Bug or not bug? That is the question

Quentin Perez, Pierre-Antoine Jean, Christelle Urtado, Sylvain Vauttier

Bug or not bug? That is the question

Quentin Perez, Pierre-Antoine Jean, Christelle Urtado and Sylvain Vauttier

EuroMov Digital Health in Motion, Univ Montpellier, IMT Mines Ales, Ales, France
Email: {Quentin.Perez, Pierre-Antoine.Jean, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr

***Abstract*—Nowadays, development teams often rely on tools such as Jira or Bugzilla to manage backlogs of issues to be solved to develop or maintain software. Although they relate to many different concerns (*e.g.*, bug fixing, new feature development, architecture refactoring), few means are proposed to identify and classify these different kinds of issues, except for non mandatory labels that can be manually associated to them. This may lead to a lack of issue classification or to issue misclassification that may impact automatic issue management (planning, assignment) or issue-derived metrics. Automatic issue classification thus is a relevant topic for assisting backlog management. This paper proposes a binary classification solution for discriminating bug from non bug issues. This solution combines natural language processing (TF-IDF) and classification (multi-layer perceptron) techniques, selected after comparing commonly used solutions to classify issues. Moreover, hyper-parameters of the neural network are optimized using a genetic algorithm. The obtained results, as compared to existing works on a commonly used benchmark, show significant improvements on the F1 measure for all datasets.**

***Index Terms*—Bug Classification, Bug Tickets, Empirical Software Engineering, Natural Language Processing, Neural Network Optimization, Genetic Algorithm,**

I. Introduction

Bug reporting and fixing are prominent software development activities. A study conducted by Sneed [1] evaluates that maintenance activities account for an average 20% of the effort made by a developer on a software project. In the same way, another study conducted by the University of Cambridge [2] estimates that the annual cost of software bugs is about $156 billion and amounts to 50% of maintenance costs. This situation creates a practical issue which is reported by Anvik *et al.* [3] as quotes from a Mozilla developer: "Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle". Automated solutions are therefore needed to help developers cope with the mass of tickets that may be submitted to issue tracker tools like Jira[1] or Bugzilla[2]. Automatic issue classification, that is required for subsequent automatic management steps like prioritization, planning or assignment, still is a challenge. Detecting misclassified issues (*i.e.*, issues tagged as bugs by developers but being actually requests for enhancement) is not simpler. Systems based on textual pattern matching are not adequate because tickets are written in free form natural language. For instance, basic buzzwords detection (*e.g.*, bug) is not robust enough. Efficient classification requires sophisticated information retrieval and machine learning techniques. This paper presents a solution based on a binary classification that separates bug and non-bug related issues. We studied and implemented a process that builds an efficient classification scheme. It relies on a combination of natural language processing (TF-IDF), statistical feature selection (Chi-square) and automatic classification (Multi-Layer Perceptron). To select this latter approach, several classification methods were benchmarked and Multi-Layer Perceptron (MLP) was selected as the most efficient. Furthermore, a MLP hyper-parameter (number of layers, layer sizes) optimization step fine-tunes the process thanks to a genetic algorithm. Obtained results have been compared to state-of-the-art thanks to a commonly used benchmark provided by Herzig *et al.* [4] containing 5,591 issue tickets already labeled as bugs or non bugs. Our solution improves classification performance, evaluated by the F1 measure, on all datasets.

The remainder of this paper is organized as follows. Section II presents existing binary bug classification approaches our proposal can be compared to. Section III details our designed approach to create a binary bug classifier. Section IV analyzes the obtained results. Section V discusses threats to validity. Section VI presents related works on closely related problems that inspired the solution we propose in this paper. Section VII concludes and proposes perspectives for this work.

II. Existing Binary Bug Classification Approaches

Bug management has always been a major concern for software quality. Recent works [5], [6], [7], [8] have studied how to predict different bug characteristics such as their number, locations, density, or severity. Prediction systems are based on data collected from bug repositories. Various statistical models have been implemented to reach specific aims such as ticket classification, bug assignment, bug severity evaluation and duplicate bug ticket detection.

Binary bug classification, that distinguishes bug tickets from other issues, is an actively studied research subject with many practical and industrial applications. Indeed, a manual analysis of 7,000 issue tickets from five open-source projects (HTTPClient, Jackrabbit, Lucene-Java, Rhino and Tomcat 5) concludes that 33.8% of bug tickets were misclassified [4]. This misclassification strongly impacts the efficiency of bug management (planing, assignment, metrics, etc.) in software development projects.

[1]https://www.atlassian.com/software/jira/bug-tracking
[2]https://www.bugzilla.org/

TABLE I
HERZIG *et al.* [4] DATASET DETAILS

	Maintainer	Tracker type	#Reports	#Labelled "BUG"	#Labelled "NBUG"
HTTPClient	Apache	JIRA	746	305	441
Jackrabbit	Apache	JIRA	2443	697	1746
Lucene-Java	Apache	JIRA	2402	938	1464
		Total:	**5591**	**1940**	**3651**

After a manual curating phase, Herzig *et al.* [4] have labelled their dataset with tags corresponding to six categories, including a specific category for bug tickets. This provides us with a mean to easily separate bug tickets from non bug issues.

Table I provides details on the dataset. Bug tickets are labelled as `BUG` while other issue tickets are labelled as `NBUG`. Many works [9], [10], [11], [12], [13], [14] use this dataset, and more precisely a subset composed of the HTTPClient, Jackrabbit and Lucene-Java projects, as a benchmark to design and validate bug ticket automatic classification approaches. All the issue tickets extracted by Herzig *et al.* for these three projects come from the Bugzilla Issue Tracking System (ITS).

The approach proposed in this paper is evaluated on the same project subset. Our results can thus be compared to six other recent (from 2013 to 2019) works that use various ways for bug binary classification, most of them based on machine learning techniques. Their results are presented in Table II[3]. Columns **HTTPClient**, **Jackrabbit** and **Lucene** provide the F1 Measure on each subset of tickets from each project evaluated independently. The **Mean Projects** column shows the mean of those three F1 measures whereas the **Cross-Project** column gives the F1 measure calculated on the whole subset by mixing data from all three projects.

Terdchanakul *et al.* [9] describe a topic-based model with N-gram IDF. N-gram IDF is used to extract key terms of any length from texts. These key terms are then used as features to classify bug reports. The classification algorithms they experiment are Logistic Regression (LR) and Random Forest (RF). Chawla *et al.* [10] describe an automated approach based on fuzzy set theory. Pingclasai *et al.* [11] adopt a topic-based classification using three classifier algorithms: Decision Tree (DT), Naive Bayes (NB) and LR. Topics are modeled thanks to Latent Dirichlet Allocation. The output of this process is a collection of topic membership vectors used as input features in classifiers. F1 measure and k-fold (k=10) are used as the evaluation protocol. Luaphol *et al.* [12] aim to discover the most efficient features for binary bug report classification. This study compares seven ways of processing textual information using sub-sequences of words: unigrams, bigrams, camel case, unigrams and bigrams, unigrams and camel case, bigrams and camel case, and all kinds of sub-sequences together. The experimental results show that unigrams may be the most efficient features for binary bug report classification. These features are processed through various classification algorithms: NB, LR, Support Vector Machine (SVM) and Radial Basis Function (RBF) kernel. Pandey *et al.* [13] and Pingclasai *et al.* [11] analyze how machine learning techniques may be used to perform issue classification. Authors evaluate the performance (in terms of F1 measure and average accuracy) of several classification algorithms: NB, Linear Discriminant Analysis, k-Nearest Neighbors (kNN), SVM with various kernels, Decision Trees and RF separately. Qin *et al.* [14] propose a bug classification method based on Long Short-Term Memory (LSTM), a typical recurrent neural network which is widely used in text classification tasks, with a softmax layer to classify data. A softmax layer assigns decimal probabilities to each class of a multi-class problem. The sum of these decimal probabilities must be 1. This additional constraint is known to fasten learning convergence.

Table II presents results obtained by the five works described above. The results are ordered by descending values of **Cross-Project** measure. Unfortunately, three studies [13], [10], [11] do not provide this information: this is why the only way to compare these results is to compute the **Mean Projects** value. Evaluation protocols are also different depending on the studies. Only three studies [9], [11], [13] use a robust evaluation protocol, *i.e.*, k-fold cross-validation (k=10). Other works [10], [12], [14] use a less robust protocol (90/10 or 70/30 split). The lowest **Cross-Project** measure is obtained by Qin *et al.* [14] (0.746) using a Training/Test split.

Best results are those of Terdchanakul *et al.* [9] on all measures. Moreover, they use a robust evaluation protocol making them our chosen reference to compare our results to in the remaining of this paper.

III. PROPOSED BUG CLASSIFICATION APPROACH

A. Approach Overview

This section provides an overview of our proposed approach for binary bug classification, from corpus extraction to classifier optimization using genetic algorithms. For reproducibility purposes, the source code and data used for this paper are available online[4]. As shown on Figure 1, our bug classification process is composed of five main steps.

The first step is **corpus extraction**. Bug tickets all have a unique identifier as provided by the dataset of Herzig *et al.* [4]. These identifiers are used to retrieve the ticket corpus we use as our benchmark, using the HTTP APIs provided by the Jira ITS. Data is compiled in a unique JSON file.

[3]Confidence intervals are not reported because these figures were not provided in the papers.

[4]https://github.com/qperez/ATTIC

TABLE II
STATE-OF-THE-ART ON BUG CLASSIFICATION WITH THE DATASET OF HERZIG *et al.*

Study	F1 Measure					Evaluation Protocol
	HTTPClient	Jackrabbit	Lucence	Mean Projects	Cross-Project	
Terdchanakul *et al.* [9]	0.814	0.805	0.884	0.834	0.814	10-fold
Chawla *et al.* [10]	0.830	0.780	0.840	0.817	–	Training/Test split 80/20
Pingclasai *et al.* [11]	0.758	0.767	0.818	0.781	–	10-fold
Luaphol *et al.* [12]	–	–	–	–	0.770	Training/Test split 70/30
Pandey *et al.* [13]	0.687	0.759	0.708	0.718	–	10-fold
Qin *et al.* [14]	0.757	0.771	0.717	0.748	0.746	Training/Test split 90/10

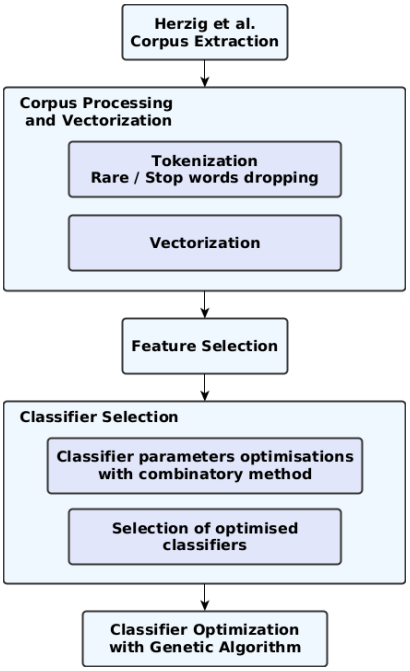

Fig. 1. Bug Classification Process Overview.

The second step is **corpus processing**. Text is filtered (to keep only relevant information) and converted into numerical data using Natural Language Processing (NLP) techniques. The text of each ticket is tokenized (*i.e.*, decomposed in its smallest units). Tuples of successive tokens (sliding windows), called n-grams, are then built. N-gram frequencies are calculated and n-grams are filtered when they are over-represented in the corpus. These over-represented n-grams often correspond to stop-words (*e.g.*, articles such as "the", "an", "a"). Rare words (*i.e.*, words appearing only once or twice in a given document) are also dropped from the corpus. Finally, each ticket is represented as a vector representing the n-gram frequencies of its content. This way, each ticket is transformed into a numerical vector that provides a statistical model of its textual content in a format that can further be processed by classifiers.
The third step is **feature selection**. Vectors representing n-grams (features) have a huge number of dimensions. However, all n-gram frequencies do not have the same importance for ticket classification. A statistical selection is therefore performed to select the best relevant dimensions in order to reduce the size of the vectors. This improves the performance of machine learning (classifier training), regarding both computation complexity and classification accuracy.
The fourth step consists in **classifier selection**. As multiple classifier types are available, selecting the right classifier type according to its performance is our first goal. Having an annotated dataset, we explored supervised learning techniques. The Scikit-learn API [15] provides six kinds of classifiers corresponding to the state-of-the-art:

- **Stochastic Gradient Descent (SGD)** uses the iterative gradient descent method to minimize an objective function written as a sum of functions:

$$Q(w) = \frac{1}{n}\sum_{i=1}^{n} Q_i(w),$$

where w is the parameter to be estimated in order to minimize function $Q(w)$. Q_i corresponds to the i-th observation in the training dataset.
- **Support Vector Machines (SVM)** are non probabilistic classifiers based on linear algebra. Training SVMs creates hyperplanes that separate multidimensional data into different classes. SVMs optimize hyperplane positions by maximizing their distance with the nearest data. These classifiers generally reach a good accuracy.
- **Random Forest (RF)** is a parallel learning method based on multiple randomly constructed decision trees. Each tree of the random forest is trained on a random subset of data according to the bagging principle, with a random subset of features according to the principle of random projections.
- **Ridge Regression (RR)** is based on linear least squares regression with a $L2$ regularization (known as Tikhonov's): a penalty equal to the sum of the squares of the weights, multiplied by a α penalty strength factor, is added to the f_{loss} objective function being minimized. Ridge regression can thus be defined as follows:

$$Obj = f_{loss}(w) + \alpha \cdot \sum w^2$$

f_{loss} depends on the underling task (*e.g.*, cross-entropy loss for classification). α is generally adjusted during model validation and is called the regularization parameter.

- **k-Nearest Neighbors (kNN)** is a non-parametric method in which the model stores the data of the training dataset to perform the the classification of the test dataset. To assess the class of a new input, kNN looks for its closest k neighbors using a distance formula (*e.g.*, Euclidean distance) and chooses the class of the majority of neighbours.
- **Multi-Layer Perceptron (MLP)** is a type of formal neural network that is organized in several layers. Information flows from the neurons of the input layer to the neurons of the output layer through weighted connections. Supervised training incrementally adjusts the weights of connections (error backpropagation) so that the expected outputs can be learned by the MLP. Through the use of multi-layers, a MLP is able to classify data that is not linearly separable (using multiple learned hyperplanes).

Main classifier parameters, called their hyper-parameters, are tested by a combinatorial method to find the best configuration. Finally, the classifier that produces best results is selected: MLP performs best and will thus be used in the remaining.
The last step is MLP **classifier optimization using a genetic algorithm**. Optimization regards number of input features (*i.e.*, size of the input layer) and the structure of the MLP (number and sizes of hidden layers). We chose these parameters because their optimization is not efficient with combinatorial or dichotomous methods (as compared to the hyper-parameters configured in the previous step).
The result of our approach is a classifier based on MLP with its hyper-parameters optimized with a genetic algorithm. Performances of our optimized MLP classifier will further be compared to results obtained by Terdchanakul *et al.* [9] in Section IV.
After this coarse grained presentation, next subsection provides further details on our approach.

B. Detailed Approach

1) Bug Ticket Corpus Extraction: Data used in this study are based on the corpus of Herzig *et al.* [4]. 5,991 bug tickets are extracted from three popular software projects: *Lucene*, *JackRabbit* and *HttpClient*. After the raw extraction, tickets are mapped to their classification (*i.e.*, bug or not bug) as given by the Herzig corpus, thanks to corresponding ticket identifiers. Finally, a JSON file containing the set of tickets and their classification is produced. This file is subsequently used for text processing but also to test and train classifiers. A sample of annotated issue ticket is given by Listing 1. The JSON data structure is composed of six fields:

- `key` represents the unique identifier for the issue ticket in the ITS, named ticket ID.
- `summary` is the short description of the issue.
- `description` is the long description of the issue. For a bug, it should describe the way it appears and the problems it causes.
- `classification` is the category in which Herzig *et al.* classified the issue.

```
{
  "key": "HTTPCLIENT-126",
  "summary": "Default charset",
  "description": "As defined in RFC2616 the
      default character set is ISO-8859-1 an
      not US-ASCII \nas defined in
      HttpMethodBase. See \"3.7.1
      Canonicalization and Text Defaults\" at\
      nRFC 2616",
  "classified": "IMPROVEMENT",
  "type": "BUG",
  "label": "NBUG"
}
```

Listing 1. Sample issue ticket used to train and test classifiers.

- `type` is the issue category as mentioned by the issue opener in the repository.
- `label` is our binary classification of the issue: `BUG` for a ticket classified by Herzig *et al.* as a bug (in field `classification`). Other categories arc classified `NBUG`.

2) Corpus Processing: After the extraction, a bag-of-word processing is performed on the corpus in order to code textual data into a vector representation that classifiers can use. This step is not neutral as it may significantly impact the classifier performance. We rely on the Scikit-learn API [15], which is widely used in machine learning projects conducted by public laboratories and companies as a standard, configurable framework providing a wide range of methods. As shown in Figure 2, corpus processing is divided into two main steps: Natural Language Processing and Feature Selection.

Natural Language Processing. In this step, the corpus is cleaned and then projected into a vector representation. `TfidfVectorizer` provided by the Scikit-learn API [15] is used to process the textual data and compute statistical data using the Term Frequency Inverse Document Frequency (TF-IDF) method [16].

(a) **Tokenisation**. Textual data is divided into text units called tokens. Each token tok represents a word. From these tokens, n-grams are created. A n-gram is a sequence of n contiguous tokens that can formally be written as follows: $n\text{-gram} = \{tok_1, tok_2, ..., tok_n\}$. In this paper we use uni-grams, bi-grams and tri-grams. Listing 2 shows some uni-grams created from the bug ticket of Listing 1. In turn, Listing 3 and Listing 4 respectively show samples of bi-grams and tri-grams.

```
"see", "371", "canonicalization", "text",
"defaults", "rfc", "2616"
```

Listing 2. Sample of uni-grams created with the last sentence in Listing 1.

```
"see 371", "371 canonicalization"
"canonicalization text", "text defaults",
"defaults rfc", "rfc 2616"
```

Listing 3. Sample of bi-grams created with the last sentence in Listing 1.

```
"see 371 canonicalization",
"371 canonicalization text",
"canonicalization text defaults",
"text defaults rfc", "defaults rfc 2616"
```

Listing 4. Sample of tri-grams created with the last sentence in Listing 1.

(b) **Stop words and rare words dropping**. Tokens with high occurrence rates in the corpus (often stop words) are not relevant for classification. Hence, tokens appearing in more than 50% of the tickets of the corpus are filtered. Conversely, tokens with very low occurrence rates are also often not relevant. Thus, tokens appearing in less than 2 tickets from the corpus are also filtered. An issue ticket contains two main parts: a summary and a description. An example of summary and description is given on Listing 1. The summary part is actually the title of the ticket while the description part explains the reported issue. Some studies [13] use only the summary and ignore the description. In our work, we have experimented classification using only the summary or only the description. However, results are better when both are used. Besides, to capitalize on previous studies [13], [17] about the importance of the summary, we have experimented to increase its weight by duplicating its content. After iterative tests, it happens that duplicating three times the contents of the ticket summary produces the best results.

(c) **N-gram frequency calculation**. Occurrences for each uni-gram, bi-gram or tri-gram is counted. As a result, a tuple is produced for each n-gram containing its frequency in each ticket.

(d) **TF-IDF normalisation**. The Term Frequency Inverse Document Frequency (TF-IDF) method implemented in Scikit-learn is used to weight and normalize n-gram frequencies in the corpus. This statistical method evaluates the importance of a t term (a n-gram) contained in a d document (a ticket), relatively to the D corpus of documents (the whole ticket dataset).
The weight increases with the number of occurrences of the t term in the d document and decreases with the frequency of the t term in the D corpus thus emphasizing terms which presence is discriminating.
The TF-IDF formula is:

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D)$$

tf corresponds to the number of occurrences of a t term in the d document (noted $n_{t,d}$) divided by the total number of terms in the d document (noted n_d):

$$tf(t, d) = 1 + \log\left(\frac{n_{t,d}}{\sum n_d}\right)$$

tf is calculated as $1 + log(tf)$ to cap the weight of very frequent terms.

idf represents the importance of the t term in the D corpus as a whole. The number of documents where the t term appears in the D corpus is noted $n_{t,D}$:

$$idf(t, D) = \log\left(\frac{1 + |D|}{1 + n_{t,D}}\right) + 1$$

idf is defined here in its smooth version by adding 1 after the log function. Constant 1 is added on numerator and denominator to prevent division by zero.
This process returns a sparse matrix with TF-IDF weights for all n-grams and all tickets of the corpus.

Feature selection. Selection of the best features (the n-grams) is performed before classifier training in order to avoid irrelevant and noisy features. Indeed, the initial number of features is huge: 24,496 features considering only uni-grams, 63,925 features counting both uni-grams and bi-grams and 99,351 features including uni-grams, bi-grams and tri-grams. The number of selected features impacts training time, model fitting and classifier quality.
A Chi-square test is used to measure the association relation between our categorical variables (`BUG` or `NBUG`) and our features. The Chi-square test statistically selects a number of most relevant features for each categorical variable that is fixed by the user. We empirically have fixed the number of features to be selected by the Chi-square test to 30,000. To determine this value, we have sampled the feature number in the range $[5,000 - 60,000]$ with 5,000 increments. For each sample value, we have computed the F1 measure of each classifier tested in this paper (using 10-fold validation). Below 30,000, all F1 measures are very low which means that classifiers all perform poorly. For each value over 30,000, F1 measures of the tested classifiers are too different to be easily comparable. It thus appears that 30,000 is the best trade-off for the global performance of classifiers.

3) Classifier Selection: Several classifiers are compared in order to select the best suited one. To do so, their hyper-parameters are set using a brute force optimization method and their performances compared on sets of 30,000 selected features.
Figure 3 shows the six classifiers used in this study and described above. All are configured with hyper-parameters which values influence the model building process. Choosing correct values for hyper-parameters is crucial to have the best possible prediction. To do so, a Grid-Search algorithm is used to optimize the hyper-parameters of each classifier. Grid-Search, also called parameter sweep, is a brute-force method that searches for an optimal parameter value combination using their n-fold Cartesian product. A discrete set of values to be tested is provided for each hyper-parameter. All possible n-tuples of hyper-parameter values are generated and used to initialize the classifier. Finally classifiers are trained with 30.000 features and evaluated by a 10-fold. The Multi-Layer Perceptron (MLP) produces the best results (as evaluated by the F1 measure) and is therefore selected as our classification technique.

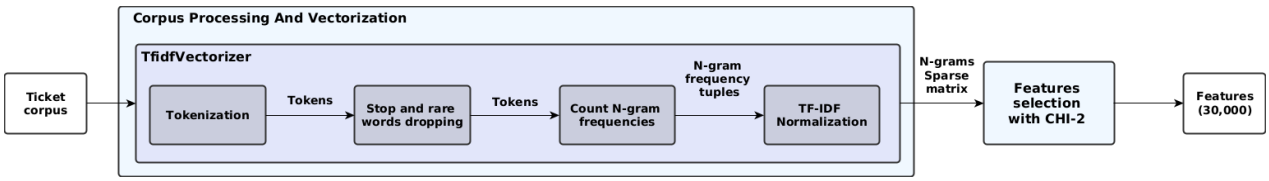

Fig. 2. Corpus processing steps.

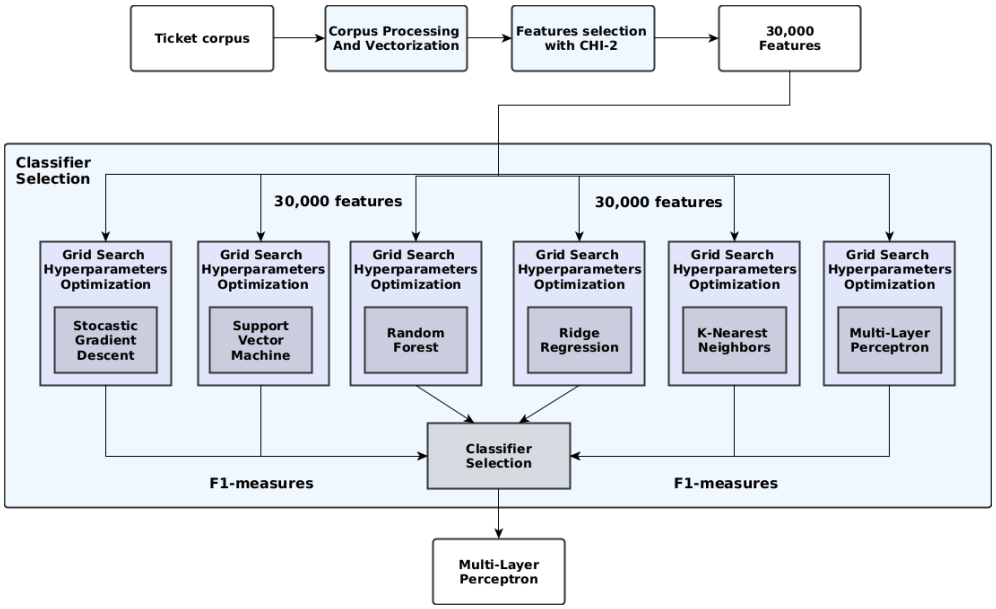

Fig. 3. Classifier selection process.

4) Classifier Optimization: Hyper-parameter value optimization is central for prediction quality. Hyper-parameter values cannot be fine-tuned by the coarse Grid-Search method proposed above, that requires lists of discrete values to create combinations. A precise classifier optimization needs to traverse a continuous value interval for each hyper-parameter to better explore the search space.

To do so, our MLP classifier is further optimized using a genetic algorithm (GA) [18]. GAs can optimize values precisely within continuous value intervals.

Three MLP hyper-parameters must be optimized:

- number of features: size of the feature vectors used to train and test by the MLP classifier (equivalent to the number of neurons on the input layer of the MLP),
- number of hidden layers for the MLP,
- hidden layers sizes: numbers of neurons on each hidden layer.

Figure 4 presents the GA we have implemented. GAs are dedicated to the search for optimal solutions using stochastic techniques inspired from evolutionary biological mechanisms such as mutation, crossover and natural selection. An initial set of solutions, called the population, is randomly generated. Each solution, called an individual, is described by a set of characteristics stored in a list of values called its chromosome. Each individual is evaluated thanks to an objective function. The score of each individual is called its fitness. Like in natural selection theory, individuals with the highest fitness have the best chances to survive and to reproduce themselves. A subset of the best individuals, called the parents, are selected to generate a set of new individuals, called their offspring. Each individual in the offspring is generated by a couple of parents thanks to a crossover of their chromosomes. Random mutations are applied to maintain diversity in the population in order to balance search space exploration with optimization convergence. The offspring replace the least fitted individuals and creates a new generation of the population (which size remains constant throughout generations). This process is iterated until a stop condition is reached (for instance a time limit or a maximum number of iterations). Thanks to this iterative selection and the recombination of the best known solutions, GA are eventually able to find optimized and even optimal solutions for complex combinatorial problems.

The initial population is generated using hyper-parameter values that are chosen randomly inside fixed intervals:

- number of features ($[20,000 - 60,000]$). These bounds correspond to the lower bound used in the Grid-Search step and approximately two-thirds of the maximum number of features (99,351). The value empirically determined during the feature selection phase is in this interval (30,000).

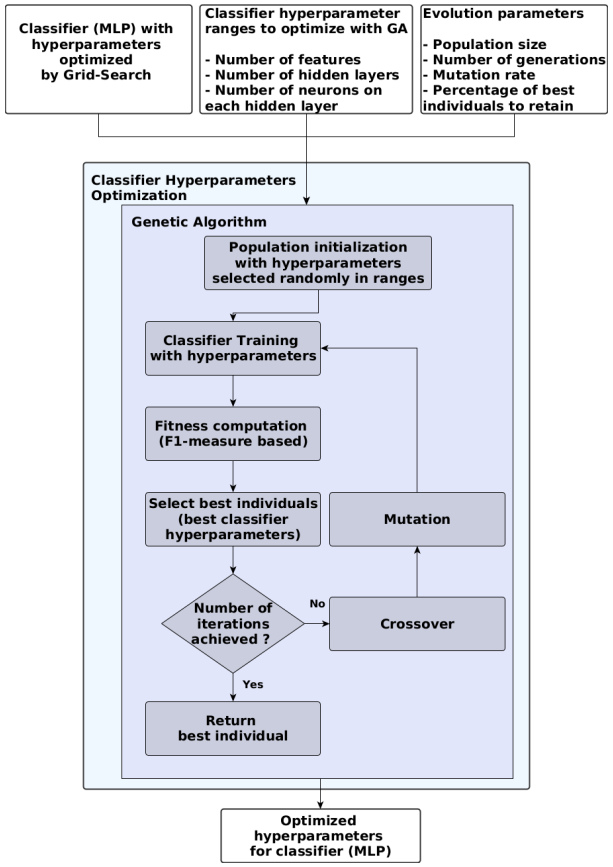

Fig. 4. Evolutionary system implemented to find best MLP parameters.

- number of hidden layers ($[2-15]$). The upper bound is chosen to limit computation time.
- hidden layer sizes ($[1-30]$). This range creates variability in the generated structures with reasonable computation time.

Besides, our proposed GA has its own hyper-parameters that control different aspects of its evolutionary strategy. These parameters are used to create a population, select the best individuals and introduce diversity at each generation:

- Population size. We run different experiments with 50, 100, 200 and finally 300 individuals, following De Jong's recommendation to chose moderated population sizes [19].
- Number of generations. It is fixed to 150 iterations to have an acceptable computing time.
- Percentage of best individuals to retain in the population ($p_{ret} = 20\%$).
- Probability of mutation for each individual ($p_{mut} = 0.1$). The p_{mut} value has been studied in many works [19], [20], [21] and is thus fixed to its recommended value.
- Random selection level ($p_{sel} = 0.3$). This parameter controls the probability to randomly select a parent in order to maintain diversity in the population [22].

As shown in Figure 4, the GA processes seven steps:

1) The population is initialized with randomly generated individuals. Each individual is described by hyper-parameter values used to configure its corresponding MLP: a number of features and a tuple of hidden neuron layer sizes (tuple size corresponds to the number of hidden layers). An individual i can be described for instance as $i(25540, (11, 25, 18, 6, 13))$: 25540 features (or entry neurons in first layer) and 5 neuron layers of respectively 11, 25, 18, 6 and 13 neurons from the first hidden layer to the last hidden layer.
2) The MLP corresponding to each new individual is trained, using a 75/25 split of train/test dataset. This strategy is chosen for its speed and low computational cost.
3) Fitness of individuals is based on the performance of the MLP on the test dataset. F1 measure is chosen because it requires both high precision and high recall.
4) Best individuals are selected according to their fitness and the percentage (p_{ret}) of individuals to retain .
5) The selected individuals reproduce and their offspring is generated thanks to a crossover of their chromosomes. Couples of parents are randomly chosen among the selected individuals. Crossover is performed both on the number of features and on the neurons layers :
   - Crossover on the number of features is performed using a mean between the number of features of the two parents.
   - Crossover on neurons layers is done using a single cut point at the middle of the hidden layers of the two parents a and b [23]. If the number of layers is odd the cutting point is rounded down. Children thus inherit half of their structures from each of their parents, as shown in the following example. The first half of the layers for parent a is transmitted whereas the rest is deleted. For parent b, the second half is transmitted.

$$i_{parent_a}(21912, (12, 23, 8, 4))$$
$$Crossover$$
$$i_{parent_b}(30023, (4, 23, 5, 13, 27))$$
$$\Downarrow$$
$$i_{child}(\lfloor(21912 + 30023)/2\rfloor, (12, 23, 8, 4) \cdot (4, 23, 5, 13, 27))$$
$$\Downarrow$$
$$i_{child}(25967, (12, 23, 5, 13, 27))$$

6) Mutation is performed. It consists in changing a random gene in an individual. In this approach, mutation impacts solely hidden neuron layers. Three kinds of mutation, randomly chosen from with an equal probability ($= \frac{1}{3}p_{mut}$ each), are possible:
   - Addition: adds a neurons layer randomly generated in the individual. Example:

$$i(21912, (12, 23, 45))$$
$$Addition$$
$$\Downarrow$$
$$i(21912, (12, 23, 45, 18))$$

- Deletion: deletes a neurons layer in the individual. Example:

$$i(21912, (12, 23, 45))$$
$$Deletion$$
$$\Downarrow$$
$$i(21912, (12, 45))$$

- Substitution: chooses a number of neurons randomly in the hidden layer sizes bounds. Then selects randomly a layer in the individual to change it by the number of neurons previously chosen. Example:

$$i(21912, (12, 23, 45))$$
$$Substitution$$
$$\Downarrow$$
$$i(21912, (7, 23, 45))$$

7) The best individual found is returned when the fixed number of generations is reached. This individual provides the best optimized values found for MLP hyper-parameters.

IV. Results

A. Classifier Selection

As stated in Section III-B3, we compare the performance of six kinds of classifiers using F1 measure: Stochastic Gradient Descent (SGD), Support Vector Machines (SVMs), Random Forest (RF), Ridge Regression (RR), k-Nearest Neighbors (kNN) and Multi-Layer Perceptron (MLP). Hyper-parameters for each classifier coarsely optimized by the Grid-Search step and performances are evaluated with a 10-fold. Results for this experiment are presented in Table III. The Grid-Search algorithm finds a set of optimized hyper-parameters for each classifier (see Table III). Using these hyper-parameters, MLP obtains the best results (0.868). Hence, MLP is selected for our classification approach and parameterized as proposed by the Grid-Search algorithm as a starting configuration.

TABLE III
Results obtained with SciKit classifiers using a 10-fold cross-validation and 30,000 selected features.

Classifier	Grid-Search Optimized Classifier Parameters	F1 Measure	Evaluation Protocol
MLP	activation='tanh' learning_rate='adaptive' max_iter=100 random_state=0	**0.868 CI 95%: 0.034**	10-fold
SVM	C=100 gamma='scale'	0.857 CI 95%: 0.042	10-fold
SGD	loss='modified_huber' max_iter=5000 random_state=0	0.841 CI 95%: 0.037	10-fold
RR	random_state=0	0.819 CI 95%: 0.050	10-fold
RF	criterion='entropy' n_estimators=20 random_state=0	0.610 CI 95%: 0.089	10-fold
kNN	weights='distance' n_neighbors=2	0.449 CI 95%: 0.107	10-fold

B. Intermediate Results

In order to fine-tune our classification process, we study the influence of each step on global performance. We therefore compute F1 Measure and accuracy metrics for five different settings. Results are shown in Figure 5.

Setting 1 uses uni-grams and TF-IDF without logarithmic term frequency attenuation. No feature selection is done. MLP uses the default hyper-parameter values set by the Scikit-Learn API.

In Setting 2, TF-IDF vectorization uses uni-grams, bi-grams and logarithmic term frequency attenuation. Rare and stop words are filtered. All features are used (63,924). MLP hyper-parameters are set to default. A noticeable gain in terms of precision (+0.068) and accuracy (+0.101) can be measured on cross-project as compared to Setting 1. F1 measure improvement is smaller (+0.027) because of a lower recall.

In Setting 3, TF-IDF vectorization uses uni-grams, bi-grams and tri-grams. This setting also performs a Chi-square selection on features (30,000 features). MLP hyper-parameters are still set to their default. Tri-grams and feature selection have a positive influence on the F1 measure (+0.092) thanks to improvements on recall (+0.070) as compared to Setting 2.

Setting 4 uses the same TF-IDF vectorization and feature selection as in Setting 3. The MLP is configured with the hyper-parameter values optimized by the Grid-Search step (see Table III). Gain of performance on all measures is close to null.

Setting 5 uses same TF-IDF and feature selection as in Settings 3 and 4. MLP hyper-parameters are now optimized by the GA. Recall is improved for all projects and on cross-project (+0.068) as compared to Setting 4. Precision is slightly degraded but remains rather stable, especially on cross-project. This way, F1 measure is improved for all projects and on cross-project (+0.031). Classification is thus a little less precise but more robust.

Results of GA and comparison with the state-of-the-art are described in the following section.

C. Classifier Fine-Tuning with a Genetic Algorithm

As stated in Section III-B4, we optimize the hyper-parameters of the MLP classifier thanks to a GA. Besides, we use the other options of Setting 4, as described in Section IV-B. The GA is run with hyper-parameter values as defined in Section III-B4 on 150 generations. Figure 6 shows the evolution of fitness over generations with four different population sizes. Smaller population sizes with 50 and 100 individuals have the best results while larger populations have the worst results. These differences can be due to the parent selection mechanism we implemented. As parents are randomly selected in a population subset based on a fixed proportion of the best individuals (20%), smaller population entails smaller parent sets, containing on average better fitted individuals. This may result in a more elitist evolution strategy leading to a quicker fitness improvement. Confirmation of this hypothesis is a perspective. GA executed with 50 individuals and 150 generations returns an optimized individual with a configuration that uses 7 hidden layers with sizes varying from 9 to 15 and 37,362 input features: $i(37362, (15, 9, 10, 11, 9, 15, 11))$.

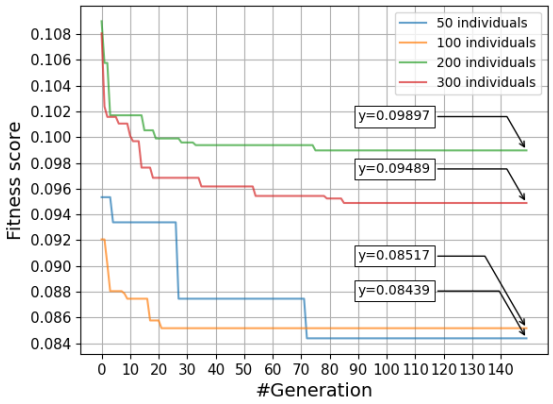

Fig. 6. Fitness evolution through 150 generations with genetic algorithm.

To compare the best individual returned by the GA to state-of-the-art results, we use this individual to perform a 10-fold validation on the benchmark of 5,591 tickets. Our results are presented in Table IV as compared to those obtained by Terdchanakul *et al.* (best results from state-of-the-art).

Our proposed approach obtains a gain of +0.016 on the mean F1 measure over all the projects, as compared with the scores obtained by Terdchanakul *et al.*. Our solution improves results on two out of three projects: HTTPClient (+0.002) and Jackrabbit (+0.081). A low deterioration is observed on Lucene (-0.010). F1 measure on cross-project is much improved (+0.081) increasing from 0.814 to almost 0.9. Our result (0.896) is also very good because it is very close to 1. As shown in Figure 5, our solution achieves 0.881 for recall, 0.913 for precision and 0.929 for accuracy.

Final result:

> Our results are not only high but also well balanced between recall and precision, thus highlighting the quality of our proposed classifier, that outperforms scores obtained by Terdchanakul *et al.*.

V. Threats To Validity

This section analyzes the threats to the validity of our proposal.

Internal Threats. The major internal threat comes from the quality of the dataset created by Herzig *et al.* [4] and used in this paper. Labeling errors resulting from this manual process could affect classifier training and *in fine* prediction quality. However, as a standard dataset, it is considered of good quality by many academics [9], [10], [11], [12], [13], [14] that work issue ticket classification. An existing bias would affect equally all state-of-the-art works based on it.

External Threats. External validity refers to the generalizibility of the treatment/condition outcomes. Projects

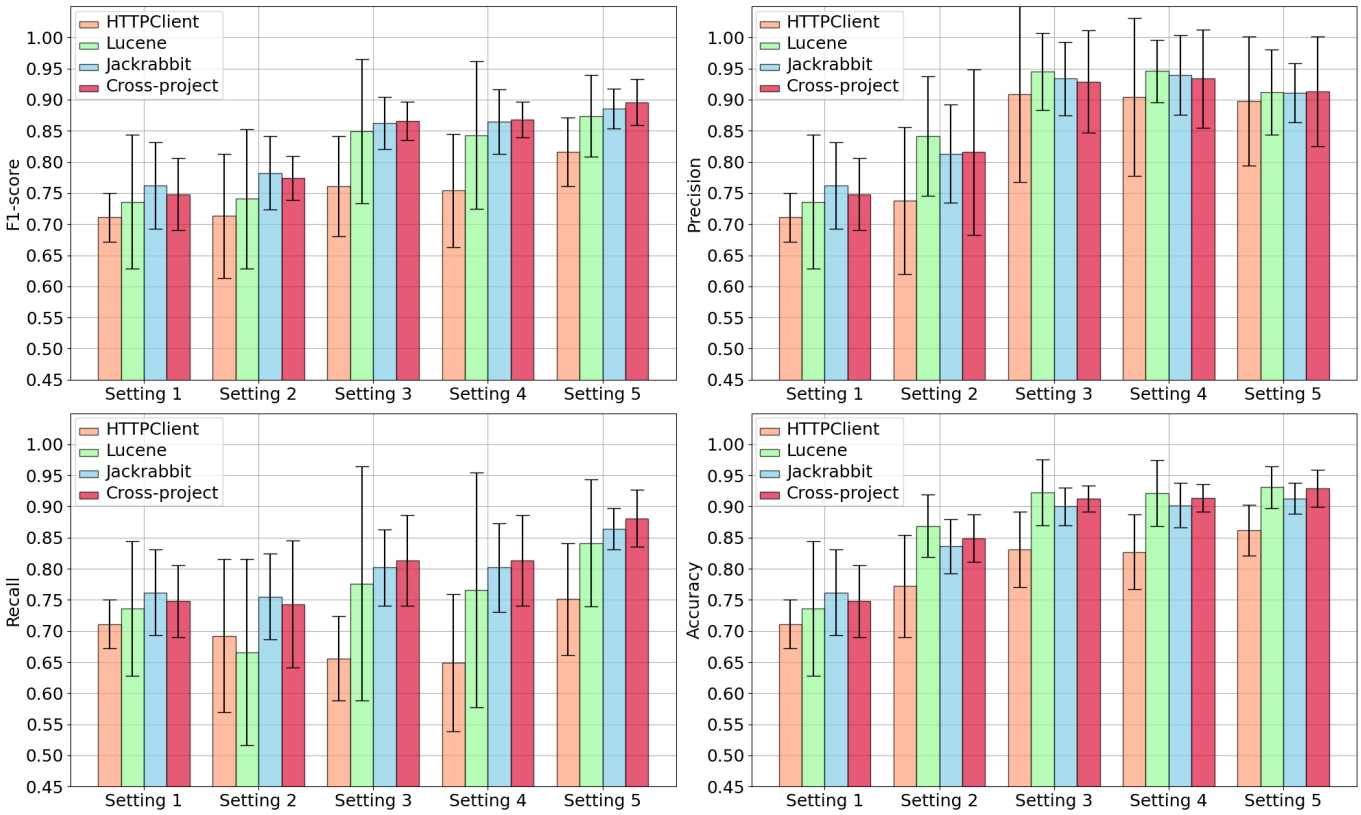

Fig. 5. Result of measurements on the 5 settings tested.

TABLE IV
COMPARISON BETWEEN RESULTS OBTAINED BY TERDCHANAKUL *et al.* [9] AND OUR APPROACH.

F1 Measure									**Evaluation Protocol**
Terdchanakul *et al.* [9]									
HTTPClient		Jackrabbit		Lucene		Mean Projects	Cross-Project		10-Fold
0.814		0.805		0.884		**0.843**	**0.814**		
Approach presented in this paper: MLP optimized with GA									
HTTPClient		Jackrabbit		Lucene		Mean Projects	Cross-Project		10-Fold
0.816 (+0.002)	CI 95%: 0.055	0.886 (+0.081)	CI 95%: 0.032	0.874 (-0.010)	CI 95%: 0.066	**0.859 (+0.016)**	**0.896 (+0.082)**	CI 95%: 0.037	

from which the dataset is generated are open-source and only written in Java. If we have a close look to selected features some are Java-dependant: class names (*e.g.*, `NullPointerExpection`, `DefaultHttpClient`, `FileInputStream`) or Java keywords (*e.g.*, `new`, `catch`, `throws`) that are used in ticket description are selected as features. Thus, generalization to projects written with other programming languages and technologies is yet to be studied. The performance of AI techniques is also sensitive to computation resources. Comparing resource consumption for different approaches is tricky because of the diverse hardware and software used. Our classifier training process, while rather complex, was executed on a classic PC configuration (Intel core i5-7600 cadenced to 3.5 GHz with 8GB of RAM) and takes from a few minutes (training of a single MLP) to a few days (training and evolution of a large population of MLPs). However, resource consumption is not an actual limitation for this approach as it only regards the evolutionary selection and the training of the MPLs. Once selected and trained, the use of the MLP is fast and has a small footprint. Moreover, best results were obtained with the smaller MLP populations.

VI. RELATED WORKS

Ticket classification is a widely studied field. If we extend the scope of related works beyond the binary bug classification solutions presented and discussed in Section II, we see that many statistical models have been used for closely related problems.

Few approaches exist that perform bug ticket classification on alternate datasets. Kallis *et al.* [24] propose a binary bug ticket classifier based on a pre-trained text classification tool and transfer learning. In a following article [25], they develop a GitHub plugin named Ticket Tagger that automatically classifies tickets as they are written (on the fly). As compared to our proposal, they use a wider yet not curated dataset. Indeed, they have collected 30,000 tickets from 12,112 heterogeneous projects labbeled in an *ad hoc* manner by project contributors. Unlike the dataset proposed by Herzig *et al.* [4], this one has not been manually curated and can thus supposedly suffer form flaws. A perspective is to compare our solution to theirs.

Other closely related approach perform multi-class ticket classification (*i.e.*, a more precise ticket classification in more than two predefined categories), bug-assignment (*i.e.*, automatic distribution of bug-tickets to expert developers), bug severity evaluation and duplicated bug ticket detection. Some of these approaches have commonalities with the proposal of this paper:

- **Some use neural networks as their classification technique.** Kukkar *et al.* [5], for example, have worked on bug severity classification. They have developed a method to classify bug ticket severity based on deep learning with convolutional neural networks mixed with random forest boosting.
- **Some use the same labeled ticket dataset.** Limsettho *et al.* [6] have proposed a multi-class classification approach, based on Herzig *et al.* dataset [4], in order to dispatch tickets into five categories: Request For Enhancement, Bug, Improvement, Task and Test. They used NLP techniques combined with Hierarchical Dirichlet Process and Latent Dirichlet Allocation.
- **Some use genetic algorithms to fine-tune their classifiers.** For example, Miller *et al.* [26] and Whitley *et al.* [27] have used GAs to find an optimized solution for connecting neurons. Another way of optimizing neural networks is to use GAs for the improvement of feed forward neural network weights [28], [27]. Others studies [29], [30] also fine-tuned hyper-parameters using GAs.

Even if the problems they tackle are slightly different from ours, these related work inspired us and comforted our choices and results.

VII. CONCLUSION

In this paper, we propose a solution for the automatic classification of issue tickets into two categories: bug related or not. Our solution combines natural language processing (TF-IDF), machine learning (MLP) and optimisation (genetic algorithms). For each technique we use, we have extensively studied the influence of their main options and parameters on classification performance. The hyper-parameters of the MLP are ultimately automatically optimized thanks to the GA we designed and implemented. Our solution has been validated on a standard dataset provided by Herzig *et al.* containing 5,991 tickets coming from 3 popular software projects. Our results can therefore be rigorously compared with state-of-the-art works. We computed F1 measure, recall and precision for our best MLP classifier, optimized with our genetic algorithm. We obtained a F1 Measure of 0.896 on cross-project tickets (*i.e.*, whole dataset). This corresponds to

a noticeable performance improvement (gain of +0.082) as compared the best score established by Terdchanakul *et al.* on the same dataset. The balance between recall (0.881) and precision (0.913) is reasonably good, confirming the quality of our classifier.

These results open many perspectives. A first idea is to further improve classifier quality by optimizing more parameters with the genetic algorithm. In the same way, we also plan to study if the performance of the genetic algorithm itself can be further improved. Another perspective is to evaluate the robustness and genericity of this kind of approach, when applied to larger sets, possibly mixing different programming languages and technologies. Testing the performance of our solution on other ticket classification problems would also be very interesting, as ticket binary classification for other kinds of issues or multi-class classification such as bug sub-category prediction. A practical perspective is to experiment the use of automatic classification as an assistant during ticket redaction by developers. As the performance of automatic classification is high (at least for bugs), a misclassification could indicate an unclear content to be improved. First experiments on a prototype bug ticket writer assistant have been made and are promising. A first version of such an assistant is available online.[5]

REFERENCES

[1] H. M. Sneed, "A cost model for software maintenance & evolution," in *20th International Conference on Software Maintenance*. Chicago, USA: IEEE, September 2004, pp. 264–273.

[2] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Business School Technical Report*, 2013.

[3] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *OOPSLA workshop on Eclipse Technology eXchange*. San Diego, USA: ACM, October 2005, pp. 35–39.

[4] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *35th International Conference on Software Engineering*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. San Francisco, USA: IEEE, May 2013, pp. 392–401.

[5] A. Kukkar, R. Mohana, A. Nayyar, J. Kim, B. Kang, and N. K. Chilamkurti, "A novel deep-learning-based bug severity classification technique using convolutional neural networks and random forest with boosting," *Sensors*, vol. 19, no. 13, p. 2964, 2019.

[6] N. Limsettho, H. Hata, and K. Matsumoto, "Comparing hierarchical dirichlet process with latent dirichlet allocation in bug report multiclass classification," in *15th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. Las Vegas, USA: IEEE, June 2014, pp. 1–6.

[7] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization," in *16th International Conference on Software Engineering & Knowledge Engineering*, F. Maurer and G. Ruhe, Eds., Banff, Canada, June 2004, pp. 92–97.

[8] S. Mani, A. Sankaran, and R. Aralikatte, "Deeptriage: Exploring the effectiveness of deep learning for bug triaging," in *ACM India Joint International Conference on Data Science and Management of Data*, R. Krishnapuram and P. Singla, Eds. Kolkata, India: ACM, January 2019, pp. 171–179.

[9] P. Terdchanakul, H. Hata, P. Phannachitta, and K. Matsumoto, "Bug or not? bug report classification using n-gram IDF," in *IEEE International Conference on Software Maintenance and Evolution*. Shanghai, China: IEEE, September 2017, pp. 534–538.

[5]http://ec2co-ecsel-1iegdism3qjis-2023048106.eu-west-3.elb.amazonaws.com/

[10] I. Chawla and S. K. Singh, "An automated approach for bug categorization using fuzzy logic," in *8th India Software Engineering Conference*, S. Padmanabhuni, R. Nambiar, P. T. Devanbu, M. K. Ramanathan, and A. Sureka, Eds. Bangalore, India: ACM, February 2015, pp. 90–99.

[11] N. Pingclasai, H. Hata, and K. Matsumoto, "Classifying bug reports to bugs and other requests using topic modeling," in *20th Asia-Pacific Software Engineering Conference*, P. Muenchaisri and G. Rothermel, Eds., vol. 2. Bangkok, Thailand: IEEE, December 2013, pp. 13–18.

[12] B. Luaphol, B. Srikudkao, T. Kachai, N. Srikanjanapert, J. Polpinij, and P. Bheganan, "Feature comparison for automatic bug report classification," in *15th International Conference on Computing and Information Technology*. Bangkok, Thailand: Springer, July 2019, pp. 69–78.

[13] N. Pandey, D. K. Sanyal, A. Hudait, and A. Sen, "Automated classification of software issue reports using machine learning techniques: an empirical study," *Innovations in Systems and Software Engineering*, vol. 13, no. 4, pp. 279–297, 2017.

[14] H. Qin and X. Sun, "Classifying bug reports into bugs and non-bugs using LSTM," in *10th Asia-Pacific Symposium on Internetware*. Beijing, China: ACM, September 2018, pp. 20:1–20:4.

[15] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the Scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[16] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, 1972.

[17] A. J. Ko, B. A. Myers, and D. H. Chau, "A linguistic analysis of how people describe software problems," in *IEEE Symposium on Visual Languages and Human-Centric Computing*. Brighton, UK: IEEE, September 2006, pp. 127–134.

[18] J. H. Holland, "Genetic algorithms," *Scientific American*, vol. 267, no. 1, pp. 66–73, 1992.

[19] K. A. D. Jong and W. M. Spears, "Using genetic algorithms to solve NP-complete problems," in *3rd International Conference on Genetic Algorithms*, J. D. Schaffer, Ed. Fairfax, USA: Morgan Kaufmann, June 1989, pp. 124–132.

[20] N. M. Razali, J. Geraghty *et al.*, "Genetic algorithm performance with different selection strategies in solving tsp," in *World congress on engineering*, vol. 2, no. 1. London, UK: International Association of Engineers Hong Kong, 2011, pp. 1–6.

[21] M. Gen, Runwei Cheng, and Dingwei Wang, "Genetic algorithms for solving shortest path problems," in *IEEE International Conference on Evolutionary Computation*, Indianapolis, USA, 1997, pp. 401–406.

[22] D. Thierens and D. Goldberg, "Elitist recombination: an integrated selection recombination GA," in *1st conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, vol. 1, Orlando, USA, 1994, pp. 508–512.

[23] W. M. Spears and V. Anand, "A study of crossover operators in genetic programming," in *Methodologies for Intelligent Systems*, Z. W. Ras and M. Zemankova, Eds. Springer, 1991, pp. 409–418.

[24] R. Kallis, A. Di Sorbo, G. Canfora, and S. Panichella, "Predicting issue types on GitHub," *Science of Computer Programming*, vol. 205, p. 102598, 2021.

[25] R. Kallis, A. D. Sorbo, G. Canfora, and S. Panichella, "Ticket Tagger: Machine learning driven issue classification," in *35th International Conference on Software Maintenance and Evolution*. Cleveland, USA: IEEE, September 2019, pp. 406–409.

[26] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing neural networks using genetic algorithms," in *3rd International Conference on Genetic Algorithms*, J. D. Schaffer, Ed. Fairfax, USA: Morgan Kaufmann, June 1989, pp. 379–384.

[27] L. D. Whitley, T. Starkweather, and C. Bogart, "Genetic algorithms and neural networks: optimizing connections and connectivity," *Parallel Computing*, vol. 14, no. 3, pp. 347–361, 1990.

[28] X. Yao and Y. Liu, "A new evolutionary system for evolving artificial neural networks," *IEEE Transactions on Neural Networks*, vol. 8, no. 3, pp. 694–713, 1997.

[29] F. H. Leung, H. Lam, S. Ling, and P. K. Tam, "Tuning of the structure and parameters of a neural network using an improved genetic algorithm," *IEEE Transactions on Neural Networks*, vol. 14, no. 1, pp. 79–88, 2003.

[30] M. Bashiri and A. F. Geranmayeh, “Tuning the parameters of an artificial neural network using central composite design and genetic algorithm,” *Scientia Iranica*, vol. 18, no. 6, pp. 1600–1608, 2011.