mathematics

MDPI

Article

Enhancing the Performance of Software Authorship Attribution Using an Ensemble of Deep Autoencoders

Gabriela Czibula *, Mihaiela Lupea and Anamaria Briciu

Department of Computer Science, Babeş-Bolyai University, 400347 Cluj-Napoca, Romania; mihaela.lupea@ubbcluj.ro (M.L.); anamaria.briciu@ubbcluj.ro (A.B.)
* Correspondence: gabriela.czibula@ubbcluj.ro or gabis@cs.ubbcluj.ro; Tel.: +40-264-405327

Abstract: Software authorship attribution, defined as the problem of software authentication and resolution of source code ownership, is of major relevance in the software engineering field. Authorship analysis of source code is more difficult than the classic task on literature, but it would be of great use in various software development activities such as software maintenance, software quality analysis or project management. This paper addresses the problem of code authorship attribution and introduces, as a proof of concept, a new supervised classification model *AutoSoft* for identifying the developer of a certain piece of code. The proposed model is composed of an ensemble of autoencoders that are trained to encode and recognize the programming style of software developers. An extension of the *AutoSoft* classifier, able to recognize an unknown developer (a developer that was not seen during the training), is also discussed and evaluated. Experiments conducted on software programs collected from the Google Code Jam data set highlight the performance of the proposed model in various test settings. A comparison to existing similar solutions for code authorship attribution indicates that *AutoSoft* outperforms most of them. Moreover, *AutoSoft* provides the advantage of adaptability, illustrated through a series of extensions such as the definition of class membership probabilities and the re-framing of the *AutoSoft* system to address one-class classification.

Keywords: software authorship attribution; natural language processing; deep learning

MSC: 68T07; 68T50

Citation: Czibula, G.; Lupea, M.; Briciu, A. Enhancing the Performance of Software Authorship Attribution Using an Ensemble of Deep Autoencoders. *Mathematics* **2022**, *10*, 2572. https://doi.org/10.3390/math10152572

Academic Editors: Liang Zou, Liang Zhao and Yonghui Xu

Received: 19 June 2022
Accepted: 22 July 2022
Published: 24 July 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Authorship attribution (AA), in its broad definition, is a field that has been extensively studied, as the problem of document authentication and resolution of text ownership disputes has been around for centuries. In recent decades, researchers have proposed a number of automatic authorship attribution systems based on machine learning and deep learning techniques, but the focus remained on the classic task based on literary texts. The authorship attribution systems can help solve plagiarism and copyright infringement disputes in both academic and corporate settings.

Code authorship identification, or software authorship attribution (SAA), is the process of identifying programmers based on their distinctive programming styles. Style is based on various factors, such as the programmer's preferences in the way to write code, naming of the variables, programming proficiency and experience, and the thinking process to solve a programming task [1].

While it is true that programming languages have much less flexible grammars than natural languages, it is widely accepted that a programmer's coding style can still be defined as referring to the tendencies in the expression of logic constructs, data structure definition, variable and constant names and calls to fixed and temporary data sets [2].

It is only recently that the field of source code authorship attribution has gained attraction. The growing interest in SAA is due to the practical needs in the academic, economic and societal fields. Plagiarism detection and ghostwriting (detection of outsourced student

programming assignments) are specific tasks solved with SAA in the academic field. In the cybersecurity domain, in which both individuals and organizations are targets, the cyber-attacks based on malicious software (adware, spyware, virus, worms, and more) are important issues that can be prevented with the help of SAA systems. The software engineering field benefits from SAA in solving different tasks such as software maintenance, software quality analysis, software release assessment [3], project management and plagiarism detection with important effects in the copyright and licensing issues [4].

As an important aspect in SAA, the data used for evaluation, Bogomolov et al. [4] investigate the limitations of the existing data sets. The very good results obtained for source codes from programming competitions, books and students' assignments decrease dramatically when code from real-world software projects is tested. The concept of *work context* that captures specific aspects of the software project (domain, team, internal coding conventions) was introduced and used in the evaluation of the performance of the authorship attribution models. In the same paper, a novel data collection technique has also been proposed with the aim of obtaining more realistic data, with multiple authors per project and different programming languages used. These data sets better reflect a real-world environment, but also need language-independent models to solve the SAA task.

In the deep learning literature, autoencoders [5] are powerful models applied in a variety of problems including image analysis [6], protein analysis [7,8], and speech processing [9]. Autoencoders (AEs) are formed by two neural networks (an encoder and a decoder) that are self-supervisedly trained to rebuild the input by approximating the identity function. The input is compressed by the encoder into a hidden (latent-space) representation, and then the decoder rebuilds the input from this representation. An autoencoder is trained to encode as much information as possible about the class of instances it was trained on.

In this paper, a supervised multi-class classification model, *AutoSoft*, for software authorship attribution is introduced as a proof of concept. The model is composed of an ensemble of autoencoders that are trained to encode and recognize the programming style of software developers. Subsequently, *AutoSoft* will predict the author of a certain source code fragment, based on the similarity between the given code and the information learned (through the autoencoders) about each software developer. We are exploiting the ability of autoencoders to encode, through their latent representations, patterns about the coding-style of specific software developers. In this proposal, the representation of the software programs is inspired from the Natural Language Processing (NLP) domain [10]. A program, processed as a text (a sequence of specific tokens), is represented as a distributed vector provided by a `doc2vec` model [11]. Experiments will be performed on software programs collected from an international programming competition organized by Google and previously used in the software authorship attribution literature. The obtained results empirically prove our hypothesis that autoencoders are able to capture, from a computational perspective, relevant knowledge about how developers are writing their code. An additional strength of *AutoSoft* is the fact that it can be extended to recognize not only the classes of the original authors (developers) on which it was trained, but an "unknown" class as well. *AutoSoft* solves the software authorship attribution task in a closed-set configuration, meaning that at the testing stage the classifier identifies the author of a source code from a set of given developers, whose programs were used in training. *AutoSoft* was extended to $AutoSoft^{ext}$ classifier with the aim of solving the multi-class classification task and the novelty detection task at the same time. It is an open-set recognition approach [12] to the SAA problem with an open testing space. Besides source codes authored by a set of known developers (used in training), other (novel) codes, written by unseen developers (unknown during training) should be classified at the testing time. The novelty detection refers to assigning a software program that was not written by a known author to the "unknown" class. As far as we are aware of, the approach proposed in this paper is new in the literature regarding software authorship attribution.

To summarize, the paper is focused towards answering the following research questions:

RQ1 How to design a supervised classifier based on an ensemble of autoencoders for predicting the software developer that is likely to author a certain source code, considering the encoded coding-style for the developers?
RQ2 Does the proposed classifier improve the software authorship performance compared to conventional classifiers from the machine learning literature?
RQ3 Could such a classification model that works in a closed-set configuration, be extended to work in an open-set configuration, with the aim not only to recognize the classes of developers it was trained on, but to detect an "unknown" class/developer as well?

The rest of the paper is structured as follows: Section 2 presents the relevance and difficulty aspects of the software authorship attribution (SAA) task in the software engineering field. Section 3 is dedicated to a literature review in the SAA domain. The detailed description of *AutoSoft*, the proposed deep autoencoder–based classification model for recognizing the developer of a software program, is the subject of Section 4. The experimental results and discussions are presented in Section 5. In Section 6, an extension of the *AutoSoft* classifier is proposed, with the aim of identifying "unknown" instances, software programs that are not authored by the developers on which *AutoSoft* has been trained. The threats to validity of our study are exposed in Section 7. Section 8 summarizes the main contributions of the paper and proposes directions for future work.

2. Problem Relevance and Difficulty

The software engineering field can benefit from developing efficient authorship attribution systems. For example, in terms of software maintenance, existing works investigate the possibility of automation in the assignment of developers to bugs in open source bug repositories [13] and identification of developers that are familiar with certain parts of code in a large project in order to make the process easier for both team members in understanding each other's work and for team leaders, when a new team member needs to be brought up to speed [14,15].

The aspect of code ownership is also examined with respect to software quality analysis. Bird et al. [16] defined measures of ownership related to software quality and explored their effect on pre-release and post-release defects in two large industrial software projects. Some works also argue for including reviewing activity when tracking code ownership and establishing chain of responsibility [17]. The impact of code ownership and developer experience on software quality is examined by Rahman and Devanbu [18], with findings suggesting that the specialized experience of a developer with respect to a target file is more valuable than generalized experience with the project.

Code authorship attribution would also be relevant for project or team managers, by helping them in identifying the software developer who authored a certain piece of code. This way, a team manager could identify if the code submitted by a software developer in the team's source code management system is indeed authored by the developer, which is a closed-set approach to SAA. The testing space for an SAA tool can be opened, allowing to identify as authors of software programs not only the team's members but also unknown developers. This open-set recognition approach to SAA is also beneficial in the software management field. On the other hand, if a certain developer is prone to introduce bugs in its code, then SAA would allow more rigorous testing of software components and modules written by that developer and thus would reduce the risk of preserving software defects in the code.

Authorship analysis of source code is more difficult than the classic task on literature, for a number of reasons which include the restricted set of natural language stylistic characteristics that also apply to this type of text, and, from another perspective, code reuse, the frequent development of a program by a team of developers and not a single programmer, and the possibility that structural and layout characteristics may be altered by code formatters [19].

3. Literature Review

The section starts by reviewing the features and algorithms used by existing work in software authorship attribution. Then, Section 3.2 describes the Google Code Jam (GCJ) data set, whilst the software authorship attribution approaches that considered this data set are discussed in Section 3.3.

3.1. Features and Algorithms Used in the SAA Task

Early work in the field of software authorship analysis involved using typographic or layout characteristics of programs [20] to assess similarity between a series of authors. In addition, early focus was on research into software forensics, a branch of software authorship analysis that is not concerned with specifically identifying the author of a program but their features (e.g., preference for certain data structures, programming skill and level of expertise, formatting style, comment styles and variable name choice) [21]. One of the earliest attempts to identify the author of a program is that of Krsul and Spafford [19], who used a set of stylistic characteristics including layout (e.g., indentation, placement of brackets and comments), style (e.g., mean variable length, mean comment length) and program structure (e.g., lines of code per function, usage of data structures) metrics. They obtained relatively good results in an experiment with 29 authors and a total of 88 files (73% correctly identified instances) and provided an interesting discussion on classification results as related to programmer background.

Over the years, researchers have started investigating other sets of features besides stylistic ones, as these required a good deal of manual feature engineering, and no universally efficient set of features was discovered. Moreover, there are cases when the source code is not available, only the binary code. Nonetheless, Rosenblum et al. [22] provided evidence that programmer style survives the compilation process, and, given the right set of features, the task of programmer identification can be solved to a satisfying degree.

Recent work into source-code-based authorship identification is focused on lexical or syntactic features rather than format and layout ones, as they are more robust. In the case of lexical features, some approaches draw inspiration from authorship attribution tasks designed with natural language in mind. In particular, source code N-grams are used [23–25], with byte-level N-grams as the most frequent choice [26].

Syntactic features are generally based on Abstract Syntax Trees (ASTs) derived from source code and have proven successful in solving various authorship attribution tasks [27,28]. For SAA in different programming languages, Bogomolov et al. [4] proposed two language-agnostic models: Random Forest Model and Neural Network Model (`code2vec` [29]) using path-based representation of code generated from the AST. *CroLSSim* [30], a tool for detecting semantically related software across various programming languages was developed using AST-MDrep (Abstract Syntax Tree—Methods Description) features for codes and LSA (Latent Semantic Analysis) to reduce the high-dimensional space. Finally, some studies propose automatic feature learning using deep learning techniques [1].

Distributed representations of source code are less common in software-related tasks. A neural model for representing snippets of code as fixed length vectors was proposed and used to predict method names [29]. Mateless et al. [31] presented a technique to generate package embeddings and use the obtained representations in a task of authorship attribution with good results. Other existing work uses distributed representations of source code to recover problem statement to coded solution [32] or assess and review student assignments [33].

3.2. The Google Code Jam Data Set

GCJ [34] is an international programming competition organized by Google which requires contestants to solve a series of algorithmic problems over multiple rounds in a fixed amount of time. Contestants are free to use any programming language and development environment they wish. In every round, a small number of problems is given, usually between 3 and 6.

The Google Code Jam data set [35] is considered well-suited for the task of authorship attribution because each program is guaranteed to be authored by a single person and, moreover, it provides a collection of functionally equivalent programs solved by different authors [36]. However, the GCJ data set is also seen as artificial, as its context greatly differs from that of professionally developed software, in which multiple programmers work on the same project under clear style guidelines, usually for longer periods of time that involve revisions, refactorings and multiple project versions.

3.3. Related Work

An in-depth analysis of the SAA literature revealed various machine learning-based techniques developed for identifying the author (i.e., software developers) of a certain source code fragment. Supervised classifiers, ranging from classical to deep learning models, have been proposed and evaluated on the GCJ data set [1,22,27,28,36–41].

Rosenblum et al. [22] propose a binary code representation based on instruction-level and structural characteristics of programs, namely idioms (instruction sequence templates), subgraphs from the Control Flow Graph (CFG) of a program and byte N-gram features. The authors employ a feature selection step before using the representation to train a Support Vector Machine classifier that obtains good results on the 2009 and 2010 GCJ sets. In addition, 78% accuracy is achieved on the GCJ 2009 data set, 77% accuracy on the 2010 data set for 20 authors, and 51% on the GCJ 2010 data set for 191 authors.

Alrabaee et al. [39] build on the work of Rosenblum et al. [22], proposing a multi-layered approach to malware authorship attribution which incorporates preprocessing, syntax-based attribution and semantic-based attribution. The authors report superior results and provide more insight towards developing a formal definition of a programmer's style with respect to the features explored. Caliskan et al. [38] extend the idea of coding style and use a set of approximately 120,000 layout-based, lexical and syntactic features. On GCJ 2008–2014 subsets for C/C++ and Python programming languages, they obtain good performances. In particular, for Python, the authors experiment with a subset of 23 programmers, on which they achieve 87.93% accuracy, and a subset of 229 programmers, on which they achieve 53.91% accuracy (GCJ 2014). C/C++ classification is more successful, which the authors attribute to the choice of features that do not translate well to the Python programming language. However, they also provide scores obtained for top-5 authors classification (an instance is considered correctly classified if the true author is in the first five identified authors), a task which generates better results: 99.52% accuracy for 23 programmers and 75.69% for 229 programmers. Reducing the set of candidates to a small number of candidate authors may be a more feasible task in a real-world setting; to this point, *AutoSoft* (the classifier proposed in this paper) provides the possibility of computing the probability of a test instance to belong to a certain class.

The authors of Python codes, solutions in GCJ 2018 and 2020, have been successfully identified by Frankel and Ghosh [41] (the best accuracy in different testing configurations was between 90% and 100%) based on a combination of AST features and N-gram data, using Logistic Regression and a deep learning approach.

More recently, researchers have focused on deep learning models such as Long Short-Term Memory (LSTM) and Bidirectional Long Short-Term Memory Networks (BiLSTM) [28,37] or Convolutional Neural Networks (CNN) [1] to solve the software authorship problem with very good results. Alsulami et al. [28] use LSTM and and BiLSTM networks with AST-based features in a classification task involving two Python GCJ data sets with 25 and 70 authors, respectively. They obtain 96% accuracy for the 25 authors data set and an accuracy of 88.86% for the 70 authors data set. The authors in [37] use deep representations of TF-IDF features obtained using LSTM and GRU networks with a Random Forest classifier in large-scale experiments on GCJ 2008–2016 subsets involving multiple programming languages. For Python, they achieved 100% accuracy for 100 programmers, 98.92% for 150 programmers and 94.67% for 2300 programmers. Another work [1] exploited TF-IDF and word embedding representations, but in conjunction with Convolutional Neural Networks (CNNs). For a GCJ 2008–2016 Python data set, they obtain between 72–98.8%

accuracy for 150 programmers and between 62.66–94.6% accuracy for 1500 programmers in a series of experiments in which they the vary network architecture and the type of representation used.

4. Methodology

In this section, the *AutoSoft* classification model for software authorship attribution is proposed, with the goal of answering RQ1. An ensemble of deep AEs (one AE for each author/developer) is used to learn and encode the most relevant characteristics (both structural and conceptual) of the software programs developed by the same author/developer. Based on the low-dimensional latent representations of the programs, provided by AEs, the classifier will be able to distinguish different authors. In the training stage, each of the AEs is trained on the software programs written by a certain author/developer. In the classification stage, a new software program sp will be assigned to the author/developer corresponding to the autoencoder A if sp is highly similar to the information encoded by A and dissimilar to the information encoded by the other autoencoders.

The task of software authorship attribution may be modelled as a multi-class classification problem. The set of classes is a set of authors/developers $\mathcal{DEV} = \{Dev_1, Dev_2, \ldots, Dev_n\}$. The input instances are software programs from the set $\mathcal{SSP} = \{sp_1, sp_2, \ldots, sp_r\}$, written by the given software developers. The *AutoSoft* classifier will be trained on the set $\mathcal{SSP}$ of software programs labeled with their author/developer and will learn to predict the most likely software developer, $dev \in \mathcal{DEV}$, who authored a software program sp^{new} unseen during training.

From a machine learning perspective, the goal is to approximate a target function $f : \mathcal{SSP} \rightarrow \mathcal{DEV}$ that maps a software program sp from $\mathcal{SSP}$ to a certain class/developer $dev \in \mathcal{DEV}$.

The architecture of the *AutoSoft* classification model is depicted in Figure 1. The main stages of the proposed approach are: data preprocessing and representation, training and evaluation and they will be detailed in the next sections.

4.1. Data Preprocessing and Representation

In this stage, the software programs will be preprocessed by a lexical analyzer and then distributed vector representations of the programs will be generated.

One of the most efficient representations of variable-length pieces of text (sentences, paragraphs, documents) in Natural Language Processing tasks is generated by the `doc2vec` (*Paragraph vector*) model [11]. The Authorship Attribution (AA) task for poetry was addressed in the paper [42] using `doc2vec` representation of poems and a deep autoencoder–based classification model. These distributed representations provided features that discriminated very well between the poems' authors. However, Natural Languages (NL) are more complex than Programming Languages (PL) in all three aspects: vocabulary, syntax and semantics. At the lowest preprocessing level, the lexical level, both a natural language document and a software program, can be considered texts composed of specific tokens: words, punctuation marks in NL and keywords, operands, constants and variables in PL.

In the current approach, a text representation of a software program, provided by the `doc2vec` model, is proposed. The `doc2vec` model [11] consists of a simple neural network with one hidden layer. This neural network is designed to solve word prediction tasks; however, the goal is not to definitively solve these tasks, but to learn fixed-length dense vector representations for documents during the training of these networks. Two `doc2vec` models are available: Distributed Memory (PV-DM), and Distributed Bag of Words (PV-DBOW). In the PV-DM model, the paragraph vector is concatenated or averaged with a series of word vectors, representing the context, with the paragraph vector asked to contribute to the task of predicting the next word in that context. Thus, through this task, the paragraph vector is learned along with the word vectors. In contrast, in the PV-DBOW model, the paragraph vector is trained to predict words (or tokens) in small windows randomly sampled from that paragraph.

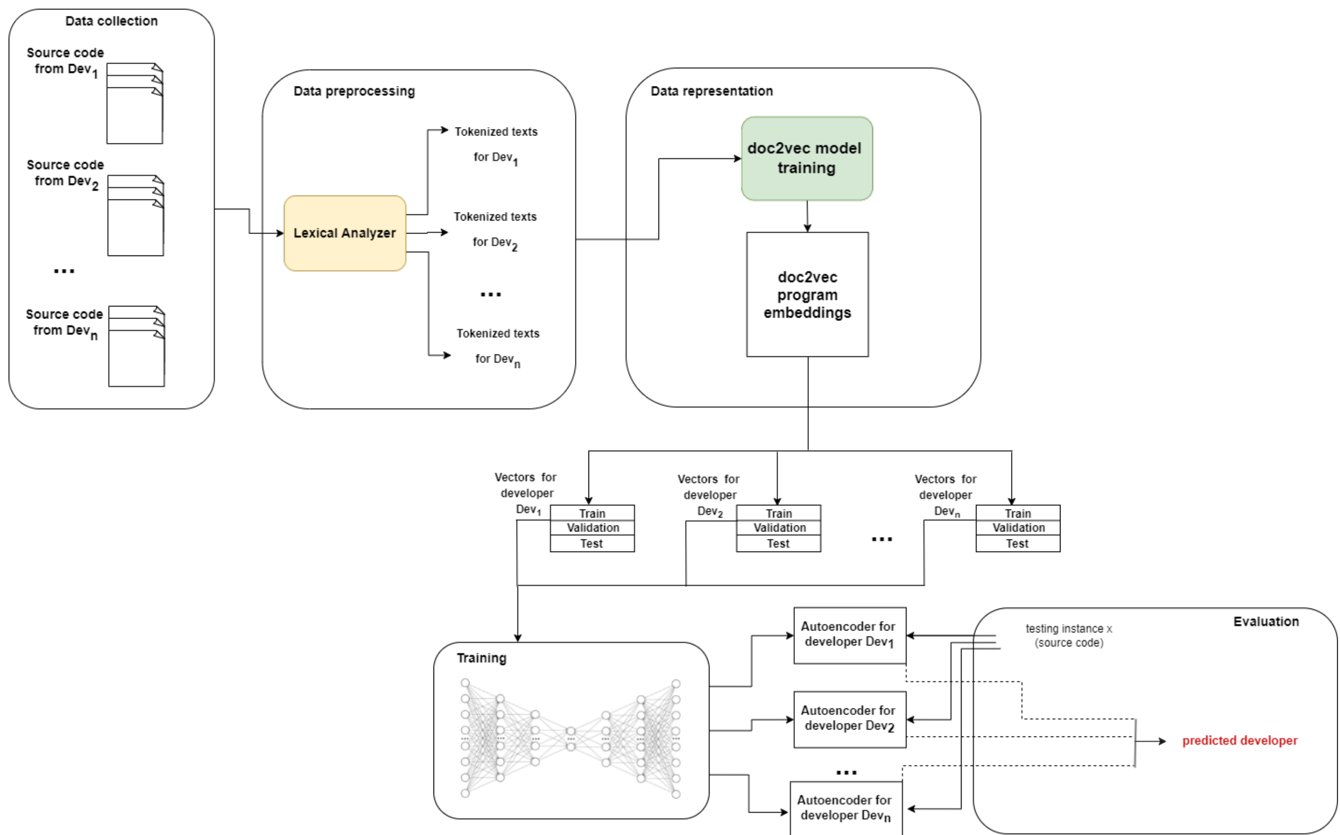

Figure 1. *AutoSoft* classification model.

In the first step, by applying a lexical analyzer for the specific programming language, a list of tokens will be identified for a software program. Operators, keywords, variable names and literals are considered tokens, while comments in natural language are excluded.

The `doc2vec` model is trained on a corpus of software programs, considering different sequences of N-grams of tokens to capture syntactic patterns in the codes. The input to the model therefore consists of a list of sequential token N-grams. For instance, from the following line of code: `for` i `in range(5)` and $N = 3$, the list generated and used as input to the `doc2vec` model is [`for` i `in`, i `in range`, `in range (`, `range ( 5`, `( 5 )`]. From this representation of the raw source code, hidden features of the programs are learned and expressed as numerical values in fixed dimensional vectors. Based on the learned features, similarities between programs (as documents) can be calculated in the latent dimensional space generated for the corpus of programs (as documents). After training, for a software program, the model infers a vector that is a distributed representation, called *program embedding*. If *sp* is a software program and *embed* the inferring function corresponding to the `doc2vec` model, the program embedding of sp is denoted by pe, where $pe = embed(sp)$. The program embeddings are the input data for autoencoders.

4.2. Training

The proposed classification model, *AutoSoft*, for software authorship attribution is an eager inductive learning model that will be built during training (through induction) and subsequently it will be applied (through deduction) on a testing set in order to evaluate its predictive performance.

The *AutoSoft* classifier is based on n autoencoders $A_1, A_2, \ldots, A_n$, corresponding to the developers $Dev_1, Dev_2, \ldots, Dev_n$. Let us denote by S_i the set of programs written by the developer Dev_i (i.e., $\mathcal{SSP} = \bigcup_{i=1}^{n} S_i$). A self-supervised training of A_i on the embeddings of the software programs from S_i is performed for each autoencoder. Thus, through its

latent state representation, A_i will unsupervisedly learn features relevant for developer Dev_i which will be useful in discriminating among different authors (developers).

A series of autoencoder architectures were examined in the experimental step. The best results for input vectors of size 150 and 300 were obtained for an architecture which consists of an input layer with a number of neurons equal to the dimensionality of the input data, followed by eleven hidden layers. There are five hidden layers with 128, 32, 16, 8, and 4 neurons, two neurons on the encoding layer and five symmetric hidden layers for decoding. Previous work [42] supports the idea that, for learned `doc2vec` feature vectors, regardless of the type of source texts, the best results are obtained by performing an initial sudden reduction of the input dimensions followed by a gradual reduction to a two-dimensional encoding.

As far as the activation function is concerned, for all the hidden layers, the ReLU activation function [5] is used, except for the encoding layer, for which linear activation is used. With the linear activation function, the output size is equal to the input size (i.e., m neurons). Stochastic gradient descent enhanced with the Adam optimizer [5] is used to train the network. A minibatch perspective is employed, and an early stopping criterion based on the convergence on the validation set loss is used.

4.3. Evaluation

After the *AutoSoft* classifier is trained (see Section 4.2), it is tested in order to evaluate its performance. For testing 10% from each data set, S_i ($\forall 1 \leq i \leq n$) is used, i.e., 10% software programs (taken for each developer Dev_i) which were unseen during training.

4.3.1. Classification

During the classification stage, when a new software program sp has to be classified, *AutoSoft* searches for the autoencoder A that maximizes the similarity between the program embedding $pe = embed(sp)$ and $\widetilde{pe}_A$ (the vector reconstructed by the autoencoder A for the input pe). The similarity between two software programs sp_1 and sp_2 represented by their program embeddings $pe_1 = embed(sp_1)$ and $pe_2 = embed(sp_2)$ is expressed as $sim(sp_1, sp_2) = sim(pe_1, pe_2) = cos(pe_1, pe_2)$, which is the cosine similarity between the vectors pe_1 and pe_2, scaled to [0, 1]. If the input program embedding pe of sp is the most similar to $\widetilde{pe}_{A_i}$ (its reconstruction provided by the autoencoder A_i), it is very likely that sp has a high structural and conceptual similarity to the information encoded by A_i and thus it is highly probable to be authored/developed by Dev_i.

At the decision level, for each testing instance sp, *AutoSoft* determines the probabilities $p_1(sp)$, $p_2(sp), \ldots, p_n(sp)$, where $p_i(sp)$ represents the probability that the software program sp belongs to class Dev_i, where $1 \leq i \leq n$. Let us denote by $sim(pe, \widetilde{pe}_{A_i})$ the similarity between the embedding $pe = embed(sp)$ and its reconstruction, $\widetilde{pe}_{A_i}$, through the autoencoder A_i. The probabilities $p_i(sp), \forall i \in \{1, 2, \ldots, n\}$ are defined in Formula (1):

$$p_i(sp) = \frac{e^{sim(pe, \widetilde{pe}_{A_i})}}{\sum_{j=1}^{n} e^{sim(pe, \widetilde{pe}_{A_j})}} \tag{1}$$

The probability $p_i(sp)$ is positively correlated with $sim(pe, \widetilde{pe}_{A_i})$. Thus, $\arg\max_{i=1,n} p_i(sp) = \arg\max_{i=1,n} sim(pe, \widetilde{pe}_{A_i})$, meaning that an instance sp, with pe as its program embedding, will be classified by *AutoSoft* as being written by the author Dev_k such that $k = \arg\max_{i=1,n} sim(pe, \widetilde{pe}_{A_i})$.

4.3.2. Experimental Methodology

For testing, a cross-validation methodology is employed to precisely evaluate the performance of the proposed model and account for randomness in the selection of data. The training/validation/testing split is repeated 10 times.

The performance of the *AutoSoft* classifier on a given testing set is evaluated by first determining *Precision* (denoted by $Prec_i$), *Recall* ($Recall_i$) and *F*1-*score* ($F1_i$) values for the developer classes Dev_i, $i \in \{1, 2, \ldots, n\}$. The *F*1-*score* value is computed as the harmonic mean between the *Precision* and *Recall* values [43]. For a developer class Dev_i, the *F*1-*score* is calculated using Formula (2):

$$F1_i = \frac{2 \cdot Prec_i \cdot Recall_i}{Prec_i + Recall_i} \tag{2}$$

As the data sets S_i used are imbalanced (see Section 5.1), we use aggregated *Precision*, *Recall* and *F*1 measures, which are computed as the weighted averages of the $Prec_i$, $Recall_i$, $F1_i$ values obtained for the classes. These measures are defined in the Formulas (3)–(5), where w_i represents the cardinality of S_i:

$$Precision = \frac{\sum_{i=1}^{n}(w_i \cdot Prec_i)}{\sum_{i=1}^{n} w_i} \tag{3}$$

$$Recall = \frac{\sum_{i=1}^{n}(w_i \cdot Recall_i)}{\sum_{i=1}^{n} w_i} \tag{4}$$

$$F1 = \frac{\sum_{i=1}^{n}(w_i \cdot F1_i)}{\sum_{i=1}^{n} w_i} \tag{5}$$

These aggregated *Precision*, *Recall* and *F*1 values are averaged over the 10 runs of the cross-validation process. The 95% confidence interval [44] is computed for the mean values.

5. Experimental Results

This section presents the experiments performed for assessing the performance of the *AutoSoft* classifier, together with the results obtained and their analysis.

5.1. Data Description and Analysis

The experiments for evaluating the performance of our *AutoSoft* classification model were conducted on a subset of Python programs from the 2008–2020 GCJ data set.

We have identified 16,112 distinct authors that have written at least nine programs in Python for the Google Code Jam challenges from 2008 to 2020. From these, for the initial experiments, the most proficient 87 programmers have been selected. The experiments are focused on the subsets of 5 developers (each with more than 200 files), 12 developers (each with more than 150 files) and 87 developers (each with more than 100 files).

Data set statistics for these subsets are provided in Table 1. For preprocessing the source codes, a Python lexical analyzer [45] was employed and the library *gensim* [46] was used for the `doc2vec` model.

Table 1. Data set description.

	Subset		
	5 Developers	**12 Developers**	**87 Developers**
No. of files per developer	≥200	≥150	≥100
Total. no. files	1132	2357	11,089
Total. no. tokens	799,824	1,395,560	4,563,661
Median tokens per file	378.5	386	309
Median lines per file	61	65	52
Avg. no. tokens per file	706.56	592.09	411.55
Avg. no. lines per file	60.95	75.51	61.43

The average number of tokens per line ranges between 5 and 9 for 86 out of 87 authors, with an author accruing a mean of 26.89 tokens per line due to some hard-coded lists of values with many elements.

As it can be seen in Figure 2, attributing authorship of source code among five users is a fairly simple task, with a representation based on unigrams managing to separate author instances very well. We use the *difficulty* measure [47] to express how difficult it is to classify the instances from a labeled data set. The overall classification difficulty is computed as the percentage of instances from the data set for which the nearest neighbour (excluding the class label when computing the similarity) has a different label than its true one. We define a classification difficulty measure for each developer class that is equal to the percentage of instances from the given developer that have a nearest neighbor (computed using the cosine similarity between their program embeddings) an instance pertaining to a different developer class. Consequently, it follows that classes which have a higher value of difficulty present more challenges when classifiers attempt to identify them.

The difficulties presented below are calculated considering `doc2vec` vector size of 300. Computing the difficulty of the classification task for each author in the five authors setting confirms the low complexity of the overall task, even when using unigram representation: the highest classification difficulty is obtained for developer Dev_4, and is 0.022, and the overall difficulty of the classification task is 0.0129.

Increasing the number of authors expectedly increases the difficulty of the task as well, which can be seen in Figure 3. Some authors in this subset (12 authors) may be easily distinguished, while for others it is difficult to find clear margins of separation. Figure 4 shows how the classification task difficulty varies with the value of N in N-gram size. For the majority of developers (10 out of 12), using unigram features for representation makes the classification of their instances more difficult than any other representation with $1 < N \leq 8$. Generally, minimum difficulty is obtained for N between 3 and 6. Therefore, we expect that these values for N will generate better results in the overall classification task than the basic unigram-based representation.

Figure 5 shows the value ranges for the classification difficulties with unigram representation and 5-grams representation for the subset of 87 developers. For unigram representation, there are 42 programmers with the classification difficulty ranging between 0 and 0.05, while, for 5-gram representation, there are 61 developers for which the classification difficulty falls into this category. Similarly, in the 5-gram representation, the computed classification difficulty for most programmers ranges between 0 and 0.2, with only three developers for which a difficulty between 0.3 and 0.4 is obtained ([0.3–0.35) for two programmers and [0.35–0.4) for one programmer). In contrast, for unigram representation, the classification difficulty reaches 0.51, with 13 developers having a difficulty greater than 0.2 ([0.2–0.25) for 5 programmers, [0.25–0.3) for 2 programmers, [0.3–0.35) for 1 programmer, [0.35–0.4) for 2 programmers, and 1 programmer for each of the intervals [0.4–0.45), [0.45–0.5) and [0.5–0.55)).

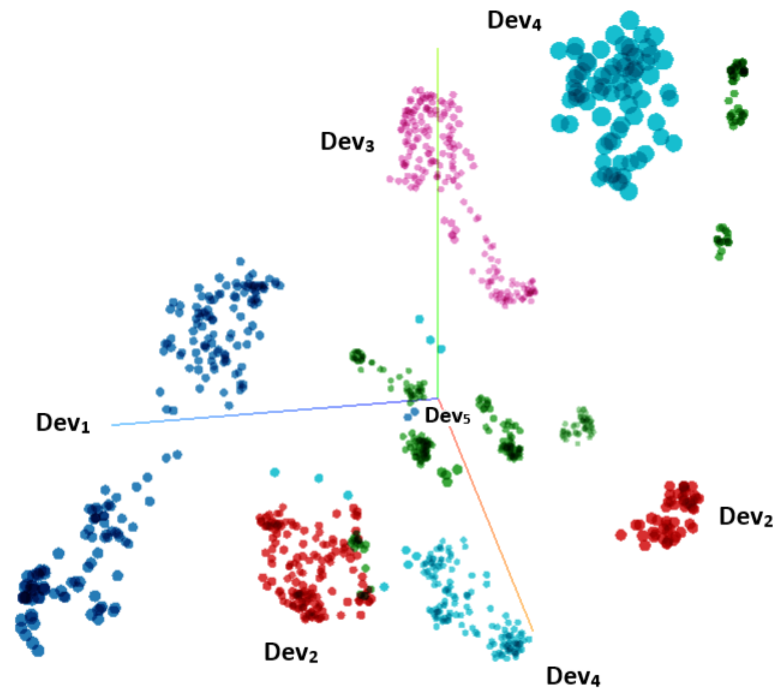

Figure 2. T-SNE visualization for the subset of five developers, unigram features and doc2vec vector size of 300.

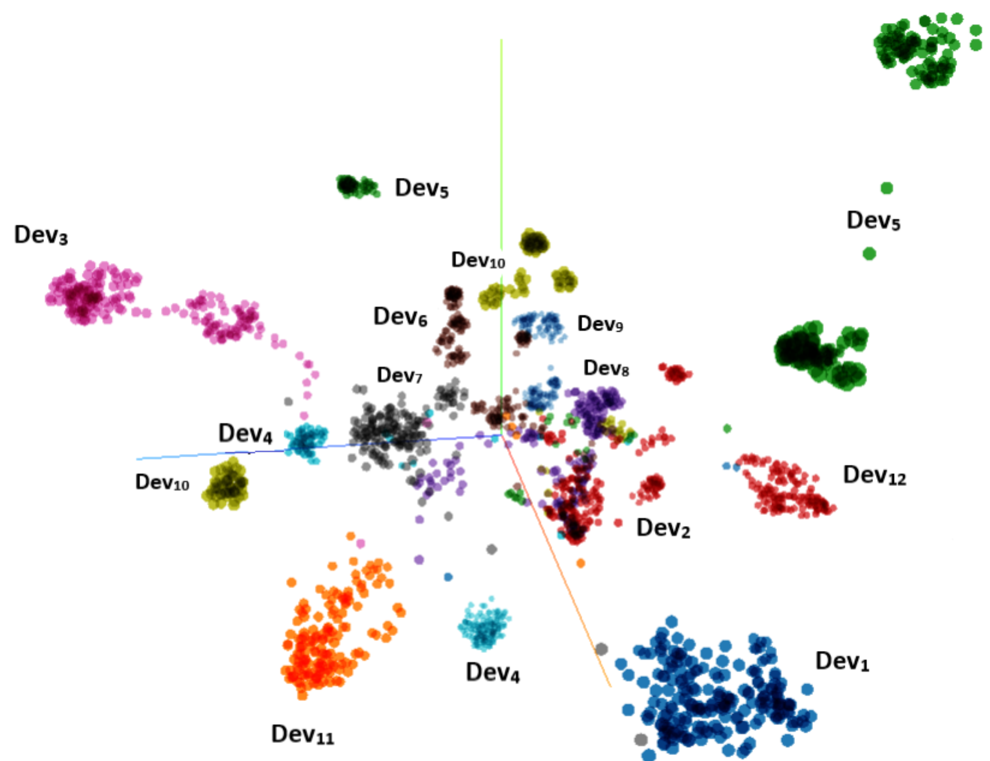

Figure 3. T-SNE visualization for the subset of 12 developers, unigram features and doc2vec vector size of 300.

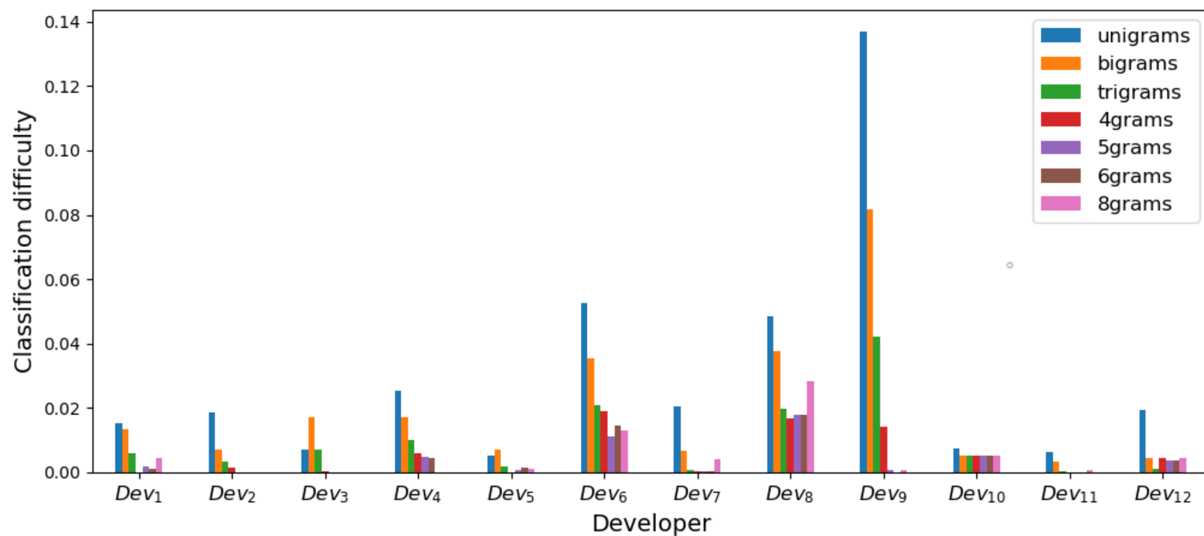

Figure 4. Classification difficulty for 12 developers with respect to N-gram size for doc2vec vector size of 300.

For all subsets, the overall classification task difficulty is lower for N-gram size with $N > 1$, indicating that coding style is better captured by learning distributed representations over sequences of tokens, rather than unigrams.

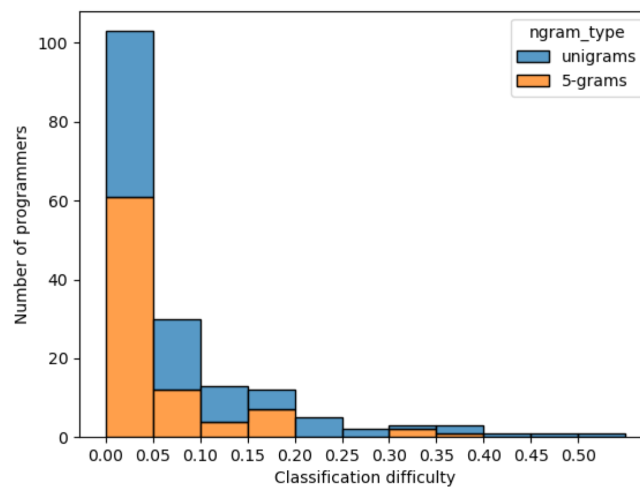

Figure 5. Histogram of classification difficulty for 87 developers and *unigram* features (blue) and *5-gram* features (yellow) and `doc2vec` vector size of 300.

5.2. Results and Discussion

The experimental results are further presented and discussed.

5.2.1. Results

Table 2 illustrates the performance of *AutoSoft*, in terms of *Precision*, *Recall* and *F1-score*, for `doc2vec` vectors of sizes 150 and 300, as well as five different N-gram sizes, in the task of differentiating between 5, 12 and 87 developers. We have experimentally determined that the best performing `doc2vec` model on the considered data set is PV-DBOW, with the specification that word-vectors are also learned in a skip-gram fashion [48]. Including this step leads to better document representations than using the default randomized word embeddings, becoming easier to learn the document embedding in such a way that it is close to its more critical content words [49].

The values depicted in the table represent the average performance obtained during the cross-validation, together with the 95% confidence interval of the mean.

For the subset of five developers, similar results are obtained for all N-gram sizes, with any $N > 1$ generating a slight increase in performance when 150 features are considered. For doc2vec vectors of size 300, the lowest number of misclassified instances is obtained when using 8-gram models.

For 12 authors and a vector size of 150, the models trained on 5-gram vectors perform better than other types of N-grams. For a vector size of 300, the N-gram size that generates the best results is $N = 3$ (see Table 2).

Table 2. *AutoSoft* results with respect to N-gram size for subsets of 5, 12 and 87 developers. In addition, 95% confidence intervals are used for the results.

	Number of Features	Performance Measure	*N*-Gram Size				
			1	**3**	**5**	**6**	**8**
5 developers	150	*Precision*	0.984 ± 0.008	**0.993** ± 0.004	0.989 ± 0.007	0.988 ± 0.006	0.988 ± 0.005
		Recall	0.982 ± 0.009	**0.993** ± 0.004	0.988 ± 0.009	0.987 ± 0.008	0.988 ± 0.005
		*F*1	0.983 ± 0.008	**0.993** ± 0.004	0.988 ± 0.009	0.987 ± 0.008	0.988 ± 0.005
	300	*Precision*	0.986 ± 0.007	0.984 ± 0.006	0.985 ± 0.007	0.991 ± 0.007	**0.992** ± 0.005
		Recall	0.986 ± 0.007	0.98 ± 0.008	0.985 ± 0.007	0.991 ± 0.007	**0.992** ± 0.005
		*F*1	0.986 ± 0.007	0.98 ± 0.005	0.985 ± 0.007	0.991 ± 0.007	**0.992** ± 0.005
12 developers	150	*Precision*	0.968 ± 0.006	0.98 ± 0.005	**0.984** ± 0.005	0.98 ± 0.007	0.973 ± 0.006
		Recall	0.966 ± 0.007	0.979 ± 0.005	**0.982** ± 0.006	0.978 ± 0.007	0.97 ± 0.007
		*F*1	0.966 ± 0.007	0.979 ± 0.005	**0.982** ± 0.006	0.978 ± 0.007	0.97 ± 0.007
	300	*Precision*	0.977 ± 0.007	**0.984** ± 0.007	0.98 ± 0.004	0.979 ± 0.005	0.978 ± 0.008
		Recall	0.975 ± 0.007	**0.981** ± 0.008	0.978 ± 0.005	0.977 ± 0.006	0.977 ± 0.008
		*F*1	0.975 ± 0.007	**0.981** ± 0.008	0.979 ± 0.005	0.977 ± 0.006	0.977 ± 0.008
87 developers	150	*Precision*	0.882 ± 0.004	0.892 ± 0.004	**0.913** ± 0.004	0.911 ± 0.004	0.906 ± 0.006
		Recall	0.868 ± 0.005	0.88 ± 0.004	**0.901** ± 0.005	0.899 ± 0.004	0.895 ± 0.007
		*F*1	0.866 ± 0.005	0.88 ± 0.004	**0.898** ± 0.005	0.896 ± 0.004	0.889 ± 0.008
	300	*Precision*	0.913 ± 0.003	0.918 ± 0.006	**0.922** ± 0.004	0.914 ± 0.004	0.904 ± 0.007
		Recall	0.902 ± 0.003	0.911 ± 0.005	**0.913** ± 0.004	0.905 ± 0.006	0.894 ± 0.007
		*F*1	0.902 ± 0.003	0.909 ± 0.007	**0.913** ± 0.004	0.904 ± 0.005	0.89 ± 0.005

The benefit of employing N-grams becomes clear in the more difficult task of distinguishing between 87 developers. In this case, the best value for N is 5 for both vector sizes considered. Increasing the value of N beyond 5 proves to be disadvantageous, with scores starting to decrease.

Taking into consideration these results, we conclude that (1) using doc2vec models trained on N-grams rather than simple unigrams is beneficial, with better results obtained for any $N > 1$ in all classification settings, and (2) no particular N-gram size is most advantageous in all cases.

Intuitively, it may be argued that N-gram size and the number of doc2vec features should be inversely proportional; i.e., increasing the size of one provides sufficient additional information and so increasing the size of the other becomes ineffective. However, this does not always hold. For instance, the results on the five developers subset show that the best choices are $N = 3$ with a vector size of 150 and $N = 8$ with a vector size of 300. Therefore, the ideal N-gram size is heavily dependent on the considered data set. This observation is also supported by the results obtained on the 87 developers subset, where $N = 5$ generates the best results for both vector sizes.

Burrows and Tahaghoghi [23] identify the values of 6 and 8 for N as the most effective in a task of source code authorship attribution that uses similarity measures and document ranking techniques to attribute authorship. In our experiments, generally, a value of 5 for N is the most advantageous, but there are certain subsets of authors that are better

distinguished by 3-grams or 8-grams. However, this is not a limitation of using an N-gram approach, since a grid search procedure may be used for determining the optimal value for N.

5.2.2. Discussion

The *AutoSoft* model has been proposed in this paper for software authorship attribution and experimentally evaluated in Section 5. The software programs were preprocessed by a lexical analyzer to express them as sequences of specific tokens and then distributed vector representations of the programs were generated using the `doc2vec` model [11] trained on a corpus of software programs, considering different N-grams of tokens. The underlying idea behind employing the paragraph vector model for obtaining program embeddings is that the `doc2vec` model captures high level semantic information, with the use of N-grams providing additional information about preferred syntactic patterns. Such a representation is very likely to be able to capture features about the code structure and organization such as code reorderings, source code control, and data dependencies.

`Doc2vec` models have been previously employed in the search-based software engineering literature [50,51] for unsupervisedly learning conceptual-based features from the source codes further used to express the semantics of the source code. Unlike the classical NLP models, `doc2vec` also considers the semantic distance between words/tokens [52] (e.g., *public* will be closer to *protected* than to *integer*) and the words/tokens order in small contexts. Thus, the vector representation of the source code provided by a `doc2vec` model trained on token N-grams (with various sizes) is likely to capture several important aspects related to the programming style in which software developers write their source codes. Few examples of the developer's programming style that may be captured are as follows: (1) the preference of a developer for iterating the elements from a data container, such as using an iterator, a range-based `for` statement or a `for` loop with indexes; (2) the preference of the developer for writing the code in a specific programming paradigm such as imperative or declarative (functional, logic) or object-oriented; (3) the preference for writing the code in an iterative or recursive manner; (4) the preference for specific data types or software libraries. Other methods able to capture more complex information about the source code such as `code2vec` [29], AST, Dependency Graphs will be further envisaged.

In order to highlight the performance of *AutoSoft* with respect to existing classification models (in the SAA and supervised learning domains) and to answer the research question RQ2, we decided to perform a comparison in two directions.

First, the performance of *AutoSoft* is compared to those of similar classifiers developed in the literature for code authorship attribution. An exact comparison between *AutoSoft* and the related work described in Section 3.3 cannot be made since the data sets, the evaluation methodology and the performance evaluation metrics used in the previously published papers differ from ours. Most of the related approaches provide only the accuracy for performance evaluation, despite the imbalanced nature of the classification. However, considering only the magnitude of the performance metrics presented in Table 3, the best *F*1-*score* values provided by *AutoSoft* (ranging between 0.902 and 0.986) outperform most of the performances reported in the literature for the SAA task (accuracy values ranging from 0.51 to 1).

Secondly, we decided to compare *AutoSoft*'s performance, in terms of *F*1-*score*, with the performance of four well-known classifiers from the classical supervised learning literature. The classifiers used for comparison were selected based on the learning paradigm and are known in the literature to have very good predictive performances: *Support Vector Classifiers* (SVC) are based on the statistical learning theory and are large margin classifiers; *Random Forests* (RF) are ensemble learners based on the bagging paradigm, among the best classifiers from classical machine learning; *Gaussian Naive Bayes* (GNB) are classifiers from the Bayesian learning framework, able to handle continuous data and with a good performance on text classification; *k-Nearest Neighbors* (kNN) are classifiers from the lazy learning paradigm. The experiments conducted for evaluating *AutoSoft* and the other

four classifiers are performed on the same GCJ data set, using the same experimental methodology and evaluation measures (described in Section 4.3.2).

For implementing SVC, RF, GNB and kNN classifiers, the Python language and the *scikit-learn* [53] machine learning library were used. For all the algorithms used in the comparison, a grid search was applied for determining the optimal parameters.

Table 3 presents the comparison between *AutoSoft* and classifiers from the literature in terms of *F*1-*score*, for various numbers of authors (5, 12, 87), various types of features (unigrams, trigrams, 5-grams) and using distributed vector representations of size 300. For all classifiers, the mean *F*1-*score* for the ten runs of the cross-validation is computed. The best performance in each evaluation is highlighted. To summarize the results of the comparison, we expressed in the last rows from Table 3 three values: (1) the number of evaluations in which the performance of *AutoSoft* is higher than the performance of other classifiers (denoted by WINS); (2) the number of evaluations in which *AutoSoft* is surpassed (in terms of *F*1-*score*) by other classifiers (denoted by LOSSES); and (3) the number of evaluations in which *AutoSoft* has the same performance as another classifier (denoted by DRAWS). One observes that, in 24 out of 36 cases (66%), the comparison is won by the *AutoSoft* classifier.

Table 3. Comparison between *AutoSoft* and classifiers from the literature in terms of *F*1-*score*, WINS, LOSSES and DRAWS.

Type of Features	Number of Authors	Classifiers				
		AutoSoft	SVC	RF	GNB	kNN
unigrams	5	0.986	**0.993**	0.981	0.963	0.975
	12	0.975	**0.993**	0.965	0.946	0.958
	87	0.902	**0.953**	0.735	0.841	0.854
trigrams	5	0.98	**0.994**	0.98	0.972	0.96
	12	0.981	**0.992**	0.976	0.947	0.936
	87	0.909	**0.953**	0.734	0.855	0.817
5-grams	5	0.985	**0.998**	0.982	0.971	0.99
	12	0.979	**0.99**	0.976	0.935	0.983
	87	0.913	**0.95**	0.785	0.835	0.909
	AutoSoft WINS				**24**	
	AutoSoft LOSSES				11	
	AutoSoft DRAWS				1	

The results from Table 3 reveal a surprisingly good performance of the kNN classifier. We note that, when using 5-grams and 5 or 12 authors, kNN slightly outperforms *AutoSoft*. However, kNN is outperformed by *AutoSoft* for 87 developers. A possible explanation is that kNN is a lazy learner, able to find local approximations of the target function to be learned. On the other hand, *AutoSoft* is an eager learner that finds, from the entire training data set, a global approximation of the target function. Thus, it is likely that, for specific instances, the lazy learners (kNN, in our case) would find better (local) approximations of the target function than the (global) ones provided by an eager learner (*AutoSoft*, in our case). One also observes that *AutoSoft* compares favorably to other classifiers from the literature. In 7 out of 9 performed experiments (combination of *N*-gram size and number of authors), *AutoSoft* performs as well or better than 75% of the other classifiers (3 out of 4), being outperformed only by SVC, and by a small margin in [0.007–0.051]. In the remaining experiments, *AutoSoft* has a performance higher than or equal to that of 50% of the other classifiers, with KNN performing surprisingly well for 5-gram features. We

also note that, for all nine experiments depicted in Table 3, the $F1$ values provided by *AutoSoft* exceed the average $F1$ scores obtained for the other classifiers. Figure 6 illustrates this comparison, where the OX axis denotes the performed experiments and the OY axis denotes the $F1$ value.

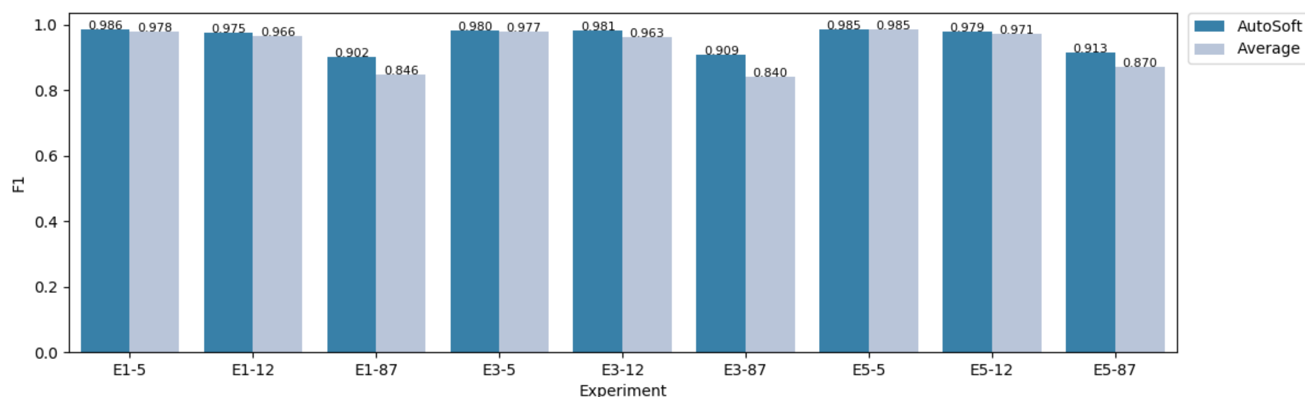

Figure 6. $F1$-*scores* obtained for *AutoSoft* and the average $F1$-*scores* obtained for the other classifiers in experiments EX-Y shown in Table 3, where X = N-gram size and Y = number of developers.

To strengthen the previous analysis and for verifying the statistical significance of the differences observed between the performance of *AutoSoft* and the performance of SVC, RF, GNB and kNN classifiers, a two-tailed paired Wilcoxon signed-rank test [54,55] has been used. The sample of $F1$ values obtained after evaluating *AutoSoft* on all experiments depicted in Table 3 was tested against the sample of $F1$ values obtained for the other classifiers (SVC, RF, GNB and kNN). A p-value of 0.01369 was obtained, confirming that the improvement achieved by *AutoSoft* is statistically significant, at a significance level of $alpha = 0.05$.

Table 3 also shows that *AutoSoft* scales well as the number of authors increases, as opposed to some of the other classifiers. As for the N-gram size, in general, better performances are obtained with N values of 3 and 5 as opposed to using unigrams, both for *AutoSoft* and for most of the other classifiers. This is especially evident in the classification task which considers 87 developers.

In conclusion, the research question RQ2 can now be answered. The performance on the multi-class software authorship attribution task is equaled or improved by using the proposed classifier, *AutoSoft*, in a majority of the cases defined for evaluation. Moreover, the definition of *AutoSoft* allows the computation of the probability that a test instance (software program) was written by a given developer, thus providing additional information that might prove useful in real-world scenarios.

6. Extension of the *AutoSoft* Classifier

One of the major advantages of our *AutoSoft* proposal is that the classification model can be easily extended to recognize not only the set of original authors (developers) on which it was trained, but an "unknown" class as well. More specifically, if the testing instance (source code) does not resemble the programming style of either of the developers, then it will be assigned to an additional class (the "unknown" class). From a practical perspective, such an extension would help a project manager to detect a source code that was not developed by any of his/her team members.

From a theoretical perspective, our aim is to extend a multi-class classifier that works in a closed-set configuration to a classifier which classifies instances from an open testing space. In an open-set configuration, the classifier solves the multi-class classification task and the novelty detection (represented by an "unknown" class) at the same time.

Let us consider the theoretical model introduced in Section 4. The classical *AutoSoft* model was designed to classify any source code (received during testing) as written by

one of the software developers $Dev_1, Dev_2, \ldots, Dev_n$ on which it was trained. This happens even if the testing instance (source code) was not developed by any of the authors considered in training.

Thus, we aim to extend the decision-making process during the classification stage (Section 4.3.1) to include the probability of classifying a testing instance (software program) sp as belonging to an "unknown" class (i.e., other than the given $Dev_1, Dev_2, \ldots, Dev_n$ classes). Introducing the extended classifier, $AutoSoft^{ext}$, we try to answer the research question RQ3.

The training step of $AutoSoft^{ext}$ is the same as the training of the original *AutoSoft* classifier, described in Section 4.2.

The classification stage described in Section 4.3.1 will be extended, by including a prior step for deciding the likelihood (denoted by $p_{unknown}(sp)$) that the software program sp belongs to the "unknown" class. The intuition behind the decision of classifying the author of sp as an "unknown" one is the following: if the program embedding $pe = embed(sp)$ is "distant" enough from each of the autoencoders $A_1, A_2, \ldots, A_n$ (corresponding to the developers $Dev_1, Dev_2, \ldots, Dev_n$), then it is likely to have an unknown author (that was not encountered during the training). The embedding pe of a certain software program is considered to be "distant enough" from an autoencoder A if the loss of A for the input vector pe is greater than a certain threshold τ.

6.1. Classification Stage for the $AutoSoft^{ext}$ Classifier

Let us denote by $l_i(sp)$ the loss of the autoencoder A_i assigned to the i-th software developer for the embedding $pe = embed(sp)$ of the testing instance sp. By τ_i, we denote the threshold used for deciding if the structure of pe does not resemble that of the instances of class Dev_i encoded by A_i and by $dist_i(sp) = l_i(sp) - \tau_i$ the "distance" between sp and A_i.

The probability $p_{unknown}(sp)$ is defined as shown in Formula (6):

$$p_{unknown}(sp) = 0.5 + \frac{\prod_{i=1}^{j} dist_i(sp)}{2 \cdot \prod_{i=1}^{j} (l_i(sp) + \tau_i)}, \tag{6}$$

where

$$j = \begin{cases} n & \text{if } dist_i(sp) \geq 0 \ \forall i \in [1, n]; \\ \min\limits_{i=1,n} \{i | dist_i(sp) < 0\} & \text{otherwise.} \end{cases}$$

It can be easily proved that $0 \leq p_{unknown}(sp) \leq 1$ for each testing instance sp. Moreover, $p_{unknown}(sp)$ is greater than 0.5 if and only if $dist_i(sp) \geq 0$,$\forall$i, $1 \leq i \leq n$ (or, equivalently, $l_i(sp) \geq \tau_i$, $\forall$i, $1 \leq i \leq n$), i.e., sp belongs to the "unknown" class. Otherwise, if $p_{unknown}(sp) < 0.5$, it follows that sp belongs to one of the classes $Dev_1, Dev_2, \ldots, Dev_n$. In this case, the decision-making process described in Section 4.3.1 will be applied for deciding the output class.

Thus, the classification for the testing instance sp is decided as shown in Algorithm 1.

6.2. Evaluation of $AutoSoft^{ext}$

In the following, we evaluate the performance of $AutoSoft^{ext}$ on predicting the "unknown" class. For this purpose, the problem is modelled as a binary classification task, in which the goal is to decide if the author of a certain software program sp belongs to the original set of authors ($\mathcal{O}riginal = \{Dev_1, Dev_2, \ldots, Dev_n\}$) on which $AutoSoft^{ext}$ has been trained, or it belongs to an "unknown" class.

We decided to evaluate the binary classification task (classification in two classes: "orginal" vs. "unknown"), since our aim is to highlight the ability of our $AutoSoft^{ext}$ model to detect the "unknown" class. Subsequently, after it has been decided that the testing instance does not belong to the "unknown" class, the author from the original set will be

identified as for the *AutoSoft* classifier (see Section 4.3.1), and this performance has been already evaluated in Section 5.

Algorithm 1 Classification for the testing instance sp, considering an additional "unknown" class.

```
  function CLASSIFY(n, A, sp)
Require:
  n —the number of original software developers
  A —the set of n trained autoencoders
  sp —the testing instance (software program) to be classified
Ensure:
  return the predicted class c ∈ {1, ..., n} ∪ {"unknown"}

    if p_unknown(sp) ≥ 0.5 then
        c ← "unknown"
    else
        // The output class c belongs to {1, 2, ..., n}
        // The decision-making process from Section 4.3.1 is applied for deciding
        // the output class c ∈ {1, 2, ..., n}
    end if
    return c
  end function
```

6.2.1. Testing

In our binary classification task, the *positive* class is represented by the set of original authors, i.e., $\mathcal{O}riginal$, while the *negative* class is represented by the "unknown" class.

A testing set consists of test instances from the $\mathcal{O}riginal$ set, specifically the 10% testing set used in the multi-class classification experiment. Additionally, a number of instances from the "unknown" class are included. They are represented by codes of authors whose instances were not included in the training of the `doc2vec` model and the training of the ensemble of autoencoders. Instead, the program embeddings of these codes are inferred from the existent `doc2vec` model trained on the $\mathcal{O}riginal$ authors' instances. In addition, 10% of the total instances of an "unknown" developer are randomly selected.

A confusion matrix is computed for each testing set, with the following values: *TP* (*true positives*—the codes developed by "original" authors and predicted correctly), *FP* (*false positives*—the codes classified incorrectly as written by "original" developers), *TN* (*true negatives*—the codes written by "unknown" authors and recognized as such) and *FN* (*false negatives*—the codes developed by "original" authors but classified as written by "unknown" developers). The performance of the $AutoSoft^{ext}$ model on a certain testing data set is then evaluated with respect to a series of measures based on these confusion matrix values (*TP*, *TN*, *FP* and *FN*) used for binary classification performance evaluation: accuracy, precision, recall, F-score (F1) and specificity.

To account for the randomness involved in the selection of the testing data sets, the testing is repeated 10 times. The obtained values for the performance measures are then averaged over the 10 runs, and a 95% CI of the average value is computed. Experimentally, we found that the best threshold τ_i for an autoencoder A_i is calculated using Formula (7):

$$\tau_i = \mu_i + 2 \cdot \sigma_i, \tag{7}$$

where μ_i is the mean, and σ_i represents the standard deviation of the losses for the training instances written by the developer Dev_i.

Table 4 presents the results of the $AutoSoft^{ext}$ model with respect to two N-gram sizes on seven different tasks involving the set of $\mathcal{O}riginal$ developers ($n = 5$) and seven different "unknown" developers. The five developers included in the $\mathcal{O}riginal$ set are the developers constituting the five developer subsets presented in Section 5.1.

Table 4. *AutoSoft*^{ext} results with respect to N-gram size for seven "unknown" authors and an $\mathcal{O}riginal$ set with $n = 5$. In addition, 95% confidence intervals are used for the results.

Developer	N-Gram Type	Performance Measures				
		Accuracy	Precision	Recall	$F1$	Specificity
Dev_{u1}	unigrams	0.904 ± 0.015	0.926 ± 0.012	0.965 ± 0.013	0.945 ± 0.009	0.537 ± 0.082
	5-grams	**0.974 ± 0.006**	**0.977 ± 0.007**	**0.994 ± 0.005**	**0.993 ± 0.003**	**0.858 ± 0.046**
Dev_{u2}	unigrams	0.921 ± 0.014	0.947 ± 0.01	0.965 ± 0.013	0.956 ± 0.008	0.587 ± 0.087
	5-grams	**0.988 ± 0.006**	**0.993 ± 0.004**	**0.994 ± 0.005**	**0.993 ± 0.003**	**0.947 ± 0.033**
Dev_{u3}	unigrams	0.891 ± 0.011	0.917 ± 0.006	0.965 ± 0.013	0.94 ± 0.007	0.333 ± 0.055
	5-grams	**0.979 ± 0.009**	**0.983 ± 0.01**	**0.994 ± 0.005**	**0.988 ± 0.005**	**0.867 ± 0.083**
Dev_{u4}	unigrams	0.89 ± 0.016	0.911 ± 0.01	0.965 ± 0.013	0.937 ± 0.009	0.4 ± 0.074
	5-grams	**0.977 ± 0.008**	**0.98 ± 0.007**	**0.994 ± 0.005**	**0.987 ± 0.005**	**0.872 ± 0.046**
Dev_{u5}	unigrams	0.871 ± 0.011	0.896 ± 0.007	0.965 ± 0.013	0.929 ± 0.007	0.2 ± 0.063
	5-grams	**0.965 ± 0.009**	**0.968 ± 0.01**	**0.994 ± 0.005**	**0.981 ± 0.005**	**0.762 ± 0.079**
Dev_{u6}	unigrams	0.924 ± 0.017	0.948 ± 0.009	0.965 ± 0.013	0.956 ± 0.01	0.679 ± 0.054
	5-grams	**0.992 ± 0.005**	**0.997 ± 0.003**	**0.994 ± 0.005**	**0.996 ± 0.003**	**0.984 ± 0.016**
Dev_{u7}	unigrams	0.899 ± 0.014	0.922 ± 0.011	0.965 ± 0.013	0.943 ± 0.008	0.483 ± 0.078
	5-grams	**0.97 ± 0.009**	**0.972 ± 0.007**	**0.994 ± 0.005**	**0.983 ± 0.003**	**0.817 ± 0.046**

As it can be seen, the *AutoSoft*^{ext} is able to recognize the positive class (the instances belonging to the $\mathcal{O}riginal$ set) very well, for both unigram and 5-gram representations. However, there is a significant difference in the ability of the *AutoSoft*^{ext} classifier to recognize the negative class in the two test settings. It is evident that, when using a 5-grams `doc2vec` model trained on the $\mathcal{O}riginal$ set instances, the representation is much more specific to the coding style of this developer group. In contrast, using a unigram representation, inferred "unknown" class `doc2vec` vectors are very similar to $\mathcal{O}riginal$ set ones.

The performance of $AutoSoft^{ext}$ in the unigram setting is hindered especially by the high number of *false positives*—the instances written by "unknown" authors but classified as written by "original" developers. One explanation may be that it is more difficult for the developer style to be captured in terms of unigrams. In this case, the part of the `doc2vec` vocabulary common to all developers would be much larger, therefore overshadowing the contribution of the fewer distinguishing features in building the document representations. While distinguishing unigrams are mostly limited to variable and constant names, and, to a lesser extent, libraries used, N-grams may be able to capture particularities in the expression of logic constructs or the definition of data structures and their subsequent use (see Section 4.1 for examples of N-grams given a fragment of code). Consequently, the autoencoders manage to better encode each developer's style given representations based on sequences of tokens rather than ones based on single tokens. This conclusion is also supported by the results obtained in the multi-class setting, where any $N > 1$ leads to an increase in performance with respect to the results obtained using unigram-based representations. With a better delimited style for the "original" developers that follows the use of a representation based on 5-grams, the number of *false positives* thus decreases.

This can also be seen in Figure 7. A side-by-side visualization of all $\mathcal{O}riginal$ and "unknown" instances for a given run for unigrams and 5-grams representation shows the efficiency of the latter one in distinguishing between the two classes.

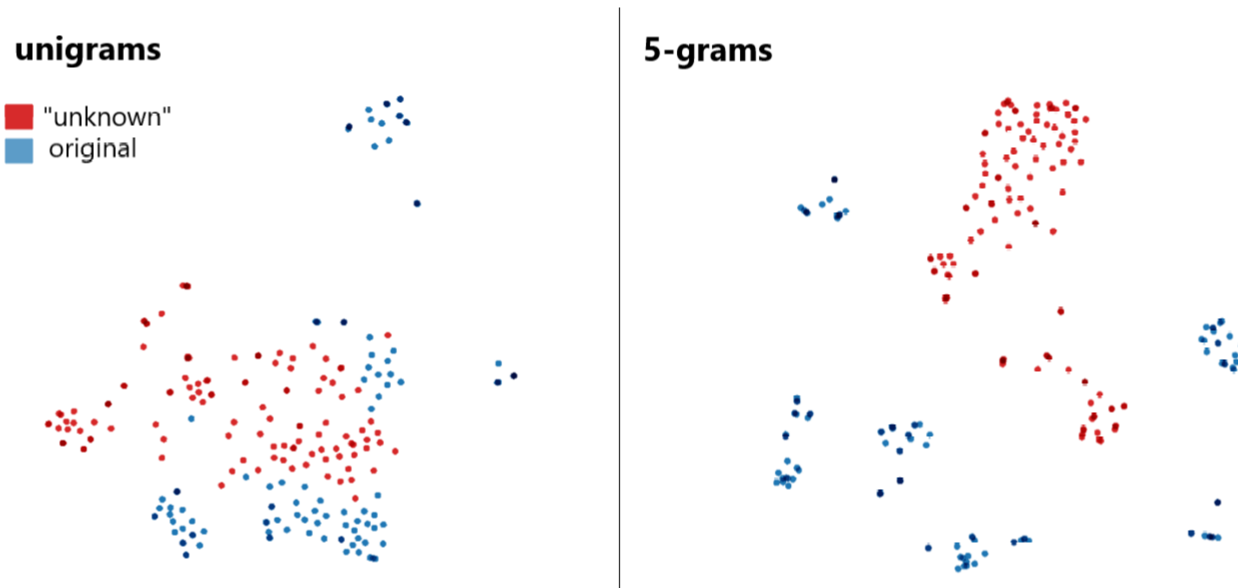

Figure 7. T-SNE visualization for all *Original* ($n = 5$) and "unknown" instances in a random state with respect to N-gram size.

6.2.2. Comparison to $OneClassSVM$

We note that the classical models (SVC, RF, GNB, kNN) used in Section 5.2.2 are not able to distinguish an "unknown" class. Instead, to evaluate the performance of $AutoSoft^{ext}$, we use One-Class SVM classifiers [56,57], specifically the OneClassSVM implementation from the *scikit-learn* library [53], which is the only model we have found that tackles a similar problem to ours. One-Class Support Vector Machines ($OSVM$)-based algorithms are broadly used for one-class classification [58].

The main idea of $OSVM$s is to construct a hyper-plane around the data (in a feature space) in such a way that the distance from the hyper-plane to the origin is maximal and regions that contain no data are separated. In other words, $OSVM$'s employ a strategy of mapping the given data into a feature space corresponding to the chosen kernel, with a goal of separating this data from the origin with maximum margin. Thus, a binary function that takes the value "+1" in a "small" region that captures most of the data points, and "−1" elsewhere is learned [56].

We adapt the $OSVM$ methodology to mirror that of $AutoSoft^{ext}$. The motivation for this adaptation is twofold. First, as the default $OSVM$ implementation is designed for use in binary classification contexts, an adaptation was needed to fit our multi-class setting. This was achieved by training an $OSVM$ model for each class/developer, and obtaining a prediction from each of these trained models for a given test instance. Secondly, to obtain a valid comparison between $AutoSoft^{ext}$ and the $OSVM$ classifier, the methodology was modified in such a way that it followed the steps of the $AutoSoft^{ext}$ model. Thus, we proposed an ensemble of $OSVM$ classifiers to mirror the ensemble of autoencoders. Then, much like the program embedding, *pe* was considered to belong to the "unknown" class if it was "distant" enough from each of the autoencoders in the ensemble, it is considered to be authored by an unknown author by the $OSVM$-based ensemble of classifiers if it lies outside the class boundaries for each of the trained $OSVM$ models.

Similar methodologies were employed to detect and classify scars, marks and tattoos found in forensic images [59] and for text genre identification in web pages [60]. Differences between these approaches and ours reside in the fact that, in [59], the test instance can belong to multiple classes, while, in [60], the instance is labeled according to the most confident classifier that identifies the instance as belonging to the genre on which it was trained.

Specifically, we train five $OSVM$ models, one for each developer from the $\mathcal{O}riginal$ set ($n = 5$). The decision for each instance is then taken as follows: if any of the $OSVM$ models return a "+1" label (that signifies membership to the positive class), then the instance is considered to be written by one of the developers in the $\mathcal{O}riginal$ set; else, if all $OSVM$ models return a "-1" label, the instance is considered to belong to the "unknown" class.

Table 5 shows that the $AutoSoft^{ext}$ model compares favorably to the $OSVM$ classifier in most cases. $AutoSoft^{ext}$ always obtains a higher $Recall$ than the $OSVM$ classifier, which surpasses our model in terms of $Precision$ and $Specificity$ in 6 out of 7 cases. That is, the $OSVM$ classifier manages to better recognize the "unknown" instances, an improvement that comes at the expense of misclassifying positive class instances. Moreover, for 5-grams features, the difference in the performance of $AutoSoft^{ext}$ and $OSVM$ with respect to classification of "unknown" instances is relatively similar, with good performance obtained for the $AutoSoft^{ext}$ in addition to almost perfect $Recall$.

In conclusion, RQ3 can now be answered. The proposed extension to the $AutoSoft$ model, $AutoSoft^{ext}$, can be used to recognize whether a given test instance was written by an original developer (from a given set), or by some "unknown" developer. $AutoSoft^{ext}$ obtains good performance in the binary classification task, and compares favorably with existing one-class classification models such as One-Class Support Vector Machines.

The $AutoSoft^{ext}$ model is proposed as proof of concept supporting the hypothesis that the $AutoSoft$ model introduced in Section 4 for solving the software authorship attribution task in a closed-set configuration can easily be transformed to address the SAA problem in an open-set configuration as well. More experiments to validate this hypothesis are beyond the scope of this paper and are marked as future work, including an analysis of the performance of $AutoSoft^{ext}$ when the number of developers in the $\mathcal{O}riginal$ set is increased, and the subsequent comparison with the $OSVM$ classifier. Such a change in the experimental setting would be best served by an extensive discussion with regard to aspects such as: choice of train and testing instances from the Google Code Jam dataset, and, from a broader perspective, the coding context and the subsequent programming style variability.

The first of these aspects refers primarily to an investigation into the number of instances needed to efficiently learn a developer's programming style. However, this may also imply a discussion on programmer proficiency, experience, and skill, as in the Google Code Jam data set, developers that have more programs included either participated in more rounds, providing high-quality solutions to advance, or entered the contest in multiple years. The coding context and programming style variability are connected aspects, as in the current formulation, for a testing instance to be identified as belonging to an "unknown" developer, the programming style found within must be significantly different from that of a given number of developers (those in the $\mathcal{O}riginal$ set). This raises the question of how much variability can be found in programming style in particular coding contexts—a contest like Google Code Jam, student assignments for computer science classes or real world teams working on industry projects. For instance, programming style variability in solutions to contest-like coding problems is different from that in student assignments (which may also be restricted to having the same structure, particular classes, etc.) and, perhaps even more obviously, from code produced in corporate contexts. The number of authors from the $\mathcal{O}riginal$ set, then, should be chosen in concordance with this expected variability, perhaps through experimental determination. Such experiments and discussion, while extremely interesting, are outside the scope of this paper.

Table 5. Comparison between $AutoSoft^{ext}$ and $OSVM$ with respect to N-gram size for seven "unknown" authors and an $\mathcal{O}riginal$ set with $n = 5$.

Developer	Performance Measure	Unigrams		5-Grams	
		$AutoSoft^{ext}$	$OSVM$	$AutoSoft^{ext}$	$OSVM$
	$Accuracy$	**0.904**	0.783	**0.974**	0.763
	$Precision$	**0.926**	0.91	0.977	**1**
Dev_{u1}	$Recall$	**0.965**	0.829	**0.994**	0.724
	$F1$	**0.945**	0.842	**0.985**	0.839
	$Specificity$	**0.537**	0.505	0.858	**1**
	$Accuracy$	**0.921**	0.815	**0.988**	0.756
	$Precision$	0.947	**0.956**	0.993	**1**
Dev_{u2}	$Recall$	**0.965**	0.829	**0.994**	0.724
	$F1$	**0.956**	0.887	**0.993**	0.839
	$Specificity$	0.587	**0.707**	0.947	**1**
	$Accuracy$	**0.891**	0.808	**0.979**	0.756
	$Precision$	0.917	**0.948**	0.983	**1**
Dev_{u3}	$Recall$	**0.965**	0.829	**0.994**	0.724
	$F1$	**0.94**	0.884	**0.988**	0.839
	$Specificity$	0.333	**0.647**	0.867	**1**
	$Accuracy$	**0.89**	0.786	**0.977**	0.761
	$Precision$	0.911	**0.916**	0.98	**1**
Dev_{u4}	$Recall$	**0.965**	0.829	0.994	0.724
	$F1$	**0.937**	0.87	**0.987**	0.839
	$Specificity$	0.4	**0.517**	0.872	**1**
	$Accuracy$	**0.871**	0.797	**0.965**	0.758
	$Precision$	0.896	**0.933**	0.968	**1**
Dev_{u5}	$Recall$	**0.965**	0.829	**0.994**	0.724
	$F1$	**0.929**	0.877	**0.981**	0.839
	$Specificity$	0.2	**0.569**	0.762	**1**
	$Accuracy$	**0.924**	0.824	**0.992**	0.763
	$Precision$	0.948	**0.961**	**0.997**	1
Dev_{u6}	$Recall$	**0.965**	0.829	**0.994**	0.724
	$F1$	**0.956**	0.889	**0.996**	0.839
	$Specificity$	0.679	**0.795**	0.984	**1**
	$Accuracy$	**0.899**	0.842	**0.97**	0.761
	$Precision$	0.922	**0.986**	0.972	**1**
Dev_{u7}	$Recall$	**0.965**	0.829	**0.994**	0.724
	$F1$	**0.943**	0.9	**0.983**	0.839
	$Specificity$	0.483	**0.922**	0.817	**1**

7. Threats to Validity

The experimental evaluation and analysis of our *AutoSoft* classifier may be influenced by some threats to validity and biases that may affect the obtained results. The issues which may have affected the experimental results and their analysis are further discussed.

Regarding *construct validity* [61], the performance of our *AutoSoft* model for authorship attribution has been extensively assessed using evaluation metrics used in the literature for imbalanced multi-class classification. For reducing the threats to construct validity, throughout our experiments, we followed best practices in building and evaluating ML models such as: model validation has been used in the training, cross-validation and statistical analysis were employed for a more precise evaluation of *AutoSoft*.

In what concerns *internal validity*, it is about internal parameters and experimental settings which could influence the experimental results. As any eager supervised classifier that is built during training, the *AutoSoft* model strongly depends on internal hyperparameters such as: the architecture used for the autoencoders (layers, neurons on each layer, latent space dimensionality, activation functions, etc.), the optimization algorithm used during the training stage, etc. For minimizing threats to internal validity, various autoencoder architectures and hyperparameters settings were examined in the experimental step for hyperparameters tuning, and the resulting model has been cross-validated.

Regarding the threats to *external validity*, namely the extent to which the results obtained by *AutoSoft* may be generalized, we have chosen a public data set that has been previously employed in the literature for software authorship attribution. Still, the experimental evaluation has to be further extended to other data sets, open source projects written in different programming languages (Java, C++, Python), to achieve better generalization. Closed-source projects should also be considered for evaluating *AutoSoft*'s performance, to test if the findings of the current study are still valid.

In what concerns *reliability*, the methodology used for data collection, the AEs architectures used for training the *AutoSoft* model as well as the testing methodology have been detailed in Section 4 in order to allow the reproducibility of the results. The data used in the experiments is a subset of data found at [35], publicly available at [62]. As far as the experimental methodology is concerned, we applied cross-validation by repeating the same experiment and provided a statistical analysis by computing confidence intervals for the obtained performance measures, in order to increase the precision of the results. For ensuring a better interpretation of the results and the validity of the conclusions, *AutoSoft* has been extensively evaluated by considering various numbers of authors and type of features. In addition, the statistical significance of the improvement achieved by *AutoSoft* has been confirmed through a statistical test.

8. Conclusions and Future Work

To solve the problem of software authorship attribution, the *AutoSoft* classification model, based on an ensemble of deep AEs (one AE for each author/developer), was introduced. The representation of the software programs was inspired from the NLP domain. A lexical analysis followed by the program embedding step (`doc2vec` model) provides a distributed representation of a software program, representation that captures contextual, syntactic and semantic aspects of the code, as a text. The deep autoencoders applied to program embeddings proved to uncover relevant hidden features of the software programs that successfully distinguished the authors/developers. The decision in classification (the predicted developer of a testing software program) is based on the probabilities calculated using the cosine similarities between the program embedding and the reconstructions provided by the autoencoders. *AutoSoft* is the answer to RQ1 proposed as the first step of research in the paper.

From a theoretical perspective, the advantages of *AutoSoft* model can be summarized as follows:

(1) Its generality is based on the fact that it contains only one language-dependent component: a lexical analyzer. Therefore, this classification model can be used for author-

ship attribution of any texts: software programs in programming languages or documents in natural languages.

(2) The calculated probabilities can be applied not only to predict the most likely author of a software program, but also to find the most probable and close 2–3 authors, information used further to compare the work of different authors.

(3) *AutoSoft* was easily extended to $AutoSoft^{ext}$ classifier with the aim of identifying also "unknown" instances, i.e., software programs that are not authored by the given (original) developers. Thus, the research question RQ3 was answered.

(4) *AutoSoft* classifier can be adapted to solve the co-authorship attribution problem. Given an unseen program, significantly higher probabilities for a subset of developers compared with the others will predict a group of co-authors.

In contrast, the limitations of the *AutoSoft* model can be identified as the following: (1) the dependence on the quality of the trained `doc2vec` model, with all that it entails: the number of instances needed in the training corpus, as well as the search for the optimal model parameter values; (2) the requirement for each considered developer in the training set to have enough training instances such that the autoencoder manages to learn the developer's programming style; and (3) the *AutoSoft* training and classification stages have a duration directly proportional to the number of initial authors considered, with a higher ratio than other models. However, each of these limitations can be addressed. First, while using a model like `doc2vec` to learn document representations incurs an extra step, the benefits of capturing high level semantic and contextual information that eludes simpler Bag-of-Words models are evident, especially when a higher number of authors is considered. Additionally, by using a metric such as the difficulty measure, described in Section 5.1, the quality of the resulting `doc2vec` representations can be easily evaluated. Secondly, the disadvantage of the larger number of training instances needed by the autoencoder can be compensated by the fact that, as opposed to the classic multi-class models considered, a focus on individual authors can provide important insights into their programming style, both in a focused context, of assessing particularities of one developer, and in a larger one, with the possibility of finer-grained distinctions in developer groups. Finally, we address the third point by marking performance as a more important aspect than training time. In the proposed context, we value the richness of information provided by the *AutoSoft* model in comparison to other models from the literature, such as class membership probabilities, and the flexibility in adapting *AutoSoft* to a series of tasks, some of which were described in Section 6 and some proposed as future research directions in the next paragraphs. Nonetheless, the inclusion of precursory steps to the multi-class classification, such as the proposed extension that differentiates between *original* and *unknown* instances, may reduce the time needed for the multi-class classification step, as only *original* instances should be further fed to the *AutoSoft* model.

With the goal of answering the research question RQ2, the performance of *AutoSoft* was evaluated and compared with other supervised classifiers in the experiments conducted on a subset of Python programs from the 2008–2020 Google Code Jam data set. The comparative results illustrated in Table 3 showed that *AutoSoft* (with an $F1$-*score* ranging from 0.902 to 0.986) performs similarly or better than most of the classifiers (Random Forests, Gaussian Naive Bayes and k-Nearest Neighbors classifiers), being outperformed only by a Support Vector Classifier.

In conclusion, all the research questions stated at the beginning of the paper were answered.

As further work, we aim at solving the co-authorship problem by adapting the classification decision of *AutoSoft* model. Another future direction is to use in *AutoSoft* the representation of software programs based on another NLP technique, Latent Semantic Indexing (LSI) [63], which is language-independent. LSI vector space is a latent, low-dimensional space that captures the semantic and conceptual content of texts.

The evaluation of both classification models on real data, open source projects written in different programming languages—Java, C++ and Python—is also a future purpose.

Author Contributions: Conceptualization, G.C., M.L. and A.B.; methodology, G.C., M.L. and A.B.; software, A.B.; validation, G.C., M.L. and A.B.; formal analysis, G.C., M.L. and A.B.; investigation, G.C., M.L. and A.B.; resources, G.C., M.L. and A.B.; data curation, A.B.; writing—original draft preparation, G.C.; writing—review and editing, G.C., M.L. and A.B.; visualization, G.C., M.L. and A.B.; funding acquisition, G.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by a grant of the Ministry of Research, Innovation and Digitization, CNCS/CCCDI—UEFISCDI, project number PN-III-P4-ID-PCE-2020-0800, within PNCDI III.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The authors would like to thank the editor and the anonymous reviewers for their useful suggestions and comments that helped to improve the paper and the presentation.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Abuhamad, M.; Rhim, J.S.; AbuHmed, T.; Ullah, S.; Kang, S.; Nyang, D. Code authorship identification using convolutional neural networks. *Future Gener. Comput. Syst.* **2019**, *95*, 104–115. [CrossRef]
2. Sallis, P.; Aakjaer, A.; MacDonell, S. Software forensics: Old methods for a new science. In Proceedings of the 1996 International Conference Software Engineering: Education and Practice, Dunedin, New Zealand, 24–27 January 1996; pp. 481–485.
3. Tian, Q.; Fang, C.C.; Yeh, C.W. Software Release Assessment under Multiple Alternatives with Consideration of Debuggers; Learning Rate and Imperfect Debugging Environment. *Mathematics* **2022**, *10*, 1744. [CrossRef]
4. Bogomolov, E.; Kovalenko, V.; Rebryk, Y.; Bacchelli, A.; Bryksin, T. Authorship attribution of source code: A language-agnostic approach and applicability in software engineering. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021; pp. 932–944.
5. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016.
6. Le, Q. Building high-level features using large scale unsupervised learning. In Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, Vancouver, BC, Canada, 26–31 May 2013; pp. 8595–8598.
7. Teletin, M.; Czibula, G.; Codre, C. AutoSimP: An Approach for Predicting Proteins' Structural Similarities Using an Ensemble of Deep Autoencoders. In *Knowledge Science, Engineering and Management*; Douligeris, C., Karagiannis, D., Apostolou, D., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 49–54.
8. Czibula, G.; Albu, A.I.; Bocicor, M.I.; Chira, C. AutoPPI: An Ensemble of Deep Autoencoders for Protein–Protein Interaction Prediction. *Entropy* **2021**, *23*, 643. [CrossRef]
9. Deng, J.; Zhang, Z.; Marchi, E.; Schuller, B. Sparse autoencoder-based feature transfer learning for speech emotion recognition. In Proceedings of the 2013 Humaine Association Conference on Affective Computing and Intelligent Interaction, Geneva, Switzerland, 2–5 September 2013; pp. 511–516.
10. Tatar, D.; Czibula, G.S.; Mihis, A.D.; Mihalcea, R. Textual Entailment as a Directional Relation. *J. Res. Pract. Inf. Technol.* **2009**, *41*, 53–64.
11. Le, Q.; Mikolov, T. Distributed Representations of Sentences and Documents. In Proceedings of the 31st International Conference on Machine Learning, Beijing, China, 21–26 June 2014; pp. 1188–1196.
12. Chuanxing, G.; Sheng-Jun, H.; Songcan, C. Recent Advances in Open Set Recognition: A Survey. *IEEE Trans. Pattern Anal. Mach. Intell.* **2021**, *43*, 3614–3631.
13. Anvik, J.; Hiew, L.; Murphy, G.C. Who should fix this bug? In Proceedings of the 28th International Conference on Software Engineering, Shanghai, China, 20–28 May 2006; pp. 361–370.
14. Fritz, T.; Ou, J.; Murphy, G.C.; Murphy-Hill, E. A degree-of-knowledge model to capture source code familiarity. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Cape Town, South Africa, 2–8 May 2010; Volume 1, pp. 385–394.
15. Girba, T.; Kuhn, A.; Seeberger, M.; Ducasse, S. How developers drive software evolution. In Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE'05), Lisbon, Portugal, 5–6 September 2005; pp. 113–122.
16. Bird, C.; Nagappan, N.; Murphy, B.; Gall, H.; Devanbu, P. Don't touch my code! Examining the effects of ownership on software quality. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Szeged, Hungary, 5–9 September 2011; pp. 4–14.

17. Thongtanunam, P.; McIntosh, S.; Hassan, A.E.; Iida, H. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 1039–1050.
18. Rahman, F.; Devanbu, P. Ownership, experience and defects: A fine-grained study of authorship. In Proceedings of the 33rd International Conference on Software Engineering, Honolulu, HI, USA, 21–28 May 2011; pp. 491–500.
19. Krsul, I.; Spafford, E.H. Authorship analysis: Identifying the author of a program. *Comput. Secur.* **1997**, *16*, 233–257. [CrossRef]
20. Oman, P.W.; Cook, C.R. Programming style authorship analysis. In Proceedings of the 17th Conference on ACM Annual Computer Science Conference, Kentucky, Louisville, 21–23 February 1989; pp. 320–326.
21. Spafford, E.H.; Weeber, S.A. Software forensics: Can we track code to its authors? *Comput. Secur.* **1993**, *12*, 585–595. [CrossRef]
22. Rosenblum, N.; Zhu, X.; Miller, B.P. Who wrote this code? Identifying the authors of program binaries. In *European Symposium on Research in Computer Security*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 172–189.
23. Burrows, S.; Tahaghoghi, S.M. Source code authorship attribution using n-grams. In Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, 10 December 2007; pp. 32–39.
24. Frantzeskou, G.; Stamatatos, E.; Gritzalis, S.; Katsikas, S. Source code author identification based on n-gram author profiles. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*; Springer: Boston, MA, USA, 2006; pp. 508–515.
25. Tennyson, M.F. A Replicated Comparative Study of Source Code Authorship Attribution. In Proceedings of the 2013 3rd International Workshop on Replication in Empirical Software Engineering Research, Baltimore, MD, USA, 9 October 2013; pp. 76–83. [CrossRef]
26. Frantzeskou, G.; Stamatatos, E.; Gritzalis, S.; Chaski, C.E.; Howald, B.S. Identifying authorship by byte-level n-grams: The source code author profile (SCAP) method. *Int. J. Digit. Evid.* **2007**, *6*, 1–18.
27. Ullah, F.; Jabbar, S.; AlTurjman, F. Programmers' de-anonymization using a hybrid approach of abstract syntax tree and deep learning. *Technol. Forecast. Soc. Chang.* **2020**, *159*, 120186. [CrossRef]
28. Alsulami, B.; Dauber, E.; Harang, R.; Mancoridis, S.; Greenstadt, R. Source code authorship attribution using long short-term memory based networks. In *European Symposium on Research in Computer Security*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 65–82.
29. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. Code2vec: Learning Distributed Representations of Code. CoRR. 2018. Available online: http://xxx.lanl.gov/abs/1803.09473 (accessed on 15 March 2021).
30. Ullah, F.; Naeem, M.R.; Naeem, H.; Cheng, X.; Alazab, M. CroLSSim: Cross-language software similarity detector using hybrid approach of LSA-based AST-MDrep features and CNN-LSTM model. *Int. J. Intell. Syst.* **2022**, *2022*, 1–28. [CrossRef]
31. Mateless, R.; Tsur, O.; Moskovitch, R. Pkg2Vec: Hierarchical package embedding for code authorship attribution. *Future Gener. Comput. Syst.* **2021**, *116*, 49–60. [CrossRef]
32. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional neural networks over tree structures for programming language processing. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016; pp. 1287–1293.
33. Piech, C.; Huang, J.; Nguyen, A.; Phulsuksombati, M.; Sahami, M.; Guibas, L. Learning program embeddings to propagate feedback on student code. In Proceedings of the International Conference on Machine Learning, Lille, France, 6–11 July 2015; pp. 1093–1102.
34. Google. Google Code Jam Competition. Available online: https://codingcompetitions.withgoogle.com/codejam (accessed on 15 September 2021).
35. Petrik, J. GCJ Data Set. Available online: https://github.com/Jur1cek/gcj-dataset (accessed on 15 September 2021).
36. Simko, L.; Zettlemoyer, L.; Kohno, T. Recognizing and Imitating Programmer Style: Adversaries in Program Authorship Attribution. *Proc. Priv. Enhancing Technol.* **2018**, *2018*, 127–144. [CrossRef]
37. Abuhamad, M.; AbuHmed, T.; Mohaisen, A.; Nyang, D. Large-scale and language-oblivious code authorship identification. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 101–114.
38. Caliskan-Islam, A.; Harang, R.; Liu, A.; Narayanan, A.; Voss, C.; Yamaguchi, F.; Greenstadt, R. De-anonymizing programmers via code stylometry. In Proceedings of the 24th *USENIX* Security Symposium (*USENIX* Security 15), Washington, DC, USA, 12–14 August 2015; pp. 255–270.
39. Alrabaee, S.; Saleem, N.; Preda, S.; Wang, L.; Debbabi, M. Oba2: An onion approach to binary code authorship attribution. *Digit. Investig.* **2014**, *11*, S94–S103. [CrossRef]
40. Caliskan, A.; Yamaguchi, F.; Dauber, E.; Harang, R.E.; Rieck, K.; Greenstadt, R.; Narayanan, A. When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries. In Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, CA, USA, 18–21 February 2018; pp. 1–13.
41. Frankel, S.F.; Ghosh, K. Machine Learning Approaches for Authorship Attribution using Source Code Stylometry. In Proceedings of the 2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, 15–18 December 2021; pp. 3298–3304. [CrossRef]
42. Briciu, A.; Czibula, G.; Lupea, M. A deep autoencoder-based classification model for supervised authorship attribution. *Procedia Comput. Sci.* **2021**, *192*, 119–128. [CrossRef]
43. Gu, Q.; Zhu, L.; Cai, Z. Evaluation Measures of the Classification Performance of Imbalanced Data Sets. In *Computational Intelligence and Intelligent Systems*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 461–471.

44. Brown, L.; Cat, T.; DasGupta, A. Interval Estimation for a proportion. *Stat. Sci.* **2001**, *16*, 101–133. [CrossRef]
45. Freegle1643. Python Lexical Analyzer. Available online: https://github.com/Freegle1643/Lexical-Analyzer (accessed on 18 September 2021).
46. Rehurek, R.; Sojka, P. Gensim–Python framework for vector space modelling. *NLP Centre Fac. Inform. Masaryk Univ. Brno Czech Repub.* **2011**, *3*, 2.
47. Boetticher, G.D. *Advances in Machine Learning Applications in Software Engineering*; IGI Global: Hershey, PA, USA, 2007.
48. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.S.; Dean, J. Distributed representations of words and phrases and their compositionality. *Adv. Neural Inf. Process. Syst.* **2013**, *26*, 3111–3119.
49. Lau, J.H.; Baldwin, T. An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation. In Proceedings of the 1st Workshop on Representation Learning for NLP, Berlin, Germany, 11 August 2016 ; pp. 78–86.
50. Miholca, D.L.; Czibula, G. Software Defect Prediction Using a Hybrid Model Based on Semantic Features Learned from the Source Code. In Proceedings of the Knowledge Science, Engineering and Management: 12th International Conference, KSEM 2019, Athens, Greece, 28–30 August 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 262–274. [CrossRef]
51. Miholca, D.L.; Czibula, G.; Tomescu, V. COMET: A conceptual coupling based metrics suite for software defect prediction. *Procedia Comput. Sci.* **2020**, *176*, 31–40. [CrossRef]
52. Le, Q.V.; Mikolov, T. Distributed Representations of Sentences and Documents. *Comput. Res. Repos. (CoRR)* **2014**, 1–9. [CrossRef]
53. Scikit-learn. Machine Learning in Python. Available online: http://scikit-learn.org/stable/ (accessed on 1 December 2021).
54. King, A.P.; Eckersley, R.J. Chapter 6—Inferential Statistics III: Nonparametric Hypothesis Testing. In *Statistics for Biomedical Engineers and Scientists*; Academic Press: Cambridge, MA, USA, 2019; pp. 119–145.
55. Google. Online Web Statistical Calculators. Available online: https://astatsa.com/WilcoxonTest/ (accessed on 1 February 2022).
56. Schölkopf, B.; Williamson, R.C.; Smola, A.J.; Shawe-Taylor, J.; Platt, J.C. Support vector method for novelty detection. *Adv. Neural Inf. Process. Syst.* **1999**, *12*, 582–588.
57. Tax, D.M.; Duin, R.P. Support vector data description. *Mach. Learn.* **2004**, *54*, 45–66. [CrossRef]
58. Khan, S.S.; Madden, M.G. One-class classification: Taxonomy of study and review of techniques. *Knowl. Eng. Rev.* **2014**, *29*, 345–374. [CrossRef]
59. Heflin, B.; Scheirer, W.; Boult, T.E. Detecting and classifying scars, marks, and tattoos found in the wild. In Proceedings of the 2012 IEEE Fifth International Conference on Biometrics: Theory, Applications and Systems (BTAS), Arlington, VA, USA, 23–27 September 2012; pp. 31–38.
60. Pritsos, D.A.; Stamatatos, E. Open-set classification for automated genre identification. In *European Conference on Information Retrieval*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 207–217.
61. Runeson, P.; Höst, M. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empir. Softw. Eng.* **2009**, *14*, 131–164. [CrossRef]
62. Briciu, A. AutoSoft Data. Available online: https://github.com/anamariabriciu/AutoSoft (accessed on 14 April 2022).
63. Maletic, J.; Marcus, A. Using latent semantic analysis to identify similarities in source code to support program understanding. In Proceedings of the 12th IEEE Internationals Conference on Tools with Artificial Intelligence (ICTAI 2000), Vancouver, BC, Canada, 15 November 2000; pp. 46–53. [CrossRef]