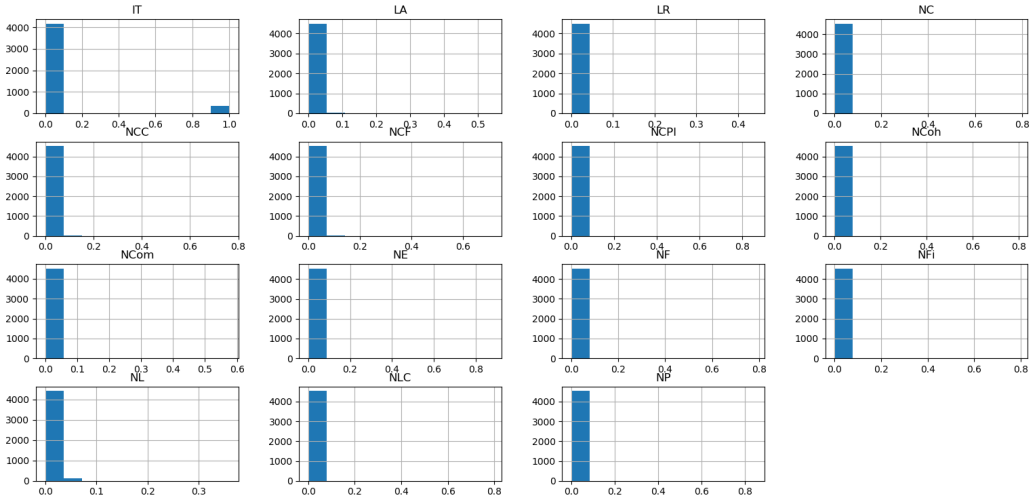

Figure 24: Issue type histogram - sklearn repository

From the histogram it can be seen that there is an inequality in the distribution of the features, with small exception for NL.

RQ8: How are the features correlated in the dataset for issue type model?

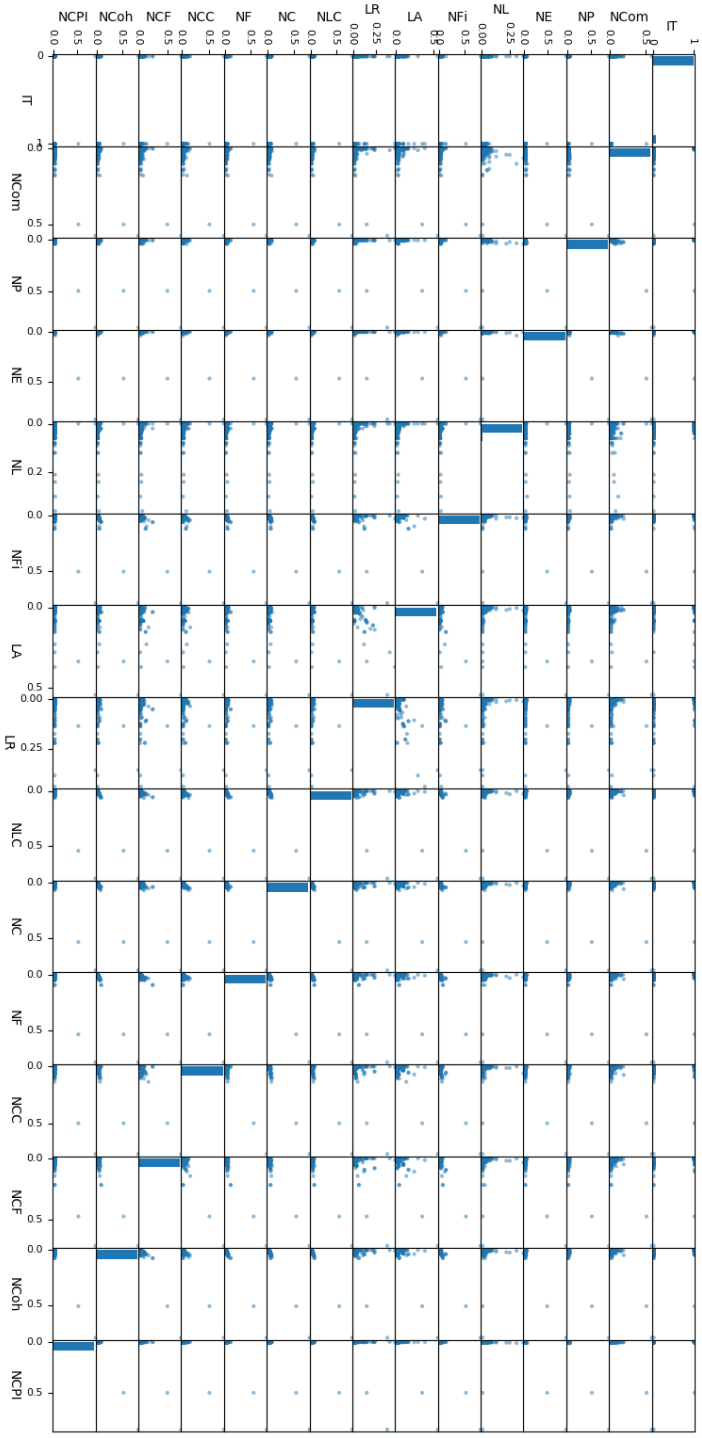

Figure 25: Issue type scatter plot matrix - sklearn repository

The scatter plot matrix shows that there is weak correlation between the

features, with small exception for LA-LR.

RQ9: What is the behaviour of the ROC Curve for issue type model?

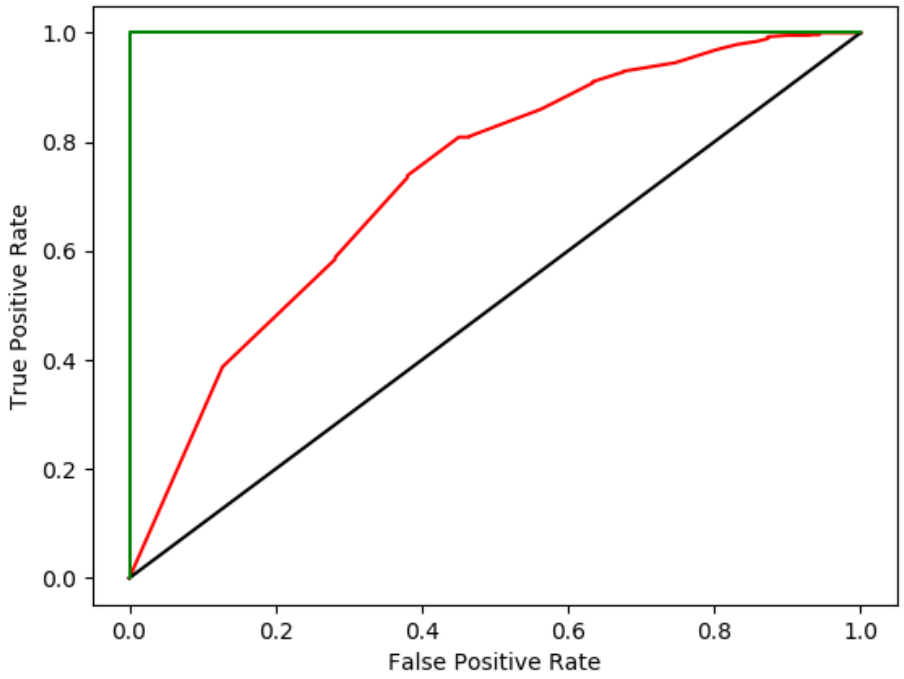

Figure 26: Issue type ROC Curve - sklearn repository

RQ10: How does the issue severity model behave?

Metrics severity	Run #1	Run #2	Run #3
Dimension dataset	523		
Accuracy score	0.838	0.838	0.828

Table 14: Metrics severity - sklearn repository

RQ11: How are the features distributed in the dataset for issue severity model?

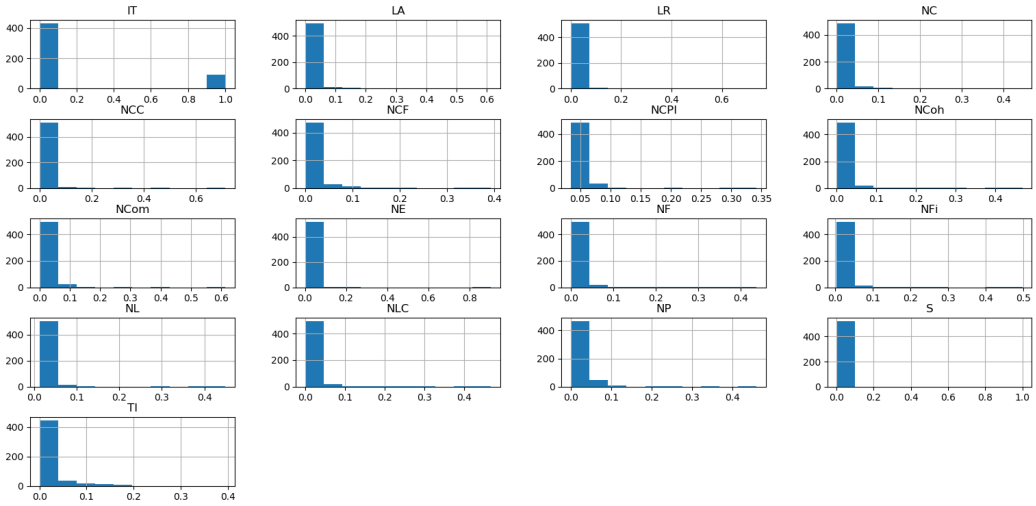

Figure 27: Severity histogram - sklearn repository

From the histogram it can be seen that there is an inequality in the distribution of the features, with small exception for TI, NCom, NP, NF, NCPI, NC.

RQ12: How are the features correlated in the dataset for issue severity model?

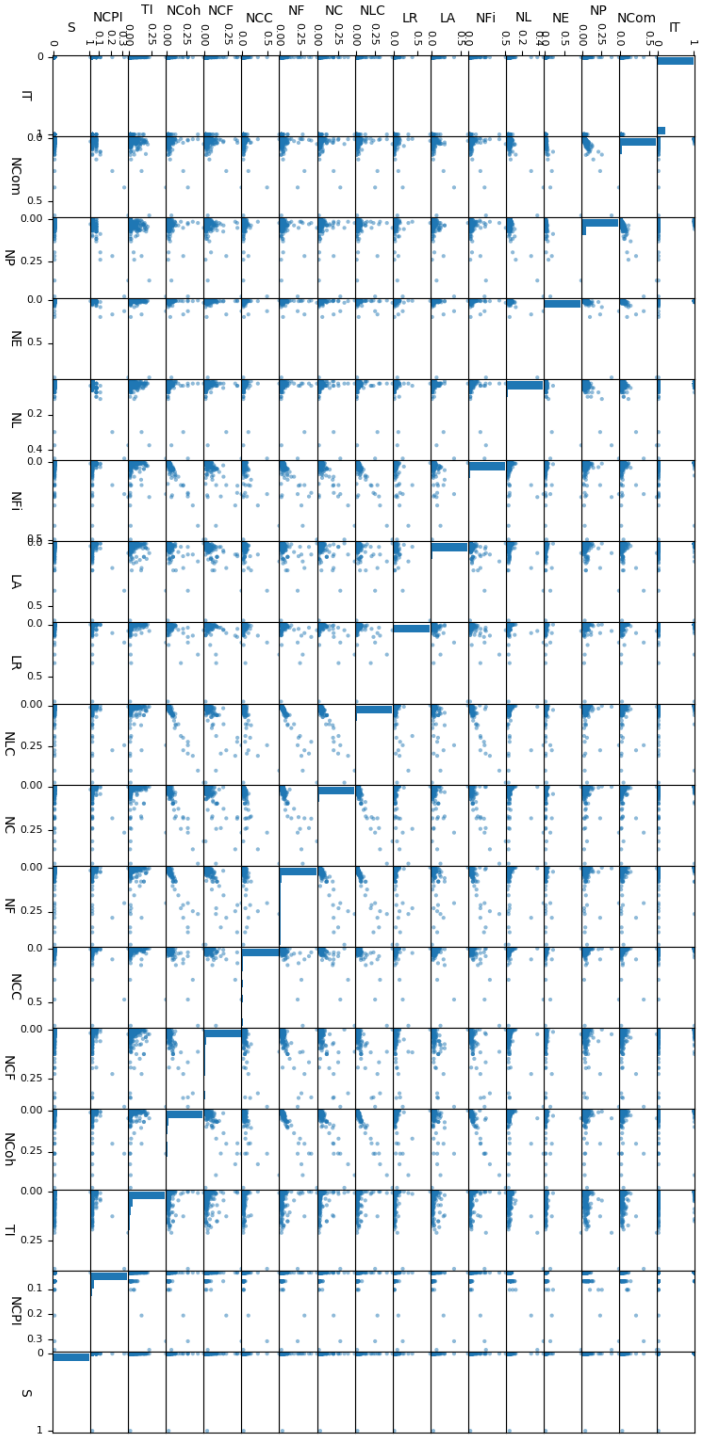

Figure 28: Severity scatter plot matrix - sklearn repository

The scatter plot matrix shows that there is a strong positive correlation

between NFi-NCoh, NLC-NCoh, NLC-NC, NF-NCoh.

RQ13: What is the behaviour of the ROC Curve for issue severity model?

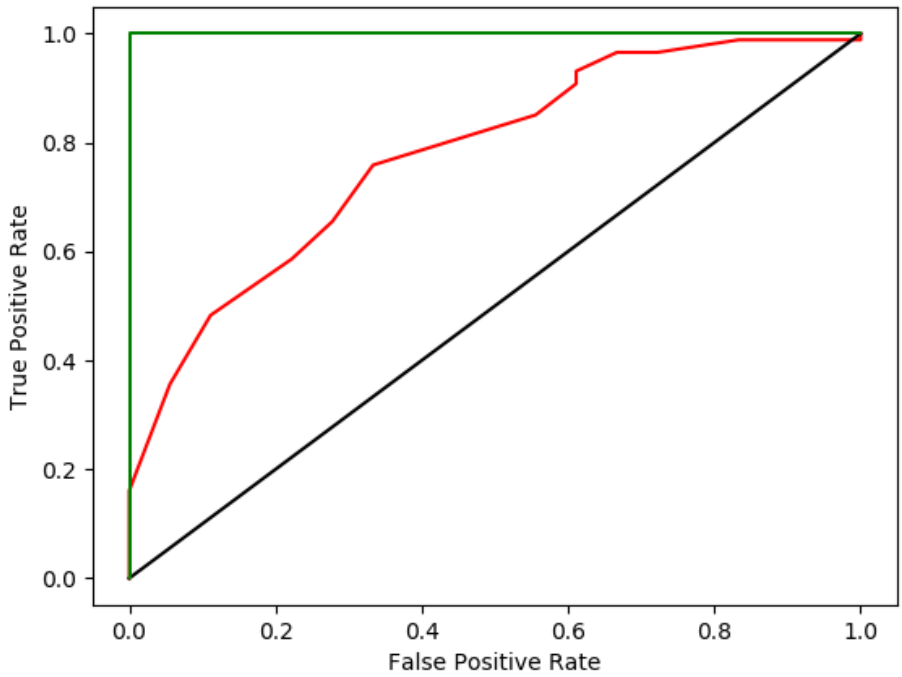

Figure 29: Severity ROC Curve - sklearn repository

6.1.3 Numpy repository

Numpy is an open source library that is used for scientific computing with the Python language.

The following tables summarize information about the SQLite and Neo4J databases which will be used for dataset generation:

PyGitHub DB	Numpy
Database size	5.22 GB
#issues	13865
#labels	77
#issue events	116378
#issue comments	74329
#pull requests	6501
#commits	20714
#commit files	66594
#commit comments	256

Table 15: PyGitHub DB SQLite information numpy repository

Neo4J DB	Numpy
#nodes	217621
#relations	398549
#changes classes	30295
#changes functions	97321

Table 16: BugFinder DB Neo4J information numpy repository

RQ6: How does the issue type model behave?

Metrics issue	Run #1	Run #2	Run #3
Dimension dataset	2697		
Accuracy score	0.883	0.890	0.890

Table 17: Metrics issue type - numpy repository

RQ7: How are the features distributed in the dataset for issue type model?

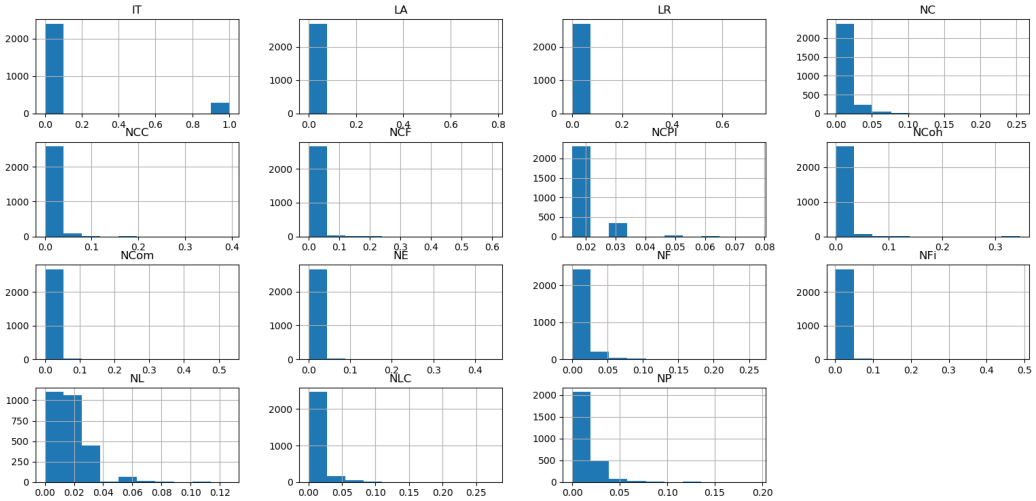

Figure 30: Issue type histogram - numpy repository

From the histogram it can be seen that there is an strong distribution of the NL feature, and small distribution for NLC, NP, NF, NCPI, NC.

RQ8: How are the features correlated in the dataset for issue type model?

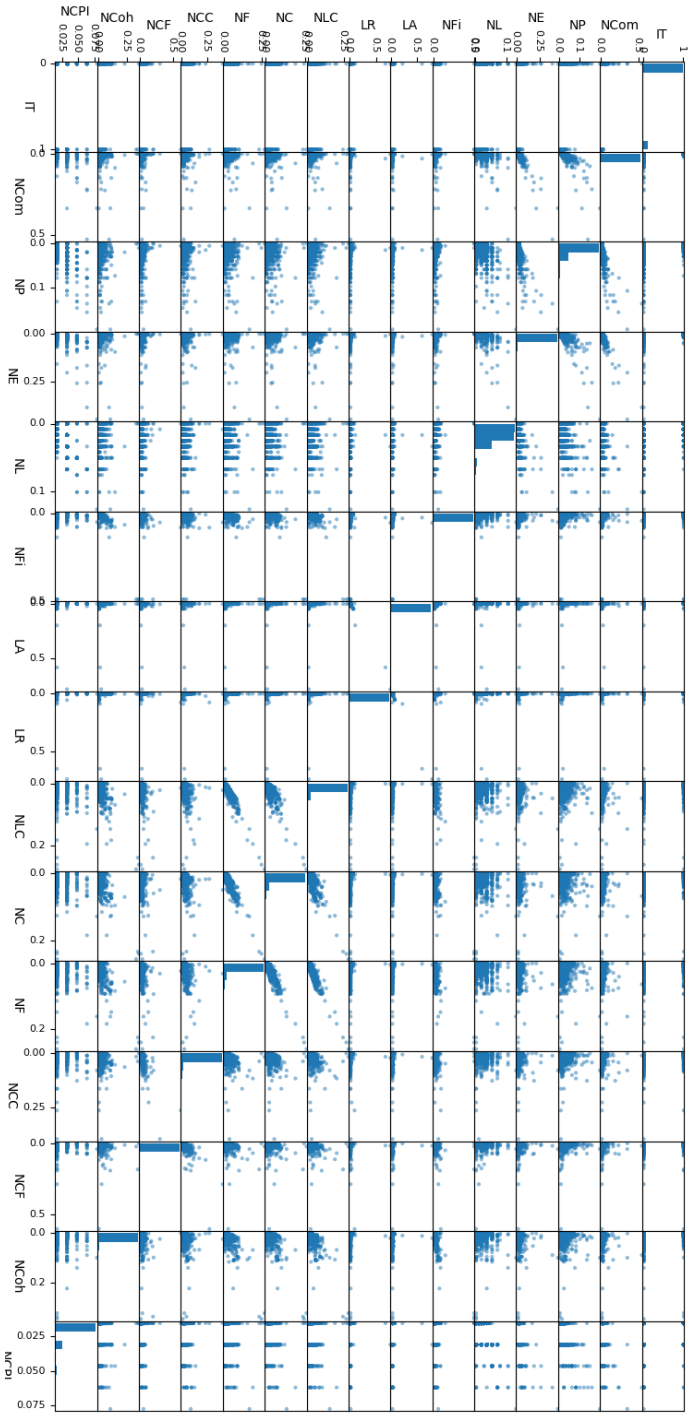

Figure 31: Issue type scatter plot matrix - numpy repository

The scatter plot matrix shows that there is a strong positive correlation

between NLC-NF, NLC-NC, NC-NF.

RQ9: What is the behaviour of the ROC Curve for issue type model?

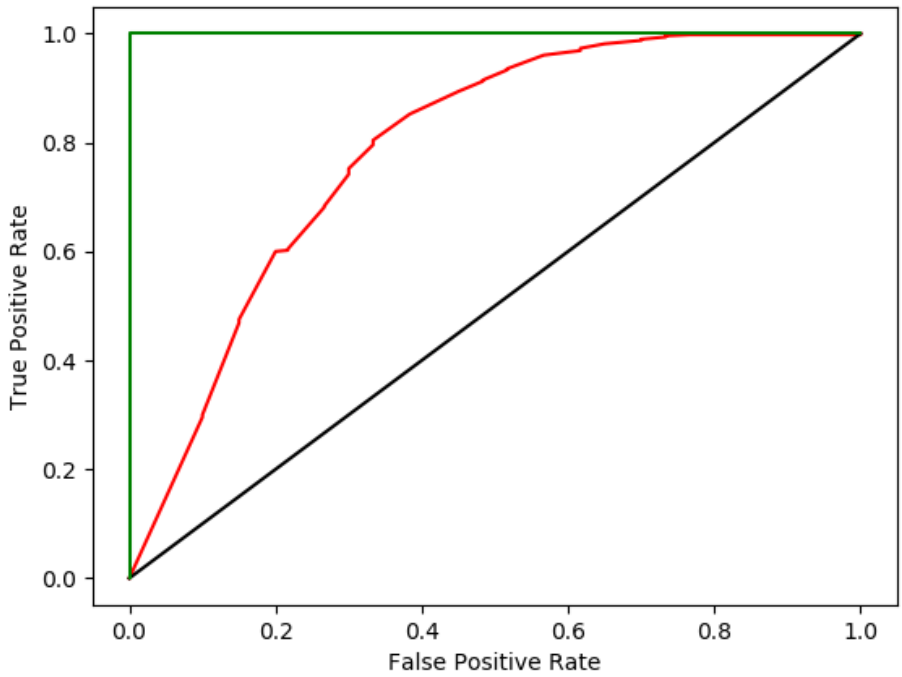

Figure 32: Issue type ROC Curve - numpy repository

RQ10: How does the issue severity model behave?

Note: for this repository the dataset for severity computation was too small (dimension dataset: 119 samples), so I had to combine the dataset with the dataset from another repository. Combining the Numpy dataset with Pandas dataset produced better results.

Metrics severity	Run #1	Run #2	Run #3
Dimension dataset	523		
Accuracy score	0.875	0.882	0.886

Table 18: Metrics severity - sklearn + numpy repository

RQ11: How are the features distributed in the dataset for issue severity model?

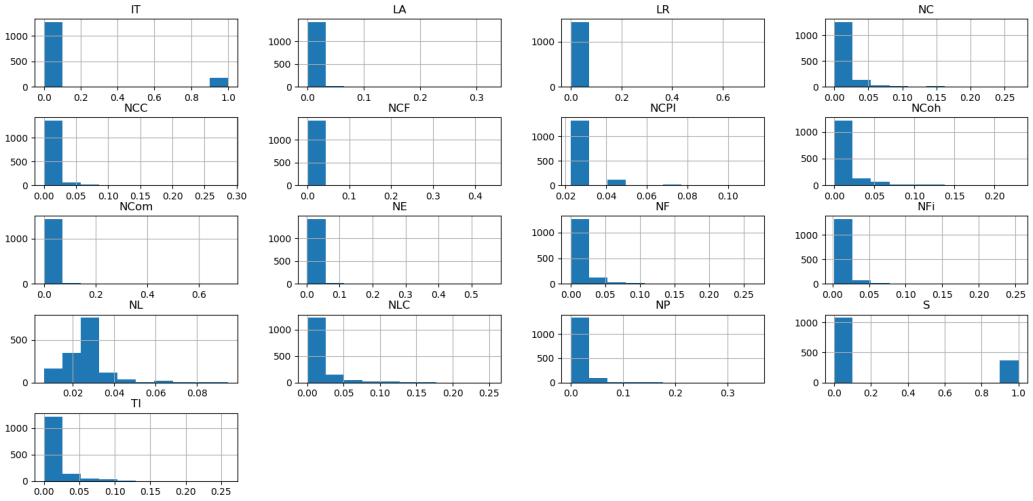

Figure 33: Severity histogram - sklearn + numpy repository

From the histogram it can be seen that there is an strong distribution of the NL feature, and small distribution for TI, NLC, NP, NF, NCPI, NFi, NCoh, NC.

RQ12: How are the features correlated in the dataset for issue severity model?

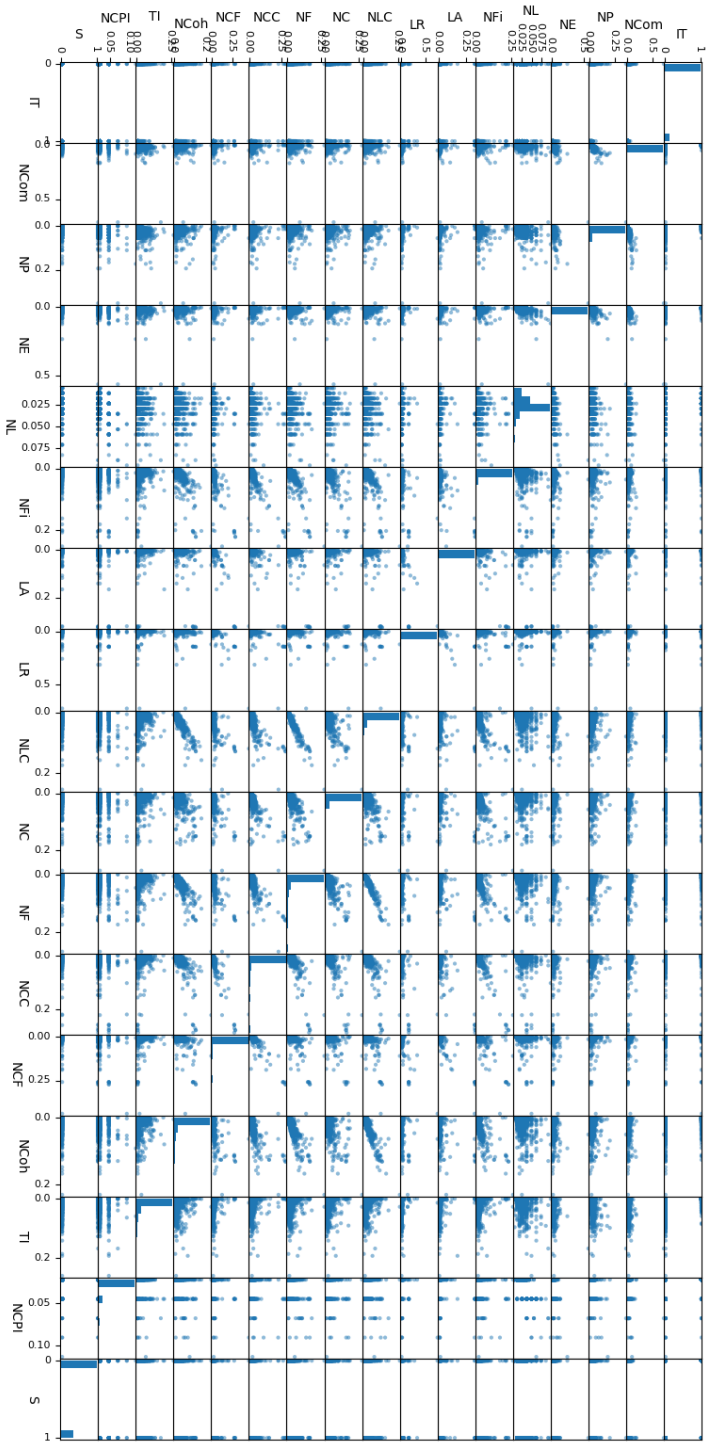

Figure 34: Severity scatter plot matrix - sklearn + numpy repository

The scatter plot matrix shows that there is a strong positive correlation between NLC-NCoh, NLC-NF, NC-NF, NF-NCoh.

RQ13: What is the behaviour of the ROC Curve for issue severity model?

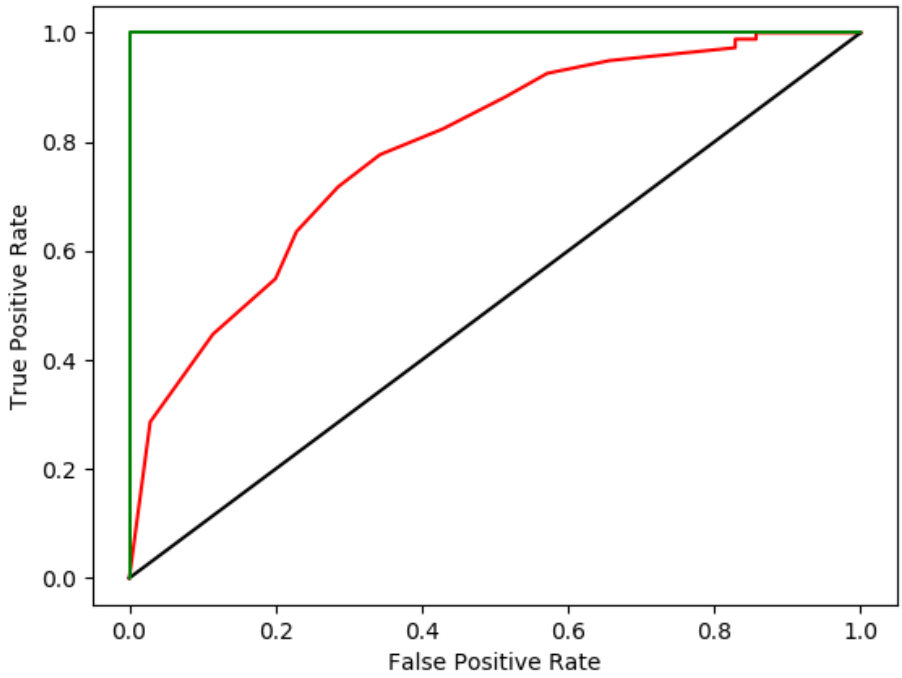

Figure 35: Severity ROC Curve - sklearn + numpy repository

7 Conclusions

7.1 Final Thoughts

This project started with the idea of analyzing topics that were treated partially in the literature, such as: bug type prediction, bug severity, security impacts, bug location.

In order to analyze those topics, I had to acquire some datasets for testing the various models. Because there was no software that could generate the datasets that suited those needs, I developed several modules written in python that dealt with dataset generation and manipulation.

Finally, I created models for the issue type prediction, severity issue and tested those models with some GitHub repositories (numpy, pandas, sklearn).

7.2 Further Improvements

This project was done with modularity in mind, so it should be easy to change it and add new features.

Improvements PyGitHub module: Currently, the module saves data in a local SQLite database. To improve performance this can be changed by using a better database management system:
- adding a new configuration file inside the config folder e.g. “config/postgreSQL.py” that has the same function names as “config/sqliteConfig.py”
- changing inside “controller/PyController.py”, the dabase object “self.sqlConfig = SQLiteConfig()” to “self.sqlConfig = PostgreSQLConfig()”

Improvements BugFinder module: One possible improvement could be the addition of new programming languages. This can be achieved by:
- adding tokenizers for those languages e.g “util/JAVATokenizer.py”
- modifying the function “getConnectionsFile()” inside “BugFinder.py” in order to add connections between files of that programming language

Another improvement can be the analysis of the data produced for bug location. Currently, there is an algorithm that identifies candidate commits that introduced the bug. The improvement could consist in choosing the most probable commit that caused the bug.

Finally, another improvement could be related to bug security. There is an algorithm that identifies the issues related to security from the neo4j graph, by using pattern matching for certain security related words. Unfortunately, I could not build a security model because the dataset was too small. One solution to this problem could be the analysis of a larger repository, e.g. linux, which has more than 1million commits.

7.3 Future Developments

One addition to the project could be related to the cross-project topic, which was treated partially in the literature.

Possible ideas: understand from the configuration files which modules/libraries the project uses, then get the bugs for those modules/libraries. If the project uses an older version of the module, then it has those bugs.

8 Appendix

1. DeP: Defect prediction
2. IDE: Integrated development environment
3. IT: IssueType
4. LA: LinesAdded
5. LOC: Number of lines of code
6. LR: LinesRemoved
7. NC: NumberClasses
8. NCC: NumberChangedClasses
9. NCF: NumberChangedFunctions
10. NCPI: NumberCommitsPerIssue
11. NCoh: NbCohesion
12. NE: NumberEvents
13. NF: NumberFunctions
14. NFi: NumberFiles
15. NL: NumberLabels
16. NLC: NumberLinesCode
17. NP: NumberPeople
18. Ncom: NumberComments
19. Neo4J: A graph database that uses Cypher Query Language
20. RNIN: RepoNbIssueNb
21. S: Severity
22. SHA: Secure Hash Algorithm
23. SQLite: Open source application for SQL databases
24. TI: TimeInterval
25. UML: Unified Modeling Language

References

[1] Andreas Zeller, Thomas Zimmermann and Rahul Premr, [*Predicting defects for Eclipse*], http://cs.queensu.ca/~ahmed/home/teaching/CISC880/F10/papers/promise2007-dataset-20a.pdf

[2] Andreas Zeller, Thomas Zimmermann, E. James Whitehead and Sunghun Kim, [*Predicting faults from cached history*], http://thomas-zimmermann.com/publications/files/kim-icse-2007.pdf

[3] Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller, [*Predicting Bugs from History*], https://www.researchgate.net/publication/249606972_Predicting_Bugs_from_History

[4] Le Hoang Son, Nakul Pritam, Manju Khari, Raghvendra Kumar, Pham Thi Minh Phuong and Pham Huy Thong, [*Empirical Study of Software Defect Prediction - A Systematic Mapping*], https://www.researchgate.net/publication/331093339_Empirical_Study_of_Software_Defect_Prediction_A_Systematic_Mapping

[5] Junaid Ali Reshi and Satwinder Singh, [*Predicting Software Faults Using Software Metrics - A Review*], https://www.researchgate.net/publication/309834003_Predicting_Software_Faults_Using_Software_Metrics_A_Review

[6] Thomas J. Ostrand, Elaine J. Weyuker and Robert M. Bell, [*Predicting the Location and Number of Faults in Large Software Systems*], https://www.researchgate.net/publication/261170848_Predicting_the_Location_and_Number_of_Faults_in_Large_Software_Systems/download

[7] Maurice Dawson, Darrell Norman Burrell, Emad Rahim, Stephen Brewster, [*Integrating Software Assurance into the Software Development Life Cycle (SDLC)*], https://www.researchgate.net/publication/255965523_Integrating_Software_Assurance_into_the_Software_Development_Life_Cycle_SDLC

[8] Hans Raukas, [*Some Approaches for Software Defect Prediction*], https://www.semanticscholar.org/paper/Some-Approaches-for-Software-Defect-Prediction-Raukas/52dcf7da299874fb855dd68fd82df037621a5d4e

[9] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno, [*Bug prediction based on fine-grained module histories*], https://www.researchgate.net/

publication/254041680_Bug_prediction_based_on_fine-grained_module_histories

[10] Ishani Aroraa, Vivek Tetarwala, Anju Saha, [*Open Issues in Software Defect Prediction*], https://www.researchgate.net/publication/275720992_Open_Issues_in_Software_Defect_Prediction

[11] Awni Hammouri, Mustafa Hammad, Mohammad Alnabhan, Fatima Alsarayrah, [*Software Bug Prediction using Machine Learning Approach*], https://www.researchgate.net/publication/323536716_Software_Bug_Prediction_using_Machine_Learning_Approach

[12] *Threshold model*, https://en.wikipedia.org/wiki/Threshold_model

[13] *Support-vector machine*, https://en.wikipedia.org/wiki/Support-vector_machine

[14] *Regression analysis*, https://en.wikipedia.org/wiki/Regression_analysis

[15] *Artificial neural network*, https://en.wikipedia.org/wiki/Artificial_neural_network

[16] *Discriminant Analysis*, https://en.wikipedia.org/wiki/Linear_discriminant_analysis

[17] *Decision tree*, https://en.wikipedia.org/wiki/Decision_tree_learning

[18] *Bayesian learners*, https://en.wikipedia.org/wiki/Naive_Bayes_classifier

[19] *H-measure*, https://cran.r-project.org/web/packages/hmeasure/hmeasure.pdf

[20] *Metrics machine learning*, https://uni-obuda.hu/journal/Catal_36.pdf

[21] *Microsoft Visual Studio IDE*, https://visualstudio.microsoft.com/

[22] *Python library for GitHub API*, https://pygithub.readthedocs.io/en/latest/index.html

[23] *UML*, https://en.wikipedia.org/wiki/Unified_Modeling_Language

[24] *SQLite*, https://www.sqlite.org/

[25] *Neo4j*, https://neo4j.com/